An Adaptive Data-Shipping Architecture for Client Caching Data Management Systems

Kaladhar Voruganti (kaladhar@us.ibm.com)
IBM Almaden Research Laboratory, San Jose, USA

M. Tamer Özsu (tozsu@uwaterloo.ca)
University of Waterloo, School of Computer Science, Waterloo, Canada

Ronald C. Unrau (rcu@yottayotta.com)
YottaYotta, Edmonton, Canada

Abstract. Data-shipping is an important form of data distribution architecture where data objects are retrieved from the server, and are cached and operated upon at the client nodes. This architecture reduces network latency and increases resource utilization at the client. Object database management systems (ODBMS), file-systems, mobile data management systems, multi-tiered Web-server systems and hybrid query-shipping/data-shipping architectures all use some variant of the data-shipping. Despite a decade of research, there is still a lack of consensus amongst the proponents of ODBMSs as to the type of data shipping architectures and algorithms that should be used. The absence of both robust (with respect to performance) algorithms, and a comprehensive performance study comparing the competing algorithms are the key reasons for this lack of agreement. In this paper we address both of these problems. We first present an adaptive data-shipping architecture which utilizes adaptive data transfer, cache consistency and recovery algorithms to improve the robustness (with respect to performance) of a data-shipping ODBMS. We then present a comprehensive performance study which evaluates the competing client-server architectures and algorithms. The study verifies the robustness of the new adaptive data-shipping architecture, provides new insights into the performance of the different competing algorithms, and helps to overturn some existing notions about some of the algorithms.

Keywords: Client-Server DBMSs, Data-Shipping, Recovery, Cache-Consistency, Data Transfer, Caching, ODBMSs, Buffer Management, Pointer Swizzling

1. Introduction

Fine-tuning the performance of a database management system (DBMS) is a difficult task due to the complex interaction between the various components of a DBMS. Adaptive systems that can dynamically adapt to changing workloads and system configurations have been identified as a high priority requirement [3, 18, 17].

Data-shipping and *query-shipping* are two popular types of client-server data distribution architectures. In *data-shipping*, objects are retrieved from the servers into the client caches, and operations are performed on data in client caches. In *query-shipping* (also known as *function-shipping*), a query is shipped from the client to the server. The query is executed at the server

and the query result is shipped to the client. *Data-shipping* client-server architecture is emerging as a popular data distribution alternative because it helps to better utilize the hardware resources present at the clients. Furthermore, client caching also helps to reduce network latency because a client can prefetch useful data objects and store them in its cache before they are accessed by the client application. Therefore, *data-shipping* architecture is utilized in various configurations by file systems, multi-tiered Web server systems, hybrid query-shipping/data-shipping object-relational systems, and mobile device-based software.

In order to provide robust performance across varying workloads and system configurations, the need for adaptive systems was identified as a key requirement for *data-shipping* client-server ODBMSs [11]. However, even after a decade of research, currently no robust adaptive algorithms exist for the various components of a *data-shipping* systems, such as data transfer, cache consistency, recovery, buffer management and pointer swizzling. Hence, there are many competing proposals, each with its respective limited strengths, but there does not exist a single *data-shipping* architecture that satisfies the needs of all (or most) of the important workloads and system configurations.

In this paper we propose a general adaptive *data-shipping* architecture which consists of adaptive data transfer, cache consistency and recovery data management services. We also present a detailed performance study which shows that the adaptive *data-shipping* architecture is more robust with respect to performance than all of the existing ODBMS *data-shipping* architectures.

Furthermore, we also discuss how the new algorithms presented in this paper can be used by the current *data-shipping* ODBMSs, hybrid function-shipping/data-shipping object-relational systems, multi-tiered Web server systems and mobile systems.

1.1. Adaptive Data Transfer

From the very beginning of client-server ODBMS research it has been recognized that there is a need for an adaptive data transfer mechanism which can transfer either pages or objects from the server to the client [11]. It is not efficient to transfer pages from the server to the client when the application data access pattern does not match the way in which data is clustered on disk pages. On the other hand, the efficiency of a grouped object server approach depends upon the accuracy of the object grouping mechanism and it is difficult to design a general purpose object grouping algorithm that is robust under all application data access patterns [11, 23]. Hence, it would be very useful to design an adaptive data transfer approach which could dynamically switch between transferring pages or object groups.

1.2. ADAPTIVE CACHE CONSISTENCY

The fundamental problem with current ODBMS client-server cache consistency algorithms is that they can either provide good performance or low abort rate, but not both. Algorithms that provide good performance are optimistic in nature and, therefore, inherently abort prone [1]. High abort rates are not acceptable in interactive user environments. Similarly, algorithms which provide a low abort rate are too conservative and, therefore, they incur high blocking and messaging overheads [9]. Thus, there is a need for an adaptive cache consistency algorithm that can provide both good performance as well as a low abort rate.

1.3. ADAPTIVE RECOVERY

The two prominent recovery approaches utilized by the client-server ODBMSs are the redo-at-server approach [6] and the ARIES approach [4]. In the redo-at-server approach the logs are sent to the server, which applies them to the correct data page. In the client-server ARIES approach, both the logs and the data pages are sent from the client to the server. The problem with the redo-at-server approach is that if the server buffers are contended, it can result in of a high number of reads of the data pages corresponding to the log records which are needed to apply the log records. The problem with the client-server ARIES approach is that it incurs a high network overhead, because, even if only a small portion of a page has been updated at the client, the entire page is returned to the server. Therefore, there is a need for an adaptive recovery algorithm which can reduce the problems associated with each of these recovery approaches.

1.4. PAPER CONTRIBUTIONS

The following are the key contributions of this paper:

- It provides a comprehensive survey of all the data-shipping architectures and algorithms that have been designed, within the context of ODBMSs, over the last decade.

- It then proposes a new adaptive hybrid server architecture which contains a new data transfer algorithm, a new cache consistency algorithm, and a new recovery algorithm. The resulting architecture attains the strengths of the existing systems while avoiding their weaknesses. Thus, the architecture proposed in this paper satisfies the long standing requirement of a robust data-shipping architecture [11]. The unifying theme amongst the proposed algorithms is that they dynamically adapt at runtime. The adaptive data transfer algorithm dynamically decides between

sending pages or objects between the clients and the server. The adaptive cache consistency algorithm dynamically decides between operating in a pessimistic (asynchronous) or an optimistic (deferred) manner. The adaptive recovery algorithm dynamically decides between redo-at-server and ARIES-CSA recovery approaches. A hybrid server architecture, where the clients and the server can efficiently handle both pages and objects, is a prerequisite for the adaptive algorithms and it is proposed in this paper.

- Another key focus of this paper is to better understand the interaction between the different sub-components of a client-server data-shipping architecture within the context of database transaction semantics. These interactions are complicated, and, to date, there has not been an integrated multi-user study examining them. This paper contains a performance study that compares the adaptive hybrid client-server architecture with many of the popular client-server architectures. The study verifies that the adaptive hybrid server architecture is indeed more robust, with respect to performance, than other architectures. It also provides new insights into the interactions between the different client-server subsystems and helps to overturn some existing notions about the data-shipping architectures and algorithms.

1.5. PAPER ORGANIZATION

The remainder of the paper is organized as follows:

- Section 2 presents an overview of the hybrid server architecture that is proposed in this paper. It also lists the work that has been previously done in data transfer, cache consistency, recovery, pointer swizzling, and buffer management areas. It then presents the details of the new adaptive data transfer, cache consistency and recovery algorithms. It also presents a new recovery algorithm extension that allows for the update of an object at both the clients and the server.

- Section 3 describes the experiment setup. It contains a description of both the system setup and the workloads.

- Section 4 presents a performance study that evaluates the new algorithms proposed in this paper. It contains an integrated performance study that compares the performance of the adaptive hybrid server architecture with the leading client-server architectures.

- Section 5 presents the key conclusions of the paper. Finally, it discusses how the work presented in this paper can be used by the emerging mobile systems, and multi-tiered web systems.

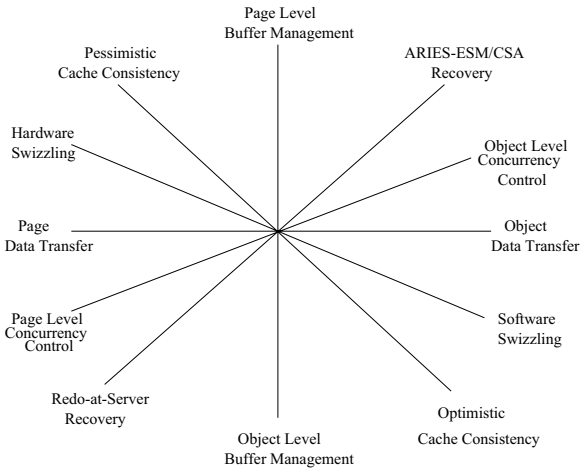

Figure 1. Client-Server System Components

2. Adaptive Hybrid Server Architecture

The new client-server architecture proposed in this paper is both adaptive and hybrid in nature. The architecture is adaptive because the data transfer, cache consistency, buffer management and recovery algorithms can adapt between the two end-points of their respective axes in Figure 1. The adaptive notions of these different components are described below in their respective subsections. The proposed hybrid server architecture is a prerequisite to the adaptive data transfer mechanism. Since the adaptive data transfer mechanism can transfer both pages and objects between the clients and the servers, the hybrid server architecture needs to efficiently handle both pages and objects. The data transfer mechanism dictates the types of algorithms that can be used by client buffer management, server buffer management, pointer swizzling, concurrency control and recovery system components. Therefore, the adaptive data transfer mechanism makes it necessary for these different system components to be hybrid in nature.

The adaptive data transfer, cache consistency and recovery algorithm details are provided in the subsections below. This section introduces the adaptive and hybrid client-server architecture. This new architecture consists of the following innovations:

- **Adaptive Data Transfer:** Clients and servers can transfer either physical pages or group of logical objects between themselves. There are advantages and disadvantanges of both approaches [11, 35]. Thus, in the hybrid server architecture, the server and the clients dynamically decide whether to transfer pages or objects among themselves. This notion of adaptive data transfer is very useful in mobile systems where the devices

have memory, CPU, battery power and bandwidth constraints. In these systems it is desirable to only receive useful data because battery power gets wasted when unused data is received. Adaptive data transfer mechanism is also useful in wide-area network systems that encrypt data before their transmission on the wire. It is not desirable to encrypt unused data.

- **Adaptive Recovery:** In client-server DBMSs, log records are generated at the clients during the execution of an application program. If the server does not persistently store the client generated log records, then it has to rely on the clients during its restart recovery. Even though storing client generated logs only on local client disks reduces the work that has to be performed by the server [31], this solution is unacceptable in most client-server environments because it is not desirable for reliable servers to depend on potentially unreliable clients to recover from server failures.

  If the server is managing log disks, then the clients can return only the updated page (whole-page logging) [38], return both updated pages and log records (ARIES approach) [26, 14], or return only the log records or updated objects (redo at server) [38]. Therefore, in the redo-at-server approach, the clients can return either updated objects, or log records. A previous performance study [38] has shown that there are advantages and disadvantages to each one of the different log transfer approaches. Thus, the hybrid server dynamically decides to operate in either ARIES mode or in redo-at-server mode. Furthermore, the recovery mechanism is hybrid because it can handle the case when there are either pages or objects present in the client cache. The recovery algorithm proposed in this paper also allows for the same object to get updated at both the server and the client within the same transaction. This feature is useful in hybrid data-shipping/function-shipping systems, where data items can get updated at both the clients and the server.

- **Adaptive Cache Consistency:** The DBMS cache consistency algorithms can be classified as avoidance-based or detection-based. Avoidance-based algorithms do not allow for the presence of stale cache data in the client caches, which is permitted in detection-based algorithms. Detection-based algorithms perform commit time validation to check if the transaction has accessed stale objects, and abort if this is the case. Stale data refers to the presence of an older version of data in a client's cache that has been concurrently updated and committed by another client. Avoidance-based and detection-based algorithms can, in turn, be classified as synchronous, asynchronous or deferred, depending upon when they inform the server that a write operation is performed. In synchronous algorithms, the client sends a lock escalation message at the time it wants

to perform a write operation and it blocks until the server grants it permission. In asynchronous algorithms, the client sends a lock escalation message at the time of its write operation but does not block waiting for a server response (it optimistically continues). In deferred algorithms, the client optimistically defers informing the server about its write operation until commit time. In deferred avoidance-based algorithms, the server blocks a client transaction at commit time if the client has updated an object that has been read by other clients [13]. Figure 2 depicts this classification along with some of the popular cache consistency algorithms.

	Synchronous	Asynchronous	Deferred
Avoidance-based	CBL [12] ACBL [9]	AACC [30]	O2PL [8]
Detection-based	C2PL [8]	NWL [36]	AOCC [1]

Figure 2. DBMS Cache Consistency Algorithms

In the hybrid server architecture, the clients and the server dynamically decide whether to send synchronous, asynchronous or deferred lock escalation messages. The adaptive cache consistency mechanism is useful because it provides a good combination of low abort rates and high performance. Low abort rates are useful in multi-tiered client-server environments where intermediate servers can also act as clients for other servers. Re-execution of the aborted transactions at the intermediate servers will utilize precious CPU and memory resources which, in turn, can degrade the performance of other concurrently executing transactions.

- **Hybrid Buffer Management:** As the clients and the server can transfer both pages and objects among themselves, the client and server buffer management components must be able to handle both pages and objects. The hybrid server architecture uses a dual page/object buffer at the clients [20]. The dual buffer at the client stores well clustered pages in the page buffer, and isolated objects from badly clustered pages in the object buffer. The hybrid architecture also uses a dual page/modified object buffer at the server [23]. The modified object buffer at the server stores the updated objects that have been returned by the clients, and the page buffer at the server acts as a staging buffer for temporarily storing the pages that have been requested by the clients. The notion of hybrid buffer complements an adaptive data transfer mechanism. However, even if only pages are transferred from the server to the client, a hybrid

buffer mechanism is useful because it allows the client to retain only useful objects from badly clustered pages. Hybrid buffers are useful in multi-tiered client-server architectures where the network costs are not an issue, but the client memory is scarce due to either device hardware constraints (mobile devices) or the client memory being shared by many users (like in a mid-tiered server in a multi-tiered Web architecture).

- **Hybrid Concurrency Control:** As the clients and the server can manipulate data at either the page or the object level, it is necessary to be able to lock data at both of these levels. Moreover, the concurrency control mechanism can dynamically escalate and de-escalate between page and object-level locks. Concurrency control algorithms which can dynamically switch between page and object-level locking have been previously developed for page servers [9].

- **Software Pointer Swizzling:** Object identifiers are an integral part of object DBMSs. An object identifier uniquely identifies an object in the database; they are system assigned and immutable. They can be either logical or physical. A physical object identifier (POID) stores the physical disk address of the object within the identifier itself, whereas a logical object identifier (LOID) has a level of indirection to point to the object. An intermediate mapping data structure (hash table or B tree) is usually employed to determine the location of an object from its LOID. LOIDs provide more flexibility with respect to object migration, replication and deletion than POIDs, but they incur mapping overhead that is not present with POIDs. The task of converting an object identifier stored on disk into a memory pointer is known as *pointer swizzling*. ODBMSs employ (hardware or software) pointer swizzling to improve the navigation operation response time.

  In the hardware pointer swizzling approach, the page level virtual memory facilities (page faulting mechanism) provided by the operating system are used to detect when a page has been accessed, and the page is sent from the server and loaded into the client cache. All the pointers present in a page are eagerly swizzled to point to the target virtual memory frames corresponding to the target pages. However, these pages are only brought into the client's cache when the pointers pointing to the page are actually accessed. In the software swizzling approach, a function call interface is provided to the client applications to access the pointers. The function code performs residency checks and dereferencing of pointers.

  In the hybrid server architecture, since the client cache might contain either pages or objects, the pointer swizzling mechanism cannot manage data solely at the page level. Since the hardware pointer swizzling

mechanism relies on operating system provided page level support, it cannot efficiently provide object-level buffering or concurrency control. Therefore, the software pointer swizzling mechanism is used by the new architecture, which adopts the mechanism that is used by the SHORE [6]. In the software pointer swizzling mechanism, there is a level of indirection between the source and the target object (via an object table).

Thus, as discussed above, it makes sense to view the new hybrid server architecture that we propose as consisting of useful services (data transfer, cache consistency, recovery, buffer management) that can be utilized either individually or in combination by many different types of client-server data-shipping architectures.

Figure 3 summarizes the client-server DBMS data-shipping research that has been performed over the last decade on data transfer, cache consistency, buffer management, recovery and pointer swizzling. The hybrid architecture proposed in this paper adds to this collection of research effort. A comprehensive discussion of all of the previous algorithms can be found [34, 35].

Year	CC				BM		RY		PS		DT	
	P/LA	P/HA	O/LA	O/HA	P	O	P	O	P	O	P	O
1990	[39]							[21]			[11]	[11]
1991	[8]	[36]					[14]	[22]				
1992	[12]											
1993												
1994	[9]		[9]		[20]		[26]		[37]		[29]	[15]
1995		[1]		[1]	[16]	[16]	[38]				[4]	
1996	[13]						[31]			[23]		[23]
1997					[10]							
1998	[30]											
1999			[35]					[35]			[35]	[35]
2000						[5]						
2001					[19]							
2002						[24]						

Figure 3. A decade of research into page and object server data-shipping architectures (CC: Cache Consistency, BM: Buffer Management, RY: Recovery, PS: Pointer Swizzling, DT: Data Transfer, P: Page Server, O: Object Server, HA: High abort, LA: Low abort)

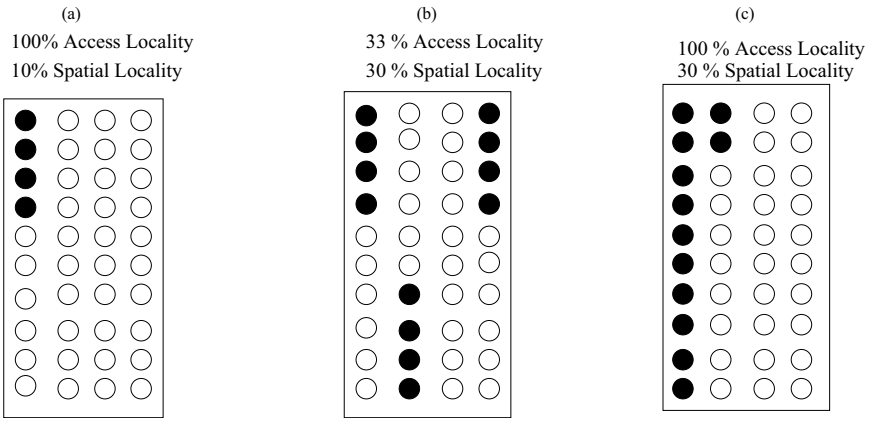

Figure 4. Different Locality Combinations

2.1. DATA TRANSFER

Page servers [11] and object servers [23] are currently the two most prominent client-server ODBMS architectures. Page servers send disk pages from the server to the client, while object servers send logical object groups. Previous performance studies have shown that the performance of the data transfer mechanisms of both page server and object server architectures suffers for certain important workloads and system configurations [11, 9, 23]. This section presents an adaptive data transfer mechanism which builds upon the strengths of both page and object data transfer approaches while avoiding their weaknesses. The adaptive data transfer mechanism is described in two sections. The first section describes the concept of data clustering, which plays a major role in determining the performance of the different data transfer algorithms. The second section provides an intuitive overview of the adaptive data transfer algorithm.

2.1.1. *Data Clustering*

Data clustering [11, 32] refers to how well the application data access pattern matches data placement on disk. Since page servers transfer disk pages from the server to the client, the data clustering pattern is an important performance determining factor for page servers. Transferring well-clustered disk pages, which match the application data access pattern (good clustering), helps the page server to take advantage of spatial locality and, thus, prefetch useful objects that will be accessed in the future. However, badly clustered disk pages, which do not match the application data access pattern, degrade page server performance by reducing effective network utilization and client buffer utilization. In this paper, data clustering is defined by *spatial locality*, *temporal locality* and *access locality* parameters. These probabilistic values are

specified with respect to a particular application. That is, a particular page can be viewed to have both good spatial locality with respect to one application, and bad spatial locality with respect to another application. The definitions of the three locality values are as follows:

- **Spatial Locality:** is the ratio of the number of bytes accessed in a page to the size of the page. As shown in Figure 4(a), spatial locality of 10 percent for a 4 Kilobyte sized page with 100 byte sized objects means 4 objects on the page have been accessed, and, as shown in Figures 4(b) and 4(c), a spatial locality of 30 percent means 12 objects on the page have been accessed.

- **Access Locality:** is the ratio of number of contiguously accessed bytes in a page to the total size of the page. For example, for a 4 Kilobyte sized page with 100 byte sized objects, access locality of 100 percent in conjunction with the spatial locality of 30 percent means that 12 contiguous objects on the page are accessed [refer to Figure 4(c)] and, an access locality of 33 percent with a spatial locality of 30 percent means that only 4 out of the 12 objects are accessed in a contiguous manner on the page [refer to Figure 4(b)].

- **Temporal Locality:** is defined as the probability that the previously accessed bytes on a page will be accessed again when a subsequent access is made to the same page. For example, for a 4 Kilobyte sized page with 100 byte sized objects, a temporal locality of 100 percent in conjunction with 10 percent spatial locality and 50 percent access locality means that both the first and the subsequent accesses to the page (separated by accesses to other pages) access the same 4 objects, whereas a temporal locality of 50 percent means that there is a 50 percent chance that the subsequent accesses to the page will access new objects on the page.

Spatial, temporal and access localities together determine the relationship between the data access pattern and the data placement on disk. These localities together represent how data is clustered and they are specified as percentage values between 0 and 100 percent. Data clustering is an important parameter which plays a key role in determining the performance characteristics of page and object servers. The three locality values described in this section are varied in the experiment process to analyze the performance of page and object servers under different clustering scenarios.

2.1.2. *Overview of Adaptive Data Transfer Mechanism*

The client initially (during a cold start) sends an object identifier and requests the corresponding data page, on which the object resides, from the server. The server returns the requested page to the client. Upon receiving the page,

the client stores the page in its page buffer and keeps track of the number of objects that have been accessed in that page. If the number of used objects is low, and if there is a need to eject the data page from the client buffer due to data contention, the client copies the objects that are in use into its object buffer and ejects the page. The goal of the dual buffering strategy is to try to increase the client buffer utilization and reduce the client cache miss overhead. The client also requests a page from the server when the accessed objects are spread across the page in a non-contiguous manner (low access locality). In these situations, the adaptive server architecture sends pages from the server to the client with the goal of reducing the number of client cache misses, because the object grouping mechanism at the server forms object groups that consist of contiguously placed objects.

If the client determines that not many of the objects are accessed (low spatial locality), then the client switches and requests a group of objects to try to reduce the server-to-client network overhead by not sending badly clustered pages from the server to the client. Depending upon the application data access pattern, the client dynamically changes the size of the requested object group. The client sends the object group size as a hint along with the data request to the server. The object group size is specified as a percentage of the page size instead of as an absolute number in order to be able to handle variably-sized objects and pages. If the size of the object group starts to increase, then the client switches over to requesting pages in order to try to lower the group forming overhead at the server, and the group disassembling overhead at the client.

When the server receives a client's hint for an object group, the server has the option to override the client hint and send pages if the server determines that its buffers are contended (i.e., if the modified object buffer (MOB) is not able to batch many updates to the pages and is, therefore, performing a high number of installation reads). If the server disk utilization is high, then the installation reads may interfere with normal read operations that are performed to read client requested data. Therefore, the server explicitly informs the clients when it is busy by piggybacking this hint along with other messages. If the page has not already been flushed from the client page cache, then the client uses the server provided busy hint to return updated pages rather than updated objects to the server. However, if the client dual buffer mechanism needs to discard a badly clustered page and retain only useful objects, the client ignores the server provided busy hint. The server and the clients send hints to each other, but have the freedom to override these hints depending upon their local run-time conditions.

If the server buffer is not busy, the client returns updated objects to the server because it wants to both reduce the network overhead and increase the server MOB absorption. Thus, returning updated objects from the client to the server helps to reduce the client-to-server data transfer overhead and installa-

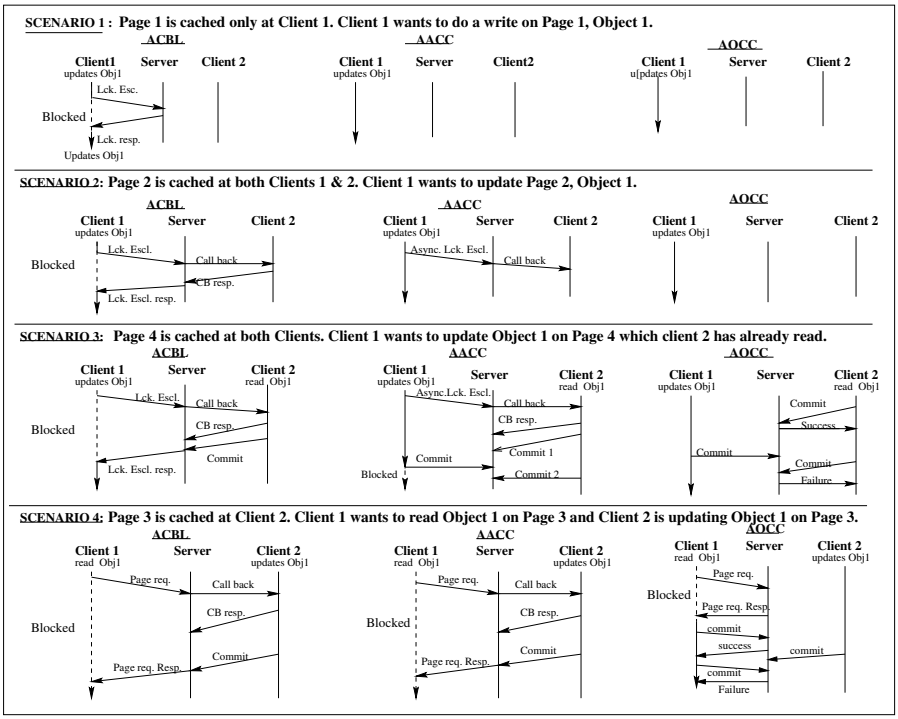

Figure 5. Cache Consistency Scenarios

tion write overhead. A detailed description of the implementation details of the data transfer algorithm can be found in [34].

2.2. CACHE CONSISTENCY

Adaptive Callback Locking (ACBL) [9] and Adaptive Optimistic Concurrency Control (AOCC) [1] are the two prominent client-server ODBMS cache consistency algorithms. AOCC is an optimistic algorithm which, for certain workloads, has better performance than ACBL, but is susceptible to high abort rates. ACBL is a pessimistic algorithm that has a lower abort rate than AOCC, but its performance trails AOCC's due to higher message processing and blocking overheads. This section presents the Asynchronous Avoidance-based Cache Consistency (AACC) algorithm that builds upon the strengths of AOCC and ACBL while avoiding their weaknesses. It is an adaptive algorithm because the clients and the server can dynamically adapt between sending synchronous, asynchronous or deferred lock messages.

This section first intuitatively motivates the operation of AACC by presenting four cache consistency scenarios, and it then provides the algorithm details.

2.2.1. *Intuitive Description of AACC*

This section uses the four scenarios of Figure 5 to discuss how AACC handles the four previously discussed overheads associated with client-server DBMS cache consistency algorithms.

- **Scenario 1:** In ACBL, the client sends an explicit lock escalation message when it wants to update object 1 on page 1. In AOCC the client defers informing the server about the write operation until commit time. Therefore, it does not encounter the message passing overhead that is present in ACBL. Normally, a delay in informing the server about updates until commit time increases the probability that another client has read the same object. This, in turn, increases the probability of an abort. However, since this page is cached only at a single client, the chances of conflicts are remote. AACC tries to capitalize on this insight by introducing the notion of private and shared pages. That is, when a server sends a page to the client, it also informs the client whether that page is cached elsewhere. If it is not, then the client piggybacks its write lock request instead of sending an explicit lock request. Therefore, as shown in Figure 5 scenario 1, if Client 1 wants to update object 1 on page 1, that is cached only at client 1 in private-read lock mode, then the client goes ahead with the update without sending an explicit write lock message. The client informs the server about this update by piggybacking the lock escalation message on a subsequent message to the server. The lock message is piggybacked instead of being deferred until commit time, to reduce the risk of a read/write conflict for the particular object. Thus, the notion of shared and private page read locks reduces the write lock message traffic in AACC.

- **Scenario 2:** This scenario shows the message blocking overhead that is present in ACBL since client 1 waits until its write lock request for object 1 on page 2 is granted by the server. The server, in turn, issues a lock callback message to client 2, and only grants the request to client 1 after hearing from client 2. Once again AOCC defers the write lock request until commit time, but increases the probability of a locking conflict and a subsequent transaction abort. AACC does not want to send synchronous locking messages as in ACBL, but at the same time does not want to defer the lock message to the server (informing about the update) until commit time and thus, increase the probability of an abort. Hence, in AACC, when Client 1 wants to update object 1 on page 2 which is cached at both clients 1 and 2 in shared-read lock mode, Client 1 sends an asynchronous lock escalation message to the server and continues without blocking. The server, in turn, forwards this message to client 2, which invalidates page 2, but informs the server about this invalidation by piggybacking the information on a subsequent message.

Thus, asynchronous locking messages have been introduced in AACC to reduce the write lock message blocking overhead. The sending of an asynchronous lock message does not delay the informing of the update to the server and it thus, reduces the abort probability. The piggybacking of the callback response from client 2 also reduces the lock messaging overhead.

- **Scenario 3:** In this scenario, ACBL requires client 1 to remain blocked until client 2 commits. In AOCC, client 1 does not block and, therefore, there is a greater probability of a transaction abort. AACC does not want to increase the probability of a transaction abort, but at the same time, it wants to reduce the time a transaction remains blocked due to a locking conflict. In AACC, client 1 does not block at the point there is a possibility of a read/write conflict, but it instead blocks at commit time. Therefore, instead of remaining blocked until client 2 transaction commits, client 1 is able to continue with its transaction execution from the point of the conflict until its transaction commit point, and is thus able to increase the overall system throughput. For example, in scenario 3, client 1 sends an asynchronous message to the server indicating its update. The server then forwards this message to client 2. Client 2 notices that there is a conflict and sends an explicit response to the server. The server then performs deadlock processing and notes that client 1 can only commit after client 2 has committed in order to prevent stale cache aborts. Therefore, client 1 can go ahead with its commit if client 2 commits at commit point 1 but client 1 blocks if client 2 commits at commit point 2.

- **Scenario 4:** In this scenario, ACBL blocks when there is a lock conflict, whereas AOCC increases its probability for a transaction abort by not blocking one of the conflicting transactions. A high number of aborts are not acceptable in many interactive transaction domains, and in environments where the cost of abort processing is high. Similar to ACBL, AACC also blocks when there is an explicit locking conflict. In scenario 4, Client 1 wants to read object 1 on page 3, which is present only at client 2. Moreover, client 2 holds an exclusive page level lock on page 3 and it is also updating object 1 on page 3. Upon receiving the page 3 read request from client 1, the server sends a callback message to client 2. Since client 2 is using the object, it sends a negative response to the server and thus client 1 blocks until client 2 does a commit. Just like the use of asynchronous lock escalation messages, instead of deferred lock escalation messages reduces the abort probability (as described above), the use of synchronous lock escalation messages reduces the probability of an abort even further. In order to ensure that it has a low abort rate as ACBL, AACC incorporates two deadlock avoidance optimizations [30].

Furthermore, since ACBL and AACC are avoidance-based algorithms, they do not allow for the presence of stale cache data in the client cache.

2.2.2. *AACC Detailed Description*

The following is a detailed description of AACC.

- **Data Request:** When a client wants to access an object whose page is not in its cache, it sends a page request to the server. When the server receives the request, it checks to see whether the page is cached at other clients.
  - If the page is not cached anywhere else, it returns the page to the client in *private-read* mode.
  - If the page is cached at another client in *private-read* mode, then the page is returned to the requesting client in *shared-read* mode. The server also informs, via a piggyback message, the client holding the page in *private-read* mode to change the page lock to *shared-read* mode. The inherent message delay may cause situations where one client has the page in *private-read* mode and other clients have the same page in *shared-read* mode.
  - If the page is cached elsewhere in *shared-read* mode, then the server returns the page to the client in *shared-read* mode.
  - If the page is cached at another client in *page- write* mode, then the server issues a callback message to the remote client indicating the object and the page that is being requested. Upon receiving the callback, the remote client checks to see whether it is using the particular object. If not, it changes the page lock to *shared-read* and returns the object identifiers of the objects on that page that have been updated. If it is using the requested object, it informs the server that it cannot satisfy the request.
  - Upon receiving a positive callback response, the server marks off the objects that are updated at the remote client and sends the page to the requesting client. If the server receives a negative callback response, it blocks the requesting client until the client that holds the write lock commits.
- **Updates on Private-Read Locked Pages:** When a client is performing an update on a *private-read* locked page, the client changes the page lock mode to *page-write*. The client then informs the server about this update by piggybacking the information on a subsequent message. Upon receiving the piggybacked message regarding the update and the lock escalation to the *private-read* locked page, the server does the following:

  - If the page is residing at other clients in *shared-read* lock mode, then the server sends an invalidation message to the affected clients. The invalidation message requests the clients to purge the object and/or page from their caches. The server also informs the client that has performed the update to change its page lock for the updated page from *page-write* to *shared-read* if other clients are using the page but not that object.
  - If the page is not present at other clients or has been successfully invalidated, then the server updates its lock tables to indicate that the client has a *page-write* lock for the page.

- **Updates on Shared-Read Locked Page:** When a client is performing an update on a *shared-read* locked page, it sends an asynchronous lock escalation message to the server and continues with its processing. When the server receives this message, it sends callback messages (indicating both the object and the page) to the other clients that have cached this page.

  - If the client that receives the callback message is not using the page, it simply invalidates it, and informs the server via a piggybacked message.
  - If the client is using the page but not the object, then it invalidates the object and informs the server via a piggybacked message.
  - If the client is using the object, then it sends a callback response indicating that there is a conflict.

- **Callback Processing:** When the server receives a callback response indicating that there is a conflict, it performs deadlock detection processing, and if there are no deadlocks, the client that has performed the initial update cannot commit before the client that is reading the object. Here, the server deadlock detection processing involves a check to see whether clients have updated objects that have been read by other clients. For example, if client 1 has updated an object read by client 2 and client 2 has updated an object read by client 1, then neither of these clients can commit their respective transactions and the server randomly aborts one of the conflicting transactions. If the server receives piggybacked callback responses from all the relevant clients indicating that they have invalidated the page, it sends an asynchronous message asking the client updating the initial page to upgrade its page lock from *shared-read* to *page-write* mode.

- **Commit Processing:** At commit time, the client sends the logs to the server. The client also piggybacks messages informing the server of updates to *private-read* locked pages. If a client has performed updates to a

private-read locked page, and this is being piggybacked on the commit message, then the server checks to make sure that no other client has that page in its cache in *shared-read* mode; and if another client does have that page, the server sends a callback message to that client. The server only allows the commit to proceed after receiving replies to all the pending callback messages from the necessary clients. At commit time, the server checks to see whether the particular client can go ahead with its commit or whether it should remain blocked since it has updated an object that has been read by another client. The server also moves logs to persistent storage, and then informs the client that it can go ahead with the commit. The client changes *page-write* page locks to *private-read* locks, and *write* object locks to *read* locks. The client relinquishes the objects that have pending callback messages on them from the server. The client then informs the server about its lock de-escalations; the server updates its page and object level lock tables accordingly. It also activates the other client transactions that are waiting for this client to commit.

2.3. RECOVERY

The current client-server recovery solutions cannot handle adaptive data transfers between the server and the clients because they are explicitly designed for page servers. This section first proposes an adaptive recovery mechanism that allows the server to send either pages or objects to the clients, and that allows the clients to return either pages (pages/log records) or objects (log records that are re-applied by the server to the data pages). It then proposes a recovery algorithm enhancement that allows for the update of an object at both the clients and the server within the same transaction. We assume that the reader is familiar with centralized ARIES [25] and client-server ARIES (ARIES-CSA) recovery algorithms [26].

2.3.1. *Hybrid Server Recovery Solution*

The following problems need to be addressed for the hybrid recovery solution:

- **Absence of pages at the client:** The log records generated at the client, the client dirty page table, and the state of a page with respect to the log (PageLSN) all require page-level information. Each generated log record contains a log sequence number (LSN). The LSNs are generated and handled in the same manner as in ARIES-CSA. Each page contains a PageLSN, which indicates whether the impact of a log record has been captured on the page. In hybrid servers, objects can exist at the clients without their corresponding pages. Hence, the page-level information

might not always be available at the clients. The hybrid server passes to the client the PageLSN and the page id information along with the requested data. After the client receives a group of objects, in addition to creating resident object table (ROT) entries, it also creates the resident page table (RPT) entry. For each received object, the client stores the PageLSN in the corresponding page entry in the RPT. This allows the client to generate LSNs for the log records corresponding to the page, and also RecLSN values for the page in the dirty page table. RecLSN refers to the log record of the earliest update on the page that is not present on disk. Thus, even though the clients might have only objects and not their corresponding pages in their caches, the clients still keep track of the necessary recovery information for the objects at page-level.

- **Presence of updated objects at the server:** The updated objects returned by the clients are stored in the server MOB and they are installed on their corresponding home pages in a lazy manner using a background thread [16]. The pages corresponding to the updated objects might not be residing in the server page buffer. Therefore, it is necessary to keep track of the state of the updated objects in the MOB with respect to the log records. That is, if a client fails and the server is doing restart processing, then the server needs to know the state of the objects in the MOB in order to correctly perform the redo operations. In page servers, the dirty page table at the server keeps track of the pages in the server buffer. Consequently, in addition to the dirty page table, the server maintains a *dirty object table* (DOT) to keep track of dirty objects. Each DOT entry contains the LSN of both the earliest and the latest log records that correspond to an update on the corresponding object.

- **Fine-Granularity Locking:** In client-server DBMSs, different objects belonging to a page can be simultaneously updated at different client sites. In centralized systems the LSNs are generated centrally, so the combination of PageLSN and the LSN of the log record is sufficient to assess whether the page contains the update represented by a log record. In client-server systems, since the clients generate the log record LSNs, two clients can generate the same LSN for log records pertaining to a page. Therefore, the PageLSN alone cannot correctly indicate whether the page contains the update represented by a particular log record. Two of the previous page server recovery solutions do not allow the simultaneous update of a page at multiple client sites [14, 26]. A more recent proposal [31] permits this and requires the server to write a replacement log record to the log disk before an updated page is written to data disk. For every client that has performed an update since the last time the page was written to disk, the *replacement* log record contains details (client ID and client specific PageLSN) about the client's update to the page.

Thus it overcomes the problems encountered due to the generation of the same PageLSN value at multiple clients. However, the proposed fine-granularity locking solution [31] does not handle the variable object size case where the object size can dynamically increase. If the size of two objects on the same page is simultaneously increased at two different clients, then the space left on a particular page may not be enough to hold both of the objects. The hybrid server recovery solution also uses the notion of *replacement* log records to allow simultaneous updates to a page at multiple client locations. In addition the following steps are necessary to allow for simultaneous updates to variable sized objects:

- At the server, if the space on a page is not enough to hold the updated object, the server moves the object to another page. The server updates the LOID-to-POID mapping data structures and writes a *hasBeenMoved* log record to the log disk to keep track of the object re-location.
- When sending the object updates to the server, the clients send the LOID information of the object that has been updated, which is used to determine the new location of the updated object. The LOID-to-POID mapping information at the client is changed at commit time.
- During the analysis phase of the recovery operation, the server constructs a list of the *hasBeenMoved* log records. This list is used during the redo phase of recovery.
- During the redo phase, if applying a log record to a page can lead to an overflow of the page, then the object is moved to another page. Before generating a *hasBeenMoved* log record, the server first checks to see whether there already exists a previously generated record in the *hasBeenMoved* list. If an entry exists, then the server uses the information about the new page, that is present in the entry, to redo the operation. If a new *hasBeenMoved* entry is being generated, then the LOID-to-POID mapping data structure is updated accordingly.

- **Returning pages or logs to the server:** In the hybrid server architecture, clients return either pages and log records or only log records (redo-at-server recovery). In the latter, the log records have to be installed on their corresponding home pages (ARIES-CSA avoids this). Therefore, each log record is classified at the client as a redo-at-server (RDS) log record or a non-redo-at-server (NRDS) log record. At the server, the RDS log record is stored both in the server log buffer and also in the MOB, whereas the NRDS log record is only stored in the server log buffer. The RDS is stored in the MOB to reduce the installation read

overhead. If the client decides to return a page to the server, then it generates a NRDS log record, otherwise it generates a RDS log record. When the client dynamically decides to switch from the redo-at-server mode to ARIES-CSA mode, the following processing is performed at the client and the server:

- **Changing from Redo-at-Server mode to ARIES-CSA mode at the Client:**
  (i) **Processing at the Client:** The client ensures that for the subsequent updates, it only generates NRDS log records.
  (ii) **Processing at the Server:** An RDS log record corresponding to the updated page might already be present in the server MOB. Therefore, following the state change from redo-at-server mode to ARIES-CSA mode at the client, the server can also receive the updated page from either the same client or a different client. Upon receiving the updated page, the server installs the RDS log present in the MOB on the page only if the particular object has not been write-locked by a different client (that is, its corresponding page has also not been write-locked by a different client). If the server receives the corresponding updated page and the page has been write-locked either by the same client or a different client, then the server discards the RDS present in the MOB because the effect of the RDS is already present on the page.
- **Changing from ARIES-CSA mode to Redo-at-Server mode at the Client:**
  (i) **Processing at the Client:** The client has to ensure that the pages corresponding to the NRDS logs are sent to the server either when the page is flushed from the client data buffer, or if the RDS log is getting sent to the server, or at commit time.
  (ii) **Processing at the Server:** If the page corresponding to the RDS log is already present in the MOB, then the server eagerly installs the RDS to the page.

2.3.2. *Updates Performed at both Clients and Server*

Existing client-server recovery solutions cannot handle the case when updates are performed at both clients and the server. This section first discusses the new recovery issues that are encountered when updates are performed at both the clients and the server and then proposes solutions to these issues within the context of an ARIES-style client-server recovery algorithm.

- **Correct Order of Log Execution:** Since the same data item can be updated at both the server and the clients within the same transaction, it is

necessary to ensure the correct order of log application during rollbacks and failure recovery. For example, if a data item was updated first at the client, and subsequently at the server, then during rollback processing it is necessary to ensure that the effects of the server updates are undone before the effects of the client updates are undone. In order to solve this issue, it is necessary to have some co-ordination between the server and the client LSN generation process.

During the application program execution, if the server passes control to the client, or the client passes control to the server, they also pass the LSN values of the latest log records generated for the updated pages to each other. This, in turn, helps the receiver of the LSN value to ensure that the subsequent log records have an LSN value that is greater than the current LSN value. In order to ensure that the address of the previous log record (used during rollback processing) is properly set, when a client transfers application execution control to the server, it also returns the log records that it has generated along with the updated data to the server. This allows the server to properly set the address of the previous log records.

- **Client Participation in Rollback Processing:** It is desirable for the clients to be also responsible for rollback processing because this offloads work from the server. The notion of *transfer-control* log record helps to facilitate client participation in rollback processing in a hybrid function-shipping/data-shipping environment. When a client passes control of application execution to the server, it passes along the log records it has generated along with the updated data. The server then generates a special log record known as the *control-transfer* log record. The control-transfer log record is positioned in the PrevLSN log chain for the particular transaction. The control-transfer log record's type field is set to *control-transfer*. The PrevLSN field of this log record is set to the LSN of the last log record generated for the transaction at the client. If the server transfers control to a client, the server generates another control-transfer log record and the control-transfer record becomes the previous log record to the logs that will be generated at the client. When the server encounters a control-transfer log record during rollback processing, it passes on control to the client along with the appropriate log records (up to the previous control-transfer log record, or the saveLSN log record) and the updated data. Similarly, when the client encounters a control-transfer log record during rollback processing, it passes on control and the updated data to the server. Thus, the *control-transfer* log record transfers the control of rollback processing between the client and the server.

The performance of the adaptive (redo-at-server and ARIES) recovery solution proposed in this section is compared with both the ARIES and the redo-at-server recovery solutions in Section 4 of this paper.

3. Experimental Setup

The baseline setup of this performance study is similar to the previous client-server performance studies [11, 9, 37, 23, 1, 35], which were useful in validating our results. As in the previous performance studies, the input work comes to the clients as a stream of object and page identifiers from a workload generator; it comes to the server from the clients via the network. The number of clients was chosen to ensure that the server and client resources and the network resources do not become a bottleneck, which would prevent us from gaining insights into the different algorithms and architectures. Disks are modeled at the server and not at the clients. The server is responsible for managing data and log disks. A buffer manager, a lock manager, and a recovery manager have been modeled at both the clients and the server. The data buffers use the second chance (LRU-like) buffer replacement algorithm, and the log buffers and modified object buffers use the FIFO buffer replacement policy. The server buffer space is partitioned equally between the page buffer and the modified object buffer (MOB). We configured the client dual buffer in a manner similar to the initial dual buffer study [20] where the buffer is configured as best as possible, given the application's profile and the total size of the client's buffer. The client and the server CPUs have a high priority and a low priority input queue for managing system and user requests respectively [9]. Each disk has a single FIFO input queue. We use a fast disk I/O rate for installation I/O (because the I/O for the data in the MOB is intelligently scheduled) and a slow disk I/O rate for normal user read operations. The LAN network model consists of FIFO server (separate queues for the server-to-clients and clients-to-server interaction) with the specified bandwidth. In order to prevent network saturation, we ran our experiments assuming a 100Mbps switched network. The network cost consists of fixed and variable transmission costs along with the wire propagation cost. Every message has a separate fixed sending and receiving cost associated with it; the size of the message determines the variable cost component of the message.

Figure 6 lists the costs of the different operations that are considered in this performance study which are similar to the ones used in previous performance studies [9, 1, 37, 38]. The group forming cost consists of the cost of creating the object group header, the cost of copying the objects from the page, and the cost of determining the object's lock group. The registration cost also includes the cost of loading objects into the client object buffer. We ensured that the type and size of the object identifiers and the object representation

Parameter	Description	Value
Client CPU speed	Instr. rate of client CPU	50 MIPS
Server CPU speed	Instr. rate of server CPU	100 MIPS
ClientBuffSize	Client buffer size	1-12% DB size
ClientLogBuffSize	Client log buffer size	11-2.5% DB size
ClientBuffSize	Server buffer size	1-50% DB size
ServerDisks	Disks at server	4 disks
FetchDiskTime	General disk access time	3322 microsecs/KB
InstDiskAccessTime	MOB disk I/O time	1288 microsecs/KB
FixNetworkCost	Fixed number of instr./msg	2000-10000 cycles
VariableNetworkCost	Instr./msg byte	2-7 cycles/byte
Network Bandwidth	Network bandwidth	10-155 Mbps
DiskSetupCost	CPU cost for disk I/O	5000 cycles
CacheLookup/Locking	Lookup time for objects/page	300 cycles
Register/Unregister	Instr to register/unregister a copy	300 cycles
Hardware Swizzling	Pointer swizzling cost/page	50000 cycles
DeadlockDetection	Deadlock detection cost	300 cycles
CopyMergeInstr	Instr. to merge 2 copies of a page	800 cycles/object
Software Swizzling	Swizzling cost/pointer	80 cycles
Database Size	Sixe of the database	2400 pages
PageSize	Size of a page	4K - 8K
ObjectSize	Size of an object	100 bytes - 1 KB
GroupFormCost	Group forming cost/object	100 cycles
NumberClients	Client workstations	12
Indirection Cost	Pointer indirection cost/access	15 cycles
Delay Probability	Prob. for delaying message	50%
Delay Time	Time a message is delayed	10 msec
Software Unswizzling	Unswizzling cost/pointer	30 cycles
LogProcCost	Logging data structures update	50 cycles/log

Figure 6. System Parameters

mechanism is the same across all of the architectures. The hardware pointer swizzling cost is encountered by the hardare pointer swizzling algorithm [37], and the software pointer swizzling and unswizzling cost is encountered by the software pointer swizzling algorithms [23, 6]. The LogProcCost captures the cost associated with updating the logging data structures at the client during the generation of each log record. Finally, the deadlock detection cost is in-

curred at the server, when the server performs deadlock detection processing after discovering a locking conflict.

3.1. WORKLOAD MODEL

The multi-user OO7 benchmark has been developed to study the performance of object DBMSs [7]. However, this benchmark is inadequate for client-server concurrency control and data transfer studies. It is under-specified for client-server concurrency control/cache consistency studies, because it does not contain data sharing patterns and transaction sizes. It is also under-specified for a data transfer study, because it does not contain the notion of data clustering. Therefore, we borrowed data sharing notions from the previous concurrency control studies [9, 1], and the data clustering notions from the initial data transfer study (ACOB benchmark) [11]. We obtained the transaction size and length characteristics from the market surveys performed by the commercial ODBMS vendors [27]. The key findings of their survey is that the majority of the application domains using ODBMSs use short (in terms of time) and small (number of objects accessed) transactions with multiple readers and few updaters operating on each object. Moreover, most of these applications use small objects. We have ensured that our base workloads satisfy these transaction characteristics.

In our workload, each client has its own hot region (hotness indicates affinity) and there is a shared common region between all the clients. Each region is composed of a number of base assembly objects. Each base assembly object is connected to 10 complex objects. Each complex object consists of 4 atomic objects. A transaction consists of a series of traversal operations. Each traversal operation consists of accessing a base assembly and all of the complex objects (along with their atomic objects) connected to that base assembly object. The spatial, temporal and access locality probabilities together determine the data access pattern of the application with respect to how the data is clustered on a disk page. These locality values are varied between 0 and 100 percent in the experiments.

In this study we examine *Private*, *Sh-HotCold* (Shared Hot-Cold) and *Hi-Con* (High Contention) data sharing patterns [9, 1]. There is no data contention in the *Private* workload, but one encounters read-write and write-write conflicts in the *Sh-HotCold* workload. In the *Private* workload 80 percent of the traversal operations in a transaction are performed on the client's hot region and 20 percent of the traversal operations are performed on the shared region. Moreover, the clients only update the data in their hot regions. In the *Sh-HotCold* workload, 80 percent of the traversal operations in a transaction are performed on the client's hot region, 10 percent of the traversal operations are performed on the shared region, and 10 percent of the traversal operations are performed on the rest of the database (including other clients' hot re-

gions). The clients can update objects in all of the regions. In the *Hi-Con* workload, clients access the shared region 80 percent of the time, and the hot regions of the other clients 20 percent of the time. The clients perform write operations in both the shared region and the hot regions of the other clients. Upon accessing an object, the object write probability determines whether the object will be updated. There is a CPU instruction cost associated with the read and write operations (read and write access think costs). The transaction think time is the delay between the start of two consecutive transactions at the clients. Finally, the transaction abort variance value is varied between 0 and 50 percent. This value determines the probability of each object in the transaction being accessed again during the re-execution of the transaction. Figure 7 describes the workload parameters used in this study.

Parameter	Setting
Transaction size	180 - 220 objects
Spatial/temporal/access localities	10-90 %
Per client region	50 pages
Shared region	50 pages
Object write probability	0 - 30 %
Read access think cost	50 cycles/byte
Write access think cost	100 cycles/byte
Think time between transactions	0
Abort variance	0 - 50 %

Figure 7. Workload Parameters

4. Performance Study

We compare the proposed adaptive hybrid server architecture with other leading data shipping ODBMS architectures. The comparison is based on an integrated performance study that looks at the system components, data transfer, cache consistency/concurrency control, recovery, buffer management and pointer swizzling in an integrated manner. This is the first *integrated* study in this field and the results demonstrate the interplay among different system components under different algorithms. Data granularity (*page* versus *object*) has a major impact on the performance of the different data transfer, cache consistency, recovery, buffer management and pointer swizzling algorithms. The current ODBMS client-server architectures are either page-based or object-based. The focus of this performance study is to determine if

the hybrid client-server architecture, which can dynamically adapt between page and object level granularities, is more robust than either exclusively page-based and object-based architectures.

We also present a performance study of cache consistency algorithms that goes beyond data granularity concerns. The study compares the performance of AACC, which was described in Section 2, with other leading client-server ODBMS cache consistency algorithms for different workloads and system configurations. The focus of the study is to evaluate whether AACC has a better combination of performance and abort rate than ACBL and AOCC. In this paper we have tried to utilize CPU, network and disks speeds that reflect the current trend in computing hardware. However, our goal is to understand the general performance trends of the different algorithms and the relative difference in performance between the different algorithms as the CPU, network and disk speeds are varied [34].

Each of the experiments in this section describes the system and workload parameters, followed by the primary and secondary graphs. 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 three times using three different random number seeds and each run consisted of twenty thousand transactions. The parameters described in Section 3 are used for all the experiments unless explicitly noted.

4.1. Cache Consistency Study

We present the cache consistency study first since the client-server architectures that are compared in the integrated performance study all use the same AACC cache consistency algorithm. In this study, AACC is compared with the AOCC and ACBL. The data transfer, recovery, buffer management and pointer swizzling components have been fixed during this study. Lock granularity related issues are not examined at this stage (using only page level locking). The server transfers pages to the clients and the clients return updated objects back to the server. The clients use a page buffer and the server contains both a staging page buffer and a modified object buffer. The redo-at-server recovery and software pointer swizzling are utilized by all of the systems under comparison.

4.1.1. *Private Workload Experiments*

In the private workload, the clients only perform updates on their private hot regions and do not perform any updates on the shared or other client regions. Private workload is indicative of computer-aided design (CAD) environments where the users perform updates on their private data, but also do reads on shared data. Due to the absence of data contention, no aborts occur in this

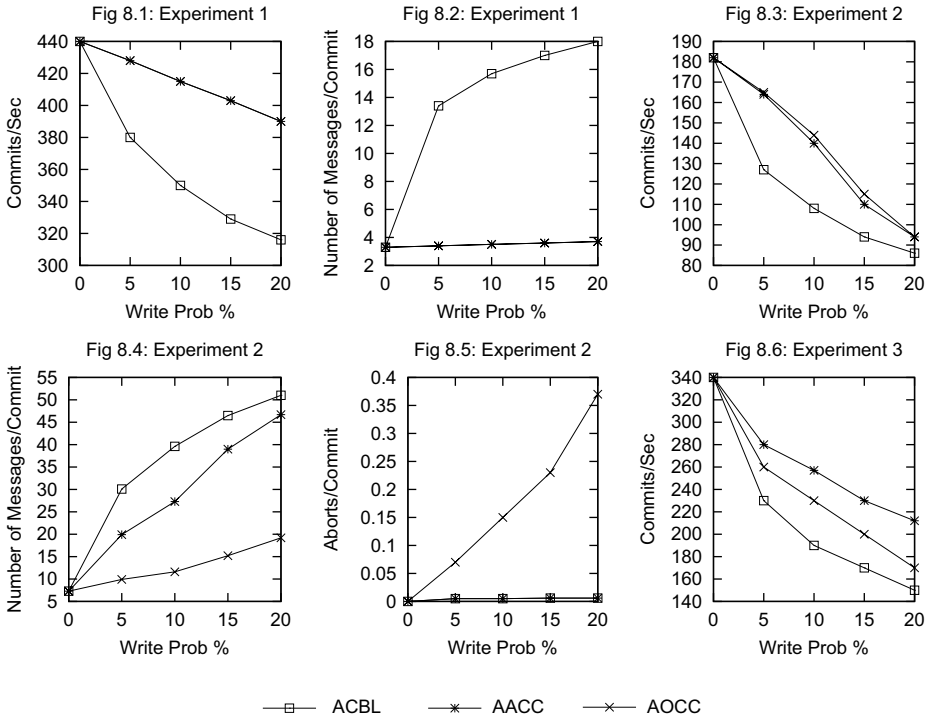

Figure 8. Cache Consistency Experiments

workload. The write probability is varied on the x-axis for private workload experiments and overall system throughput in commits-per-second is measured on the y-axis. The client and server data buffers are large. The access and temporal localities have been set to 50%. 100% percent of the work is performed at the client and the network speed is set at 100 Mbps. The CPU speeds of the server and the client are 100 MIPS and 50 MIPS, respectively.

Experiment 1: Private Workload. The objective of this experiment is to assess the performance of the algorithms without any data contention. In this experiment the spatial locality has been set to 20%. As shown in Figure 8.1, AOCC and AACC perform identically and they both outperform ACBL for all write probabilities. In ACBL, the clients send explicit lock escalation messages to the server to obtain exclusive page level locks for every page that is updated and they block until the server responds. In AOCC, all the write notifications are deferred until commit time, while in AACC, the *shared-private* optimization ensures that all update notifications are sent to the server in a piggy-backed manner. As evident in Figure 8.2, ACBL

sends more messages than AOCC and AACC, because in every transaction ACBL sends a lock escalation message for every page (objects on the page) it updates. Therefore, the higher message transmission and message blocking overhead increases ACBL's write lock acquisition overhead, which, in turn, makes its performance inferior to that of AOCC and AACC.

4.1.2. *Shared-HotCold Workload*

Sh-HotCold workload data contention level is indicative of the data contention level present in most client caching applications. Therefore, its results are very important. Due to the presence of data contention, stale cache aborts are possible in AOCC, and deadlock aborts are possible in AACC and ACBL. In these experiments, the client and server data buffers are large. The spatial and temporal localities have been set to 50%. The network speed is 100 Mbps.

Experiment 2: Slower CPU Speeds The objective of this experiment is to assess the performance of the different algorithms for Sh-HotCold workload when using slower CPU speeds. The server and client CPU speeds are 50 MIPS and 25 MIPS respectively. These CPU speeds, are classified as being *slower* than the faster CPU speeds that are used in the subsequent experiments. The slower CPU speeds were used by previous cache consistency performance studies [1, 9], and they are used here to help analyze the change in the performance when there is a change in CPU speeds. The abort variance value has been set to 50%. As evident from Figure 8.3, AOCC and AACC outperform ACBL for all non-zero write probabilities. At 0% write probability, since no updates are performed, the performance of all three algorithms is identical. Figure 8.4 shows that AOCC sends fewer messages than AACC, and AACC sends fewer messages than ACBL. Thus, the lower message overhead helps AOCC. However, Figure 8.5 shows that AOCC has a higher abort rate than AACC and ACBL. The abort processing overhead associated with this high abort rate degrades AOCC's performance and allows AACC to almost match AACC performance. ACBL is not able to match AOCC performance because ACBL uses synchronous lock escalation messages and, therefore, incurs higher message blocking overhead. Even though AOCC has a higher abort rate, as shown in Figure 8.3, its performance does not degrade drastically, because AOCC uses a fast abort processing mechanism that keeps the undo log records in the client cache, and, thus eliminates the need to fetch them from the server. However, as the write probability approaches 18%, the number of aborts in AOCC start to become a factor. Here AACC performance is identical to AOCC performance.

AACC outperforms ACBL for the entire range of write probabilities. AACC uses fewer messages than ACBL, because, in AACC, the write lock messages

Costs in microseconds/commit	ACBL		AACC		AOCC	
	Exp. 3	Exp. 4	Exp. 3	Exp. 4	Exp. 3	Exp. 4
Data Request	1659	1233	1504	1000	1405	1392
Write Lock Request	2000	1084	390	200	0	0
Client Application Processing	4047	2024	4047	2024	4400	2280
Commit	290	173	517	370	309	187

Figure 9. Experiments 2 and 3 Cost Breakdown for 10% Write Probability

for private pages are piggybacked on other messages. Furthermore, in AACC, clients also piggyback callback responses if there are no lock conflicts. Since AACC uses asynchronous lock escalation messages on shared pages, AACC has lower message blocking costs than ACBL. Finally, the deadlock avoidance techniques used by AACC allow it to have an abort rate (as seen in Figure 8.5) which is as low as ACBL's abort rate.

Experiment 3: Increase in CPU Speeds The objective of this experiment is to assess the performance of the different algorithms when there is an improvement in the CPU speeds. As shown in Figure 8.6, AACC outperforms AOCC and ACBL. This experiment's setup is the same as Experiment 3, except the server CPU speed is 100 MIPS and the client CPU speed is 50 MIPS. This result is important because a previous study comparing AOCC and ACBL [1] indicates that AOCC always outperforms avoidance-based cache consistency algorithms. Faster CPUs help AACC and ACBL to reduce the CPU overhead associated with sending messages and they also help to reduce the execution time of a transaction. This, in turn, reduces the transaction blocking time in AACC and ACBL (due to locking conflicts) and increases overall throughput. Since AOCC has a higher abort rate than ACBL and AACC, with 50 percent abort variance, the restarted transactions in AOCC can access new pages which may not be present in either the client or the server buffer. This results in AOCC performing more I/Os per transaction than ACBL and AACC. Moreover, faster CPUs increase disk utilization quicker than slower CPUs, and this leads to higher relative disk I/O costs for AOCC than ACBL and AACC.

Figure 9 gives the cost breakdown for all of the three algorithms for both slower (experiment 2) and faster CPUs (experiment 3), respectively, for 10% write probability. The four costs presented in this figure are, 1) data request: the cost to obtain objects from the server (includes disk and network cost, and blocking related cost) and to put them in the client cache, 2) write-lock request: the cost for obtaining write locks from the server (includes blocking related cost), 3) client application processing: the cost for performing

Costs in microseconds/commit	ACBL	AACC	AOCC
Data Request	1233	1000	1018
Write Lock Request	1084	200	0
Client Application Processing	2024	2024	2161
Commit	173	370	187

Figure 10. Zero Abort Variance Cost Breakdown for 10% Write Probability

application related processing at the client (includes the aborted transaction processing cost) and 4) commit: the transaction commit processing cost. As seen in these figures, when going from slower CPUs to faster CPUs, the cost to get objects and locks from the server to the client decreases in AACC and ACBL because faster CPUs reduce the blocking overhead due to lock conflicts in these two algorithms. Faster CPUs also reduce the message transmission times in these two algorithms. As evident from a decrease in the client processing cost, faster CPU reduces the abort processing costs in AOCC. However, the decrease in the object request and lock request costs in AACC are larger in comparison to the decrease in the object request and client processing costs in AOCC. Thus, AACC is able to outperform AOCC. However, the synchronous nature of ACBL does not sufficiently reduce the locking costs (blocking overhead is still large) to allow ACBL to outperform AOCC.

Experiment 4: Zero Abort Variance The purpose of this experiment is to assess the impact of abort variance. It uses the same setup as Experiment 3, except that the abort variance is set at 0%. As evident from Figure 10, AOCC outperforms ACBL and AACC, because with no abort variance, AOCC is able to find most of the data in the client cache during abort processing, reducing the number of data request messages to the server. This, in turn, ensures that it does not perform more disk I/Os than the other algorithms. As shown in Figure 10, the locking overhead present in AACC and ACBL allow AOCC to outperform them. However, as the write probability increases to 18%, AACC outperforms AOCC (there is a cross-over in the graph) because the number of aborts in AOCC is high, and the abort processing cost of AOCC starts to dominate, degrading its performance.

Experiment 5: 50 Percent Server Work Allocation The purpose of the server allocation workload is to assess the impact of abort processing overhead for AOCC when work is partly performed at the server. The server and the clients have large buffers and they use fast CPUs. This experiment uses 100 Mbps network and 50% abort variance. Sh-HotCold workload is used both at the server and at the clients. 50% of the work is performed at the server and 50%

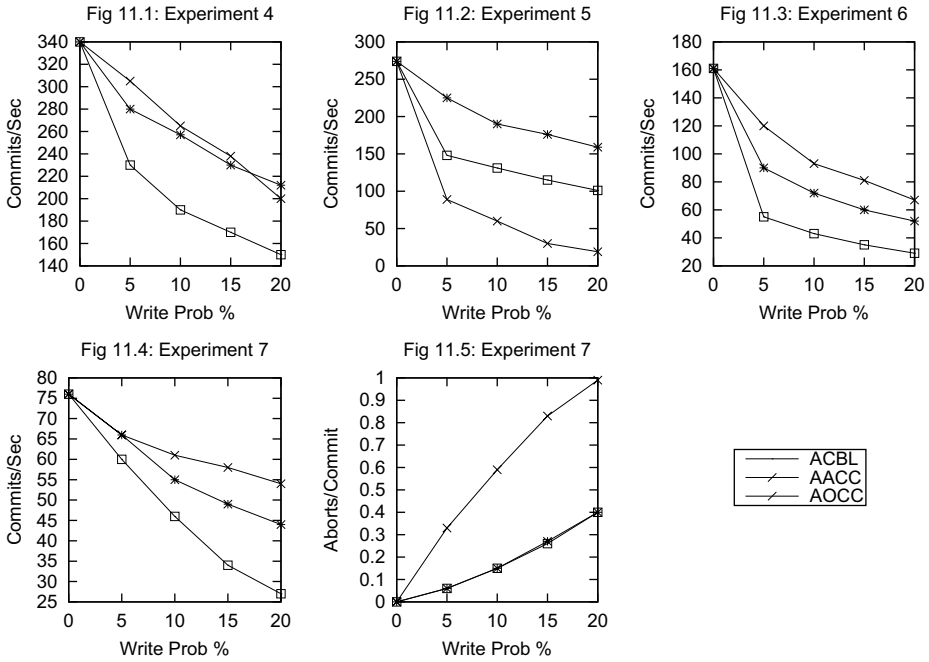

Figure 11. Cache Consistency Experiments

of the work is performed at the clients. That is, 50% of the traversal operations are carried out at the client and 50% are carried out at the server. As can be seen in Figure 11.2, AACC and ACBL outperform AOCC as the write probability increases because AOCC incurs more transaction aborts. In the environment where the work is strictly performed at the client, if a transaction aborts then the re-execution of the failed transaction has very little impact on the performance of the other transactions. However, in this experiment, since application processing is also performed at the server, when a transaction aborts it also impacts the performance of other client transactions. The key reason is that server resources such as the CPU, buffers, data disks and log disk are shared by all of the clients. Therefore, during the transaction abort processing, the necessary data pages and logs might not be present in the server buffer and have to be retrieved from disk. This slows down abort processing of the failed client transaction, and also degrades the throughput of the entire system. Since AACC and ACBL are avoidance-based, they incur fewer aborts than AOCC, and hence are able to outperform AOCC. Previously, it was shown within the context of centralized DBMSs that in medium to highly contended servers, the performance of optimistic concurrency control algorithms suffers due to high abort processing costs [2]. The results of this experiment concur with the assessment of the previous study.

Costs in microseconds/commit	ACBL	AACC	AOCC
Data Request	11780	11770	11398
Write Lock Request	7863	805	0
Client Application Processing	2306	2295	3055
Commit	160	1350	187

Figure 12. HiCon Costs with 10% Write Probability

Experiment 6: Network Delay This experiment assesses the impact of network delays (such as those experienced in wide area network environments) on the performance of the three algorithms. This experiment uses faster CPUs, 100 Mbps network, large client and server buffers and an abort variance of 50%. Figure 11.3 shows that the performance of the three algorithms degrades, for the Sh-HotCold workload, as the network delay is introduced in comparison to a network with no delays (Figure 8.6). However, the performance of ACBL degrades much more (percentage-wise) than the performance of AOCC and AACC, because ACBL uses synchronous lock escalation messages whereas AACC and AOCC use asynchronous and deferred lock escalation messages, respectively. In ACBL, the clients remain blocked until their lock escalation and subsequent callback messages (if necessary) are processed. AOCC outperforms AACC because the latter uses more messages than AOCC. Even though AACC uses asynchronous lock escalation messages, a client's transaction synchronizes with other clients at commit time to ensure that there are no conflicts. Thus, unexpected message delays increase the commit processing time in AACC, and its performance trails AOCC's performance.

4.1.3. *HiCon Workload*

In HiCon workload, the clients access the shared data region 80% of the time and the data region of other clients 20% of the time. This is a skewed data access pattern that is not usually present in data-shipping applications [9]. It is being examined here to test the behavior of the different cache consistency algorithms under extreme data contention situations.

Experiment 7: HiCon Data Contention This experiment uses 100 Mbps network, 50% abort variance, and smaller server buffer and faster CPU speeds. In the previous experiments, all of these factors helped AACC and ACBL more than AOCC. Therefore, the objective of this setup is to see the impact of the HiCon data sharing pattern on the three algorithms. As shown in Figure 11.4, even with faster CPUs and 50% abort variance, AOCC outperforms AACC which outperforms ACBL. However, as shown in Figure 11.5, AOCC has a

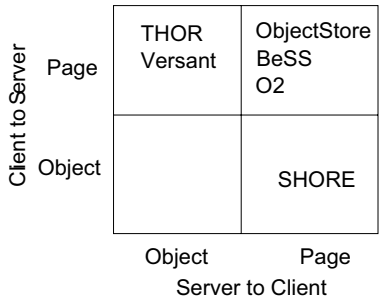

Figure 13. ODBMS Classification According to Data Transfer

higher abort rate (aborts/commits) than ACBL and AACC. One would expect algorithms with a high abort rate to perform worse than algorithms with lower abort rates. As described in Section 2, the read/write conflict blocking rates of AACC and ACBL are higher than the abort rate of AOCC. That is, for every blocking transaction in AACC and ACBL, the equivalent situation can lead to either an abort or a commit in AOCC. As shown in Figure 12, the time a transaction remains blocked in ACBL and AACC (higher object request and write lock request costs) is more than the abort processing cost in AOCC. Thus, AOCC outperforms ACBL and AACC even though it encounters a higher abort rate.

4.2. Integrated Performance Study

The hybrid server (HybSrv) architecture proposed in this paper is compared with a software-based page server (PageSoft), a hardware-based page server (PageHard), and an object server (ObjSrv). The software-based page server falls under the Page-Object server classification of Figure 13 and is similar to SHORE [6]. The hardware-based page server falls under the Page-Page server classification, and is similar to ObjectStore [22] and BeSS [4] in that it sends pages in both directions during client-server interaction. The object server architecture falls under the Object-Object server classification and is similar to Versant [33] and Thor [23]. The existing hardware page server systems [4, 22] employ page level data transfer, concurrency control and buffer management. As a representative of these systems, PageHard also adheres to the page level restrictions and this is the key distinguishing feature between PageHard and the other architectures. The data transfer mechanism from the server to the client is the key distinguishing factor between PageSoft and ObjSrv. The ability to send pages or objects from the server to the client, and to return pages or objects from the client are the key distinguishing factors between HybSrv and the other architectures (PageSoft, ObjSrv and PageHard). The overall performance of a system is also affected by other issues such as query processing, query optimization, indexing and others, which are not

	Server->Client	Recovery Client->Server	Cache Consistency	Pointer-Swizzling	Server Buffer	Client Buffer
PageHard	Page	ARIES Logs and Page	Page Level AACC	Hardware	Page/ Modified Page	Page
PageSoft	Page	Redo At Server Logs	Adaptive AACC	Software	Page/ Modified Object	Dual
ObjSrv	Objects	Redo At Server Logs	Adaptive AACC	Software	Page/ Modified Object	Object
HybSrv	Page/ Objects	ARIES Or Redo at Server Page/Logs	Adaptive AACC	Software	Page/ Modified Dual	Dual

Figure 14. Systems Under Comparison

considered here. The latest advances in cache consistency, buffer management, and recovery strategies are incorporated into all of the systems under comparison in this study (see Figure 14), ensuring that they all benefit from the same advantages. Therefore, the systems under comparison are similar, but not identical to their commercial/research counterparts. In the remainder of this section, the details of these architectures are described.

4.2.1. *Hardware-Based Page Server (PageHard)*

In the hardware-based page server architecture, the client requests a page from the server by sending it the page identifier. The server responds to the request by returning the appropriate disk page to the client. This architecture uses the hardware-based pointer swizzling mechanism as in ObjectStore [22] and BeSS [4]. Since the hardware-based pointer swizzling mechanism relies on the operating system virtual memory faulting (page level) mechanism, it is efficient for the clients to only deal with pages. This, in turn, makes it necessary for the server to return pages to the clients, and for the clients to manage a page level data buffer. The server manages a page level staging read buffer and a modified page buffer (MPB), which is a page level version of MOB. Even though ObjectStore and BeSS use the ACBL cache consistency mechanism, PageHard uses AACC because of its better performance. A hardware-based pointer swizzling system relies on the operating system provided page protection mechanism for read and write lock support. This makes it difficult to provide object level locking support, and, thus, PageHard uses the page-level version of AACC. PageHard (like BeSS) uses the ARIES recovery algorithm, because it has been shown that the log disk becomes a bottleneck during whole-page logging [38], which is used in ObjectStore. Similar to ObjectStore and BeSS, at commit time the clients return updated pages back to the server. In this architecture the clients maintain a page level undo log buffer. Log records are generated by the client by performing a *page difference* operation [38], and they are stored by the server in a log buffer from where they are flushed to the log disk when the buffer is full or when the transaction has reached the commit point. The data buffers use the LRU like (second chance) buffer replacement policy and the log buffers use the

FIFO buffer replacement policy. PageHard uses 8 byte pointers to represent OIDs because using 4 byte pointers limits the amount of addressable virtual memory to 4 gigabytes, and this, in turn, restricts the size of the database that can be accessed by a client.

4.2.2. *Software-based Page Server (PageSoft)*
In the software-based page server architecture the client request and server response are identical to PageHard. PageSoft uses software pointer swizzling mechanism and it uses LOIDs that are 8 bytes long. Since the software pointer swizzling mechanism does not use the operating system page faulting mechanism to load data, the clients have the flexibility to manipulate both pages and objects. Thus, this architecture provides both page level and object level concurrency control support. PageSoft also uses AACC because of its superior performance. In SHORE, the clients receive pages from the server but then the relevant objects are copied from the page buffer into an object buffer before an application can access them. In order to reduce this copying overhead, PageSoft uses a hybrid dual buffer at the clients [20]. The dual buffer allows clients to store well clustered pages as well as isolated objects from badly clustered pages. Similar to SHORE, PageSoft utilizes the redo-at-server logging mechanism. However, unlike SHORE, which does not contain a MOB, PageSoft contains both a staging read buffer and a MOB. The clients maintain an object level log buffer and generate log records using the *difference* operation. The server also maintains a staging log buffer from where log records are flushed to the log disk either when the log buffer gets full or at commit points. In this architecture, the data buffers use the LRU like (second chance) buffer replacement policy and the log buffers use the FIFO buffer replacement policy.

4.2.3. *Grouped Object Server (ObjSrv)*
In this architecture, the client requests objects by sending object identifiers to the server. The server returns a group of objects to satisfy the client's object request. This data transfer mechanism is similar to Versant ODBMS [33] and the THOR storage manager [23]. Similar to THOR [23], in this architecture the clients dynamically determine the size of the object group and the server forms object groups using objects that reside on contiguous disk locations. The server contains a staging read buffer and a MOB buffer. The server also contains a log staging buffer for writing the log records to the log disk. Since the clients receive a group of objects, they must maintain an object level buffer. Furthermore, they cannot return updated pages back to the server because the clients only deal with objects. Since clients in object server do not deal with pages, the clients cannot efficiently use the hardware pointer swizzling mechanism [38] and, therefore, they use the software pointer swizzling mechanism. This architecture utilizes the redo-at-server recovery mechanism.

Similar to THOR, the server stores the redo logs in its MOB. Unlike THOR, which employs an optimistic cache consistency mechanism (AOCC), this architecture uses the object server version of AACC which we have shown to be more robust. In this architecture, the data buffers use the LRU like (second chance) buffer replacement policy and the log buffers use the FIFO buffer replacement policy.

4.2.4. *Hybrid server (HybSrv)*
In the new architecture that we propose, the clients can request either pages or objects from the server, and the server can return either pages or groups of objects to the clients. The clients can also return both updated pages and objects to the server. When a client returns updated objects, it uses a redo-at-server recovery mechanism, and when it returns updated pages, it uses ARIES-CSA type recovery mechanism. HybSrv uses software pointer swizzling, since it has to efficiently handle both pages and objects. Since the clients can receive either pages or objects, they maintain a dual buffer. The server maintains a staging read buffer, as well as a modified dual (page/object) buffer. The clients maintain an object level log buffer and generate log records using the *difference* operation. The server also maintains a staging log buffer from where log records are flushed to the log disk either when the log buffer gets full or at commit points. In this architecture, the data buffers use the LRU like (second chance) buffer replacement policy and the log buffers use the FIFO buffer replacement policy.

4.2.5. *Large Client and Large Server Buffers*
The purpose of this experiment is to assess the impact of pointer swizzling mechanism and client to server data transfer mechanism on the overall performance of the different architectures. In this setup both the client and the server have large buffers. In steady state, the client cache is loaded, and, therefore, there should be few client cache misses. Due to these conditions, buffer management and server-to-client data transfer are not the performance differentiating factors between the different architectures. Instead, pointer access and client-to-server data transfer are the key performance determining issues. The client buffer is large enough to hold the client working set and the server buffer is 75% of the database size. There are three pointers from each object to other objects. Both the spatial and temporal locality have been set at 50% and the network speed is set to 100 Mbps.

Experiment 8: Large-Large Private This experiment uses the private data sharing pattern. Therefore, concurrency control and cache consistency are not an issue in this experiment. Write probability is varied on the x-axis and the overall system throughput, in commits per second, is measured. As seen in Figure 15.1, PageHard is outperformed by all of the architectures for

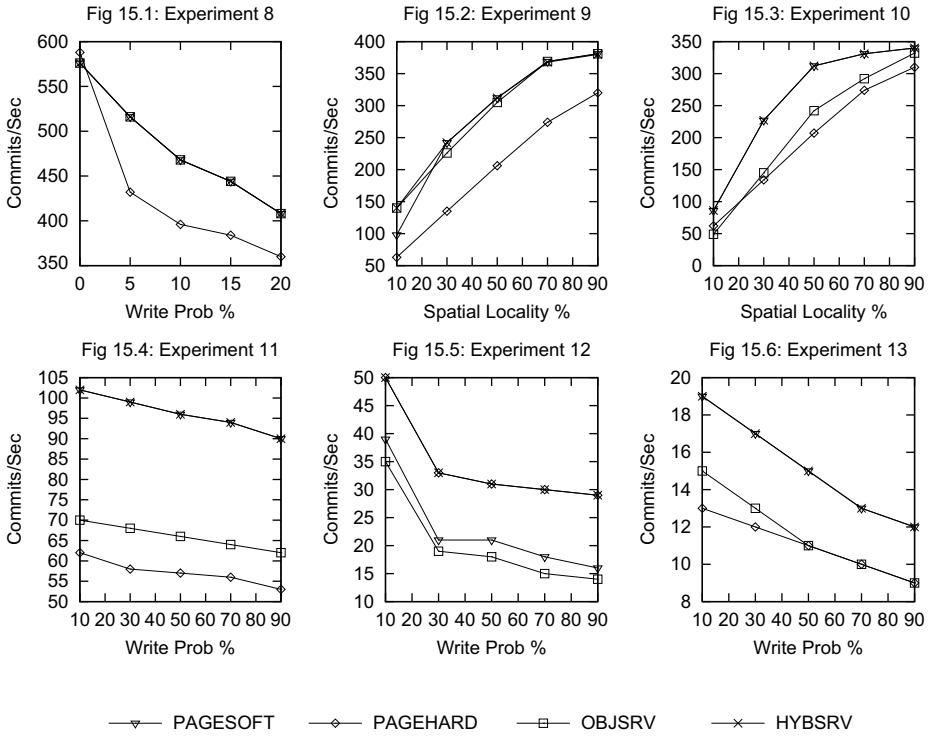

Figure 15. Data Shipping Architecture Peformance Study

write probabilities greater than zero. However, there is no difference between the performance of the different architectures that are using software pointer swizzling mechanism. Moreover, as shown in Figure 16, the pointer swizzling related costs are small in comparison to other costs. Therefore, they are not a major component in determining the overall performance in environments where the client cache can hold the entire client working set and the applications are performing some processing. For 0% write probability, PageHard slightly outperforms the other architectures because it does not encounter pointer indirection cost. However, as the write probability increases, the other architectures outperform PageHard, because they send updated objects to the server, whereas, PageHard sends sparsely updated pages to the server and encounters higher communication overhead.

4.2.6. *Small Client and Large Server Buffers*

In this system configuration (referred to as Small/Large) the client's working set does not fit into its cache even if the client has a lot of physical memory. This is possible if the size of the working set is very large or if the client buffer

Costs in microseconds/commit	PageHard	PageSoft	ObjServ	HybServ
Read	16500	16500	16500	16500
Write	7280	7280	7280	7280
Client-to-server data transfer	5665	1155	1155	1155
Pointer manipulation	0	249	249	249

Figure 16. Large-Large Buffer Setup: Private Workload Cost Breakdown

Costs in microseconds/commit	PageHard		PageSoft		ObjServ		HybServ	
	10%	30%	10%	30%	10%	30%	10%	30%
Client Cache Misses/Commit	33.9	8.9	25.2	7.9	26.4	10.3	25.3	7.9
Server-to-client data transfer	9409	3167	6393	1076	3070	1434	1072	1077

Figure 17. Good Access Locality Cost Breakdown

is shared by multiple applications. The client buffer is 1.5% of the database size and the server buffer is 75% of the database size. The primary goal of this configuration is to compare the server to client data transfer mechanisms. Network speed, page size, and clustering are varied for this buffer setup to assess the robustness of the different data transfer methods.

Experiment 9: Small-Large with Good Access Locality This experiment uses private workload with 10% write probability. The spatial locality has been varied to see the relationship between clustering and client buffer size. Temporal locality has been set at 50% and access locality has been set at 90%. As shown in Figure 15.2, PageHard performs the worst during low spatial locality because it manages the client cache strictly at page level and this leads to low client buffer utilization. The clients in ObjSrv, HybSrv and PageSoft only retain useful objects in their cache. As shown in Figure 17, the low client buffer utilization in PageHard leads to a higher number of client cache misses and this, in turn, degrades PageHard's performance. Returning updated pages instead of updated objects increases the network overhead encountered by PageHard and this also contributes towards the lower performance of PageHard. Since PageHard employs the hardware pointer swizzling mechanism, the cost of loading pages into the client cache is higher for PageHard than for the architectures employing the software pointer swizzling mechanisms.

The second important result is that ObjSrv and HybSrv outperform PageSoft during bad spatial locality, because, as shown in Figure 17, PageSoft encounters higher network overhead as a result of sending badly clustered pages from the server to the client. Since HybSrv is able to switch and operate

	PageHard	PageSoft	ObjServ	HybServ
Client Cache Misses/Commit	34.0	28.9	107.0	29.0
Server-to-Client Data Transfer Cost (Microseconds/Commit)	9088.0	7368.0	9226.0	7363.0

Figure 18. Bad Access Locality Cost Breakdown

as an object server, its performance is better than the architectures that send badly clustered pages to the clients.

Experiment 10: Small-Large with Bad Access Locality The purpose of this experiment is to assess whether sending a group of objects is beneficial when the access locality is bad. The setup of this experiment is similar to Experiment 9. However, in this case, access locality has been set to 10%. As seen in Figure 15.3, PageSoft and HybSrv outperform PageHard and ObjSrv. PageHard is outperformed as a result of the higher cost of loading pages into the client cache (due to hardware pointer swizzling) and returning updated pages to the server. ObjSrv is outperformed by PageSoft because in this workload access locality is bad, causing the ObjSrv server grouping mechanism to be less accurate. During bad access locality, multiple non-contiguous objects on a page are accessed together, and due to the inaccuracy of the object grouping mechanism, clients in ObjSrv have to make multiple data requests to the server. As shown in Figure 18, ObjSrv encounters a higher number of misses in the client cache, and, therefore, it sends more data requests to the server (incurs greater network overhead). Since HybSrv switches over to sending pages during periods of bad access locality, it outperforms ObjSrv. During periods of bad access and bad spatial locality, it is better to send pages to the clients and let the client dual buffering mechanism retain only useful objects in the client cache.

Thus, even during bad spatial locality, if the access locality is bad, it is desirable to send pages from the server to the client because storing pages improves the client cache hit rate. Experiments 9 and 10 are important because, unlike a previous study comparing grouped object servers and page servers [23], the results of these experiments show that the ObjSrv object grouping mechanism does not always outperform the architectures that send pages from the server to the client. Similarly, the page server architecture using a dual buffer at the client does not always outperform the grouped object server approach. These two experiments justify the need for an adaptive

hybrid server to client data transfer approach.

4.2.7. *Large Client and Small Server Buffers*

In Large/Small configuration, the server buffer is small and cannot hold the working sets of the active clients (contended server buffer), but the client buffer is large enough to hold the client's working set. The Large/Small experiment evaluates the different client to server data transfer mechanisms, assessing whether it is efficient to return log records, updated pages and log records, or switch between these options. In this experiment the client buffer is set at 12.5% of the database size and the server buffer size is varied between 10% and 1% of the database size. These experiments are run using the Private workload configuration because the focus here is to assess the performance of client to server data transfer and server buffer management mechanisms. The network speed is set at 100 Mbps. Write probability is varied in these experiments. The spatial, temporal and access localities have been set to 10%, 50% and 90%, respectively.

Experiment 11: Server Buffer with Medium Contention The purpose of this experiment is to assess the impact of client-to-server data transfer mechanism on server buffer management. In this experiment, the server buffer size has been set to 10% of the database size. As shown in Figure 15.4, PageSoft, HybSrv, and ObjSrv outperform PageHard because they return updated objects to the server, whereas PageHard returns updated pages to the server. Storing updated objects in the MOB increases the absorption capability of the server buffer. That is, the MOB allows for the batched installation of the multiple updated objects to their corresponding disk pages. As shown in Figure 19, the MOB helps the architectures returning updated objects to have fewer installation I/O operations than PageHard. Objects belonging to the same page that have been updated across multiple transactions by multiple clients can be installed together. However, in schemes that send updated pages to the server, if data clustering is poor, then whole pages are stored at the server even if only a small portion of the page has been updated. This results in low server buffer utilization, and, thus, the writing of updated pages to disk interfere with the normal read disk traffic (to satisfy client read requests). Since HybSrv returns updated objects, its performance is also better than PageHard's performance.

Another key result of this experiment is that ObjSrv's performance trails PageSoft and HybSrv performance. Unlike Experiment 9, PageSoft outperforms ObjSrv, because with a small server buffer, a client cache miss also results in a server cache miss. Thus, the importance of client cache management accuracy becomes more important when server buffer is small. HybSrv also outperforms ObjSrv because in HybSrv, the server realizes that its buffers

	PageHard	PageSoft	ObjServ	HybServ
Installation I/Os per Commit	13.7	10.8	10.8	10.8

Figure 19. Experiment 11: Medium Server Buffer Contention

	PageHard	PageSoft	ObjServ	HybServ
Installation I/Os per Commit	14.2	18.4	18.4	14.2

Figure 20. Experiment 12: High Server Buffer Contention

are contended, and thus it sends pages to the clients to try to minimize the client cache misses.

Experiment 12: Highly Contended Server Buffer The purpose of this experiment is to assess the impact of client to server data transfer mechanism on server buffer management when the server buffers are very contended. In this experiment, the server buffer size has been set to 1% of the database size. This experiment is supposed to represent the situation where the server buffer is extremely contended due to the simultaneous processing of multiple client requests. The server disk utilization varies between 90 and 95 percent. As shown in Figure 15.5, PageHard and HybSrv outperform ObjSrv and PageSoft because PageHard and HybSrv return updated pages to the server, whereas ObjSrv and PageSoft return updated objects to the server. In HybSrv, the server realizes that its buffers are contended and it sends a hint to the client indicating that the client should return updated pages. Since the server buffer is contended, the MOB absorption capability is very low. As shown in Figure 20, in schemes that return updated objects to the server, there is a low chance for batching the installation of multiple object updates to a page. Therefore, the schemes returning updated objects perform a higher number of installation I/Os than the schemes returning updated pages to the server.

4.2.8. *Small Client and Small Server Buffers*

In Small/Small configuration, the server buffer is too small to hold the working sets of the active clients and the client buffer is small and it cannot hold the working set of the client. In this experiment, both the client and the server buffer sizes have been set to 1% of the database size. Since both server and client buffers are contended, the Small/Small buffer configuration helps to evaluate whether the client buffer or the server buffer has more impact on the overall performance. This experiment uses the Private data sharing pattern. The spatial locality, access locality and temporal locality values have been set to 10, 90 and 50%, respectively.

Experiment 13: Small-Small The purpose of this experiment is to assess the relative importance of client and server buffers. As shown in Figure 15.6, PageSoft, HybSrv and ObjSrv outperform PageHard. Even though, as in Experiment 12, the server buffer configuration is very small, unlike Experiment 12, PageHard's performance trails the performance of the PageSoft and HybSrv because PageHard manages the client buffers strictly at page level. Since the spatial locality of the data access pattern is low, PageHard encounters low client buffer utilization. Thus, the gains made by PageHard due to the absence of installation reads at the server (during high server buffer contention) are mitigated due to the low client buffer utilization. Thus, this experiment shows that the efficiency of client buffer management is more important than the efficiency of the server buffer management. ObjSrv's performance trails PageSoft and HybSrv performance because of the high probability of a miss in the client cache being also a miss in the server cache. Since the temporal locality in this experiment is only 50%, ObjSrv incurs more client cache misses than PageSoft and HybSrv due to the inability of the server object grouping mechanism to construct accurate object groups.

5. Conclusions and Future Work

In this paper, we proposed a new *adaptive* server architecture that incorporates new adaptive data transfer, cache consistency and recovery mechanisms. A prerequisite of adaptiveness is a hybrid server architecture that can efficiently handle both disk pages and logical objects. The hybrid server architecture incorporates a new concurrency control enhancement.

As shown by the performance study in Section 4, the existing client-server architectures and algorithms are not robust across different workloads and system configurations. Hence the need for adaptive algorithms which can dynamically adapt as the workload and system environment changes. Adaptive systems that minimize system tuning and the configuration activities of programmers and system administrators have been identified as an important database system research area [17, 3, 18]. The work presented in this paper addresses this important research area within the context of client-server ODBMSs. The reported performance study verifies that it is possible to develop more robust adaptive algorithms and systems. The cache consistency study and the integrated performance study provide many interesting insights, some of which overturn commonly accepted beliefs. These are discussed below.

5.1. CACHE CONSISTENCY STUDY CONCLUSIONS

A study that compares ACBL and AOCC cache consistency/concurrency control algorithms has shown that AOCC, which is an optimistic algorithm, outperforms ACBL, a pessimistic algorithm, even while encountering a high abort rate [1]. However, this paper has shown that even within the client-server ODBMS context, algorithms such as AACC can provide a low abort rate and can outperform optimistic algorithms, such as AOCC, with respect to overall system throughput.

AACC is an asynchronous cache consistency algorithm which outperforms the synchronous ACBL cache consistency algorithm. Previously, it was thought that synchronous cache consistency, such as Callback Locking (CBL) algorithms, outperform asynchronous cache consistency algorithms, such as No-Wait-Locking-Notify (NWL-Notify) [36], because asynchronous algorithms incur higher abort rates. In this paper we have shown that an asynchronous algorithm such as AACC consistently outperforms a synchronous algorithm such as ACBL. The reason for this is that NWL-Notify is a detection-based algorithm and therefore encounters stale cache aborts. As an avoidance-based algorithm, AACC does not have this problem. Furthermore, NWL-Notify does not have an efficient abort processing mechanism as present in AOCC. It is the combination of optimistic detection-based and an inefficient abort processing mechanism allows CBL to outperform NWL-Notify.

AACC algorithm has a better combination of performance and abort rate than both ACBL or AOCC because it incorporates the following key enhancements:

- **Avoidance-Based:** AACC is an avoidance-based algorithm that, as stated above, does not encounter stale cache aborts. As shown in Section 4, the deadlock abort rate of AACC is much lower than the stale cache abort rate of AOCC. This, in turn, allows AACC to outperform AOCC for many key workloads and system configurations. Previously, detection-based asynchronous cache consistency algorithms were thought to be abort prone. However, AACC is avoidance-based and, therefore, it does not encounter stale cache aborts. Furthermore, AACC uses new deadlock avoidance techniques which ensure that its deadlock abort rate is as low as ACBL's abort rate. We have shown that asynchronous messaging and avoidance-based notions are a good combination.

- **Shared/Private Regions:** The notions of shared and private page lock modes contribute to AACC's good performance when working with private workloads because they reduce the number of explicit messages. Unlike ACBL, which sends explicit lock escalation messages when updating pages that are only accessed by a single client, in AACC the server informs the clients that these pages are only present at the par-

ticular client (private lock mode), and thus the client piggybacks its lock escalation messages to the server.

- **Piggybacking Callback Messages:** In AACC, when a client receives a callback message, it sends an explicit callback response only if there is an object-level conflict. Otherwise, the client piggybacks its callback response to the server. Piggybacking of callback messages, in conjunction with piggybacking lock escalation messages for private pages, reduces the total number of messages sent between the client and the server. This helps AACC to outperform ACBL.

- **Asynchronous Messages:** AACC uses asynchronous lock escalation messages, which do not incur the blocking overhead common to systems that use synchronous lock escalation messages. The blocking overhead increases when the server and the network are heavily utilized. Asynchronous messages also reduce the number of deadlocks that are present in algorithms using deferred lock escalations. By sending the intent-to-update message to the server immediately, AACC reduces the window in which deadlocks can occur. One of the major drawbacks of deferred avoidance-based algorithms, such as O2PL [12], is that they incur a high deadlock rate due to deferring its lock escalation messages until commit time.

In addition to proposing the AACC algorithm for page servers, we have also adapted the AACC algorithm for object servers. Previously, data transfer and cache consistency/concurrency have been shown to be orthogonal to each other for page servers. We have extended this orthogonality to object servers.

5.2. Integrated Study Conclusions

A new adaptive data transfer algorithm has been proposed and its performance has been evaluated as part of the integrated performance study. The adaptive data transfer mechanism contains the following new features which help it to be robust, with respect to performance, as the workload and system configuration change:

- **Adaptive Server-to-Client Data Transfer:** The adaptive server-to-client data transfer mechanism helps to reduce the network overhead in situations where the clients access badly clustered pages. This optimization is very useful in low bandwidth environments, such as mobile networks and slow speed modem connections. The adaptive server-to-client data transfer mechanism can be used by page server architectures that employ a dual client buffer.

- **Adaptive Client-to-Server Data Transfer:** The adaptive client-to-server data transfer mechanism proposed here takes server buffer contention

level, client buffer management, and network cost into account when deciding whether to return updated pages or objects to the server. The previous client-to-server data transfer approaches did not take server buffer contention level into account [16, 28]. The adaptive client-to-server data transfer mechanism can be used by existing page server architecture systems.

- **Support for Varying Object and Page Sizes:** The previous object group forming mechanism [23] did not consider varying object sizes and page sizes into account, whereas the object group forming mechanism used by the adaptive data transfer mechanism handles varying object and page sizes. This object group forming mechanism can be used by existing grouped object server architectures.

- **Support for Varying Access Locality:** The previous object group forming mechanism [23] did not account for non-contiguous access to a page because the clients only kept track of the number of objects that have been accessed in the client cache, and did not care about the access locality characteristics. Therefore, the previous object group forming mechanism did not consider the notion of access locality; as a consequence, the performance of grouped object servers suffers during bad access locality. The adaptive data transfer mechanism presented here takes access locality characteristics into account and uses this information to switch between requesting pages and object groups. This optimization can be used by the existing object server architectures.

We have also proposed a new object server recovery algorithm. All of the previous client-server recovery work has been conducted within the context of page servers. Moreover, this is also the first time that recovery issues have been studied for architectures where updates are performed both at the clients and the server.

The integrated performance study has provided the following useful insights:

- It is desirable to have an adaptive data transfer architecture where pages and objects can be transferred both from the server to the client, and from the clients to the server.

- Concurrency control enhancements that allow object servers to use AACC ensure that object servers can efficiently use low abort algorithms, and hence they can compete with page servers. Thus, concurrency control is not a liability for object servers. Previously, it was shown that object servers cannot efficiently use a pessimistic concurrency control algorithm [9].

- Previously, the redo-at-server recovery paradigm was shown to be unscalable [38]. In Section 4, it has been shown that a MOB allows the redo-at-server recovery paradigm to successfully compete with an ARIES-CSA style recovery mechanism which sends both log records and updated pages to the server.

- A previous study focusing solely on pointer swizzling [37] has shown that the hardware swizzling outperforms software swizzling for most workloads. However, we have shown that the architectures using software pointer swizzling outperform the architectures that employ hardware swizzling for most workloads and system configurations since the latter employ page-level client buffer management, page-level locking and page-level client to server data transfer mechanisms.

- Previously, it was shown that the object grouping mechanism allows grouped object servers to outperform page servers [23]. However, the performance study in Section 4 has shown that object grouping techniques that are executed at the server are only effective if the data access pattern has high access locality. Usually, the inefficiency of the server-based object grouping mechanisms leads to higher client cache miss rates, and this, in turn, leads to a greater number of object requests being sent from the clients to the server. It is preferable to use dual page/object buffers at the client because they allow clients to have a high client cache hit rate even during periods of low access locality by caching pages, and they allow the clients to discard badly clustered pages when the page spatial locality is low.

- Previously, two separate studies on MOBs arrived at different conclusions with respect to whether it is beneficial to send updated pages or objects to the server. The initial study [28] indicated that it is better to return updated pages to the server. A subsequent study [16] countered that it is better to return updated objects to the server. The results in this paper give the insight that the server buffer size is a key factor which determines whether it is desirable to return updated pages or objects to the server, thus, clarifying the previous results. If the server buffer size is very small then it is better to return updated pages, otherwise it is better to return updated objects.

5.3. Future Work

The use of the new data transfer, cache consistency and recovery algorithms is not limited to only the client-server ODBMS domain. Instead, these algorithms with proper modifications can be also utilized by the other emerging client-server architectures.

The adaptive data transfer mechanism will be most useful in mobile environments. These environments have network bandwidth and battery power constraints, and therefore, it is important for mobile clients to not transmit and receive poorly clustered pages. Transferring poorly clustered pages has higher network latency and higher consumption of client battery power than when only isolated objects from poorly clustered pages are transferred. Since the adaptive data transfer algorithm tries to send only relevant data across the network it will be very useful in mobile environments. It is also not desirable to send poorly clustered pages in wide-area Web architectures where data need to be encrypted because encryption of unused data wastes system resources.

The adaptive cache consistency algorithm (AACC) developed in this paper will be useful in mobile, Web, and hybrid function-shipping data-shipping environments because it has a better combination of low abort rate and high performance than the existing client-server cache consistency algorithms. Re-execution of aborted transactions at mobile clients increases battery power consumption. Thus, the low abort feature of AACC is important for mobile environments. The low abort feature is also important for Web applications that require end user interaction during transaction aborts because end users do not like to see their transactions abort. Finally, a low abort rate is also necessary in environments where server resources are contended. When the server resources (such as CPU and disks) are contended, the re-execution of an aborted transaction has an impact on the performance of all the other user transactions running at the server. In multi-tiered Web server architectures, intermediate servers can also act as clients with respect to other servers in the hierarchy. Therefore, it is desirable to have low aborting cache consistency algorithms that do not require clients to re-execute aborted transactions because this will result in high CPU utilization at the mid-tiered server nodes.

References

1. Adya, A., R. Gruber, B. Liskov, and U. Maheshwari: 1995, 'Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks'. In: *Proc. ACM SIGMOD International Conference on Management of Data*. pp. 23–34.
2. Agrawal, R., M. Carey, and M. Livny: 1987, 'Concurrency Control Performance Modeling: Alternatives and Implications'. *ACM Transactions on Database Systems* **12**(4), 609–654.
3. Bernstein, P., M. Brodie, S. Ceri, M. Franklin, and et al: 1998, 'Asilomar Report'. *SIGMOD Record* **27**(4), 74–80.
4. Biliris, A. and E. Panagos: 1995, 'A High Performance Configurable Storage Manager'. In: *Proceedings of ICDE*. pp. 35–43.
5. Cao, Q., J. Torrellas, and H. Jagadish: 2000, 'Unified Fine-Granularity Buffering of Index and Data: Approach and Implementation'. In: *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors*. pp. 175–186.

6. Carey, M., D. DeWitt, M. Franklin, N. Hall, and et al.: 1994a, 'Shoring Up Persistent Applications'. In: *Proc. ACM SIGMOD International Conference on Management of Data*. pp. 383–394.
7. Carey, M., D. DeWitt, and J. Naughton: 1993, 'The OO7 Benchmark'. In: *Proc. ACM SIGMOD International Conference on Management of Data*. pp. 12–21.
8. Carey, M., M. Franklin, M. Livny, and E. Shekita: 1991, 'Data Caching Tradeoffs in Client-Server DBMS Architectures'. In: *Proc. ACM SIGMOD International Conference on Management of Data*. pp. 357–366.
9. Carey, M., M. Franklin, and M. Zaharioudakis: 1994b, 'Fine Grained Sharing in a Page Server OODBMS'. In: *Proc. ACM SIGMOD International Conference on Management of Data*. pp. 359–370.
10. Castro, M., A. Adya, B. Liskov, and A. Myers: 1997, 'HAC:Hybrid Adaptive Caching for Distributed Storage Systems'. In: *Proceedings of ACM Symposium on Operating System Principles*. pp. 102–115.
11. DeWitt, D., P. Futtersack, D. Maier, and F. Velez: 1990, 'A study of three alternative workstation-server architectures for OODBS'. In: *Proc. 16th International Conference on Very Large Data Bases*. pp. 107–121.
12. Franklin, M., M. Carey, and M. Livny: 1992a, 'Global Memory Management in Client-Server DBMS'. In: *Proc. 16th International Conference on Very Large Data Bases*. pp. 596–609.
13. Franklin, M., M. Carey, and M. Livny: 1997, 'Transactional Client-Server Cache Consistency: Alternatives and Performance'. *ACM Transactions on Database Systems* **22**(3), 315–363.
14. Franklin, M., M. Zwilling, C. Tan, M. Carey, and D. DeWitt: 1992b, 'Crash Recovery in Client-Server EXODUS'. In: *Proc. ACM SIGMOD International Conference on Management of Data*. pp. 165–174.
15. Gerlhof, C. and A. Kemper: 1994, 'A Multi-Threaded Architecture for Prefetching in Object Bases'. In: *Proc. 4th EDBT Conference*. pp. 351–364.
16. Ghemawat, S.: 1995, 'The Modified Object Buffer: A Storage Management Technique for Object-Oriented Databases'. Ph.D. thesis, MIT.
17. Gray, J.: 1999, 'What Next? A dozen remaining IT problems'. In: *Turing Award 1999 Lecture: www.research.microsoft.com/ gray/*.
18. Hamilton, J.: 1999, 'Networked Data Management Design Points'. In: *Proc. 25th International Conference on Very Large Data Bases*. pp. 202–206.
19. Karlsson, J., A. Lal, C. Leung, and T. Pham: 2001, 'IBM DB2 Everyplace: A Small Footprint Relational Database System'. In: *Proc. 17th International Conference on Data Engineering*. pp. 230–232.
20. Kemper, A. and D. Kossmann: 1994, 'Dual-Buffering Strategies in Object Bases'. In: *Proc. 20th International Conference on Very Large Data Bases*. pp. 427–438.
21. Kim, W., J. Garza, N. Ballou, and D. Woelk: 1990, 'Architecture of the ORION Next-Generation Database System'. *IEEE Transactions on Knowledge and Data Engineering* **2**(13), 109–124.
22. Lamb, C., G. Landis, J. Orenstein, and D. Weinreb: 1991, 'The ObjectStore database system'. *Communications of the ACM* **34**(10), 50–63.
23. Liskov, B., A. Adya, M. Castro, M. Day, and et al: 1996, 'Safe and Efficient Sharing of Persistent Objects in Thor'. In: *Proc. ACM SIGMOD International Conference on Management of Data*. pp. 318–329.
24. Luo, Q., S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton: 2002, 'Middle-tier Database Caching for e-Business'. In: *Proc. ACM SIGMOD International Conference on Management of Data*. pp. 600–611.

25. Mohan, C., D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz: 1992, 'ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging'. *ACM Transactions on Database Systems* **17**(1), 94–162.
26. Mohan, C. and I. Narang: 1994, 'ARIES/CSA: A Method for Database Recovery in Client-Server Architectures'. In: *Proc. ACM SIGMOD International Conference on Management of Data*. pp. 55–66.
27. Objectivity: 1998, 'White Paper: Choosing an Object Database'. In: *www.objectivity.com/ObjectDatabase/WP/Choosing/Choosing.html*.
28. O'Toole, J. and L. Shrira: 1994a, 'Hybrid caching for large scale object systems'. In: *Proc. Workshop on Persistent Object Systems*. pp. 99–114.
29. O'Toole, J. and L. Shrira: 1994b, 'Opportunistic Log: Efficient Installation Reads in Reliable Object Server'. In: *Proc. 1st Usenix Symposium on Operating Systems Design and Implementation OSDI*. pp. 39–48.
30. Özsu, M.T., K. Voruganti, and R. Unrau: 1998, 'An Asynchronous Avoidance-based Cache Consistency Algorithm for Client Caching DBMSs'. In: *Proc. 24th International Conference on Very Large Data Bases*. pp. 440–451.
31. Panagos, E., A. Biliris, H. Jagadish, and R. Rastogi: 1996, 'Fine-granularity Locking and Client-Based Logging for Distributed Architectures'. In: *Proceedings of EDBT Conference*. pp. 388–402.
32. Tsangaris, M. and J. Naughton: 1992, 'On the performance of object clustering techniques'. In: *Proc. ACM SIGMOD International Conference on Management of Data*. pp. 144–153.
33. Versant: 1998, 'Versant Delivers Next Generation Suite of Database Products'. In: *http://www.versant.com*.
34. Voruganti, K.: 2001, 'An Adaptive Hybrid Server Architecture for Client-Caching ODBMSs'. Ph.D. thesis, University of Alberta.
35. Voruganti, K., M.T. Özsu, and R. Unrau: 1999, 'An Adaptive Hybrid Server Architecture for Client Caching ODBMSs'. In: *Proc. 25th International Conference on Very Large Data Bases*. pp. 150–161.
36. Wang, Y. and L. Rowe: 1991, 'Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture'. In: *Proc. ACM SIGMOD International Conference on Management of Data*. pp. 367–376.
37. White, S. and D. DeWitt: 1994, 'QuickStore: A high performance mapped object store'. In: *Proc. ACM SIGMOD International Conference on Management of Data*. pp. 395–406.
38. White, S. and D. DeWitt: 1995, 'Implementing Crash Recovery in QuickStore: A Performance Study'. In: *Proc. ACM SIGMOD International Conference on Management of Data*. pp. 187–198.
39. Wilkinson, K. and M. Neimat: 1990, 'Maintaining Consistency of Client-Cached Data'. In: *Proc. 16th International Conference on Very Large Data Bases*. pp. 122–133.