

# Self-tuning Speculation for Maintaining the Consistency of Client-Cached Data

Keqiang Wu, David J. Lilja  
Department of Electrical and Computer Engineering  
Minnesota Supercomputing Institute  
University of Minnesota  
Minneapolis, MN, USA

**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 rely completely on a centralized server or servers to provide concurrency control, which imposes a limitation on the scalability and performance of these systems. In addition, traditional asynchronous and deferred protocols are “blindly” optimistic on the cached data and do not exploit data sharing information. This paper presents a new protocol, Self-tuning Active Data-aware Cache Consistency (SADCC), that employs parallel communication and self-tuning speculation to improve system performance. Using parallel communication with simultaneous client-server and client-client communication in high contention environments, SADCC reduces the network latency for detecting data conflicts by 50%, while increasing message volume overhead by only about 4.8%. By being aware of the global states of cached data, clients can self-tune between optimistic and pessimistic consistency control to reduce the abort rate by statistically quantifying the speculation cost. We compare SADCC with the leading cache consistency algorithms, Active Data-aware Cache Consistency (ADCC) and Asynchronous Avoidance-based Cache Consistency (AACC), in a page server DBMS architecture with page-level consistency. The simulation results show that, in a non-contention environment, both SADCC and ADCC display a slight reduction (an average of 2.3%) in performance compared to AACC with a high-speed network environment. With high contention, however, SADCC has an average of 14% higher throughput than AACC and 6% higher throughput than ADCC.

**Keywords**—*self-tuning speculation; parallel communication; concurrency control; data-shipping DBMS*

## I. INTRODUCTION

Client/server DBMS architectures fall into two main categories, *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. 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.

The data-shipping architecture has been inspired by the dramatic improvements in computer price-performance and in the performance and availability of network communication. The technology advance has made it desirable and practical to offload more functionality from the server to the client workstations [12]. 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 item to reside in multiple client caches. Moreover, when inter-transaction caching is used, an item may

remain cached locally when the transaction that initially accessed it has completed. Concurrency control for cached data must be enforced to ensure that all clients have consistent data copies in their caches and that they see a serializable view of the database.

Research on concurrency control for client-server database systems has been motivated by the inherently distributed nature of the advanced applications that DBMSs support, e.g. electronic commerce and distributed multimedia applications. The primary disadvantage of traditional protocols [1][3][8][13][16][17] is the server-based communication path, i.e. whenever a client needs data or permission to perform an operation on cached data, it must always send a request to the server sometime prior to transaction commit. 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. Putting the server on the critical path unavoidably increases the communication latency. A protocol [18] that employs both client-client and server-based communication was recently proposed to address this problem. However, in current optimistic schemes (asynchronous [13][18] or deferred [1]), clients always proceed to write to locally-cached copies under the assumption that no other clients are currently using the data. If this optimism turns out to be wrong, then the transaction has to abort. This simple assumption makes current asynchronous and deferred schemes *blindly* optimistic in a high contention environment, which can result in significant lost work for users in the highly interactive OODBMS environments for which page servers are often intended. For example, the high abort rate in high contention workloads make optimistic approaches, such as Adaptive Optimistic Concurrency Control (AOCC) [1], unsuitable for interactive application domains [8][13].

In this paper, we propose an efficient client cache consistency algorithm which employs both parallel communication (client-client and server-based) and self-tuning speculation. The primary contributions of this research are:

- 1. A new algorithm, Self-tuning Active Data-aware Cache Consistency (SADCC), that self-tunes between optimistic and pessimistic consistency control to improve performance.**
- 2. A demonstration that the integration of self-tuning speculation and parallel communication in SADCC improves throughput by 6% compared to ADCC [18] and by 14% compared to AACC [13] in a high contention environment.**
- 3. A demonstration that, with high contention and transaction-level temporal locality, write/write conflicts can dominate other overheads when the transaction granularity (size) is reduced.**

Since non-adaptive callback schemes have been shown in the past to be the best overall choice for DBMS systems [8], we compare SADCC with the two leading cache consistency algorithms, Active Data-aware Cache Consistency (ADCC) [18] and Asynchronous Avoidance-based Cache Consistency (AACC) [13]. Similar to previous work [7][8][18], 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 [12].

The remainder of the paper is organized as follows. Section II briefly reviews the existing protocols. Section III describes the proposed SADCC algorithm in detail. Section IV describes the experimental setup; while Section V the simulation results. Section VI briefly reviews related work. Finally, Section VII summarizes our conclusions.

## II. EXISTING PROTOCOLS

In AACC [13], clients totally rely on the server to enforce cache consistency. ADCC [18] employs both server-based and client-client communications. As a result, the server in ADCC is partially offloaded from the critical path for concurrency control. However, both have similar asynchronous write behavior, i.e., clients always proceed to write locally cached copies under the assumption that no other clients are currently using the data. In addition to the deadlock abort, both AACC and ADCC encounter the mis-speculation abort due to false speculations. The performance of ADCC with respect to AACC has not been studied before. AACC was originally proposed for adaptive locking, which switches locking between the page and object levels. Since this paper focuses on page level consistency, we study AACC with page level locking only. In this section, we briefly review these two protocols.

### A. The AACC Protocol

Similar to traditional protocols [1][3][8][16][17], AACC [13] allows only the server to keep track of data cached by clients. There are three modes for cached pages, *private-read*, *shared-read*, and *write*. Locally cached page copies are always guaranteed to be valid, so clients can read them without contacting the server.

When a client (e.g. client A) wants to read a page which is not in its cache, it sends a read request to the server. (1) If the page is not cached anywhere else, the server returns the page to client A in *private-read* mode. (2) If the page is exclusively cached by client B in *private-read* mode, the server returns the page to client A in *shared-read* mode. The server also informs client B to change the page lock to *shared-read* mode via a *piggyback message*. The inherent message delay may cause situations where B has the page in *private-read* mode but A has the same page in *shared-read* mode. The potential problems are resolved by commit-time validation. (3) If the page is cached by clients B and C in *shared-read* mode, the server returns the page to A in *shared-read* mode. (4) If the page is cached at client B in *write* mode, the server blocks the requesting client, A, until client B has relinquished the *write* lock.

When A wants to write a cached page, it proceeds to update speculatively. At the same time, A takes some additional actions depending on the page state. (1) If the cached page is in *private-read* mode, A informs the server about this update by *piggybacking* the information on a subsequent message. Upon receiving this piggyback message, the server sends callback requests to related clients if this page is residing at other clients in *shared-read* mode. If no other clients cache the page, the server updates its lock table to indicate that A has a *write* lock for the page. (2) If the cached page is in *share-read* mode, A informs the server about this update via an explicit lock escalation message. Upon receiving this message, the server sends callback requests to related clients. The server cannot grant a *write*

lock to A before receiving positive response from all related clients indicating that they have invalidated the page. If the server receives a negative callback response indicating that there is a conflict, it performs deadlock processing. If there are no deadlocks, the client that has performed the initial update cannot commit before the client that is reading the data.

It is noted that, due to the potential inconsistent states held by different clients (e.g. A and B) on the same page, A might update a page that B has read or written. Consequently, when a client finishes the execution phase, it must conduct commit-time validation. Before the server responds, the client has to block. If the client receives a negative response from the server, it must abort the transaction.

### B. *The ADCC Protocol*

ADCC [18] allows not only the server but also the clients to maintain a directory for each cached page. The related directory information is tagged with the data page and sent to the requester. The server uses *Lazy Updates* to reduce the overhead for maintaining the consistency among the server and client directories.

Similar to AACC, clients can cache data across transaction boundaries and can read local data without contacting the server. A page in the client buffer may be in one of five states. The three stable states are: (1) ***exclusive-clean*** - only one unmodified (read-only) cached copy exists in the clients; (2) ***shared*** - more than one read-only cached copy whose whereabouts are indicated by the directory; (3) ***modified*** - only one modified page is cached. Two transition states are: (4) ***busy*** - the transaction is in the process of updating the data; (5) ***speculative-busy*** - which is similar to the *busy* state except that it implies that the update is speculative.

When a client updates a cached page, it changes the page state to one of the transition states. It will not change the page state to a stable state until the transaction has committed or aborted. At the same time, it informs the server about the speculation and sends an invalidation request to the other sharing clients. When a page is in one of the transition states, the client ignores any invalidation requests for this page from peers. The *speculation-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.

A page in the server may be in one of five states. The three stable states are: (1) ***unowned*** - no cached copies in clients; (2) ***shared*** - two or more read-only cached copies whose whereabouts are indicated by the directory; (3) ***exclusive*** - one read or write cached copy is in a client, indicated by the directory. The two transition states are: (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 yet committed. 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.

If the server receives a request from A for a page which is exclusively cached by another client, i.e. client B, the server forwards this request to B which then provides the data to A. Because of this design, a client can update an exclusively cached page without contacting the server, which is different from AACC.

### III. A SELF-TUNING ACTIVE DATA-AWARE CACHE CONSISTENCY (SADCC) ALGORITHM

SADCC is an extension of ADCC, which also employs both direct client-client (P2P) and server-based communication. Consequently, SADCC shares two important advantages with ADCC: (1) a shorter path for detecting read/write and write/write conflict; and (2) better scalability via partially offloading the server from the critical path. In addition, compared to ADCC, SADCC reduces the message overhead in several scenarios by using piggyback messages. Most importantly, SADCC employs a self-tuning speculation technique to improve performance and reduce the abort rate in a high contention environment.

#### A. Self-tuning speculation

Current asynchronous schemes, including ADCC and AACC, proceed to write a locally-cached copy under the assumption that the write intention declaration will succeed. If this assumption turns out to be incorrect, then the transaction must abort. In contrast to these previous protocols, SADCC statistically quantifies the risk of this assumption to self-tune between optimistic and pessimistic consistency control.

Figure 1 shows a generic example of a potential write/write conflict with  $n$  clients who have cached the same page  $m$ . Only client  $A_1$  wants to write the cached page  $m$  at time  $t_1$ . To simplify the explanation, we only show the actions of client  $A_1$ . In SADCC,  $A_1$  can be in either *optimistic* or *pessimistic* mode, depending on the total number of clients sharing a page. If  $n$  exceeds a certain threshold,  $n_r$ ,  $A_1$  switches to the pessimistic mode. While in the pessimistic mode (Figure 1-a),  $A_1$  informs the server (`spec`) of its write request and the related mode. At the same time,  $A_1$  sends an invalidation request (`inv`) to each of the other  $(n-1)$  clients. Before the server responds,  $A_1$  has to block (i.e. stops). If it is in the optimistic mode,  $A_1$  behaves similarly to the pessimistic mode, except that  $A_1$  proceeds with the update before the server responds.

Client  $A_1$  in ADCC (Figure 1-b) takes similar actions as SADCC in the optimistic mode. However, the server in ADCC may take different actions from the server in SADCC. In ADCC, when the server receives a speculation write request, it always sends an *explicit* message to the requesting client. In contrast, in SADCC the server responds to the requesting client using a *piggyback* message if the speculation succeeds, or an *explicit* message if the speculation is false or the requesting client is in pessimistic mode. Compared to SADCC, ADCC has higher message overhead.

In AACC (Figure 1-c),  $A_1$  always proceeds to update ( $Spec\ wr$ ) and informs the server of the speculative update ( $spec$ ), and it is the server that callbacks all copies cached by the other clients. The server cannot grant a *write* lock to  $A_1$  before receiving positive response from  $A_2, \dots, A_n$ .

If  $T$  is the average one-way trip delay for a message between the client and the server or two clients, it takes an average of time  $T$  in SADCC and ADCC, but  $2T$  in AACC, for the destinations to be aware of  $A_1$ 's write intention. If there are  $i$  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 being aware of  $A_1$ 's write intention, this leads to  $i$  conflicts in all three schemes. The unsuccessful speculative clients in ADCC and AACC have to abort. However, there is no speculation-related abort in SADCC when in the pessimistic mode. The overheads of messages, aborts due to mis-speculation, and conflicts are shown on TABLE I.

TABLE I. COST FOR SADCC, ADCC, AND AACC FOR  $(i+1)$  CONCURRENT UPDATES ON A PAGE WHICH IS CACHED BY  $N$  CLIENTS. THE LETTERS IN PARANTHESES, O AND P, DENOTE THE OPTIMISTIC AND PESSIMISTIC MODES, RESPECTIVELY.

Cost	SADCC (o)	SADCC (p)	ADCC	AACC
Msg #	$i(n+1)+2n-1$	$i(n+1)+2n$	$i(n+1)+2n$	$3i+n+2$
Speculation-Abort #	$i$	0	$i$	$i$
Conflict #	$i$	$i$	$i$	$i$

We use the commonly-used HOTCOLD workload [7][8] as a concrete example to illustrate the self-tuning speculation technique. Similar analysis can be applied on UNIFORM and HICON workloads. Consider the HOTCOLD workload in TABLE III, each client has a *hotBound* (e.g. 50) page range from which *hotAccProb* (e.g. 80%) of its accesses are drawn. The remaining *coldAccProb* (i.e.,  $1 - hotAccProb$ ) of the accesses are distributed uniformly among the remaining pages in the database. All pages are accessed with a certain write probability, *hotWrProb* (e.g. 20%) and *coldWrProb* (e.g. 20%). The hot ranges of the clients do not overlap. However, it is noted that the hot range pages of each client are also in the cold ranges of all of the other clients.

Assume page  $m$  shown in Figure 1 is in the hot range of  $A_n$ . Then the scenario for  $i$  concurrent updates consists of the following two possible cases. The probability for this scenario to occur,  $P_i$ , is estimated as follows, where  $p$  is the probability for  $A_2, \dots, A_{n-1}$  to write the locally cached page  $m$  before being aware of  $A_1$ 's write intention, and  $DT$  is equal to  $T$  for SADCC and ADCC, and  $2T$  for AACC.

$$p = (coldAccProb) \frac{(coldWrProb)}{(databaseSize - hotRange)} (throughput)(transSize) (DT)$$

Case 1.  $i$  clients among  $A_2, \dots, A_{n-1}$  want to update page  $m$ , but  $A_n$  does not.

$$P_i(1) = [1 - (hotAccProb) \frac{(hotWrProb)}{(hotRange)} (throughput)(transSize)] p^i (1-p)^{n-2-i} \frac{(n-2)!}{i!(n-2-i)!} (DT)$$

Case 2.  $A_n$  and the other  $(i-1)$  clients among  $A_2, \dots, A_{n-1}$  want to update page  $m$ .

$$P_i(2) = [(hotAccProb) \frac{(hotWrProb)}{(hotRange)} (throughput)(transSize)] p^{(i-1)} (1-p)^{n-2-(i-1)} \frac{(n-2)!}{(i-1)![n-2-(i-1)]!} (DT)$$

$$P_i = P_i(1) + P_i(2)$$

Therefore, the expected cost is:

$$\exp[overhead] = \sum_{i=1}^{n-1} (overhead \times P_i) \dots \dots (1)$$

Recent measurements have shown that the one-way network latency  $T$  is typically less than 28ms for communication within North America [11]. Assuming the same configuration and throughput for SADCC, ADCC, and AACC, and the threshold for SADCC is  $n_i=6$ , Figures 2 and 3 show the expected overheads, which are computed using equation (1) and the following system parameters.

$databaseSize=2000pages, T=28ms, throughput=25TPS, transSize=20pages,$

$hotRange=50pages, hotAccProb=0.8, hotWrProb=0.2, coldAccProb=0.2, coldWrProb=0.2.$

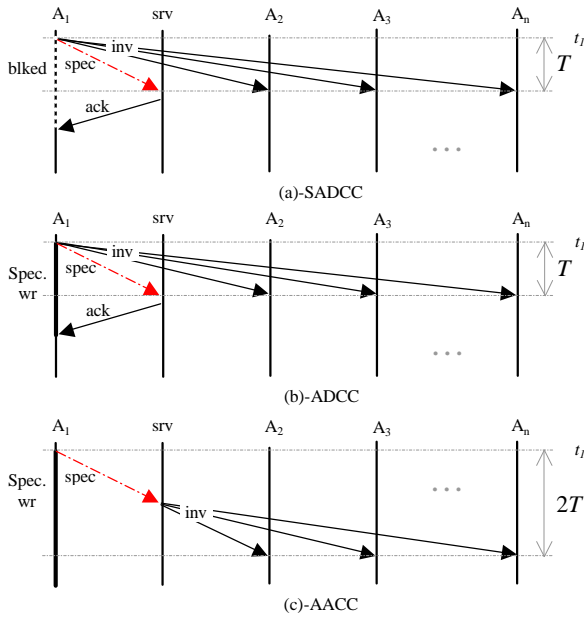


Figure 1. A generic example of potential write/write conflict for a page cached by  $n$  clients, where (a) represents SADCC in its pessimistic mode, and  $T$  is the average one-way trip delay for a message between the client and the server or two clients.

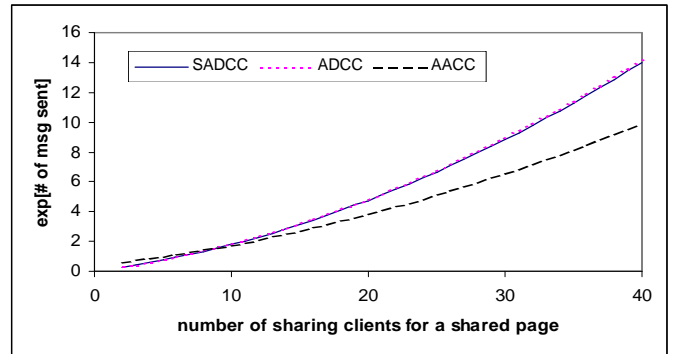


Figure 2. Message overhead due to false speculation.

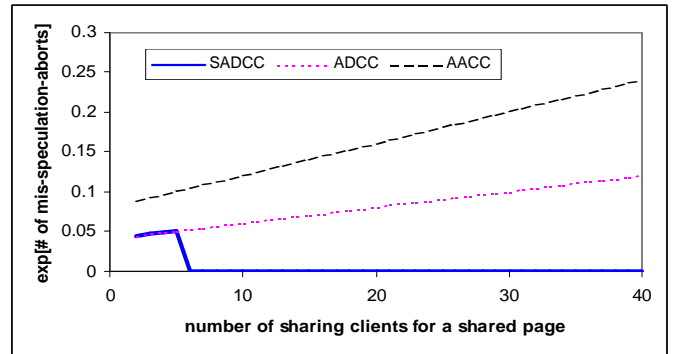


Figure 3. Abort overhead due to false speculation.

For a small number of sharing clients, SADCC has lowest message overhead, although the message overhead in ADCC is close to SADCC. When the number of sharing clients is larger than 9, the message overhead for SADCC and ADCC gradually exceeds AACC.

However, ADCC consistently displays a lower abort overhead than AACC. SADCC has a similar abort overhead as ADCC under the optimistic mode, i.e. the number of sharing clients is less than 6, but it has no aborts due to mis-speculation under the pessimistic mode.

The primary goal of the asynchronous protocols, such as ADCC and AACC, is to achieve a compromise between the synchronous and deferred techniques. They aim to mitigate the cost of interaction with the server while at the same time lowering the abort rate. However, if the asynchronous update leads to a wrong speculation, to block a write operation before receiving permission is a better choice than non-blocking and then aborting the whole transaction. Considering that an unsuccessful non-blocking update operation results in an abort of the whole transaction, SADCC uses a heuristic criterion to adaptively switch between optimistic (non-blocking) and pessimistic (blocking) for update on a shared cached page based on the following restriction, equation (2). This criterion determines the threshold of number of sharing clients,  $n_s$ , at which SADCC switches between optimistic and pessimistic modes.

$$\text{Expected number of aborts due to false speculation} < 1/\text{TransactionSize} \dots\dots (2)$$

### B. Scenario Descriptions

To assist in understanding the algorithm, we present four scenarios in Figure 4 to illustrate the speculation and blocking behaviors in the three schemes. These scenarios are similar to those used in previous studies [13] in client-server DBMS cache consistency algorithms. While we focus on the interaction among the server and two or three clients, the discussion is valid for any number of clients. To simplify the illustration, we only consider SADCC with optimistic mode in scenario 1.

**Scenario 1:** Client A wants to update page  $m$  which is cached by both A and B (the state is shared for SADCC and ADCC and shared-read for AACC). B does not use that page. In SADCC, A sends speculation (`spec`) and invalidation requests (`inv`) to the server and B, respectively. At the same time, assuming the optimistic mode, A speculatively conducts the update. (If the mode is pessimistic, A blocks before the server responds.) The server grants the speculation (`pgybk ack`) via a piggyback message, and B invalidates its cached copy and informs A (`ack`) directly.

ADCC behave similarly except that the server grants the speculation (`ack`) via an explicit message.

In AACC, when the server receives the lock-escalation request (`LE`) from A, the server sends a callback message (`CB`) to B. Since B is not using page  $m$ , B invalidates its copy and informs the server (`pgybk ack`) via a piggyback message. It is noted that, when all operations in A's transaction have been finished, AACC requires commit-time validation. A has to send a commit-validation request (`CV`) to the server. Before the server response, A has to block (i.e. stop). The server does not allow A to commit until the server has received all positive responses (`ack`) to its callback requests.

Before being aware of A's write intention, the other two scenarios are possible. (1) If B has read the data, then A is not allowed to commit before B. (2) If B has proceeded to update the data, then either A or B has to abort. The race for write/write conflict is resolved

at the server. The client whose speculation request arrives at the server earlier is the winner. The server in ADCC always responds to A using an explicit message; while in SADCC, the server responds to A using a piggyback message if A's speculation succeeds, or an explicit message if A's speculation is false or A is in pessimistic mode. As piggybacking does not generate additional messages, compared to ADCC, SADCC typically saves one message. In average, it takes time  $T$  in SADCC and ADCC, but  $2T$  in AACC, for B to be aware of A's write intention. As a result, compared to AACC, SADCC and ADCC reduce the potential conflict interval by approximately 50% and partially remove the server from issuing callback message (CB) requests to B.

**Scenario 2:** Page  $m$  is cached only by B (the state is exclusive for SADCC and ADCC, and private-read for AACC), but B is not using it. A wants to update it. In both SADCC and ADCC, A sends a request ( $wr$ ) to the server. The server forwards the request ( $fw d req$ ) to the exclusive owner, B, which provides the data ( $fw d data$ ) directly to A and invalidates its own copy.

In AACC, when the server receives the lock-escalation request from A, the server sends a callback message (CB) to B. Since B is not using page  $m$ , B invalidates its copy and informs the server ( $ack$ ). Upon receiving the positive response from B, the server provides the page ( $fw d data$ ) to A.

Consider the other two possible scenarios before B is aware of A's write intention. (1) If B has read the data, then the server in AACC blocks A's write request and will not provide the data to A until B has committed; in contrast, client B in SADCC and ADCC still forwards the data to A but will also inform A of the read conflict. Therefore, A is not allowed to commit before B. (2) If B has updated the data, then A will be blocked in all three schemes. However, in AACC, the server will not provide the data to A until B has committed. In contrast, in SADCC and ADCC, client B directly forwards the data to A after B commits. In general, compared with SADCC and ADCC, AACC makes A block at least  $3T$  (including  $2T$  due to B's commit-time validation) time longer. Similar to Scenario 1, SADCC and ADCC partially remove the server from the critical path.

**Scenario 3:** Page  $m$  is cached by both B and C (the state is shared for SADCC and ADCC and shared-read for AACC), but neither uses it. A wants to update it. In SADCC, when the server receives A's request ( $wr$ ), it provides the data ( $fw d data$ ) 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 both B and C invalidate their copies. After invalidating the cached copies, B and C informs the server using piggyback messages ( $pgybk ack$ ).

ADCC performs similar to SADCC, except that B and C inform A of the invalidation directly using explicit messages ( $ack$ ). Compared to SADCC, ADCC generates two more messages.

In AACC, the server does not grant a write lock to A until the invalidation acknowledgements ( $ack$ ) from both B and C arrive. A blocks until the server responds. Compared to SADCC and ADCC, AACC makes A block at least  $2T$  longer.

**Scenario 4:** Page  $m$  is cached by both  $B$  and  $C$  (the state is shared for SADCC and ADCC and shared-read for AACC), but neither is using it.  $A$  wants to read it. In SADCC, 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  $C$ .

ADCC requires the client to take similar actions as SADCC.

In AACC, the server returns the page in *shared-read* mode to  $A$  immediately.

Compared to AACC, SADCC and ADCC increase the bandwidth consumption slightly due to the two piggybacked control messages.

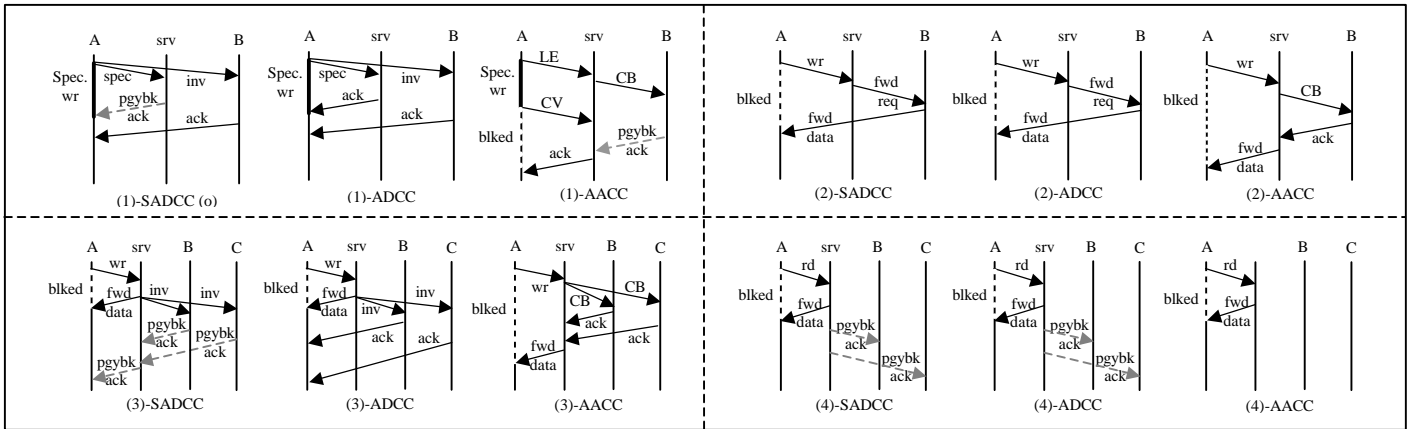


Figure 4. SADCC cache consistency scenarios that involve the server (srv) and three clients (A, B and C). Each arc denotes a message. A dashed arc denotes that the related message is piggybacked with a subsequent message. Each scenario is compared with ADCC and AACC, where CB, LE, and CV denote requests for callback, lock-escalation, and commit-time validation, respectively. More messages are handled in parallel in SADCC and ADCC than in AACC.

#### IV. A PAGE SERVER DBMS MODEL FOR PERFORMANCE EVALUATION

##### A. The System Model

We use a simulation model similar to those used in previous client cache consistency performance studies [1][3][7][8][13][16][17][18], 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.

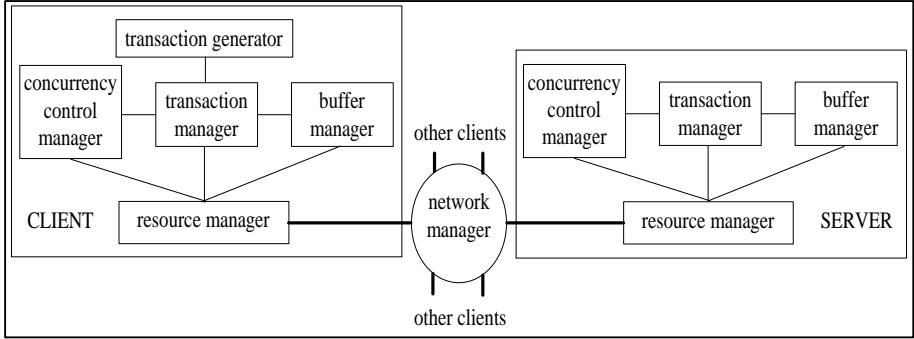


Figure 5. Model of a page-server client-server database management system.

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 [13].

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 II 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 [7][8][18].

*B. The Workload Model*

The multi-user OO7 benchmark has been developed for the object DBMS performance study [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 [3][5]. The UNIFORM, HOTCOLD, HICON and PRIVATE workloads have been

widely used for client caching consistency study [1][7]. These cover a wide spectrum of data contention levels and spatial (per-client) data locality. However, these workloads do not embed sufficient temporal locality at transaction level, i.e. a write operation is likely to update a page which has been read by the same transaction, which is common in reality [15]. For example, a transaction process for purchasing an airplane ticket typically consists of following steps: (1) A customer is queried for a desired flight time and cities. Information about the desired flight is located in database pages *A*, *B* and *C*. (2) The customer is told about the options, and selects a flight whose data, including the number of reservations for that flight is in *A*. A reservation on that flight is made for customer. (3) The customer selects a seat for the flight; seat data for the flight is in database page *D*, and so forth. This transaction can be represented as a sequence of read and write actions as follows.

RD(A); RD(B); RD(C); WR(A); RD(D); WR(D); .....

TABLE II. PARAMETERS USED IN THE SIMULATION STUDY.

System Parameter		
pageSize	Size of a page	4Kbyte
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 (AACC)	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

In this study, we extend the UNIFORM, HOTCOLD, HICON and PRIVATE data sharing patterns by embedding the transaction-level temporal locality, e.g. “**RD(A)**; RD(B); RD(C); **WR(A)**,” shown in the above example. In order to embed such temporal locality, we randomly pair *x*% of write operations with read operations within the same transaction. Each pair performs read/write operations on the same page. The level of temporal locality can be adjusted by varying the value *x*. If *x*=0, the workloads are the same as those used in previous study [7][8][18]; if *x*=100, the workloads have the highest level of read/write temporal locality among read and write operations. In this study, we set *x*=50. Adding such temporal locality might slightly change the overall distribution of spatial locality but

are equal for all protocols. TABLE III summarizes the workloads that are examined in this paper. For completeness, we give a brief review of the workloads below.

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 considered 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 spatial 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 spatial locality and a moderate amount of sharing and data contention among clients. Each client accesses the data from its hot region (80% of the time) and the cold region (20% of the time). The clients can update pages in both regions. 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. 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.

TABLE III. WORKLOAD PARAMETERS FOR CLIENT  $i$ .

Parameter	UNIFORM	HOTCOLD	HICON	PRIVATE
transSize	20/10/5 pages	20 pages	20 pages	16 pages
hotBounds	-	p to p+49, p=50(i-1)+1	1 to 400	p to p+24, p=25(i-1)+1
coldBounds	All of DB	Rest of DB	Rest of DB	1001 ~ 2000
hotAccProb	-	0.8	0.8	0.8
coldAccProb	1.0	0.2	0.2	0.2
hotWrProb	-	0.2	0.1	0.2
coldWrProb	0.2/0.4/0.6	0.2	0	0.0
perPageInstr	30,000	30,000	30,000	30,000
thinkTime	0	0	0	0

## V. RESULTS AND DISCUSSION

We use the cost and workload settings described in TABLE II and TABLE III to obtain the following results. Similar to previous studies [1][3][7][8][13][16][17][18], the overall system throughput (transactions per second) and abort rate are the major performance metric in this paper. 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 [13]. 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 impact of transaction size on abort rate. 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.

#### A. *The UNIFORM Workload*

The UNIFORM workload has no spatial (per-client) locality. The accesses are distributed uniformly throughout the whole database. All pages are accessed with the same write probability (*coldWrProb*). As SADCC is fundamentally distinguished from ADCC and AACC in self-tuning speculation, we first examine the impact of self-tuning speculation under different levels of contention with the transaction size as 20, and 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. Finally, we study the impact of transaction size on the abort rate with *coldWrProb=0.2*.

*Impact of contention/data-sharing:* Figure 6 shows that at *coldWrProb=0.2*, as the client population increases from 10 to 40, the throughputs of all protocols drop. At the client number as 10 and 20, the three protocols display similar behavior. As the client number increases from 20 to 30 and 40, AADCC gradually outperforms both ADCC and AACC. When *coldWrProb* varies from 0.2, 0.4 to 0.6, the throughput gap between SADCC and ADCC ranges from 0.9% to 4.2% under low data contention (*coldWrProb=0.20*), changes from 2% to 8.6% under medium data contention (*coldWrProb=0.40*), and varies from 2.8% to 8.9% under high data contention (*coldWrProb=0.60*); while the gap between SADCC and AACC corresponds to 0.7% to 6.9%, 6.5% to 16%, and 9.6% to 18.4% respectively.

This performance trend can be explained from several aspects. First of all, 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 to the traditional server-based communication path, both SADCC and ADCC reduce this latency using direct client-client (P2P) communication. Earlier discovery of data conflict can improve the performance [8].

Second, AACC relies on the server for callback handling and data forwarding, while SADCC and ADCC partially offloads this function from the server to the clients. When the contention increases, the server in AACC has to handle more lock callbacks, while in SADCC and ADCC, many more invalidations occur directly between the clients. Therefore, the server is more heavily loaded in AACC than in SADCC and ADCC. When the contention is high, increasing the client population causes the performance to degrade further.

Third, the P2P communication path also reduces the potential conflict interval. Consider the example in Figure 1, assuming that page  $m$  is not in use until A wants to update it at time  $t_j$ . On average, the invalidation request from  $A_1$  in SADCC and ADCC reaches  $A_2$  at  $(t_j+T)$ , while the corresponding callback message in AACC arrives at  $A_2$  at  $(t_j+2T)$ . Therefore, in SADCC and ADCC if  $A_2$  accesses or update the data during the interval  $[t_j, t_j+T]$ ,  $A_1$ 's update will be blocked or aborted. However, this interval becomes  $[t_j, t_j+2T]$  in AACC. The reduction in the potential blocking/abort window by SADCC and ADCC produces less contention than AACC (Figure 7).

Fourth, compared to ADCC, SADCC has lower message overhead by using piggyback messages, e.g. scenarios 1 and 3 in Figure 4.

Finally, when the contention increases, both ADCC and SADCC become *blindly* optimistic. In contrast, SADCC adaptively switches to pessimistic if the speculation risk is high, which reduces the abort rate due to mis-speculation. The integration of self-tuning speculation and P2P communication makes SADCC have the highest throughput (Figure 6) and lowest abort rate (Figure 7) among the three protocols.

Compared to AACC, both SADCC and ADCC need to piggyback additional control information (256 bytes) between the server and clients for maintaining the client caching directory consistency, e.g. scenario 4 in Figure 4. This piggyback information does not generate additional messages but does consume additional network bandwidth. However, compared to the typical message size of a data page ( $\geq 4$  Kbyte), this overhead is generally low. For example, Figure 8 shows that, in average, SADCC and ADCC have about 4.8% bandwidth overhead compared to AACC. Consequently, with contention, this small overhead does not prevent SADCC and ADCC from outperforming ADCC in a high-speed network environment.

*Impact of transaction granularity:* The aborts in asynchronous schemes, such as SADCC, ADCC and AACC, typically are resulted from either deadlock or mis-speculation. Due to temporal locality between read/write operations within the same transaction, the write/write conflict becomes important as the contention increases. Similar to AACC, SADCC and ADCC also employ two optimization techniques, Sneak-Through and Blocking Reversal [13], to reduce deadlock-abort. However, due to lack of global data-sharing information, the self-tuning speculation and P2P communication used in SADCC for reducing abort rate cannot be applied on AACC without substantial changes.

Figure 9 shows that, with the write probability as 0.2, when the transaction size varies from 20 to 5, the mis-speculation abort becomes dominant over the deadlock abort. For example, when the transaction size is 5, the abort is almost purely due to mis-speculation. This trend follows the intuition. The shorter is the transaction size, the faster does a transaction finish. In other words, in average, a transaction with smaller size has shorter blocking time. Therefore, the lower probability is the deadlock to occur.

The direct P2P communication in SADCC and ADCC helps reduce the abort rate because of the earlier discovery of read/write and write/write conflicts. Using the self-tuning speculation optimization, SADCC further reduces the abort rate due to mis-speculation by more than 30% in average (Figure 9). It is noted that, when SADCC switches from optimistic to pessimistic, it reduces the abort rate due

to mis-speculation but might generate additional blocks. A small portion of these blocks might lead to additional deadlock-aborts. For example, when client population is 30 and transaction size is 20, SADCC has the highest deadlock-abort rate among the three protocols. However, compared to the large reduction in mis-speculation aborts, these additional deadlock-aborts are insignificant. Figure 9 shows that SADCC has the lowest overall abort rate among the three protocols.

From the above discussion, it can be seen that SADCC improves throughput and reduces abort rate compared to ADCC and AACC. The increased data contention in the UNIFORM workload makes the integration of self-tuning speculation and P2P communication an important aspect for producing these improvements.

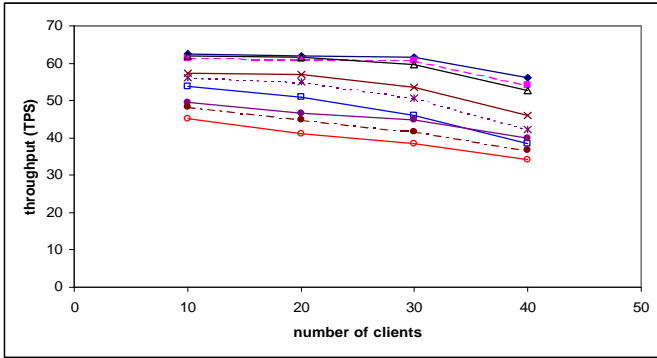


Figure 6. Throughput (UNIFORM).

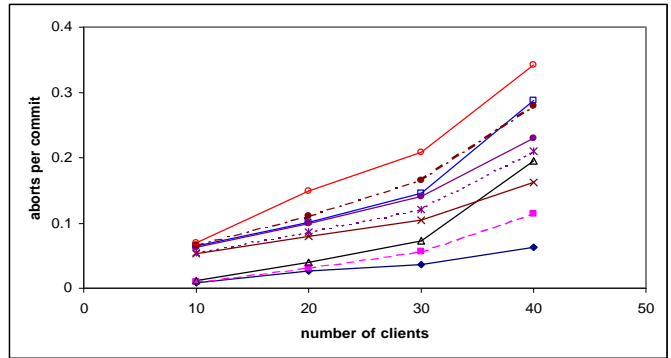


Figure 7. Aborts per commit (UNIFORM).

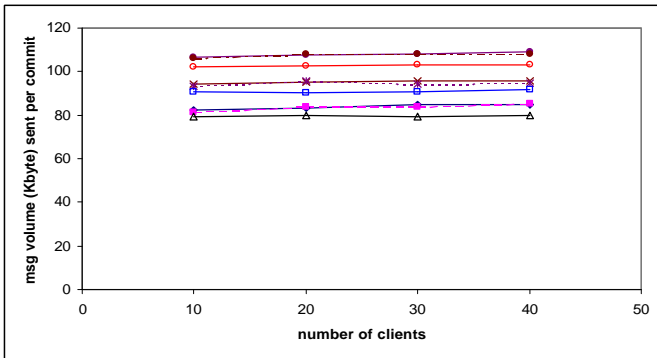


Figure 8. Message volume (Kbyte) sent per commit.

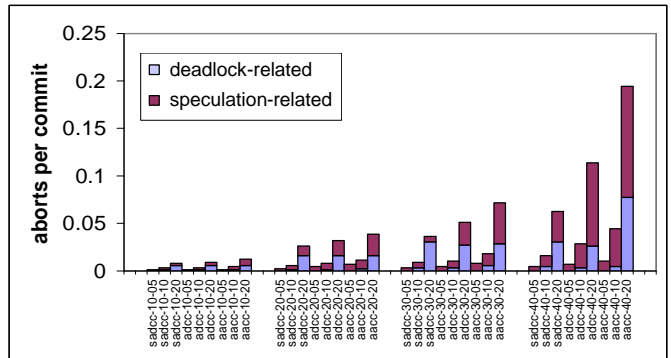
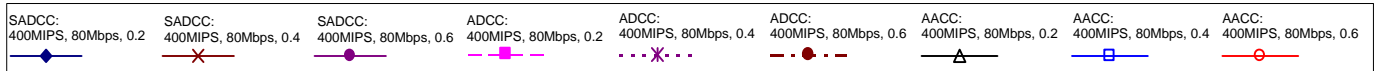


Figure 9. Abort rate (UNIFORM). In notation  $x$ - $y$ - $z$ ,  $x$ ,  $y$  and  $z$  denote the protocol name, client population and transaction size, respectively.



### B. The HOTCOLD Workload

The HOTCOLD workload has high spatial locality and moderate write/read and write/write sharing among the clients. 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.

*Impact of a fast CPU:* When the server CPU speed increases from 200 to 400 MIPS, SADCC, ADCC and AACC show improvements in throughput. The reduction in transaction execution time reduces the write/read conflict blocking time in all algorithms. Figure 10 shows that for 10 to 40 clients, doubling the server CPU speed increases the throughput in AACC around 28% on average, while the improvement in ADCC is only about 20%. With slow CPUs, the server in AACC is heavily used, i.e. the server CPU's utilization increases rapidly and exceeds 86% when the number of clients reaches 20 (Figure 11), which explains why the fast server CPU helps AACC more than SADCC and ADCC.

*Impact of a fast network:* It has been shown that there exists an optimal number of clients to achieve the highest overall system throughput, because the overhead for concurrency control gradually catches up with the gain due to client caching as the number of clients increases [18]. When the network bandwidth increases from 80 to 800 Mbps, SADCC and ADCC not only display significant increase for all client population (an average of 62%), but also improve the scalability by moving the optimal number of client population to 20. However, AACC increases the throughput by 43%, and more important, the overall throughput consistently drops as the number of clients increases from 10 to 40. It is noted that, with fast network, the server CPU's utilization in AACC quickly reaches 91% at the client number as 20 and the server almost saturates with the utilization as 97% for client number as 30 (Figure 11). On the other hand, the increased bandwidth improves both the server and average clients CPU utilization in SADCC and ADCC (Figures 11 and 12). Consequently, SADCC and ADCC display significant improvements in throughput (Figure 10).

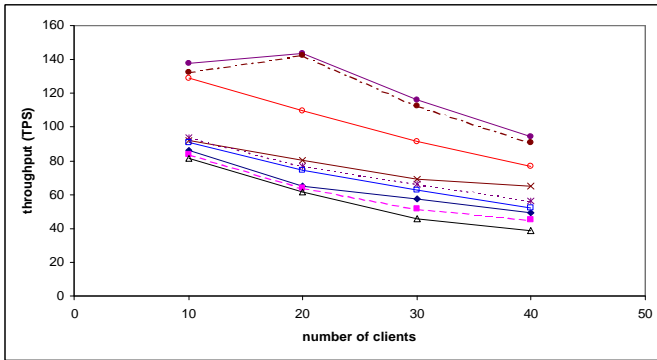


Figure 10. Throughput (HOTCOLD).

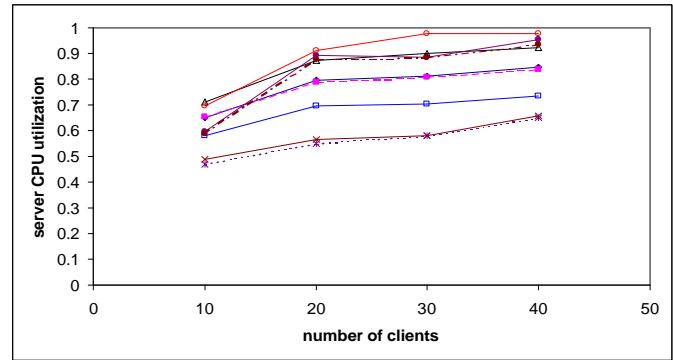
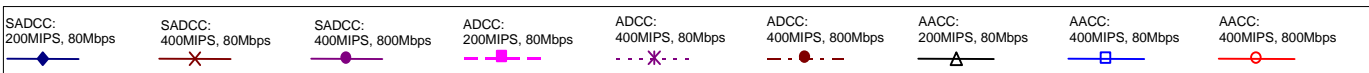


Figure 11. Server CPU utilization (HOTCOLD).



The server is often eventually the bottleneck for performance and scalability due to the excess demands for pages [7][15]. However, because the server is always on the critical path for enforcing concurrency control in AACC, 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, SADCC and ADCC will benefit more from expected technology advancements than AACC. At the same time, SADCC performs better (an average of 8%)

than ADCC at the large number (i.e. 40) of clients, since the contention increases as the client population increases and the self-tuning speculation helps SADCC reduce wasted work.

### C. The HICON Workload

In HICON workload, all clients access the shared data region 80% of the time and the rest of the database 20% of the time. All write operations are conducted on the shared data. It displays a skewed data access pattern. This workload is not usually present in data-shipping applications [7]. We include this workload primarily to examine the robustness of SADCC under extreme data contention situations.

SADCC, ADCC and AACC suffer from increased conflict rates as clients are added. Figure 13 shows that SADCC and ADCC outperforms AACC consistently since AACC suffers more from data conflicts. The shorter communication path in SADCC and ADCC also reduces the blocking time. SADCC displays slightly higher throughput than ADCC because of the self-tuning speculation and lower message overhead (e.g. scenarios 1 and 3 in Figure 4).

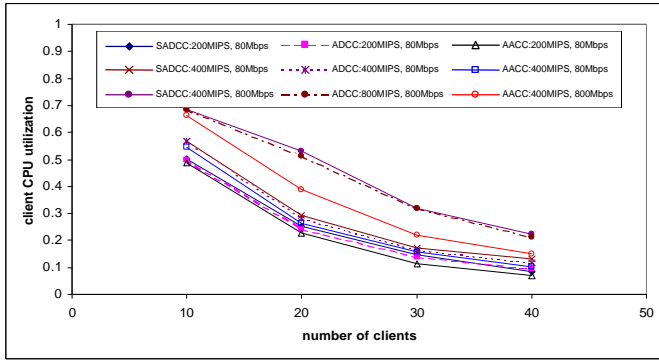


Figure 12. Server CPU utilization (HOTCOLD).

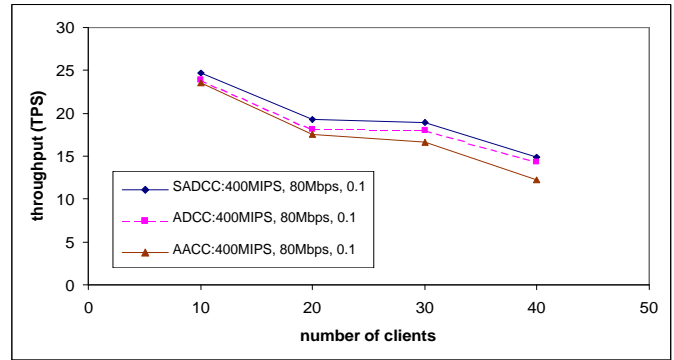


Figure 13. Throughput (HICON).

Increasing the data contention leads to increases in the block and abort rates (Figure 14). Consequently, all three schemes exhibit thrashing behavior. For example, for 40 clients, SADCC, ADCC and AACC produce about 0.32, 0.33 and 0.42 aborts per commit, respectively. SADCC and ADCC have lower abort rates than AACC. 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]. More importantly, the deadlock abort overhead dominates the other overheads (Figure 14). Since all write operations in the HICON workload can be conducted on a small set of shared data only, the results (Figure 14) imply that, although most updates are frequently occurred on a small set of data items, aborts due to false speculation are not common. This indicates that using the number of write operations or conflicts for a unit time [6] is not an accurate criterion to identify the “hot/cold” data spots for asynchronous/synchronous cache consistency algorithms.

#### D. The PRIVATE Workload

The PRIVATE workload has the highest spatial 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 15 shows that, while SADCC and ADCC perform similarly, AACC slightly outperform ADCC at large number of population. As no write/read or write/write data sharing exists, no direct client-client communication occurs. SADCC and ADCC essentially downgrade to the server-based communication. The primary overhead of SADCC and ADCC over AACC is the additional network bandwidth for small control message (256 bytes) for maintaining client directory consistency. This overhead increases as the client population increases. However, compared with typical size of a data page (4K bytes), this overhead is low. Therefore, SADCC and ADCC do not exhibit significant downgrade. In average, the performance gap is 5.3% when the network bandwidth is 80 Mbps. This gap drops to 2.3% when the network bandwidth increases to 800 Mbps.

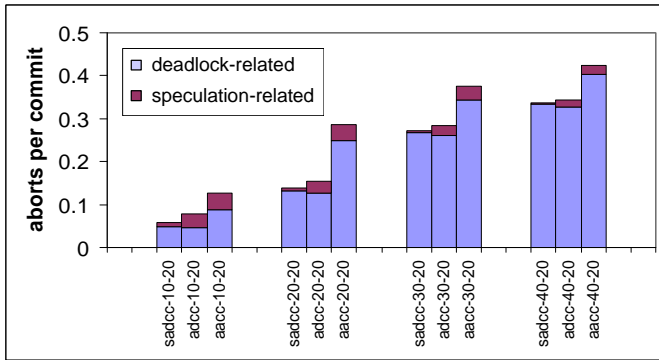


Figure 14. Abort rate (HICON). In notation  $x$ - $y$ - $z$ ,  $x$ ,  $y$  and  $z$  denote the protocol name, client population and transaction size, respectively.

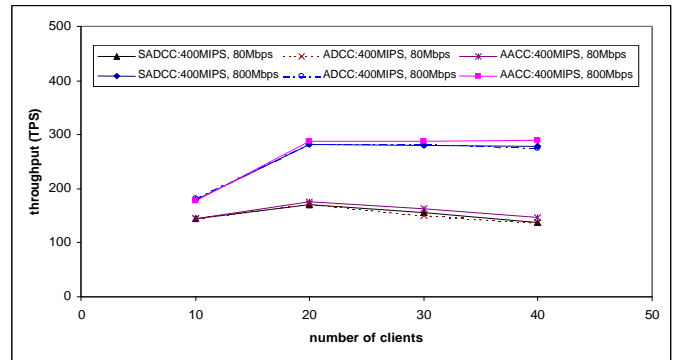


Figure 15. Throughput (PRIVATE).

#### E. Discussion

An important feature of AACC is that a client uses piggyback message to inform the server of updates on *private-read* locked pages. The objective was to reduce the message overhead [13]. However, it may lead to **late discovery** of conflict under high contention environment. For example, Figure 16-a shows an example of late-abort due to this piggyback approach. Client A proceeds to update (Spec wr) a locally cached page which is under *private-read* lock at time  $t_0$  and informs the server via a piggyback message (pgybk LE). At the same time, client B sends a read request (rd) which arrives at the server at  $(t_0+15)$ . Upon receiving the read request from B, the server forwards the data to B (data) but informs A to change the page state from *private-read* to *share-read* via a piggyback message (to simplify the illustration, this message is not shown on Figure 16-a). If A's piggyback message (pgybk LE) arrives at the server later than  $(t_0+15)$ , then A's speculation is not allowed to commit before B. Furthermore, if B also wants to update the page after

caching that page, B proceeds the update and informs the server using an explicit message (LE) which arrives at the server at  $(t_0+45)$ . As a result, at least one of the two clients has to abort (if the message (pgybk LE) arrives at the server before  $(t_0+45)$ , B has to abort; otherwise, A has to abort). In contrast (Figure 16-b), if A informs its speculation to the server using an explicit message (LE), then A's update will not be blocked by B's read request (rd) and no additional aborts occur.

Another important hallmark is that AACC requires commit-time validation. When client A finishes the execution phase, it informs the server of the intention to commit. Before receiving the acknowledgement from the server, A has to stop. The server checks whether A can go ahead with its commit or not. If A has updated a page that has been read by another client, the server blocks A. If A has updated a page that has been read and then updated by another client (e.g. Figure 16-a), the server informs A to abort. The necessity for commit-time validation makes AACC more like the deferred schemes, such as AOCC [1].

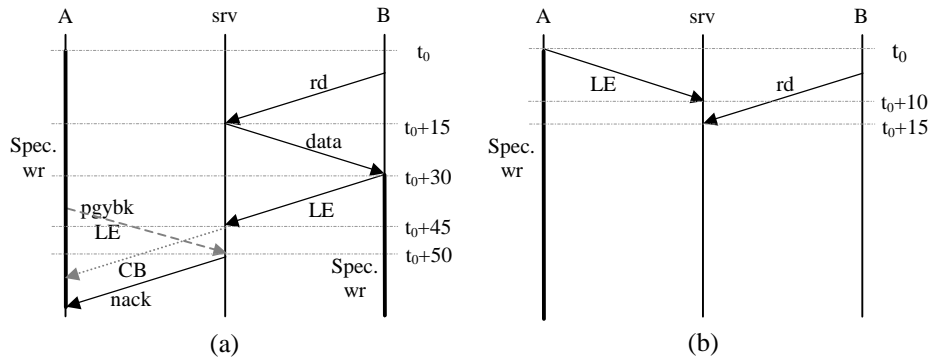


Figure 16. An example comparing the piggyback message approach in AACC to an asynchronous approach without using piggyback messages, where CB and LE denote callback and lock-escalation requests, respectively.

Compared to AACC, ADCC uses parallel communication with simultaneous client-server and client-client communication, which may generate more messages. However, a significant portion of messages is handled in parallel in ADCC but in sequential in AACC. Note that due to the different overheads associated with the sequential and parallel message, the total number of messages sent is not an accurate metric for comparing the performance of these two schemes.

SADCC is developed based on ADCC but advances further. SADCC self-tunes between optimistic and pessimistic by statistically evaluating the speculation risk. To apply the self-tuning technique in a real-world workload, the information of access pattern can be obtained by dynamically monitoring the database. For example, by counting the accessing frequency within a certain time window, the hot/cold regions and write/read probabilities can be profiled.

In addition to the self-tuning speculation, piggyback messages are also employed in SADCC to reduce the message overhead. However, the design on using piggyback messages in SADCC is fundamentally different from AACC. Using piggyback messages reduces the total number of messages but may lead to significant communication delay, even if the network traffic is low. As discussed

in Figure 16, the piggyback approach used in AACC requires commit-time validation and may result in late abort. In contrast, SADCC *piggybacks* acknowledgements for *successful speculative updates* but sends *explicit* messages for *speculation requests* and *unsuccessful speculative updates*. If the network traffic is not the bottleneck, this design has little impact on performance, since the successful speculative client does not block when it wants to update a cached data. The network technology has evolved from 10Mbps in the early 1990s to Gigabits today. The bandwidth utilization in today's Gigabit networks is typically low [9][14]. The design of using explicit messages for speculation request, invalidation, and negative responses in SADCC is due to the fact that earlier discovery of conflicts can lower the abort rate and cost [8] and the substantial advance in network technology.

To summarize, the key strength that distinguishes SADCC from ADCC and AACC is the self-tuning speculation technique. ***By adaptively switching between optimistic and pessimistic consistency control, SADCC reduces the speculation cost and improves the performance in a high contention environment.***

The second strength of SADCC is the use of P2P communication for detecting data conflicts, which is similar to ADCC. 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. Compared to AACC, ***the shorter communication path leads to fewer aborts and higher throughput in SADCC in a high contention environment.***

The third strength of SADCC 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, SADCC can better exploit client resources. In AACC, the server is the only source for enforcing cache consistency and providing data. ***SADCC removes the server partially from the critical path, which makes SADCC perform better for large number of clients.***

Nevertheless, compared to AACC, the advantages of SADCC do not come for free. Similar to ADCC, SADCC 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. For extreme situations which require a large number of clients to cache data, SADCC and ADCC can reduce the overhead by using a coarse directory representation, i.e. using a flag to represent a group of clients [10]. In addition, both SADCC and ADCC piggyback control information (256byte) to maintain the directory consistency, which consumes additional network bandwidth. Compared to the typical message size of a data page ( $\geq 4$  Kbytes), the bandwidth overhead is low. Moreover, there has been significant advance in network technology during the past decade. A recent study shows that, with 2GB networks, typically less than 20% of the peak bandwidth is used [14].

## VI. RELATED WORK

When a client wants to update a cached page copy in data-shipping systems, in order to maintain cache consistency, the server must be informed of this write intention sometime prior to the transaction commits. There are two classes ranging from pessimistic (synchronous) to optimistic (asynchronous and deferred) techniques representing different tradeoffs between write intention declaration overhead and possible transaction aborts.

The pessimistic approaches, such as Callback Locking (CBL) [8] and Caching Two-Phase Locking (C2PL) [16], require clients to contact the server at the time that they first decide to update a page to which they do not currently possess write permission. The clients have to block before the server response. In the optimistic approaches, such as Adaptive Optimistic Concurrency Control (AOCC) [1], Cache Locks [17], Notify Locks [17], and Optimistic Two-Phase Locking (O2PL) [3], declaration of write intentions are deferred until the transaction finishes its execution phase. The asynchronous approaches are a compromise between the pessimistic and optimistic approaches. The asynchronous approaches, e.g. Active Data-aware Cache Consistency (ADCC) [18], Asynchronous Avoidance-based Cache Consistency (AACC) [13] and No-Wait Locking (NWL) [16], allow client to proceed to write the local copy under the assumption that the write intention declaration will succeed. If this optimism turns out to be incorrect, then the transaction must abort. TABLE IV compares SADCC with the algorithms proposed during the last decade.

TABLE IV. COMPARISON OF CACHE CONSISTENCY ALGORITHMS.

Protocols	Invalid access prevention	Validity check initialization	Optimistic/Pessimistic	Serialization mechanism	Consistency granularity	Communication path
CBL	Avoidance-based	Synchronous	Pessimistic	One-tier dir, Lock-based	Page	Srv-based
AACC	Avoidance-based	Asynchronous	Optimistic	One-tier dir, Lock-based	Page + object	Srv-based
ADCC	Avoidance-based	Asynchronous	Optimistic	Two-tier dir, State-based	Page	P2P + Srv-based
SADCC	Avoidance-based	Asynchronous	Self-tuning	Two-tier dir, State-based	Page	P2P + Srv-based
Notify Locks	Avoidance-based	Deferred	Optimistic	One-tier dir, Lock-based	Page	Srv-based
O2PL	Avoidance-based	Deferred	Optimistic	One-tier dir, Lock-based	Page	Srv-based
C2PL	Detection-based	Synchronous	Pessimistic	One-tier dir, Lock-based	Page	Srv-based
NWL	Detection-based	Asynchronous	Optimistic	One-tier dir, Lock-based	Page	Srv-based
AOCC	Detection-based	Deferred	Optimistic	One-tier dir, Time-stamp	Page + object	Srv-based
Cache Locks	Detection-based	Deferred	Optimistic	One-tier dir, Lock-based	Page	Srv-based

The simple assumption that no other clients want to update the same page makes traditional asynchronous and deferred approaches *blindly* optimistic in high contention environments. A possible approach to reduce the high abort rate in optimistic schemes was also proposed in [1], which assumed the system can automatically detect which data are under high contention, then the system switches to a more pessimistic protocol for such data. However, no techniques were provided for detecting or predicting “hot spot” data.

The most recent published hybrid concurrency control scheme we have found is [6], where the data contention “temperature” is maintained for each item. The scheme totally relies on the server to set the temperature state as hot or cold, depending on the number of write operations for a unit time, and inform relevant clients the temperature state conversion. Then clients dynamically switch between a synchronous scheme, C2PL [16], and a deferred scheme, Cache Locks [17]. However, SADCC self-tunes between synchronous and asynchronous consistency control. The fundamental difference between the pessimistic (synchronous) and optimistic (deferred and asynchronous) schemes is the speculative update. Due to the lack of speculation, the pessimistic approaches encounter deadlock-related aborts only. Because clients defer all of their write notification messages until commit time, the deferred approaches encounter stale cache aborts only. By contrast, the asynchronous approaches encounter both deadlock and mis-speculation related aborts. SADCC focuses on reducing the mis-speculation aborts by self-tuning between synchronous and asynchronous consistency control. The mis-speculation aborts are mainly due to the simultaneous speculative update on locally cached data, which therefore leads to aborts of unsuccessful speculative clients. It should be noted that a conflict between a write on a locally cached data and another remote write on non-cached data does not result in mis-speculation aborts. ***If the data-sharing does not exist, a speculative update scheme is not likely to produce aborts due to false speculation.*** Consequently, the update frequency [6] does not accurately catch the “hot/cold spot” data which is more appropriate for synchronous/asynchronous schemes. For example, in our study, if a “hot” data is exclusively cached by a client, adopting a pessimistic approach can only deteriorate the performance. In addition, due to the passive nature of this scheme [6], it does not catch the changing user data access pattern accurately [13]. Without the global sharing information of cached data, it is difficult for clients to detect or predict “hot/cold spot” data accurately for synchronous/asynchronous consistency control.

The granularity of concurrency control (locking and callbacks) leads to two categories of algorithms, adaptive and non-adaptive. The non-adaptive schemes conduct locking and callbacks at single granularity, i.e. page or object, while the adaptive schemes adaptively switch between the page and object level. Franklin [7] demonstrated that non-adaptive schemes are the best overall choices. Carey *et. al.* [5] showed that adaptive schemes have better performance for workloads that exhibit fine-grained read/write sharing at the expense of additional complexity. It 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 [7].

## VII. CONCLUSIONS

An efficient cache consistency protocol, Self-tuning Active Data-aware Cache Consistency (SADCC), has been proposed for data-sharing DBMS architectures. Using P2P communication, SADCC reduces network latency for invalidation messages for read/write and write/write sharing by 50% compared to the server-based communication scheme, while only increasing the network bandwidth overhead by around 4.8%. By statistically evaluating the speculation risk, SADCC adaptively switches between optimistic and pessimistic. By integrating self-tuning speculation and parallel communication (client-client and server-client), SADCC not only improves the throughput but also reduces the abort rate. The experimental study shows that SADCC has an average of 14% higher throughput than AACC and 6% higher throughput than ADCC under the high contention workloads; while under the non-contention workload, both SADCC and ADCC display a slight reduction (2.3%) in throughput compared to AACC with a high-speed network environment.

## ACKNOWLEDGMENT

The authors thank Michael J. Franklin and Kaladhar Voruganti for some helpful suggestions and discussion in the early stages of this research, and anonymous reviewers for comments. This project was supported by the University of Minnesota Digital Technology Center (DTC) Intelligent Storage Consortium (DISC), and the Minnesota Supercomputing Institute.

## REFERENCES

- [1] Adya, A., R. Gruber, B. Liskov, and U. Maheshwari, "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., M. Franklin, and A. Tomasic, "Dynamic query operator scheduling for wide-area remote access," *Distributed and Parallel Databases*, vol. 6(3): pp. 217-246, 1998.
- [3] Carey, M. J., M. J. Franklin, M. Livny, E. J. Shekita, "Data Caching Tradeoffs in Client-Server DBMS Architectures," in *Proceedings of the ACM SIGMOD*, pp. 357-366, May 1991.
- [4] Carey, M.J., D. DeWitt, and J. Naughton, "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., M.J. Franklin, and M. Zaharioudakis, "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] Chung, I., J. Lee, and C. Hwang, "A Contention Based Dynamic Consistency Maintenance Scheme For Client Cache," in *CIKM Conference Proceedings*, pp. 363-370, 1997.

- [7] Franklin, M.J., *Client Data Caching: A Foundation for High Performance Object Database Systems*, Kluwer Academic Publishers, Boston, MA, 1996.
- [8] Franklin, M.J., M.J. Carey, and M. Livny, "Transactional client-server cache consistency: alternatives and performance," *ACM Transactions on Database Systems*, vol. 22(3), pp. 315-363, September 1997.
- [9] Fritz, J., "Gigabit Ethernet hits second gear," <http://www.nwfusion.com/research/2000/0320revgig.html>
- [10] Laudon, J. and D. Lenoski, "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.
- [11] MCI Corp., "Network Latency Statistics, <http://global.mci.com/about/network/latency/>
- [12] Objectivity Database Systems Inc. *Objectivity/DB Technical Overview*. <http://www.objectivity.com/DevCentral/Products/TechDocs/pdfs/techOverview6.pdf>
- [13] Ozsü, M.T., K. Voruganti, and R. Unrau, "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.
- [14] Pargal, S., *Future Technologies for Storage Networks*. Compellent Technologies Inc. April 2003. <http://www.dtc.umn.edu/diskcon/>
- [15] Silberschatz, A., H. Korth, and S. Sudarshan, *Database System Concepts*, 4<sup>th</sup> ed., McGraw Hill, 2001.
- [16] Wang, Y., and L.A. Rowe, "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.
- [17] Wilkinson, K., and M.-A. Neimat, "Maintaining consistency of client-cached data," in *Proceedings of the Conference on Very Large Data Bases (VLDB)*. Brisbane, Australia, pp. 122-133, August 1990.
- [18] Wu, K., P. Chuang, and D.J. Lilja, "An Active Data-aware Cache Consistency Protocol for Highly-Scalable Data-Shipping DBMS Architectures," *ACM International Conference on Computing Frontiers*, pp. 222-234, April 2004.