DSVM Consistency Protocols For Nested Object Transactions

by

Yahong Sui

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

Winnipeg, Manitoba, Canada, 1998

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

0-612-32257-2

Canada

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

DSVM CONSISTENCY PROTOCOLS FOR NESTED OBJECT TRANSACTIONS

BY

YAHONG SUI

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Abstract

In this dissertation, we consider a distributed shared virtual memory (DSVM) system that allows multiple concurrent nested object transactions to make transactional updates to the shared object space from different nodes across a network. To maintain correctness, the multiple copies of any object which are "cached" in the nodes' memories must be kept consistent. This dissertation presents a new memory consistency protocol, lazy object transactional entry consistency (LOTEC), that has lower communication requirements in an object-based software DSVM system, and can consequently achieve higher performance. LOTEC achieves this reduction in communication by deferring the transfer of an object's updated pages across the network until those pages are referenced by an acquiring transaction. Further, it is compatible with a newly developed concurrency control protocol, nested *object* two-phase locking.

We show the correctness of the developed protocols and then evaluate the performance of a simulated DSVM system using LOTEC by comparing it with two other DSVM consistency protocols (object transactional entry consistency (OTEC) and conservative object transactional entry consistency (COTEC)) which are also described in the dissertation. The simulation results indicate that LOTEC will have the best performance in the described object-based DSVM system.

Acknowledgements

I would like to thank my advisor Dr. Peter Graham who has provided me with guidance, advice, encouragement, and support throughout my Master studies. I wish to thank the members of my committee, Dr. K Barker and Dr. R.D. McLeod, for reading the thesis and offering invaluable advice to me. I would like to take this opportunity to give a special thanks to my dear friends, Tim Fletcher, his wife Kim Fletcher, and their three children, who were the first friends I made in Winnipeg. Their hospitality and kindness left me many beautiful memories of friendly Manitobans.

I would like to thank my parents, sisters, and brother for their constant love and support. Most of all, I would like to thank my husband, for without his understanding and encouragement this work would never have been finished.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Distributed Shared Virtual Memory System (DSVMS) described by Peters, *et al.* [PGB97] builds a distributed persistent object programming environment in a single, 64 bit, shared address space. It provides a uniform view of a persistent object space in memory that is visible to all processes on all interconnected nodes across time. Similar to Distributed Shared Memory (DSM) systems [LH89, LLG+92, CBZ95, KDCZ94, BZS93], the local memory at each node is treated as a cache of the global persistent object space. Since distinct and possibly distributed processes can concurrently access any shared object in the system and because objects are operated on locally, a shared object can be cached in several local memories. Thus, the system must provide an efficient mechanism for ensuring cache (i.e. processor local memory) consistency between nodes. Consistency maintenance is related to concurrency control and the nesting of invocations on objects makes concurrency control more difficult. Method invocations on objects must be serializable [BHG87] as nested object transactions and this impacts consistency maintenance.

This thesis addresses the design and simulation of novel consistency protocols

for maintaining memory consistency with closed nested object transactions in a persistent object system implemented in a page based distributed shared memory.

1.1 Challenges

Existing memory consistency models [Lam79, LLG+92, KCZ92, BZS93, CBZ95] were designed and used in DSM systems for parallel computing. The characteristics of such systems are different from those of the proposed DSVMS. Rather than focusing on the parallel execution of a single program, the DSVMS must support consistent virtual memory for multiple concurrent processes that perform transactional updates to the shared memory space from different nodes. Thus, new memory consistency protocols must be developed to make DSVMS practical.

All processing in a persistent object system is performed as method invocations on objects at their persistent locations. A method of one object may invoke a method on another object. Thus, method invocations may be nested and should be treated as *nested atomic object transactions*. To ensure that each nested object transaction accesses only consistent object states, the system must ensure the serializability of object transactions. Simple mutex locks, as are conventionally used in controlling consistency protocols in DSM systems, do not support serializability in an object system. The fundamental difference between mutex locks and the locks required to support nested object transactions is the need for a delayed lock release process. The existing two-phase locking rules for flat transactions need to be modified to support *nested object* two-phase locking.

Due to the possibility of concurrent access, multiple copies of a shared object can exist in one or more memories (i.e. caches) at the same time. The cost of transferring updated object pages to maintain memory consistency is potentially

high communication overhead in the proposed DSVMS. A highly efficient mechanism for maintaining memory consistency across the network is critical to keep this overhead to a minimum. Any such mechanism must also be compatible with nested object two-phase locking.

1.2 Motivation

Memory consistency is key to the success of building a DSVMS. An efficient memory consistency protocol enables the use of a DSVMS to store persistent objects which are operated on using nested object transactions. An execution environment offering transactional guarantees greatly simplifies programming in a distributed system. Efficient memory consistency allows shared objects to be cached, thereby significantly reducing the effective latency of remote memory access and yielding higher overall system performance.

Processing in a large scale DSVMS is extremely network intensive. The high cost of network communications can hurt performance in two ways. First, large amounts of communications can lead to bottlenecks on conventional networks. (The best results in DSM systems for parallel processing have been obtained using high bandwidth, low latency, switched Asynchronous Transfer Mode (ATM) networks [MS95] which partially address this problem.) Second, the software overhead incurred during message sending or receiving may introduce high end-to-end message latencies. (Each message sent or received has to pass through the operating system kernel and between multiple levels of network communication protocols from/to the user application to/from the network interface.)

The communication overhead associated with memory consistency maintenance in the proposed DSVMS is critical because the implementation of method invocation

is by shipping object pages to the invoking node. Reducing both the number of messages exchanged by memory consistency maintenance and the size of those messages is very important.

1.3 Organization

The rest of this thesis is organized as follow: Chapter 2 discusses background material and related work. Chapter 3 presents the assumed environment and details the problem of maintaining memory consistency in a page based, distributed persistent object system. Chapter 4 describes new memory consistency protocols for closed nested object transactions in the proposed DSVMS and also shows their correctness. Chapter 5 presents and discusses some results obtained from a network load analysis performed using randomly generated transaction structures and object reference patterns. Finally, Chapter 6 provides conclusions and discusses directions for future work.

Chapter 2

Background and Related Work

As the working environment of this thesis, Distributed Shared Virtual Memory (DSVM) systems will be presented in this chapter. Further, since a DSVM system is a base for the development of Persistent Object Systems (POSs), the discussion will begin with an introduction to POSs. Then, transactions and serializability will be discussed as well as memory consistency models in DSM systems. The reason for surveying memory consistency models in DSM systems is that memory consistency maintenance in a DSVM system is based on that used in DSM systems which is where memory consistency protocols were originally developed.

2.1 Persistent Object Systems

An object, as discussed in [Kim90], represents a real-word entity. It is an abstraction defined by a system wide unique object identifier (OID), a set of attributes which define the state of the object, and a set of methods which are the only means of manipulating the attributes and thereby modifying the object's state.

While an object may be used simply as an abstract data type, many additional benefits are offered, such as support for complex structure, encapsulation, inheritance, and improved software re-use. These features are desirable in simplifying the process of developing applications using object-oriented techniques.

Persistence [ABC+83] offers the potential to greatly simplify application programming. The idea behind persistence is that all data in a system should be able to persist for as long as that data is required. Furthermore, with *orthogonal persistence*, *all* data may be persistent and that data may be manipulated in a uniform manner regardless of the length of time it persists.

In conventional operating systems, long lived data is treated in a fundamentally different manner from short lived data. Traditionally, long term data is held on a backing store (e.g. a database or a file system) and cannot be directly addressed; short term data, on the other hand, is managed by a programming language which accesses it in directly addressable physical or virtual memory. To map between the two types of data, two different mechanisms must be used. A file system, or database management system, must provide storage capability and explicit format translation code must be written and included in each program to convert from in-memory to on-disk format.

Persistent systems simplify programming because they hide the traditional distinction between short term and long term storage from the application programmer. They allow all data to persist for an arbitrary length of time, possibly longer than the life time of the creating process, and they support manipulation of data in a uniform manner, regardless of how long it persists. Such systems usually store all data in a persistent store which automatically manages the transfer between long and short term storage in a manner that is transparent to the application programmer. This makes program development significantly easier.

Adding persistence and transparent distribution to object systems, to create so called *distributed persistent object systems* [VRH93, VBD+92, JR92], provides significant benefits including a flexible and powerful programming environment, enhanced sharing, and improved performance for many applications. However, it also presents implementation difficulties due to the limitations of both the software and hardware supporting environments. Peters, *et al.* [PGB97] describe a distributed shared virtual memory system and its use as a base for the development of POSs. This system provides an excellent implementation platform for distributed POSs due to its inherent transparency.

2.2 Distributed Shared Virtual Memory Systems

A *Distributed Shared Virtual Memory* system [GB93, GBBZ93, BPG95, MGB96, PGB97] provides a transparently distributed persistent object programming environment in a single, 64 bit, shared address space. It gives a uniform view of the persistent object space in memory across multiple interconnected machines. In such a system, objects can be shared and transparently accessed in memory by participating processes from different nodes concurrently. This makes it possible for a system user to work in the simplest possible programming environment since the DSVM system hides the unnecessary distinctions between local and remote objects, and between primary and secondary memory.

The introduction of *virtual memory* made it easier for programmers to deal with the limited amount of physical memory available. Most conventional virtual memory systems, however, support only private address spaces. In such systems, each process has its own private, virtual address space. Since a process is only allowed to reference addresses within its own address space, there are effective

protection boundaries between processes. Because a process is prevented from accessing addresses within another process' address space, private address spaces make sharing data more difficult among processes even though sharing data is a fundamental requirement for modern, multi-process applications. In particular, pointer-based data structures cannot be shared among processes because pointers lose significance across address spaces. Systems have to copy shared data between private virtual memories and ensure the shared data are placed within the same range of virtual addresses as in the original private virtual memory to permit pointer passing. This is inconvenient and expensive. In object oriented systems, any given persistent object can refer to other persistent objects. Such references are expressed via unique object identifiers (OID_s). Private virtual addresses are not suitable for use as unique system wide object identifiers because theses addresses would not be valid in all private address spaces. One of the main difficulties to the implementation of persistent object-oriented systems is the need to construct and efficiently handle such unique systemwide object identifiers. A software address translation mechanism, *pointer swizzling* [KK93, Wil92, WD92], has been provided to manage persistent object references. Unfortunately, pointer swizzling can also cause significant overhead in accessing shared objects.

The recent appearance of architectures supporting linear 64-bit virtual address spaces makes it possible to create a *shared virtual address space* (SVAS) model [CLFL94] which can accommodate all processes and objects in a system. Such an address space is large enough to directly incorporate all the secondary memory of most computer systems, even distributed systems. This fundamentally changes the way that operating systems can use the address space.

In a SVAS, all processes share a single, large virtual address space. All processes can access any address within the shared address space. Data appear at the same

memory locations to *all processes* [PGB97]. Without address space separation to provide protection among processes, however, protection domains [CLFL94] or some other technique must be used to restrict a process' access to a specific set of virtual pages.

A SVAS enhances data sharing because all processors see the same address space regardless of their location and therefore pointer values are meaningful to all processes. Thus, data may be conveniently shared between processes in memory. Since each byte of data in the system has a unique address in the SVAS, issuing an appropriate address is all that is required for accessing shared data.

A SVAS offers additional benefits in implementing persistent object-oriented systems. Since the 64 bit virtual address space is linear and virtual addresses within a shared virtual memory are consistent across all processes, the persistent location of each object in the SVAS can be used as its unique, system wide object identifier. Such a large virtual address space can also contain all the active processes and the data on which the processes operate. Since the shared address space is never destroyed and is valid across all processes, the SVAS can provide persistence for an object's state [MGB95]. All data are referenced in a uniform manner by providing their persistent virtual addresses (OID_s), and these addresses can be passed freely because a pointer retains its meaning independent of its location, even across nodes or on secondary storage. Therefore, the overhead involved in swizzling object references and changing data formats is eliminated.

A DSVM system is created by providing a SVAS across a distributed system, so that the same address space is visible to all processes on *all nodes*. Once persistence is added, data appear at the same memory locations to all processes across all nodes for *all time* [PGB97].

In a DSVM system, distribution is completely transparent to user processes. If

a shared data item is not already local, when its address is referenced, the system will obtain the corresponding object from the network. Thus, a process does not need to know the location of a shared object.

A DSVM system may store any kind of data. The DSVM system described in [PGB97] and used in this thesis stores objects. It offers specific advantages over other forms of POSs. For example, object method invocations are accomplished by a simple subroutine call without swizzling. Additionally, support for such advanced features as nested object transactions may be transparently provided [PGB97].

2.3 Transactions and Serializability

A transaction [OV91] is a basic, atomic execution unit of consistent and reliable computing, composed of a sequence of indivisible operation executions. Transactions have four properties (the so-called ACID properties). They are Atomicity, Consistency, Isolation, and Durability. Atomicity refers to the fact that a transaction must be a single unit of work, namely, either all or none of its operations should be executed. Consistency ensures the correctness of a transaction. In other words, a transaction should leave the shared data in a consistent state after it commits or aborts. Isolation means that transactions cannot unintentionally affect each other. Durability declares that the effects of committed transactions must persist even in the presence of failures. These ACID properties of transactions guarantee correct concurrent execution as well as reliability.

Transactions may be divided into two broad categories: flat transactions and nested transactions.

2.3.1 Flat Transactions

Flat Transactions

Conventional transactions are *flat*. A flat transaction has a single starting point, a body which consists of read and write actions on atomic data, followed by a single commit or abort operation. The correct execution of a set of flat transactions on a consistent system will leave the system in a new consistent state. When a set of transactions execute concurrently, their operations may be interleaved. A *serial* execution represents an execution order in which there is no interleaving of the operations of different transactions. Each transaction executes from beginning to end before the next one can start. By definition, a serial execution of transactions is correct. Serial executions can, however, lead to poor performance because they do not take advantage of possible concurrency. On the other hand, transaction concurrency must be managed carefully because arbitrary concurrency in the execution of conflicting operations can lead to inconsistency.

Serializability for Flat Transactions

Conflict serializability [BHG87] is the most widely used correctness criterion for concurrent executions of flat transactions. The serializability theory represents executions of transactions as partial orders of operations. The conflicting operations of two transactions must be ordered such that transactions appear to execute serially.

The concurrent execution of a set of transactions is *serializable* if and only if it is *equivalent* to some serial execution of the transactions. From this correctness criterion, a concurrency control algorithm is regarded as correct if it ensures that

any interleaved execution of transactions is equivalent to some serial execution of the transactions.

Concurrency control algorithms are commonly classified into two categories: optimistic and pessimistic. Optimistic concurrency control tends to avoid delaying conflicting operations. Under this scenario, transactions are allowed to execute concurrently and their correct execution is verified once a transaction completes. Invalid transactions must be "rolled back" (i.e. have their effects cancelled). Pessimistic concurrency control delays potentially conflicting operations when first identified. It ensures that executions will be correct before they are allowed to occur but sometimes limits potential, valid concurrency.

Two-phase Locking

The most common pessimistic concurrency control algorithm is *two-phase locking* (2PL). It ensures serializability because of the way that locks are obtained. Two operations are said to *conflict* if they both operate on the same data item and at least one of them is a write [BHG87]. Two-phase locking associates two types of locks with data items: read locks and write locks. Multiple transactions are allowed to concurrently hold the same read lock, but a write lock may be held by only one transaction at a time. Two locks are in conflict if their corresponding operations are conflicting. A transaction is required to obtain the appropriate lock for each data item it accesses. If a write lock is held by another transaction, it must wait for that transaction to release the lock. No locks can be freed until all necessary locks have been acquired. When a transaction releases a lock it can no longer obtain any additional locks. This results in two phases, namely, a lock acquisition phase followed by a lock release phase. By obtaining and releasing locks in this manner, two-phase locking produces only conflict serializable executions. Bernstein, *et al.*

[BHG87] provide a formal proof of the correctness of two-phase locking.

Two-phase locking guarantees serializability, but deadlock and cascading aborts[1] are still possible. A *strict* two-phase locking technique forces a transaction to retain all of its locks until it completes. It guarantees a strict execution which is recoverable and avoids cascading aborts. In this thesis, strict two-phase locking will be extended to accommodate closed nested *object* transactions.

2.3.2 Nested Transactions

Nested Transactions

The concept of *nested* transactions was introduced by Eliot Moss [Mos85]. Weihl [Wei89] describes two kinds of nesting in transactions. One is the nesting of procedures which corresponds to nested transactions as proposed by Moss. The other is the nesting of layers of data abstractions. Since the invocations of an object's methods result in the nesting of procedures, only procedural nesting is discussed here.

In addition to read and write operations, a nested transaction may "contain" other transactions (sub-transactions) with their own beginning and termination points. These sub-transactions have the same properties as their parents. One transaction (parent) may have one or more sub-transactions (children) which may themselves in turn have their own sub-transactions, therefore, nesting may be to arbitrary depth. A transaction with no parent is a root transaction and it, which along with its descendants, forms a transaction tree which is also called a *transaction family*. Transaction families normally appear to be atomic with respect to other transaction families. Sub-transactions of one transaction family, however,

[1] When a transaction aborts, its effects may affect other transactions. Aborting these transactions may trigger further abortions. This is called cascading abort.

can execute simultaneously. Each parent transaction controls the execution of its sub-transactions with respect to one other so that all sub-transactions in the family are properly synchronized. In other words, parent transactions specify which sub-transactions execute concurrently.

Nested transactions offer two fundamental benefits. One is that they provide a potentially finer-level of concurrency among transactions. The other is that it is possible for sub-transactions to recover from failures independently of other sub-transactions. Nested transactions are naturally suited to object systems as method invocations introduce new nesting levels.

There are two forms of nested transactions which are determined by their termination characteristics: *open* nested transactions and *closed* nested transactions. Both closed and open nested transactions hide updates to data items, which are made by one sub-transaction, from all other transactions until the sub-transaction completes. Closed nesting permits only sub-transactions of the same parent to see changes after the sub-transaction completes. Other transactions outside of the transaction family see the updates only after the root transaction successfully commits. The fundamental principle of closed nesting is that no partial results of any transaction family can be made visible to other transaction families. Open nesting allows all other transactions (within and outside the transaction family) to see updates after a sub-transaction completes. The focus of this thesis is on closed nested transactions.

Serializability for Nested Transactions

In conventional flat transactions, conflict serializability is classically defined between transactions because there is no concept of sub-transactions. The system may exe-

cute transactions in *any* order, as long as the effect of any concurrent execution of the transactions is the same as that of *some* serial execution, even though different serial orders may produce different effects. Since the sub-transactions of a nested transaction family can execute simultaneously, the correctness of sub-transaction concurrency *within* a transaction family must also be ensured. Thus, conflict serializability should not only be defined between nested transaction families, but also between sub-transactions within one transaction family.

In a nested transaction family, sub-transactions may, of course, be nested and they will be executed in the order in which they are encountered. Since sub-transaction execution is synchronous, sub-transactions complete before subsequent sub-transactions are executed. Thus, the only correct serial execution order for a set of sub-transactions in one transaction family is depth-first. The condition for serializability with a set of sub-transactions in one transaction family is the same as with a set of flat transactions. A concurrent execution of a set of sub-transactions in one transaction family is serializable if and only if its execution is equivalent to the serial execution order of the set of sub-transactions. However, unlike with flat transactions, this serial execution order does not depend on user specification but is strictly depth-first. Therefore, a concurrent execution of a set of nested transactions (transaction families) is serializable if and only if its execution is equivalent to some serial execution of the nested transactions.

Moss' Nested Two-phase Locking rules

The work of Moss focused primarily on the closed nested transaction model. In a closed nested transaction, the results of a sub-transaction are not visible outside its parent before its parent commits. Thus, the results of any sub-transaction in a transaction family are not visible to any other transaction family until the root

transaction completes. In Moss' model, only leaf transactions can directly access data. Parent transactions perform only coordination and supervisory functions.

Traditional two-phase locking rules do not suffice for the synchronization of nested transaction families with each other. Additional rules are needed to manage concurrency between the sub-transactions in each transaction family because of nesting. Moss' work enhanced concurrency between sub-transactions (leaf transactions) from different transaction families which are encountered in a depth-first order in each nested transaction family.

Simple mutex locks are insufficient to support serializability with nested transactions. Moss uses a lock inheritance mechanism to enhance concurrency between the sub-transactions of a single nested transaction. When a sub-transaction completes successfully, it is said to have "committed". Any updates become "permanent" (visible to other transaction families) only if all sub-transactions and the root transaction also commit. However, such commitment is relative. An additional operation introduced for the purpose of nesting is the "pre-commit". When a sub-transaction pre-commits, the locks cannot be entirely released. The reason is that the sub-transaction's ancestors can still abort, undoing its changes. The solution is to pass all the sub-transaction's locks up to its parent who *retains* them. A retained lock may subsequently be acquired by a descendant of the transaction which retains it. Similarly, when a sub-transaction aborts, the locks cannot be released, because some of its locks might have been acquired from an ancestor. This means that some other sub-transactions have also accessed the data guarded by these locks before the aborted sub-transaction. The changes made by those sub-transactions should not be seen outside of the ancestor. Therefore, when a sub-transaction aborts, not all its locks are released. If any of its ancestors retain any of its locks, those ancestors continue to retain those lock.

Moss has shown how to extend the (strict) two-phase locking rules to accommodate closed nested transactions. He defines nested two-phase locking concurrency control rules for the closed nested transaction model. His rules ensure serializability among nested transactions (transaction families) if all of them are encountered in a user's preferred order and all sub-transactions of each nested transaction family are encountered in depth-first order. Lynch [Lyn83] has proved the correctness of Moss' nested two-phase locking rules, that is, by using Moss' nested two-phase locking rules, all executions of nested transactions are shown to be serializable. Moss' exclusive nested two-phase locking rules for exclusive locking are summarized as follows:

1. Sub-transaction T may hold a lock if no other transaction holds the lock or all transactions that retain the lock are ancestors of T.

2. When sub-transaction T commits, the parent of T inherits T's locks (either held or retained). After that, the parent retains the locks.

3. When a transaction aborts, it releases all locks it holds or retains. If any of its ancestors retain any of these locks they continue to do so.

2.4 Memory Consistency Models in DSM Systems

This section provides details about research specific to memory consistency models in DSM systems. It begins by presenting DSM systems and the memory consistency problem in DSM systems in Section 2.4.1. Section 2.4.2 follows with a survey of various memory consistency models and protocols used in DSM systems.

2.4.1 DSM Systems and the Memory Consistency Problem

There are two basic paradigms for parallel programming and for building parallel machines, *shared memory* and *distributed memory* (i.e. message-passing). The shared memory paradigm provides a single physical memory shared by multiple-processors via hardware. Thus, any update to shared data is visible to all processors in the system. Since the shared memory model is a natural extension of a single CPU system, it is easier to program. However, it has a serious bottleneck because the memory is accessed via a system bus which quickly becomes saturated. This limits the system size. *Cache memories* may be added as an important way to reduce the average memory access time. This is especially important as a first step fowards *scaleable* multiprocessor architectures (See Figure 2.1).

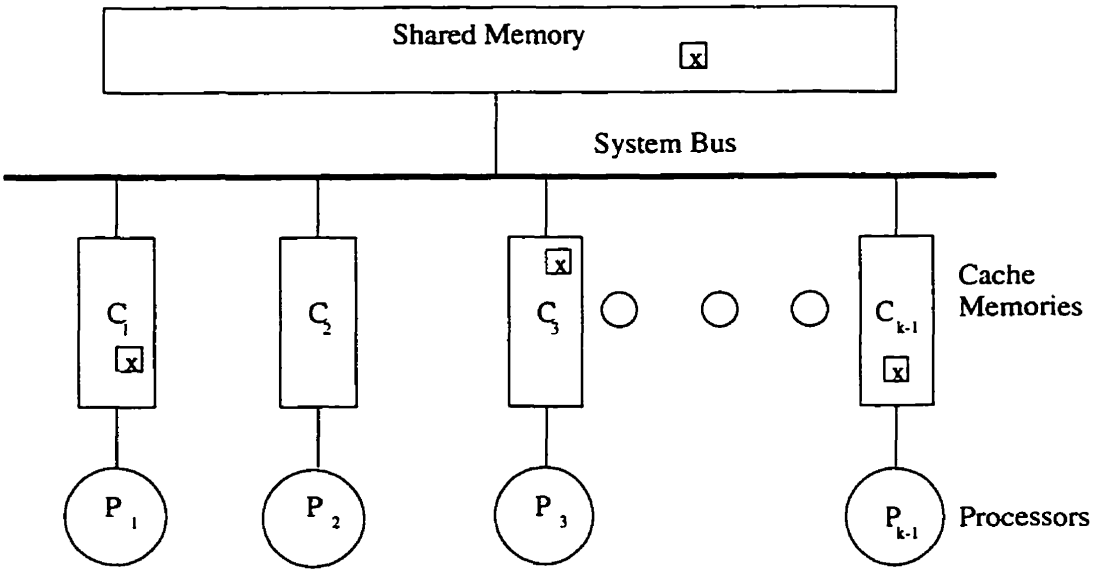


Figure 2.1: Shared Memory System with Caches

This approach gives rise to the *cache consistency* problem because data sharing

can result in several copies of a shared data item in one or multiple caches and main memory at the same time. When a process accesses an in-cache shared data item, and if the data item is not consistent with the most recent version of the data item, the process observes stale data. To maintain a consistent view of the shared memory, these in-cache copies must be kept consistent in a manner that is completely transparent to the user of the machine. Since data that has been read-cached in distributed memories can become inconsistent only when a process updates the data, there are two policies for maintaining cache consistency: write-invalidate and write-update. With an invalidation policy, once a processor acquires exclusive ownership it invalidates all copies before performing the write. With an update policy, writes to shared data are buffered, and consistency is enforced at synchronization points. Instead of invalidating all copies, the processor updates them.

Distributed memory machines consist of a collection of independent processors with their own, local memories connected by a high speed interconnection network. Communication between processors is via message passing. Distributed memory machines do not suffer from the aforementioned bottleneck or the cache consistency problem. However, they do require the programmer to partition the data between memories and manage communication explicitly.

Distributed shared memory (DSM) systems combine the advantages of shared memory and distributed memory machines. A DSM system provides an abstraction of shared memory in a physically non-shared (i.e. distributed) memory machine. Each memory is physically independent and communication takes place through explicit massage passing. Thus, DSM systems offer many benefits including ease of programming, ease of implementation, and enhanced scalability. Figure 2.2 shows the conceptual structure of a DSM system.

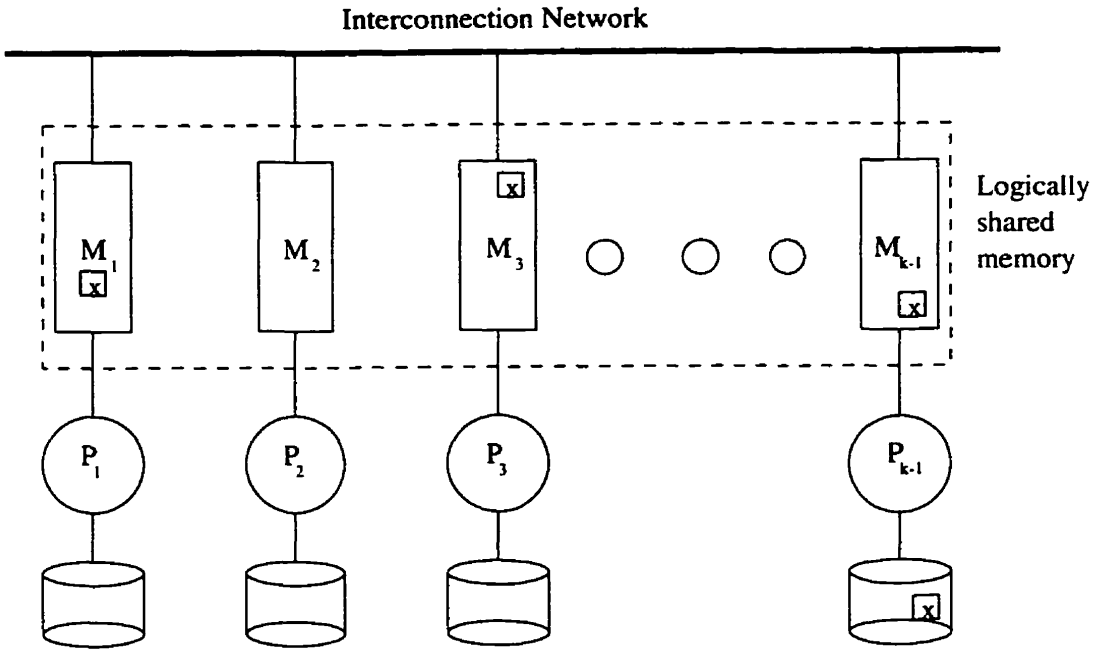


Figure 2.2: Distributed Shared Memory System

A DSM system provides the illusion that all memories are globally and transparently shared via an interconnection network. They allow each process to access any shared data item in the system directly without the programmer having to worry about where the data item is and how it can be obtained. To achieve this goal, the DSM system provides a virtual address space shared among processes across the entire network, and the local memory of each processor is effectively used as a cache of the global address space. Hence, the cache (memory) consistency problem is still a concern. Independent processes executing at different nodes must see a consistent view of the shared memory. Changes to cached data have to be detected and updates or invalidates have to be propagated to other processors caching the data. A consistency mechanism is required to ensure that processors reference only current data which are up-to-date.

2.4.2 Memory Consistency Models

A memory model for a DSM system is a specification of how the memory operations of a program will appear to execute to the programmer [Adv93]. In other words, it is a model of how updates to shared memory are reflected to the processes in the system. The memory model, therefore, specifies the values that may be returned by the read operations of a process executed on a DSM system. For a uniprocessor, a correct memory model is well defined. A memory is consistent if the value returned by a read operation is always the same as the value written by the most recent write operation to the same address. For multi-cached shared memory systems, caching of data complicates the ordering of accesses by introducing multiple copies of the same location. The cache consistency model is complicated because the definition of "most recent write operation" becomes unclear when there may be multiple processors accessing different copies of the same address. For a DSM system, however, the memory consistency problem differs from that in multi-cached shared memory systems and is even more complex because there is no physical shared memory and there may also be a far greater number of processors in the DSM.

The memory model affects the performance of a DSM system in many ways. It determines when a processor can execute multiple memory operations in parallel or out of program order, when memory operations are allowed to overlap other memory operations, when updates for a shared data item by one processor can be made visible to other processors, and how much inter-processor communication a memory operation will cause. Many verified solutions to the memory consistency problem in DSM systems have been proposed and successfully implemented. The design goal for consistency protocols is to achieve the best possible performance in a DSM system. However, high memory latency and limited bandwidth make this

difficult. Therefore, the key issue in designing memory consistency protocols is to reduce both the memory latency caused by remote memory accesses and the large amount of communication required by memory consistency maintenance.

There are two kinds of memory consistency models for DSM systems: strong consistency or sequential consistency [Lam79] and relaxed/weak consistency (e.g., release consistency [LLG+92, CBZ95], lazy release consistency [KCZ92], entry consistency [BZS93], etc.). These models have all been implemented in either hardware or software or both in the past few years.

Sequential Consistency (SC) is a natural extension of the uniprocessor model. A system is sequentially consistent if (1) all memory operations appear to execute one at a time in some total order, and (2) all memory operations of a given processor appear to execute in program order. In other words, sequential consistency requires that any update to shared data becomes visible to all other processors before the updating processor is allowed to issue another memory access. Essentially, sequential consistency ensures that the view of the memory is consistent at *all times* from all processors.

Such a strict requirement imposes serious restrictions on efficient performance. The Ivy system [LH89] supports a page-based single-writer, invalidate protocol to implement sequential consistency. The weakness of Ivy is that it can cause an excessively large amount of communication overhead because invalidate messages for updated shared data must be sent out immediately and for every write operation. For example, in Figure 2.3, data item x is updated by the same processor (P_1) repeatedly with no intervening access by other processors. If P_1 and P_2 both cache shared data item x, P_1 must send invalidate messages for data item x to P_2 immediately for every write operation before it is allowed to issue another read or write to shared data.

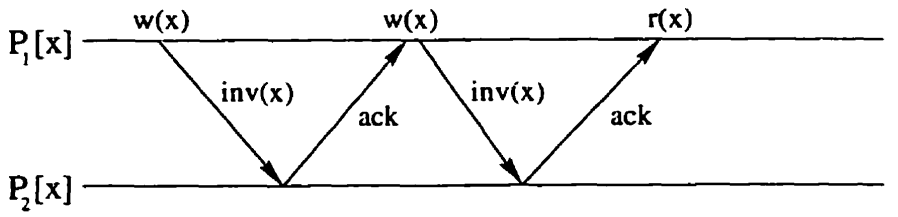


Figure 2.3: Sequential Consistency in Ivy

Sequential consistency precludes many performance enhancing optimizations in both hardware and software, such as write buffers, pipelining execution and reordering of operations. To improve the performance of shared virtual memory systems, researchers have proposed relaxed memory models that impose constraints weaker than sequential consistency. Relaxed consistency takes advantage of the fact that programmers use synchronization operations, such as lock acquisition and barrier entry, to separate accesses to shared data by different threads running on different processors. Therefore the system only needs to guarantee that the memory is consistent at selected synchronization points. Relaxed consistency models are far more efficient in implementing DSM systems.

Release Consistency (RC) [GLL+90] is a form of relaxed memory consistency. In release consistency, each shared memory access is classified as either a synchronization access or an ordinary access. Furthermore, synchronization accesses are made explicit and categorized into "acquires", which signal the beginning of a series of accesses to shared memory, and "releases", which signal the end of a series of accesses to shared memory. Release consistency exploits the fact that in a critical section a programmer has already assured that no other processor is accessing the protected data. Thus, all previous updates of shared data are consistent only before a release of a synchronization variable is observed by any processor. Release consistency allows the effects of shared memory accesses to be delayed until a subsequent

release is performed by the same processor.

Researchers in hardware DSM have proposed the release consistency model to reduce the memory latency associated with remote memory accesses, for instance. Dash [LLG+92] implements a form of release consistency using a write-invalidate policy. It reduces the memory latency by pipelining the invalidation messages caused by writes to shared data. The processor is stalled only when it executes a lock release, at which time it must wait for all its previous writes to be performed remotely(see Figure 2.4).

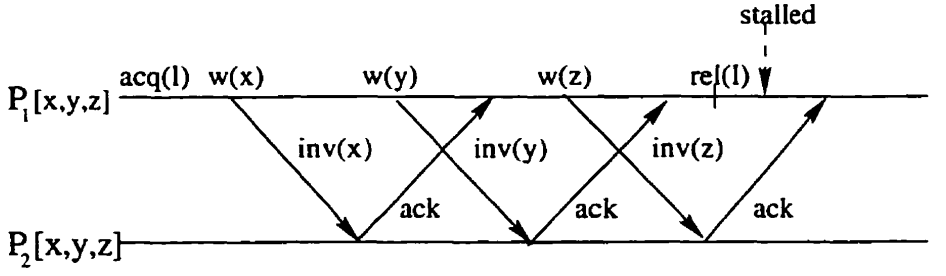


Figure 2.4: Pipelining Invalidations in Dash

In a *software* DSM system, the overhead of exchanging messages is very high. Since sending a message is more expensive than it is in a hardware DSM system, it is more important to reduce the number of messages exchanged than it is to hide their latency by pipelining. Ideally, the number of messages exchanged in a software DSM system should equal the number of messages exchanged in a message-passing implementation of the same application [KCZ92]. For this reason, the Munin system's write-shared protocol with update-with-timeout policy [CBZ95] implements RC and reduces the number of messages exchanged by buffering modifications until a lock is released. Ideally, it reduces the number of messages transferred from one per write to one per critical section when there is a single replica of the shared

data.[2] At the release point, all modifications are sent to all processors who cache the data modified by the releasing processor. All modifications going to the same destination are merged into a single message. In the example shown in Figure 2.5, pages X, Y, and Z are cached in both processor P_1 and P_2. The updates for page X, Y and Z by processor P_1 are buffered. At the time P_1 releases the lock, the updates are merged into a single message and propagated to processor P_2 which also caches those pages.

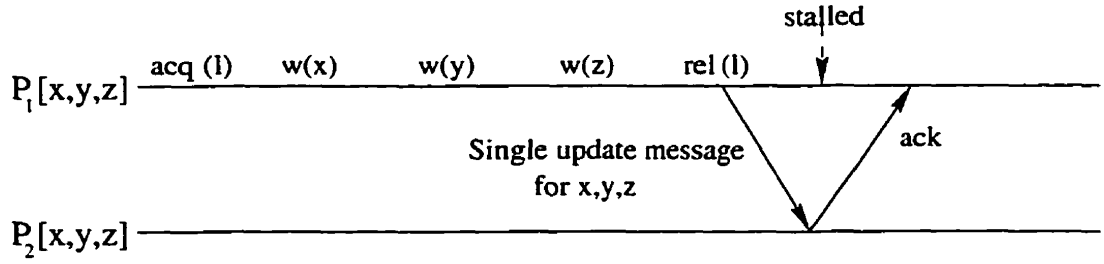


Figure 2.5: Buffering and Merging Updates in Munin

Munin's protocol, however, may still send a number of unnecessary messages, because it propagates updated data to *all* processors who cache the data when the corresponding lock is released. Keleher [PK95] has shown that there may be two forms of unnecessary communication using Munin's protocol with an update or invalidate policy. First, since some targets of the invalidations or updates never access the invalidated or updated data, they would not notice if their copies became inconsistent. Hence, invalidation or update messages which are propagated to them are useless. Second, some invalidation or update messages travel the same route as a subsequent lock transfer. Therefore, an invalidation or update message could be eliminated by piggybacking the invalidation or update message on the lock transfer. This is shown in Figure 2.6. Three processors P_1, P_2 and P_3 all cache shared pages

[2] If there are x replicas of the shared data, it reduces the number of messages transferred from x per write to x per critical section.

X and Y, and exchange the lock for the shared pages. Processor P_1 updates page X and then releases the lock. The invalidate messages for page X are sent to processor P_2 and P_3. After processor P_2 updates page Y, at the time of the release, invalidate messages for page Y are sent to P_1 and P_3. Then processor P_3 reads page Y and releases the lock. First, since processor P_2 and P_3 do not access page X and P_1 does not access page Y, all invalidation messages except the one from P_2 to P_3 for page Y are useless. Second, the invalidate message to p_2 travels the same route as the lock transfer requested by P_2. Similarly, the invalidate message to p_3 travels the same route as the lock transfer requested by P_3.

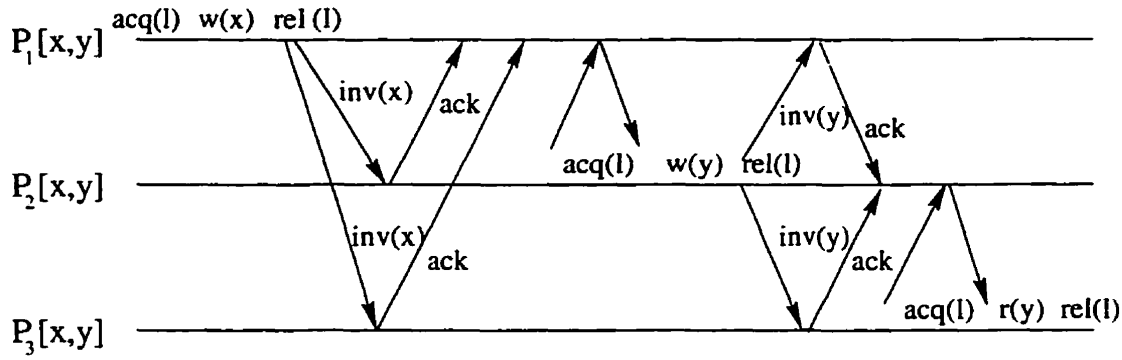


Figure 2.6: Remote Memory Accesses in Munin with the Invalidate Policy

Logically, it suffices to update or invalidate a cached shared data item for a processor only when the processor acquires access to the data. TreadMarks [KDCZ94] implements Lazy Release Consistency (LRC) [KCZ92] which does not make modifications globally visible at the time of a release. Instead, lazy release consistency guarantees only that a processor that acquires a lock will see all modifications that precede the lock acquisition. By not propagating modifications globally at the time of the release, and by piggybacking data movement on lock transfer messages, lazy release consistency reduces both the number of messages and the amount of data transferred between processors. Figure 2.7 shows the same example as Figure 2.6

under lazy release consistency with the invalidate policy.

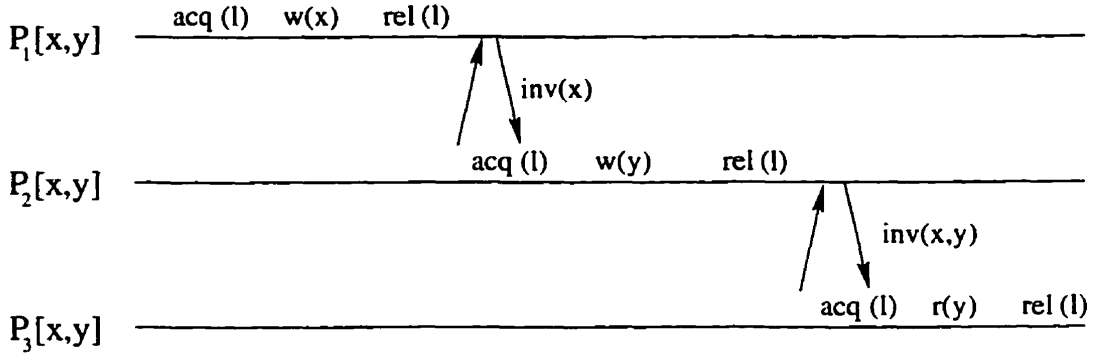


Figure 2.7: Remote Memory Accesses in TreadMarks with the Invalidate Policy

Midway's memory consistency protocol [BZS93, BNBMJZ91] uses an update policy and implements Entry Consistency (EC) which is similar to lazy release consistency. Entry consistency guarantees that shared data become consistent at a processor only when the processor acquires a synchronization object. Furthermore, entry consistency requires each shared data object to be attached to a synchronization object. The only data that is guaranteed to be consistent is that guarded by the acquired synchronization object. As a result, entry consistency generally requires less data traffic than lazy release consistency. This can be shown by comparing Figure 2.8 to Figure 2.7. Processor P_1 updates data item x on page X. When processor P_2 attempts to update a data item y on page Y, since the corresponding lock for y is free, it acquires the lock and updates y without any modification transfer. Once processor P_3 acquires the lock for data item y, the modification for y is sent to P_3.

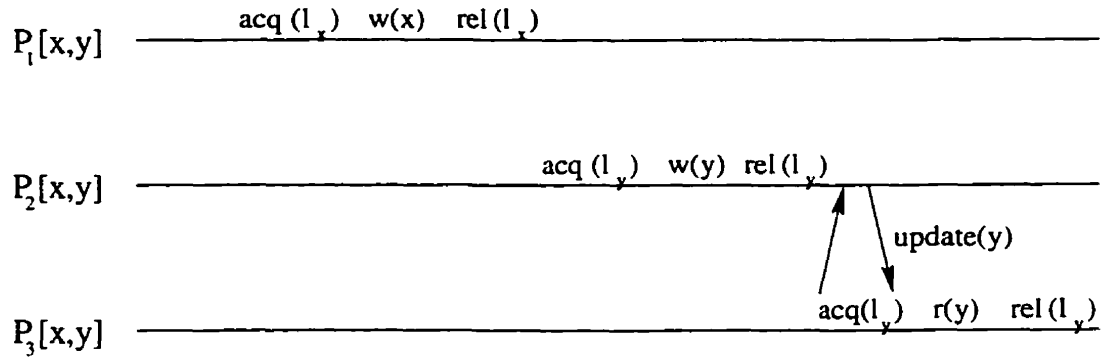


Figure 2.8: Remote Memory Accesses in Midway under the Update Policy

Chapter 3

Problem Evaluation and Environment

Before discussing memory consistency problems in DSVM systems, it is necessary to introduce the assumed environment of this thesis. This includes the definitions associated with objects, a nested object transaction model for the DSVM system, and object access in a page based DSVM system. The memory consistency problem in a persistent DSVM system and design issues for memory consistency protocols will then be discussed.

3.1 Assumed Environment

This section begins by formally defining objects following the notation of Graham [PCG94]. It is followed by defining the nested *object* transaction model. The model uses conflicts at the object level to define correct executions between [sub]-transactions. This allows serialization based on two-phase, object-level locking.

Finally, object access in a *page based* DSVM system will be presented.

3.1.1 Object Model and Properties

The proposed DSVM system contains large collections of uniquely identified persistent objects. To concentrate on memory consistency issues in a DSVM-based persistent object system and ignore the irrelevant details that would unnecessarily complicate the design, the core object concepts of [Kim90] are assumed but unnecessary details are omitted. Objects, as discussed previously, logically contain structural and behavioural components. The structural component is a set of uniquely identified data items (attributes) whose values constitute the state of the object. The behavioural component is a set of procedures (methods) which are the only means of accessing and manipulating the structural component, namely, of modifying the state of the object. Each object has its own unique object identifier (OID). In the proposed DSVM system, the OID is the object's virtual address in the persistent object space.

The following formal definition for objects is by Graham [PCG94]. The j^{th} attribute of object O_i is denoted a_{ij}, and an object's k^{th} method is identified using the notation m_{ik}.

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 system-wide unique identification to each object. Point (2) identifies the attributes of the object. Point (3) specifies the methods of the object.

Hadzilacos, *et al.* [HH91] divide the steps of a method's execution into local steps and message steps. The local steps access an object's local attributes and message steps access non-local attributes via method invocations on other objects.

All objects which share the same set of attributes and methods are grouped into a *class*. A class is associated with a single object type from which specific objects may be instantiated. An object type defines attributes which are stored in each object instantiated from it as well as a set of methods which may be applied to those attributes. An object belongs to only one class as an instance of that class.

Encapsulation, *Inheritance*, and *Polymorphism* are supported by the object model in this thesis. Encapsulation is a fundamental feature of object-orientation. It provides data abstraction and data independence. It also ensures that objects' attributes may only be accessed and updated by their local methods. This property guarantees locality of effect from which a benefit is provided to infer the correctness of concurrent method invocations. Classes are potentially related by inheritance. A class may be derived from another class (the base class). The derived class is called a sub-class of the base class and the base class is called a super-class of the derived class. Both attributes and methods may be inherited. Inheritance provides an important basis for software re-use which increases programmer productivity. Polymorphism permits objects of one class to be treated as if they were of their declared class or any super class thereof.

3.1.2 Nested Object Transactions

In a DSVM system, multiple, concurrent users access objects by invoking methods that manipulate their attributes. An access submitted by a DSVM user consists of a single object method invocation, and that method execution may invoke other methods. Thus, method invocations may be nested, and method executions either by users or other objects should be treated as nested atomic object transactions. Each method invocation begins a new [sub-]transaction. A *root transaction* is the initial method invocation made by a DSVM user and therefore it has no ancestors. The root transaction and its descendants, together, constitute a transaction tree, which we refer to as *transaction family*.

The following nomenclature is used to describe nested object transactions. The system contains a set of objects $O = \{O_1, O_2, ..., O_m\}$. An invocation of method k on object i made by a DSVM user j is denoted m^j_{ik}. It begins a nested object transaction (the root transaction) T^j_i. Recall that each execution of method m^j_{ik} has local steps which access object local attributes, and message steps which access non-local attributes via method invocations on other objects. Operations of a nested object transaction then may contain reads and writes which operate on attributes of the object the transaction is executing on. They may also contain sub-transactions which are initiated by method invocations on other objects. As mentioned previously, when the execution of a sub-transaction completes, it enters a pre-commit state indicating it is ready to commit. Thus, the operation set of T^j_i may contain reads, writes, pre-commits, and sub-transactions. The operation 'pc' denotes entry into the pre-commit state by a sub-transaction. The set of sub-transactions of a root transaction T^j_i is denoted $OT_j = \{m^j_{s_1t_1}, m^j_{s_2t_2}, ..., m^j_{s_nt_n}\}$, such that $m^j_{s_it_i} \in OT_j$ denotes a method invocation of T^j_i where the method t_i is invoked on object s_i. The set of all operations of a nested object transaction T^j_i is

denoted $OS_j = \{\cup_k O_{jk}\} \cup OT_j$, where $O_{jk} \in \{read, write, pc\}$ is an operation k of the nested object transaction T_i^j. A transaction's termination condition is denoted by $N_j \in \{Commit, Abort\}$.

One advantage of nested over flat transactions is that sub-transactions may execute concurrently in one transaction family. To ensure serializability between such sub-transactions, the internal semantics of the nested object transactions must be considered. Zapp [MEZ93] defines a boolean function *depends*. It takes two operations as inputs, at least one being a sub-transaction, and returns "true" if there is a dependence relation between the operations due to the internal semantics of the nested object transaction.

In the proposed DSVM system, operations of one nested transaction are reads, writes, pre-commits, and method invocations. The notion of conflict between reads and writes is well understood. Graham [PCG94] provides the following definition of conflicting methods.

Definition 3.2 Two methods in an object *conflict* if they contain steps which access attributes in a conflicting manner[1]. If two methods m_{ij} and m_{ik} conflict then this is denoted $m_{ij} © m_{ik}$. ■

Object-level locking is employed in this thesis. Thus, conflicts only occur between method invocations (i.e. [sub-]transactions). For exclusive object-level locking, the definition of conflicting methods is that two methods *conflict* if they are invoked on the same object. This is a very simple and conservative conflict criterion.

The nested object transaction model which is used for this thesis is now defined by referencing the definition of Graham [PCG94].

[1]Two steps conflict if they both access the same attribute and at least one is a write operation.

Definition 3.3 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. (a) for any two O_{jl}, $O_{jk} \in OS_j$, if $O_{jl} = m_{su}^j$, $O_{jk} = m_{sv}^j$, then either $O_{jl} \prec_j O_{jk}$ or $O_{jk} \prec_j O_{jl}$,

   (b) for any two O_{jl}, $O_{jk} \in OS_j$, if $O_{jl} = m_{st}^j$ and $depends(O_{jl}, O_{jk})$, or $depends(O_{jk}, O_{jl})$, then $O_{jl} \prec_j O_{jk}$, or $O_{jk} \prec_j O_{jl}$, respectively.
3. if $O_{jl} = pc$, O_{jl} is unique and $\forall O_{jk} \in OS_j$, $l \neq k$, $O_{jk} \prec_j O_{jl}$,
4. $\forall O_{jl} \in OS_j$, $O_{jl} \prec_j N_j$,
5. the termination conditions of all $m_l^j \in OS_j$ are consistent and equal to N_j.

■

Point (1) enumerates all the operations performed by the nested object transaction. Point (2) addresses the ordering relation of the nested object transaction. Point (2a) orders the conflicting method invocations of the nested object transaction. Point (2b) allows concurrent execution of sub-transactions of the nested object transaction as long as a partial order of all the operations obeys the *depends* function. Point (3) indicates that all operations of the nested object transaction must occur before its only pre-commit operation. Point (4) prevents any operation of a nested object transaction from occurring after the transaction terminates. Point (5) ensures that all of a transaction's sub-transactions either commit or abort with their parents.

For the purpose of this thesis, it is assumed that the dependence relationship of point (2b) has been defined, and dependence analysis has been performed to determine the appropriate partial order (see [PCG94] for details).

3.1.3 Object Access in a Page based DSVM System

The high level structure of the DSVM system proposed by [PGB97, MGB96] is shown in Figure 3.1. The DSVM system is assumed to consist of some number of nodes connected by a high bandwidth, low latency network. Each node is disked and contains a significant physical memory. The SVAS is globally distributed across all nodes. Therefore, the same SVAS is visible to all processes regardless of their physical execution location. The persistent object store is stored collectively on the disks provided by each processor.

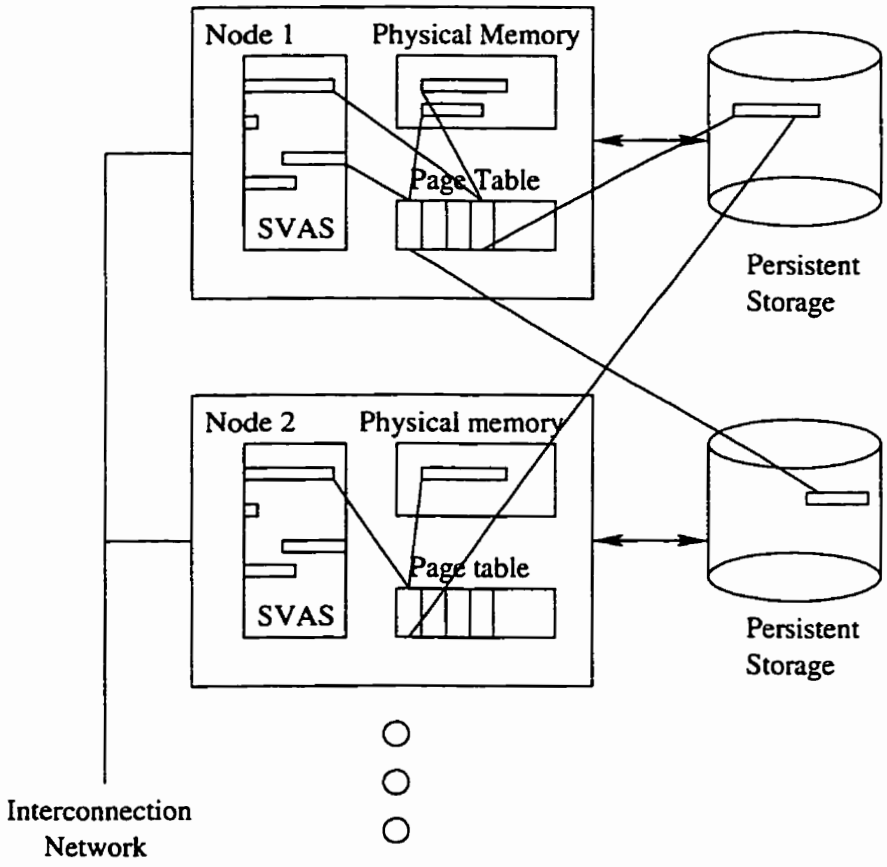


Figure 3.1: High Level Structure of the DSVM System

In a page based DSVM system, the SVAS is partitioned into virtual memory pages. The local memory of each processor is used as a cache for the SVAS. A per-processor page table is used to map virtual pages into page frames. Object management takes place as a direct consequence of virtual memory operations. During a virtual memory access, there are two types of faults that may occur and which are detected in hardware: a *segmentation fault*, which occurs if a process references a region of virtual memory for which no mapping exists in the page table; or a *page fault*, which occurs if a reference is made to a mapped region of a processor's address space but the corresponding page is not resident in real memory.

Mathew, *et al.* [MGB96] presented the design of a *Global Directory of Objects* (GDO) to manage the potentially huge number of objects in a DSVM system. It provides object lookup by virtual address for all objects and persistently stores all the management information for the objects which may include persistent storage information, concurrency control and consistency information. All method invocations on an object are assumed to execute beginning at the start address for that object (i.e. its OID) in virtual memory. Thus, when an object transaction invokes a method on an object, the virtual address corresponding to the start of the accessed object is referenced. On a segmentation fault, control is transferred to the DSVM system. The faulting virtual address is used as the key in searching the GDO. If an entry is not found in the GDO, an invalid object reference has been detected. Otherwise, a valid object was referenced and the DSVM system adds page table entries to the processor's page table for the object's pages. Initially, these entries indicate that the corresponding pages are not memory resident. The system then also retrieves an up-to-date copy of the object's first page into the memory (page in) to avoid a page fault immediately occurring. Page table entries are built using information contained in the corresponding GDO entry. In response to a page fault,

the DSVM system uses the information in the page table entry for the faulting page to retrieve a copy of the object's page and make it resident in the real memory.

To ensure memory consistency and proper concurrency control, the system must unmap an object's pages once access to it is complete. By doing so, a segmentation fault is guaranteed to occur when the object is accessed again, thereby permitting the DSVM system to effect any required object management functions. For example, it is necessary to have the DSVM system gain control so that object-level lock management may be performed. It is also important that the DSVM system gains control at access completion to permit the freeing of locks and other management functions. Unmapping an object, however, does not mean that its pages need to be flushed back to persistent storage. An object's pages can be flushed lazily using a write back queue from which available page frames are replaced as needed in a Least Recently Used (LRU) fashion. If the subsequent acquiring [sub-]transaction is from the same node that last updated the object, it may reclaim any pages that are in the write back queue. If the subsequent acquiring [sub-]transaction is from a different node, the up-to-date pages of the object must be transferred to that node.

3.2 Memory Consistency in a DSVM System

Distinct and possibly distributed processes can concurrently access any shared object in a DSVM system. This sharing can result in several copies of a shared object existing in multiple memories at the same time. This requires that the DSVM system must ensure that the in-memory copies of persistent objects at each node are always consistent.

In most existing software DSM systems for parallel computing [LH89, KDCZ94, BZS93, CBZ95], the sequentially consistent execution is chosen as the base system

view of all memory models, where a sequentially consistent execution is an execution of a program that could have been produced by a sequential consistency system. Therefore, the correctness criterion for memory models is that certain memory operations on which various optimizations can be applied do not violate sequentially consistent execution and provide significant performance benefits. In other words, memory models should guarantee that certain programs (i.e. data-race-free programs [Adv93]) execute as if they were running on a sequentially consistent memory system.

Rather than focusing on the parallel execution of a single program, a DSVM-based system must support consistent virtual memory for multiple processes that perform transactional updates to the shared memory space concurrently from different nodes. Additionally, when a persistent object system is built in DSVM, the nesting of invocations on objects makes consistency maintenance more complicated. Serializability is a suitable correctness criterion for concurrent executions of nested object transactions in the proposed DSVM based persistent object system. To ensure that in-memory copies of persistent objects at each node are always consistent, in other words, to ensure that each transaction sees only consistent persistent objects, the DSVM system must ensure that method invocations on objects are serializable as nested transactions and must use a memory consistency model that is compatible with the serialization of nested object transactions. Thus, an object transactional consistency protocol is needed for the proposed DSVM system.

3.3 Consistency Protocol Design Issues

This section presents the issues that arise in the design of memory consistency protocols for the proposed DSVM based persistent object system. It begins by

discussing serializability with closed nested *object* transactions, emphasizing the difference between Moss' [Mos85] and our transaction model. The effects of the differences in serializability with closed nested object transactions are also analyzed. Factors that affect the performance of such consistency protocols are then examined and problems arising in *page based* DSVM systems are discussed.

3.3.1 Serializability with Closed Nested Object Transactions

Simple mutual exclusion (mutex) locks, as are used in conventional consistency protocols, do not support serializability with nested object transactions. As discussed, Moss provides rules for nested two-phase locking to support serializability for closed nested transactions in a non-object system. The fundamental difference between mutex locks and the locks required to support closed nested transactions is the multi-stage release process. Locks are released first for access by sub-transactions which have the same parent transaction as the releasing sub-transaction, then, for sub-transactions which have the same ancestors in turn, and finally, for all other transactions after commitment of the root transaction. Moss' rules also restrict internal transactions to manipulate data directly. If a parent transaction needs to access data, one need only introduce a new child to perform the action on the parent's behalf. Thus, only leaf transactions can manipulate data directly in Moss' model.

In the proposed DSVM system, all data are objects. Due to encapsulation, the only way to access an object is via a method invocation on it. Object method executions have local steps which access on object's local attributes and message steps which access non-local attributes via method invocations on other objects.

Thus, both leaf and non-leaf transactions in a given transaction family are allowed to access object attributes as well as to create sub-transactions.

Two problems immediately present themselves because of encapsulation. First, unlike Moss' transaction model, parent transactions not only just retain locks from their descendent transactions, but must also hold locks for their own needs because all sub-transactions of a nested object transaction including the root transaction can access object attributes directly.

Secondly, by using Moss' two-phase locking rules, deadlock may be introduced in one transaction family if both a parent and its descendent access the same object directly or indirectly (so called recursive invocations). When a parent transaction locks an object first, none of its descendent transactions can obtain the lock to access the same object because the lock will be held until the parent transaction commits. On the other hand, the parent transaction cannot commit without its descendant transactions committing.

Therefore, Moss' closed nested two-phase locking rules need to be extended to ensure the serializability of nested *object* transactions as defined in this thesis.

3.3.2 Memory Consistency and Performance Issues

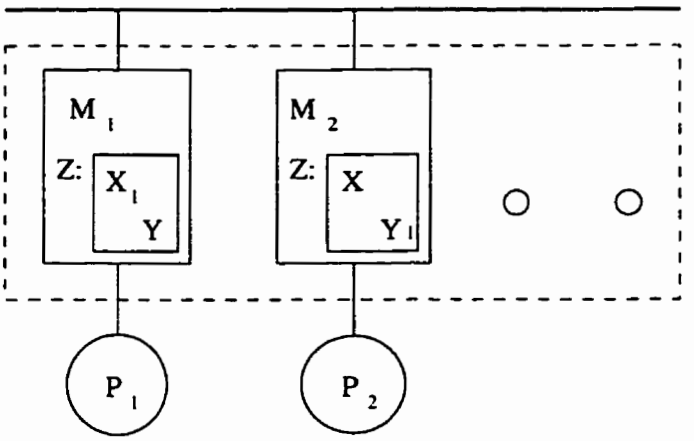
Once the serializability of closed nested object transactions is guaranteed, a consistency model which supports nesting and objects is needed which is compatible with the serialization of closed nested object transactions in a distributed persistent object environment. The performance of such an object transactional consistency model becomes the next important design criterion. There are two primary obstacles to obtaining better performance in a software implementation of a memory consistency model.

Since shared objects may be accessed concurrently by different object transactions from different nodes in the DSVM system, several copies of a shared object may exist in one or multiple memories at the same time. Thus, maintaining memory consistency may lead to significant network traffic due to the required transfer of updated object pages. The high cost of network communication can hurt performance because large amounts of communication can lead to bottlenecks. More important, in a DSVM system is the software overhead associated with a message being sent or received. Since sending a message in a software DSVM system is more expensive then in a hardware DSVM system, it is very important to reduce the number of consistency related messages (which consist of both short control messages and large object updates) which must be sent. A highly efficient mechanism for maintaining memory consistency across processors is critical to a practical DSVM implementation. To minimize the cost of consistency maintenance in the DSVM system, a relaxed memory consistency model is needed.

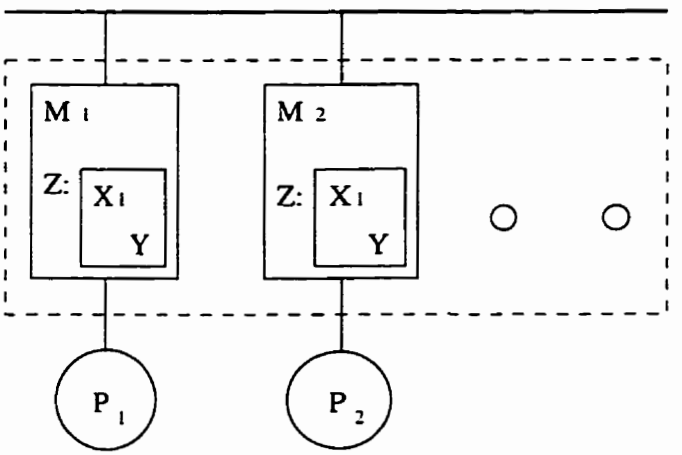
In page based systems, the shared virtual address space is divided up into pages. Transfers between memory and disk are always in units of pages. With virtual memory pages as the units of consistency, the potentially large size of pages makes the system prone to *false sharing* [KJE93]. This problem also limits the performance of page based systems. False sharing occurs when two or more separate processors concurrently update different shared data items that co-exist within a shared page which is cached in their local memories. This may lead to memory inconsistency (i.e. lost updates) because the transfer unit is a page. Consider the example shown in Figure 3.2. Processors P_1 and P_2 concurrently update different shared data items x and y respectively which are both located on the same page Z (shown in Figure 3.2 (1)). If P_2 then attempts to access data item x, with the invalidate policy the page Z which is cached in its own memory is invalidated and the page which

contains the updated x_1 is transfered from P_1 to P_2. As a result, however, the up-to-date y_1 updated by processor P_2 is lost (shown in Figure 3.2 (2)).

To prevent false sharing, conventional protocols require processors to grant exclusive access to an entire page before it can be modified. Therefore, multiple processors may contest the ownership of a page. Thus, the false shared page has to travel across the network, even though the local copy of the page would have sufficed since the access is to different data items. The page then "ping-pongs" back and forth between different processors. This results in heavy network traffic and represents unnecessary communication. Consider the following scenario. Assume that processor P_1 holds a writable copy of a given page. When processor P_2 attempts to write to the page, the system retrieves the page from P_1 and invalidates P_1's own copy. When P_1 attempts to write to the page again, the opposite sequence of events will occur. As each processor writes to the page which is held by the other processor, the page will travel across the network. Consistency protocols for a page-based system have to deal with this problem.



(1)



(2)

Figure 3.2: An Example of False Sharing

Chapter 4

Lazy Object Transactional Entry Consistency

This chapter introduces a novel consistency protocol known as Lazy Object Transactional Entry Consistency (LOTEC) for maintaining memory consistency with closed nested object transactions in a DSVM-based persistent object system. The design of the protocol is based on certain *initial* assumptions. First, consistency control will be provided on a per-object basis via object-level locking which can be achieved by including lock and cache consistency information in each entry in the GDO [MGB96]. Second, to avoid false sharing, it is assumed that attribute values of multiple objects will not be stored on a single page but attribute values of a single object will be allowed to span multiple pages[1]. Third, only exclusive locks are supported to simplify the design. Support for shared ("read"/"write") locks can be easily derived from what is presented. Fourth, all sub-transactions of

[1]Since an object's methods are never modified, we are only concerned about attribute values of an object when we discuss transferring updated object pages as required for memory consistency maintenance.

a given root transaction will normally execute at the same node. Finally, directly or indirectly recursive invocations are not allowed within one transaction family because handling deadlock is not discussed in this thesis [2].

4.1 Serializability of Closed Nested Object Transactions

This section discusses serializability of closed nested object transactions in the proposed DSVM system. It begins by giving the definition of closed nested object two-phase locking rules in Subsection 4.1.1. Algorithms for lock operations are provided in Subsection 4.1.2. Following this, an example of the execution of nested object transactions under the closed nested object two-phase locking rules is presented in Subsection 4.1.3. The correctness of the closed nested object two-phase locking rules is discussed in Subsection 4.1.4.

4.1.1 Closed Nested Object Two-phase Locking Rules

This subsection provides the modifications necessary to Moss' closed nested two-phase exclusive locking rules (N2PL) to support closed nested *object* two-phase exclusive locking rules (O2PL). Based on Moss' N2PL rules, the rules for closed nested object transactions 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

[2]Deadlock management is generally a well understood problem.

(b) if T depends on a transaction T', T' has completed (this will be clarified shortly).

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 (both holds and retains). After that, the parent retains all the locks.

4. When a transaction T aborts, it releases all locks it and its sub-transactions hold and retain unless any of its ancestors retain any of these 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.

Rule (1) enforces an order on nested object transactions. Rule (1a) prevents two nested object transactions from concurrently accessing the same object in a conflicting manner. Conflicting nested object transactions are ordered in the same order in which the relevant lock is obtained.

If conflicting transactions are from different transaction families and the lock is held by one transaction, the requesting transaction has to wait until the root transaction of the transaction holding the lock releases it. Thus, an order is forced between conflicting transactions as well as between corresponding transaction families.

If conflicting transactions are from the same transaction family, special considerations are required. Figure 4.1 illustrates a case of conflicting sub-transactions that may arise in one transaction family. In this case, object O_2 is accessed by multiple method invocations made by a root transaction T_1^i. The conflict occurs between two sub-transactions T_{10}^i and T_{11}^i created by method invocations m_{27}^i and m_{29}^i respectively. Sub-transaction T_{10}^i holds the lock for object O_2 needed by sub-transaction T_{11}^i from the same transaction family. If T_{10}^i has completed (pre-committed), T_{11}^i should be permitted to obtain the lock. If this is not permitted, a deadlock occurs (shown in Figure 4.2). In this case, T_{11}^i waits for T_{10}^i to release the lock; T_{10}^i waits for the root transaction T_1^i to commit so that the lock can be released; and the root transaction T_1^i waits for T_{11}^i to pre-commit so that it can commit and release the lock. Since a completed conflicting sub-transaction will not execute further operations, this deadlock can be avoided if a sub-transaction is permitted to acquire a lock when a conflicting sub-transaction from the same transaction family has completed. In addition, to satisfy the closed nested object transaction definition as well as to avoid cascading aborts, an acquiring sub-transaction can hold a conflicting lock only when its ancestor retains the lock. In Figure 4.1, once sub-transaction T_{10}^i pre-commits, sub-transaction T_{11}^i is allowed to acquire the lock from its parent T_1^i which retains the lock from T_{10}^i. Therefore, an order is enforced between conflicting sub-transactions in one transaction family. This conflicting order is the same order in which the relevant lock for O_2 is obtained.

Rule (1b) enforces an order between dependent object transactions in one transaction family to satisfy serializability. Figure 4.3 illustrates an example which may arise between dependent sub-transactions in one transaction family. In this example, method m_{41}^i and method m_{32}^i are invoked on different objects, so the sub-transactions T_{10}^i and T_{11}^i created by them do not conflict. However, method m_{32}^i is

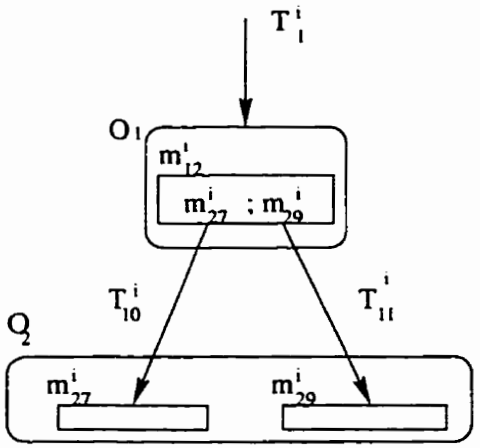


Figure 4.1: Conflicting Sub-transactions in One Transaction Family

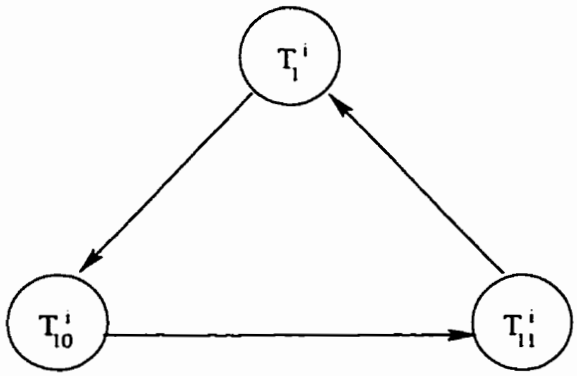


Figure 4.2: A Case of Deadlock

dependent on method m^i_{41} based on its access (within m^i_{17}) to attribute x. Therefore, sub-transaction T^i_{11} should not be executed until after sub-transaction T^i_{10} has completed. Without rule (1b), sub-transaction T^i_{10} and sub-transaction T^i_{11} could execute concurrently if they both obtained the required locks. To ensure equivalence to a serial execution, two dependent method invocations must be executed sequentially in an order consistent with the order of method invocations in their parent transaction. Therefore, rule (1b) ensures that sub-transaction T^i_{11} will be blocked until sub-transaction T^i_{10} pre-commits.

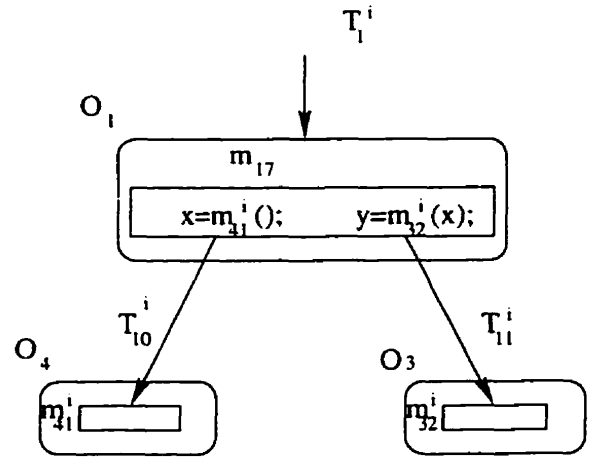


Figure 4.3: A Case of Dependent Sub-transactions

Rule (2) ensures that each nested object transaction uses strict two-phase locking in which no locks are released until the termination of the transaction. Strictness is necessary to avoid the problem of cascading aborts.

Rule (3) defines the partial release of corresponding object locks. This release permits other sub-transactions in the pre-committing sub-transaction's immediate family to see internally committed changes and to make further updates. It also precludes other sub-transactions from doing so.

Rule (4) defines the activities necessary when a nested object transaction aborts. When a transaction aborts, its locks and its sub-transactions' locks that are not retained by any of its ancestors are released so that blocked transactions can resume. For an aborting sub-transaction, locks are first released locally to sub-transactions within the transaction family. If there is no sub-transaction waiting for the locks in the transaction family at the time it aborts, the locks are then released globally to other transaction families.

Rule (5) defines the activities required when a root transaction commits. When a root transaction commits, it releases all the locks (both held by itself and retained from all its sub-transactions) to other transactions families. This makes the updates made by itself and all its sub-transactions visible to all transaction families. Serialization, therefore, is at the level of root transactions.

4.1.2 Algorithms Implementing the Closed Nested Object Two-phase Locking Rules

Before discussing the details of the algorithms for lock operations, transaction identifiers (TID_s) and a lock structure for each object must be described.

Transaction Identifiers (TID_s)

An invocation of method k on object O_i made by a DSVM user j is denoted m_{ik}^j. A root transaction created by method invocation m_{ik}^j is denoted T_i^j. A generic unique [sub-]transaction identifier has the form: $T_{i.l_1,...l_{d-2},l_{d-1}}^j$, where i is the root transaction identifier, d is the depth of the transaction nest, and $l_1...l_{d-1}$ uniquely identifies the sub-transaction by enumerating its location in the transaction family's tree.

Lock Structures

Each object has a corresponding lock structure which contains lock information for the object. Each lock structure has one lock variable to indicate the state of the lock on the corresponding object. It also has a current holder pointer to a 'holder' structure which contains the transaction identifier (TID) of the transaction

which currently holds the lock and the node identifier (NID) of the node on which the transaction executes. Since lock requests may come from different transaction families or from the same transaction family, the current_holder (see Figure 4.4) also contains a pointer to a non_local_list. All [sub-]transactions which attempt to acquire the lock from different transaction families are linked onto the non_local_list. Each element of the non_local_list uses the same data structure as the current_holder to enumerate the transactions requesting the lock. The current_holder also contains a pointer to a local_list. All sub-transactions from the same transaction family as the [sub-]transaction holding the lock attempting to acquire the lock are linked onto the local_list. Each element of the local_list also uses the same data structure as the current_holder. Therefore, a lock structure for object O_i is composed of:

lock_variable: a flag. When lock_variable is '0', the lock for O_i is free; When it is '1', the lock for O_i is held.

current_holder_pointer: points to the current_holder.

current_holder: indicates the [sub-]transaction which currently holds the lock. It has a Tid field to indicate the holding [sub-]transaction and a Nid field to indicate the node on which the [sub-]transaction executes. It also contains a pointer to a non_local_list and a pointer to a local_list.

non_local_list: is a FIFO queue for pending lock requests from [sub-]transactions in transaction families other than that of the current holder. Each element of the non_local_list utilizes the same data structure as the current_holder including pointers to next element in the non_local_list and to a local_list.

local_list: is a FIFO queue (one per transaction family) for pending lock requests from the same transaction family as the [sub-]transaction in the corresponding

element of the non_local_list. Each element of the local_list utilizes the same data structure as the current_holder including a pointer to next element of the local_list.

As mentioned earlier, the GDO [MGB96] is a global directory of objects used for managing large collections of persistent objects in a distributed environment. A GDO entry contains an object's identifier (OID) which is used as the key value for searching the GDO. Graham, *et al.* [GB93, MGB96] suggest that a lock variable for an object could be placed in each GDO directory entry. By maintaining lock variables in the globally visible object directory, the state of a lock on an object is available to all nodes. Each lock structure is thus associated with the corresponding object's GDO entry. Together with certain cache consistency information, the lock information can be used to ensure the consistency of objects when accessed by distinct and possibly distributed transactions. Figure 4.4 shows an object's GDO entry with a corresponding lock structure.

When a [sub-]transaction $T^{j}_{i,l_1,\ldots,l_{d-2},l_{d-1}}$ acquires the lock for object O_k, the required GDO entry, GDO_k, should be cached in memory to speed up subsequent object management functions. The information contained in GDO_k is used to build page table entries in the current node for object O_k. In one transaction family, multiple sub-transactions may attempt to acquire the lock for O_k. Since it is assumed that all the sub-transactions of a given root transaction will execute at the same node, lock operations for object O_k from the same transaction family will be handled locally. That is all local lock requests for a mapped object from the same transaction family may be queued on a waiting queue in the local lock structure without remote references. Thus, a local lock structure is desirable for each mapped object.

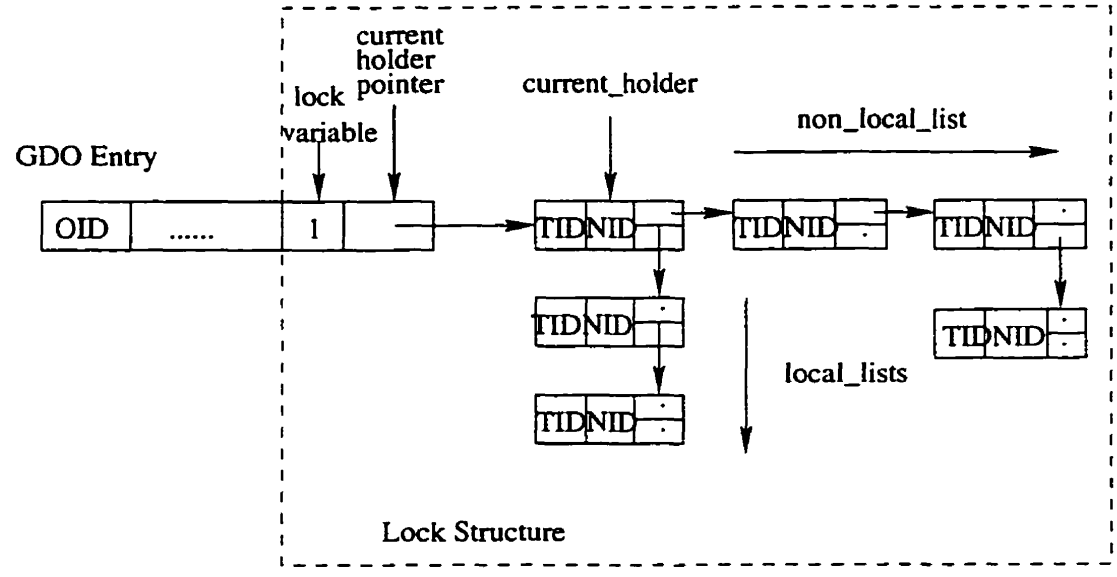


Figure 4.4: A GDO Entry with Corresponding Lock Structure

The local lock structure could, theoretically, be associated with the object's page entries in the page table. Since the page table is a hardware-read table whose structure we cannot change, we build the local lock structure in the cached GDO_k entry ($CGDO_k$). Once a [sub-]transaction $T^j_{i,l_1,\ldots,l_{d-2},l_{d-1}}$ is granted a lock L_{O_k}, it becomes the current lock holder. Since the current holder's non_local_list in GDO_k is unnecessary for handling local lock requests, the $CGDO_k$ entry does not contain the non_local_list. Thus, the lock_variable, current_holder_pointer, current_holder, and local_list compose the entire local lock structure for object O_k in the cached GDO entry $CGDO_k$. If more sub-transactions in the same transaction family attempt to acquire the lock for object O_k, they will be linked onto the local_list in $CGDO_k$.

When an object needs to be unmapped from the page table, the object's CGDO entry is unlinked from the CGDO, and then is linked onto a delayed write_back queue.

Figure 4.5 shows a cached GDO entry with its corresponding local lock structure. When the current holder $T^j_{i,l_1,\ldots,l_{d-2},l_{d-1}}$ commits, the current holder is changed to its parent $T^j_{i,l_1,\ldots,l_{d-2}}$ and the lock variable is changed to '0' (retain) in the CGDO entry. If the next sub-transaction in the local_list is a descendant of the current retainer, it acquires the lock becoming the current holder and the lock variable is set to '1' (hold). Otherwise, transaction $T^j_{i,l_1,\ldots,l_{d-2}}$ retains the lock until it commits or aborts, or a sub-transaction acquires it.

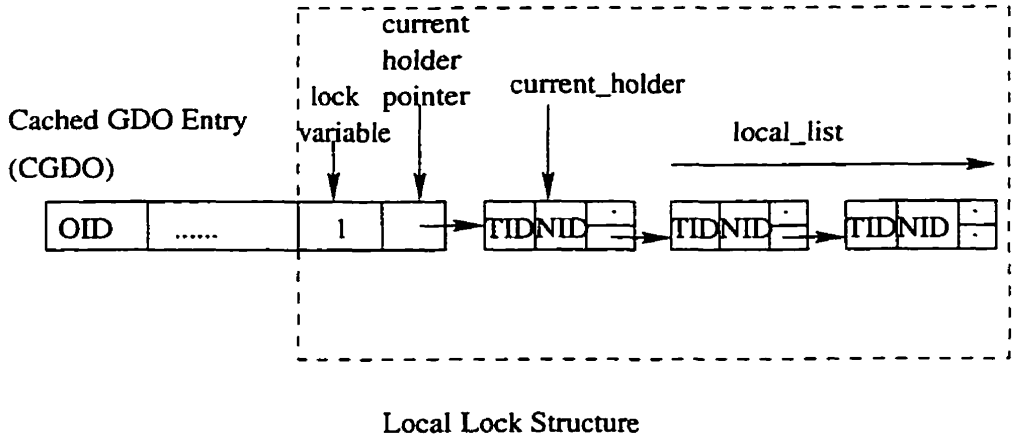


Figure 4.5: A Cached GDO Entry with Corresponding Local Lock Structure

Algorithms

The closed nested object two-phase locking rules are implemented by the transaction manager operating on certain lock structures. Each object O_i has a corresponding lock structure (LS_{o_i}) which handles all lock operations for object O_i. When the transaction manager receives a method invocation on object O_i, it "translates" the method invocation into an object transaction. The transaction manager then sends the lock operation to the appropriate lock structure, LS_{O_i}. When the lock structure, LS_{O_i}, acknowledges that the lock is set, the transaction manager

allows the object transaction to execute. Otherwise, the object transaction will be blocked until the lock is available. Thus, the transaction manager combined with the lock structures ensure that a lock is acquired before the corresponding operation is performed.

Algorithm 4.1 describes the local lock acquisition process. When an object [sub-]transaction $T^j_{i,l_1,\ldots,l_{d-2},l_{d-1}}$ attempts to access object O_k, the transaction manager invokes the Local_Lock_Acquisition routine to acquire the lock for O_k. If the object O_k is unmapped, the request is forwarded to a Global_Lock_Acquisition routine (which uses the virtual address of object O_k as the key in searching the GDO) to manage lock acquisition operation. The requesting transaction is blocked until the lock is granted from the Global_Lock_Acquisition routine. When an object O_k is mapped, if the requesting transaction and the current holder (or retainer) do not belong to the same transaction family the request is also forwarded to the Global_Lock_Acquisition routine. Otherwise, if the lock is retained by an ancestor of the requesting transaction, the request is immediately granted. If the lock is held or retained and the retainer is not an ancestor of the requesting transaction $T^j_{i,l_1,\ldots,l_{d-2},l_{d-1}}$, then $T^j_{i,l_1,\ldots,l_{d-2},l_{d-1}}$ along with the node identifier (NID_s) corresponding to the node on which the requesting transaction executes are linked onto the local_list in $CGDO_k$. The requesting transaction is blocked until the lock is granted following a Local_Lock_Release routine (which releases all locks held and retained by one sub-transaction to its parent).

Algorithm 4.1 *Local_Lock_Acquisition*

INPUT : $T^j_{i,l_1,\ldots,l_{d-2},l_{d-1}}$; /*TID of the requesting transaction*/
INPUT : O_k; /*Object being accessed */
INPUT : $CGDO_k$; /* cached GDO_k entry for object O_k*/
INPUT : NID_s; /* the node-id on which the requesting transaction executes */

IF (O_k is unmapped) **THEN**

Forward request to Global_Lock_Acquisition;
ELSE /* O_k is mapped*/
IF ($T^j_{i,l_1,....,l_{d-2},l_{d-1}}$ belongs to the same transaction family as the current holder or retainer) **THEN**
IF (the lock is retained by an ancestor of $T^j_{i,l_1,....,l_{d-2},l_{d-1}}$) **THEN**
$CGDO_k$.lock_variable ← '1'; /* $T^j_{i,l_1,....,l_{d-2},l_{d-1}}$gets the lock */
current_holder.Tid ← $T^j_{i,l_1,....,l_{d-2},l_{d-1}}$;
current_holder.Nid ← NID_s
$CGDO_k$.current_holder_pointer ← current_holder;
Send the lock grant to the requester;
ELSE
/*lock is held or retained but retainer is not an ancestor of $T^j_{i,l_1,....,l_{d-2},l_{d-1}}$*/
Link $T^j_{i,l_1,....,l_{d-2},l_{d-1}}$ along with NID_s onto the lock_list of $CGDO_k$;
ELSE
/* $T^j_{i,l_1,....,l_{d-2},l_{d-1}}$ is from a different family than the current holder or retainer*/
Forward request to Global_Lock_Acquisition;

End of Algorithm

Algorithm 4.2 describes the global lock acquisition process. This process uses the object identifier O_k as the key to search the GDO to find the GDO_k entry. If the lock for object O_k is free, the request is immediately granted. When the lock is not free, if there is a [sub-]transaction in the non_local_list which belongs to the same transaction family as the requesting transaction $T^j_{i,l_1,....,l_{d-2},l_{d-1}}$, then $T^j_{i,l_1,....,l_{d-2},l_{d-1}}$ along with the node identifier (NID_s) corresponding to the node on which the requesting transaction executes are linked onto the local_list of that transaction. $T^j_{i,l_1,....,l_{d-2},l_{d-1}}$ is blocked until the lock is granted by the Local_Lock_Release routine. Otherwise, the requesting transaction's identifier $T^j_{i,l_1,....,l_{d-2},l_{d-1}}$ along with the node identifier NID_s are linked onto GDO_k's non_local_list. Then the requesting transaction is blocked until the lock is granted by a Global_Lock_Release routine (which releases all locks held or retained by one transaction family at its completion to other transaction families).

Algorithm 4.2 *Global_Lock_Acquisition*

INPUT : $T^{j}_{i,l_1,\ldots,l_{d-2},l_{d-1}}$; /*TID of the requiring transaction*/
INPUT : O_k; /*Object being accessed */
INPUT : GDO_k; /* GDO_k entry for object O_k*/
INPUT : NID_s; /* the node-id on which node the requesting transaction executes */

GDO_k = GDO_Lookup(O_k);
IF (GDO_k.lock_variable = 0) **THEN** /*the lock is free*/
    GDO_k.lock_variable ← '1'; /* $T^{j}_{i,l_1,\ldots,l_{d-2},l_{d-1}}$ gets the lock*/
    current_holder.Tid ← $T^{j}_{i,l_1,\ldots,l_{d-2},l_{d-1}}$;
    current_holder.Nid ← NID_s;
    GDO_k.current_holder ← current_holder;
    Send the lock grant to the requester with a copy of GDO_k to build $CGDO_k$;
ELSE /*lock is not free*/
    IF (there is a transaction in the non_local_list which belongs to the same family as $T^{j}_{i,l_1,\ldots,l_{d-2},l_{d-1}}$) **THEN**
        Link $T^{j}_{i,l_1,\ldots,l_{d-2},l_{d-1}}$ and NID_s onto the local_list of that family;
    ELSE
        Link $T^{j}_{i,l_1,\ldots,l_{d-2},l_{d-1}}$ and NID_s onto the non_local_list;

End of Algorithm

Algorithm 4.3 describes the local lock release process. When a sub-transaction pre-commits, locks retained and held by the sub-transaction are passed up to its parent who retains the locks. If the next requesting sub-transaction in the local_list of the CGDO entry is a descendant of the current retainer, it acquires the lock. When a root transaction commits or aborts, the release request is forwarded to the Global_Lock_Release routine. When a sub-transaction aborts, the locks which are not retained by any of its ancestors are first released locally to sub-transactions within the transaction family. If there is no sub-transaction in the family waiting for the lock at the time it aborts, the release request is forwarded to the Global_Lock_Release routine. If the aborting sub-transaction's ancestors retain the locks, they continue to do so until they [pre-]commit, abort, or the locks are acquired by their descendants.

Algorithm 4.3 *Local_Lock_Release*

INPUT : $T^{j}_{i,l_1,....,l_{d-2},l_{d-1}}$;/*TID of the releasing or aborting transaction*/
INPUT : OID_LIST;
/*all objects retained and held by the releasing or aborting transaction*/
INPUT : Release-mode; /* $\in$ {PC, RCA, SA} */
INPUT : CGDO; /* cached GDO entries for objects in OID_LIST*/

CASE (Release-mode = PC): /*a sub-transaction pre-commits*/
  FOREACH (O_k in the OID_LIST) **DO**
    current_holder.Tid $\leftarrow$ $T^{j}_{i,l_1,....,l_{d-2}}$;/* releases the lock to the parent*/
    $CGDO_k$.lock_variable $\leftarrow$ '0'; /*parent retains the lock*/
    IF ($T^{j}_{i,l_1,....,l_{d-2}}$ is an ancestor of the next transaction in the local_list)
    THEN
      $CGDO_k$.current_holder_pointer $\leftarrow$ next TID in the local_list;
      $CGDO_k$.lock_variable $\leftarrow$ '1'; /*holds the lock*/
      Send the lock grant to the requester;
CASE(Release-mode = RCA): /*a root transaction commits or aborts*/
  Forward release request to Global_Lock_Release;
  /*Releases locks to other transaction families*/
CASE(Release-mode = SA): /*a sub-transaction aborts*/
  FOREACH (O_k in the OID_LIST) **DO**
    IF (O_k is retained by an ancestor of $T^{j}_{i,l_1,....,l_{d-2},l_{d-1}}$) **THEN**
      current.holder.Tid $\leftarrow$ TID of the ancestor;
      $CGDO_k$.lock_variable $\leftarrow$ '0';/*the ancestor retains the lock*/
      Unlink O_k from OID_LIST;
      IF (the next transaction in the local_list is the ancestor's descendant)
      THEN
        $CGDO_k$.current_holder_pointer $\leftarrow$ next TID in the local_list;
        $CGDO_k$.lock_variable $\leftarrow$ '1'; /*holds the lock*/
        Send the lock grant to the requester;
    ELSE
      IF ($CGDO_k$.current_holder_pointer != NULL) **THEN**
        $CGDO_k$.current_holder_pointer $\leftarrow$ next TID in the local_list;
        Unlink O_k from OID_LIST; /* releases the lock locally first */
  IF (OID_LIST != NULL) **THEN**
    Forward release request to Global_Lock_Release;
    /* releases locks globally */

End of Algorithm

Algorithm 4.4 describes the global lock release process. Since an original lock

holder in a GDO entry may not be a root transaction, the lock is released to other transaction families by its root transaction at the time the root transaction commits because it passed the lock up to its parent when it pre-committed. This implies that when a root transaction commits and releases a lock to other transaction families, the current lock holder transaction in the corresponding GDO entry may actually be recorded as a descendant of the releasing root transaction. When a root transaction commits, if there is no other transaction waiting for the lock in the non_local_list, the lock is set to free. Otherwise, the current lock holder will be changed from the releasing transaction or a descendant of the releasing transaction to the next requesting [sub-]transaction in the non_local_list.

Algorithm 4.4 *Global_Lock_Release*

INPUT : $T^{j}_{i,l_1,\ldots,l_{d-2},l_{d-1}}$; /*TID of the releasing or aborting transaction*/
INPUT : OID_LIST;
/*all objects retained and held by the releasing or aborting transaction*/
INPUT : GDO; /* GDO entries for objects in OID_LIST*/

FOREACH (O_k in the OID_LIST) **DO**
IF (($T^{j}_{i,l_1,\ldots,l_{d-2},l_{d-1}}$ = current_holder.Tid) or ($T^{j}_{i,l_1,\ldots,l_{d-2},l_{d-1}}$ is an ancestor of current_holder.Tid)) **THEN**
IF (no other transaction waits for the lock) **THEN**
GDO_k.lock_variable ← '0'; /*lock is free*/
GDO_k.current_holder_pointer ← 'NULL';
ELSE
/*the next requesting transaction in the non_local_list gets the lock*/
GDO_k.current_holder_pointer ← the next TID in the non_local_list;
Send the lock grant to the requester with a copy of GDO_k to build $CGDO_k$;

End of Algorithm

4.1.3 An Example

To understand lock management under the closed nested object two-phase locking rules, and especially to understand exactly what happens when the locks are released, an example of the execution of the nested object transactions shown in Figure 4.6 is presented. There are three transaction families $T_0^{a_0}$, $T_5^{a_1}$, and $T_9^{b_0}$ in this example. Transactions $T_0^{a_0}$ and $T_5^{a_1}$ execute on node A, and transaction $T_9^{b_0}$ executes on node B. Assume that both nodes A and B initially have a copy of O_4 but they were unmapped after their last accesses. Conflicting sub-transactions arrive at the shared object O_4 in the order: $T_{001}^{a_0} \rightarrow T_{500}^{a_1} \rightarrow T_{010}^{a_0} \rightarrow T_{510}^{a_1} \rightarrow T_{90}^{b_0}$.

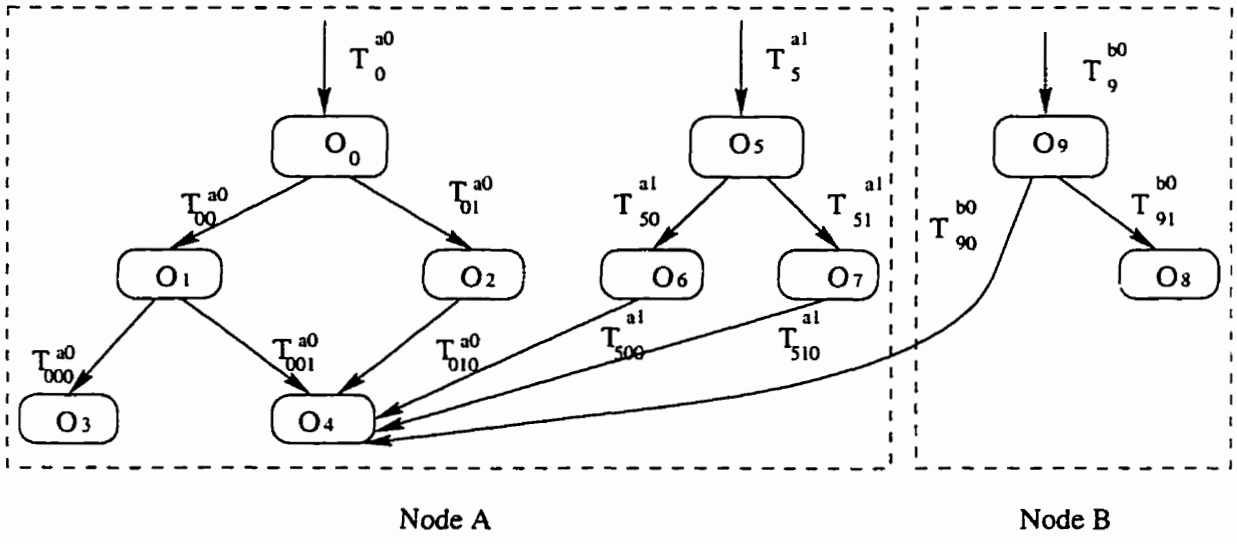


Figure 4.6: A Example of the Execution of Nested Object Transactions

On node A, the transaction manager invokes the Local_Lock_Acquisition routine to acquire lock L_{O_4} for sub-transaction $T_{001}^{a_0}$. Since O_4 is unmapped on node A, the request is forwarded to the Global_Lock_Acquisition routine. If the lock is free, $T_{001}^{a_0}$ becomes the current holder in GDO_4. GDO_4 is then cached in memory on node A as $CGDO_4$. When the transaction manager invokes the Local_Lock_Acquisition

routine to acquire lock L_{O_4} for sub-transaction $T^{a_1}_{500}$ on node A, although object O_4 is mapped on node A, because $T^{a_1}_{500}$ and the current holder $T^{a_0}_{001}$ do not belong to the same transaction family the request is forwarded to the Global_Lock_Acquisition routine. The Global_Lock_Acquisition routine then links $T^{a_1}_{500}$ onto the non_local_list in GDO_4 because the lock is not free. When the transaction manager invokes the Local_Lock_Acquisition routine to acquire lock L_{O_4} for sub-transaction $T^{a_0}_{010}$ on node A, since $T^{a_0}_{010}$ belongs to the same transaction family as the current holder $T^{a_0}_{001}$, it is linked onto the local_list in $CGDO_4$. When the transaction manager invokes the Local_Lock_Acquisition routine to acquire lock L_{O_4} for $T^{a_1}_{510}$, since it and the current holder $T^{a_0}_{001}$ do not belong to the same transaction family, the request is forwarded to the Global_Lock_Acquisition routine. There, it is linked onto the local_list of $T^{a_1}_{500}$ because sub-transaction $T^{a_1}_{510}$ is from the same transaction family as the waiting transaction $T^{a_1}_{500}$ which is already in the non_local_list of GDO_4. When the transaction manager invokes the Local_Lock_Acquisition routine to acquire lock L_{O_4} for sub-transaction $T^{b_0}_{90}$ on node B, since O_4 is not mapped on node B, the request is also forwarded to the Global_Lock_Acquisition routine. The Global_Lock_Acquisition routine then links $T^{b_0}_{90}$ onto the non_local_list in GDO_4. Figure 4.7 shows GDO_4 for the example following all global lock acquisition requests.

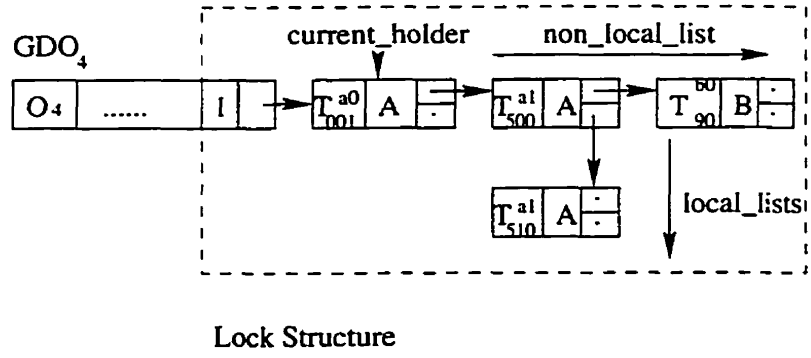


Figure 4.7: O_i's Lock Structure after Global Lock Acquisition Requests

On node A, when $T^{a_0}_{001}$ pre-commits, the Local_Lock_Release routine is invoked to release lock L_{O_4} to its parent $T^{a_0}_{00}$ which retains the lock. When $T^{a_0}_{00}$ pre-commits, the Local_Lock_Release routine is invoked to release the lock to its parent $T^{a_0}_{0}$ and the next waiting sub-transaction $T^{a_0}_{010}$ acquires lock L_{O_4} (because $T^{a_0}_{0}$ is an ancestor of the waiting sub-transaction $T^{a_0}_{010}$). When $T^{a_0}_{010}$ pre-commits, the Local_Lock_Release routine is invoked to release L_{O_4} to its parent $T^{a_0}_{01}$. Then when $T^{a_0}_{01}$ pre-commits, its root transaction $T^{a_0}_{0}$ retains the lock. Once the root transaction $T^{a_0}_{0}$ commits, the Global_Lock_Release routine is invoked to release all the locks it holds and retains to other transaction families. At this point, lock L_{O_4} is released to other conflicting [sub-]transactions waiting in the non_local_list of GDO_4. Figure 4.8 shows each state of the lock structure in $CGDO_4$ handling the lock operations, just described, for object O_4 from transaction family $T^{a_0}_{0}$ on node A.

When the Global_Lock_Release routine is invoked to release lock L_{O_4} from the root transaction $T^{a_0}_{0}$, the lock appears to be released from its descendant transaction $T^{a_0}_{001}$ in GDO_4 because $T^{a_0}_{001}$ was the original lock holder. The next waiting transaction $T^{a_1}_{500}$ in the non_local_list of GDO_4 acquires the lock. GDO_4 is cached without its non_local_list in the memory of node A as $CGDO_4$ once again. When $T^{a_1}_{500}$ pre-commits, the Local_Lock_Release routine is invoked to release the lock to its parent $T^{a_1}_{50}$ which retains the lock. When $T^{a_1}_{50}$ pre-commits, the Local_Lock_Release routine is invoked to release the lock to the root transaction $T^{a_1}_{5}$ of $T^{a_1}_{50}$ and the next waiting transaction $T^{a_1}_{510}$ acquires the lock (because $T^{a_1}_{5}$ is an ancestor of $T^{a_1}_{510}$). Similarly, the lock is passed up to $T^{a_1}_{51}$ from $T^{a_1}_{510}$, then up to $T^{a_1}_{5}$ from $T^{a_1}_{51}$ after $T^{a_1}_{510}$ and $T^{a_1}_{51}$ complete in sequence. Once the root transaction $T^{a_1}_{5}$ commits, the Global_Lock_Release routine is invoked to release all the locks $T^{a_1}_{5}$ holds and retains to other transaction families. At this point, lock L_{O_4} is released to other [sub-]transactions waiting in the non_local_list in GDO_4. Figure 4.9 shows the state

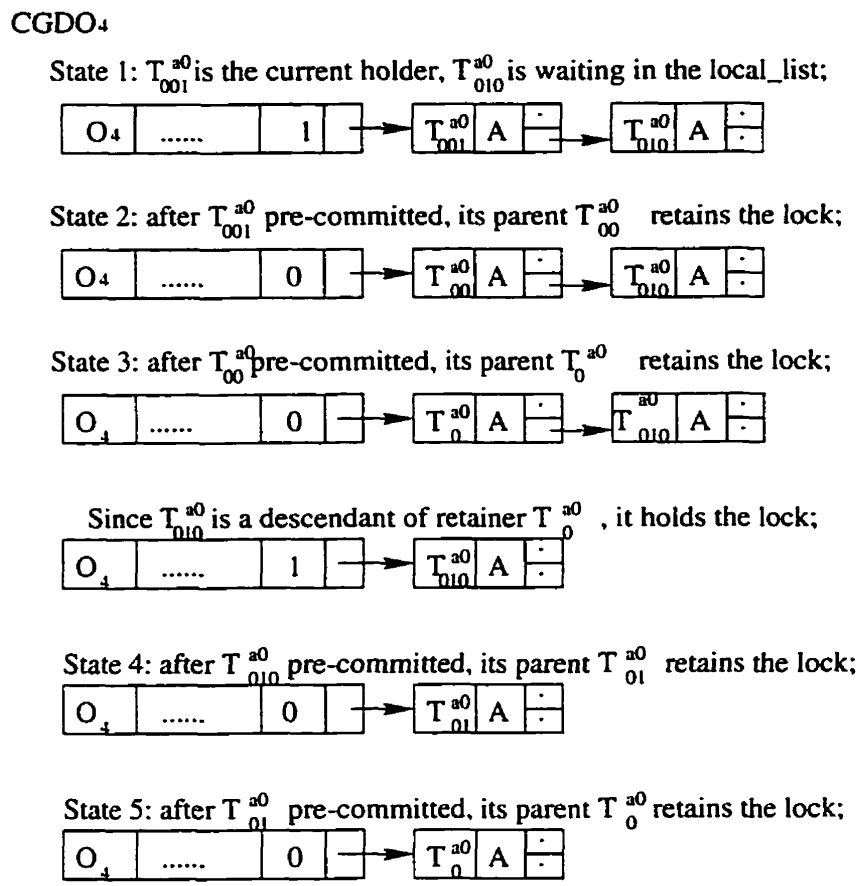


Figure 4.8: Lock Operations for O_4 for Family T_0^{a0} on Node A

of the lock structure for object O_4 from the transaction family T_5^{a1} on node A as the lock operations are handled.

When the Global_Lock_Release routine is invoked to release lock L_{O_4} from root transaction T_5^{a1}, the next waiting transaction T_{90}^{b0} in the non_local_list of GDO_4 acquires the lock. GDO_4 is cached in node B as $CGDO_4$. Figure 4.10 shows each state of the lock structure in GDO_4 while handling the lock operations for object O_4 from different transaction families.

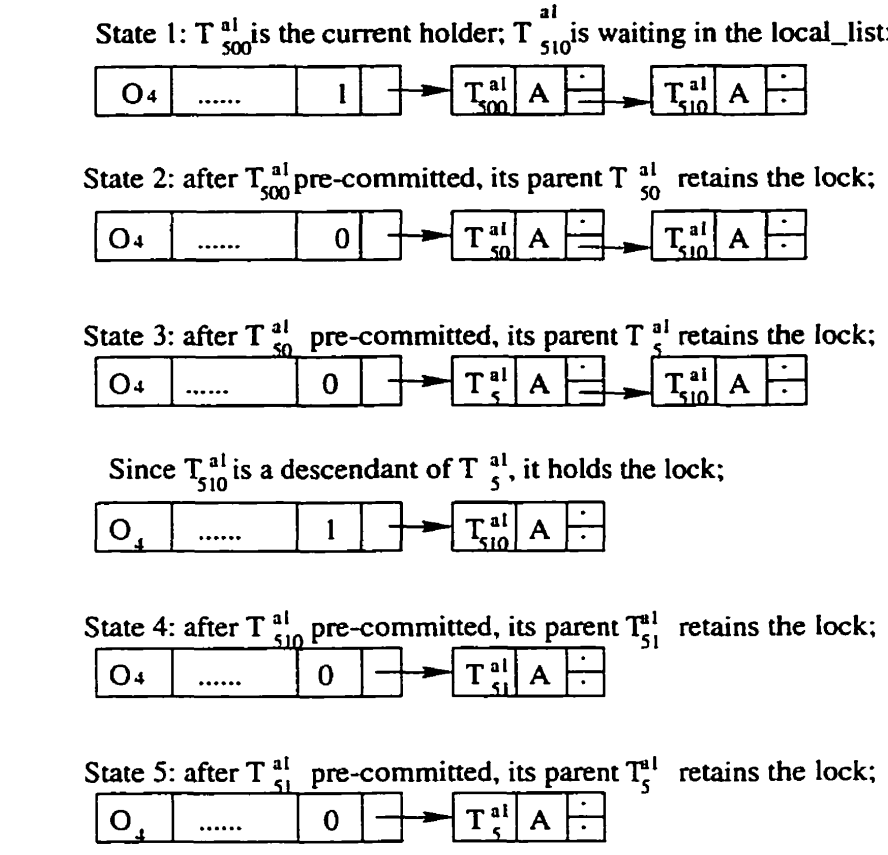


Figure 4.9: Lock Operations for O_4 for Family T_5^{al} on Node A

4.1.4 Correctness

The closed nested object two-phase locking rules are similar to Moss' closed nested two-phase locking rules. They both employ strict two-phase locking which is conflict serializable for each [sub-]transaction. Since the transaction model is not the same as Moss', the characteristics of objects and their access (via method invocation) make a non-object system and a persistent object system different. As discussed in Chapter 3, in a nested object transaction family, any transaction (not just leaf transactions) can directly access object attributes. The closed nested object two-

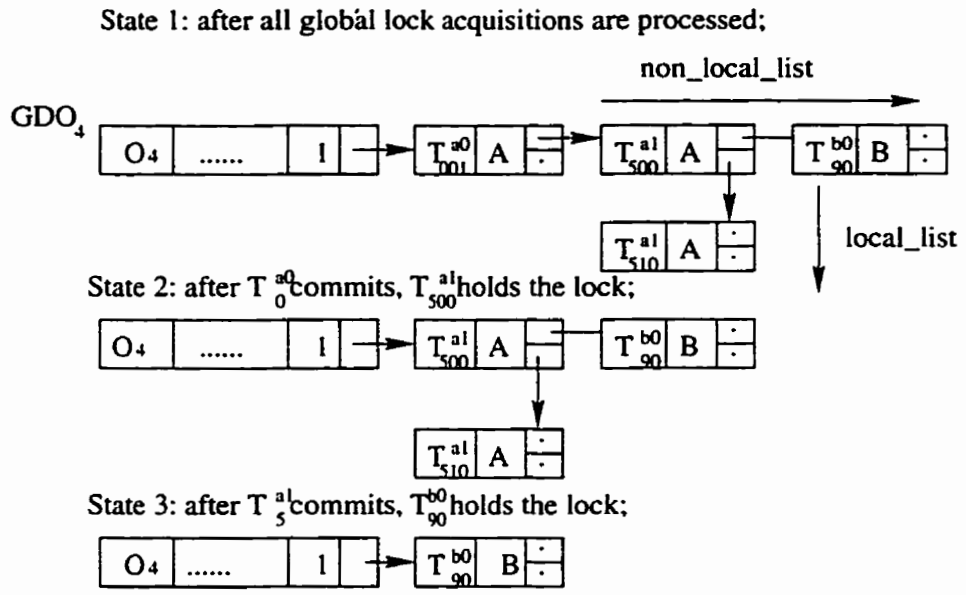


Figure 4.10: Lock Operations from Different Transaction Families in GDO_4 Entry

phase locking rules take into account these special features and differences and add additional functionality.

A method invoked by a nested object transaction does not affect the serialization of the nested object transaction unless it is on the same object (i.e., a directly or indirectly recursive invocation). When a parent transaction invokes a method on another object, due to encapsulation, the parent transaction and its descendant transaction can not conflict with each other. When a parent transaction invokes a method on the same object directly or indirectly, it introduces deadlock because the parent transaction can directly access the objects' attributes too. To avoid this kind of deadlock, we assume that no directly or indirectly recursive invocations occur in a transaction family.

Due to encapsulation, transactions executed on different objects cannot conflict with each other. Only when multiple object transactions execute concurrently on

the same object can conflicts occur. To ensure serializability, conflicting method invocations must execute in the same order as in some serial execution (i.e., be serializable). Rule (1), discussed in Subsection 4.1.1, prevents two nested object transactions from concurrently accessing an object in a conflicting manner.

If conflicting object transactions are from different transaction families, they are delayed until the root transaction of the conflicting transaction holding the lock commits or aborts. As a result, an order is introduced between conflicting transactions as well as corresponding transaction families. This order is the same order in which the relevant lock is obtained. A partial order produced using nested object two-phase locking rules among a set of transaction families is equivalent to a serial order which is the user's preferred order.

If conflicting transactions are from the same transaction family, no two conflicting sub-transactions will have an ancestor-descendant relationship because there are no recursive invocations allowed. Therefore, no deadlock occurs. By rule (1a), a sub-transaction is permitted to hold a conflicting lock only if all the conflicting sub-transactions which execute before the requesting sub-transaction have pre-committed and the lock has been retained by an ancestor of the requesting sub-transaction. Thus, only one sub-transaction within a transaction family may hold a lock at a time. Any other sub-transaction that requests the same lock will be blocked until the sub-transaction holding the lock pre-commits and an ancestor of the requesting sub-transaction comes to retain the lock. Consequently, an order will result between conflicting sub-transactions within a transaction family. This order is the same order in which the relevant lock is obtained. Therefore, if all sub-transactions in one transaction family are invoked in a depth-first order, which is the only correct serial order within a transaction family, the enforced order among conflicting sub-transactions is consistent with the depth-first order. In

addition, rule (1b) precludes all dependent sub-transactions from executing concurrently. It makes such sub-transactions execute sequentially in an order consistent with the order of method invocations in their parent transaction. Therefore, if all sub-transactions in one transaction family are invoked in the depth-first order, a partial order scheduled according to the closed nested object two-phase locking rules in one transaction family is equivalent to a depth-first serial order.

Since the special features made by objects and the closed nested object transaction model do not affect the serialization order in one transaction family or in a set of transaction families, a serializable execution order will be produced by using closed nested object two-phase locking rules.

4.2 Object Transactional Entry Consistency

The *Object Transactional Entry Consistency* (OTEC) protocol is presented in this section. This is followed by a discussion of the necessary modifications to the lock operation algorithms provided in Subsection 4.1.2 to support OTEC. Then the update transfer algorithm for OTEC and modified lock release algorithms are provided. An example is given to illustrate how OTEC works to ensure memory consistency in the proposed page based DSVM system.

4.2.1 Object Transactional Entry Consistency

The closed nested object two-phase locking rules ensure that the concurrent execution of a set of nested object transactions is serializable. To maintain memory consistency, a DSVM system needs to ensure that concurrently executing nested object transactions reference only object copies which are up-to-date. A DSVM

system must ensure that each nested object transaction never accesses a shared object without having received all updates to the shared object. Thus, a memory consistency protocol is required which efficiently transfers all updated objects (guarded by corresponding locks) to maintain memory consistency.

When a partial release of a lock is performed in one transaction family, the lock is granted to a single sub-transaction from the local lock list in the object's CGDO entry. Due to the data shipping nature of the DSVM system, all sub-transactions of a given root transaction will normally execute at the same node. Therefore, no network communication is required for such partial releases. When a lock is fully released from a root transaction, the object guarded by the lock is free to be acquired by other transaction families which may be from different nodes. Thus, to ensure the consistency of in-memory copies of an object across interconnected nodes, network communication is required to transfer the updated object at this time. This may result in significant network traffic. Instead of propagating an updated object to all nodes which cache the object, it can be propagated to other in-memory copies in a lazy fashion thereby reducing the amount of consistency information communicated. For example, propagation may be done as in lazy release consistency [KCZ92] or entry consistency [BZS93]. The updated object is transferred to a remote node only when an existing [sub-]transaction from another family at the remote node is granted object access.

Lazy release consistency and entry consistency are refinements of release consistency. To ensure the efficient update of cached data, lazy release consistency and entry consistency both lazily *pull* modifications across the interconnection network during lock acquisition (i.e. only for the acquiring processor). In this way, both the number of control messages and the amount of data exchanged are reduced.

On a lock acquisition, lazy release consistency requires *all updates to any shared*

data that precede the acquisition to be propagated to the acquiring processor. Since entry consistency requires each shared data object to be attached to a synchronization object, on a lock acquisition, only *updates to the shared data object associated with the acquired lock* that precede the acquisition must be propagated to the acquiring processor. As a result, entry consistency requires less data traffic than lazy release consistency.

In the proposed DSVM system, locking is explicitly tied to data objects, therefore, the entry consistency model provides a natural fit. The *Object Transactional Entry Consistency* protocol is defined by combining the closed nested object two-phase locking rules with the entry consistency model.

Definition 4.1 A memory model is said to be object transactional entry consistent, if:

When a lock is fully released from a root transaction, the subsequent acquiring [sub-]transaction TID_s from another family at a remote node is not allowed to access the shared object guarded by the lock until all updates to the object have been performed with respect to TID_s. ■

Informally, a shared access is considered *performed* at a process when its result is visible at that process [PK95]. In a page based system since one object's attributes can span multiple pages, instead of transferring all the pages of an updated object, only the pages which were updated more recently than those residing at the remote node need to be transferred. By doing this, OTEC can ensure that distinct, concurrent method invocations on objects executing at different nodes see a consistent view of the shared memory. Further, since OTEC pulls only an object's updated pages across network, only during the global lock acquisition, and only for the subsequent acquiring [sub-]transaction from a remote node, it further reduces

both the number of control messages and the *amount of* data exchange required for memory consistency maintenance.

This can be clarified by considering the example as shown in Figure 4.11. Transaction families T_i^{a1} and T_k^{a2} execute on node A. Transaction families T_j^b and T_i^c execute on nodes B and C respectively. Assume that all three nodes initially have an identical copy of object O_i that spans five pages in their memories (in the write-back queues since O_i has been unmapped at all nodes). Conflicting [sub-]transactions arrive at the shared object O_i in the order:

$$T_i^{a1} \rightarrow T_{j0}^b \rightarrow T_{j1}^b \rightarrow T_{ko}^{a2} \rightarrow T_i^c$$

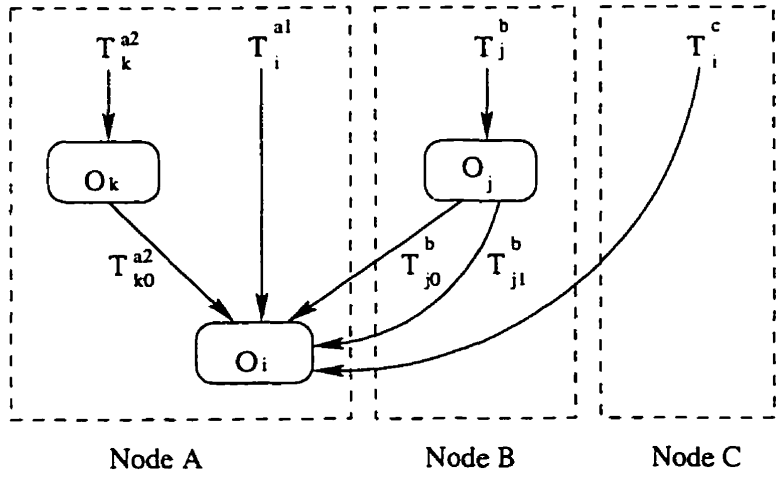


Figure 4.11: An Example of an Execution of Four Transaction Families

Transaction T_i^{a1} is granted the lock for O_i first and modifies page 0, 1, 2, and 3 of O_i. When it releases the lock, the next waiting sub-transaction T_{j0}^b is granted the lock. Before T_{j0}^b can access the object, the four updated pages must be transferred to node B from node A (shown in Figure 4.12 (A)). After T_{j0}^b modifies pages 0 and 3, it releases the lock to it parent T_j^b. Then the waiting sub-transaction T_{j1}^b is granted the lock from its ancestor T_j^b. Since T_{j0}^b and T_{j1}^b are from the same

transaction family on the same node B, no network traffic is required at this time (T^b_{j0} and T^b_{j1} shared a single copy of O_i on node B). After T^b_{j1} modifies page 1, it releases the lock to its parent T^b_j. Once the root transaction T^b_j releases the lock, the next waiting sub-transaction T^{a2}_{k0} on node A is granted the lock. Pages 0, 1, and 3 which were updated by T^b_{j0} and T^b_{j1} are transfered to node A from node B before T^{a2}_{k0} modifies page 4 (shown in Figure 4.12 (B)). When T^{a2}_{k0} pre-commits, its parent T^{a2}_k retains the lock. Once the root transaction T^{a2}_k commits, the lock is released to the next waiting transaction T^c_i on node C. The pages 0, 1, 2, 3, and 4 updated by T^{a1}_i, T^b_j, and T^{a2}_k are transferred to node C from node A before T^c_i modifies pages 0 and 2 (shown in Figure 4.12 (B)). Under OTEC, the last updating node C has all up-to-date pages of object O_i. However, not all the up-to-date pages may be updated on node C (e.g. the up-to-date pages 1, 3, and 4 on node C were updated on other nodes A and B respectively).

4.2.2 Algorithms for OTEC

Object access in a page based DSVM system was described in Subsection 3.1.3. If a set of [sub-]transactions at different nodes concurrently access a shared object O_i, when the lock for O_i is released by a sub-transaction's root transaction on one node the object O_i is unmapped from that node's page table and the object's pages are linked into the node's write-back queue. Once the subsequent requesting [sub-]transaction is granted the lock, its segmentation fault is resolved and object O_i's pages are mapped into the requesting node's page table. The required subset GDO_i is cached in the requesting node's memory as $CGDO_i$.

Under OTEC, if the subsequent acquiring [sub-]transaction is from the same node as the releasing transaction, the acquiring [sub-]transaction can reclaim O_i's

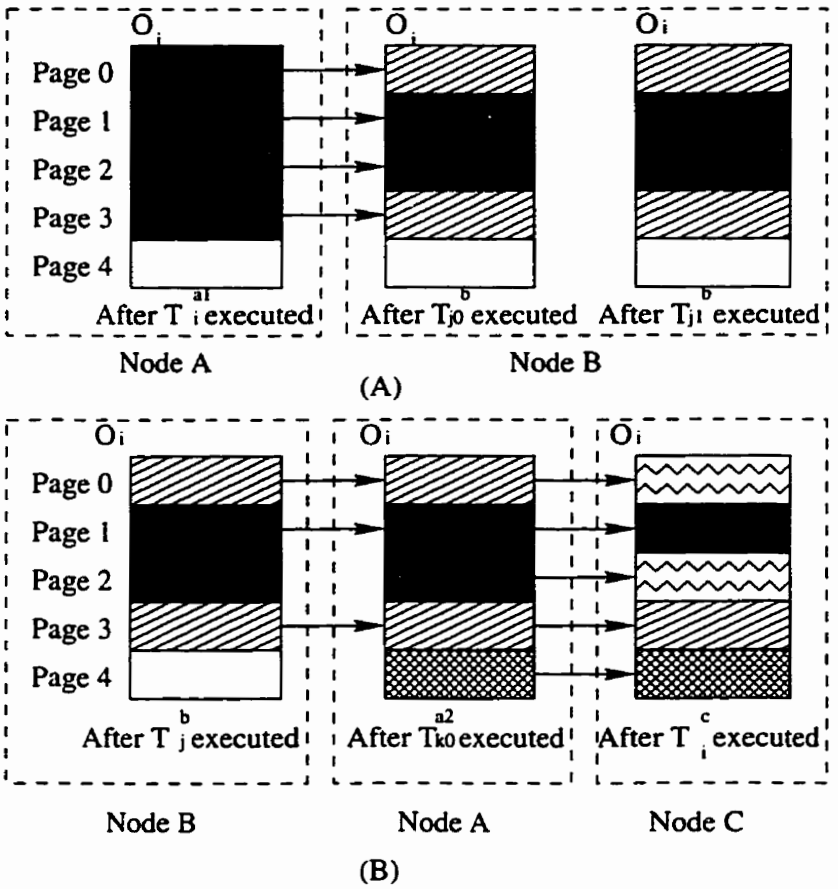


Figure 4.12: An Example of Updated O_i Transferred under OTEC

pages from the write-back queue without transferring any updated pages from other nodes. If the subsequent acquiring [sub-]transaction is from a different node, the system uses the information in $CGDO_i$ to transfer object O_i's updated pages to the acquiring node. Since object O_i's pages are all up-to-date on its *last updater* which is the node on which the last updating transaction executed, each *object*'s last updater must be tracked to allow the system to transfer updated object pages. Although object O_i's pages may not be all up-to-date on other nodes, some of the nodes (specific pages' last updaters) may still have some up-to-date pages. When a [sub-]transaction acquires the lock for O_i on such nodes, those pages do not need

to be transfered from the object's last updater.

To determine which pages were updated more recently than the pages residing at a remote acquiring node, the system has to know each *page*'s last updater. In the example shown in Figure 4.12, after transaction T_j^b releases the lock for object O_i, all O_i's five pages are up-to-date on its last updater (node B). When sub-transaction T_{k0}^{a2} on node A is granted the lock, since page 2's last updater is node A and since page 4 has not been subsequently updated, these two pages are up-to-date on the acquiring node A. Thus, only pages 0, 1, and 3 which were updated more recently than the pages residing at node A are transfered to node A from O_i's last updater (node B) before T_{k0}^{a2} modifies page 4. When transaction T_i^c is granted the lock on node C, all five pages are stale on node C because they were updated on nodes A and B respectively. Thus, object O_i's five pages are transfered from O_i's last updater (node A) to node C before T_i^c accesses pages 0 and 2. Therefore, each *page*'s last updater must be maintained to allow the system to determine which pages have the most recent version of the data.

Each object's last updating node identifier as well as each page's last updating node identifier are maintained in the object's GDO entry. The last updating node identifier for each object and each page can be updated during lock release using information on "dirty" pages from the virtual memory hardware. The algorithms defined in Subsection 4.1.2 for the local lock release process and the global lock release process need to be modified to deal with this information.

Before discussing the modification of the algorithms for lock release and the update transfer algorithm for OTEC, modifications to the data structures must be described.

OID_LIST

In Algorithm 4.3 (Local_Lock_Release) and Algorithm 4.4 (Global_lock_Release), the data structure OID_LIST was used for linking all objects held or retained by the releasing or aborting transaction. Each OID_LIST's element simply contained the objects' identifier (OID). Now, it needs to include a field to indicate whether or not the corresponding object has been updated. It also needs a field to indicate how many pages the object spans and a field for each object page to indicate if that page was updated. Therefore, each element of an OID_LIST is now composed of:

oid: indicates the object that is held or retained by the releasing or aborting transaction.

update: a flag. When it is '0', the object has not been updated; When it is '1', the object has been updated.

page_no: indicates how many pages the object spans.

page_l: a flag for each page l. When it is '0', the page l has not been updated: When it is '1', the page l has been updated.

GDO

Each object's GDO entry needs a field, last_updater, to indicate the object's last updating node (i.e. the object's "last updater"). It also needs a field, page_no, to indicate how many pages the object spans and a field, $page_l$, for each page to indicate page l's last updating node (i.e. the page's last updater). When an object's GDO entry is built, its $page_l$ fields are set to empty to indicate the object has the same version as in persistent storage. As discussed, an object's last updater is not

necessarily the same as each page's last updater, but once the object's pages which were updated more recently than those residing at a remote acquiring node are determined, they can always be retrieved from the object's last updater.

Algorithms

Algorithm 4.5 describes the modified local lock acquisition process for OTEC[3]. When a root transaction aborts, all updates to objects made by it and its sub-transactions have to be removed. So all *update* fields in its OID_LIST must be set to '0' to indicate that no updates were made to the objects linked in the OID_LIST. Then the release request is forwarded to the Global_Lock_Release routine. When a sub-transaction aborts, the locks which are not retained by any of its ancestors are first released locally to sub-transactions within the transaction family. If there is no sub-transaction in the family waiting for the locks at the time it aborts, the release request is forwarded to the Global_Lock_Release routine with all *update* fields in its OID_LIST set to '0'.

Algorithm 4.5 *Local_Lock_Release*

INPUT : $T^j_{i,l_1,\ldots,l_{d-2},l_{d-1}}$; /*TID of the releasing or aborting transaction*/
INPUT : OID_LIST;
/*all objects retained and held by the releasing or aborting transaction*/
INPUT : Release-mode; /* $\in$ {PC, RC, RA, SA} */
INPUT : NID_s; /*node-id on which the releasing or aborting transaction executes*/
INPUT : CGDO; /* cached GDO entries for objects in OID_LIST*/

CASE (Release-mode = PC): /*a sub-transaction pre-commits*/
    FOREACH (O_k in the OID_LIST) **DO**
        current_holder.Tid $\leftarrow$ $T^j_{i,l_1,\ldots,l_{d-2}}$; /* release locks to the parent*/
        $CGDO_k$.lock_variable $\leftarrow$ '0'; /*parent retains the lock*/

[3]This is a modification of Algorithm 4.3. The new lines are italicized in the pseudo-code of Algorithm 4.5.

```
        IF (T^j_{i,l1,...,ld-2} is an ancestor of the next transaction in the local_list)
        THEN
            CGDO_k.current_holder_pointer ← next TID in the local_list;
            CGDO_k.lock_variable ← '1'; /*holds the lock*/
            Send the lock grant to the requester;
CASE(Release-mode = RC): /*a root transaction commits*/
    Forward release request to Global_Lock_Release;
    /*Releases locks to other transaction families*/
CASE (Release-mode = RA): /*a root transaction aborts*/
    FOREACH (O_k in the OID_LIST) DO
        OID_LIST_k.update ← '0';
    Forward release request to Global_Lock_Release;
    /*Releases locks to other transaction families*/
CASE(Release-mode = SA): /*a sub-transaction aborts*/
    FOREACH (O_k in the OID_LIST) DO
        IF (O_k is retained by an ancestor of T^j_{i,l1,...,ld-2,ld-1}) THEN
            current.holder.Tid ← TID of the ancestor;
            CGDO_k.lock_variable ← '0';/*the ancestor retains the lock*/
            Unlink O_k from OID_LIST;
            IF (the next transaction in the local_list is the ancestor's descendant)
            THEN
                CGDO_k.current_holder_pointer ← next TID in the local_list;
                CGDO_k.lock_variable ← '1'; /*holds the lock*/
                Send the lock grant to the requester;
        ELSE
            IF (CGDO_k.current_holder_pointer != NULL) THEN
                CGDO_k.current_holder_pointer ← next TID in the local_list;
                Unlink O_k from OID_LIST; /*releases the lock locally first;*/
            ELSE
                OID_LIST_k.update ← '0';
    IF (OID_LIST != NULL) THEN
        Forward release request to Global_Lock_Release;
```

End of Algorithm

Algorithm 4.6 describes the global lock release process for OTEC[4]. For each object O_k in a committed root transaction's OID_LIST, the node identifier (NID_s)

[4]This is a modification of Algorithm 4.4. The new lines are italicized in the pseudo-code of Algorithm 4.6.

on which node the root transaction executed is stored in the last_updater field in GDO_k entry as object O_k's last updater. Then for each page l , if page l has been updated, the NID_s is stored in the $page_l$ field in GDO_k as page l's last updater to indicate that page l was updated at node NID_s. When a root transaction or a sub-transaction aborts, updates are discarded. Therefore, it is not needed to update the last_updater field in a released object's GDO entry.

Algorithm 4.6 *Global_Lock_Release*

INPUT : $T^j_{i,l_1,...,l_{d-2},l_{d-1}}$; /*TID of the releasing or aborting transaction*/
INPUT : OID_LIST;
/*all objects retained and held by the releasing or aborting transaction*/
INPUT : NID_s; /*node-id on which the releasing root transaction executes*/
INPUT : GDO; /* GDO entry for objects in OID_LIST*/

FOREACH (O_k in the OID_LIST) **DO**
IF (($T^j_{i,l_1,...,l_{d-2},l_{d-1}}$ = current_holder.Tid) or ($T^j_{i,l_1,...,l_{d-2},l_{d-1}}$ is an ancestor of current_holder.Tid)) **THEN**
IF (OID_LIST$_k$.update = '1') THEN /O_k has been updated*/*
*GDO$_k$. last_updater — 'NID_s'; /*stores object's last updater */*
FOREACH (page$_l$ of O_k in GDO$_k$) DO
IF (OID_LIST$_k$.page_l = '1') THEN
/ page l has been updated */*
GDO$_k$.Page$_l$ = 'NID_s';
/ stores each page's last updater in GDO$_k$ */*
IF (no other transaction waits for the lock) **THEN**
GDO$_k$.lock_variable — '0'; /*lock is free*/
GDO$_k$.current_holder_pointer — 'NULL';
ELSE
/*the next requiring transaction in the non_local_list gets the lock*/
GDO$_k$.current_holder_pointer — the next TID in the non_local_list; Send the lock grant to the requester and copy of GDO_k to build $CGDO_k$;

End of Algorithm

Algorithm 4.7 describes the update transfer process for OTEC. When a lock is fully released from a root transaction, if an existing [sub-]transaction from another

family is granted access to the corresponding object at a remote node, the object's pages which were updated more recently than those residing in the remote node's memory must be transferred to the remote node. To determine which updated pages of an object on the releasing node are more recent than those residing on the remote node. the Updates_Transfer algorithm compares each page l's last updater stored in $CGDO_k$ with the remote acquiring node identifier. If the remote acquiring node identifier is not the same as page l's last updater and the $page_l$ field in the CGDO entry is not empty, then the page in the remote acquiring node is stale. In this case, the up-to-date page l needs to be transferred from the object's last updater to the remote acquiring node.

Algorithm 4.7 *Updates_Transfer*

INPUT : $T^j_{i,l_1,\ldots,l_{d-2},l_{d-1}}$; /*TID of the releasing transaction*/
INPUT : NID_s; /*node-id on which the acquiring transaction executes*/
INPUT : O_k; /* Object being accessed */
INPUT : $CGDO_k$; /*CGDO entry for object O_k */

VAR Page_List : list of O_k's updated pages which need to be transferred from object O_k's last updater to the acquiring node NID_s;

FOREACH $CGDO_k.Page_l$ **DO**
    IF ($CGDO_k.Page_l \neq NID_s$) and ($CGDO_k.Page_l \neq$ 'NULL') **THEN**
        Link page l into Page_List;
Transfer all pages in the Page_List from $CGDO_k$.last_updater to the remote acquiring node NID_s;

End of Algorithm

4.2.3 Example of Updated Page Transfer under OTEC

We use the example shown in Figure 4.11 to illustrate how OTEC ensures memory consistency, especially, how it determines an object's up-to-date pages in a page-based DSVM system. The lock operations on shared object O_i's lock structure in

GDO_i and in each cached $CGDO_i$ are the same as was described in Subsection 4.1.3. A partial order is enforced among [sub-]transactions within a transaction family as well as among a set of transaction families as shown in the initial state of GDO_i in Figure 4.13. How the updated pages of O_i are transferred across the three nodes was shown in Figure 4.12. Now we focus on how to determine which object's pages were updated more recently than their local copies by using the algorithms defined in Subsection 4.2.2.

When T_i^{a1} globally releases the lock on node A, the node identifier A is stored in GDO_i's last_updater field and in the $page_0$, $page_1$, $page_2$, and $page_3$ fields (shown in Figure 4.13, State 1). After the next transaction T_{j0}^b is granted the lock, each page l's last updater stored in $CGDO_i$ is compared with the acquiring node identifier B. If they are not the same and the $page_l$ field in $CGDO_i$ is not empty, then page l in the acquiring node, B, is stale. Therefore, the updated pages 0, 1, 2, and 3 are transferred from O_i's last updater node A to node B before T_{j0}^b modifies pages 0 and page 3 of O_i. After T_{j0}^b pre-commits, the lock is locally released to its parent T_j^b and sub-transaction T_{j1}^b is then granted the lock by T_j^b. T_{j1}^b modifies page 1 without network traffic because T_{j0}^b and T_{j1}^b belong to the same transaction family and therefore share a single copy of O_i at node B. After T_{j1}^b pre-commits, the lock is passed up to its parent T_j^b. Once T_j^b commits, during the global lock release process, node identifier B is stored in GDO_i's last_updater field as well as in the $page_0$, $page_1$, and $page_3$ fields (shown in Figure 4.13, State 2). When the next sub-transaction T_{k0}^{a2} is granted the lock, since page 2's last updater (node A), stored in $CGDO_i$'s $Page_2$ field, is the same as the acquiring node A, and since the $Page_4$ field is empty, they are up-to-date on the acquiring node A. So only the updated pages 0, 1, and 3 are transfered from node B to node A. T_{k0}^{a2} then modifies page 4. Once the lock is globally released from the root transaction T_k^{a2}, node identifier

A is stored in GDO_i's last_updater field as well as in the $page_4$ field (shown in Figure 4.13, State 3). When the next transaction T_i^c is granted the lock, the last updaters of pages 0, 1, 2, 3 and 4 (which are nodes B, B, A, B, and A respectively) are all different than the acquiring node C. Therefore, the updated pages 0, 1, 2, 3, and 4 are transferred from O_i's last updater node B to node C before T_i^c is allowed to modify pages 0 and 2. Finally, when T_i^c globally releases the lock, the node identifier C is stored in GDO_i's last_updater and $page_0$ and $page_3$ fields (shown in Figure 4.13, State 4).

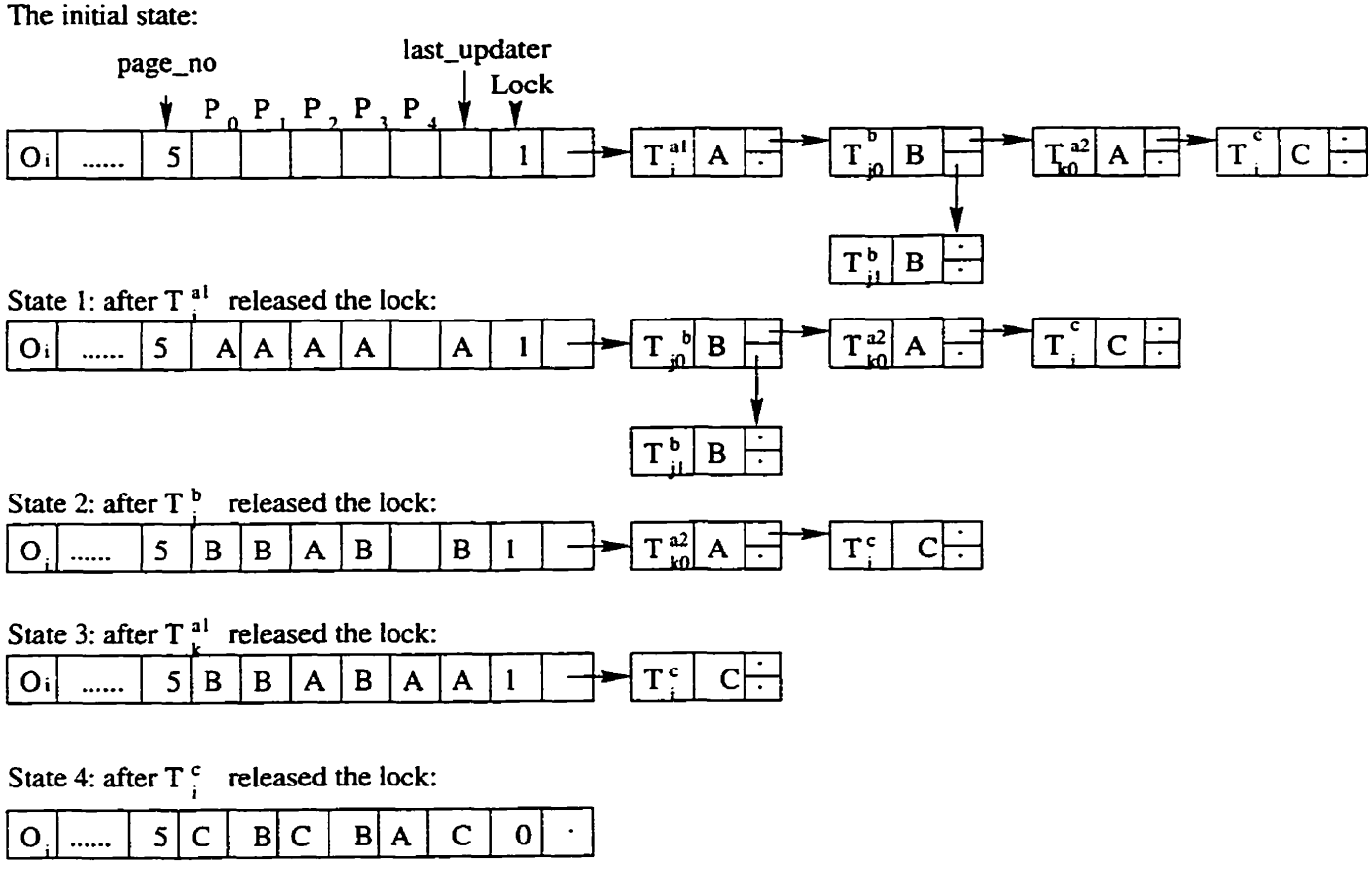


Figure 4.13: Lock Operations and Page Updates Information in GDO_i under OTEC

4.3 Lazy Object Transactional Entry Consistency

This section introduces the *Lazy Object Transactional Entry Consistency* (LOTEC) protocol for a page based DSVM system. It begins by providing the definition of lazy object transactional entry consistency. Then the update transfer algorithm for LOTEC and the modified lock release algorithms are provided. Following this, an example to show how LOTEC works to ensure memory consistency in the proposed DSVM system is presented.

4.3.1 Lazy Object Transactional Entry Consistency

In a page based virtual memory environment, there may still be some unnecessary data transferred while using OTEC. It is possible that a nested object transaction will not update all the pages of a shared object and it is also possible that a subsequent [sub-]transaction may not access all the stale pages of the shared object. This means that only a subset of the pages to be referenced by a subsequent [sub-]transaction may actually be stale.

In a persistent object system, each method of an object may access certain attributes. The compiler knows which attributes are in which pages (assuming objects are aligned to page boundaries). Therefore, the compiler can conservatively estimate which attributes a method may access. Thus, in the proposed DSVM system, it is possible for the system to transfer only the stale pages which may actually be referenced by the subsequent [sub-]transaction at the time of a lock acquisition. By doing so, we can further reduce unnecessary data transfer between nodes.

In the example shown in Figure 4.12, transaction T_{j0}^{b} attempts to modify pages

0 and 3 of O_i. The OTEC model requires *all updated pages* for object O_i made by T_i^{a1} (in this case, pages 0, 1, 2 and 3), to be transferred to node B from O_i's last updater node A, even though T_{j0}^{b} is not going to access page 1 and page 2. When sub-transaction T_{k0}^{a2} is granted the lock, updated pages 0, 1, and 3 are transfered from node B to node A even though T_{k0}^{a1} is not going to access those pages. Once transaction T_i^c is granted the lock released by T_k^{a2} and attempts to modify pages 0 and 2, all updated pages 0, 1, 2, 3, and 4 made by T_i^{a1} and T_j^b and T_{k0}^{a2} respectively are transferred to node C from the object's last updater, node A, even though T_i^c is not going to access pages 1, 3, and 4. In total, twelve updated pages are transferred across the network.

Figure 4.14 shows the same example using the modified LOTEC protocol. When T_{j0}^{b} is granted the lock released by T_i^{a1} and attempts to modify pages 0 and page 3, only the updated pages 0 and 3 are transferred to node B. When T_{j1}^{b} is granted the lock from its root transaction T_j^b which retained the lock from T_{j0}^{b}, the referenced updated page 1 is transferred from page 1's last updater, node A, to node B (Shown in Figure 4.14 (A)). When sub-transaction T_{k0}^{a2} is granted the lock from T_j^b and attempts to modify page 4, there is no transfer needed because page 4 is up-to-date on node A. Once T_i^c is granted the lock from T_k^{a2} and attempts to modify pages 0 and 2, the referenced updated page 0 and page 2 made by T_{j0}^{b} and T_i^{a1} are transferred to node C from page 0's last updater node B and page 2's last updater node A respectively (shown in Figure 4.14 (B)). In total, only five updated pages are transferred across the network. Therefore, using LOTEC, the number of pages exchanged is reduced relative to OTEC.

There are two issues associated with using LOTEC in a page based system. First, under the non-lazy OTEC protocol, the system retrieves *all* updated pages of an object for the next acquiring transaction at lock acquisition. All object's pages

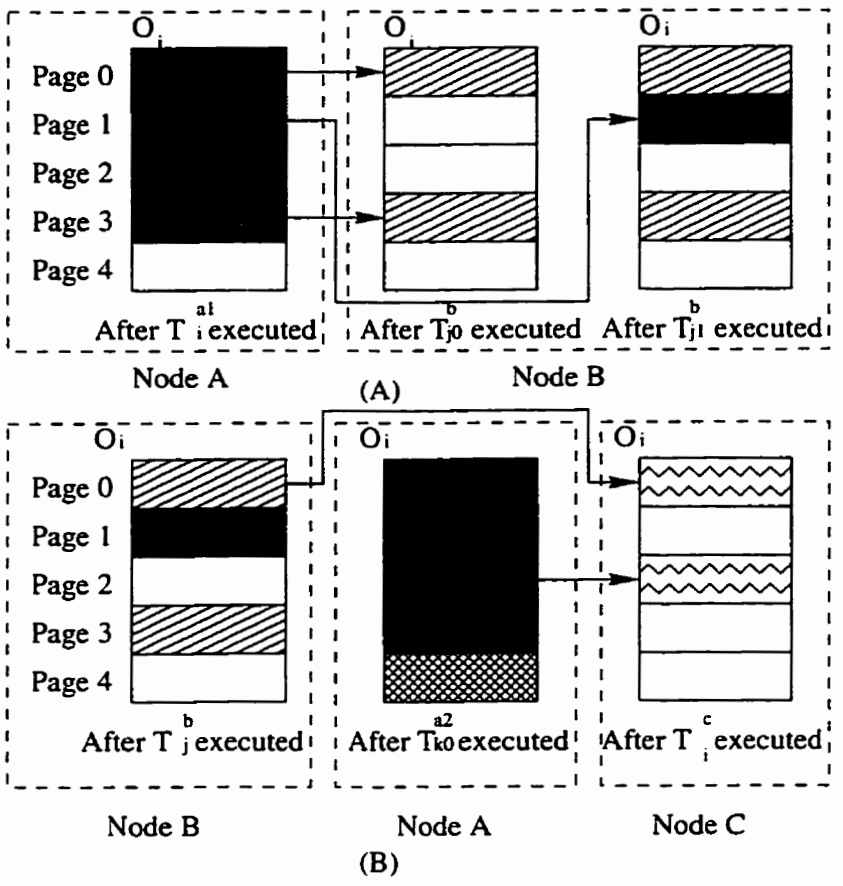


Figure 4.14: An Example of Updated O_i Transferred under LOTEC

are up-to-date on the releasing node (the object's last updater). Therefore, if a [sub-]transaction releases a lock locally or globally, the next acquiring [sub-]transaction from the same node may access the object guarded by that lock without needing to get up-to-date pages from other nodes. Thus, no network traffic is involved at the time the lock is exchanged within one transaction family or between transaction families on the same node. Under the LOTEC protocol, the system retrieves only the required *subset* of an object's updated pages that will be referenced by the next acquiring transaction. So there may still be stale object pages in the releasing node (i.e. the object's last updater). If the next acquiring [sub-]transaction from

either the same transaction family or a different transaction family at the same node attempts to access different pages than the releasing [sub-]transaction, and if those pages are stale, then the next acquiring [sub-]transaction needs to communicate with other nodes to get the up-to-date pages. Thus, network traffic may be required at the time the lock is exchanged within one transaction family as well as between transaction families on the same node. This is not additional communication only deferred communication. This delayed transfer is illustrated using the example shown in Figure 4.14 (A). When transaction T_{j1}^{b} on node B is granted the lock retained by its root transaction T_{j}^{b} and attempts to access page 1, the system must then transfer page 1 from page 1's last updater (node A) to the acquiring node B. This is because the last updating sub-transaction T_{j0}^{b} did not access page 1. Page 1 is still stale at node B.

The second issue arising when using LOTEC is that it is possible that the updated pages of an object might need to be transferred from multiple nodes (shown in Figure 4.14 (B) for node C) during lock acquisition. All updated pages of an object are transferred from the same node during lock acquisition under OTEC (shown in Figure 4.12). So LOTEC requires more message exchanging than OTEC, but the messages are smaller. Note, however, that LOTEC effectively distributes transfers over different links in a switched network. This allows beneficial concurrency in communication.

LOTEC pulls the updated pages of an object across the the network only when those pages are *referenced* by the next acquiring [sub-]transaction. Thus, LOTEC will require more short control messages than OTEC because the system may need to retrieve referenced updated pages from different nodes and communication may be required during the lock acquisition within one transaction family. Compared with exchanging large object updates, however, LOTEC will reduce unnecessary

data exchange in a page based system.

Definition 4.2 A memory model is said to be lazy object transactional entry consistent, if:

When a lock is released from a [sub-]transaction, the subsequent acquiring [sub-]transaction TID_s is not allowed to access the shared object guarded by the lock until updates to the object's pages which may be referenced by it have been performed with respect to TID_s. ■

4.3.2 Algorithms for LOTEC

Since LOTEC lazily pulls a subset of an object's updated pages across the network when those pages are accessed by a subsequent [sub-]transaction, the last updater of an object may not have all the up-to-date pages in its memory. Thus, all updated pages may not be retrieved from the object's last updater and the last_updater field for an object's last updater is no longer necessary in its GDO entry. Because, under LOTEC, the system may need to transfer the referenced updated pages for the next acquiring sub-transaction at the time the lock is exchanged within one transaction family, the pages updated by the releasing sub-transaction need to be recorded in the object's CGDO entry once the lock is *locally* released to releaser's parent. This allows the system to retrieve referenced updated pages from the pages' last updaters at the time the lock is granted to the subsequent acquiring sub-transaction in the same transaction family running on the same node.

Algorithm 4.8 describes the modified local lock release process for LOTEC[5]. When a sub-transaction pre-commits, during local lock release (to its parent), node

[5]This is a modification of Algorithm 4.5.The new lines are italicized in the pseudo-code of Algorithm 4.8.

identifier NID_s needs to be recorded in the corresponding *page* fields in $CGDO_k$ to indicate that those pages were updated on node NID_s.

Algorithm 4.8 *Local_Lock_Release*

INPUT : $T^j_{i,l_1,...,l_{d-2},l_{d-1}}$; /*TID of the releasing or aborting transaction*/
INPUT : OID_LIST;
/*all objects retained and held by the releasing or aborting transaction*/
INPUT : Release-mode; /* $\in$ {PC, RC, RA, SA} */
INPUT : NID_s;
/*node-id on which the releasing or aborting transaction executes*/
INPUT : CGDO; /* cached GDO entries for objects in OID_LIST*/

CASE (Release-mode = PC): /*a sub-transaction pre-commits*/
  FOREACH (O_k in the OID_LIST) **DO**
    FOREACH (OID_LIST_k.page_l in OID_LIST_k) DO
      IF (page_l = 1) THEN
        $CGDO_k.page_l \leftarrow NID_s$;
    current_holder.Tid $\leftarrow T^j_{i,l_1,...,l_{d-2}}$; /* releases locks to the parent*/
    $CGDO_k$.lock_avriable $\leftarrow$ '0'; /*parent retains the lock*/
    IF ($T^j_{i,l_1,...,l_{d-2}}$ is an ancestor of the next transaction in the local_list) **THEN**
      $CGDO_k$.current_holder_pointer $\leftarrow$ next TID in the local_list;
      $CGDO_k$.lock_variable $\leftarrow$ '1'; /*holds the lock*/
      Send the lock grant to the requester;
CASE(Release-mode = RC): /*a root transaction commits*/
  Forward release request to Global_Lock_Release;
  /*Releases locks to other transaction families*/
CASE(Release-mode = RA): /*a root transaction aborts*/
  FOREACH O_k in the OID_LIST **DO**
    OID_LIST_k.update $\leftarrow$ '0';
  Forward release request to Global_Lock_Release;
  /*Releases locks to other transaction families*/
CASE(Release-mode = SA): /*a sub-transaction aborts*/
  FOREACH (O_k in the OID_LIST) **DO**
    IF (O_k is retained by an ancestor of $T^j_{i,l_1,...,l_{d-2},l_{d-1}}$) **THEN**
      current.holder.Tid $\leftarrow$ TID of the ancestor;
      $CGDO_k$.lock_variable $\leftarrow$ '0';/*the ancestor retains the lock*/
      Unlink O_k from OID_LIST;
      IF (the next transaction in the local_list is the ancestor's descendant) **THEN**

$CGDO_k$.current_holder_pointer — next TID in the local_list;
$CGDO_k$.lock_variable — '1'; /*holds the lock*/
Send the lock grant to the requester;
ELSE
IF ($CGDO_k$.current_holder_pointer != NULL) **THEN**
$CGDO_k$.current_holder_pointer — next TID in the local_list;
unlink O_k from OID_LIST; /*releases the lock locally first:*/
ELSE
OID_LIST_k.update — '0';
IF (OID_LIST != NULL) **THEN**
Forward release request to Global_Lock_Release;

End of Algorithm

Algorithm 4.9 describes the modified global lock release process for the LOTEC protocol [6]. The only difference from Algorithm 4.6 is that there is no need to set the last_updater for each object's GDO entry.

Algorithm 4.9 *Global_Lock_Release*

INPUT : $T^j_{i,l_1,...,l_{d-2},l_{d-1}}$; /*TID of the releasing or aborting transaction*/
INPUT : OID_LIST;
/*all objects retained and held by the releasing or aborting transaction*/
INPUT : NID_s; /*node-id on which the releasing root transaction executes*/
INPUT : GDO; /* GDO entry for objects in OID_LIST*/

FOREACH O_k in the OID_LIST **DO**
IF (($T^j_{i,l_1,...,l_{d-2},l_{d-1}}$ = GDO_k.current_holder) or ($T^j_{i,l_1,...,l_{d-2},l_{d-1}}$ is an ancestor of GDO_k.current_holder)) **THEN**
IF (OID_LIST_k.update = '1') **THEN** /* O_k has been updated*/
/*The following line from Algorithm 4.6 has been removed*/
/*$GDO_k.last\_updater$ — 'NID_s';*/
/*stores object's last updater */
FOREACH $page_l$ of O_k in GDO_k **DO**
IF (OID_LIST_k.page_l = '1') **THEN**
/* page l has been updated */

[6]This is a modification of Algorithm 4.6. One line is removed from Algorithm 4.6 and it is italicized in the pseudo-code of the Algorithm 4.9.

$GDO_k.page_l$ = 'NID_s';
/*stores each page's last updater in GDO_k. */
IF (no other transaction waits for the lock) **THEN**
GDO_k.lock_variable ← '0'; /*lock is free*/
GDO_k.current_holder_pointer ← 'NULL';
ELSE
/*the next requiring transaction in the non_local_list gets the lock*/
GDO_k.current_holder_pointer ← the next TID in the non_local_list; Send the lock grant to the requester and copy of GDO_k to build $CGDO_k$;

End of Algorithm

Algorithm 4.10 describes the update transfer process for LOTEC. To determine which referenced updated pages of object O_k are more recent than others, the Updates_Transfer algorithm compares the last updater of each *referenced* page l stored in $CGDO_K$ with the acquiring node identifier. If the acquiring node identifier is not the same as a referenced page l's last updater and the $page_l$ field in $CGDO_k$ is not empty, then the referenced page l in the acquiring node is stale and the updated page needs to be transferred from page l's last updater. Different pages may, of course, be transferred from different nodes depending on their last updaters.

Algorithm 4.10 *Updates_Transfer*

INPUT : $T^j_{i,l_1,\ldots,l_{d-2},l_{d-1}}$; /*TID of the releasing transaction*/
INPUT : NID_s; /*node-id on which the acquiring transaction executes*/
INPUT : O_k; /*Object being accessed*/
INPUT : $CGDO_k$; /* CGDO entry for object O_k */
INPUT : Page_List; /* list all pages of O_k which will be referenced; each element of Page_List has a last_updater field to store the page's last updater */

FROEACH ($Page\_List_l$ for O_k in Page_List) **DO**
IF ($CGDO_k.page_l \neq NID_s$) and ($CGDO_k.page_l \neq$ 'NULL') **THEN**
$Page\_List_l$.last_updater ← $CGDO_k.page_l$;
ELSE
Unlink $page_l$ from the Page_List;

Transfer pages listed in the Page_List from pages's last updating nodes. if some pages have the same last updating node, they will be transferred from the same node all in one;

End of Algorithm

4.3.3 Example of Updated Page Transfer under LOTEC

The same example shown in Figure 4.11 is used to illustrate how LOTEC ensures memory consistency, especially, how an object's updated pages which will be referenced by the next acquiring transaction are determined in a page based DSVM system. The lock operations in the shared object O_i's lock structure in GDO_i and in each cached $CGDO_i$ entry are the same as was described in Subsection 4.1.3. A partial order is enforced among [sub-]transactions within one transaction family as well as among a set of transaction families as shown in the initial state of GDO_i in Figure 4.15. How the referenced updated pages of O_i are transferred across the three nodes was shown in Figure 4.14. We now focus on how to determine which referenced updated pages of an object are more recent than others by using the algorithms defined in Subsection 4.3.2.

When transaction T_i^{a1} globally releases the lock on node A, the node identifier A is stored in GDO_i's $page_0$, $page_1$, $page_2$, and $page_3$ fields as their last updater (shown in Figure 4.15, State 1). After the next transaction, T_{j0}^b, is granted the lock on node B, each referenced page's last updater stored in $CGDO_i$ is compared with the acquiring node identifier B. Since T_{j0}^b attempts to modify page 0 and page 3, and they were both last updated at the releasing node A which is not the same as the acquiring node B, the updated pages 0 and 3 of O_i are transferred to node B. After T_{j0}^b pre-commits, the lock is locally released to its parent T_j^b and node identifier B is stored in $CGDO_i$'s $page_0$ and $page_3$ fields. Sub-transaction T_{j1}^b is then granted the

lock from T_j^b. Since the referenced page 1's last updater in $CGDO_k$ is not the same as the acquiring node B, the updated page 1 is transferred from its last updater, node A, to the acquiring node B before T_{j1}^b is allowed to modify it. After T_{j1}^b pre-commits, the lock is locally released to its parent T_j^b and node identifier B is stored in $CGDO_i$'s $page_1$ field. When T_j^b commits, during the global lock releasing, node identifier B is stored in GDO_i's $page_0$, $page_1$, and $page_3$ fields as the corresponding pages' last updater (shown in Figure 4.15, State 2). When the next sub-transaction T_{k0}^{a2} is granted the lock on node A, because the referenced $page_4$ field in $CGDO_i$ is empty (which means that it has not been updated), no transfer is needed before T_{k0}^{a2} modifies page 4. After T_k^{a2} commits, page 4's last updater, node A, is stored in the $page_4$ field in GDO_i (shown in Figure 4.15, State 3). Once the next transaction T_i^c running on node C is granted the lock, since the node identifiers in the referenced $page_0$ and $page_2$ fields of $CGDO_i$ are B and A respectively, which are not the same as the acquiring node C, the updated pages 0 and 2 are transferred from their last updaters node B and A, respectively, to the acquiring node C. After T_i^c commits, it globally releases the lock and node identifier C is stored in GDO_i's $page_0$ and $page_2$ fields as their last updater (shown in Figure 4.15, State 4).

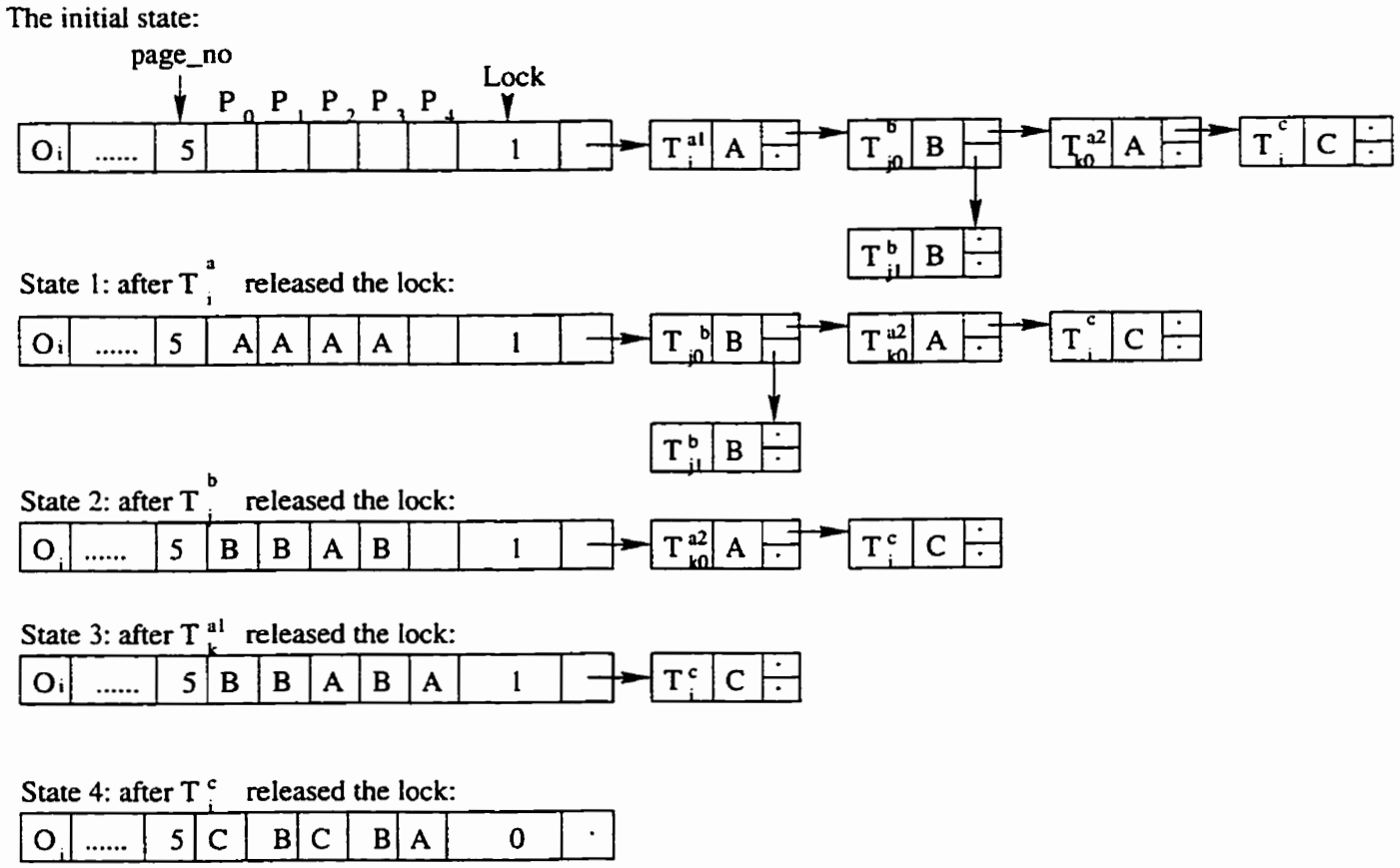


Figure 4.15: Lock Operations, Page Updates Information in GDO_4 under LOTEC

Chapter 5

Network Load Analysis

This chapter presents the results of analysis performed to assess the network performance of OTEC and LOTEC. The closed nested object two-phase exclusive locking rules (O2PL) are used to provide concurrency control on a per-object basis in a simulated DSVM system. We then measure the number of short control messages and the number of object data exchange messages required to maintain memory consistency for an arbitrary shared object using each of three protocols: OTEC, LOTEC, and a conservative object transactional entry consistency protocol (COTEC) which transfers the entire object state at the time the lock is granted to a remote node. The simulation strategy and parameters are discussed first. This is followed by a description of the design of the simulator. The chapter concludes by presenting the results of the study.

While this is not a "traditional" simulation, it does provide a suitable abstraction of the described consistency protocols that allows us to gather information on the network load induced by their use. This is the key consideration in the acceptance of a consistency protocol. By incorporating the actual consistency and

lock management algorithms into the simulation process we also increased our confidence in their correctness and gained insight into potential implementation issues. Some readers may consider our approach to be emulation rather than simulation.

5.1 Simulation Strategy

In a DSVM system, when closed nested object two-phase exclusive locking is used to synchronize conflicting nested object transactions, deadlock may occur between transaction families. Since deadlock is not discussed in this thesis, rather than measuring the number of exchanged messages for all objects shared by all concurrently executing nested object transaction families, we simply measure the number of exchanged messages for an arbitrary individual object shared by the concurrently executing nested object transaction families. By doing so, we can evaluate the performance of the three memory consistency protocols for any object shared by [sub-]transactions without concern for deadlock between transaction families.

For any selected shared object, a partial order among a set of conflicting transaction families which concurrently execute in the system is produced using O2PL. Since blocked conflicting transaction families cannot execute until the acquiring transaction family commits, this partial order is equivalent to a serial order in which all conflicting transaction families execute sequentially. Therefore, we can randomly select an execution order for a set of concurrently executed transaction families as the partial order produced by O2PL for any shared object. When all transaction families execute in the randomly selected order sequentially, an execution order produced using O2PL for any selected object among all conflicting [sub-]transactions is the same as the partial order which is assumed.

We evaluate the network performance of the three memory consistency protocols

(OTEC, LOTEC, and COTEC) by counting the number of short control messages and the amount of data exchanged for large object updates in maintaining memory consistency. The simulation allows us to compare the performance of the three protocols.

5.1.1 Message Counts

Since concurrency control is provided on a per-object basis via object-level locking, communication occurs on lock and unlock operations. To maintain consistency, an updated object is transfered only when a transaction acquires the lock using any of COTEC, OTEC, or LOTEC. Thus messages are counted for the acquiring transaction during lock acquisition.

When a [sub-]transaction acquires a lock from a remote node, two short control messages and one long data message are required to maintain memory consistency under COTEC and OTEC. One short control message is sent to the object's last updating node to request forwarding of the updated object to the acquiring node. Then the object's last updating node sends a long data message (consisting of the updated object's pages) in response to the acquiring node's request. After the acquiring node receives the updated object, it sends a short control message to the object's last updating node to acknowledge successful receipt of the data. For OTEC, the long data message includes only the object's *updated* pages while for COTEC it includes *all* the object's pages.

For LOTEC, two short control messages and one long data message are required not only when the acquiring [sub-]transaction is from a remote node, but possibly also when the acquiring [sub-]transaction is from the same node. Only three messages are required if referenced stale pages have the same last updater. Otherwise,

more messages are required. These are exchanged between the acquiring node and the different pages' last updaters. For LOTEC, the long data message includes only the object's updated pages that are *referenced* by the acquiring [sub-]transaction.

To compare the three different memory consistency protocols, exchanged messages are counted by size, in bytes. We simulate where each control message is 32, 64 or 128 bytes, and where the page size is 1024, 2048, or 4096 bytes.

5.1.2 Parameters for the Simulation

We have designed a set of experiments to study the performance of the three memory consistency protocols described earlier. In the simulated system, there are 16 nodes. To ensure that every node has active transactions, we assume that each node creates 4 nested object transaction families. Each parent transaction can have 0 to 10 sub-transactions and the depth of a nested transaction family can be 1 to 5. Therefore, there are 64 nested transaction families concurrently executing across 16 nodes in the system. Each transaction family is created randomly, and can have at most 11,110 sub-transactions. It is assumed that there are no directly or indirectly recursive invocations on an object (to avoid deadlock in one transaction family). When each transaction family and its sub-transactions are randomly created, the objects they invoke are randomly selected and checked to avoid deadlock.

The maximum number of pages for each object and the maximum number of objects in the system are specified as input parameters. When the GDO is created, the number of pages for each object is set between 1 and the maximum number of pages. This ensures that objects in the system have varying numbers of pages. When a nested transaction family is randomly created, each transaction can update any number of pages in the invoked object.

The values chosen for these parameters have been selected to maximize the probability of conflicting transactions since this is the interesting case. Fewer conflicting transactions will never affect network load negatively.

5.2 The Simulator

We have constructed a simple simulator to evaluate the protocols presented earlier. The DSVM system modeled by the simulator consists of a collection of nodes interconnected by a network. Each node in the network has a processor and local memory which acts as a cache. The simulator also maintains a GDO, and a global lock server using the closed nested object two-phase exclusive locking rules for synchronization among transaction families across all nodes.

Each node is modeled as a set of three processes: a transaction generator, a transaction manager, and a local lock server. The transaction generator generates a set of nested object transaction families which concurrently execute on the node. Each transaction family can access any shared object described in the GDO as long as no directly or indirectly recursive invocations occur within the transaction family.

The transaction manager receives lock operation requests from transactions and then sends the requests to the local or global lock server. For a lock request, once the local or global lock server acknowledges that the lock is set, the transaction manager allows the acquiring transaction to execute. Otherwise the requesting transaction will be blocked until the lock is available. When the lock is granted from the global lock server, the transaction manager caches the object's GDO entry in the local memory. Before the acquiring transactions are allowed to execute, the transaction manager builds an updated version of the object using the current

memory consistency protocol (COTEC, OTEC, or LOTEC). For a lock release request, when a root transaction requests lock release, the transaction manager "unmaps" all objects which are held or retained by the root transaction, then sends a request to the global lock server to release the locks to other transaction families. When the release request is from a sub-transaction, the transaction manager sends the request to the local lock server. Once the local or global lock server responds, the lock is released to the next waiting [sub-]transaction and the transaction manager resumes that [sub-]transaction.

Using O2PL, the local lock server handles all lock operations locally which are sent by the transaction manager for the sub-transactions from the same transaction family as the current lock holder.

5.3 Results and Discussion

This section evaluates the performance of COTEC, OTEC, and LOTEC in the simulated DSVM system. We analyze differences in performance by changing the maximum number of pages per object and the maximum number of objects in the system for a randomly selected subset of the objects which are shared by randomly created nested transactions. The size of control messages and the size of pages are also varied. It is assumed that all 16 nodes initially have an updated copy of any shared object in their memories (caches). Subsection 5.3.1 presents some of the simulation results. It is followed by a discussion of their significance in Subsection 5.3.2.

5.3.1 Results

Figures 5.1 to 5.4[1] present the results obtained by changing the maximum number of pages per object and the maximum number of objects in the system, and show the total number of bytes for messages transfered during lock acquisition under the three memory consistency protocols for a randomly selected subset of the shared objects. The short control message size is fixed at 64 bytes, and the page size is 2048 bytes.

Figure 5.1 shows the results for the three protocols when each object spans 1 to 5 pages and 20 objects are shared repeatedly by 33 to 102 randomly created [sub-]transactions across 10 to 16 nodes. When objects have a small number of pages and are shared by many [sub-]transactions repeatedly, COTEC and OTEC have very similar performance. This is because when an object with a small number of pages is shared by multiple sub-transactions repeatedly within one transaction family, although conflicting sub-transactions may update different sets of the object's pages, all its pages or nearly all its pages may be updated at the time the lock is released from the root transaction. In such a case, OTEC will transfer the same or nearly the same number of pages as LOTEC for a subsequent remote access. The same reasoning applies to LOTEC's performance. However, since LOTEC delays transferring updated pages until they are referenced, LOTEC achieves slightly better performance than both COTEC and OTEC. In one case LOTEC may actually have to transfer more bytes than both COTEC and OTEC because it may transfer the same pages as COTEC and OTEC but does so using more control messages (e.g. object O_6). When a shared object has a single page, the three protocols always have the same performance (e.g. objects O_0 and O_{15}).

[1]Raw result data is presented in Appendix A.

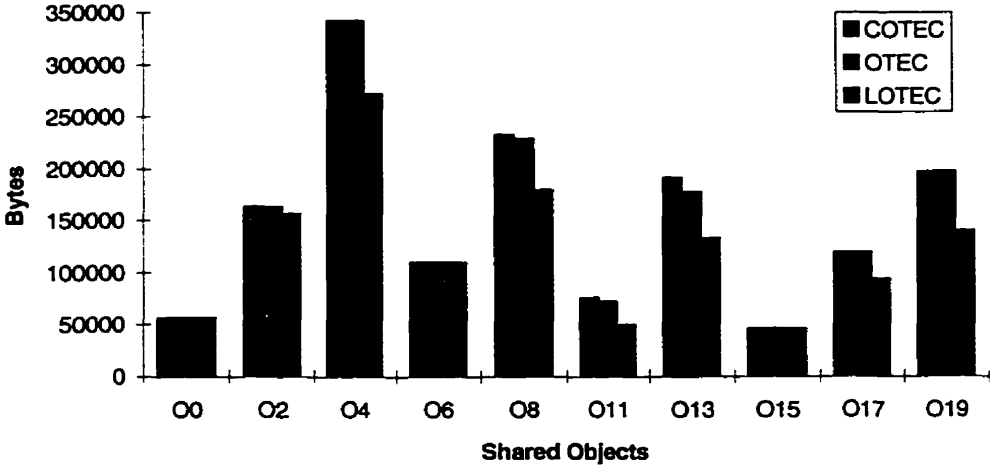


Figure 5.1: 1-5 pages/object, 20 objects, 33-102 conflicting transactions

Figure 5.2 shows the results for three protocols when each object spans 10 to 20 pages (i.e. larger objects) and 20 objects are shared by 33 to 85 randomly created [sub-]transactions across 12 to 16 nodes. Compared with the previous scenario, OTEC has better performance than COTEC in every case although their performance is still very close. LOTEC has much better performance than both COTEC and OTEC in every case. This is because when an object spans a large number of pages the possibility that all its pages are updated by multiple sub-transactions within one transaction family is smaller than when an object spans a small number of pages.

Figure 5.3 shows the results for the three protocols when each object spans 1 to 5 pages and 100 objects are shared by 5 to 24 randomly created [sub-]transactions across 3 to 9 nodes. Compared with the first test case, an object is now shared by less sub-transactions within one transaction family. OTEC has better performance than COTEC in most cases, and LOTEC also has better performance than both COTEC and OTEC in most cases. This is because when an object with a small

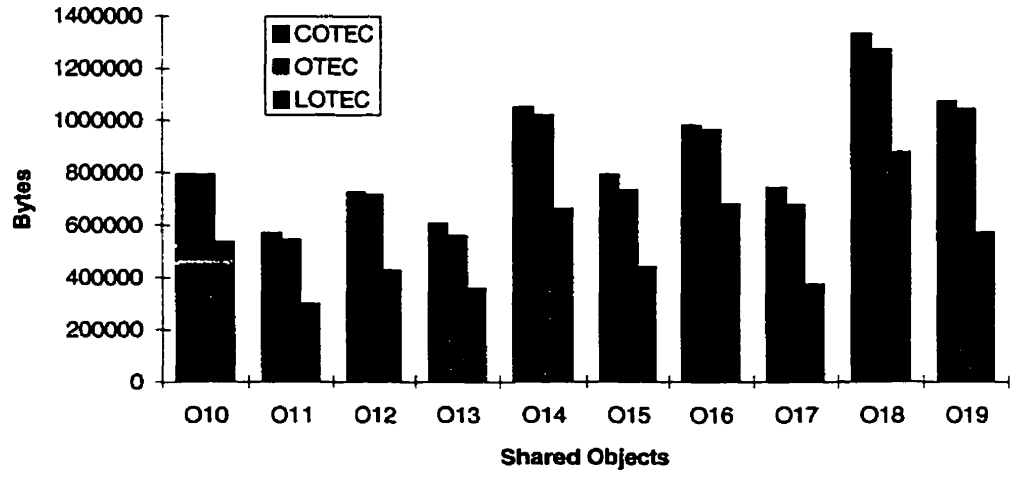


Figure 5.2: 10-20 pages/object, 20 objects, 33-85 conflicting transactions

number of pages is shared by fewer sub-transactions from the same transaction family, the possibility that all its pages are updated within one transaction family is smaller than when it is shared by a large number of sub-transactions repeatedly within one transaction family. As discussed previously, when an object has a single page, the three protocols have the same performance (e.g. object O_{25}), and when LOTEC transfers the same pages as COTEC and OTEC but requires more short control messages than they do, it has worse performance (e.g. object O_{67}). In such a situation, the performance of LOTEC is only marginally worse because the size of a control message is relatively small compared to the size of a page. There is one case in the results where LOTEC has the same performance as both COTEC and OTEC but the shared object does not have a single page. This is because LOTEC transfers the same number of pages and requires the same number of short control messages as COTEC and OTEC (e.g. object O_{46}). This happens when conflicting [sub-]transactions access all of an object's pages. When an object has fewer pages, this is more likely to happen.

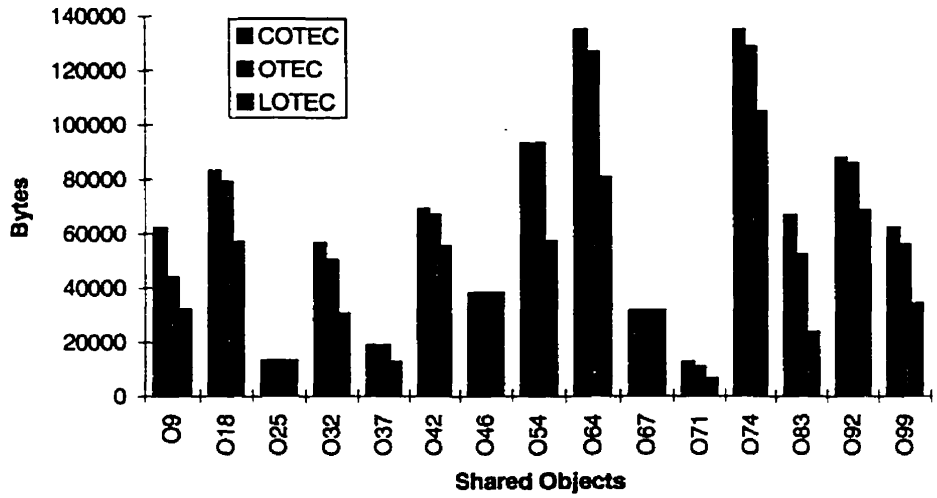


Figure 5.3: 1-5 pages/object, 100 objects, 5-24 conflicting transactions

Figure 5.4 shows the results for the three protocols when each object spans 10 to 20 pages and 100 objects are shared by 3 to 22 randomly created [sub-]transactions across 2 to 11 nodes. Compared with the previous three test cases, objects now span a large number of pages and are shared by less [sub-]transactions within one transaction family. In this case, both OTEC and LOTEC achieve better performance. LOTEC appears to offer increased benefit as the level of concurrent sharing across distributed nodes increases (e.g. comparing object O_{18} shared by 22 [sub-]transactions from 11 nodes to O_{12} shared by 12 [sub-]transactions from 5 nodes).

Figure 5.5 shows the total number of bytes of all the messages transferred under COTEC, OTEC, and LOTEC for object O_{18} in the second test case. The size of short control messages varies between 32, 64 and 128 bytes, and the size of pages varies between 1024, 2048 and 4096 bytes. 68 short control messages and 34 long data messages (646 pages under COTEC and 618 pages under OTEC) among 15 nodes are transferred under COTEC and OTEC. 210 short control messages and 105

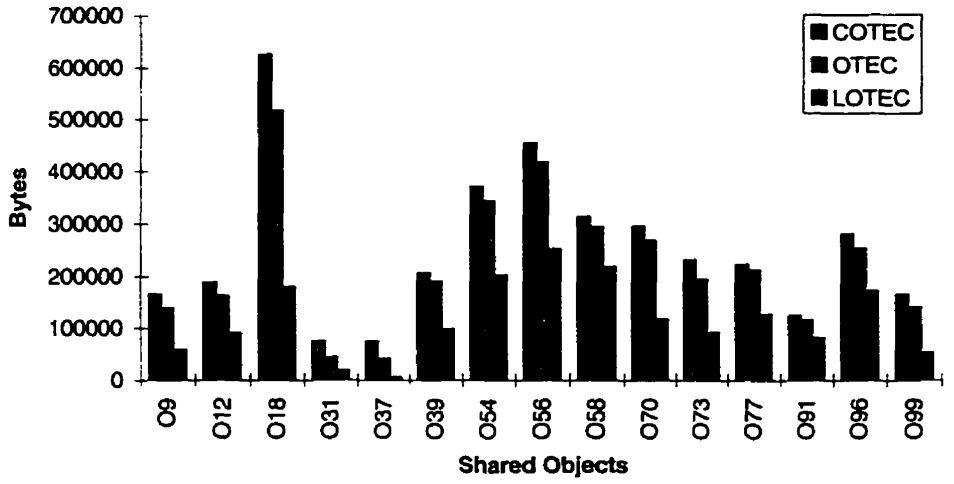


Figure 5.4: 10-20 pages/object, 100 objects, 3-22 conflicting transactions

long data messages (420 pages) are transferred under LOTEC. Note that LOTEC transfers less bytes than both COTEC and OTEC as the page size increases. When the size of short control messages decreases, the benefit of LOTEC increases only slightly. The large number of short control messages required by LOTEC does not hurt its performance in terms of the amount data sent.

5.3.2 Discussion

All simulation results indicate that LOTEC has the best performance in terms of the amount of data sent by the three memory consistency protocols. Results also show that LOTEC transfers at most the same set of pages as COTEC and OTEC, but it may require more short control messages than COTEC and OTEC because there is no single last updater for all an object's pages.

LOTEC only transfers *referenced* updated pages to an acquiring transaction during lock acquisition. When a [sub-]transaction (pre-)commits, only the refer-

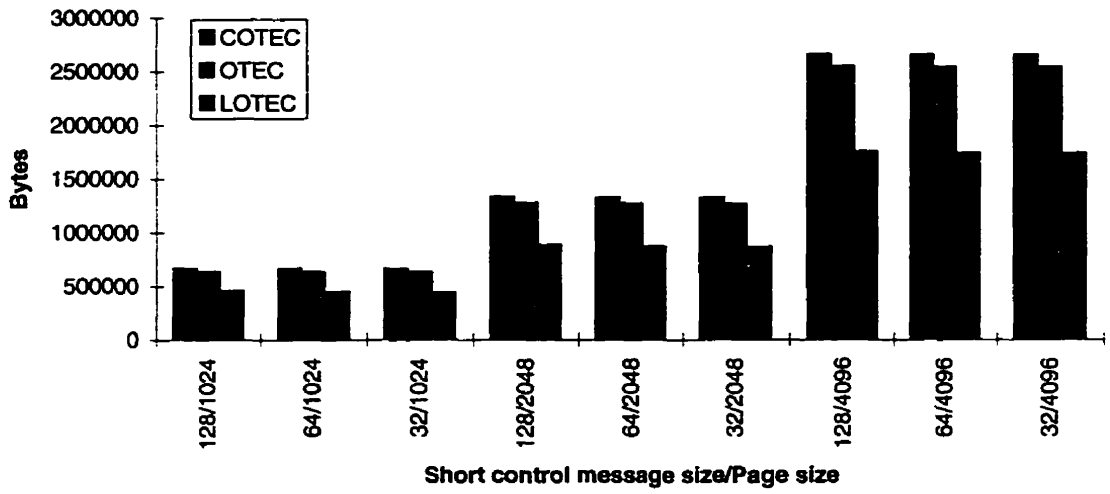


Figure 5.5: The results by changing the size of the short and the long messages

enced pages are updated and other pages may still be stale in the memory of the acquiring processor. If the acquiring transaction is from the same node as the releasing transaction but references different object pages which are stale, LOTEC needs to retrieve the updated pages from those pages' last updating nodes. If referenced pages are last updated on different nodes, LOTEC needs to retrieve the pages from *different* nodes. Therefore, there are two cases when LOTEC has potentially the same or worse performance than COTEC or OTEC. The first case is when LOTEC transfers the same number of updated pages and requires the same number of short control messages as the other protocols. The other case is when LOTEC retrieves the same number of pages or a few pages less than them but requires more short control messages. Since the size of control messages is typically much smaller than the size of data messages, in the worse case, LOTEC will only transfer a few more bytes than OTEC.

The results show that, in general, LOTEC transfers less pages than COTEC

and OTEC in the simulated DSVM system especially when the sharing level among distributed nodes increases. Since the worst case for LOTEC will happen only occasionally and it hardly hurts the performance when compared to the number of large data updates reduced by LOTEC, we conclude from our simulation results that LOTEC will, generally, transfer far fewer bytes than both COTEC and OTEC in total in a large scaled distributed persistent object system. Further, when objects span more pages or the page size is bigger, LOTEC achieves even better performance.

We have assessed the effectiveness of the three consistency protocols by counting messages and their lengths, then we evaluated the cost of each message in terms of its length alone (i.e. time to send was determined solely by the number of bytes sent). However, the real message cost is actually a function of message length and network communication protocol overhead which is the time required to push a message into the network interface at the sending end and pull it out at the receiving end. Thus, the message cost is composed of both software overhead and network latency. Recent advances in network technology have dramatically improved communication performance for network applications. The development of high bandwidth, and low latency networks has shifted the bottleneck in communication from the limited bandwidth of networks to the high latency of the network communication software. Despite this, small messages are important in many applications (e.g. LOTEC).

To investigate the value of the consistency protocols presented in this thesis in a conventional network environment with high software overhead and in a new network environment with much lower software overhead, we re-assessed the performance of the three protocols in terms of message cost based on both the estimated software overhead and the transmission rate of the network. Message cost is mea-

sured as $T_{message} = 2N_cT_s + L_l / T_b$, where N_c is the number of messages (small and large), T_s is the software cost for each message sent or received, L_l is the total length of the transfered messages, and T_b is the transmission rate.

Figures 5.6, 5.7, and 5.8 show the results of the modified message cost under the three protocols for object O_{14} in the second test case. 68 short control messages and 34 long data messages (510 pages) are transfered under COTEC, while with the same number of messages required as COTEC, 495 pages are transferred under OTEC. 188 short control messages and 94 long data messages (316 pages) are transferred under LOTEC. The software cost for each message sent and received varies between 100μsec, 20μsec, 5μsec, 1μsec, and 500$n$sec. The transmission rate of the network varies between 10Mbytes/sec, 100Mbytes/sec, and 1Gbyte/sec [2]. The size of short control messages is 32 bytes, and the page size is 4096 bytes.

Figure 5.6 shows that in a network environment with 10Mbytes/sec transmission rate, the message cost decreases under LOTEC as the software cost is reduced. In a network environment with 100Mbytes/sec transmission rate, the message cost decreases again when the software cost decreases (shown in Figure 5.7). However, in a network environment with 1Gbyte/sec transmission rate, the software cost becomes critical and must be very low. LOTEC achieves better performance than both COTEC and OTEC until the software cost reduces to 1μsec (shown in Figure 5.8). Therefore, the importance of reducing per-message software overhead is significant to make our protocols practical particularly for high speed networks. Development of new mechanisms with much lower software overhead will provide a better environment to support applications which have many short messages like LOTEC. Recently, such mechanisms have been developed which reduce software

[2]Good performance for LOTEC would also be expected for lower transmission rates more typical of wide area networks (e.g. 100Kbps).

overhead successfully (e.g. Active messages [ECGS92] and U-Net [EBBV95]).

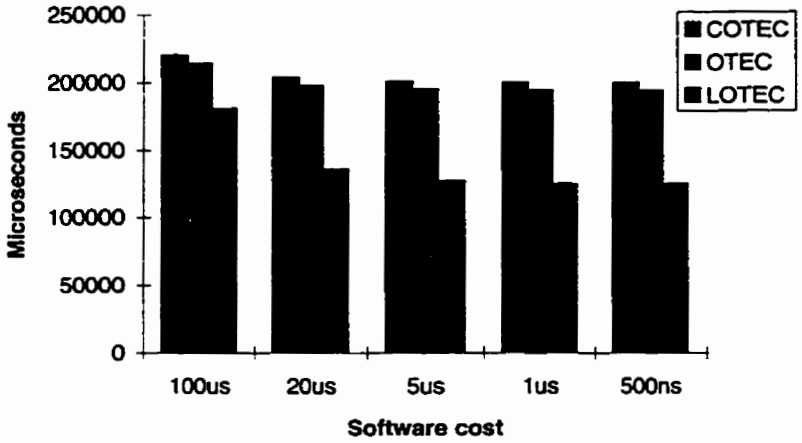


Figure 5.6: Message cost for O_{14} when transmission rate is 10Mbps

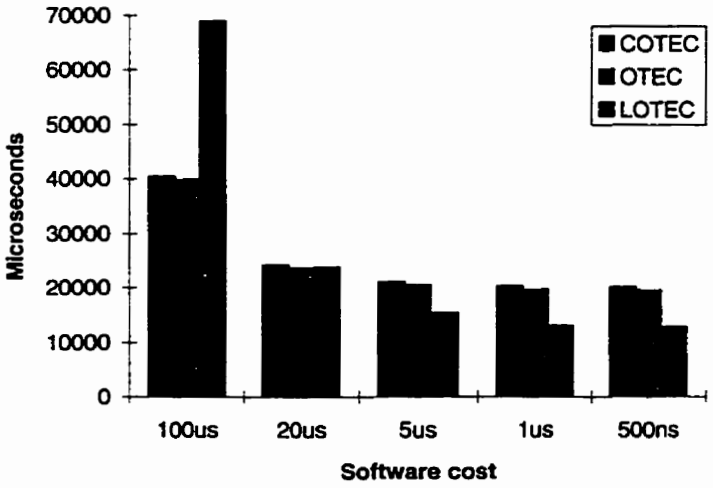


Figure 5.7: Message cost for O_{14} when transmission rate is 100Mbps

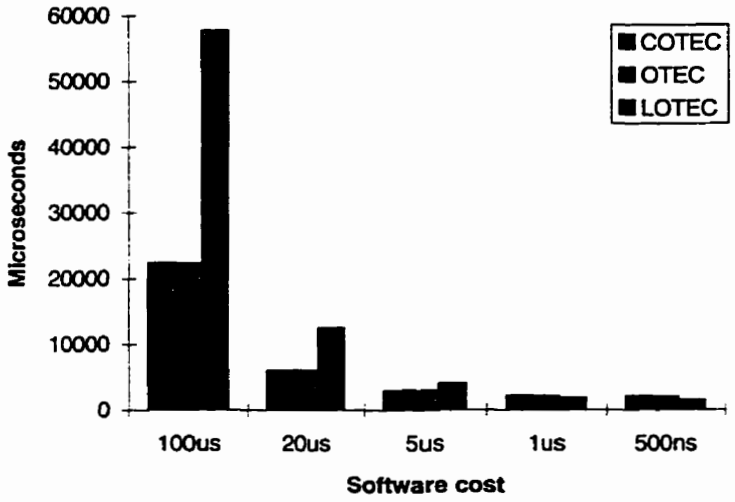


Figure 5.8: Message cost for O_{14} when transmission rate is 1Gbps

Chapter 6

Conclusions and Future Work

This thesis has presented two new DSVM consistency protocols, OTEC and LOTEC, for closed nested object transactions in a page based persistent object system. The performance of the protocols has been demonstrated via a simulated DSVM system.

6.1 Contributions

A model of nested object transactions has been introduced to suit method executions in a persistent object system. In the proposed DSVM system, multiple, concurrent users can access objects by invoking methods that manipulate their attributes. An access submitted by a DSVM user consists of a single object method invocation that may invoke other methods on other objects producing method invocations that are nested. Method executions, either by users or other objects, are therefore treated as closed nested atomic object transactions.

Moss' closed nested two-phase exclusive locking rules have been modified to define the closed nested *object* two-phase exclusive locking rules (O2PL). O2PL takes

into account the characteristics of a persistent object system, and adds additional functionality. It provides for correct concurrent executions of closed nested object transactions in the DSVM system.

A DSVM memory consistency protocol, OTEC, for closed nested object transactions in a page based persistent object system was also presented. It was defined by combining the closed nested object two-phase locking rules developed and the DSM memory consistency protocol, entry consistency. OTEC pulls all *updated* pages of a shared object across the network when the object is subsequently accessed by an acquiring [sub-]transaction at a remote node. To achieve better performance, a second DSVM consistency protocol that also supports closed nested object transactions was developed. LOTEC defers communication by pulling updated pages of a shared object across the network only when those pages may be *referenced* by an acquiring [sub-]transaction. In this way, LOTEC often transfers fewer pages than OTEC. Since there is no unique last updating node for a given shared object, LOTEC may need more short control messages than OTEC. Compared to transferring large updated object pages, the LOTEC protocol successfully reduces unnecessary data exchange.

A simple DSVM simulator has been developed to assess the network performance of the developed protocols under various load and sharing conditions. We evaluate the performance of OTEC, LOTEC,and a conservative baseline protocol known as COTEC by counting the total number of bytes of transferred messages (both short control messages and large object updates) for maintaining memory consistency in the simulated DSVM system. Overall, the results indicate that LOTEC has the best performance of the three protocols.

6.2 Future Work

The work presented in this thesis is a first step towards providing efficient memory consistency in the proposed DSVM system. There is still much work to be done in this area including the elimination of certain assumptions made in the dissertation and extensions of the work.

One simplifying assumption was that multiple pages would not be stored on a single page to avoid false sharing. Without this assumption, an exclusive-write protocol can handle the false sharing problem, but it may result in additional network traffic due to unnecessary communication. Modifications of certain multiple write-protocols [CBZ95, KCZ92] might be used to address this problem.

Another simplifying assumption was that all sub-transactions of a given root transaction will normally execute at the same node. This constraint can be relaxed to achieve greater parallelism. The lock structure for each object in the GDO may need to be changed to handle the lock operations for conflicting sub-transactions within a transaction family but which are executing on different nodes. Local lock release operations within a transaction family may also need to be handled globally to support the multi-stage release process across distributed nodes.

Our simulation results generally show that when objects span multiple pages LOTEC's performance improves. However, object-level locking is likely too coarse when objects span a large number of pages as it may prevent concurrency within the object. Additionally, since object-level locking is supported in this thesis, to avoid deadlocks in one transaction family, it was assumed that no directly or indirectly recursive invocations on the same object are allowed within a transaction family. Page-level locking can relax this constraint as long as no directly or indirectly recursive invocations on the methods that manipulate the same pages occur. It

also can enhance concurrency within the object. To facilitate page-level locking, Mathew, *et al.* [MGB95] suggest the use of a persistent global page table with one entry for each persistently stored page and a lock variable in each entry.

Consistency protocols presented in this thesis load pages individually on demand to keep communication overhead to a minimum. If a consistency protocol can predict which pages will be accessed by a method, communication latency can be hidden by prefetching and thereby overlapping communication with computation. This might be particularly useful in a system without support for low software overhead messaging.

Finally, only exclusive locks were supported in the thesis. Adding support for shared ("read") locks rather than just exclusive locks should be straight forward.

Appendix A

Detailed Simulation Results

Tables A.1 to A.12 present the raw data of simulation results for test cases 1 through 4 under the three consistency protocols, COTEC, OTEC, and LOTEC. Each table lists a set of randomly selected shared object identifiers, the number of pages transferred, the number of short control messages and long data messages required, and how many [sub-]transactions access the shared object concurrently.

Table A.1: Simulation results under COTEC for test case 1

OID_s	#transferred pages	#short messages	#long data messages	#[sub-]transactions
0	26	52	26	61
2	78	52	26	92
4	165	66	33	102
6	52	52	26	95
8	112	56	28	101
11	36	36	18	33
13	92	46	23	66
15	21	42	21	60
17	57	38	19	46
19	95	38	19	48

Table A.2: Simulation results under OTEC for test case 1

OID_s	#transferred pages	#short messages	#long data messages	#[sub-]transactions
0	26	52	26	61
2	78	52	26	92
4	165	66	33	102
6	52	52	26	95
8	110	56	28	101
11	34	36	18	33
13	85	46	23	66
15	21	42	21	60
17	57	38	19	46
19	95	38	19	48

Table A.3: Simulation results under LOTEC for test case 1

OID_s	#transferred pages	#short messages	#long data messages	#[sub-]transactions
0	26	52	26	61
2	74	78	39	92
4	129	118	59	102
6	52	54	27	95
8	85	86	43	101
11	23	42	21	33
13	63	56	28	66
15	21	42	21	60
17	44	60	30	46
19	66	68	34	48

Table A.4: Simulation results under COTEC for test case 2

OID_s	#transferred pages	#short messages	#long data messages	#[sub-]transactions
10	385	70	35	85
11	276	46	23	40
12	351	54	27	49
13	294	42	21	48
14	510	68	34	75
15	384	48	24	54
16	476	56	28	64
17	360	40	20	33
18	646	68	34	77
19	520	52	26	50

Table A.5: Simulation results under OTEC for test case 2

OID_s	#transferred pages	#short messages	#long data messages	#[sub-]transactions
10	384	70	35	85
11	264	46	23	40
12	347	54	27	49
13	271	42	21	48
14	495	68	34	75
15	355	48	24	54
16	469	56	28	64
17	328	40	20	33
18	618	68	34	77
19	506	52	26	50

Table A.6: Simulation results under LOTEC for test case 2

OID_s	#transferred pages	#short messages	#long data messages	#[sub-]transactions
10	256	160	80	85
11	141	116	58	40
12	204	122	61	49
13	170	100	50	48
14	316	188	94	75
15	211	86	43	54
16	326	152	76	64
17	178	110	55	33
18	420	210	105	77
19	272	164	82	50

Table A.7: Simulation results under COTEC for test case 3

OID_s	#transferred pages	#short messages	#long data messages	#[sub-]transactions
9	30	12	6	7
18	40	20	10	14
25	6	12	6	9
32	27	18	9	11
37	9	6	3	5
42	33	22	11	18
46	18	18	9	18
54	45	18	9	13
64	65	26	13	20
67	15	10	5	6
71	6	6	3	5
74	65	26	13	17
83	32	16	8	13
92	42	28	14	24
99	30	12	6	8

Table A.8: Simulation results under OTEC for test case 3

OID_s	#transferred pages	#short messages	#long data messages	#[sub-]transactions
9	21	12	6	7
18	38	20	10	14
25	6	12	6	9
32	24	18	9	11
37	9	6	3	5
42	32	22	11	18
46	18	18	9	18
54	45	18	9	13
64	61	26	13	20
67	15	10	5	6
71	5	6	3	5
74	62	26	13	17
83	25	16	8	13
92	41	28	14	24
99	27	12	6	8

Table A.9: Simulation results under LOTEC for test case 3

OID_s	#transferred pages	#short messages	#long data messages	#[sub-]transactions
9	15	18	9	7
18	27	24	12	14
25	6	12	6	9
32	19	24	12	11
37	6	6	3	5
42	26	30	15	18
46	18	18	9	18
54	27	30	15	13
64	38	44	22	20
67	15	12	6	6
71	3	6	3	5
74	50	38	19	17
83	11	18	9	13
92	32	42	21	24
99	16	22	11	8

Table A.10: Simulation results under COTEC for test case 4

OID_s	#transferred pages	#short messages	#long data messages	#[sub-]transactions
9	80	16	8	9
12	91	14	7	12
18	304	32	16	22
31	36	6	3	5
37	36	4	2	3
39	100	10	5	8
54	180	24	12	18
56	221	26	13	17
58	152	16	8	17
70	143	26	13	17
73	112	16	8	10
77	108	12	6	10
91	60	10	5	7
96	136	16	8	12
99	80	8	4	6

Table A.11: Simulation results under OTEC for test case 4

OID_s	#transferred pages	#short messages	#long data messages	#[sub-]transactions
9	67	16	8	9
12	78	14	7	12
18	251	32	16	22
31	21	6	3	5
37	20	4	2	3
39	92	10	5	8
54	167	24	12	18
56	203	26	13	17
58	143	16	8	17
70	130	26	13	17
73	94	16	8	10
77	103	12	6	10
91	56	10	5	7
96	123	16	8	12
99	68	8	4	6

Table A.12: Simulation results under LOTEC for test case 4

OID_s	#transferred pages	#short messages	#long data messages	#[sub-]transactions
9	28	20	10	9
12	43	28	14	12
18	85	72	36	22
31	8	8	4	5
37	2	4	2	3
39	47	20	10	8
54	97	54	27	18
56	121	62	31	17
58	105	34	17	17
70	55	62	31	17
73	44	30	15	10
77	61	30	15	10
91	39	20	10	7
96	82	40	20	12
99	26	10	5	6

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*, 26:360–365, November 1983.

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

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

[BNBMJZ91] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, School of Computer Science, Carnegie Mellon University, September 1991.

[BPG95] K. Barker, R. Peters, and G. Graham. Distributed Shared Virtual Memory DSVM for Interoperability of Heterogeneous Information Systems. In *OOPSLA Workshop on Object Interoperability*, 1995.

[BZS93] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway Distributed shared Memory System. In *Proceedings of the Spring COMPCON*, pages 528–537, February 1993.

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

[CLFL94] J.S. Chase, H.M. Levy, M.J. Feeley, and E.D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. *ACM Transaction on Computer Systems*, 12(4):271 - 307, November 1994.

[EBBV95] T.V. Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A User-level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Principles*, December 1995.

[ECGS92] T.V. Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256 - 266, May 1992.

[GB93] P. Graham and K. Barker. Distributed Object Base Implementation Using a Single Shared Address Space. In *Proceedings of the Mid-Continent Information Systems Conference*, pages 62 - 77, 1993.

[GBBZ93] P. Graham, K. Barker, S. Bhar, and M. Zapp. A Paged Distributed Shared Virtual Memory System Supporting Persistent Objects. Technical report, The University of Manitoba, 1993.

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

[HH91] T. Hadzilacos and V. Hadzilacos. Transaction Synchronisation in Object Bases. *Journal of Computer and Systems Science*, 43:2 - 24. 1991.

[JR92] John Rosenberg. Architecture and Operating System Support for Orthogonal Persistence. *USENIX Computing Systems*, 5(3):305–335. 1992.

[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.

[KDCZ94] P. Keleher, S. Dwarkadas, A.L. Cox, 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.

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

[KJE93] V. Khera, R.P. LaRowe Jr., and C.S. Ellis. An Architecture-independent Analysis of False Sharing. Technical Report DUKE-TR-1993-13, Department of Computer Science, Duke University, October 1993.

[KK93] A. Kemper and D. Kossmann. Adaptable Pointer Swizzling Strategies in Object Bases. In *Proceedings of the International Conference on Data Engineering*, pages 155–162, 1993.

[Lam79] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transaction on Computers*, C-28(9):690 -691, September 1979.

[LH89] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transaction on Computer Systems*, 17(4):321 - 359, November 1989.

[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.

[Lyn83] N. A. Lynch. Concurrency Control for Resilient Nested Transactions. *Proc. 2nd ACM SIGACT-SOGMOD Symp. on Prin. of Database Sys.*, pages 166–181, March 1983.

[MEZ93] Michael Edward Zapp. Concurrency Control in Object-Based System. Master's thesis, University of Manitoba, Computer Science, June 1993.

[MGB95] J. Mathew, P. Graham, and K. Barker. Object Directory Design Issues for a Distributed Shared Virtual Memory System Supporting Persistent Objects. Technical Report TR-95/04, The University of Manitoba, Department of Computer Science, July 1995.

[MGB96] J.A. Mathew, P. Graham, and K. Barker. Object Directory Design for a Fully Distributed Persistent Object System. *Object Oriented Database System Symposium of the Engineering Systems Design and Analysis Conference*, 2:75 - 88, July 1996.

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

[MS95] D.E. McDysan and D.L. Spohn. *ATM: Theory and Application.* McGraw Hill, 1995.

[OV91] M.T. Özsu and P. Valduriez. *Principles of Distributed Database Systems.* Englewood Cliffs, NJ:Prentice-Hall, 1991.

[PCG94] Peter C. J. Graham. *Applications of Static Analysis to Concurrency Control and Recovery in Objectbase Systems.* PhD thesis, University of Manitoba, Computer Science, September 1994.

[PGB97] R.J. Peters, P. Graham, and K.E. Barker. A Shared Environment to Support Multiple Advanced Application Systems. In *Proceedings of Workshop on Information Technologies and Systems (WITS' 97).* December 1997.

[PK95] Peter Keleher. *Lazy Release Consistency for Distributed Shared Memory.* PhD thesis, Rice University, January 1995.

[VBD+92] F. Vaughan, T. L. Basso, A. Dearle, C. Marlin, and C. Barter. Casper: a Cached Architecture Supporting Persistence. *USENIX Computing Systems*, 5(3):337–359, 1992.

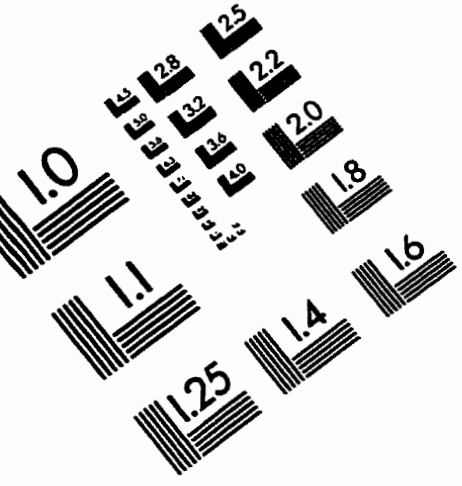
[VRH93] J. Vochteloo, S. Russell, and G. Heiser. Capability-Based Protection in a Persistent Global Virtual Memory System. Technical Report SCSE 9303, The University of New South Wales, School of Computer Science and Engineering, March 1993.

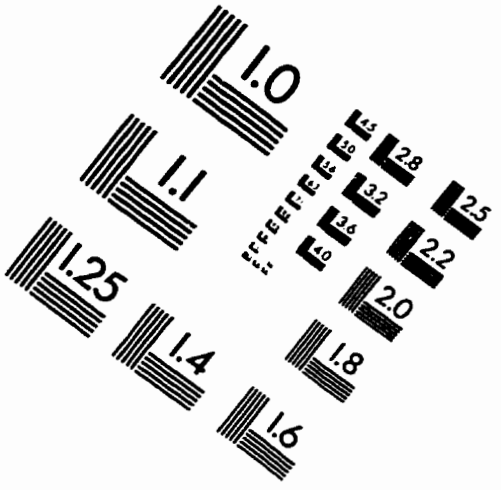
[WD92] S. J. White and D. J. Dewitt. A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies. In *Proceedings of the 18th International Conference on Very Large Data Bases*, October 1992.

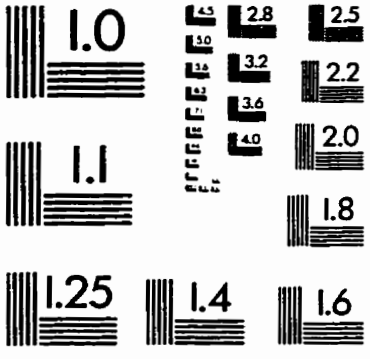
[Wei89] W.E. Weihl. *Theory of Nested Transactions.* ACM Press, 1989.

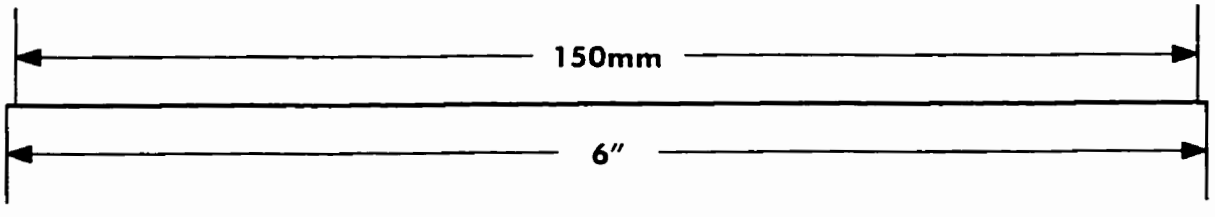
[Wil92] P. R. Wilson. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Addresses on Standard Hardware. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, pages 364–377, 1992.

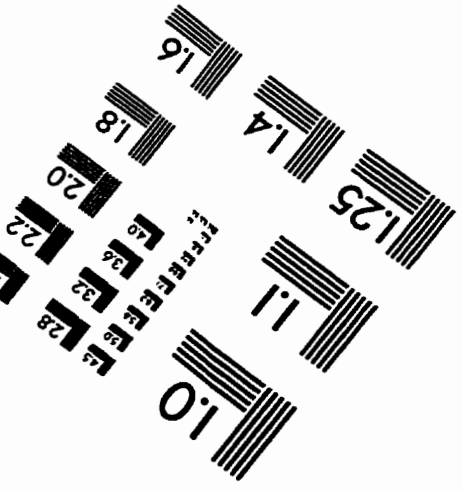
IMAGE EVALUATION
TEST TARGET (QA-3)

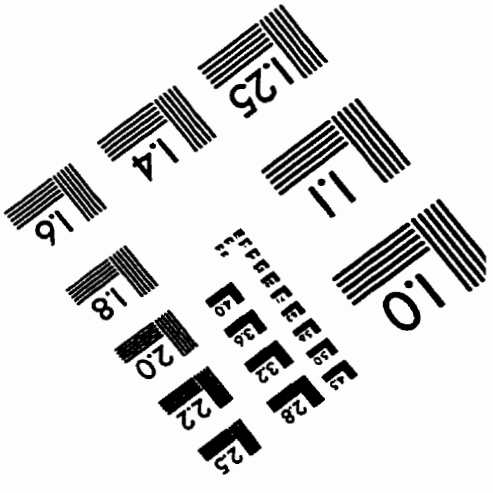












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989