

An Active Data-aware Cache Consistency Protocol for Highly-Scalable Data-Shipping DBMS Architectures

Keqiang Wu

Peng-fei Chuang

David J. Lilja

Department of Electrical and Computer Engineering

DTC Intelligent Storage Consortium

University of Minnesota, 200 Union St. S.E. Minneapolis, MN 55455, USA

{kqw, pengfei, Lilja}@ece.umn.edu

ABSTRACT

In a data-shipping database system, data items are retrieved from the server machines, cached and processed at the client machines, and then shipped back to the server. Current cache consistency approaches typically rely on a centralized server or servers to enforce the necessary concurrency control actions. This centralized server imposes a limitation on the scalability and performance of these systems. This paper presents a new consistency protocol, Active Data-aware Cache Consistency (ADCC), that allows clients to be aware of the global state of their cached data via a two-tier directory. Using parallel communication with simultaneous client-server and client-client messages, ADCC reduces the network latency for detecting data conflicts by 50%, while increasing message overhead by about 8% only. In addition, ADCC improves scalability by partially offloading the concurrency control function from the server to the clients. An optimization, Lazy Update, is introduced to reduce the message overhead for maintaining client directory consistency. We implement ADCC in a page server DBMS architecture and compare it with the leading cache consistency algorithm, Callback Locking (CBL), which is the most widely implemented algorithm in commercial DBMSs. Our performance study shows that ADCC has a similar or lower abort rate, higher throughput, and better scalability for important workloads and system configurations. Both the simulation results and the analytic study indicate that the message overhead is low and that ADCC produces better behavior compared to the traditional server-based communication under high contention workloads.

Categories & Subject Descriptors:

H.2.4 [Database Management]: Systems – concurrency, object-oriented databases, transaction processing.

General Terms: Algorithms, Design, Management, Performance.

Keywords: active control; cache consistency; parallel communication; data-shipping; DBMS

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CF'04, April 14–16, 2004, Ischia, Italy.

Copyright 2004 ACM 1-58113-741-9/04/0004...\$5.00.

1. INTRODUCTION

In client/server DBMS architectures, there are two categories of systems, *query-shipping* and *data-shipping*. In query-shipping systems, such as a relational client/server DBMS, clients send a query to the server, which processes the query and sends the results back to the clients. In contrast, most commercial object-oriented database management systems (ODBMS) have adopted the data-shipping technique. When a client wants to access data, it sends a request for the specific data items (e.g. objects or pages) to the server. The data items are shipped from the server to the clients, so that the clients can run applications and perform operations on cached data. More detailed information on the difference between these two DBMS architectures is available in [8].

The data-shipping architecture is inspired by the dramatic improvements in computer price-performance and in the performance and availability of network communication. The technology advances have made it desirable and practical to offload more functionality from the server to the client workstations [13]. Typically, data-shipping systems can be structured either as page servers, in which clients and servers interact using physical units of data (e.g. pages or groups of pages), or object servers, which interact using logical units of data (e.g. objects). The granularity of the data transferred from the server to the clients is the fundamental difference between page and object servers.

Having the data available at the clients can reduce the number of client-server interactions to thereby free server resources (CPU and disks), thus decreasing client-transaction response time. Local caching allows copies of a database page to reside in multiple client caches. Moreover, when inter-transaction caching is used, a page may remain cached locally when the transaction that accessed it has completed. Concurrency control for cached pages must be enforced to ensure that all clients have consistent page copies in their caches and that they see a serializable view of the database.

Research on cache consistency protocols for data-shipping database systems has been motivated by the inherently distributed nature of the advanced applications that DBMSs support, e.g. electronic commerce, distributed multimedia applications, etc. The primary disadvantage of current protocols [1][3][9][14][18][19] is the server-based communication path. In other words, whenever a client needs permission to update a cached data item, it must always send a request to the server before the transaction commits. Before granting the request, the server must issue callback requests to all sites (except the requester) that hold a cached copy of the requested data. Since the server is the only source for enforcing cache consistency, two potential drawbacks arise:

- (1) It is well known that the earlier discovery of data conflicts can improve performance and lower the abort rate/cost [9]. However, since the server is always on the critical path, this unavoidably increases the communication latency for detecting data conflicts.
- (2) Excessive demands for pages from the server often limit the performance and scalability of traditional caching algorithms, such as CBL [7][8][17]. As the power of client workstations has increased dramatically, it is more desirable to transfer additional processing to the client machines [13]. The centralized server design limits this transition.

To address these problems, we propose an efficient client cache consistency algorithm, *Active Data-aware Cache Consistency* (ADCC), which employs both client-client (P2P) and server-based communications. Via a two-tier directory, ADCC allows not only the server but also the clients to track the global state of cached data. An optimization, Lazy Update, is introduced to reduce the message overhead for maintaining the client directory consistency. The primary contributions of this paper are:

- (1) *It proposes an algorithm, Active Data-aware Cache Consistency, that decreases the latency for detecting data conflicts by 50%, while increasing message overhead by only about 8%.*
- (2) *This algorithm significantly reduces the potential interval for write/read and write/write conflicts.*
- (3) *This algorithm relieves the server bottleneck by partially offloading some functionality for concurrency control (such as callback) from the server to the clients.*

Since non-adaptive callback schemes are the best overall choices of existing schemes [8], we compare ADCC with the leading cache consistency algorithm, Callback Locking (CBL), which is the most widely implemented algorithm in commercial DBMSs. Similar to previous work [9], this study focuses on a page-server architecture with page level consistency, since a page-server architecture usually provides superior performance over an object-server architecture [13]. While security and robustness are important issues, they are beyond the scope of this work, which focuses primarily on performance.

The remainder of the paper is organized as follows. Section 2 describes the proposed ADCC algorithm. Section 3 describes the experimental setup. Section 4 presents the simulation results and discussion. Section 5 describes related work. Finally, Section 6 summarizes and concludes.

2. AN ACTIVE DATA-AWARE CACHE CONSISTENCY (ADCC) ALGORITHM

Before describing ADCC, we give a brief review of CBL [8][9]. In CBL, clients can cache data across transaction boundaries. Locally cached page copies are always guaranteed to be valid, so clients can read them without contacting the server. However, clients need to obtain write permission from the server before they can proceed with a write operation. A server typically maintains a table to keep track of which clients have locks (read or write) for a page. If the page is cached at other clients, the server sends callback messages to other clients asking them to downgrade or relinquish their locks. Before receiving the acknowledgement from all related clients for

its callback request, the server cannot grant write permission to the requesting client.

There are two major CBL variants, CBL-R and CBL-A. In CBL-R, write permissions are granted only for the duration of a single transaction. In CBL-A, in contrast, write permissions are retained at the clients until being called back, or until the corresponding page is dropped from the cache.

In contrast to CBL, ADCC allows not only the server but also the clients to maintain a directory for each cached page. The directory for a page is organized as a bit vector of p presence flags (p is the total number of clients that cache the same page), indicating which client has a copy of that page in its cache, together with the state information. The related directory information is tagged with the data page and sent to the requester. Figure 1 shows a schematic of the directory structures.

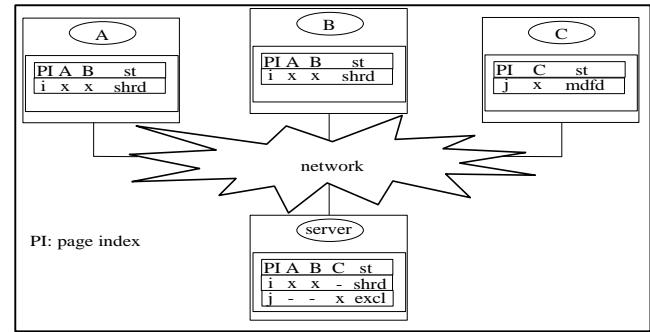


Figure 1. Schematic of the client and server directories. Page i is cached at both A and B, but not C. Page j is cached only at C.

2.1 Client States and Protocol Descriptions

A page in a client (e.g. client A) buffer may be in one of five valid states. Three of these are stable states: (1) *exclusive-clean (excc)* - only one unmodified (read-only) cached copy in the clients; (2) *shared (shrd)* - more than one read-only cached copy whose whereabouts are indicated by the presence vector; (3) *modified (mdfd)* - only one modified page is cached. The transition states are: (4) *busy (bsy)* - the transaction is in the process of updating the data; (5) *speculative-busy (sbsy)* - which is similar to the *busy* state except that it implies the update is speculative.

When client A updates a cached page, it changes the page state to one of the transition states and will not change the page state to a stable state until the transaction has committed or aborted. When a page is in one of the transition states, the client ignores any invalidation requests for this page from peers. The *speculative-busy* state converts to the *busy* state if the server grants the speculation, or the transaction aborts if the server finds that the speculation is incorrect. Figure 2 shows the simplified state transition diagram for a page in client A's buffer.

i) Handling A's read requests

If the desired page is in its local cache, client A accesses it without contacting the server. If the page is not in its cache (represented by *inv* in Figure 2), it sends a read request for the page to the server (shown in Figure 3). When the page first arrives at A, depending on the tagged directory information of the previous state of this page, A

sets the state as *exclusive-clean* or *shared* (the state transits from *inv* to *excc* or *shrd* in Figure 2).

ii) Handling A's write requests

If a page is in A's cache in the *exclusive-clean* or *modified* states, A changes the client page state to *busy* and updates the page without contacting the server. If the cached page is in the *shared* state, A continues the update, changes the state to *speculative-busy*, and sends a speculation request to the server and invalidation requests to the other sharing clients. When A receives an acknowledgement message (*ack*) from the server, A changes the page state to *busy*. A is not allowed to commit until it has received an *ack* message from all necessary clients. If the speculation is incorrect, A aborts the transaction. If the state is *busy* or *speculative-busy*, client A ignores any invalidation requests for that page from the other clients. Any race between multiple speculative updates is resolved at the server. If the page is not in A's cache, client A sends a write request for the page to the server (shown in Figure 3). When the page first arrives at A, A sets the state as *busy* and continues the update (the state transits from *inv* to *bsy* with *Sdata/Awr* in Figure 2).

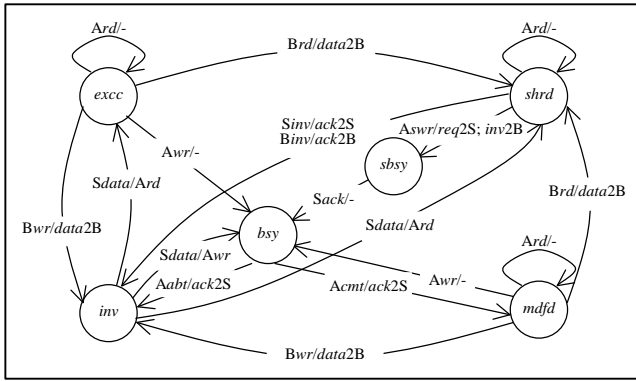


Figure 2. State transition diagram for a page in client A's buffer. To show the use of busy states, interactions among two clients and the server are shown. For simplicity, scenarios with conflict transactions are not shown. The abbreviations inside the circles denote a page state in A's buffer. The notation X/Y means if message X is received, then in addition to the state change, action Y is generated. "-" means no action. If multiple X/Y pairs are associated with an arc, it means that multiple inputs can cause the same state transition. "x2y" means that client A sends x to y. "A", "B" and "S" denote client A, client B and server, respectively. "swr", "abt" and "cmt" denote speculative write, abort, and commit, respectively. For example, "Brd/data2B" denotes that, when A receives a read request from client B for a page in the *exclusive-clean* or *modified* states, A forwards the data message to B and changes the page state to *shared*; "Aswr/req2S; inv2B" denotes that, when A wants to update a locally cached page which is also cached by B, A speculatively continues the update, changes the page state from *shared* to *speculative-busy*, sends a speculation request to the server, and sends an invalidation request to B. "Sdata/Ard" denotes that, when a data page which was requested by A arrives at client A from the server, depending on the original page state in the server buffer, A changes the page state to *exclusive-clean* or *shared* and conducts the *rd* operation.

iii) Handling data requests from other clients

If client A is the *exclusive-owner* of a page, A may receive a data request which requires A to provide the data page to client B. If B's intended operation is read and A is not updating the data, client A forwards the page to B, and changes the page state to *shared*. If B's intended operation is a write, then A forwards the page to B and invalidates its copy if there is no conflict. Otherwise, A forwards the page to B and informs the server of the conflict if A has read the page, or blocks the request and informs the server of the conflict if A has updated the page. If there is a conflict, B is not allowed to commit before A. These conflict scenarios are not shown in Figure 2, but are discussed in Section 2.3.

iv) Handling invalidation requests

If client A is one of the sharing clients of a page, A may receive an invalidation request from another sharing client B or the server because of a speculative or non-speculative update. If A's transaction has read the page but not committed yet, A blocks the request and informs the server of the conflict. The remote client is not allowed to commit before A.

2.2 Server States and Protocol Descriptions

A page in the server buffer may be in one of five states. Three of these are stable states: (1) *unowned* - no cached copies in the clients; (3) *exclusive* - one read or write cached copy is in a client, indicated by the corresponding presence flag; (2) *shared* - two or more read-only cached copies whose whereabouts are indicated by the presence flags. Two of the states are transition states: (4) *busy-shared* and (5) *busy-exclusive*, which correspond to read or write requests that might still be in progress, or the related transaction has not committed yet. The server directory also maintains an entry indicating which client is busy if the page is in a transition state. The transition states are used to avoid race conditions and to provide serialization. If a request for a page in the transition state arrives at the server, the server either blocks the request or informs the requesting client to abort if blocking would lead to a deadlock.

An *exclusive* directory state in the server means that the page may be in either the *modified* or *exclusive-clean* states in a client buffer. If a client requires a page which is cached exclusively by another client, the server forwards the request to the exclusive owner. On the other hand, the exclusive owner is allowed to modify the page without contacting the server. A simplified state transition diagram for pages in a server's buffer is shown in Figure 3.

i) Handling data requests for read operations

When the server receives a page read request from client C, it checks the page state and owner information and responds to these requests as follows, given these possible states.

- *unowned*: the server returns the page and sets the state to *exclusive*.
- *exclusive-B*: the server forwards the request to client B, changes the state to *busy-shared*, and sets C to the busy client (we use the shorthand, *busyShared-C*). This shorthand means that the server thinks client C is in the process of receiving the page which is being provided directly by the previous exclusive owner. After receiving an *ack* message (via a piggyback message) from C indicating that it has received the data, the server changes the state to *shared-C-B*.

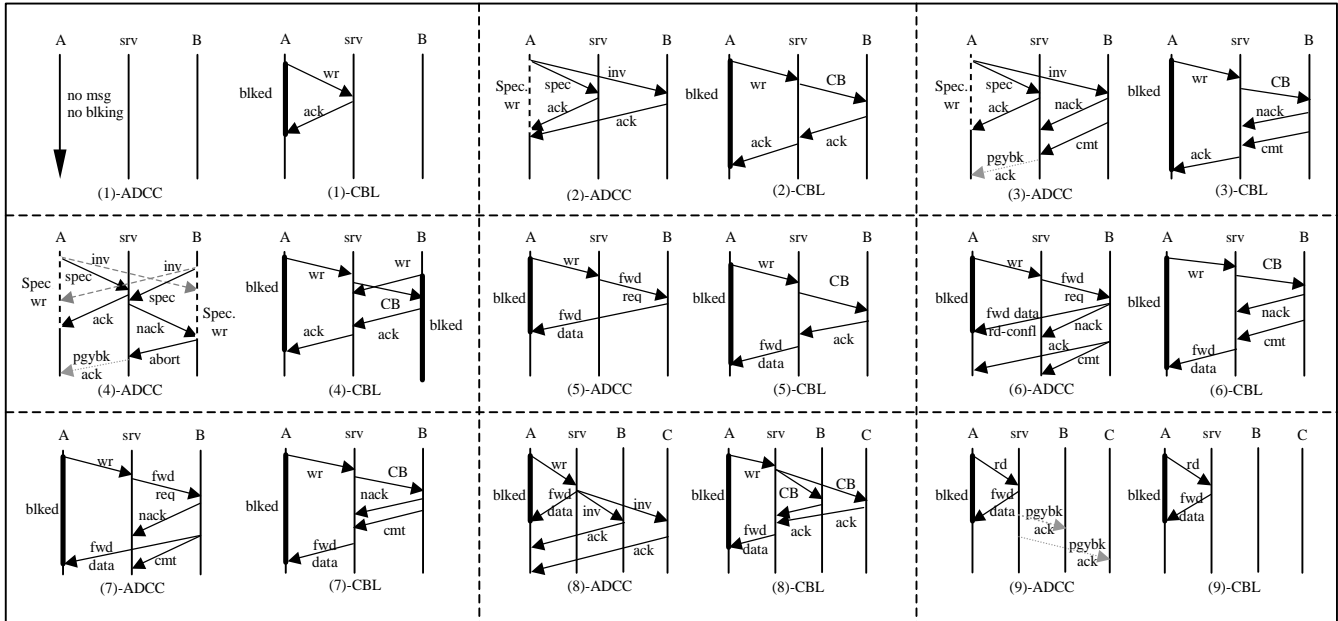


Figure 4. ADCC cache consistency scenarios that involve the server (srv) and three clients (A, B and C). Each arc denotes a message. A dashed arc indicates that the related request will be ignored by the recipient and a dotted arc denotes that the related message is piggybacked with a subsequent message. Each scenario is compared with CBL. Note that more messages are handled in parallel in ADCC than in CBL.

server grants (ack) a write lock to A. A and B have to block until the server responds. The server will not grant a write-lock to B until A commits.

Scenario 5: Page *m* is cached only by B, but B is not using it. A wants to update it. In ADCC, A sends a request (wr) to the server. The server forwards the request (fwd req) to the exclusive owner, B, which forwards the data (fwd data) directly to A and invalidates its own copy.

In CBL, the server behavior is similar to Scenario 2, except that the server forwards the page (fwd data) to A after B responds (ack). Again, A blocks before the server responds.

Scenario 6: Page *m* is cached only by B. B has read it but the transaction has not committed. A wants to update it. In ADCC, A sends a request (wr) to the server, which is forwarded to B (fwd req). B informs the server (nack) and A (rd-confli) of the read conflict, but still forwards the page (fwd data) to A immediately. However, A is not allowed to commit until the commit acknowledgement (ack) from B arrives.

In CBL, the server behavior is similar to Scenario 3, except that the server forwards the page to A after B commits. Again, A blocks while waiting for the server to respond.

Scenario 7: A wants to update page *m* which is exclusively owned by B. B has updated the page, but has not committed it yet. In ADCC, this scenario is similar to Scenario 6, except B does not forward the data to A until B has committed.

In CBL, the server behaves the same as in Scenario 6.

Scenario 8: Page *m* is cached by both B and C, but neither uses it. A wants to update it. In ADCC, when the server receives A's

request (wr), it provides the data (fwd data) to A immediately and sends invalidation requests (inv) to B and C. A unblocks after the page arrives. However, A is not allowed to commit until the invalidation acknowledgements (ack) from both B and C arrive.

In CBL, the server does not grant a write lock to A until both B and C invalidate their copies. A blocks until the server responds.

Scenario 9: Page *m* is cached by both B and C, but neither is using it. A wants to read it. In ADCC, when the server receives A's request (rd), it provides the data (fwd data) to A immediately and piggybacks the acknowledgement (pgybk ack) of having A as a new sharer on subsequent messages to B and C.

In CBL, the server responds to A immediately.

2.4 Deadlock Solution

Similar to CBL, we use a centralized approach for handling deadlocks. The server maintains a Conflict Log Table (CLT) to keep track of conflicts. When a client commits or aborts, the server removes all entries in the CLT related to that client. When the server receives a nack response from a client, or when the server receives a read or write request for a page whose server page state is *busy-shared* or *busy-exclusive*, the server performs deadlock detection.

When the server receives a nack response (for example, Scenarios 3, 6 and 7 in Figure 4) from client B indicating that there is a conflict on a request of page *m* from client A, it checks the CLT. If there are no deadlocks, the server adds an entry into the CLT, indicating that A is blocked by B for the request of page *m*. Consequently, A cannot commit before B.

The *busy-shared* state occurs under the following scenario. When the server receives a read request from client A for page m which is only cached by client B, the server forwards the request to B, changes the directory state to *busy-shared* and sets the *busy-client* to A. We use the shorthand, *busy-shared-A*, to indicate that the server thinks A is in the process of receiving the page. The page state remains in *busy-shared-A* until A acknowledges (via a piggyback message) the server that it has received the requested page from the original exclusive owner. If the server receives a read or write request for page m from another client, say C, before the acknowledgement from A arrives, the server first checks in the CLT if A (the busy client) is blocked by B (the original exclusive owner). If A is not currently blocked by B, the server simply defers the request from C until the server page state turns to stable. Otherwise, the server conducts deadlock detection for C being blocked by B. If there are no deadlocks, the server updates the CLT with a new entry, i.e. C's request on page m is blocked by B. The server will not handle this request from C until B has committed.

The *busy-exclusive* state occurs under all scenarios in Figure 4 except Scenarios 1 and 9. When the server receives a write request from A, the server changes the page state to *busy-exclusive* and sets A as the busy client. The server keeps this busy state until A commits or aborts. If the server receives a read or write request from client D for page m , the server checks the CLT to see whether D being blocked by A leads to a deadlock. If there are no deadlocks, the server updates the CLT with a new entry, indicating that D is blocked by A on the request of page m . The server will not handle this request from D until A has committed.

If the server detects a deadlock (a cycle in the wait-for graph), it informs the requester to abort. To avoid starvation, we ensure that clients can be picked for abortion only a small finite number of times [17].

2.5 Committing Process

At commit time, the client sends the logs to the server, changes the state of updated pages from *busy* to *modified*, and then activates the other client requests that are waiting for this client to commit.

When the server receives a commit message, it removes those entries in the CLT related to the committing transaction, changes the *busy-exclusive* state of those updated pages (with *busy-client* as the committing client) to *exclusive*, moves the logs on to a persistent storage area, and activates the other client transactions that are waiting for this client to commit.

2.6 Lazy Updates of the Client Directory

A potential drawback of ADCC is the additional messages used to maintain client directory consistency. The server needs to inform the necessary clients to update their directories when a client has removed its cached copy or has just cached a copy. An optimization used in ADCC is to have the server inform the related clients via a *piggybacked* message. The inherent message delay due to the piggybacking may cause situations where some clients have an inconsistent directory relative to the server for the same page. As a result, those clients may have outdated presence flags in their directories. However, the client page states are still *shared*, i.e. clients B and C in Scenario 9 may not know that A has cached page m . Such directory inconsistency causes a problem only when those clients want to update the page. However, as described in Section 2.3 (Scenarios 3 and 4), such updates are speculative. When the

server receives a speculative update request for a shared page, the server compares its directory with that of the client. If the server detects that the client directory is outdated, it grants the speculation, but at the same time it informs the client of the discrepancy. If there are some new sharing clients, the speculative client is not allowed to commit before the new sharing clients invalidate their copies.

2.7 Quantitative Comparison with CBL

Table 1 compares ADCC with CBL for the nine scenarios in Figure 4. In this table, *Par* denotes “partially”, *# ttl msg* means the total number of messages generated, *blocked* denotes whether the client has to wait when it wants to perform an operation, and *# srv msg* denotes the total number of messages sent and received by the server. The number in parentheses denotes the number of sequential steps in the longest communication path. In Scenario 3, for example, the longest path is between A and B in ADCC. A sends an *inv* to B (1st step). After receiving the *inv*, B sends a *nack* to the server (2nd step). After B commits, it sends a message to the server (3rd step). The server piggybacks this information on a subsequent message to A (4th step). Piggybacking does not generate an additional message. In CBL, in contrast, all of the five messages have to be sent and received in that particular order. The sending/receiving of the messages constitutes the five sequential steps shown in the table for CBL. Note that the number of sequential communication steps for ADCC is always less than or equal to the number of communication steps needed for CBL.

TABLE 1. Comparing the Cost of ADCC with CBL for the Nine Scenarios in Figure 4.

Scenario		1	2	3	4	5	6	7	8	9
ADCC	# ttl msg	0(0)	4(2)	5(4)	7(4)	3(3)	6(4)	5(4)	6(3)	2(2)
	blocked	No	No	No	No	Yes	Par	Yes	Par	Yes
	# srv msg	0	2	4	5	2	4	4	4	2
CBL	# ttl msg	2(2)	4(4)	5(5)	5(4)	4(4)	5(5)	5(5)	6(4)	2(2)
	blocked	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	# srv msg	2	4	5	5	4	5	5	6	2

Compared with CBL, ADCC provides several advantages: (1) ADCC reduces the number of sequential communication steps in seven scenarios; (2) it removes blocking for four scenarios, and reduces the blocking time for two scenarios; and (3) it typically reduces the total number of messages that are sent and received by the server (except in Scenarios 4 and 9), as the server is partially removed from the critical path.

In Scenario 6, ADCC requires an additional control message compared to CBL but it reduces the blocking time.

If T is the average one-way trip delay for a message between the client and the server or two clients, then in Scenario 4, it takes time T in ADCC, but $2T$ in CBL, for B to be aware of A's write intention. Although ADCC requires two additional messages, it reduces the latency for invalidation requests by 50%. As a result, ADCC reduces the potential conflict interval by approximately 50%. In addition, the two additional messages are small control messages. More importantly, most messages in ADCC are handled in parallel so that the critical path in ADCC is no longer than in CBL. ADCC requires more messages than CBL as the number of clients involved increases. However, the analysis in Section 4 shows that the expected message cost in ADCC is actually similar or lower than CBL for generic write/write conflicts.

It is worthwhile discussing a scenario for read requests which is not included in Figure 4. When client A wants to read page m which is exclusively owned by client B but not in use by B, in ADCC the server forwards A's request to B and B then forwards the page to A and changes the page state to *shared*. ADCC requires 3 messages (2 control + 1 data). In contrast, CBL performs similar to Scenario 9 or Scenario 5, depending on if B has a read or write lock for page m , which generates 2 (1 control + 1 data) or 4 (3 control + 1 data) messages. CBL might outperform ADCC for this type of read operation. However, when the page is cached by more than one client, this advantage disappears. Similar to the design of CBL-R and CBL-A, whether to allow the exclusive owner to keep the write permission is a design choice. Since we focus on concurrency control under data sharing, the design in ADCC brings significant benefits for write requests, such as in Scenario 1. This approach also partially offloads the server from serving data, which is similar to the data forwarding used in global memory management [7].

In general, ADCC reduces the network latency significantly but consumes slightly more network bandwidth due to additional control information (256 byte) and related directory information tagged with data messages. Compared to the typical message size of a data page (≥ 4 Kbyte), the bandwidth overhead is generally low. In addition, network technology has evolved from 10 Mbps in the early 1990s to Gigabits today. The bandwidth utilization in today's Gigabit networks is typically low [10]. A recent study shows that, with 2 GB networks, typically less than 20% of the peak bandwidth is used [15].

3. A PAGE SERVER DBMS MODEL FOR PERFORMANCE EVALUATION

3.1 The System Model

We use a simulation model similar to those used in previous client cache consistency performance studies [1][3][8][9][14][18][19], as depicted in Figure 5. The model consists of a single server and a varying number of client workstations, which are connected via a network. The number of clients is a parameter of the model. Each client node consists of: (1) a transaction generator, which submits transactions to the client one after another, (2) a buffer manager, which manages the buffer pool using an LRU page replacement policy, (3) a transaction manager, which coordinates the execution of client transactions, (4) a concurrency control manager, which implements consistency management functions and is algorithm dependent, and (5) a resource manager, which models CPU activity and provides access to the network. Transactions themselves are each modeled as a string of page references (i.e. page reads and writes) with a unique transaction ID. If a transaction aborts, it restarts the same transaction with a new ID. After a transaction commits, a new transaction is submitted after a specified thinking period.

The server model is similar to that of the clients except that the work for the server always arrives via the network. The resource manager models disk activity as well as CPU activity and network access. The server's transaction manager coordinates the server's operation based on the stream of incoming client requests.

The network manager models communication among the clients and server as a FIFO server with a specified bandwidth. The communication latency consists of a fixed CPU overhead for protocol processing at both the sending and receiving sites per

message and a variable transmission delay on the network. To avoid network saturation, we simulate a network with the actual load at 80% of the nominal bandwidth. Similar to [2], the unpredictable network delay is modeled by making the message provider wait for a specified time before sending the message. The delay probability and time are specified similar to [14].

The simulated CPUs employ a two-level priority scheme for input queues. System requests, such as disk I/O and packaging of network messages, are given priority over client transaction requests. The high priority queue is managed as a FIFO queue and the low priority queue is managed using processor sharing among the user requests. Each disk has a FIFO queue of I/O requests, and the disk for each request is chosen uniformly from all of the server's disks. Disk access times are drawn from a uniform distribution between a specified minimum and maximum.

Table 2 describes the parameters that are used to specify the costs of the different operations and the system configuration. These parameters are similar to the ones used in previous performance studies [9].

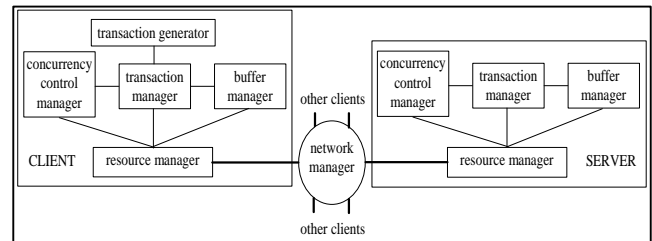


Figure 5. Model of a page-server client-server database management system.

TABLE 2. Parameters Used in the Simulation Study.

System Parameter		
pageSize	Size of a page	4 Kbyte
databaseSize	Size of database in pages	2000
numClients	Client workstations	1 to 40
client CPU speed	Instruction rate of client CPU	50 MIPS
server CPU speed	Instruction rate of server CPU	200/400 MIPS
clientBufSize	Per-client buffer size	5% of DB size
serverBufSize	Server buffer size	50% of DB size
serverDisks	Number of disks at server	4 disks
minDiskAccessTime	Minimum disk access time	4 millisecond
maxDiskAccessTime	Maximum disk access time	12 millisecond
Overhead Parameters		
fixedMsgInstr	Fixed num of instr per msg	20000
perByteMsgInstr	Num of addl instr per msg byte	4
networkBandwidth	Network bandwidth	80/800 Mbps
networkDelayProb	Probability of delaying msg	10%
networkDelayTime	Average time a msg is delayed	1 msec
lockInst (CBL)	Instr per lock/unlock pair	300 instr
controlMsgSize	Size in bytes of a control msg	256
dirLookupInst	Instr per directory lookup/setup	600 instr
readAccessTime	CPU instr cost for RD operation	50 instr/byte
writeAccessTime	CPU instr cost for WR operation	100 instr/byte
diskOverheadInstr	CPU overhead to perform I/O	5000 instr
thinkTime	Delay for submitting a new trans	0

3.2 The Workload Model

The multi-user OO7 benchmark has been developed for object DBMS performance studies [4]. However, this benchmark is under-specified for cache consistency studies because it does not include the necessary data sharing patterns or transaction sizes for

determining the data contention level of the system [5][9]. Similar to [8][9], we examine UNIFORM, HOTCOLD, PRIVATE, and HICON data sharing patterns. These cover a wide spectrum of data contention levels and are useful in assessing the robustness of the cache consistency algorithm [8][9]. Table 3 summarizes the workloads that are examined in this paper.

In these workloads, transactions are represented as a string of page reference requests in which some are for reads and the others are for writes. There is a CPU instruction cost when a client performs a read or write operation. The database consists of a set of hot regions (one for each client), and a cold region. The hot region for a client is also treated as a private region for the client. The probability of an access to a page in the hot region is specified; the remainder of the accesses are directed to cold region pages. For both regions, the probability that an access to a page in the region will involve a write (in addition to a read) is specified.

The UNIFORM workload has no per-client locality and a high level of data contention. Each client accesses the data uniformly throughout the whole database. The HOTCOLD workload has a high degree of locality per client and a moderate amount of sharing and data contention among clients. Each client accesses the data from its private region (80% of the time) and the cold region (20% of the time). The clients can update pages in both regions. PRIVATE is a CAD-like workload with the highest per-client data locality and no contention. The only inter-client sharing in the workload involves read-only data. HICON is a skewed workload, which is unlikely in client-server DBMS environment. Nevertheless, it is typically used to expose the performance tradeoffs in a very high contention environment.

TABLE 3. Workload Parameter for Client i .

Parameter	UNIFORM	HOTCOLD	PRIVATE	HICON
transSize	20 pages	20 pages	16 pages	20 pages
hotBounds	-	p to $p+49$, $p=50(i-1)+1$	p to $p+24$, $p=25(i-1)+1$	1 to 400
coldBounds	All of DB	Rest of DB	1001 ~ 2000	Rest of DB
hotAccProb	-	0.8	0.8	0.8
coldAccProb	1.0	0.2	0.2	0.2
hotWrProb	-	0.2	0.2	0.25
coldWrProb	0.1/0.2/0.4	0.2	0.0	0
perPageInstr	30,000	30,000	30,000	30,000
thinkTime	0	0	0	0

4. RESULTS AND DISCUSSION

We use the cost and workload settings described in Tables 2 and 3 to obtain the following results. This paper uses CBL-A for the PRIVATE workload comparison since CBL-A performs slightly better than CBL-R for this workload [9]. For the other three workloads, we use CBL-R as the basis. The large client cache size assumption is not realistic in situations where the transaction size is very large or the client workstation buffer is shared by multiple transactions [14]. Consequently, we use a small client cache (5% of the active database size). We examine the impact of the relative gap among client/server CPU performance, server disk I/O performance, and network bandwidth under different workloads on the overall performance and scalability. Similar to previous studies [1][3][8][9][14][18][19], the system throughput (transactions per second) and abort rate (aborts per commit) are the major performance metrics in this paper. To ensure the statistical validity of the results, the 90 percent confidence intervals for system throughput in commits/second were calculated using batched

means. The confidence intervals were within a few percent of the mean. Each experiment was run ten times using ten different random number seeds.

4.1 The UNIFORM Workload

The UNIFORM workload has no per-client locality. Consequently, it does not benefit much from caching. As the fundamental difference between ADCC and CBL or the other protocols is P2P communication under the write/read and write/write data sharing, we first examine the impact of P2P communication under different levels of contention with the server CPU speed and network bandwidth as 400 MIPS and 80 Mbps, respectively. This configuration prevents the server/client CPU and network bandwidth from becoming a bottleneck as the client population increases. Therefore, we can focus on the impact of different levels of contention.

Figure 6 shows that ADCC outperforms CBL for all configurations for this workload. When the contention increases, the throughputs of both protocols drop, but the gap between ADCC and CBL increases. As the client population increases from 8 to 40, the throughput gap ranges from 5% to 17% under low data contention (coldWrProb=0.10), changes from 11% to 21% under medium data contention (coldWrProb=0.20), and varies from 15% to 54% under high data contention (coldWrProb=0.40).

This performance trend can be explained from several aspects. First of all, when write/read and write/write sharing exists, the increasing client population intensifies the data contention, since the number of cached copies for any given page increases. This contention increases the overhead of client caching, i.e., the communication latency for callback when a client wants to update a page which is also cached at other sites. Compared with the traditional server-based communication path in CBL, ADCC reduces this overhead using the direct P2P communication. Secondly, CBL relies on the server for callback handling, while ADCC partially offloads this function from the server to the clients. When the contention increases, the server in CBL has to handle more lock callbacks, while in ADCC, many more invalidations occur directly between the clients. Therefore, the server is more heavily loaded in CBL than in ADCC. When the contention is high, increasing the client population causes the performance to degrade further.

The P2P communication path in ADCC also reduces the potential conflict interval. Consider the example of Scenario 2 in Figure 4, assuming that page m is not in use until A wants to update it at time t . On average, the invalidation request from A in ADCC reaches B at $(t+T)$, while the corresponding callback message in CBL arrives at B at $(t+2T)$. Therefore, in ADCC if B accesses the data during the interval $[t, t+T]$, A's update will be blocked. However, this interval becomes $[t, t+2T]$ in CBL. Figures 7 and 8 show the block rate and abort rate for the two protocols, which confirms that the reduction in the potential blocking window by ADCC produces less contention than CBL.

Figure 4 shows that a significant portion of messages are handled in parallel in ADCC but sequentially in CBL. Due to the different overheads associated with the sequential and parallel message sending, the total number of messages sent is not an accurate metric for comparing the performance of these two schemes. Figure 9 shows that the highest contentious level of the UNIFORM workload (coldWrProb=0.4) results in ADCC sending approximately 8% more control messages (256 byte) than CBL across the range of the

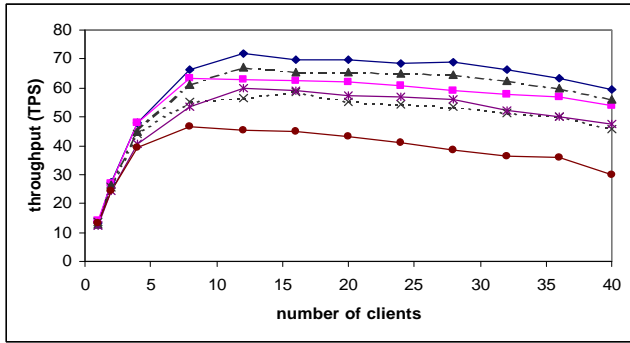


Figure 6. Throughput (UNIFORM).

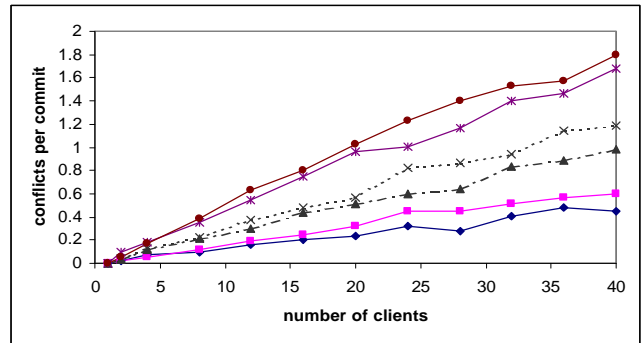


Figure 7. Conflicts per commit (UNIFORM).

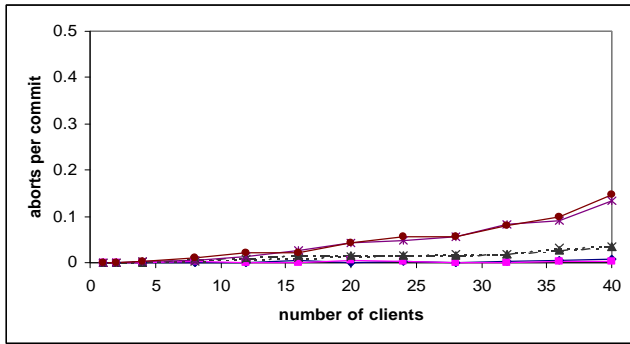


Figure 8. Aborts per commit (UNIFORM).

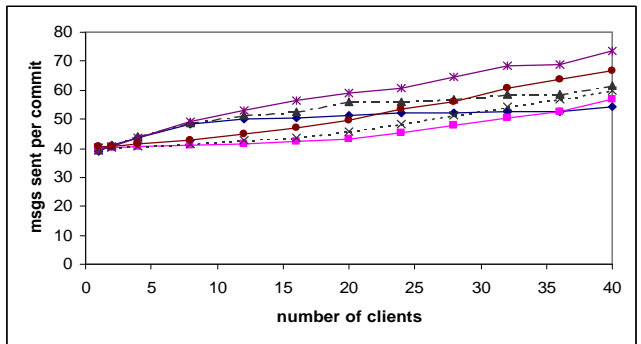
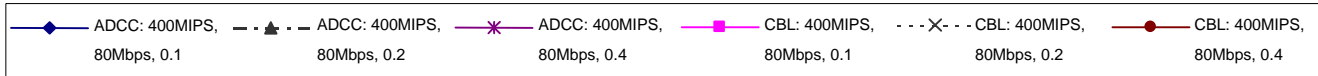


Figure 9. Messages per commit (UNIFORM).



client population. Compared to the typical message size of a data page (≥ 4 Kbyte), this message overhead is low. Despite of this overhead, the shorter communication path make ADCC outperform CBL.

From the above discussion, it can be seen that ADCC improves both performance and scalability compared to CBL. The increased data contention in the UNIFORM workload makes P2P communication an important aspect of producing these improvements.

4.2 The HOTCOLD Workload

The HOTCOLD workload has high per-client locality and moderate write/read and write/write sharing among the clients. Due to the presence of data contention, deadlock aborts are possible. By adjusting the server CPU and network bandwidth, we examine the impact of the relative gap among server/client CPU performance, network bandwidth, and disk I/O performance.

For both ADCC and CBL, the throughput increases and then gradually drops as the number of clients varies from 1 to 40 (Figure 10). The overhead for concurrency control gradually catches up with the gain due to client caching as the number of clients increases. Clearly, there exists an optimal number of clients to achieve the highest overall system throughput. This trade-off is more obvious in HOTCOLD than in UNIFORM because the per-client data locality in the HOTCOLD workload is reduced as the data contention increases.

P2P communication shortens the network latency when a client updates a locally cached page which is also cached on other sites. The asynchronous behavior reduces the message transmission overhead, which also helps ADCC outperform CBL. The shorter communication path for detecting data conflicts in ADCC also reduces the block rate (Figure 11). In addition, ADCC increases the number of clients needed to obtain the highest overall throughput by offloading the concurrency control functions partially from the server to the clients. Therefore, ADCC scales better than CBL.

Impact of a fast CPU: When the server CPU speed increases from 200 to 400 MIPS, both ADCC and CBL show improvements in throughput. The reduction in transaction execution time reduces the write/read conflict blocking time in both algorithms. Figure 10 shows that for 4 to 40 clients, doubling the server CPU speed increases the throughput in CBL around 20% on average, while the improvement in ADCC is only about 10%. With slow CPUs, the server CPU's utilization in CBL saturates when the number of clients exceeds 8 (Figure 12), which explains why the fast server CPU helps CBL more than ADCC.

Impact of a fast network: When the network bandwidth increases from 80 to 800 Mbps, both ADCC and CBL improve the throughput due to the reduced network latency. However, in CBL the server quickly saturates again with only 8 clients. On the other hand, the increased bandwidth improves the server CPU utilization in ADCC (Figure 12). As a result, it displays significant

improvements in both throughput and scalability by moving the optimum number of clients to 16 (Figure 10).

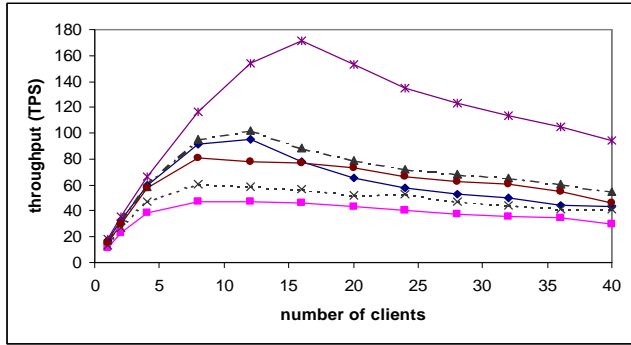


Figure 10. Throughput (HOTCOLD).

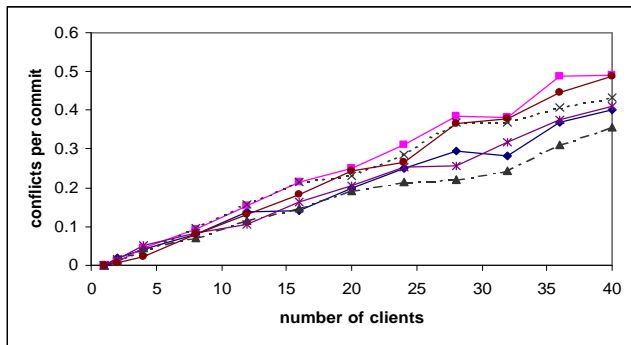


Figure 11. Conflicts per commit (HOTCOLD).

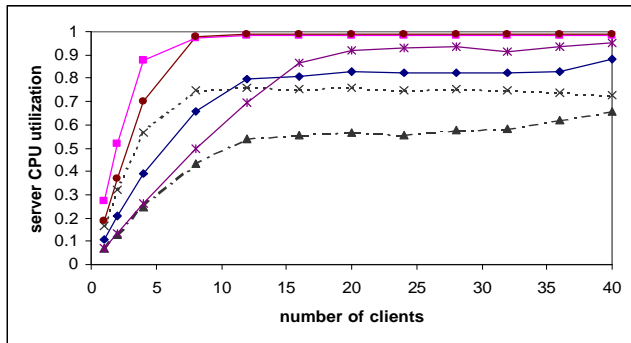
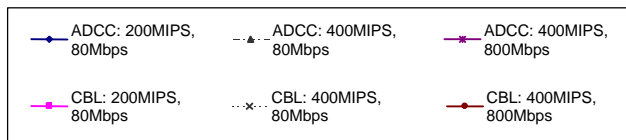


Figure 12. Server CPU utilization (HOTCOLD).



The server is often eventually the bottleneck for performance and scalability due to the excess demands for pages [8][17]. However, because the server is always on the critical path for enforcing concurrency control in CBL, this server-based communication makes the situation even worse. From the simulation results, it can be seen that, due to the server-based communication path, ADCC

will benefit more from expected technology advancements than CBL.

4.3 The PRIVATE Workload

The PRIVATE workload has the highest per-client locality among the four workloads. With this workload, the clients perform writes only on their private hot regions and there is no write/read or write/write data sharing. The lack of data contention leads to no transaction aborts.

Figure 13 shows that ADCC outperforms the pessimistic, synchronous CBL for this workload. For the fast server CPU and the high-speed network, the network and processing latencies are small. Therefore, the two algorithms perform very similarly when the client population is small.

As no write/read or write/write data sharing exists, P2P communication does not produce a direct benefit for ADCC in this situation. However, the non-blocking behavior and lower message transmission overhead allow ADCC to slightly outperform CBL-A. ADCC allows a client to continue its update on its cached pages without delay, even if this page was brought into the cache due to a read operation in a previous transaction. The client does not need to contact the server if it is the exclusive owner. On the other hand, CBL-A requires a client to send a message to the server on every initial page update and blocks until the server responds, if the client does not have a cached copy or only has a read lock for the cached copy. Compared with CBL-A, as there is no data contention in the PRIVATE workload, ADCC has slightly lower overhead for updates on locally cached data.

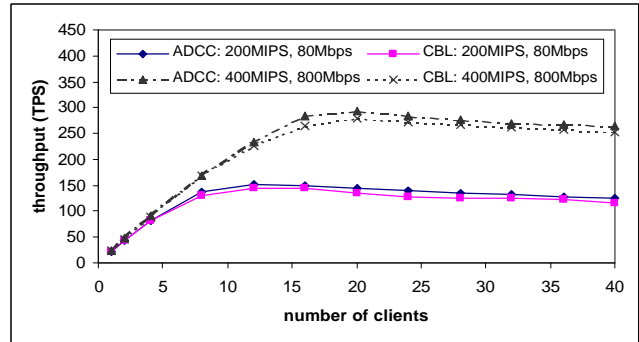


Figure 13. Throughput (PRIVATE).

4.4 The HICON Workload

The HICON workload displays a skewed data access pattern which is not usually present in data-shipping applications [9]. We include this workload primarily to examine the robustness of ADCC under extreme data contention situations.

Figures 14 and 15 show that in the range of 1 to 2 clients, ADCC and CBL behave similarly. At 4 clients and beyond, the effects of increased data contention becomes apparent and ADCC outperforms CBL significantly since CBL suffers more from blocking. The shorter communication path in ADCC also reduces the blocking time. Both ADCC and CBL suffer from increased conflict rates as clients are added.

An increase in data contention leads to high block and abort rates. Consequently, both schemes exhibit thrashing behavior when the number of clients increases beyond 2 for CBL and 4 for ADCC. For example, at 40 clients, CBL produces about 0.8 aborts per commit. The blocking overhead due to write/read conflicts dominates the other overhead. ADCC has a slightly lower abort rate than CBL. This is due to the efficient P2P communication that reduces the latency for detecting data conflicts by about 50%. Earlier discovery of data conflicts can lower the abort rate and improve the performance [8][9].

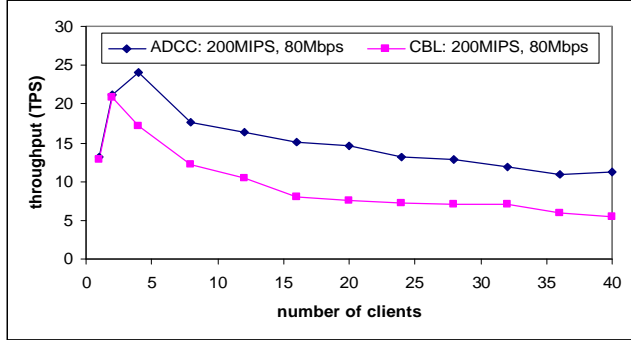


Figure 14. Throughput (HICON).

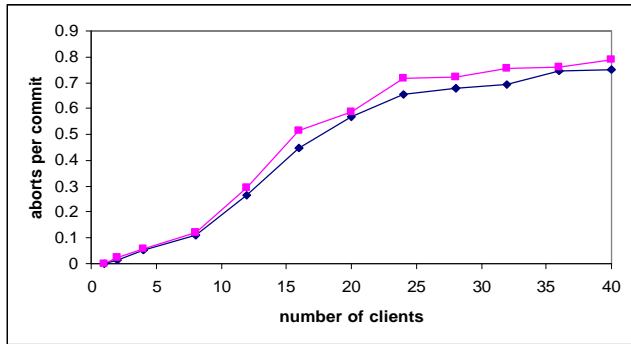


Figure 15. Abort per commit (HICON).

4.5 Discussion

A key strength of ADCC is the use of P2P communication for detecting data conflicts. P2P communication reduces the communication path under read/write and write/write sharing workloads. Consequently, it reduces the potential blocking window due to write/read and write/write conflict. It is also important for scalability as an increasing client population leads to higher overheads with client caching. Similar to CBL, ADCC encounters deadlock related aborts. *The shorter communication path leads to fewer deadlock related aborts and higher throughput in ADCC in a high contention environment.*

The second strength of ADCC is that the functionality of concurrency control is partially offloaded from the server to the clients. As the power of client workstations is increasing rapidly, ADCC can better exploit client resources. In CBL, the server is the only source for enforcing cache consistency. *ADCC removes the*

server partially from the critical path, which makes it scale much better than CBL.

Another feature of ADCC is that, similar to the detection-based No-Wait Locking protocol [18], it also has asynchronous behavior. In CBL, clients that are conducting update operations must remain blocked until the lock escalation message and the necessary callback messages have been processed at both the server and the clients. This message blocking delay increases in a heavily utilized server and network. The asynchronous behavior of ADCC, however, allows it to outperform CBL under the low contention environments.

Nevertheless, the advantages of ADCC are not free. ADCC tags some directory information into the data and speculation request messages. However, the total number of affected messages is limited. The size of related information depends on how many clients have cached the data. To ensure the performance gain due to client caching, the number of sharing clients is usually limited. Compared to the typical message size of a data page (≥ 4 Kbyte), the memory and bandwidth overhead for tagging the related directory information is low. For extreme situations which require a large number of clients to cache data, ADCC can reduce the overhead by using a coarse directory representation, i.e. using a flag to represent a group of clients [11].

Additional control messages have to be generated to maintain the directory consistency. These messages are typically small (256 byte). An optimization, *Lazy Update* of the client directory, has been designed in ADCC in order to remove this overhead. Messages for directory updates are piggybacked with other messages to reduce communication costs.

Finally, ADCC may generate more messages with the write/write conflicts than CBL. Consider a generic example of Scenario 4 (Figure 4) with n clients. Assume that A wants to update a cached page m at time t_i . As page m is cached by all n clients, A sends an invalidation request to each of the other $(n-1)$ clients. It takes time DT for the invalidation requests to reach the destinations, where DT equals to T and $2T$ for ADCC and CBL, respectively. If i clients among the other $(n-1)$ clients want to update page m , but the other $(n-1-i)$ do not use the same page before A's invalidation request arrives, this scenario leads to a total of $[(i+1)+(i+1)(n-1)+i+(n-1)+1]$ and $[(i+1)+2(n-1)+1]$ control messages for ADCC and CBL, respectively. Assume the probability of updating the same page during $[t_i, t_i+DT]$ by another client is uniform with probability p . Then the expected number of messages is:

$$\sum_{i=1}^{n-1} (2n+i+i \cdot n) P_i \text{ for ADCC} \dots\dots (1)$$

$$\sum_{i=1}^{n-1} (2n+i) P_i \text{ for CBL} \dots\dots (2)$$

where P_i is the probability that this scenario occurs. Then

$$P_i = p^i (1-p)^{n-1-i} \frac{(n-1)!}{i!(n-1-i)!} \dots\dots (3)$$

For the UNIFORM workload with the write probability (coldWrProb), p can be estimated as follows:

$$(1/\text{databaseSize})(\text{coldWrProb})DT(\text{throughput})(\text{transactionSize})$$

Recent measurements have shown that the one-way network latency T is typically less than 28 ms for communication within North America [12]. Assuming the same configuration and throughput for

ADCC and CBL, Figure 16 shows the expected numbers of messages sent in ADCC and CBL, which are computed using equations (1), (2) and (3), and following system parameters. Counter to intuition, statistically ADCC has the lower message cost because the communication latency for detecting write/write conflicts in ADCC is much smaller than that in CBL.

databaseSize=2000pages, $T=28\text{ms}$, throughput=60TPS,
transactionSize=20pages, coldWrProb=0.4.

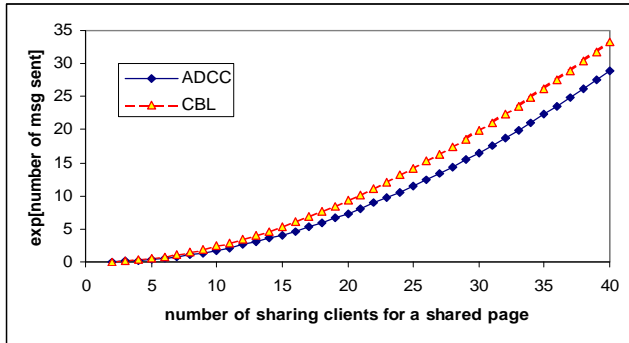


Figure 16. Msg cost comparison for ADCC and CBL.

5. RELATED WORK

The idea of ADCC originated with the memory coherency protocol used in SGI Origin multiprocessor systems [11]. ADCC is distinguished from Origin memory coherency protocol in several important aspects. ADCC is software-based and employs a two-tier directory, while the protocol in Origin systems is hardware-based and uses a single-tier directory only. Another important difference is that the coherency protocol in multiprocessor systems maintains a consistent view of memory for every processor on each memory operation. ADCC has a coarser granularity of atomicity, which requires a sequence of operations to be executed as a whole. Therefore, ADCC must handle deadlocks and aborts at the transaction (a sequence of operations) level.

Table 4 compares our algorithm, ADCC, with the algorithms proposed for data-sharing DBMS during the last decade. The algorithms can be classified into two categories according to their policy for invalid access prevention: avoidance-based and detection based [9]. All of the algorithms, except ADCC, completely rely on a centralized server for concurrency control. CBL is widely accepted as the leading algorithm due to its good performance and low abort rate [8]. In general, it has better performance than Caching Two-Phase Locking (C2PL) [18], No-Wait Locking (NWL) [18], Cache Locks [19] and Notify Locks [19]. Optimistic Two-Phase Locking (O2PL) [3] and Adaptive Optimistic Concurrency Control (AOCC) [1] have similar or higher throughput. However, the major drawback of these two optimistic approaches is the deferred consistency check, which leads to high abort rates. The abort rate is a critical issue for users in the highly interactive environments that are common for page servers. Asynchronous Avoidance-based Cache Consistency (AACC) [14] can lower the abort rate while maintaining high throughput. However, both AOCC and AACC were proposed for adaptive locking, which switches locking between the page and the object level.

It has been shown [8] that non-adaptive callback schemes are the best overall choices and so are the current system trend. Carey *et al.* [5] showed that adaptive schemes have better performance for workloads that exhibit fine-grained read-write sharing. That study [5] also showed that, for workloads with high page locality, the non-adaptive callback schemes perform as well as the adaptive schemes, or even better than the adaptive schemes, when the probability of a write is high. Whether a non-adaptive scheme or an adaptive scheme is better depends on the page sharing granularity in the workload.

Franklin *et al.* [7] extended CBL with a global memory management concept that allows clients to obtain pages from other clients. Dahlin *et al.* [6] examined a similar concept, cooperative caching, in the context of distributed file system reads. ADCC differs from global memory management and cooperative caching in several important aspects. The most significant difference is that, in the global memory management or cooperative caching approaches, the server is still the only site with knowledge of where page copies are cached in the system and the only source for enforcing cache consistency. Therefore, clients are not able to detect data conflicts via the direct client-client (P2P) communication. In ADCC, in contrast, not only the server but also the clients each maintain a directory to track the global state of locally cached data. The direct P2P communication in ADCC decreases the communication latency for detecting data conflict. As a result, clients in ADCC are more actively involved in maintaining cache consistency and the server is partially removed from the critical path. By contrast, the primary goal of global memory management or cooperative caching is to reduce the server disk accesses by utilizing the remote client memory, while ADCC focuses on maintaining cache consistency with parallel communication (client-server and client-client). To integrate ADCC with the concept of global memory management or cooperative caching is an important direction for future work.

TABLE 4. Comparison of Cache Consistency Algorithms.

Protocols	Invalid access prevention	Validity check initialization	Serialization mechanism	Consistency granularity	Comm. path
CBL	Avoidance-based	Synchronous	One-tier dir, Lock-based	Page	Srv-based
AACC	Avoidance-based	Asynchronous	One-tier dir, Lock-based	Page + object	Srv-based
ADCC	Avoidance-based	Asynchronous	Two-tier dir, State-based	Page	P2P + Srv-based
Notify locks	Avoidance-based	Deferred	One-tier dir, Lock-based	Page	Srv-based
O2PL	Avoidance-based	Deferred	One-tier dir, Lock-based	Page	Srv-based
C2PL	Detection-based	Synchronous	One-tier dir, Lock-based	Page	Srv-based
NWL	Detection-based	Asynchronous	One-tier dir, Lock-based	Page	Srv-based
AOCC	Detection-based	Deferred	One-tier dir, Time-stamp	Page + object	Srv-based
Cache locks	Detection-based	Deferred	One-tier dir, Lock-based	Page	Srv-based

Another recent paper [16] related to cooperative consistency studied the coherency of time-varying data items in a set of repositories. This study is based on an architecture that consists of one or more sources, multiple repositories and several clients. Their approach describes how to replicate data across multiple repositories so that clients can access the repository that is best positioned to meet their data coherency goal. In addition, they used data fidelity as the

evaluation metric. Our study, in contrast, focuses on transaction level concurrency control for a data-shipping DBMS with the goal of improving performance and scalability.

6. CONCLUSIONS

In order to reduce the latency for detecting data conflicts and to relieve the server bottleneck, an efficient cache consistency protocol, Active Data-aware Cache Consistency, has been proposed for highly-scalable data-shipping DBMS architectures. By allowing clients to be aware of the global state of their cached data, the clients are actively involved in maintaining the cache consistency. An optimization for the client directory consistency, Lazy Update, has been designed to reduce the message overhead for maintaining client directory consistency. Using P2P communication, ADCC reduces network latency for invalidation messages for write/read and write/write sharing by 50% compared to the server-based communication scheme, while increasing message overhead by only around 8%. Shortening the communication path not only improves throughput but also reduces the abort rate. By partially removing the server from the critical path for cache consistency, ADCC scales better than CBL. Both the simulation results and the analysis indicate that the overhead of ADCC is low. The experimental study shows that ADCC outperforms CBL under the four workloads tested. In particular, without per-client data locality, the increasing level of data contention leads to a higher performance gap between ADCC and CBL.

7. ACKNOWLEDGMENT

The authors thank Chris Hescott, Joshua J Yi, Baris M Kazar, Ying Chen, Sreekumar Kodakara and Jon Weissman for their helpful comments on a previous draft of this paper. The authors also thank Kaladhar Voruganti for some helpful discussions. This project was supported by the University of Minnesota Digital Technology Center (DTC) Intelligent Storage Consortium (DISC), and the Minnesota Supercomputing Institute.

8. REFERENCES

- [1] Adya, A., Gruber, R., Liskov, B. and Maheshwari, U. "Efficient optimistic concurrency control using loosely synchronized clocks," in Proceedings of the ACM SIGMOD Conference on Management of Data. San Jose, CA, pp. 23–34. May 1995.
- [2] Amsaleg, L., Franklin, M. and Tomasic, A. "Dynamic query operator scheduling for wide-area remote access," Distributed and Parallel Databases, vol. 6(3): pp. 217-246, 1998.
- [3] Carey, M.J., Franklin, M.J., Livny M. and Shekita, E. J. "Data Caching Tradeoffs in Client-Server DBMS Architectures," in Proceedings of the ACM SIGMOD, pp. 357-366, May 1991.
- [4] Carey, M.J., DeWitt, D. and Naughton, J. "The OO7 benchmark," in Proceedings of the ACM SIGMOD Conference on Management of Data. Washington, DC, pp. 12–21. May 1993.
- [5] Carey, M.J., Franklin, M.J., and Zaharioudakis, M. "Fine-grained sharing in a page server OODBMS," in Proceedings of the ACM SIGMOD Conference on Management of Data. Minneapolis, MN, pp. 359–370, May 1994.
- [6] Dahlin, M.D., Wang, R.Y., Anderson, T.E. and Patterson, D.A. "Cooperative Caching: Using Remote Client Memory to Improve File System Performance," In Proc. of the First Symposium on Operating Systems Design and Implementation, pp. 267-280, November 1994.
- [7] Franklin, M.J., Carey, M.J. and Livny, M. "Global Memory Management in Client-Server DBMS Architectures," In Proc. of the International Conference on Very Large Data Bases, pp. 596–609, August 1992.
- [8] Franklin, M.J. "Client Data Caching: A Foundation for High Performance Object Database Systems," Kluwer Academic Publishers, Boston, MA, 1996.
- [9] Franklin, M.J., Carey, M.J. and Livny, M. "Transactional client-server cache consistency: alternatives and performance," ACM Transactions on Database Systems, vol. 22(3), pp. 315-363, September 1997.
- [10] Fritz, J. "Gigabit Ethernet hits second gear," <http://www.nwfusion.com/research/2000/0320revgig.html>
- [11] Laudon, J. and Lenoski, D. "The SGI Origin: A ccNUMA highly scalable server," in Proceedings of the 24th Annual International Symposium on Computer Architecture, vol. 25(2), pp. 241-251, 1997.
- [12] MCI Corp., "Network Latency Statistics," 2003. <http://global.mci.com/about/network/latency>
- [13] Objectivity Database Systems Inc. "Objectivity/DB Technical Overview," <http://www.objectivity.com/DevCentral/Products/TechDocs/pdfs/techOverview6.pdf>
- [14] Ozsu, M.T., Voruganti, K. and Unrau, R. "An asynchronous avoidance-based Cache Consistency Algorithm for Client Caching DBMSs," in Proceedings of the Conference on Very Large Data Bases (VLDB). New York, NY, pp. 440-451, 1998.
- [15] Pargal, S. "Future Technologies for Storage Networks." Compellent Technologies Inc. April 2003. <http://www.dtc.umn.edu/diskcon/>
- [16] Shah S., Ramamritham K. and Shenoy P.J. "Maintaining Coherency of Dynamic Data in Cooperating Repositories," in Proceedings of the Conference on Very Large Data Bases (VLDB). Hong Kong, pp. 526-537, 2002
- [17] Silberschatz, A., Korth, H., and Sudarshan, S. *Database System Concepts, 4th ed.*, McGraw Hill, 2001.
- [18] Wang, Y. and Rowe, L.A. "Cache consistency and concurrency control in a client/server DBMS architecture," in Proceedings of the ACM SIGMOD Conference on Management of Data. Denver, CO, pp. 367–377, May 1991.
- [19] Wilkinson, K. and Neiman, M.-A. "Maintaining consistency of client-cached data," in Proceedings of the Conference on Very Large Data Bases (VLDB). Brisbane, Australia, pp. 122-133, August 1990.