An Asynchronous Avoidance-Based Cache Consistency Algorithm for Client Caching DBMSs*

M. Tamer Özsu†
GMD-IPSI
Darmstadt, Germany

Kaladhar Voruganti
University of Alberta
Edmonton, Canada

Ronald C. Unrau‡
Cygnus Solutions
Sunnyvale, USA

Abstract

We present a new client cache consistency algorithm for client caching database management systems. The algorithm, called Asynchronous Avoidance-based Cache Consistency (AACC), provides both good performance as well as a low abort rate. We present simulation results that compare AACC with two leading cache consistency algorithms: Adaptive Callback Locking (ACBL) and Adaptive Optimistic Concurrency Control (AOCC). Callback cache consistency (e.g. ACBL) is the most widely implemented algorithm due to its low abort rate and good performance. AOCC is an optimistic algorithm that has been shown to outperform ACBL under certain workload and system configurations. Until now one could either have high performance and high abort rate as in AOCC, or relatively lower performance but the low abort rate of ACBL. Our performance study shows that AACC outperforms both ACBL and AOCC for important workloads and system configurations. AACC has the high performance of AOCC, as well as the robustness and low abort rate of ACBL.

1 Introduction

Most of the existing object and object-relational DBMSs are distributed in one form or another. This is motivated by the inherently distributed nature of the advanced applications that DBMSs have started to support (e.g, distributed multimedia applications, electronic commerce, digital libraries, medical information systems). In addition to distribution, these systems exhibit the following characteristics: they are multi-user, requiring scalable solutions; they require support for both small and large objects (image, video, audio); and they operate in both batch and interactive modes.

An important problem in distributed object and object-relational DBMSs is client cache consistency. The problem exhibits itself in multi-user systems where data are accessed by and reside in the caches of multiple clients that are connected to the servers via both local-area networks (LANs) and/or wide-area networks (WANs). Cache consistency algorithms can be classified as avoidance-based or detection-based [FLC97]. Avoidance-based algorithms prevent access to stale cache data within a transaction, whereas detection-based algorithms allow stale cache data access but detect and resolve them at commit time. Stale data refers to data in cache that is out-dated due to concurrent committed updates by another client. Adaptive Callback Locking (ACBL) is commonly accepted as the leading avoidance-based cache consistency algorithm [FC94] and Adaptive Optimistic Concurrency Control (AOCC) [AGLM95] is the leading detection-based cache consistency algorithm.

AOCC generally outperforms ACBL in environments where the client cache is sufficiently large to hold the entire transaction state (data and logs) and the application processing is strictly done at the clients [AGLM95]. AOCC achieves this even while encountering a higher abort rate than ACBL, mainly due to its efficient abort handling mechanism.

One might conclude that AOCC is a superior cache consistency algorithm since its performance is generally better than ACBL. However, performance is not the only issue; the high abort rate of AOCC makes it unsuitable for interactive application domains. Furthermore, it is necessary to evaluate how a high abort rate affects AOCC performance in environments where the application processing is performed not only at the clients but also at the servers (hybrid architectures) and when the entire transaction state cannot fit into the client cache. Hybrid architectures, where queries are sometimes executed at the client by caching the

*This research is supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada under grant OGP0000951.

†The author's permanent address is University of Alberta, Edmonton, Canada, T6G 2H1; email: ozsu@cs.ualberta.ca.

‡ This work was done when the author was with the University of Alberta, Department of Computing Science.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 24th VLDB Conference
New York, USA, 1998

necessary data, and sometimes executed at the server by shipping queries to the server, are emerging as the desirable client-server DBMS architectures [KF96]. In these systems, abort processing has an impact on the performance of all the clients. Transaction state cannot fit into the client cache in the case of large transactions, transactions accessing large objects (e.g. multimedia), or when multiple user processes share the client's cache.

These observations suggest that there is a need for algorithms which provide good performance while maintaining a low abort rate. Although an optimistic algorithm such as AOCC can outperform ACBL, most commercial client caching DBMSs continue to use ACBL (or its variants), because they also have to support applications which cannot tolerate a high abort rate. Ideally, it is desirable to use a cache consistency algorithm whose performance approaches that of the best (avoidance-based or detection-based) cache consistency algorithm while incurring a low abort rate.

1.1 Scope

In this paper, we present a new cache consistency algorithm, Asynchronous Avoidance Cache Consistency (AACC), which exhibits good performance and a low abort rate. AACC is an avoidance-based algorithm because it does not access stale cache data and this results in a significantly lower abort rate than AOCC. AACC uses a piggyback cache consistency mechanism on private data and an asynchronous cache consistency mechanism on shared data. An asynchronous message is explicitly sent by the client whereas a piggybacked message is sent along with another message. Hence, the use of non-synchronous messages allows AACC to consistently outperform ACBL while ensuring a low abort rate.

The second contribution of this paper is the presentation of performance results comparing ACBL, AOCC, and AACC. This performance study reverses the commonly held belief that asynchronous cache consistency algorithms do not outperform synchronous cache consistency algorithms such as CBL [WR91]. Moreover, the previous results indicating that an optimistic high abort algorithm such as AOCC is superior to ACBL [AGLM95] might lead one to believe that high abort rates are a necessary evil in order to obtain high performance in client caching systems. However, we show that a low abort algorithm such as AACC does outperform AOCC for the most common client caching workload and system configuration. This performance study also helps in clarifying the performance characteristics of ACBL and AOCC. An earlier study shows that AOCC performs better than ACBL [AGLM95], but that study does not consider workloads where application processing is performed at both the client and the server, nor does it consider cases where the transaction state does not completely fit in the client cache, nor when the network experiences delays (similar to those present in WANs). One would expect the performance of algorithms to be affected in these situations. Therefore, in this paper we evaluate the performance of AACC, ACBL and AOCC for these newer system and workload configurations.

1.2 Paper Organization

In section 2 we briefly present related work describing other DBMS cache consistency algorithms. Section 2 also describes ACBL and AOCC. Section 3 contains a detailed description of the proposed AACC algorithm. Sections 4 and 5 present the experimental setup and results, respectively. Section 6 contains an analysis and discussion of the performance results. Finally, section 7 contains our conclusions.

2 Background

The DBMS cache consistency algorithms can be classified as avoidance-based or detection-based. Avoidance-based and detection-based algorithms can, in turn, be classified as synchronous, asynchronous or deferred depending upon when they inform the server that a write operation is performed. In synchronous algorithms, the client sends a lock escalation message at the time it wants to perform a write operation and it blocks until the server responds. In asynchronous algorithms, the client sends a lock escalation message at the time of its write operation but does not block waiting for a server response (it optimistically continues). In deferred algorithms, the client optimistically defers informing the server about its write operation until commit time. In most of the deferred avoidance-based algorithms, the server blocks a client transaction at commit time if the client has updated an object that has been read by other clients [FLC97]. Figure 1 depicts this classification along with some of the popular cache consistency algorithms.

	Synchronous	Asynchronous	Deferred
Avoidance Based	CBL [FC94] ACBL [CFZ94]	AACC [This paper]	O2PL [CFLS91]
Detection Based	C2PL [CFLS91]	NWL [WR91]	AOCC [AGLM95]

Figure 1: DBMS Cache Consistency Algorithms

2.1 Known Performance Results

In client caching (or data shipping) systems, inter-transaction caching of data and locks is generally accepted as a performance enhancing optimization [FC94]. As well, for most user workloads, invalidation of remote cache copies during updates is preferred over propagation of updated values to the remote client sites [FC94]. Furthermore, the ability to switch between page and object level locks is generally considered to be better than strictly dealing with page level locks [CFZ94].

Within the family of avoidance-based algorithms, it has been shown [FC94] that the synchronous callback locking (CBL) algorithm, despite its higher messaging overhead, has similar performance to the optimistic two-phase locking (O2PL) [CFLS91] class of algorithms while incurring a much lower abort rate [FC94]. In O2PL, the write lock escalation message is deferred until commit time, whereas

in CBL, the clients send synchronous lock escalation messages at the time of the update operation and do not proceed until they receive a response from the server.

There are many performance studies comparing avoidance-based and detection-based algorithms [FC94, AGLM95, WR91]. The general conclusions are that synchronous avoidance-based algorithms, such as CBL, are superior to synchronous detection-based (e.g. C2PL) and asynchronous detection-based (e.g. NWL) algorithms. It has been shown that deferred detection-based algorithms (e.g. AOCC) can outperform synchronous avoidance-based algorithms (e.g. ACBL) even while encountering a high abort rate.

There has also been an attempt at developing a hybrid temperature-based algorithm [CLH97], where the data contention temperature is maintained for each object. If the temperature is high then the clients operate on the object in a pessimistic manner; if the temperature is low, the clients operate on that object in an optimistic manner. However, due to the reactive nature of this algorithm, changing user data access patterns, and dynamic addition and deletion of clients can lead to high abort rates and low performance. The performance of this approach [CLH97] with respect to AOCC and ACBL is not known.

2.2 ACBL

ACBL is a synchronous, avoidance-based cache consistency algorithm [CFZ94]. Clients cache both data and read locks across transaction boundaries but they need to obtain write permission from the server before they can proceed with write operations. ACBL can dynamically acquire either page or object level locks, and thus, it is an adaptive version of the page level CBL algorithm. Clients try to acquire page level write locks; failing that, they try to acquire object level write locks on shared pages. If the page is cached at other clients, the server sends callback messages to other clients asking them to downgrade or relinquish their locks. ACBL ensures that transactions never access stale data and therefore never have stale cache aborts. However, in ACBL, one can encounter deadlock related aborts. We utilize the following 4 scenarios (Figure 2) to highlight the key points of ACBL. For simplicity, these scenarios deal with only two clients, but the discussion is valid for *n* clients.

- **Scenario 1:** Assume that page 1 is only cached at client 1 and it has a read lock on page 1. Client 1 wants to update object 1 on page 1 and therefore it sends a message to the server to obtain a write lock for page 1. Client 1 blocks until it gets a response from the server. Since there is no one else caching page 1, the server immediately grants the write lock for page 1 to client 1. Thus, even if a page is not cached elsewhere, in ACBL, the clients send lock escalation messages to the server and block until getting a response from the server.

- **Scenario 2:** Client 1 wants to update object 1 on page 2 which is also present at client 2 due to inter-transaction caching; however, it is not being actively used at client 2. Both clients hold a read lock on the page. Client 1 sends a lock escalation message to the server and blocks until its gets a reply. The server, in turn, sends a callback message to client 2. Since client 2 is not using page 2, it invalidates page 2 from its cache and sends a callback reply to the server. The server then sends a response to client 1 granting it an exclusive lock on page 1. Thus, when a page is cached at multiple clients, in addition to the round trip message between the lock requesting client and the server, there are round trip callback messages between the server and all of the other clients where the page is cached, and the initial lock requesting client blocks until all of these messages are processed.

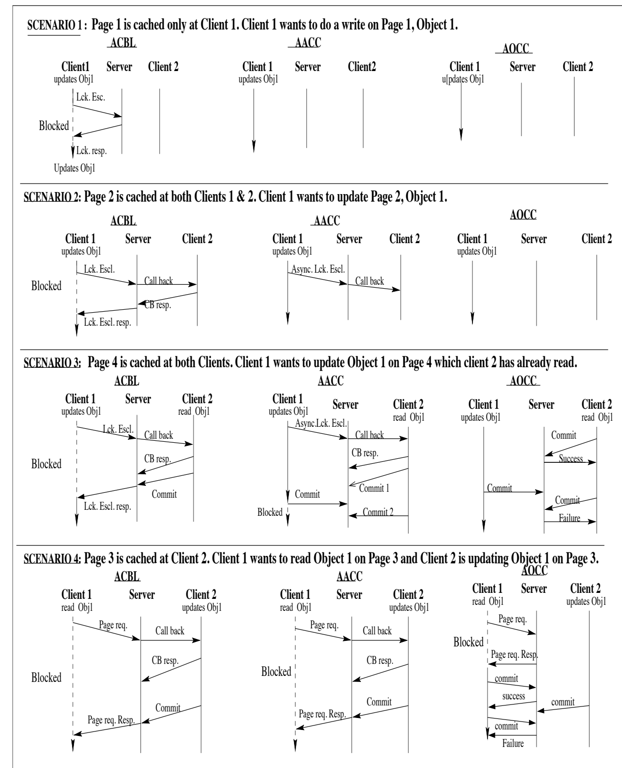

Figure 2: Cache Consistency Scenarios

- **Scenario 3:** Page 4 is shared by both clients 1 and 2. Client 1 wants to update object 1 on page 4 and client 2 has already read the object. Client 1 sends a lock escalation message to the server which then sends a callback message to client 2. Client 2 indicates that it cannot comply with the request. Client 1 stays blocked until client 2 commits and releases the page. Thus, in ACBL, update operations remain blocked until the appropriate lock is obtained from the server.

- **Scenario 4:** Client 2 holds an exclusive lock on page 3 and is updating object 1. Client 1 wants to read object 1 on page 3 and it sends a message to the server to obtain page 3. The server sends a callback message to client 2 which responds by indicating that it is updating object 1 on page 3. Client 1 remains blocked until client 2 commits. Thus, read operations remain blocked until the appropriate lock is obtained from the server.

2.3 AOCC

AOCC is a deferred, detection-based cache consistency algorithm. In AOCC, clients implicitly obtain read permissions on cached data, but if they subsequently update cached data, they defer all of their write notification messages until commit time. AOCC does not prevent the access of stale data by clients. The updates of a committed transaction result in corresponding invalidations being sent to the other affected clients. These invalidations are piggybacked on other messages. If the client that receives an object invalidation has accessed the corresponding object, then it performs a stale cache abort. Since this is an optimistic algorithm and no locking is involved, clients do not encounter read/write or write/write blocking and therefore, deadlocks do not occur in AOCC. However, in addition to stale cache aborts, AOCC is susceptible to starvation. That is, a client transaction repeatedly aborts and is not able to commit.

In AOCC, the server has to perform commit time validation on every object that has been accessed by a transaction. The server checks whether the client accessed the most recent committed version of the object. This validation overhead is not present in ACBL since the algorithm ensures that clients do not access stale data. In AOCC, the server maintains an invalidation queue for each of the clients which stores the list of committed updates of other clients that can potentially have an impact on this client. The invalidation queue is used by the server while performing commit time validation.

In AOCC, the clients do not send lock escalation messages and the server piggybacks invalidation messages to the affected clients. We now use the same scenarios as before (Figure 2) to analyze AOCC:

- **Scenario 1:** Client 1 wants to update object 1 on page 1 and it is the only client caching that page. It does not send any lock escalation messages to the server for this update; it simply goes ahead and performs its update on object 1 on page 1. The server is notified about this update by the client during its commit operation. Thus, in AOCC, there are no explicit lock escalation messages.
- **Scenario 2:** Client 1 wants to update object 1 on page 2 which is also cached at client 2. Client 1 does not send any lock escalation message to the server; it goes ahead and performs its update on the object. Client 1 informs the server about the update during its commit operation. Therefore, the server does not send any callback messages to client 2, but piggybacks an invalidation message to client 2.
- **Scenario 3:** Client 1 wants to update object 1 on page 4. This page is cached at both clients 1 and 2, and the latter has already read object 1 on page 4. Client 1 does not send any lock escalation messages to the server for the update; it informs the server during its commit operation. The server then decides whether client 2 can commit or abort. If client 2 commits before client 1, then the client 2 transaction commits (sneaks through), followed by client 1 transaction. If client 1 commits before client 2, then client 2 transaction aborts.
- **Scenario 4:** Page 3 is cached at client 2 and object 1 on this page has been updated by this client. Client 1 wants to read the same object. Client 1 goes ahead and gets page 3 from the server, and it accesses object 1. Therefore, readers never block in AOCC. If client 1 commits before client 2, then it successfully commits. If client 2 commits before client 1 then client 1 aborts.

In ACBL, a read/write conflict always results in the blocking of one of the transactions; in AOCC, the reading transaction can successfully commit (sneak through) if it reaches the commit point first, and the reading transaction aborts if the writing transaction commits first. This causes the blocking rate of ACBL to be higher than the abort rate of AOCC, but the abort rate of AOCC is higher than the abort rate of ACBL. In AOCC, when a transaction aborts, the client simply copies the undo logs that are maintained in its memory and restarts the transaction. This in turn speeds up abort processing as, for most non-conflicting objects, the client does not have to go to the server again to obtain the necessary pages.

3 Asynchronous Avoidance Cache Consistency

AACC is an asynchronous, avoidance-based cache consistency algorithm. It achieves high performance while retaining a low abort rate. AACC overcomes the fundamental problems and limitations of AOCC (high abort rate), and ACBL (high message transmission and message blocking overhead). AACC accomplishes this by applying a number of performance enhancement techniques as well as the adoption of various features of ACBL and AOCC. In this section, we first describe AACC and then illustrate its operation using the scenarios considered in section 2.

3.1 The AACC Algorithm

In AACC, as in ACBL, the clients implicitly obtain page level read locks when a page is brought into the client's cache. Clients retain page level read locks across transaction boundaries, but they relinquish page level write locks (change them to read locks) at the end of a transaction. The server and clients both play a role in lock management. The server primarily manages locks at page level, and the clients manage locks at both page and object levels. The server also manages locks at object level for objects on pages that are being simultaneously written to by multiple clients. The server performs deadlock processing when there are lock conflicts. The clients do not block at the time they perform a write operation; instead a client blocks at commit time if its updates will make a remote client's cache contain stale objects. The blocking at commit time makes AACC an avoidance-based cache consistency algorithm. In AACC, pages can be locked in *private-read*, *shared-read* and *write* modes, and objects can be locked in *read* and *write* modes. While satisfying a client's page request, in addition to returning the page, the server also informs the client as to whether the page is cached elsewhere (*shared-read* lock mode) or whether the receiving client is the only

client caching the particular page (*private-read* lock mode). This notion of *private* and *shared* lock modes is then used by the client to decide whether it needs to send a lock escalation message in asynchronous manner or in a deferred manner (via a piggyback message). The algorithm is described in detail below.

- **Data Request:** When a client wants to access an object whose page is not in its cache, it sends a page request to the server. When the server receives the request, it checks to see whether the page is cached at other clients.
  - If the page is not cached anywhere else, it returns the page to the client in *private-read* mode.
  - If the page is cached at another client in *private-read* mode then the page is returned to the requesting client in *shared-read* mode. The server also informs, via a piggy-back message, the client holding the page in *private-read* mode to change the page lock to *shared-read* mode. The inherent message delay may cause situations where one client has the page in *private-read* mode and other clients have the same page in *shared-read* mode.
  - If the page is cached elsewhere in *shared-read* mode, then the server returns the page to the client in *shared-read* mode.
  - If the page is cached at another client in *write* mode, then the server issues a callback message to the remote client indicating the object and the page that is being requested.
    * Upon receiving the callback, the remote client checks to see whether it is updating the particular object. If not, it changes the page lock to *shared-read* (means the client is sending object level lock escalation messages for future object updates on the page) and returns the object identifiers of the objects on that page that have been updated. If it is updating the requested object, it informs the server that it cannot satisfy the request.

    Upon receiving a positive callback response the server marks off the objects that are updated at the remote client and sends the page to the requesting client. If the server receives a negative callback response, it blocks the requesting client until the client that holds the write lock commits.

- **Updates on Private-Read Locked Page:** When a client is performing an update on a *private-read* locked page, the client changes the page lock mode to *write*. The client then informs the server about this update by piggybacking the information on a subsequent message. Upon receiving the piggybacked message regarding the update and the lock escalation to the *private-read* locked page the server does the following:
  - If the page is residing at other clients in *shared-read* lock mode, then the server sends an invalidation message to the affected clients. The invalidation message requests the clients to purge the object and/or page from their caches. The server also informs the client that has performed the update to change its page lock for the updated page from *write* to *shared-read* if other clients are using the page but not that object.
  - If the page is not residing at other clients or has been successfully invalidated at other clients, then the server updates its lock tables to indicate that the client has a *write* lock for the page.

- **Updates on Shared-Read Locked Page:** When a client is performing an update on a *shared-read* locked page, the client sends an asynchronous lock escalation message to the server and continues with its processing. When the server receives this message, it sends callback messages (indicating both the object and the page) to the other clients that are caching this page.
  - If the client receiving the callback message is not using the page, it simply invalidates the page, and informs the server via a piggybacked message.
  - If the client is using the page but not the object, then it invalidates the object and informs the server via a piggybacked message.
  - If the client is using the object, then it sends an asynchronous callback response indicating that there is a conflict.

- **Callback Processing:** When the server receives a callback response indicating that there is a conflict, it performs deadlock processing, and if there are no deadlocks, the client that has performed the initial update cannot commit before the client that is reading the object. Here the server deadlock processing involves a check to see whether clients have updated objects that have been read by other clients. For example, if client 1 has updated an object read by client 2 and client 2 has updated an object read by client 1, then neither one of these clients can commit their respective transactions. If the server receives piggybacked callback page messages from all the relevant clients indicating that they have invalidated the page, the server then sends an asynchronous message asking the initial page updating client to upgrade its page lock from *shared-read* to *write* mode.
- **Commit Processing:** At commit time, the client sends the logs to the server and it changes *write* locks to *private-read* locks. The client also piggybacks messages informing the server of updates to *private-read* locked pages. At commit time, the server checks to see whether the particular client can go ahead with its commit or whether it should remain blocked since it has updated an object that has been read by another client. The server also changes the page *write* locks held by the committing client to *private-read* locks in its lock table. If a client has performed updates to

a *private-read* locked page, and this is being piggybacked on the commit message, then the server checks to make sure that no other client has that page in its cache in *shared-read* mode; and if another client does have that page, the server sends a callback message to that client. The server only allows the commit to proceed after receiving replies to all the pending callback messages from the necessary clients. The server moves logs on to a persistent storage area and it also activates the other client transactions that are waiting for this client to commit.

3.2 Scenarios describing AACC

We now analyze AACC using the same set of scenarios which were utilized to describe ACBL (Figure 2):

- **Scenario 1:** Client 1 wants to update object 1 on page 1 which is cached only at client 1 in private-read lock mode. The client goes ahead with the update without sending an explicit lock escalation message. The client informs the server about this update by piggybacking the lock escalation message on a subsequent message to the server. Therefore, unlike ACBL, no synchronous lock message is sent for updating an object residing on a page that is solely present at a single client. Therefore, AACC's behavior is similar to AOCC which helps in reducing message transmission and message blocking overheads in AACC.
- **Scenario 2:** Client 1 wants to update object 1 on page 2 which is cached at both clients 1 and 2 in shared-read lock mode. Client 1 sends an asynchronous message to the server and continues without blocking. The server in turn forwards this message to client 2. Client 2 invalidates page 2, but informs the server about this invalidation by piggybacking the information on a subsequent message. Therefore, unlike ACBL, the use of an asynchronous message helps in reducing message blocking overhead. Similarly, the absence of explicit response message from client 2 to the server also helps in reducing message transmission overhead in AACC.
- **Scenario 3:** Client 1 updates object 1 on page 4 and client 2 has already read object 1 on page 4. Page 4 is present at both clients in shared-read lock mode. Client 1 sends an asynchronous message to the server indicating its update. The server then forwards this message to client 2. Client 2 notices that there is a conflict and it sends a explicit response to the server. The server then performs deadlock processing and notes that client 1 can only commit after client 2 has committed in order to prevent stale cache aborts. Therefore, client 1 can go ahead with its commit if client 2 commits at commit point 1 but client 1 blocks if client 2 commits at commit point 2. As a result, unlike AOCC, stale cache aborts do not occur in AACC.
- **Scenario 4:** Client 1 wants to read object 1 on page 3. Page 3 is only present at client 2. Moreover, client 2 holds an exclusive page level lock on page 3 and it is also updating object 1 on page 3. Upon receiving the page 3 request from client 1, the server sends a callback message to client 2. Since client 2 is using the object, it sends a negative response to the server and thus client 1 blocks until client 2 does a commit.

4 Experimental Setup

The goals of the performance study are (a) to compare the performance of AACC with AOCC and ACBL for different workload and system settings and (b) better understand the performance characteristics of ACBL and AOCC. The baseline setup of this performance study is similar to many recent client cache consistency performance studies [CFZ94, AGLM95, Gru97], which were useful in validating our results.

4.1 Basic System Model

As in the previous performance studies, this study also uses a page-server client-server architecture which leaves the disk management responsibilities to the server. The clients send their object requests to the server and then cache the pages that are returned by the server. The clients use a page level data buffer and an object level log buffer. The server uses a page level buffer and also a modified object log buffer (MOB). Our notion of the MOB log buffer is the same as the one used by the previous performance studies [AGLM95, Gru97]. The two page buffers use an LRU like (second chance) buffer replacement algorithm, and the two object buffers implement a FIFO buffer replacement algorithm. The server returns pages to the clients in response to their requests. The clients return object level logs (redo/undo) to the server. These logs are stored in the modified object buffer and the server does a *redo* operation for installing these updates back onto their respective pages in the background. The use of a MOB helps the server to reduce its installation disk read operation cost for installing the client object updates [Gru97].

Input work comes to the clients as a stream of object and page identifiers from a workload generator; it comes to the server from the clients via the network. A buffer manager, lock manager, and an object manager have been modeled at both the clients and the server. The client and the server CPUs have a high priority and a low priority input queue [CFZ94]. The high priority queues are used for dealing with system requests such as disk I/O, packaging of network messages, etc. The low priority queue deals with the user requests such as lock processing, and application processing. The high priority queue is managed as FIFO while the low priority queue is managed using processor sharing among the requests. The disks have a single FIFO input queue. We use a fast disk I/O rate (for installation reads) and a slow disk I/O rate (for normal user read operations). Disks are modeled at the server but not at the clients. Similar to the previous performance studies, the LAN network model consists of a FIFO server with a specified bandwidth. In order to prevent network saturation, we ran our experiments assuming a 80Mbps network that corresponds to a Ethernet with a nominal speed of 100Mbps. The network cost consists of fixed and variable transmission costs along with the wire propagation cost. Every message has a fixed sending and receiving message cost associated with it. The size of the message determines the

variable cost component of the message.

Cost Type	Description	Value
Client CPU Speed	Instr rate of client CPU	50 MIPS
Server CPU Speed	Instr rate of server CPU	150 MIPS
ClientBufSize	Per-client buffer size	25% of DB size
ClientLogBufSize	Per-client log buffer	2.5% of DB size
ServerBufSize	Server buffer size	50% of DB size
MOB	Modified Object Log Buffer	50% of DB size
ServerDisks	Disks at server	4 disks
FetchDiskAccessTime	General disk access time	1600microsecs/Kbyte
InstDiskAccessTime	MOB disk I/O time	1000microsecs/Kbyte
FixNetworkCost	Fixed number of instr. per msg	6000 cycles
VariableNetworkCost	Instr. per msg byte	7.17 cycles/byte
NetworkBandwidth	Network bandwidth	80 Mbps
DiskSetupCost	CPU cost for performing disk I/O	5000 cycles
CacheLookup/Locking	Lookup time for objects/pages	300 cycles
Register/Unregister	Instr. to register/unregister a copy	300 cycles
ValTimeperObj	Commit validation time at server	300 cycles
DeadlockDetection	Deadlock detection cost	300 cycles
CopyMergeInstr	Instr. to merge two copies of a page	300 cycles/object
DatabaseSize	Size of database in pages	1300
PageSize	Size of a page	4 K
ObjectSize	Size of an object	100 bytes
NumberClients	Client workstations	16

Table 1: System Parameters

Table 1 lists the costs of the different operations that are considered in this performance study. These costs are similar to the ones used in previous performance studies [CFLS91, CFZ94, AGLM95, Gru97].

4.2 Workload Model

The multi-user OO7 benchmark has been developed to study the performance of object DBMSs [MDKN94]. However, this benchmark is under-specified for concurrency control studies [Car97] because it does not include the necessary data sharing patterns or transaction length values for determining the data contention level of the system. Since data contention level is an important component of any cache consistency/concurrency control performance study, the previous studies have borrowed some of the relevant features of the OO7 benchmark - the notion of a traversal, shared and private regions, small and large databases (working sets), data clustering (from the ACOB benchmark [DFMV90]) and the size of atomic objects - and added their own data sharing patterns [CFZ94, AGLM95] and transaction lengths. The data sharing pattern, in turn, dictates the number of read/write and write/write conflicts. In this study we examine *Private*, *Sh-Hotcold* and the *HiCon* data sharing patterns [CFZ94, AGLM95]. These cover a wide spectrum of data contention levels and are, therefore, useful in assessing the robustness of the cache consistency algorithms. There is no data contention in the *Private* workload, and the data contention progressively increases in the *Sh-Hotcold* and *HiCon* workloads, respectively. The database consists of a set of private regions (one for each client), a common shared region and the other region (leftover pages). The private region for a client is also considered as a hot region for the client. In the *Private* workload each client only accesses data from its private region (80 percent of the time) and the shared region (20 percent of the time). Moreover, the clients only update the data in their private regions. In the *Sh-Hotcold* workload, each client accesses the data from its private region (80 percent of the time), the shared region (10 percent of the time) and from the rest of the database including other clients' private regions (10 percent of the time). The clients can update objects in all of the regions. In the *HiCon* workload, each client accesses data from the shared region (80 percent of the time) and from the rest of the database (20 percent of the time). The clients can update objects in all of the regions. A transaction consists of many operations. Each operation of a transaction accesses many objects from a page. The page can belong to the client's hot area (area of affinity) or to the cold area. The cluster size determines how many objects of a page are accessed per operation. The cluster size being used is similar to the ones used by the previous performance studies. The cluster write probability determines whether any of the objects in an operation (cluster) will be updated. Upon accessing an object, the client can perform a read and a write operation on the object. There is a CPU instruction cost associated with the read and write operations. Upon access of an object, the object write probability determines whether an update action will be performed on the particular object. The transaction think time is the delay between the start of two consecutive transactions at the clients. The transaction size, transaction think time, database size, buffer sizes, the client hot region size, and the object write probabilities chosen in this paper are similar to the previous performance studies. These values are specified in Tables 1 and 2. When a transaction aborts, then a decision has to be made as to whether the aborted transaction accesses the same set of objects as the original transaction, or whether it should access a different set of objects [ACL87, Gru97]. In this paper, we call this the *abort variance* of a transaction. An abort variance of 100 percent means that the restarted transaction is accessing all new objects. We choose an abort variance of 50 percent since an abort variance of 100 percent favors ACBL and an abort variance of 0 percent favors AOCC. If a failed transaction accesses the same set of objects as the initial transaction, then this favors AOCC because most of the objects which will be accessed by the aborted transaction would already reside in the client cache.

Parameter	Setting
Transaction size	200 objects
Cluster Write probability	50%
Cluster size	5 - 15 objects/page
Work allocation at client	50%, 100%
Object write probability	5% to 20%
Read access think time	50 cycles/byte
Write access think time	100 cycles/byte
Think time between trans.	0
Client/Shared Hot Regions	50 pages
Network delay probability	10%
Network delay time	5msec
Abort variance	50%

Table 2: Workload Parameters

4.3 Extensions to the Experiment Setup

The experiment setups of the previous performance studies do not completely analyze the different aspects of a cache consistency algorithm. Therefore, the above mentioned experiment setup has been extended as described below:

4.3.1 Small Client Cache

The previous performance studies concentrated on only large client caches where a client's entire transaction state fits into the client cache. This is favorable to an optimistic algorithm because during abort processing almost all of the relevant objects already reside in the client cache making abort processing inexpensive. However, the large client

cache assumption is not realistic in situations where (a) the transaction size is very large, (b) one is dealing with large multimedia objects such image, video and audio and, (c) the client station buffer is shared by multiple transactions. Though the amount of memory present at client stations is steadily increasing, the application demand usually outstrips the available memory resources. Therefore, in this study, we conduct experiments with small client caches. In order to maintain the data contention level, we kept transaction and database sizes constant.

4.3.2 Network Delay Scenario

Since many of the emerging application domains operate on the Internet, it is important to assess the impact of the unpredictable network delays that are often found in wide-area networks, on the three cache consistency algorithms. Initial message delay, slow delivery and bursty arrival are the three types of delays examined in a recent WAN performance study [AFT97]. Similar to that study [AFT97], we simulate network delay by making the message sending source wait for a specified time before sending the message. The message sending source flips a coin to determine whether a message should be delayed *(delay probability)*. The actual value of the delay *(delay time)* is chosen as a multiple of the expected time to send and receive a message. In reality, the *delay probability* and *delay time* values can vary a lot depending on the network traffic, geographic location, and intermediate node down times. Due to space limitations, we present the results obtained while using only one set of network delay parameters. This is enough to assess certain key aspects of the different cache consistency algorithms.

4.3.3 Work Allocation Scenario

The previous studies only consider data-shipping cases where all of the processing is performed at the clients. As argued earlier, the current trend is to perform some of the processing at the server and some of the processing at the clients. Therefore, we consider two work allocation cases: 100 percent processing at the clients and a 50-50 split between the client and the server respectively. We do not consider query shipping to the server since this raises many new issues not addressed in this study. Since this is a cache consistency/concurrency control study, we are more interested in ensuring that the application work is performed at both the clients and the server, and less interested in the type of work that is performed at the server (such as queries or navigations). We are primarily interested in assessing the impact of work allocation (at both the client and the server) on the AOCC abort processing overhead, because a client abort has an impact on the performance of the other clients. Therefore, the work performed at the server is modeled in the same manner as the work performed at the clients. In the 50 percent client work allocation case, the transaction uses the Sh-Hotcold data access pattern at the clients and a *Uniform* data access pattern at the server. In the *uniform* data access pattern, accessed objects are uniformly distributed over the database, and client caching is not expected to be beneficial [CFLS91]. In this workload, the server has been modified so that it can manipulate objects. For sake of recovery, the server accesses are strictly read-only and an object is not accessed both at the client and the server within the same transaction.

5 Experiments and Results

The performance results reported in this section compare the performance of ACBL, AOCC and AACC cache consistency algorithms under the following scenarios: (1) Private workload with large client caches, (2) Sh-Hotcold workload with large client caches, (3) Sh-Hotcold workload using slow CPU speeds, (4) HiCon workload with large client caches (5) Sh-Hotcold workload with small client caches (6) Sh-Hotcold and Uniform workload with 50 percent work allocation at the server and 50 percent work allocation at the clients and (7) Sh-Hotcold workload with network delay. Space limitations do not allow us to report results for experiment numbers 3, 5, 6 and 7 for workloads other than Sh-Hotcold. All of the experiments use the cost and workload settings as described in Tables 1 and 2. In cases where the default values have been changed, it has been explicitly specified. Sh-Hotcold's contention level falls in between Private and HiCon workloads and most applications are likely to exhibit this level of data contention [AGLM95]. Overall system throughput in *commits/second* is the primary performance metric in this study. We verified that the 90 percent confidence intervals for our results (using batch means) are sufficiently tight. The reported results represent the steady state performance of the system.

5.1 Private

In Private workload, the clients only perform updates on their private hot regions and do not perform any updates on the shared or other client regions. Private workload is indicative of computer-aided design (CAD) environments where the users perform updates on their private data, but also do reads on shared data. Due to the absence of data contention no aborts occur in this workload. As evident in Figure 3(a), AOCC and AACC outperform ACBL for all write probabilities. In ACBL, the clients send lock escalation messages to the server to obtain page level exclusive locks for every page that is updated, and they block until the server responds. In AOCC, all the write notifications are deferred until commit time. In AACC, the *shared-private* optimization ensures that all update notifications are sent to the server in a piggy-backed manner. As evident from Figure 3(b), ACBL sends more messages than AOCC and AACC. Thus, the message transmission and the message blocking overhead of ACBL makes its performance lower than AOCC and AACC. It is also important to note that ACBL performance decreases at a faster rate than AOCC and AACC, because as the write probability increases, clients send more lock escalation messages to the server.

5.2 Sh-Hotcold

Sh-Hotcold workload data contention level is indicative of the data contention level present in most client caching applications. Therefore, its results are very important. Due to the presence of data contention, stale cache aborts are possible in AOCC, and deadlock aborts are possible in AACC and ACBL. As evident from Figure 3(c), AACC outperforms both ACBL and AOCC. Figures 3(d) and 3(e) show the corresponding abort rates and message count respectively for the three algorithms. The lower message transmission and message blocking overhead of AACC al-

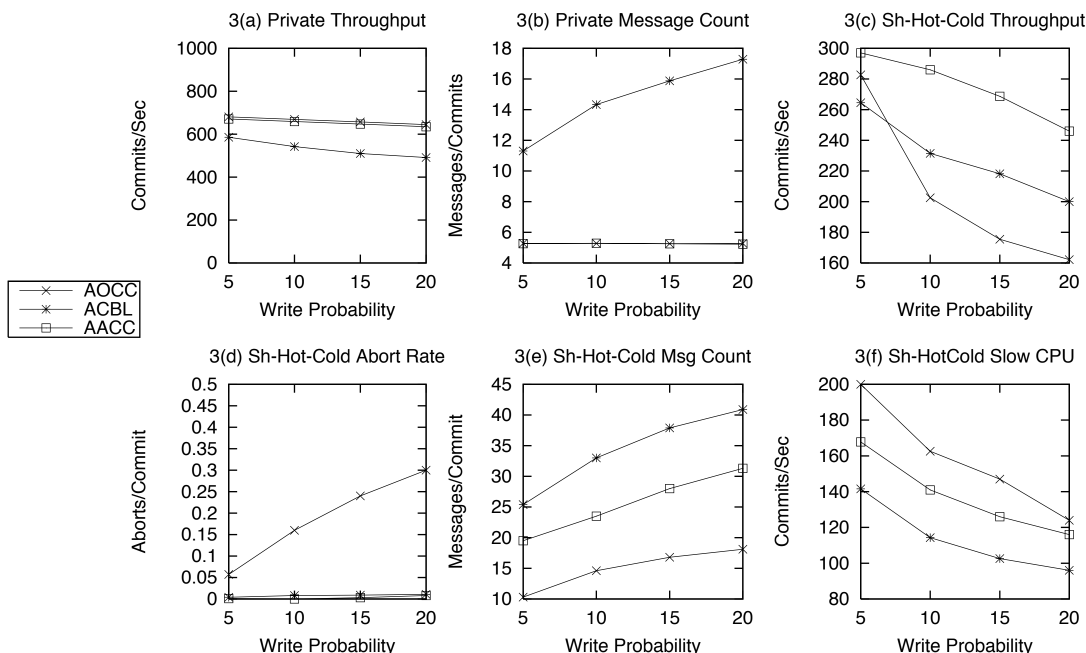

Figure 3: Private and Sh-Hotcold Results

low it to outperform ACBL. AACC and ACBL outperform AOCC, and this result seems to contradict a previous performance study [AGLM95] in which it was shown that AOCC outperforms ACBL for the Sh-Hotcold workload. However, it is important to note that the CPU speeds used in this study are faster than the ones used in the previous study. We used CPU speeds which reflect the current state of the CPU technology. Currently, CPU technology is improving at a faster rate than disk technology. Therefore, in order to study the impact of changing CPU speeds, we ran this experiment with the same slow CPU speeds of 25 MIPs and 50 MIPs as client and server CPU speeds respectively (which were used in the previous study [AGLM95]). As shown in Figure 3(f), AOCC beats ACBL and AACC. It is interesting to analyze why the change of CPU speeds causes a change in the relative performance ordering of the algorithms. First of all, fast CPUs improve the performance of all of the three algorithms in comparison to the slow CPU speeds. The reduction in transaction execution time helps in reducing the read/write conflict blocking times in AACC and ACBL. That is, since transactions execute faster, conflicting transactions block for a shorter time. Since the read/write conflict blocking rate in ACBL and AACC is higher than the abort rate in AOCC (because some transactions can sneak through), faster CPU speeds help ACBL and AACC more than they help AOCC with its faster abort processing. Fast CPU speeds also help in reducing the message processing overheads in ACBL and AACC. Another important point to note is that with fast networks, fast CPUs and large processor caches, server disks can become a bottleneck. In general, disk utilization increases much faster in AOCC than in AACC and ACBL because, in AOCC, no client blocks due to a read/write conflict. When disk utilization reaches the saturation point, this negatively affects overall performance due to increasing disk waiting times. With slow CPU speeds, the server disks become less of a bottleneck and this, in turn, reduces the disk waiting times. Therefore, we are interested in the relative speeds of CPUs and disks with respect to each other because this has an impact on the relative performance of the different cache consistency algorithms. This key result helps in better understanding the performance characteristics of ACBL and AOCC.

5.3 HiCon

In HiCon workload, the clients access the shared data region 80 percent of the time and the data region of other clients 20 percent of the time. This is a skewed data access pattern which is not usually present in data-shipping applications [CFZ94]. It is being examined here to test the behavior of the different cache consistency algorithms under extreme data contention situations. As shown in Figure 4(a), in this workload, even with client and server CPU speeds of 50 MIPs and 150 MIPs respectively, AOCC outperforms both AACC and ACBL, and AACC outperforms ACBL. However, as shown in Figure 4(b), AOCC has a higher abort rate (aborts/commits) than ACBL and AACC. One would expect algorithms with a high abort rate to perform worse than algorithms with lower abort rates. As described in section 2.4, the read/write conflict blocking rates of AACC and ACBL are higher than the abort rate of AOCC. That is, for every blocking transaction in AACC and ACBL, the equivalent situation can lead to either an abort or a commit in AOCC. Furthermore, in data-shipping client-server environments with sufficiently large client caches, the abort processing actions of a client do not have a major impact on the performance of the other clients. This is different than in centralized database systems, where a transaction's abort processing has an impact on the performance of other clients. The blocking overhead due to read/write conflicts dominates the other overheads in

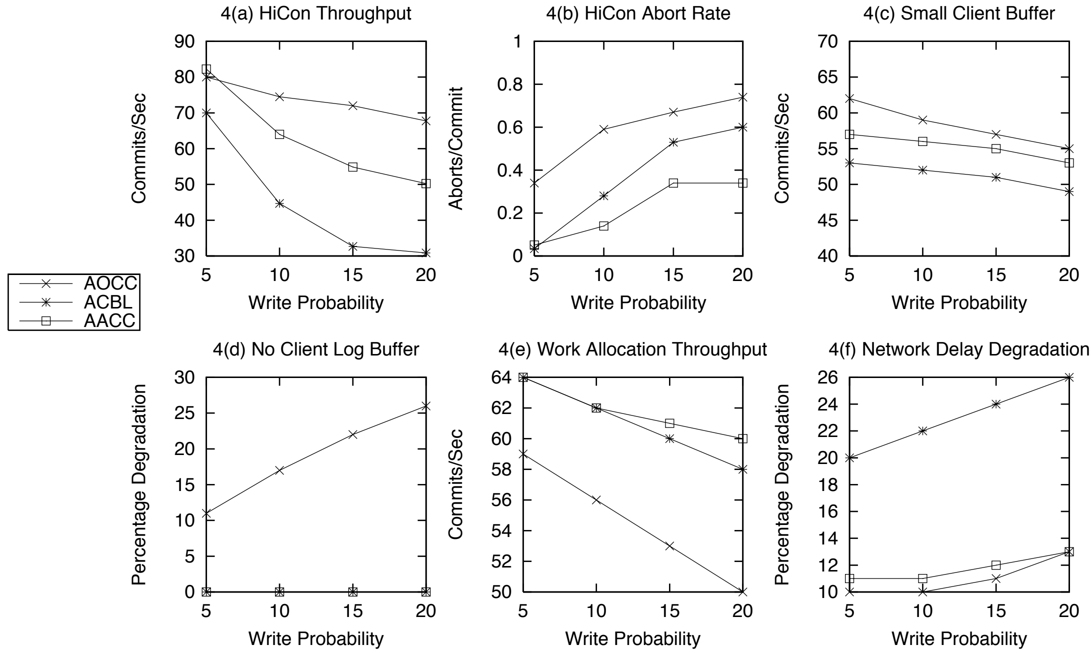

Figure 4: Hicon and System Variation Results

HiCon workloads and this, in turn, allows AOCC to outperform ACBL and AACC.

5.4 Small Client Cache

This experiment utilizes the Sh-Hotcold workload to evaluate the effects of small client cache. The client data buffer space is 25 percent of the transaction size. A small client data cache degrades the performance of all the algorithms as the clients send more requests to the server to obtain the necessary objects. In this experiment one would expect low abort rate algorithms such as AACC and ACBL to outperform AOCC because, as the transaction state does not fit into the client data cache, AOCC abort cost rises as the clients have to request the server for objects during abort processing. However, a small client data cache also has a negative impact on AACC and ACBL as it increases their read/write conflict blocking times. That is, since the overall execution time of a transaction increases (due to small client cache), blocking transactions wait for a longer time for their respective conflicting transactions to finish. The server cache and disks become more active since the client caches experience higher misses. Therefore, as evident from Figure 4(c), there is not much difference in the performance between AOCC, AACC and ACBL. However, we also ran an experiment in which clients did not have an undo client log buffer. This situation is supposed to represent situations where the client in-memory log buffer is not big enough to hold all the necessary undo log records for a transaction. The conditions of this experiment are the same as for the slow CPU speeds experiment Figure 3(f). Figure 4(d) plots the degradation in the performance (with respect to Figure 3(f)) of the three algorithms due to the lack of an undo client log buffer. One can notice that this situation negatively impacts AOCC much more than ACBL and AACC because ACBL and AACC have a very low abort rate; therefore, they rely much less on the client log buffer than AOCC. An inadequate client log buffer size forces AOCC clients to go more times to the server during abort processing. Hence, a small client data buffer hurts the performance of all three algorithms, but an inadequate client log buffer has a more negative impact on AOCC than on AACC or ACBL.

5.5 50 Percent Server Work Allocation

As evident from Figure 4(e), AACC and ACBL outperform AOCC. AOCC's performance degrades more rapidly as the write probability increases due to an increase in the number of aborts. Since application work is performed at both the client and the server, the impact of an abort is not limited to the aborting client only, but instead, it also affects the performance of all the other clients. The server CPU utilization increases as more work is performed at the server, and the server disk utilization also increases since there is no locality (due to uniform data access) in the data accesses being performed at the server. This experiment was run with CPU speeds of 50 MIPs (clients) and 150 MIPs (server) and the number of server disks was increased from 4 to 6 in order to reduce server CPU and server disk contention respectively. As already known in centralized DBMS context, medium to heavily utilized server CPUs and disks negatively impact the performance of optimistic algorithms more than they impact the performance of pessimistic algorithms [ACL87] because abort processing becomes more expensive in these situations. Moreover, it is generally more realistic to expect the server resources to be heavily utilized than under-utilized. We are using this experiment to highlight that a purely optimistic algorithm such as AOCC may not be suitable for the emerging function-shipping/data-shipping architectures. In future we intend to carry out more experiments in order to further assess the impact of cache consistency algorithms on the hybrid architecture performance.

5.6 Network Delay

Figure 4(f) shows the degradation in performance of the three algorithms for the Sh-Hotcold workload (Figure 3(c)) when a network delay is introduced. Figure 4(f) shows this degradation as a percentage of the throughput values (Figure 3(c)). As evident from Figure 4(f), the performance of all of the three algorithms degrades in comparison to an environment with no network delay (Figure 3(c)). However, it is important to note that the performance of ACBL degrades much more than the performance of AOCC and AACC because ACBL uses synchronous lock escalation messages, whereas, AACC and AOCC use asynchronous and deferred lock escalation messages respectively. In ACBL, the clients remain blocked until their lock escalation and subsequent callback messages (if necessary) are processed. Therefore, it is important to minimize the use of synchronous lock escalation and callback messages in environments, such as the Internet, with unpredictable network delays.

6 Discussion

Similar to ACBL, AACC is also an avoidance-based algorithm; therefore, AACC does not encounter stale cache aborts, but it encounters deadlock related aborts. Read/write and write/write conflicts can lead to stale cache aborts, whereas, coincidental sharing across multiple objects in a conflicting manner is required in order for deadlock related aborts to occur. In most workloads, there is a lower probability for the latter to occur. As shown in our experiments, the deadlock abort rate of ACBL and AACC is usually much lower than the stale cache abort rate of AOCC.

A key strength of AOCC is that it has a much lower messaging overhead than ACBL. This, in turn, allows AOCC to outperform ACBL. Therefore, one of the key goals of AACC is to reduce the messaging overhead, which is partially accomplished by using the concept of *shared* and *private* pages to lower the number of explicit lock escalation messages. In client caching systems, at a given point in time, many of the pages that reside in a client's cache could have been brought in by previous transactions executing at that client (inter-transaction caching). In these situations, when the client receives a callback message from the server for a page that is not used, it is not necessary for the client to immediately send a callback response to the server. Instead, the client can piggyback the page invalidation message on a subsequent message to the server. This leads to a reduction in the number of callback response messages in comparison to ACBL.

Message blocking overhead is another key factor which determines the overall performance of the cache consistency algorithms. The decision as to whether to use a synchronous, asynchronous or a deferred lock escalation message is a critical one with respect to message blocking costs. AOCC uses deferred lock escalation messages and ACBL uses synchronous lock escalation messages. Consequently, in ACBL, the clients which are performing the update operation must remain blocked until the lock escalation message and the necessary callback messages have been processed at the server and the clients. This message

SCENARIO 1

Client 1	Client 2
Read A	
Write B	
	Read B
	Write A

SCENARIO 2

Client 1	Client 2
Read A	
	Write A
Write B	
	Read B

SCENARIO 3

Client 1	Client 2
Read A	
	Write A
	Write B
Read B	

SCENARIO 4

Client 1	Client 2
Read B	
	Write A
Read A	
	Write B

SCENARIO 5

Client 1	Client 2
Read A	
	Write A
	Read B
Write B	

SCENARIO 6

Client 1	Client 2
	Write B
Read A	
	Write A
Read B	

Figure 5: Deadlock Scenarios

blocking delay increases in a heavily utilized server and network. The absence of this message blocking overhead allows AOCC and AACC to outperform ACBL.

Another important advantage of using asynchronous lock escalation messages is that it leads to fewer deadlock related aborts than what occurs with deferred lock escalation messages. Scenarios 1 and 2 of Figure 5 describe the deadlock aborts that are avoided if one uses asynchronous lock escalation messages but are possible if one uses deferred lock escalation messages. In scenario 1, an asynchronous lock escalation message prevents client 2 from reading object B, and this, in turn, prevents a deadlock. In scenario 2, an asynchronous message prevents client 2 from reading object B and this again prevents a deadlock. This is the main reason why the O2PL avoidance-based family of cache consistency algorithms [FC94] which utilize deferred messages face an increase in the deadlock rate as the data contention increases. This high deadlock rate has discouraged client caching DBMSs from using O2PL family of cache consistency algorithms [FC94]. Thus, the usage of asynchronous messages is not a compromise between using synchronous and deferred messages, but instead, it provides AACC with important advantages over ACBL and O2PL.

In order to further reduce the AACC deadlock abort rate and make it as low as the ACBL abort rate, the following two deadlock optimizations are being used in AACC:

- **Sneak-Through Deadlock Optimization:** The notion of sneak-through has been introduced in order to avoid the type of deadlocks illustrated by scenario 3 in Figure 5. Client 1 has read object A prior to that object's update by client 2. This scenario is possible since in AACC update operations never block at the time of the update even during the presence of conflicting read/write operations. The updating transaction only blocks if it reaches the commit point before the reading transaction. Therefore, client 2's update of object A will make client 2 block at commit time. If client 2 updates object B before client 1, then client 1 will normally block. In these situations, the server realizes that since client 1 is already causing client 2 to block due to its reading of object A, client 1 itself should not block on object B. Hence, the server averts a deadlock. The server maintains the information that client 1 is in sneak-through mode with respect to client 2. This sneak-through optimization helps AACC to avoid deadlocks, shown in scenario 6, which occur in ACBL.

- **Blocking Reversal Deadlock Optimization:** When the server detects a deadlock, it checks to see whether the deadlock is of the type depicted by scenario 4 in Figure 5. In this situation the server unblocks client 1 (which was blocking on object A) and instead blocks client 2 at commit time to avert a deadlock.

As our experiments have shown, the deadlock abort rate in AACC is very similar to ACBL's. However, AACC still encounters the deadlock scenario 5 (Figure 5) which is not encountered by ACBL. Finally, it is also necessary to analyze why AACC outperforms ACBL while in a previous study the NWL-Notify asynchronous algorithm performed worse than ACBL [WR91]. AACC is an avoidance-based algorithm, whereas, NWL-Notify is a detection-based algorithm. Therefore, NWL-Notify encounters stale cache aborts which do not occur in AACC. In AACC, upon the update of an object, the server invalidates the remote client caches; in NWL-Notify, the server propagates updates to the remote client caches. Invalidation has been shown to be superior to propagation for most workloads [FC94].

7 Conclusion

In this paper we presented a new cache consistency algorithm, Asynchronous Avoidance-Based Cache Consistency (AACC) algorithm which provides both good performance and low abort rate. AACC has low abort rate because it is avoidance-based; it has good performance because of its lower message processing and blocking overhead. The paper describes, in detail, the measures that are incorporated into the algorithm to reduce message, blocking and deadlock overhead. The performance study reported in this paper confirms that AACC provides significant performance gains over ACBL while maintaining a low abort rate and that it outperforms AOCC for the most common workload (Sh-Hotcold) and system configurations, while maintaining a low abort rate. These performance results are important for a number of reasons. First of all, they improve our understanding of cache consistency algorithms, in particular they reverse the commonly held belief due to a previous study [WR91] that synchronous callback locking algorithms usually outperform asynchronous algorithms. This has led to the general neglect of asynchronous cache consistency algorithms. In this paper we show that an asynchronous algorithm such as AACC can consistently outperform the best synchronous algorithm (ACBL). The second result of the performance study is that in wide area networks synchronous algorithms suffer due to increased message blocking overhead associated with unpredictable network delays. This is significant as the use of Internet widens. The third important result demonstrated by the performance study is that one does not have to tolerate high abort rates as a necessary evil to achieve high performance, as a recent comparison between AOCC and ACBL seems to suggest [AGLM95]. It is indeed possible to lower abort rates which makes the algorithm more suitable for interactive applications and for the emerging hybrid function-shipping/data-shipping architectures. In future we plan to further investigate cache consistency algorithms for hybrid data-shipping/function-shipping systems.

References

[ACL87] R. Agrawal, M. Carey, and M. Livny. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM TODS*, December 1987.

[AFT97] L. Amsaleg, M. Franklin, and A. Tomasic. Dynamic Query Operator Scheduling for Wide-Area Remote Access. Technical Report CS-TR-381, University of Maryland Tech Report, 1997.

[AGLM95] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *ACM SIGMOD Conference Proceedings*, 1995.

[Car97] M. Carey. Private Communication. 1997.

[CFLS91] M. Carey, M. Franklin, M. Livny, and E. Shekita. Data Caching Tradeoffs in Client-Server DBMS Architectures. In *ACM SIGMOD Conference Proceedings*, 1991.

[CFZ94] M. Carey, M. Franklin, and M. Zaharioudakis. Fine Grained Sharing in a Page Server OODBMS. In *ACM SIGMOD Conference Proceedings*, 1994.

[CLH97] I. Chung, J. Lee, and C. Hwang. A Contention Based Dynamic Consistency Maintenance Scheme For Client Cache. In *CIKM Conference Proceedings*, 1997.

[DFMV90] D. DeWitt, P. Futtersack, D. Maier, and F. Velez. A Study of Three Alternative Server-Workstation Architectures for OODBMS. In *VLDB Conference Proceedings*, 1990.

[FC94] M. Franklin and M. Carey. Client-Server Caching Revisited. In T. Ozsu, U. Dayal, P. Valduriez, editor, *Distributed Object Management*. Morgan Kaufmann, 1994.

[FLC97] M. Franklin, M. Livny, and M. Carey. Transactional Client-Server Cache Consistency: Alternatives and Performance. *ACM TODS*, September 1997.

[Gru97] R. Gruber. *Optimism vs. Locking: A Study of Concurrency Control for Client-Server Object-Oriented Databases*. PhD thesis, MIT, 1997.

[KF96] D. Kossmann and M. Franklin. A Study of Query Execution Strategies For Client-Server Database Systems. In *ACM SIGMOD Conference Proceedings*, 1996.

[MDKN94] M. Carey, D. DeWitt, C. Kant, and J. Naughton. A Status Report on the OO7 Benchmarking Effort. In *OOPSLA Conference Proceedings*, 1994.

[WR91] Y. Wang and L. Rowe. Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture. In *ACM SIGMOD Conference Proceedings*, 1991.