CACHE CONSISTENCY AND CONCURRENCY CONTROL IN A CLIENT/SERVER DBMS ARCHITECTURE

by

Yongdong Wang and Lawrence A. Rowe

Memorandum No. UCB/ERL M90/120

19 December 1990

COVER PAGE

CACHE CONSISTENCY AND CONCURRENCY CONTROL IN A CLIENT/SERVER DBMS ARCHITECTURE

by

Yongdong Wang and Lawrence A. Rowe

Memorandum No. UCB/ERL M90/120

19 December 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

CACHE CONSISTENCY AND CONCURRENCY CONTROL IN A CLIENT/SERVER DBMS ARCHITECTURE

by

Yongdong Wang and Lawrence A. Rowe

Memorandum No. UCB/ERL M90/120

19 December 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture†

Yongdong Wang
Lawrence A. Rowe

Computer Science Division-EECS
University of California
Berkeley, CA 94720

Abstract

This paper examines five application cache consistency algorithms in a client/server database system: two-phase locking, certification, callback locking, no-wait locking, and no-wait locking with notification. A simulator was developed to compare the average transaction response time and server throughput for these algorithms under different workloads and system configurations. Two-phase locking and callback locking dominate no-wait locking and no-wait locking with notification when the server or the network is a bottleneck. Callback locking is better than two-phase locking when the inter-transaction locality is high or when inter-transaction locality is medium and the probability of object update is low. When there is no network delay and the server is very fast, no-wait locking with notification and callback locking dominate two-phase and no-wait locking.

1. Introduction

This paper presents the results of a simulation study of database concurrency control and application program cache consistency algorithms in a distributed computing system. This work was motivated by the recent development of persistent programming languages and object-oriented database systems [3-5, 7, 10, 13-18, 20], and client/server database architectures. In a client/server architecture, the database resides on the server. Objects in the database are accessed by application programs running on client workstations. Objects are cached in the application to reduce the time required to access an object. Consequently, several copies of a shared object can exist in more than one application cache at the same time. Mechanisms must be provided to guarantee that concurrent transactions accessing the cached objects and the database do not interfere.

Traditional database application development tools do not support application program caches [9]. Consequently, the application programmer must implement a cache

† This research was supported by NASA grant NAG 2-530 and NSF grant MIP-8715557.

consistency algorithm or objects must be re-fetched for each transaction. Persistent programming language systems provide an application cache to simplify application development. In addition, a cache can improve performance by eliminating object refetches and reducing the load on the server [21]. The efficiency of the application caching mechanisms is very important to the performance of the applications and the database system.

Concurrency control and cache consistency must be coordinated since caches are invalidated on transaction boundaries. Object updates become permanent when transactions commit. In this study, we extend concurrency control algorithms to include the consistency check of cached objects. We will use the terms *concurrency control* and *cache consistency* interchangeably.

Several researches have investigated the performance of database concurrency control algorithms. Agrawal, Carey, and Livny (ACL) compared the transaction throughput rate of three algorithms for centralized database systems: (1) two-phase locking, (2) immediate-restart (wound-wait), and (3) certification (optimistic concurrency control)† [1, 2]. Two-phase locking outperformed the immediate-restart and certification algorithms for medium to high levels of resource utilization.

Carey and Livny investigated distributed concurrency control algorithms [8]. Four algorithms were studied: (1) distributed two-phase locking, (2) wound-wait, (3) basic timestamp ordering, and (4) distributed certification. Again, transaction throughput rate was compared under different degrees of contention, data replication, and message cost. Two-phase locking and certification dominated timestamp ordering and wound-wait. When the CPU cost of sending and receiving messages was low, two-phase locking was superior because it reduced transaction restarts. However, when message cost was high and data was replicated, certification outperformed two-phase locking.

Wilkinson and Neimat studied application cache consistency algorithms in a client/server database system [21]. They proposed two algorithms: (1) cache locking and (2) notify locking. They showed that for both short batch and interactive transactions cache locking is never worse than two-phase locking without caching. Furthermore, notify locking is better than cache locking when server CPU utilization is not high, but it is worse than two-phase locking without caching when the server CPU is saturated. However, they did not study other algorithms such as two-phase locking with caching. And, they did not measure transaction response time which we believe is a very important metric for applications. Lastly, their simulation model did not address the cache replacement problem.

None of the previous simulation models had an explicit buffer manager [2, 8, 21]. We believe that the addition of a buffer manager in the simulator can make a difference

† For a complete description of these algorithms and the other algorithms discussed below, see the book by Bernstein [6].

in several ways:

(1) Realistically, objects may have to be written out to disk prior to transaction commits which could cause I/O contention. A buffer manager can model this situation.

(2) The buffer manager can model hot spots better. Without a buffer manager, if several transactions read the same object, each of them is charged with the disk I/O for the object. On the other hand, only one transaction is charged with the I/O with a buffer manager.

(3) When a transaction commits, updated objects do not have to be written out as long as their logs are forced out. If the same object is updated by other transactions, it would not be written out twice. Other models charge a disk I/O for every committed object update.

(4) When a transaction restarts, objects in the buffer can be read directly. In models without a buffer manager, a restarted transaction has to read its entire read set from the disk again. Consequently, strategies that abort transactions more often are penalized because they are charged with more disk I/O's.

In this paper, we distinguish between caching within a transaction (intra-transaction) and between transactions (inter-transaction). The performance of five inter-transaction algorithms are compared: (1) two-phase locking, (2) certification, (3) callback locking, (4) no-wait locking, and (5) no-wait locking with notification. A simulator similar to the one used by ACL was used to compare average response time and throughput under different workloads and system configurations. Three workloads were included in our experiments: small batch transactions, large batch transactions, and interactive transactions. System configurations included a system where the server was a bottleneck, a fast server system, and a fast server and fast network system.

The remainder of the paper is organized as follows. Section 2 describes the algorithms that we studied. Section 3 describes the structure of our simulation model and the simulation parameters. Section 4 describes two experiments that verify the correctness of our simulator. Section 5 describes the results of our initial performance experiments. Finally, section 6 summarizes our findings.

2. Cache Consistency Algorithms

This section describes the algorithms included in our study. The algorithms are extensions or variations of existing concurrency control algorithms for centralized database systems.

Two-phase locking was included in our study because numerous studies conclude that it outperforms all other algorithms unless unrealistic assumptions are made (e.g., infinite physical resources) [2]. Certification was included because it is used by at least one object-oriented DBMS that has a client/server architecture (GemStone [7]). Moreover, some recently proposed algorithms (e.g., cache locking) incorporate ideas from the

certification algorithm. Other concurrency control algorithms (e.g., time-stamp ordering) were not included because we did not believe they would be better than two-phase locking or certification.

We assume that all locking algorithms use *in-place updates* and that certification algorithms use *deferred updates*. With in-place updates, a transaction updates the objects in the database directly. If the transaction is later aborted, the updates are rolled back (*undone*). With deferred updates, changes that a transaction makes are placed in a private buffer until it has been certified that the transaction can commit. At that point, the updates are merged into the database. We also assume that an object must be fetched to the client before it is updated. In other words, all updates are executed on the clients first. Updates are sent to the server either when an updated object is swapped out of the client cache or at transaction commit time. Transactions within a client are executed sequentially. There is at most one active transaction, called the *current transaction*, in a client at any time.

There are two models of object caching: caching within a transaction, called *intra-transaction caching*, and caching between transactions, called *inter-transaction caching*. In the case of intra-transaction caching, a client assumes that objects in its cache are invalid at the beginning of each transaction. Objects are fetched into the cache when they are first accessed by the transaction. Intra-transaction caching algorithms are easy to implement, but they cannot benefit from inter-transaction reference locality. In the case of inter-transaction caching, a client must check that objects in the cache are valid when they are accessed. This check can be accomplished in two ways: (1) the client can contact the server to verify object validity when the object is accessed (*check-on-access*) or (2) the server can notify clients whenever an object in the cache is updated by other clients (*notify-on-update*).

The two-phase locking and certification algorithms can be easily extended to support both intra- and inter-transaction caching. In the rest of this section, four algorithms that support inter-transaction caching are described: (1) two-phase locking; (2) certification; (3) callback locking; and (4) no-wait locking. A potential problem with no-wait locking is the high transaction restart rate. Update notification can be incorporated with no-wait locking to reduce transaction restarts. Thus, we also include a fifth algorithm, (5) no-wait locking with notification.

2.1. Two-Phase Locking

Inter-transaction caching attempts to use objects left in a cache by a previous transaction. Two-phase locking uses a check-on-access strategy.

We assume that each object is tagged with a version number. The version number can be the time the object is updated, the identifier of the transaction that updated it, or some other number assigned by the server. When an object is cached in a client, its version number is cached with it.

With two-phase locking, in the case of intra-transaction caching, all cached objects are locked by the current transaction and are therefore valid. But with inter-transaction caching, only those cached objects that are locked by the current transaction are guaranteed to be valid. When a transaction accesses a cached object that is not locked, it must query the server to determine if the object is valid. However, since it has to ask the server to set a lock on the object anyway, it does not need to send an extra message to check validity. Therefore, there is little additional overhead.

2.2. Certification

Certification does not block a transaction when it reads or writes an object. It checks at commit time that the referenced objects have not been modified by other transactions that have committed since this transaction started. Certification also uses a check-on-access strategy. A transaction checks with the server when it accesses a cached object for the first time. The client remembers which cached objects have been checked by the current transaction.

2.3. Callback Locking

Callback locking was first used in the Andrew File System to maintain the consistency of cached files [12]. It guarantees the validity of cached objects by retaining locks on them even after a transaction terminates. Therefore, there is no need for the client to contact the server to check object validity or to acquire a lock when a transaction accesses a cached object with the appropriate lock. However, if a transaction accesses a cached object without a retained lock or with the wrong lock (e.g., the transaction wants to update an object that has only a read lock), it will need to get the lock from the server. The server sends a message to all clients that have uncompatible locks on this object requesting them to return the locks. A client releases the lock requested by the server immediately if the object has not been accessed by the current transaction on the client. Otherwise, it waits until the current transaction terminates to release the lock. The server cannot grant the requested lock until all uncompatible locks on the object are released.

Both read and write locks can be retained. However, write locks are more likely to cause incompatibility, and thus more likely to be retrieved by the server before it is used by the next transaction. Consequently, we chose to retain only the read locks.

2.4. No-Wait Locking

In two-phase locking, the client application is blocked while waiting for a response from the server that a cached object is valid. An alternative is to assume that all objects in the cache are valid. The application program continues executing without being blocked when accessing cached objects. If the cached object is valid and the requested lock can be granted, the server does not send any response to the client. If the cached object is invalid or the requested lock cannot be granted because of a deadlock, the server aborts the transaction and the client must restart it. Note that the client must receive a

response from the server before it can commit because it needs to get locks even if all objects are valid.

This algorithm was originally proposed by Gerson for Statice [11, 20]. He called it *optimistic locking* because of the optimistic assumption that cached objects are valid and the requested locks can be granted.

2.5. No-Wait Locking with Notification

In no-wait locking, a transaction can be aborted because it has read an invalid object or it is involved in a deadlock. While nothing can be done about transaction aborts due to deadlocks, notification can be integrated into no-wait locking to reduce the transaction aborts caused by reading invalid cached objects. This is achieved by making the server send an invalidation message to clients when a cached object is updated by a committed transaction. Client transactions will read valid objects from the server at a later time and subsequently commit. Please note that notification cannot eliminate reading invalid objects because a transaction can still read an invalid object before it receives the invalidation message from the server.

Instead of invalidating the cached objects, the server can also send the updated version to the clients. Thus, the clients do not have to read the object from the server at a later time. However, if the cached object is not accessed before being swapped out of the cache, the updates sent to the client are a waste. In our study, we chose to send the updates to the clients after a transaction commits.

3. Simulation Model

This section describes our simulation model. It is based on the ACL Model for a centralized DBMS [1, 2] and the Carey and Livny model for a distributed DBMS [8]. It is similar to the client/server model developed by Wilkinson and Neimat [21].

Our model has important extensions to and modifications of the ACL Model. First, it models a client/server DBMS, which is different from a centralized or distributed DBMS. Second, it has client cache managers and a buffer manager in the server. Third, it models subobjects shared by multiple complex objects and object clustering. Fourth, it supports an arbitrary number of interleaved object reads and updates. And fifth, it models inter-transaction object reference locality.

The Wilkinson and Neimat model simulates a client/server DBMS, but it does not have a cache or buffer manager, nor does it model object sharing or clustering. It models object reference locality between transactions by varying the client cache hit ratio.

There are three main parts to the simulation model: the database model, the transaction model, and the system model. The database model captures the characteristics of the database, such as the database size and the object structures. The transaction model captures the object reference behavior of transactions in a workload. And, the system model captures the characteristics of the system's hardware and software. The physical structure of the modeled system is shown in Figure 1. It is composed of a database server and n

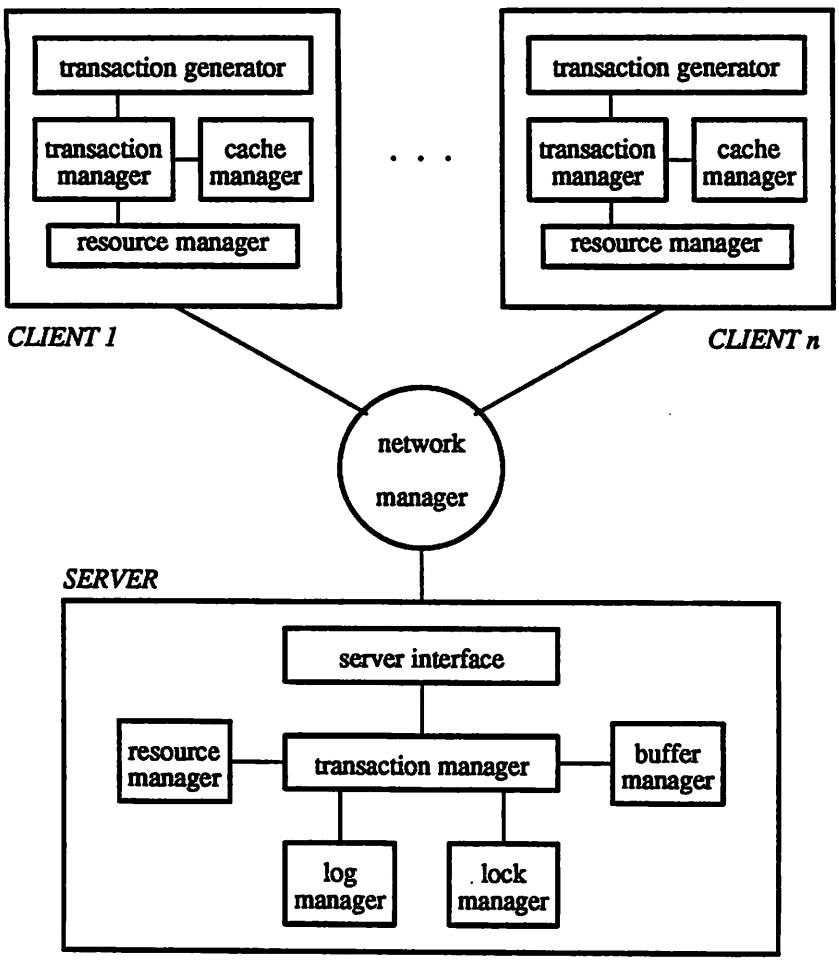


Figure 1. Client/Server DBMS Structure

clients connected by a network. We assume that there is only one application on each client workstation. Thus, the term *client* refers to both the application and the workstation.

The simulator is implemented in the simulation language CSIM, which is a C-based simulation language developed at MCC [19]. The core of the simulator consists of about 4000 lines of CSIM code. Each cache consistency algorithm adds an additional 500 to 1000 lines of code, depending on the complexity of the algorithm.

3.1. Database Model

A database is composed of several *classes* (or *relations*). A class is a collection of *objects* (or *tuples*) that have the same attributes. Logically, each class is a sequence of

atoms which is the minimum unit that can be shared by different objects. Each object contains one or more atoms. The *size* of an object is the number of atoms contained in that object. We assume that all objects in a class are of the same size. To model subobject sharing, we assume that each object of class *c* starts at a random atom within that class with equal probability, and contains that and the next *s-1* atoms where *s* is the object size. With this scheme, multiple objects can share atoms, as illustrated in Figure 2, where objects *O1* and *O2* share the atoms *Ai* through *Aj*. This design allows us to model different degrees of object sharing by varying the number of atoms and the object size in a class.

The database parameters are summarized in Table 1. *NClasses* is the number of classes in the database. *NPages* and *ObjectSize* are the number of atoms and the size of objects in each class, respectively. *ClusterFactor* is the probability that consecutive atoms in an object are stored sequentially on the disk. For simplicity, we assume that each atom corresponds to a disk page. We assume that the unit of cache consistency operations and the unit of data transportation between clients and the server is also a disk page. Consequently, the assumption that an atom is a page does not affect the results of

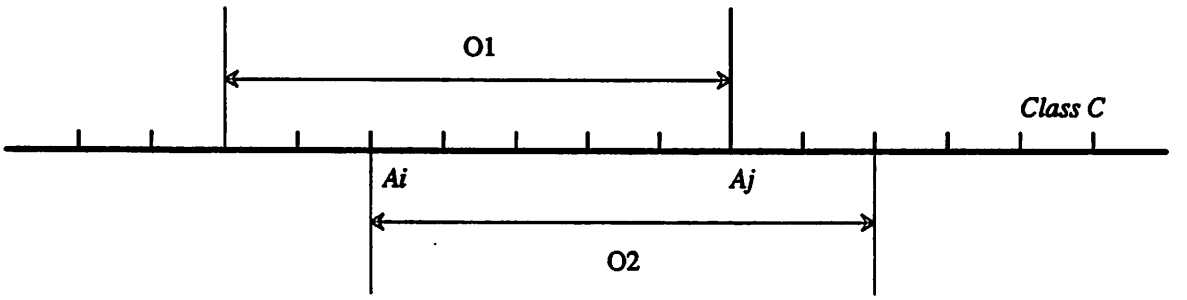


Figure 2. Two Objects Share Atoms

Parameter	Description
NClasses	Number of classes in the database
NPages[i]	Number of atoms (pages) in class *i*
ObjectSize[i]	Size of the objects in class *i*
ClusterFactor	Probability that atoms in an object are clustered together

Table 1. Database Parameters

our study.

Atoms are only shared by objects from the same class. Navigation between objects of different classes is represented in the transaction model described in the next section.

3.2. Transaction Model

The transaction model supports the following operations:

(1) BeginXact: start a new transaction,

(2) ReadObject: read an object into the client cache,

(3) UpdateObject: update an object in the client cache,

(4) UserDelay: delay by the application,

(5) CommitXact: commit a transaction,

(6) AbortXact: abort a transaction.

The operation *UserDelay* is included to model non-database processing time by the application and interactive applications.

A transaction is modeled by a finite loop of *ReadObject* and *UpdateObject* operations, as shown in Figure 3. Different types of transactions are modeled by varying the average number of *ReadObject* operations in a transaction, the read/write ratio (i.e., the ratio of the number of objects modified vs. the number of objects read), whether transactions are executed interactively or in batches, and so forth. A simulation run can simulate transactions belonging to the same type, or a mix of transactions belonging to different types. Table 2 summarizes parameters that characterize a transaction type.

The number of *ReadObject* operations in a transaction is called the *transaction size*. Transaction size is uniformly distributed between *MinXactSize* and *MaxXactSize*. The *UpdateObject* operation updates atoms of the object read by the preceding *ReadObject* operation. The parameter *ProbWrite* is the probability that each atom of the object is

```
BeginXact
        dotimes (transaction_size)
                ReadObject
                UserDelay (UpdateDelay)
                UpdateObject
                UserDelay (InternalDelay)
        end dotimes
CommitXact
```

Figure 3. A General Transaction Model

Parameter	Description
MinXactSize	Minimum number of ReadObject operations in a transaction
MaxXactSize	Maximum number of ReadObject operations in a transaction
ProbWrite	Probability that a page in an object is updated
UpdateDelay	Average think time between a ReadObject and an UpdateObject
InternalDelay	Average think time for each pass of the loop in a transaction
ExternalDelay	Average think time between two transactions
InterXactSetSize	Number of objects in InterXactSet
InterXactLoc	Probability that an object read belongs to InterXactSet

Table 2. Parameters for A Transaction Type

updated.† The delay parameters (i.e., *UpdateDelay*, *InternalDelay*, and *ExternalDelay*) are exponentially distributed delay times which are introduced to model interactive transactions. They can be set to *0* to model batch transactions.

To model inter-transaction object reference locality, we introduce a new concept, *InterXactSet*, which contains the last x objects read by the most recent transactions where x is the value of the parameter *InterXactSetSize*. The parameter *InterXactLoc* specifies the probability that an object read by the current transaction is in the *InterXactSet*. The larger the *InterXactLoc* is, the more overlap the read sets of consecutive transactions have, and the higher the inter-transaction object reference locality. This type of locality can be called *temporal locality* because the same objects are referenced repeatedly at times close to each other. We chose *InterXactSetSize* as a simulation parameter so that we can adjust it according to the size of the client cache to make sure that objects in the *InterXactSet* can almost always be found in the cache.

3.3. System Model

The system model describes the network manager, the resource manager, and the other client and server modules. The parameters for all the modules are summaried in Table 3.

3.3.1. Network Manager

The network manager models communication among clients and between clients and the server. Messages are assumed to be transferred in packets, with each packet incurring certain CPU overhead at both the sending and receiving sites and a certain

† This implies that the write set of a transaction is always a subset of its read set.

Parameter	Description
NetDelay	Average message delay time on the network
PacketSize	Maximum number of bytes in a packet
MsgCost	CPU cost of sending/receiving a message packet
NClients	Number of clients
NClientCPUs	Number of CPU's on a client
ClientMips	Speed of each client CPU
NServerCPUs	Number of CPU's on the server
ServerMips	Speed of each server CPU
NDataDisks	Number of data disks on the server
NLogDisks	Number of log disks on the server
CacheSize	Number of pages in a client cache
BufferSize	Number of pages in the server buffer pool
SeekLow	Minimal disk seek time
SeekHigh	Maximum disk seek time
DiskTran	Transfer time for one disk block
PageSize	Disk block size
InitDiskCost	CPU cost of initiating a disk access
ServerProcPage	CPU cost of processing a page on the server
ClientProcPage	CPU cost of processing a page on the client
MPL	Maximum number of active transactions allowed on the server

Table 3. System Parameters

delay on the network. Large messages are broken into packets and transferred. *NetDelay* is the mean of an exponentially distributed network delay time for packets. *PacketSize* is the maximum number of bytes allowed in a message body. And *MsgCost* is the number of instructions executed to send or receive a message on a client or server. The network manager uses a first come, first served (FCFS) policy.

3.3.2. Resource Manager

The resource managers on the clients and the server manage physical resources (e.g., CPU's and disks). *NClients* is the number of client workstations. We assume that all clients are uniform (i.e., they have the same amount of physical resources). *NClientCPUs* is the number of processors on each client, and *ClientMips* is the speed of each processor. *NServerCPUs* and *ServerMips* are the corresponding parameters for the server. We assume that all clients are diskless so objects reside in the main memory of clients. The server has a number of data disks and log disks, all with the same physical characteristics. We separate disk seek time (including rotation time) and data transfer

time so we can model sequential disk accesses. Disk seek time is uniformly distributed between *SeekLow* and *SeekHigh*. We assume that classes are uniformly distributed to the data disks. All objects in one class reside on the same disk.

PageSize is the size of a disk block. For convenience, we assume that it is also the size of memory pages. *InitDiskCost* is the number of instructions executed to initiate a disk access. *ServerProcPage* is the number of instructions executed on the server to process each page read or update by user transactions. And *ClientProcPage* is the number of instructions executed on the client to process each page read or update. We assume that all pages incur the same CPU cost.

Different resource allocation policies can be implemented. The disks use an FCFS policy. CPUs also use an FCFS policy by default. We assume that the server and client resources described above are used by the DBMS exclusively. In other words, we do not consider other processes running concurrently with the DBMS.

3.3.3. The Client

The client transaction generator generates user transactions. The client cache manager manages the client cache. It implements object lookup, object replacement on a cache miss, and flushing dirty objects on transaction commits. An LRU cache replacement policy is used (i.e., on a cache miss, the least recently used pages are replaced from the cache to make room for the new object). For intra-transaction caching algorithms, the entire cache is invalidated on transaction boundaries. For inter-transaction caching algorithms, objects are kept in the cache after a transaction terminates, and they are verified when they are accessed. The verification protocol is algorithm dependent and is implemented in the transaction manager.

If the cache is full on a cache miss, some objects are replaced from the cache. When this happens, a function in the algorithm dependent transaction manager is called. The action taken depends on the specific cache consistency algorithm. In the simplest case (e.g., two-phase locking), an object that has not been updated (i.e., a clean object) replaced from the cache is ignored. While in the callback locking algorithm, the server has to be notified when a clean object with a lock is replaced. In all algorithms, a modified object that is replaced must be sent to the server. The cache hit ratio is determined by the database size, the client cache size, and the object access pattern.

The client transaction manager is responsible for executing user transactions generated by the transaction generator. If a transaction commits, it returns to the transaction generator. If a transaction aborts, it restarts the same transaction again and again until it finally commits. Whether there is a restart delay between the time when a transaction aborts and the time when a transaction restarts depends on the particular algorithm.

The transaction manager is also responsible for communication with the server. It forms messages and sends them to the server. It also processes all messages from the server. For example, in the callback locking algorithm, the transaction manager determines whether a lock can be released when requested to do so by the server. The client

protocols of the cache consistency algorithms described in Section 2 are implemented by the client transaction manager. This module is the only client module that is algorithm dependent.

3.3.4. The Server

The buffer manager is responsible for managing the buffer pool on the server. It implements an LRU replacement policy, and holds the most recently referenced objects by transactions from all clients.

The transaction manager models the execution of transactions on the server. It implements the server protocol of the different cache consistency algorithms described in Section 2. This module is the only server module that is algorithm dependent.

The lock manager implements locking for the lock-based algorithms. Locks are usually released when a transaction terminates (except in the case of callback locking, which retains locks even after transactions terminate). In our implementation, all locks are set at the page granularity. We do not model locks on multiple granularities.

The log manager implements a log-based recovery scheme. We assume that the log is written to disks dedicated to that use. The log manager is provided to model transaction behavior more accurately. Since we allow uncommitted updates to be replaced from the buffer pool, transaction aborts involve processing the log and undoing the updates forced out to disk. Thus, protocols that cause more transaction aborts are charged for them. In previous models, transaction aborts are essentially free.

The parameter *MPL* is the multiple programming level which is the maximum number of active transactions allowed on the server. It is varied to model server contention.

3.4. Logical Structure of the Simulator

Figure 4 shows the logical queuing model of our simulator. We will use this logical model to explain how a transaction is executed by the simulator.

A transaction originates at the application generator. Operations are entered into the operations queue. An operation from the queue goes to the client transaction module (CTM). If it is an object access (read or write) operation, and the object is in the cache and the appropriate access permission (e.g., locks) have already been granted, the read and update operations are performed. Otherwise, a message is sent to server to fetch the object or get access permission. The client may or may not wait for a response, depending on the particular cache consistency algorithm.

A successful *ReadObject* operation is followed by an update delay, and a successful *UpdateObject* is followed by an internal delay. A transaction restart is followed by an optional restart delay, depending on the algorithm. Then the same operations of the aborted transaction are entered into the operations queue again. We follow the ACL model to use the average transaction response time as the mean of an exponentially distributed restart delay time. A committed transaction is followed by an external think

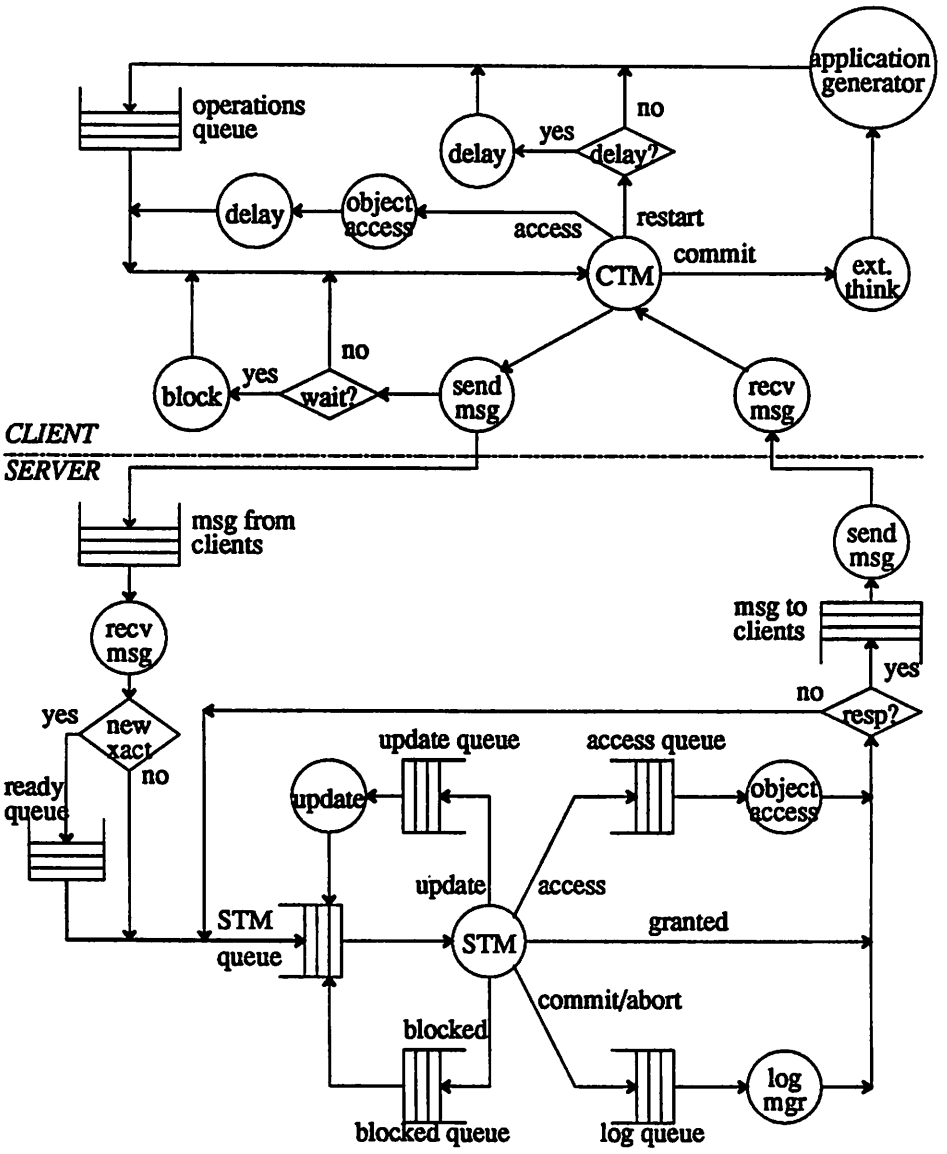


Figure 4. Logical Queuing Model

time before the next transaction is originated.

When a new transaction arrives at the server, if the server has already reached the limit of active transactions, the new transaction is placed in the ready queue. It will be activated when an active transaction commits or aborts. The message then enters the server transaction module (STM) queue. If the cache consistency request cannot be granted immediately, the transaction is blocked. When it is awakened later, it enters the

STM queue again. If a simple request (e.g., getting a lock on a particular object) is granted, it returns immediately. If an object access is requested by the message, the object is accessed directly if it is in the buffer pool. Otherwise, the request is placed on the object access queue. Commit/Abort requests are placed on the log queue where the log is processed. If a transaction is committed, and deferred updates are used, the updates are entered into the update queue where the updates are performed. Finally, if a message to the client is necessary, a message is entered into the outgoing message queue.

On a client, the CPU cost for reading and updating an object is charged after the access permission is granted. On the server, the CPU cost for reading an object is charged for each object sent to a client. The CPU cost for updating an object is charged for each updated object received from the clients. The CPU costs for sending and receiving messages are charged when a message is sent and received.

4. Simulator Verification Experiments

This section describes two experiments that we performed to verify the simulator. In the first experiment, the simulation parameters were set to those of the ACL experiments and two-phase locking and certification algorithms were simulated [2]. Since our simulator was designed for a client/server DBMS architecture, we had to make a few changes in order to compare our results with their results which were obtained on a centralized DBMS. Table 4 shows the values of the related simulation parameters for this experiment. The database had two classes, each with 500 pages, which yielded a combined database size of 1000 pages. One class resided on each data disk. Since each object had equal probability of being accessed, object I/O's were distributed evenly to the two disks. *CacheSize* was set to 12 pages, which was the *MaxXactSize*, so that updates were not

Parameter	Value	Parameter	Value
NClasses	2	*NServerCPUs*	1
NPages[i]	500	*CacheSize*	12 pages
ObjectSize[i]	1	*BufferSize*	1 page
MinXactSize	4	*SeekLow*	35 milliseconds
MaxXactSize	12	*SeekHigh*	35 milliseconds
ProbWrite	0.25	*ServerProcPage*	15,000 instructions
ExternalDelay	1 second	*NDataDisks*	2
NClients	200	*MPL*	5, 10, 25, 50, 75, 100, and 200
ServerMips	1		

Table 4. Parameters for the ACL Comparison Experiment

flushed to the server until transaction commit time, which simulated deferred updates for both algorithms. *BufferSize* was set to 1, thus forcing all dirty pages to disk at commit time. The log manager was disabled. Other irrelevant parameters (e.g., *NetDelay, MsgCost, InterXactLoc*, etc.) were set to 0. We measured the transaction throughput rate, as ACL did. The transactions executed were short transactions that read an average of 8 pages, and there were no internal delays within a transaction. We found that two-phase locking dominated certification which matches the limited resource case (i.e., 1 CPU and 2 data disks) reported by ACL.

In the second experiment, we compared the performance of intra- and inter-transaction caching algorithms for both two-phase locking and certification. This experiment was performed for two reasons. First, we wanted to check that inter-transaction caching algorithms performed better when inter-transaction locality was high. Second, since Wilkinson and Neimat only compared their algorithms against two-phase locking with intra-transaction caching, we wanted to find out the magnitude of difference between the intra- and inter-transaction caching algorithms to determine the effect of their assumption.

Parameter	Value	Parameter	Value
NClasses	40	*NClientCPUs*	1
NPages[i]	50	*ClientMips*	1
ObjectSize[i]	1	*NServerCPUs*	1
ClusterFactor	1.0	*ServerMips*	2
MinXactSize	4	*NDataDisks*	2
MaxXactSize	12	*NLogDisks*	1
ProbWrite	0.0, 0.2, and 0.5	*CacheSize*	100 pages
UpdateDelay	0	*BufferSize*	400 pages
InternalDelay	0	*SeekLow*	0 milliseconds
ExternalDelay	1 second	*SeekHigh*	44 milliseconds
InterXactSetSize	20	*DiskTran*	2 milliseconds
InterXactLoc	0.05, 0.25, 0.50, and 0.75	*PageSize*	4096 bytes
NetDelay	2 milliseconds	*InitDiskCost*	5,000 instructions
PacketSize	4096 bytes	*ServerProcPage*	10,000 instructions
MsgCost	5,000 instructions	*ClientProcPage*	20,000 instructions
NClients	2, 10, 30, and 50	*MPL*	50

Table 5. Simulation Parameter Setting

Table 5 shows the values of the simulation parameters used for this experiment and the experiments reported in the next section unless noted otherwise. The database has 40 classes, each with 50 pages. Since each page is 4K bytes, the database has 8M bytes of data.† All objects contained a single page.‡ The transactions executed were short transaction reading an average of 8 objects. There was no internal delay within transactions. The server had a 400 page buffer pool and each client had a 100 page cache.

The parameters varied were: *NClients*, *ProbWrite*, and *InterXactLoc*. *NClients* determined the server resource contention rate. *ProbWrite* and *InterXactLoc* modeled the user workload. *ProbWrite* determined the database object contention rate. *InterXactLoc* determined the inter-transaction reference locality, called locality hereafter, for consecutive transactions. Response times and throughput results are given in seconds and transactions committed per second, respectively.

Figures 5(a) and 5(b) show the average transaction response time with low locality (*InterXactLoc* set to 0.05) and different write probabilities. When the number of clients is small (i.e., no more than 10), there is little difference between the algorithms regardless of the write probability. As shown in Figure 5(a), with low write probability, there is virtually no difference between the algorithms. But, when the object update probability is high (*ProbWrite* set to 0.5), as shown in Figures 5(b), certification algorithms perform poorly compared to two-phase locking algorithms as the number of clients increases. This is because certification causes more transaction aborts which waste valuable server resources. In fact, the server is saturated so these aborts are very costly. The inter-transaction algorithms perform about the same as intra-transaction algorithms because caching objects between transactions gives them no advantage due to the lack of locality.

Figures 6(a) and 6(b) show the average transaction response time with high locality (*InterXactLoc* set to 0.5) and different write probabilities. The inter-transaction caching algorithms are clearly better than their corresponding intra-transaction algorithms regardless of the write probability or the number of clients. The largest difference (about 30%) occurs when the write probability is zero because cached objects are always valid. When the object update probability is non-zero, the difference is reduced because cached objects are more likely to be invalid and thus they may have to be refetched from the server. For two-phase locking, the difference is about 12% when *ProbWrite* is 0.5. Again, as shown in Figure 6(b), the certification algorithms perform worse than the two-phase locking algorithms because of the high transaction abort rate.

Figures 7(a) and 7(b) show the corresponding transaction throughput rates for Figures 6(a) and 6(b). The relative performance of the algorithms are the same as shown in

---

† This is a small database. However, the results do not depend on database size rather on the degree of resource and data contention. A larger database will allow more clients for the same degree of data contention.

‡ We did not study the impact of large objects or object clustering in our initial experiments.

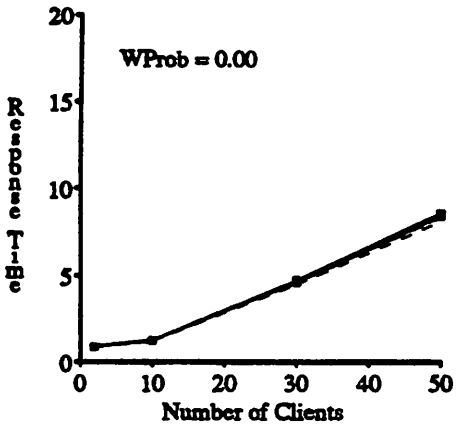


Figure 5(a). Response Time (Loc = 0.05)

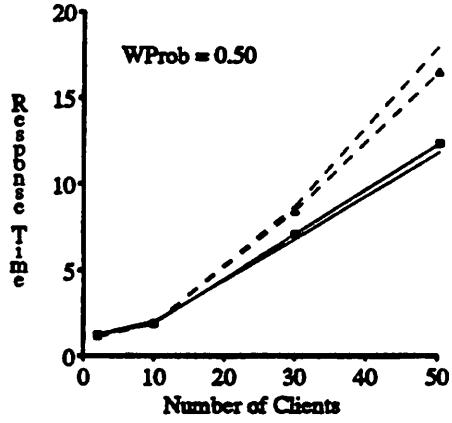


Figure 5(b). Response Time (Loc = 0.05)

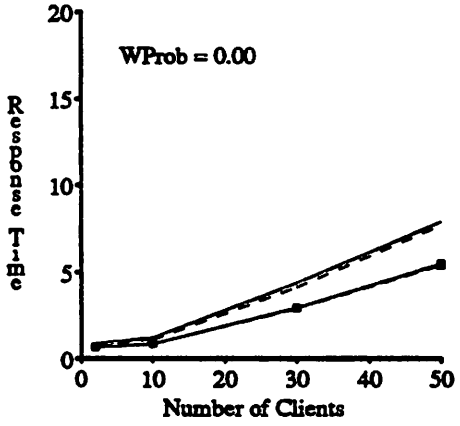


Figure 6(a). Response Time (Loc = 0.50)

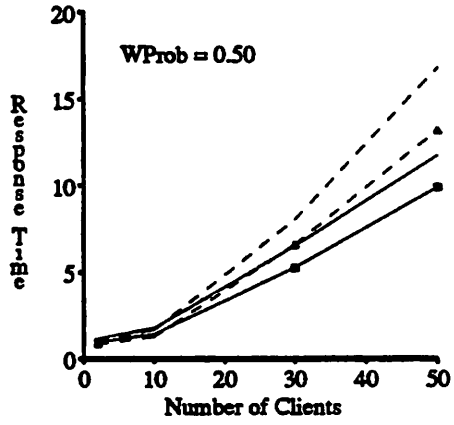


Figure 6(b). Response Time (Loc = 0.50)

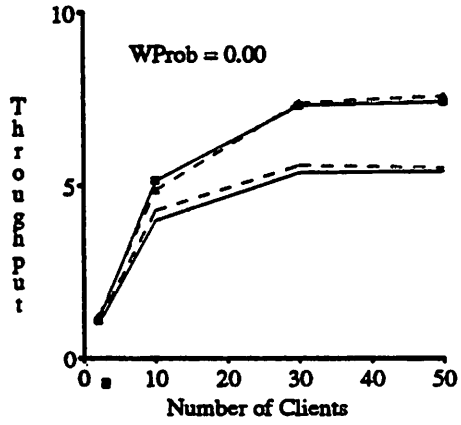


Figure 7(a). Throughput (Loc = 0.50)

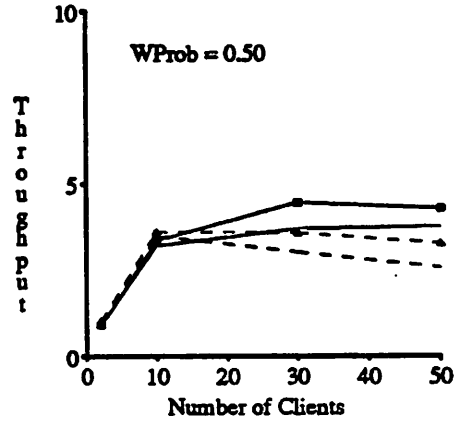


Figure 7(b). Throughput (Loc = 0.50)

Figures 6(a) and 6(b). In both figures, throughput first goes up as the number of clients increases because the server does not have enough work to do when there are a small number of clients, then the curve becomes flat as the server is saturated. When there are object updates, there will be more transaction aborts with a larger number of clients, thus the throughput rate begins to decline when the number of clients increases further.

To summarize, there is very little difference between algorithms when the locality and write probability are low. Inter-transaction algorithms dominate the corresponding intra-transaction algorithms when locality is high. Two-phase locking algorithms dominate the certification algorithms when the write probability is high and there are a large number of clients. These results match our expectations.

Wilkinson and Neimat compared other algorithms against two-phase locking with intra-transaction caching. Because two-phase locking with inter-transaction caching is 12% to 30% better when locality is high, we will use two-phase locking with inter-transaction caching, called two-phase locking hereafter, as a benchmark against which other algorithms are compared.

5. Experiments and Results

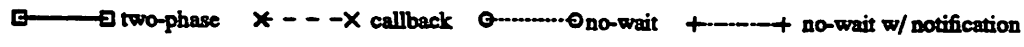
This section describes a set of experiments that summarize our initial findings. We compared the performance of callback locking, no-wait locking, and no-wait locking with notification against that of two-phase locking under the following conditions: (1) when small transactions were executed, (2) when large transactions were executed, (3) when the server had a very fast CPU, (4) when the server had a very fast CPU and there was no network delay, and (5) when interactive transactions were executed. The experiment parameters that varied from the values in Table 5 are given in the description of the experiments.

5.1. Short Transaction Experiment

In this experiment, short transactions were executed that read an average of 8 objects without any internal delay.

The transaction response time is determined by the following factors: (1) the client CPU cost, (2) the network delay caused by messages exchanged between clients and the server, (3) the server CPU and disk I/O cost, and (4) waiting time due to data contention. In this experiment, the server becomes a bottleneck when there are a large number of clients. Therefore, algorithms that can reduce server load should win. Algorithms that cause more transaction aborts lose.

Figures 8(a) through 8(c) show the average transaction response time with low locality (*InterXactLoc* set to 0.05). None of the algorithms appears to be better than two-phase locking regardless of the write probability. The four algorithms behave virtually the same when transactions are read-only, as shown in Figure 8(a). This is because they are all variations of two-phase locking, and these variations make no difference in this case.



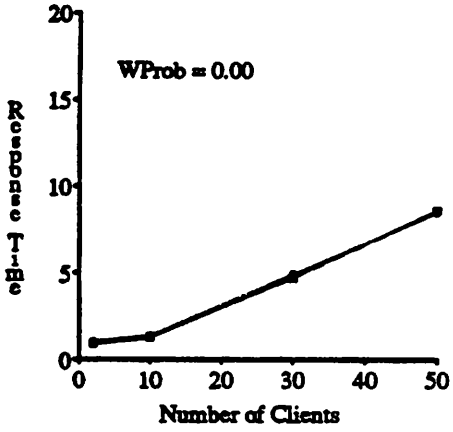


Figure 8(a). Response Time (Loc = 0.05)

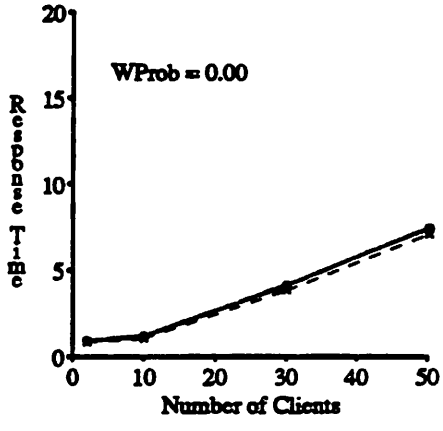


Figure 9(a). Response Time (Loc = 0.25)

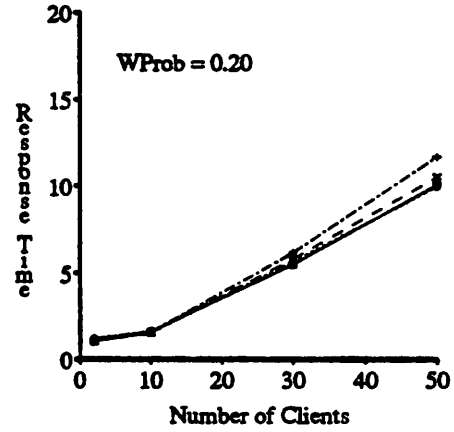


Figure 8(b). Response Time (Loc = 0.05)

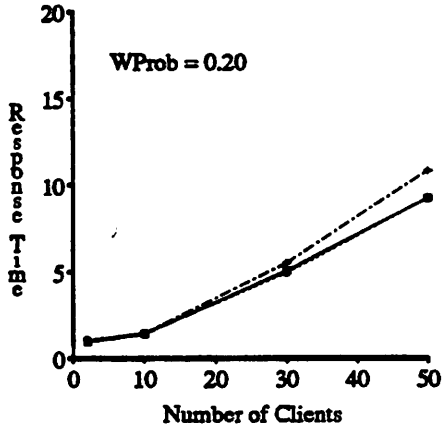


Figure 9(b). Response Time (Loc = 0.25)

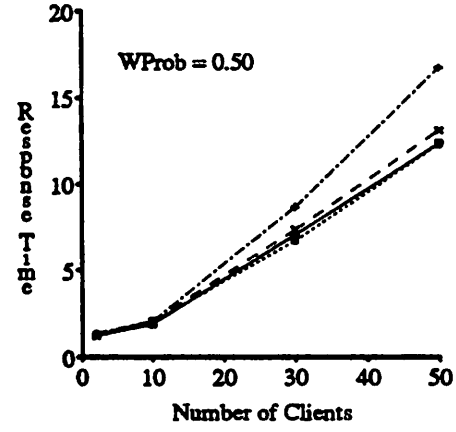


Figure 8(c). Response Time (Loc = 0.05)

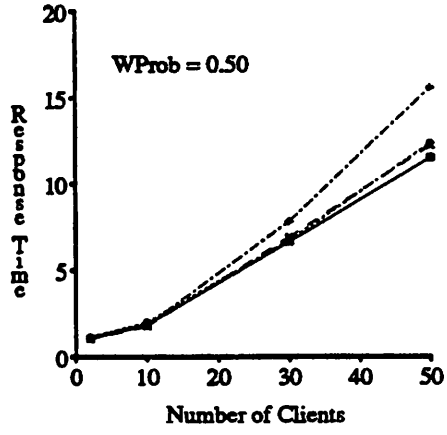


Figure 9(c). Response Time (Loc = 0.25)

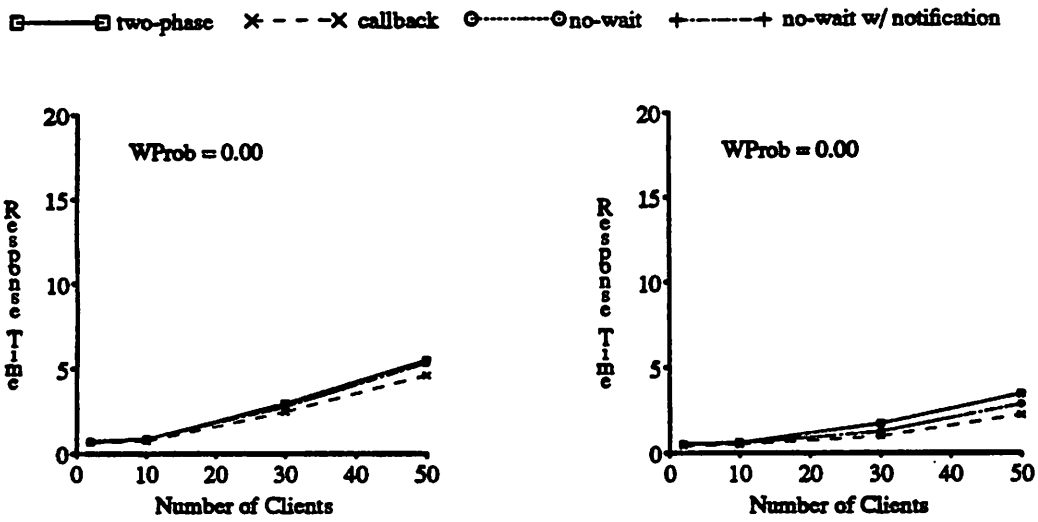


Figure 10(a). Response Time (Loc = 0.50)

Figure 11(a). Response Time (Loc = 0.75)

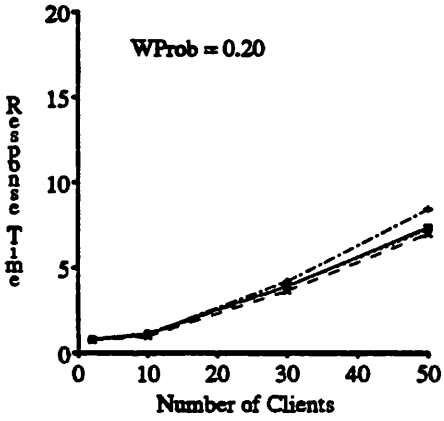


Figure 10(b). Response Time (Loc = 0.50)

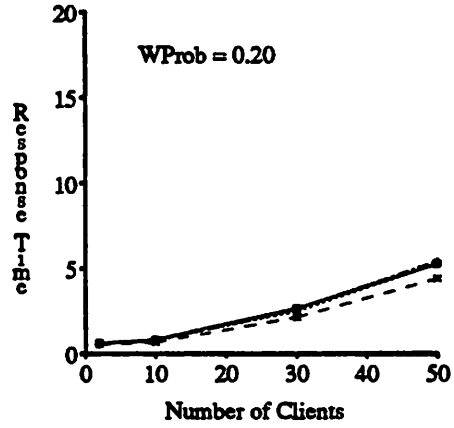


Figure 11(b). Response Time (Loc = 0.75)

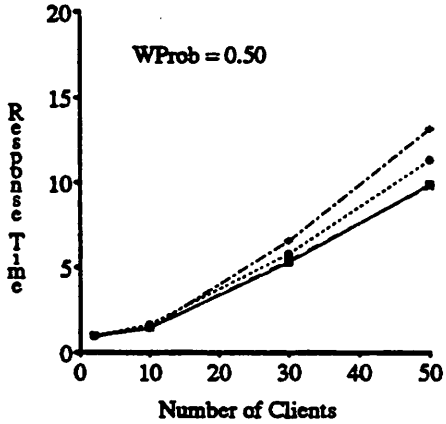


Figure 10(c). Response Time (Loc = 0.50)

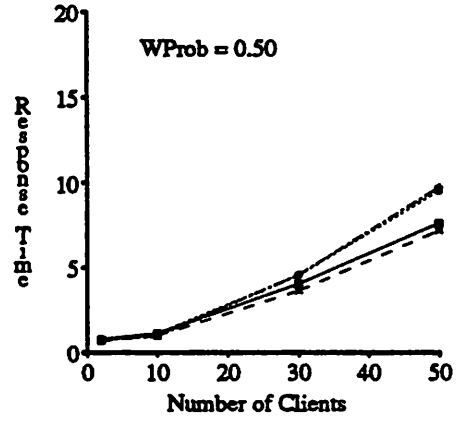


Figure 11(c). Response Time (Loc = 0.75)

When the write probability is higher, two-phase locking with notification performs poorly when there are a large number of clients, as shown in Figures 8(b) and 8(c). Under the notification protocol, the server sends updates to the clients when objects are updated. With very low locality, transactions almost never read objects cached by previous transactions. Consequently, these notification messages do not improve performance (i.e., reduce aborts) but they do increase server CPU contention which in turn increases the transaction response time. Callback locking also performs marginally worse than two-phase locking because the server has to send messages to clients asking them to return locks, but the retained locks do not help the client transactions much due to the low locality. With no-wait locking, the transaction restart rate is higher than two-phase locking. But the server does not need to respond to every client message which reduces the server CPU contention and offsets the wasted work of aborted transactions. Thus, two-phase locking is the best.

Figures 9(a) through 9(c) show the average transaction response time with medium locality (*InterXactLoc* set to 0.25). These graphes are similar to 8(a) through 8(c) because the locality is not high enough to make a significant difference. However, callback locking performs a little better than two-phase locking when there are no object updates, as shown in Figure 9(a). In this case, when a client accesses a cached object with a retained lock, it does not need to send a message to the server, which saves client waiting time and reduces the server CPU load which in turn improves the response time of other transactions. But its performance degrades for write probabilities that are higher, as shown in Figure 9(c). Thus, callback locking is best when transactions are read-only. Otherwise two-phase locking is the best.

Figures 10(a) through 10(c) show the average response time for high locality transactions (*InterXactLoc* set to 0.50). Callback locking outperforms two-phase locking by about 15% when there are no object updates, as shown in Figure 10(a). This is because clients can take advantage of the retained locks, but the server never has to send messages to the clients requesting them to return locks. No-wait locking also outperforms two-phase locking because the server does not need to respond to every client message which reduces the server load.

However, the performance of callback and no-wait locking degrade for higher write probabilities, as shown in Figures 10(b) and 10(c). When the write probability is 0.50, two-phase locking is as good as callback locking, as shown in Figure 10(c), because the cost of retrieving locks offsets the savings of retained locks. No-wait locking causes more transaction aborts, which offsets the savings of the "no-wait" protocol. Although notification can help no-wait locking to reduce transaction aborts, the notification messages cost even more than what they save. Thus, callback locking is best at low write probabilities and either two-phase or callback locking is best at higher write probabilities.

Figures 11(a) through 11(c) show the average transaction response time for very high locality transactions (*InterXactLoc* set to 0.75). Callback locking dominates the other three algorithms. When there are no object updates, as shown in Figure 11(a),

callback locking performs about 35% better than two-phase locking and 17% better than no-wait locking. No-wait locking also outperforms two-phase locking, because clients do not have to wait for responses from the server when they access cached objects which saves clients waiting time and reduces server CPU contention.

However, the performance of callback and no-wait locking degrades quickly for higher write probabilities, as shown in Figures 11(b) and 11(c). When the write probability is 0.2, callback locking only performs about 15% better than two-phase locking while no-wait locking performs about the same as two-phase locking. When the write probability is 0.5, callback locking is just slightly better than two-phase locking while no-wait locking performs worse than two-phase locking because it causes more transaction aborts. Thus, callback locking is the best.

An interesting observation from these figures is that no-wait locking with notification rarely performs better than no-wait locking. When there are no object updates, they are the same, as shown in Figures 8(a)-11(a). Notification is very sensitive to the server CPU load which is determined by the number of clients. With 2 clients, notification does not make a difference because there is only one other client to notify and there are few transaction aborts. When there are 10 clients and a high locality, as shown in Figures 10(a), 10(b), 11(a), and 11(b), notification improves no-wait locking slightly because the transaction aborts that it saves more than compensate for the cost of sending updates to clients. When there are a large number of clients (30 to 50), the server is a bottleneck, so sending updates to clients causes more contention which makes notification not worthwhile even though it can reduce the number of transaction aborts. When the locality is very high (*InterXactLoc* set to 0.75), clients access cached objects most of the time. Consequently, the server becomes less loaded and more transaction aborts can be avoided by notification. Even when there are a large number of clients, no-wait locking with notification is only slightly worse than no-wait locking, as shown in Figures 11(b) and 11(c),

We also measured transaction throughput rate for all cases. Figures 12(a) and 12(b) show the transaction throughput rate for medium and very high locality, both with medium write probability. The corresponding response time is shown in Figures 9(b) and 11(b). Transaction throughput results are the same as response time.

In summary, we conclude that in the setup of our experiments where server CPU is the bottleneck and the workload is short transactions without internal delay:

(1) Two-phase locking and callback locking dominate no-wait locking and no-wait locking with notification.

(2) Callback locking is better than two-phase locking when locality is high (greater than 0.5), or, it is better when locality is medium (between 0.25 and 0.5) and write probability is low (less than 0.2).

(3) No-wait locking is better than two-phase locking when locality is high (greater than 0.5) and write probability is low (less than 0.2). However, callback locking

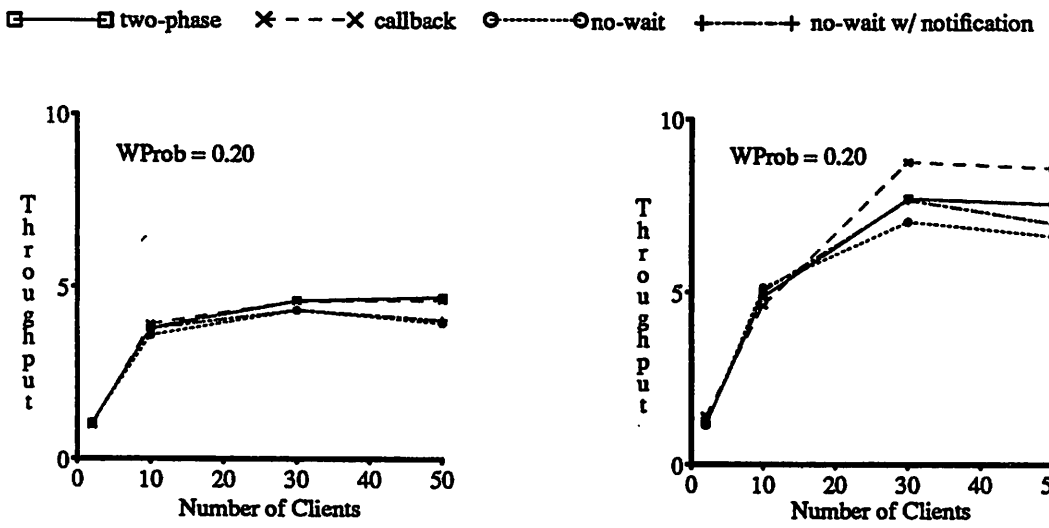


Figure 12(a). Throughput (Loc = 0.25)

Figure 12(b). Throughput (Loc = 0.75)

Locality \ WriteProb	0.00	0.20	0.50
0.05	any algorithm	two-phase locking	two-phase locking
0.25	callback locking	two-phase locking	two-phase locking
0.50	callback locking	callback locking	two-phase locking
0.75	callback locking	callback locking	callback locking

Figure 13. Summary of Algorithm Performance

is better than both of them in these cases.

(4) No-wait locking with notification is slightly better than no-wait locking only when the locality is very high (greater than 0.5) and there are a small number of clients. It does not help under other circumstances.

Figure 13 summarizes our findings. In the upper-left corner, it doesn't make any difference which algorithm is used. In the lower-left area, callback locking should be used. In the remaining area, two-phase locking should be used.

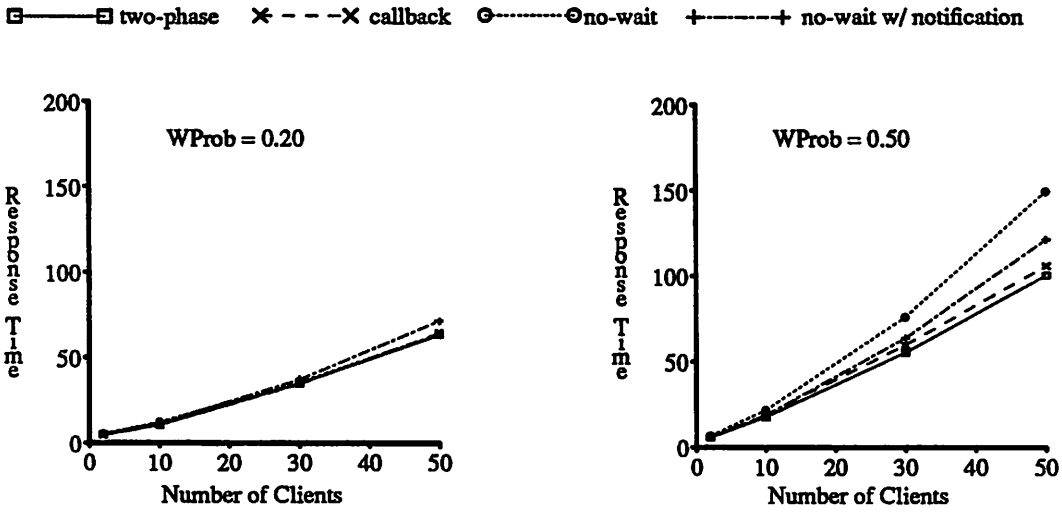


Figure 14(a) Response Time (Loc = 0.25)

Figure 14(b) Response Time (Loc = 0.25)

Figure 15(a) Response Time (Loc = 0.75)

Figure 15(b) Response Time (Loc = 0.75)

5.2. Large Transaction Experiment

In this experiment, we studied the impact of large transactions on the performance of cache consistency algorithms. *MinXactSize* was set to 20 and *MaxXactSize* to 60. Each transaction read an average of 40 objects.

Compared to the previous experiment, the server load was even higher because there were more operations within a transaction. Transaction aborts became more expensive due to the larger transaction size and the server load. Figures 14(a) and 14(b) show the transaction response time for medium locality (*InterXactLoc* set to 0.25). Figures 15(a) and 15(b) show the transaction response time for very high locality (*InterXactLoc* set to 0.75).

There are three interesting observations. First, results of this experiment are very similar to those of the previous experiment because the server is still a bottleneck. The shape of the lines in Figure 14(a) are very close to that of the same case for the previous experiment shown in Figure 9(b).

Second, both no-wait locking and callback locking degrade more rapidly for higher write probabilities than in the short transaction experiment because they cause more transaction aborts that are more expensive. When there are no object updates, both callback and no-wait locking outperform two-phase locking when locality is above medium. But with high write probabilities, as shown in Figures 14(b) and 15(b), callback locking became slightly worse than two-phase locking, and no-wait locking became much worse than two-phase locking.

Third, notification helps no-wait locking, as shown in Figures 14(b) and 15(b), because it reduces the number of transaction aborts. The savings of the reduced transaction aborts more than compensate for the cost of notification messages. However, both no-wait locking and no-wait locking with notification are still dominated by two-phase and callback locking.

5.3. Fast Server Experiment

The server was the bottleneck in the previous two experiments. In this experiment, we set the server CPU speed to 20 mips, 10 times faster than in the previous experiments. All other parameters were the same as reported in Table 5. The workload was short transactions without internal delay.

In this experiment, we found that the system bottleneck shifted from the server CPU to the network. The network became a bottleneck when the number of clients reached 30. Therefore, algorithms that can reduce the number of messages exchanged between clients and the server should have better performance. Figures 16(a) and 16(b) show the transaction response time for medium locality (*InterXactLoc* set to 0.25). Figures 17(a) and 17(b) show the transaction response time for very high locality (*InterXactLoc* set to 0.75). They are very close to the corresponding figures for the short transaction experiment (16(a) and 16(b) correspond to 9(b) and 9(c), 17(a) and 17(b) correspond to 11(b) and 11(c)). The only obvious difference is that no-wait locking with notification

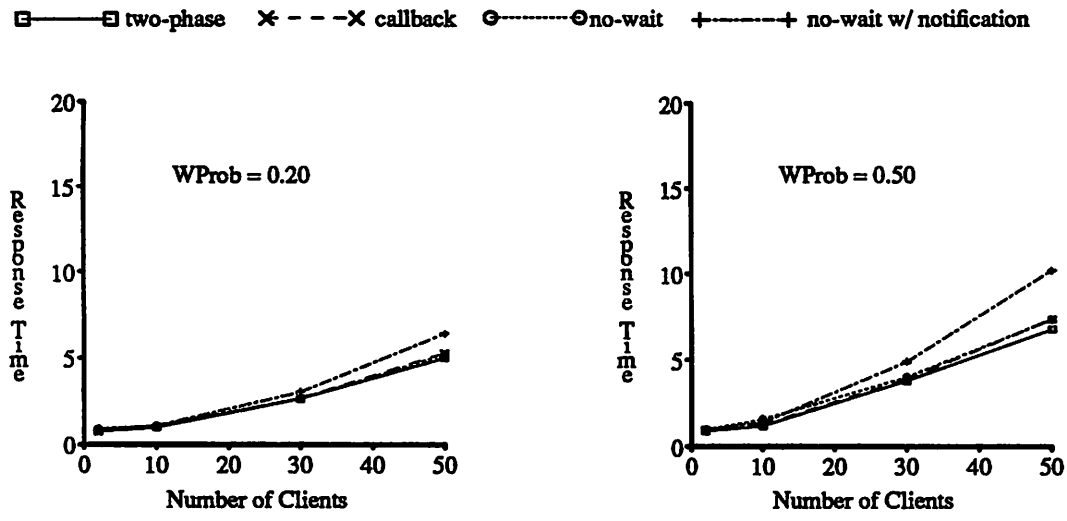


Figure 16(a) Response Time (Loc = 0.25)

Figure 16(b) Response Time (Loc = 0.25)

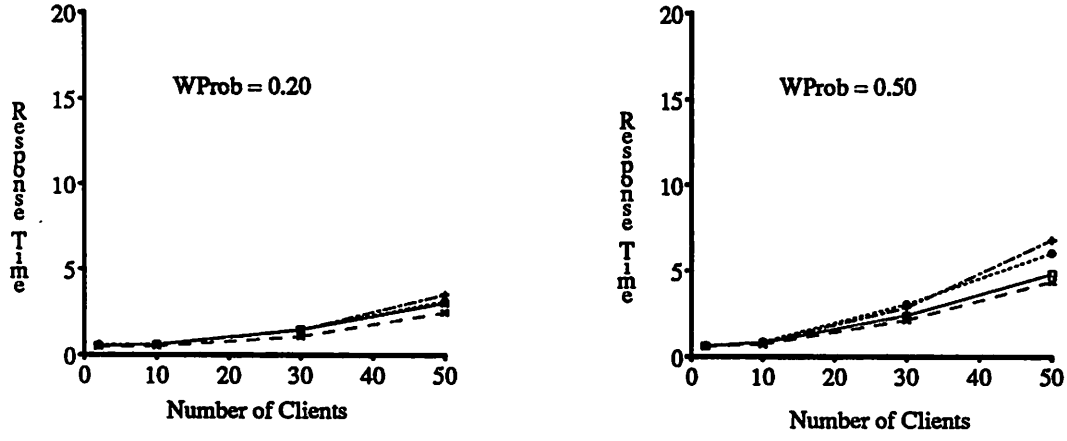


Figure 17(a) Response Time (Loc = 0.75)

Figure 17(b) Response Time (Loc = 0.75)

performs more poorly with a large number of clients due to the large number of notification messages.

This experiment gave virtually the same results as the short transaction experiment because there is a close relationship between the number of messages and the server load. Both the network and the server are resources for which clients compete. More messages increase both network and server contention. Therefore, algorithms that perform poorly in the short transaction experiment because of server contention perform equally poorly in this experiment because of network contention.

5.4. Fast Network and Fast Server Experiment

In this experiment, we wanted to find out what would happen if the network bottleneck was removed. We assumed that the network was infinitely fast and set *NetDelay* to 0. Server CPU speed was set to 20 mips. All other parameters were the same as in Table 5. A short transaction workload without internal delay was executed.

In this experiment, there was no bottleneck. The resource with the highest degree of contention was the server disks which reached a utilization rate of about 80% with 50 clients. Figures 18(a) and 18(b) show the transaction response time for medium locality (*InterXactLoc* set to 0.25). Figures 19(a) and 19(b) show the transaction response time for very high locality (*InterXactLoc* set to 0.75).

There are substantial differences between the results of this experiment and those of the short transaction experiment. Comparing Figures 18(b) and 19(b) to their corresponding Figures 9(c) and 11(b), we find that no-wait locking with notification performs significantly better. It dominates other algorithms in both cases. In previous experiments, notification performed poorly because it sent more messages than all other algorithms and messages were expensive. The cost of messages was more than the savings due to the reduced transaction aborts. However, in this experiment, with *NetDelay* set to 0 and a very fast server CPU, messages are very cheap while disk I/O's become relatively more expensive. Sending updates to clients can reduce the number of transaction aborts and the number of disk accesses because clients do not need to fetch invalid objects from the server which may require disk reads. Therefore, no-wait locking with notification outperforms the other three algorithms.

When locality is high and write probability is low, as shown in Figure 19(a), callback performs slightly better than the other algorithms (this is not obvious in the figure because of the scale). This is because notification does not make a difference when the write probability is low. And, when locality and write probability are low, there is not much difference between the algorithms.

Figures 20 and 21 show the transaction throughput rate for a medium locality (*InterXactLoc* set to 0.25) and very high locality (*InterXactLoc* set to 0.75) workload. No-wait locking with notification and callback locking dominate the other algorithms.

In summary, when locality and write probability are low, all algorithms behave virtually the same. When locality is high and write probability is low, callback locking performs best. In other situations, no-wait locking performs the best.

5.5. Interactive Transaction Experiment

In this experiment, we studied the impact of interactive transactions on the performance of the algorithms. *UpdateDelay* was set to 5 seconds and *InternalDelay* to 2 seconds. All other parameters were the same as reported in Table 5.

Because of the long internal delay and the limited number of clients, all resources are lightly used. Time saved from less communication with the server or not waiting for

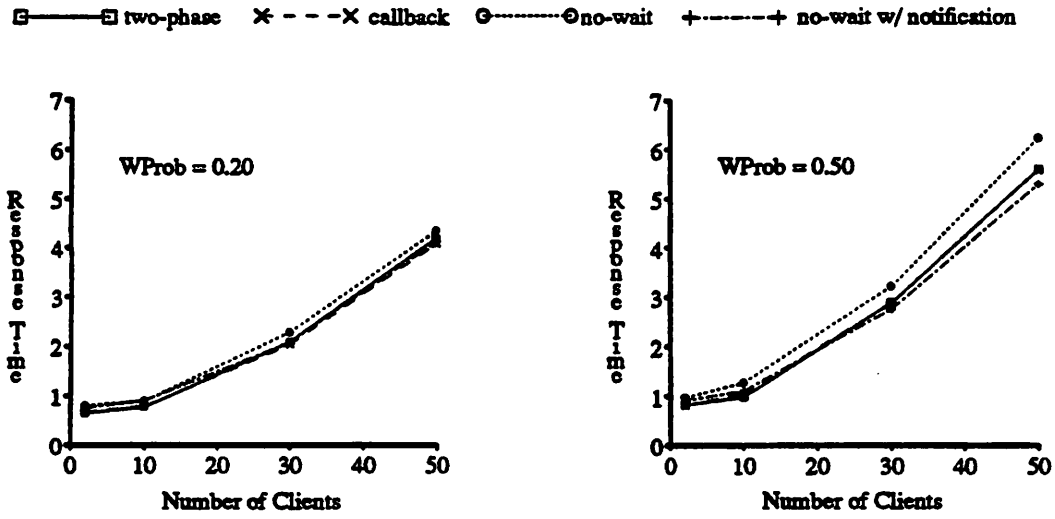


Figure 18(a) Response Time (Loc = 0.25)

Figure 18(b) Response Time (Loc = 0.25)

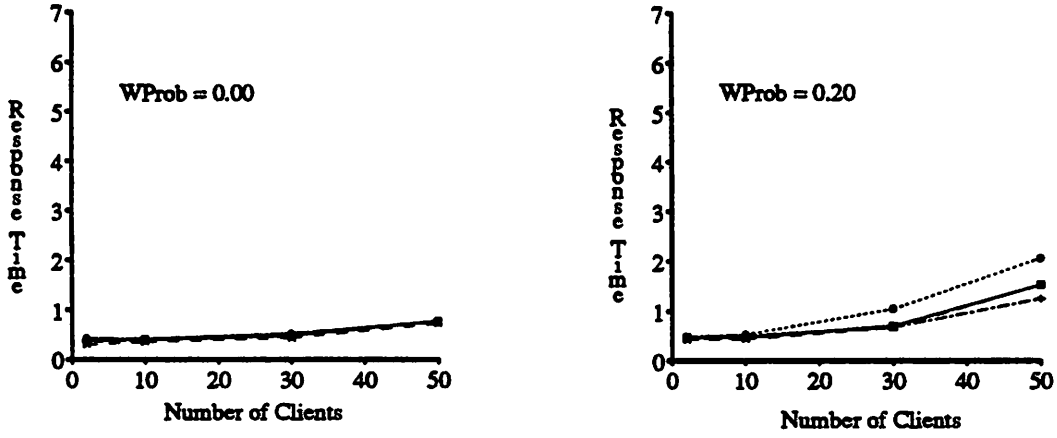


Figure 19(a) Response Time (Loc = 0.75)

Figure 19(b) Response Time (Loc = 0.75)

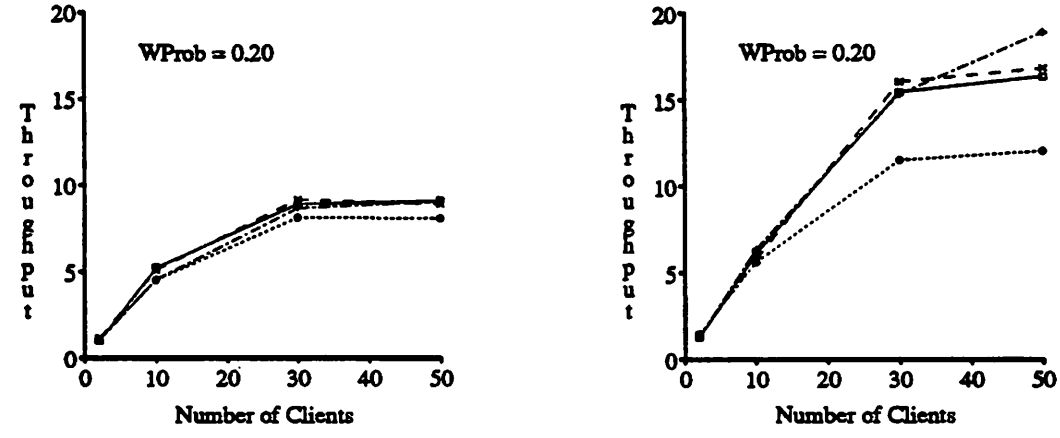


Figure 20. Throughput (Loc = 0.25)

Figure 21. Throughput (Loc = 0.75)

responses from the server is trivial compared with the internal delay time. Therefore, differences in transaction response time are mainly caused by data contention, or the number of restarts per transaction.

Figure 22(a) and 22(b) shows the transaction response time for medium locality (*InterXactLoc* set to 0.25). When there are no object updates, as shown in Figure 22(a), the response times for all algorithms are flat because they are all essentially determined by the internal delays within transactions which average 56 seconds (7 seconds for each object read and each transaction reads an average of 8 objects). However, with high write probability (*WriteProb* set to 0.50), algorithms that cause more transaction aborts perform poorly, as shown in Figure 22(b). The poor performance of callback and no-wait locking shown in Figure 22(b) are also related to the way they are implemented in our simulator. In both algorithms, clients receive asynchronous messages from the server. In the current implementation, these messages are not processed during the internal delay time.

In summary, when there are no object updates, the algorithms behave virtually the same. When the write probability is non-zero, two-phase locking is best because it has the least number of transaction aborts.

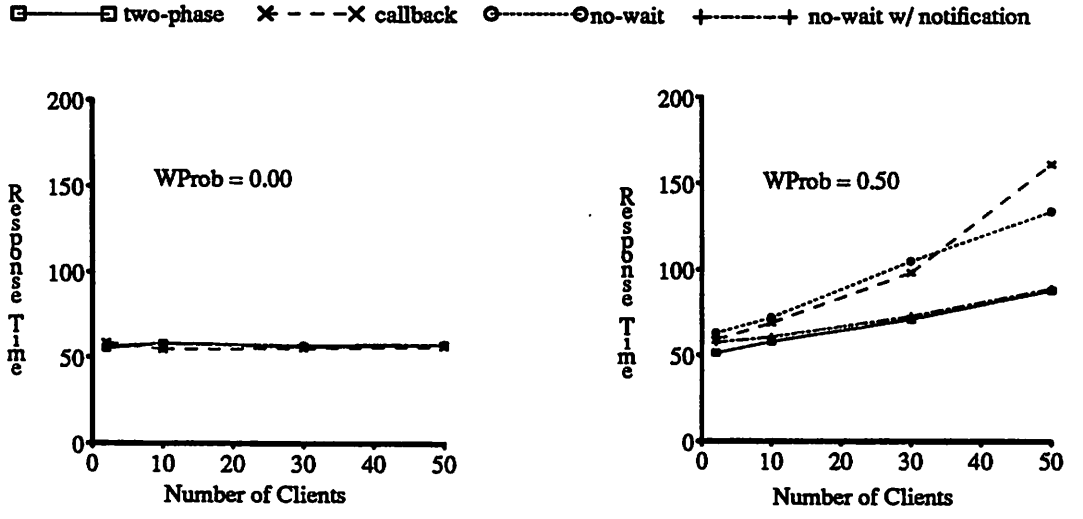


Figure 22(a) Response Time (Loc = 0.25)

Figure 22(b) Response Time (Loc = 0.25)

6. Conclusions

Five algorithms for inter-transaction caching were described and a simulation experiment was performed to compare their performance under various conditions. We found that two-phase locking and certification with inter-transaction caching almost always outperform the corresponding intra-transaction algorithms. Among the inter-transaction algorithms, when the network or the server is a bottleneck, two-phase locking and callback locking dominate no-wait locking and no-wait locking with notification. Callback locking is better than two-phase locking when the inter-transaction locality is high or when the inter-transaction locality is medium and the probability of object update is low.

When there is no network delay and the server is very fast, no-wait locking with notification and callback locking dominate two-phase and no-wait locking. Callback locking is better than no-wait locking with notification when inter-transaction locality is high and write probability is low. Otherwise no-wait locking with notification is better. For interactive transactions, two-phase locking is the best.

Current database applications typically have low inter-transaction locality and low to medium probability of object update, thus two-phase locking is used. It remains to be seen whether object-oriented DBMS applications have either high inter-transaction locality or medium inter-transaction locality and low write probability to justify changing to callback locking, or networks and server processing speed become fast enough to justify changing to no-wait locking with notification.

Another factor that has to be taken into account is how difficult it is to implement these algorithms. Two-phase locking with inter-transaction caching requires that the server maintain a version number for each object. The version number is cached with the object in each client cache. A client also needs to remember which cached objects have been locked by the current transaction. All messages between clients and the server are synchronous (i.e., all messages are initiated by clients and clients always wait for responses from the server). Therefore, the communication protocol should be easy to implement.

Callback locking requires more modifications. It is still necessary to have a version number for each object.† A client must remember which cached objects have been locked and whether they are locked by the current transaction. It also must process asynchronous messages from the server requesting the release of locks. The server lock manager needs to maintain a potentially much larger lock table. Deadlock detection becomes more difficult because if all retained locks are considered part of the wait-for graph, potentially many more deadlocks can happen. On the other hand, if retained locks are not considered part of the wait-for graph, when the server cannot get back a retained

† The version number is not necessary if a client assumes that all unlocked objects are invalid and always refetches them.

lock, it has to decide whether to continue waiting for the client to release it or to abort the client transaction.

No-wait locking also requires more modification than two-phase locking. It requires that a client remember which cached objects it has asked the server to lock but has not received a negative response. It also must handle asynchronous messages from the server requesting that a transaction be restarted. Notification requires that the server remember which objects have been cached by which clients if it sends updates to individual clients instead of broadcasting them to all clients.

Acknowledgement

We want to thank Mike Stonebraker for his advice on the simulation model and experiments, John Ousterhout for suggesting that we study callback locking, Luis Miguel, Wei Hong, Young-Chul Shim, Kevin Wilkinson for their constructive comments on our work, and Dan Gerson and Dan Weinreb for elaborating on the no-wait locking algorithm in private communication.

References

1. Agrawal, R., Carey, M. J. and Livny, M., "Models for Studying Concurrency Control Performance: Alternatives and Implications", *Proceedings of ACM-SIGMOD Conf. on Management of Data*, 1985.

2. Agrawal, R., Carey, M. J. and Livny, M., "Concurrency Control Performance Modeling: Alternatives and Implications", *ACM Transactions on Database Systems 12*, 4 (December 1987).

3. Agrawal, R. and Gehani, N. H., "ODE (Object Database and Environment): The Language and the Data Model", *Proceedings of ACM-SIGMOD Conf. on Management of Data*, Portland, Oregon, June 1989.

4. Andrews, T. and Harris, C., "Combining Language and Database Advances in an Object-Oriented Development Environment", *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, Orlando, Florida, October 1987, 430-440.

5. Bancilhon, F., et. al., "The Design and Implementation of O2, an Object-Oriented Database System", *Proceedings of the second International Workshop on Object-Oriented Database Systems*, Bad Münster am Stein-Ebernburg, FRG, September, 1988.

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

7. Bretl, R., et. al., "The GemStone Data Management System", in *Object-Oriented Concepts, Databases, and Applications*, Kim, W. and Lochovsky, F. H. (editor), 1989, ACM Press.

8. Carey, M. J. and Livny, M., "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication", *Proceedings of the 14th International Conference on Very Large Data Bases*, Los Angeles, California, August 29 - September 1, 1988.

9. Date, C. J., *An Introduction to Database Systems, Volumn I, 5th Edition*, Addison-Wesley, 1990.

10. Deux, O., et. al., "The Story of O2", *IEEE Transactions on Knowledge and Data Engineering 2*, 1 (March 1990), 91-108.

11. Gerson, D., Personal Communication, March 1989.

12. Howard, J. H., et. al., "Scale and Performance in a Distributed File System", *ACM Transactions on Computer Systems 6*, 1 (February 1988), 51-81.

13. Kempf, J. and Snyder, A., "Persistent Objects on a Database", STL-86-12, HP Labs, September, 1986.

14. Kim, W., et. al., "Architecture of the ORION Next-Generation Database System", *IEEE Transactions on Knowledge and Data Engineering 2*, 1 (March 1990), 109-124.

15. Object Design, Inc., ObjectStore DBMS, 1990.

16. Objectivity, Inc., Objectivity/DB, 1990.

17. Paepcke, A., "PCLOS: A Flexible Implementation of CLOS Persistence", *European Conference on Object-Oriented Programming*, 1988.

18. Richardson, J. E. and Carey, M. J., "Programming Constructs for Database System Implementation in EXODUS", *Proceedings of ACM-SIGMOD Conf. on Management of Data*, San Francisco, California, May 27-29, 1987.

19. Schwetman, H. D., "CSIM: A C-Based, Process-Oriented Simulation Language", *Proceedings of the 1986 Winter simulation Conference*, December 1986, 387-396.

20. Weinreb, D., et. al., "An Object-Oriented Database System to Support an Integrated Programming Environment", *IEEE Database Engineering Bulletin 11*, 2 (June 1988), 33-43.

21. Wilkinson, K. and Neimat, M., "Maintaining Consistency of Client-Cached Data", *Proceedings of the 16th International Conference on Very Large Data Bases*, 1990.