An Adaptive Hybrid Server Architecture for Client Caching Object DBMSs*

Kaladhar Voruganti M. Tamer Özsu
University of Alberta
Edmonton, Canada

Ronald C. Unrau
Cygnus Solutions
Sunnyvale, California

Abstract

Current client-server object database management systems employ either a page server or an object server architecture. Both of these architectures have their respective strengths, but they also have key drawbacks for important system and workload configurations. We propose a new hybrid server architecture which combines the best features of both page server and object server architectures while avoiding their problems. The new architecture incorporates new or adapted versions of data transfer, recovery, and cache consistency algorithms; in this paper we focus only on the data transfer and recovery issues. The data transfer mechanism allows the hybrid server to dynamically behave as both page and object server. The performance comparison of the hybrid server with object and page servers indicates that the performance of the hybrid server is more robust than the others.

1 Introduction

We propose a new hybrid server architecture for client-server object database management systems (ODBMSs) which can dynamically adapt and operate either as a page server or as an object server. Our fundamental thesis is that page servers and object servers are generally preferable under a limited set of system and workload configurations [DFMV90, CFZ94, DFB+96] and adaptive techniques that combine the features of both are likely to respond better to a larger class of system and workload configurations. Moreover, object servers are not as popular as page servers, because previous performance studies have shown that object servers are not scalable with respect to data transfer [DFMV90] and concurrency control/cache consistency algorithms [CFZ94]. In this paper we present new techniques and report performance results that show that objects servers can compete with page servers.

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

Proceedings of the 25th VLDB Conference
Edinburgh, Scotland, 1999

1.1 Background

Object servers and page servers are the two competing data-shipping architectures employed by the existing ODBMSs [DFMV90, CFZ94]. In data-shipping systems, the clients fetch data from the server into their caches and perform some of the database processing locally. Thus, from scalability standpoint, clients help the server from becoming a bottleneck by off-loading some of the work. Prefetching useful data also reduces effective network latency. Figure 1 presents a classification of the different data-shipping ODBMSs according to their data transfer mechanism. In data-shipping object server systems (THOR, Versant), the server responds to client data requests by returning logical objects to the client (the first column in Figure 1). In page server data-shipping systems (ObjectStore, BeSS, O2, SHORE), the server responds to client data requests by returning physical disk pages to the client (the second column in Figure 1). Thus, the granularity of the data transferred from the server to the clients is the key distinguishing factor between object and page servers. The data transfer granularity from the server to the client, in turn, has an impact on buffer management, concurrency control, recovery and pointer swizzling algorithms. These issues are discussed in Section 2.

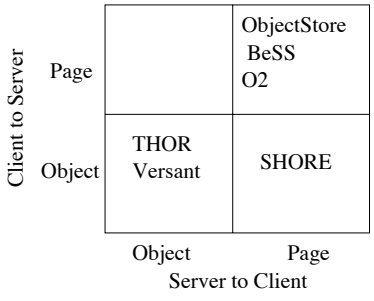

Figure 1: ODBMS Client-Server Architecture Classification According to the Data Transfer Mechanism

1.2 Motivation For Adaptive Hybrid Server

The need for a hybrid client-server architecture was recognized from the very beginning of client-server ODBMS research [DFMV90], because the page server and object server architectures are desirable under different (but overlapping) sets of workloads and system configurations. However, there has been no hybrid server system design that can dynamically change its mode of operation to effectively satisfy the needs of different workloads and system configurations.

Page server systems can outperform object servers when the application data access pattern matches the data clustering pattern on disk (which we refer to in the rest of the paper as good clustering) [DFMV90]. By receiving pages under good clustering, the clients in the page server architecture are able to prefetch useful objects that they will likely use in the future. Prefetching helps to amortize communication costs in page servers. In comparison, object servers incur higher communication overhead since they transfer individual objects from the server to the client [DFMV90]. However, when the data clustering pattern on disk does not match the data access pattern (bad clustering), transferring the entire page from the server to the clients is counter-productive since this increases the network overhead and decreases client buffer utilization. Dual client buffer schemes [KK94, CALM87], which allow the storage of well clustered pages and isolated objects from badly clustered pages help improve client buffer utilization.

Page servers are inefficient for the emerging hybrid function-shipping/data-shipping architectures [KJF96] where, in addition to requesting data from the server, the clients also send queries to be processed at the server. The server processes the queries and returns only the results back to the client. If a query result is spread across multiple disk pages each of which contains only a few objects, then it is very inefficient to send all of the disk pages to the client [DFB+96].

An important study on clustering [TN92] has shown that it is difficult to come up with good clustering when multiple applications with different data access patterns access the same data. Therefore, good clustering cannot be taken for granted and the problem of transferring badly clustered pages is a fundamental issue in page servers.

The data transfer problem of object servers can be resolved by transferring a group of objects rather than a single object from the server to the client (so called grouped object servers). A dynamic object grouping mechanism has been proposed [LAC+96] that makes grouped object servers competitive with page servers with respect to the data transfer mechanism. However, grouped object servers incur higher group forming/breakup overhead than page servers when clustering is good. Thus, there is a need for an architecture which deals with pages when the clustering is good. Cache consistency/concurrency control is still a problem in object servers because they send separate lock escalation messages (for escalating from read lock to write lock) to the server for each object [CFZ94]. Page servers are able to send lock escalation messages at the granularity of a page and are thus able to reduce locking related communication overhead. In order to avoid this problem, object servers typically use optimistic concurrency control algorithms which have no explicit lock escalation messages [LAC+96]. However, the resulting high abort rate is undesirable from performance and usability standpoints [OVU98]. Thus, at present, there does not exist an efficient, low abort concurrency control/cache consistency algorithm for object servers. Object server recovery has been a neglected area, perhaps partially as a result of giving up on object servers due to their perceived problems. Currently there are no published object server recovery algorithms that are comparable, performance-wise, to the page server recovery algorithms [MN94, FZT+92]. Finally, when clients in the object server architecture return updated objects to the server, the server has to re-install these updated objects on their corresponding home page (redo-at-server recovery) before writing the page back to disk. If the server buffer is heavily contended, then the home pages might not be present in the server buffers necessitating installation reads to retrieve the home pages from disk. The proponents of object servers have introduced the notion of a modified object buffer (MOB) [Ghe95] at the server. The MOB stores the updated objects that have been returned by the clients. The MOB helps to intelligently schedule a group of installation reads and thus reduce the installation read overhead. However, in workloads where the clients are updating large portions of a page, it is desirable to return updated pages to the server. Thus, a client-server architecture must be flexible and should allow for the return of updated pages from the client to the server.

1.3 Paper Scope and Contributions

In this paper we propose an adaptive hybrid server architecture which adjusts its behavior between a page and an object server, and is thus able to incorporate the strengths of both page and object servers. The adaptive hybrid server architecture incorporates the following contributions:

1. An adaptive data transfer mechanism that dynamically decides whether to ship pages or objects between the server and the client.

2. A new efficient object server recovery mechanism, which is used by the hybrid server and can also be adopted by existing object server architectures.

3. An efficient, low abort rate cache consistency algorithm for object servers. *Asynchronous avoidance-based cache consistency* (AACC) algorithm [OVU98] for page servers has been shown to have both good performance and and a low abort rate. We adapt AACC for object servers.

We compare the performance of the hybrid server with three other prominent architectures: grouped object servers, page servers that use hardware page faulting, and page servers that use software page handling. The results, reported in Section 5, show that performance-wise the hybrid server architecture is more robust than the others. Besides these results, the performance study is important in its own right for two reasons:

Data Transfer		Cache Consistency				Buffer Management		Recovery		Pointer Swizzling	
Page	Object	Page		Object		Page	Object	Page	Object	Page	Object
		Low Abort	High Abort	Low Abort	High Abort				[KGBW90]	[LLOW91]	
[DFVM90]								[FZT+92]			
[OS94]	[GK94]	[FC94]		[CFZ94]		[KK94]		[MN94]		[WD94]	
[BP95]		[CFZ94]	[AGLM95]		[AGLM95]		[Ghe95]	[WD95]			
	[LAC+96]	[FCL96]				[CALM97]		[PBJR96]			[LAC+96]
	[This Paper]	[OVU98]		[This Paper]					[This Paper]		[This Paper]

Figure 2: Decade of research into page and object server ODBMSs

1. This is the first multi-user client-server performance study that compares the performance of page servers and grouped object servers. Previous studies either focused on single-user systems [DFMV90, CDN93, LAC+96], or they did not consider grouped object servers [DFMV90, CFZ94].

2. This is the first multi-user client-server performance study that looks at data transfer, buffer management, cache consistency, concurrency control, recovery and pointer-swizzling system components in an integrated manner. Currently, the existing ODBMS products use different combinations of algorithms for these system components, and due to the interaction between them, it is difficult to properly assess the strengths and weakness of the different architectures under a range of important workloads.

1.4 Paper Organization

Section 2 discusses the related work. Section 3 presents the new hybrid server architecture. Section 4 describes the experimental setup and workloads. In Section 5 we present results of our experiments. Section 6 contains a discussion of the experiment results and Section 7 contains our conclusions.

2 Related Work

Figure 2 summarizes the client-server ODBMS research that has been performed over the last decade on topics relevant to this paper. These play a significant role in determining the overall performance of data-shipping client-server ODBMSs. In this section we summarize the research that has been conducted into data transfer, cache consistency, buffer management, recovery and pointer swizzling.

Data Transfer: The initial client-server performance study [DFMV90] identified page server, object server, and file server as the three alternative client-server architectures. A key conclusion was that page servers are desirable when the data access pattern matches the data clustering pattern on disk. This study prompted the development of prefetching techniques for both page and object servers [GK94, LAC+96]. A static hybrid data transfer mechanism has been implemented in Ontos [CDN93] in which, for each object type, the application programmers specify whether they want to deal with objects or pages. In this paper we propose a dynamic hybrid data transfer mechanism which uses run-time information to automatically adapt itself and operate as either a page or an object server. A partial hybrid server architecture in which the server always sends pages to the clients, but the clients can dynamically choose to return either updated pages or updated objects has been proposed [OS94]. This flexibility requires revisions in the concurrency control and recovery mechanisms, but these have not been addressed in the partial hybrid server proposal. Moreover, the study was in favor of architectures that return pages because clients updated a large portion of a page. This increased server buffer utilization and led to their algorithm always returning updated pages if they are present in the client cache. Our performance results show that returning pages is not always desirable. Finally, while dealing with large objects that span multiple pages, it is desirable to be able to transfer only portions of the large object between clients and servers. Both page and object servers have to be modified to ensure that the entire large object is not transferred as a single unit between the server and the client. Our proposal is a full hybrid architecture where both the server and the clients can transfer pages or objects. As discussed in Section 3, our proposal addresses many of the issues raised above.

Cache Consistency: Most of the existing cache consistency research has been conducted within the context of page servers (for an excellent survey see [FCL96]). For most user workloads, invalidation of remote cache copies during updates is preferred over propagation of updated values to the remote client sites. Moreover, caching of read-locks across transactions at the clients is preferred over caching of both read and write-locks [FC94]. It has been shown that adaptive locking algorithms that switch between page level and object level locks are more robust than purely page level locking algorithms [CFZ94]. Cache consistency algorithms have been classified as avoidance-based and detection-based algorithms [FCL96]. Avoidance-based algorithms do not allow for the presence of stale data (data that has been updated and committed by remote clients) in client caches, whereas detection-based algorithms permit clients to have stale data in their caches and abort transactions (*stale cache* aborts) when they have accessed stale data. Client cache consistency algorithms can also be classified according to

when the clients send a lock escalation message to the server. Clients can send lock escalation messages in a synchronous, asynchronous or deferred manner [FCL96]. In a previous paper we showed that a new adaptive algorithm, called *asynchronous avoidance-based cache consistency* (AACC), outperforms other leading cache consistency algorithms [OVU98] such as ACBL [CFZ94], which is a synchronous avoidance-based algorithm, and AOCC [AGLM95] which is a deferred detection-based algorithm. AACC is an avoidance-based algorithm, and, therefore, it has a much lower abort rate than AOCC. It uses asynchronous lock escalation messages to have lower blocking overhead than ACBL, and a lower abort rate than deferred avoidance-based algorithms. AACC also piggybacks lock escalation messages while updating data that is not cached at multiple clients to reduce its message handling overhead. Finally, since there currently does not exist a cache consistency algorithm for object servers that provides both high performance and low abort rate we have modified the page-based AACC for object servers.

Pointer Swizzling: The pointer-swizzling research has led to the development of hardware-based and software-based algorithms [WD94]. In the former the page level virtual memory facilities provided by the operating system are used to eagerly swizzle the pointers on a page. In the latter, 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 software swizzling, the swizzled pointers usually point to the target object's entry in an indirection data structure (object table). Thus, there is a level of indirection while traversing from the source object to the target object. Page servers can use either the hardware or the software pointer swizzling mechanism, but object servers use only software-based pointer swizzling, since it is inefficient to reserve virtual memory frames at the object level. Currently, most of the hardware pointer swizzling mechanisms store the swizzled in-memory version of the object identifiers on disk [WD94, LLOW91]. Since the in-memory version of an object identifier is usually smaller than the on-disk version, this reduces the size of the objects on disk, which, in turn, leads to fewer disk I/Os (in comparison to the software swizzling algorithms) while reading the objects from disk [WD94]. The second advantage of storing swizzled pointers on disk is that if a client's working set fits completely into its virtual memory address space, then the hardware pointer swizzling mechanism has to swizzle the on-disk version of the object identifier only once during its initial access. Storing of swizzled pointers on disk is an important strength of the hardware swizzling mechanism. However, the hardware pointer swizzling creates problems related to object migration (both between pages and across systems) and deletion, since there is no level of indirection between the source and the target objects. Finally, since the hardware swizzling relies on operating system provided page faulting mechanism, it usually employs page level locking, data transfer, recovery and buffer management mechanisms [LLOW91, BP95] and this has negative performance implications as demonstrated by our performance study.

Recovery: In page servers, clients can send to the server either updated pages (whole-page logging), updated pages and log records [FZT+92, MN94], or log records (redo at server) [WD95]. For page servers, returning both pages and log has been shown, in general, to have the best normal operation performance [WD95]. Existing page server recovery mechanisms use the STEAL/NO-FORCE buffer management policy where the pages on stable storage can be overwritten before a transaction commits, and pages do not need to be forced to disk in order to commit a transaction. STEAL/NO-FORCE is generally regarded as the most efficient buffer management policy, but the published object server recovery proposals [KGBW90] do not use it. The need for an efficient object server recovery algorithm has been identified as an outstanding research problem [FZT+92, MN94]. It has also been shown that for ODBMS workloads, it is not desirable to generate a log record for each update since the same object can be updated multiple times within a transaction. Instead, it is more efficient to perform a *difference* operation at commit time between the before-update and after-update copies of data and to generate a single log record [WD95].

Buffer Management: Buffer management innovations have been made for both page and object servers. Dual buffer management techniques can be utilized by clients in page servers to increase client buffer utilization [CALM87, KK94]. Dual buffering allows buffering both well clustered pages and isolated objects from badly clustered pages. Object servers can use the modified object buffer (MOB) at the server that stores the updated objects that are returned by the clients [Ghe95]. The objects stored in the MOB have to be installed on their corresponding home pages and written back to disk. The MOB allows the server to intelligently schedule installation reads using a low priority process and thus helps to amortize the installation read cost.

3 Adaptive Hybrid Server Design

The fundamental principle of the hybrid server is that it can dynamically adapt and behave as both page and as grouped object server; hence the name *adaptive hybrid server*. The hybrid server architecture that we propose in this paper has the following key components:

Data Transfer: A key feature is the ability to dynamically adapt between sending pages or a group of objects between the server and the client. Both the client and the server have an important role to play to make the adaptive behavior feasible (details are presented in Section 3.1).

Buffer Management: Since clients and the server can receive either pages or objects, both client and server buffers need to be able to handle pages and objects. The client implements a dual buffer manager similar to [KK94] which can manage both pages and objects. The server contains a staging page buffer to store the pages that are requested by the clients, and a modified object/page buffer to store the updated data returned by the clients [Ghe95].

Pointer Swizzling: Since the hybrid server needs to ma-

nipulate data at both the object and page level it can use the software pointer swizzling mechanism. However, this leads to an increase in the database size, because the software swizzling scheme stores object identifiers which are larger in size than memory pointers that are stored on disk. Therefore, in order to have the flexibility of manipulating data at an object level while storing memory pointers on disk, the hybrid server architecture uses a hybrid pointer swizzling mechanism. In this technique the source object points to an entry in an indirection structure [BP95] that, in turn, contains a pointer to the target object. The disk version of the object stores the memory pointer from the source object to the indirection structure entry. Once the appropriate part of the indirection structure has been faulted into memory, the target object is brought into memory using DBMS software (similar to the software swizzling mechanisms) as part of an object group or a page. Therefore, unlike the hardware swizzling schemes, operating system page faulting support is not used to bring in data objects and this gives hybrid swizzling the flexibility to manipulate data at both object and page levels. Moreover, unlike the hardware swizzling approach, the level of indirection enables the hybrid pointer swizzling mechanism to avoid object migration/deletion problems, and it also allows for fine-granularity object level locking.

Since the hybrid pointer swizzling approach uses the same techniques and data structures that have been developed by the hardware pointer swizzling algorithms for storing and handling memory pointers [WD94], we are not repeating the details here. The indirection structure is an additional data structure (also present in software swizzling mechanism) that is not used by the hardware pointer swizzling mechanism, but it has to be stored and retrieved from a persistent store. As a minimum, each entry in the indirection structure contains the OID of the target object, a pointer to a structure containing run-time information, and also a pointer to the target object. The indirection structure's pages are under the control of the DBMS.

Recovery: Recovery management and *client-to-server* data transfer are tightly linked because the client can either return both pages and log records, or it can return only log records when the page is not present in the client cache. The server stores both the redo and undo portions of a log record in its log buffer, but it stores only the redo portion of a log record in its MOB. If the client returns only log records then the server uses a redo-at-server recovery mechanism, but if the client returns both pages and log records, then the server dynamically switches and uses the ARIES-CSA [MN94] recovery mechanism. The recovery management details are presented in Section 3.2.

Cache Consistency/Concurrency Control: Since the clients in the hybrid server architecture handle both pages and object groups, the hybrid server should be able to support locking operations at both page and object levels. As discussed in a previous client-server locking performance study [CFZ94], it is very inefficient to send lock escalation messages from the client to the server for each individual object in an object group. Since page servers can perform locking operations at the granularity of a page, they do not incur high lock processing and lock escalation message overheads. In this paper, we adapt the page-based AACC [OVU98] algorithm for object servers. The object servers use the physical disk page itself as the granule of locking when dealing with object groups. It is important to note that this locking arrangement still gives each client the freedom to dynamically form object groups for data transfer. Therefore, there can be situations when the object group size is much smaller than the page level unit of locking. If object groups are allowed to span across multiple pages, then one can incrementally lock only those pages whose objects are accessed. Since our unit of locking is a page, even if the size of the object group is smaller than a page and the page does not exist at the client, we still lock the entire page. The server includes the id of the locking unit (page) along with each object that it sends to the client. If there are conflicts at the page level, then the clients dynamically de-escalate to individual object level locks. The clients maintain locking information at both individual object level and at the page level. The server primarily maintains locking information at the page level, but it also maintains information about objects on pages that are accessed in a conflicting manner by different clients. Thus, the hybrid server can lock data at both page and object levels. A detailed description of AACC along with a performance study that compares it with other cache consistency algorithms can be found in [OVU98].

In this paper, due to space constraints, we only provide the details of the data transfer and recovery mechanism.

3.1 Adaptive Hybrid Data Transfer Mechanism

The data transfer mechanism that is used in our server architecture is hybrid because the granularity of data that is sent from the server to the client and subsequently returned from the client to the server can be either pages or groups of objects. Clients provide hints to the server as to whether to send a page or an object group. If the clients want a group of objects, then they also provide a hint about the size of the object group. If the server notices that its buffers are contended, then the server overrides client hints and sends pages to the clients. The server also informs the client whether the server resources are contended. After performing an update, the client returns a page to the server only if a large portion of the page has been updated and the server is busy, otherwise, the client returns updated objects back to the server. We now discuss the specifics of this mechanism in detail.

Initial Client Request: A client's first request is for an object; it sends an object id to the server and requests the corresponding object. It also initializes the object group size to be equal to the page size and sends this to the server along with the object id.

Request Processing at Server: In servicing the request, the server checks if its disks and buffers are contended (the details are given below):

- If its disks and buffers are contended (server is busy), then the server ignores client's object group size hint

and sends back the page on which the requested object resides. In this case, the server does not consider whether the clustering is good or bad. Returning a page during periods of high contention helps the server to reduce the group forming overhead.

- If the server is not busy then it checks the client provided hint to see whether it should return a page or a group of objects back to the client. If the client requested a page, then the server returns the page to the client. If the client requests an object group, then the client specifies the size of the object group. The server partitions the page into *n* equal sized sub-segments whose size is equal to the size of the object group requested by the client [LAC+96]. The server then returns to the client sub-segment in which the requested object resides. The server also determines whether it is busy and piggybacks this information along with other messages.

The server determines that its resources are contended if the disk utilization, and the *page buffer miss ratio* are above their respective thresholds, otherwise it considers itself not busy. Page buffer miss ratio measures the number of server page buffer misses. We empirically determined the disk utilization and the server page buffer miss ratios to be 0.80 and 0.60 respectively. These thresholds were determined by running multiple experiments (not reported in this paper) with different threshold values. For disk utilization values between 0.70 and 0.90, the overall system throughput did not change appreciably. Similarly, the server page buffer miss ratio values between 0.50 and 0.70, the overall system throughput did not change appreciably. Disk utilization needs to be checked because, if the disks are not contended, then the clients can return objects and the installation reads can be performed in the background. Hence, returning objects to the server will not be a problem. Similarly, server page buffer miss ratio is important because if no client read requests are present at the server, then, once again, the installation reads will not be an issue. It is important to note that other heuristics using other system parameters (such as CPU and network utilization) are possible and we are now exploring these.

Client Receives Object Group: If the client receives an object group it registers the objects in the resident object table, and loads the objects in the object buffer. The client determines whether there is a good match between data access and data clustering patterns (i.e, whether the group size is accurate). It determines accuracy by keeping track of the number of objects used in the previously received object groups [LAC+96]. Therefore, if many objects in the previously received object groups have been used, then there is a good match between access and clustering. The object group size is dynamically increased if the data access pattern matches the data clustering pattern, and decreased otherwise. In this paper the object group size varies in increments of 5 objects, where the upper limit of the group size is the number of average objects allowed on a page, and the lower limit is the increment size itself. Each client adjusts the group size using two parameters: *fetch* and *use*. *Fetch* is the number of objects in the object group received by the client, and *use* is the number of objects that have been used for the first time by the client after they are fetched. When an object group of size *N* arrives, the client recalculates these parameter values using exponential forgetting [LAC+96]: *fetch = fetch/2 + N* and *use = use/2 + 1*. The client decreases the object group size by 5 objects if *use/fetch* is less than the threshold (which is empirically determined to be 0.3) and increases it by 5 objects otherwise. This threshold of 0.3 was determined by running multiple experiments. For threshold values between 0.20 and 0.40, the overall system throughput does not change appreciably. Moreover, if the client determines that the group size is over 30 percent of the page size, then it switches to page request mode for subsequent requests from the server in order to reduce the object group forming/breakup costs. We found that switching from object to page mode for group size values between 25 percent and 35 percent of the page size did not change the overall system performance appreciably.

Client Receives Page: If the client receives a page then it registers the page in the resident page table, and puts the page into its page buffer. The page stays in the client page buffer as long as there is no client buffer contention and the page is well clustered. Otherwise, the client flushes the page and retains only the objects that have been already used by moving them to the object buffer [KK94]. Once again the client determines whether the current object group size (now equal to a page) is accurate in the manner described above. The client switches back to requesting objects if the group size falls below 30 percent of the page size.

Client Returning Updated Data: When a client performs an update, it can return either an updated page or an object. If the server is busy, then the client returns an updated page if the page is present in the client buffer and more than 10 percent of the page has been updated (for a range between 5 and 10 percent, the overall system performance did not change appreciably). Otherwise, the client returns updated objects. The clients do not want to return sparsely updated pages when the server buffers are contended.

Server Receiving Updated Data: After receiving the updated objects/page from the client, the server loads them into its modified buffer, and then flushes them to disk in the background.

3.2 Hybrid Server Recovery Algorithm

Since the hybrid server can behave as either a page server or an object server, its recovery mechanism must be flexible to handle both modes of operation. In this paper we describe a recovery algorithm that meets this requirement. We should note that the algorithm can also be used by pure object servers but the details are not presented in this paper.

Our algorithm is based on the ARIES-CSA recovery algorithm [MN94], which we adapt for hybrid servers. We do not present the details of this algorithm, and instead refer the reader to [MN94]. Similar to ARIES-CSA, our recovery algorithm generates log records at the clients which

	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	Hybrid	Page/ Modified Object	Dual
ObjSrv	Objects	Redo At Server Logs	Adaptive AACC	Hybrid	Page/ Modified Object	Object
HybSrv	Page / Objects	ARIES Or Redo at Server Page/Logs	Adaptive AACC	Hybrid	Page/ Modified Dual	Dual

Figure 3: Systems under comparison

are stored persistently at the server (logs are not stored on local client disks), and the server does not rely on clients for its restart recovery. The transaction rollback operations are performed at the clients. Our algorithm uses the STEAL/NO-FORCE buffer management policy at the server. Clients generate log records by comparing the pre-update copy of the data with the post-update copy of the data. The log generation operation is performed at commit time or when data are flushed from the client buffers. Clients can take checkpoints, and the server can take a coordinated checkpoint which contacts all of the clients for their dirty page table and transaction table information. The server and client failure recovery operations use the standard ARIES 3-pass approach.

3.2.1 Hybrid Server Recovery Issues

The following recovery issues have to be addressed when ARIES-CSA recovery algorithm is extended to hybrid servers:

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 every object. After the client receives a group of objects, in addition to creating resident object table (ROT) entries, the client 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. 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 helps to keep track of the pages in the server buffer. Consequently, in addition to the dirty page table, we introduce the notion of a *dirty object table* (DOT) at the server. Each (DOT) entry contains the LSN of both the earliest and the latest (because objects do not contain PageLSN field) log records that correspond to an update on the corresponding object.

Fine-Granularity Locking: In hybrid servers, 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 enough 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 [FZT+92, MN94]. A more recent proposal [PBJR96] 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, the *replacement* log record helps to overcome the problems encountered due to the generation of the same PageLSN value at multiple clients. In our hybrid server solution, we also use the notion of *replacement* log records.

Returning pages or logs to the server: In the hybrid server architecture, clients return either both 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 whereas 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. As per the adaptive data transfer algorithm (Section 3.1), if the client decides to return a page to the server, then it generates a NRDS log record, else it generates a RDS log record. When the client returns an NRDS log record to the server, it ensures that the corresponding page is also returned to the server. The client does not return the corresponding page when it sends a RDS log record to the server.

4 Experiment Setup

We compare our hybrid server (HybSrv) architecture 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 1 and is similar to SHORE [CDF+94]. The hardware-based page server falls under the Page-Page server classification, and is similar to ObjectStore [LLOW91] and BeSS [BP95] 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 [Ver98] and Thor [LAC+96]. The existing hardware page server systems [BP95, LLOW91] employ page level data transfer, concurrency control and buffer management. As a representative of these systems PageHard also adheres to these page level restrictions, and these are the key distinguishing features 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). It is important to note that we are only conducting a performance study on the client-server related issues. The overall performance of a system is also affected by other issues such as query processing, query optimization, indexing and others, which are not considered in this paper. We have tried to incorporate the latest advances in cache consistency, pointer swizzling, buffer management and recovery strategies into all of the systems under comparison in this study (refer to Figure 3), ensuring that they all benefit from the same advantages. Therefore, the systems under comparison in this paper are similar but not the same as their commercial/research counterparts.

4.1 Basic System Model

The baseline setup of this performance study is similar to the previous client-server performance studies [DFMV90, CFZ94, WD94, LAC+96, AGLM95, OVU98], 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 MOB. We configured the client dual buffer in a manner similar to the initial dual buffer study [KK94] where the buffer is configured as best as possible, given the application's profile and the total size of the client's buffer. However, in future we plan to use a dynamically configurable dual buffer [CALM87]. The client and the server CPUs have a high priority and a low priority input queue for managing system and user requests respectively [CFZ94]. 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 80Mbps 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.

Cost Type	Description	Value
Client CPU Speed	Instr rate of client CPU	50 MIPS
Server CPU Speed	Instr rate of server CPU	100 MIPS
ClientBuffSize	Per-Client buffer Size	12% DB Size
ClientLogBuffSize	Per-Client Log buffer	2.5% DB Size
ServerBuffSize	Server Buffer Size	50% DB Size
ServerDisks	Disks at server	4 disks
FetchDiskTime	General disk access time	1600microsecs/Kbyte
InstDiskAccessTime	MOB disk I/O time	1000microsecs/Kbyte
FixNetworkCost	Fixed number of instr. per msg	6000 cycles
VariableNetwork Cost	Instr. per msg byte	4 cycles/byte
Network Bandwidth	Network Bandwidth	80Mbps
DiskSetupCost	CPU cost for performing disk I/O	5000 cycles
CacheLookup/Locking	Lookup time for objects/page	300 cycles
Register/Unregister	Instr. to register/unregister a copy	300 cycles
Page Pointer Handling	Pointer Handling Cost Per Page	40000 cycles
DeadlockDetection	Deadlock detection cost	300 cycles
CopyMergeInstr	Instr. to merge two copies of a page	300cycles/object
Pointer Handling	Pointer handling Cost Per Object	1000 cycles/object
Database Size	Size of the Database	2400 pages
PageSize	Size of a page	4K
Object Size	Size of an object	100 bytes
GroupFormCost	Group FormingCost per Object	100 cycles
NumberClients	Client Workstations	12
Indirection Cost	Ptr indirection Cost per Access	15 cycles

Figure 4: System Parameters

Figure 4 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 [CFZ94, AGLM95]. The pointer handling costs represent the overhead associated with handling the memory pointers stored on disk. We assume that the disks contain the swizzled memory pointers. These costs are similar to the ones used in the hardware pointer swizzling study [WD95]. The schemes using the hybrid swizzling approach assume that the indirection structure is well clustered with respect to the objects accessed by each client in its private region.

Since this indirection structure is not present in the hardware swizzling approach, we compensate it by allocating 10 percent more client buffer space. 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 objects lock group. The group breakup cost consists of the cost of registering each object in the group into the ROT. 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 mechanism is the same across all of the architectures.

4.2 Workload Model

The multi-user OO7 benchmark has been developed to study the performance of object DBMSs [CDN93]. However, this benchmark is inadequate for client-server concurrency control and data transfer studies. Multi-user OO7 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 [CFZ94, AGLM95], and the data clustering notions from the initial data transfer study (ACOB benchmark) [DFMV90]. We obtained the transaction size and length characteristics from the market surveys performed by the commercial ODBMS vendors [Obj98]. 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 clustering factor indicates how closely the data access pattern matches the data placement on the disk. It is desirable to have good clustering (high clustering factor) because it allows one to easily prefetch useful data and thus reduce the disk I/O and network overhead. We use a similar notion of clustering as was used in the initial page server/object server data transfer study [DFMV90]. When creating the database, the clustering factor determines the location of the complex objects connected to a base assembly object. For each complex object, a random number between 0 and 99 is generated, and if the random number is smaller than the clustering factor, the complex object is placed on the same page as its siblings. Otherwise, the complex object is placed on a different page.

In this study we examine *Private* and *Sh-Hotcold* data sharing patterns [CFZ94, AGLM95]. These two sharing patterns are the most common in ODBMS workloads [CFZ94]. 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 regions). The clients can update objects in all of the regions. 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. The transaction think time is the delay between the start of two consecutive transactions at the clients. Figure 5 describes the workload parameters used in this study.

Parameter	Setting
Transaction size Private	800 objects
Transaction size Sh-HotCold	200 objects
Clustering Factor	10 to 90 %
Per Client Region	50 pages
Shared Region	50 pages
Object write probability	2% to 30 %
Read access think cost	50 cycles/byte
Write access think cost	100 cycles/byte
Think time between trans	0

Figure 5: Workload Parameters

5 Results of Experiments

In this section we report the performance comparison of the adaptive hybrid server architecture (HybSrv) with the hardware page server architecture (PageHard), the software page server architecture (PageSoft) and the object server architecture (ObjSrv). All of the experiments use the cost and workload settings as described in Figures 4 and 5. In cases where the default values have been changed, it is explicitly specified. The average response time for a single client in seconds (for 50 transaction interval) is the primary performance metric. Data sharing patterns, server and client buffer sizes, data clustering accuracy, and write probability are the key parameters that are varied.

5.1 Large Client and Large Server Buffers

In this setup, (which we call Large/Large) both the client and the server have large buffers. In a large client buffer the entire working set of the client (all of the objects accessed by a particular client) fits in the client's cache. A large server buffer means that a large portion of the working sets of all the clients fit into the server buffer resulting in low disk utilization upon reaching steady state. Due to these conditions, buffer management is not the performance differentiating factor between different architectures. The client buffer is 12 percent of the database size

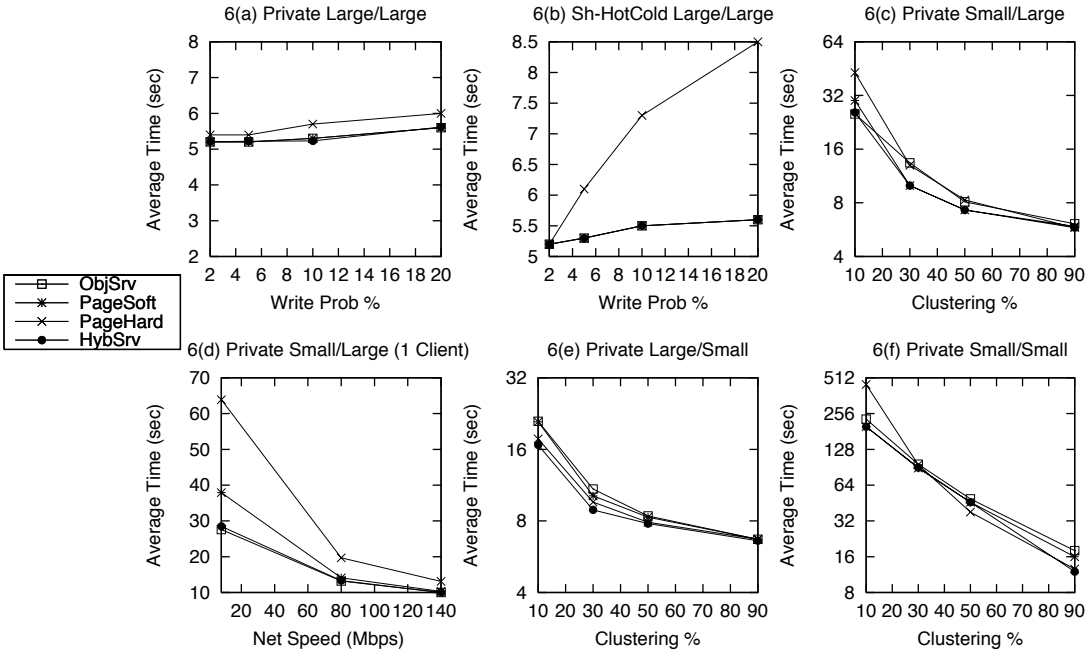

Figure 6: Experiment Results

and the server buffer is 50 percent of the database size and we ran the experiment with 50 percent clustering. In the private workload there is no data contention and, therefore, concurrency control/cache consistency is not an issue. As seen in Figure 6(a), the performance of all of the architectures is quite similar. Our hybrid pointer swizzling mechanism has allowed ObjSrv, PageSoft and HybSrv to successfully compete with PageHard in this workload. Since all the architectures store memory pointers on the disk, pointer swizzling is not an issue. PageSoft, HybSrv and ObjSrv outperform PageHard by a very small margin because they return updated objects instead of updated pages (like PageHard) and thus incur lower network overhead. In the Large/Large buffer case, the server buffer is not heavily contended, therefore, the installation read overhead is minimized for the architectures that return updated objects. For the Large/Large buffer configuration, we also ran the Sh-HotCold workload with the clustering factor fixed at 30 percent (Figure 6(b)). The software page handling architectures outperform PageHard, because they are able to lock data at a finer granularity than PageHard. The hardware page handling mechanisms rely on the operating system provided page protection mechanisms to lock data only at page level. For this experiment we don't present the results for higher clustering percentages, but we noticed that the performance of PageHard improved as the clustering became better because higher page locality leads to fewer page accesses and thus less contention for PageHard. As evident from Figure 6(b), ObjSrv and HybSrv (when it is operating as an object server) are able to successfully compete with PageSoft for the Sh-HotCold workload. This shows that AACC cache consistency algorithm, which was originally designed for page servers, can be successfully extended to object servers.

5.2 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. This is possible if the size of the working set is very large or if the client buffer is shared by multiple applications. The client buffer is 1.5 percent of the database size and the server buffer is still 50 percent of the database size. We ran the private workload with 10 percent write probability. We varied the clustering factor to see the relationship between clustering and client buffer size. As shown in Figure 6(c), PageHard performs worst during low clustering, because it manages the client cache strictly at page level. Therefore, the clients in PageHard continue to cache badly clustered pages which, in turn, leads to low client buffer utilization. However, the clients in ObjSrv, HybSrv and PageSoft only retain useful objects in their cache. The low buffer utilization in PageHard leads to a higher number of cache misses and this, in turn, degrades PageHard's performance. The second important result is that HybSrv and ObjSrv perform better than PageSoft during bad (10 percent) clustering. Even though HybSrv and ObjSrv incur a higher number of client cache misses, they outperform PageSoft because it transfers badly clustered pages from the server to the client and thus, incurs higher network overhead. The third important result is that HybSrv and PageSoft outperform ObjSrv for 30 to 50 percent clustering because in these architectures when a client caches a page it is able to get more cache hits during subsequent accesses to different regions of the page, whereas the object server (due to its grouping algorithm) has to make multiple requests to the server to get the different portions of the page. However, if over time, the client repeatedly accesses only a particular region of the page, then the object server architecture is more competitive due to the better performance of its grouping algorithm. The adaptive nature of HybSrv allows

it to behave as a page server when the clustering is good and like an object server when the clustering is bad and this allows it to be more robust (performance-wise) than page and object servers. We also ran this experiment with slow (8Mbps) and fast (140Mbps) network speeds for a single client. As evident in Figure 6(d), the architectures that send pages from the client to the server suffer even more in the presence of slow networks. This result is important for the popular low bandwidth wireless environments. For the 140 Mbps, we also reduced the software overhead (reduced the variable and fixed network overhead by 50 percent) associated with sending and receiving a message. We found that as the network speed is increased and the software overhead is reduced, the systems that send badly clustered pages are able to close the performance gap.

5.3 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 and it can hold the local working set. It would have been preferable to model the small server buffer case by keeping the server buffer size constant and by increasing the number of clients. However, the memory constraints of our simulator did not allow for this type of modeling. Therefore, by reducing the server buffer size, we are trying to capture the essence of the impact of many clients on the server buffer. We set the client buffer at 12.5 percent of the database size and the server buffer at 3 percent of the database size. We ran the Large/Small experiments for the Private workload configuration. We found that with a write probability of 20 percent, PageHard is beaten by all of the other architectures because PageHard is returning sparsely updated pages to a server whose buffer is highly contended. Therefore, the server consumes valuable buffer space much more quickly (lower server buffer absorption) in PageHard than the schemes that are returning updated objects. However, as we increase the write probability to 30 percent (Figure 6(e)), PageHard and HybSrv are able to beat PageSoft and ObjSrv because PageSoft and ObjSrv return updated objects and the installation read overhead in PageSoft and ObjSrv offsets the gains due to better MOB buffer utilization. Since HybSrv dynamically decides at the client whether to return updated pages or objects, its performance is more robust than the other algorithms.

5.4 Small Client and Small Server Buffers

In Small/Small configuration, the server buffer is small and cannot hold the working sets of the active clients and the client buffer is small and it cannot hold the working set of that particular client. We set the client buffer at 1.5 percent of the database size and the server buffer at 3 percent of the database size. We ran this experiment with 30 percent write probability for Private workload to see whether a small client buffer has any impact on the results presented in the Large/Small buffer case. Figure 6(e) shows that HybSrv, PageSoft and ObjSrv outperform PageHard during low clustering because PageHard has lower client buffer utilization than the other architectures. Moreover, HybSrv and PageSoft outperform ObjSrv because ObjSrv incurs a higher number of client buffer misses due to the object grouping algorithm, and in the Small/Small case (unlike the Small/Large case), a miss in the client buffer also leads to a miss in the server buffer. Since, HybSrv sends pages to the clients when the server is busy, it has similar client buffer hit rate as PageSoft. As the clustering percentage increases, PageHard and HybSrv outperform PageSoft and ObjSrv, because PageHard and HybSrv return updated pages to the server, and, thus, incur fewer installation reads. HybSrv returns pages since it takes into account that the server is busy and a large portion of the page has been updated.

6 Discussion

The integrated performance study has provided us with many interesting insights into ODBMS client-server architectures. A previous client-server recovery study has shown that installation reads can become a problem [WD95] for the redo-at-server recovery mechanisms. However, the combined study of recovery and server buffer management mechanism has shown that the presence of a MOB at the server prevents the degradation of the redo-at-server recovery mechanism during medium-to-low server contention.

Previously [OS94], it was thought that it is better to return an updated page to the server if the page is present at the client. However, we have found that if the server buffer is highly contended, and the page has been sparsely updated, then it is better to return updated objects because this increases the MOB buffer absorption. Another previous study [Ghe95] has shown that is desirable to return updated pages to the server if a large portion of the page has been accessed (high clustering) and updated (high write probability). Our results agree with this study, and we have also found that in addition to clustering and write probability, the server buffer contention level is also a key component which dictates whether it is desirable to return updated pages or objects.

Until now, hardware pointer swizzling systems stored memory pointers on disk to attain good performance [WD94, LLOW91]. However, since the hardware pointer swizzling systems use the operating system provided page handling mechanism, they employ page level locking, data transfer and buffer management mechanisms [LLOW91, WD94]. The combined study of data transfer, pointer swizzling and client buffer management has shown that managing client buffers at strictly page level and always returning updated pages back when the server is busy has a negative impact on the performance of hardware page handling architectures.

Initially, ODBMSs were primarily used by applications, such as computer aided design, which consisted of large transactions with little or no data contention. However, many of emerging application domains such as network management, financial trading and product information management which use ODBMSs use small transactions with read/write locking conflicts [Obj98]. Therefore, hardware page handling architectures that lock data at only page level are increasingly less suitable.

Many techniques we have developed for the hybrid server architecture can be applied to other architectures

(to object servers in particular). Lack of an efficient, low aborting cache consistency algorithm, and the absence of a Steal/No-Force recovery algorithm were considered to be two major drawbacks of object servers. Our adaptation of AACC algorithm for hybrid server eliminates one of the problems. The key insight behind this adaptation is that even though the clients in the object server architecture strictly manipulate objects, and pages do not exist in the client cache, the clients can still lock the pages corresponding to the objects. Therefore, we are essentially decoupling the data transfer and concurrency control mechanism for object servers. Previously, this flexibility was only provided to the page servers [CFZ94]. Our adaptation of ARIES-CSA page server recovery to hybrid servers provides a STEAL/NO-FORCE recovery algorithm which can also be used by object servers. The clients in the object server architecture maintain page level recovery information for the objects in the client cache. This, in turn, allows the object servers to use all of the page server recovery techniques. Thus, object servers are now quite competitive with the page server architecture and these adaptations can be pursued by the existing object server architectures.

7 Conclusion

In this paper we presented a new adaptive hybrid server architecture with more robust performance than page or object servers across a spectrum of system configurations and workloads. Our hybrid server uses a new adaptive data transfer mechanism in which the clients and server pass valuable information to each other to dynamically adapt between sending pages or objects between themselves. The paper describes, in detail, the key points of this adaptive data transfer mechanism. An adaptive data transfer method has an impact on data transfer, cache consistency/concurrency control, recovery, and buffer management mechanisms. Each of these system components need to be able to support both page and object server architectures while continuing to operate in an integrated manner. Due to the absence of efficient cache consistency and recovery algorithms for object servers, we have adapted the leading page server cache consistency [OVU98] and recovery [MN94] algorithms for object servers. These adaptations are important in their own right since they can be used by existing object sever architectures. We have also shown that it is possible for architectures to both store swizzled memory pointers on disk while maintaining the flexibility to manipulate data at the clients at object level. In addition to characterizing the behavior of the hybrid server proposal, the performance study and its results that are reported are important for a number of reasons. First of all, this study improves our understanding of the client-server architectures, in particular, it shows that with our new recovery, pointer swizzling and cache consistency adaptations, object servers can compete successfully with page servers. Thus, the belief that object servers are not scalable is no longer valid. Finally, we are now investigating alternative adaptive data transfer heuristics which take varying object size, page size, CPU and network contentions into account.

References

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

[BP95] A. Biliris and E. Panagos. A High Performance Configurable Storage Manager. In *Proceedings of 11th International Conference on Data Engineering*, 1995.

[CALM87] M. Castro, A. Adya, B. Liskov, and Andrew Myers. HAC:Hybrid Adaptive Caching for Distributed Storage Systems. In *Proceedings of ACM Symposium on Operating System Principles*, 1987.

[CDF+94] Michael Carey, D. DeWitt, M. Franklin, N. Hall, and et al. Shoring Up Persistent Applications. In *Proceedings of ACM SIGMOD Conference*, 1994.

[CDN93] M. Carey, D. DeWitt, and J. Naughton. The OO7 Benchmark. In *Proceedings of ACM SIGMOD Conference*, 1993.

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

[DFB+96] S. Dar, M. Franklin, B.T.Jonsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *Proceedings of VLDB Conference*, 1996.

[DFMV90] D. DeWitt, P. Futtersack, D. Maier, and F. Velez. A study of three alternative workstation-server architectures for OODBS. In *Proceedings of VLDB Conference*, 1990.

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

[FCL96] M. Franklin, M. Carey, and M. Livny. Transactional Client-Server Cache Consistency: Alternatives and Performance. *ACM Transactions on Database Systems*, 22(4), 1996.

[FZT+92] M. Franklin, M. Zwilling, C.K. Tan, M. Carey, and D. DeWitt. Crash Recovery in Client-Server EXODUS. In *Proceedings of ACM SIGMOD Conference*, 1992.

[Ghe95] S. Ghemawat. *The Modified Object Buffer: A Storage Management Technique for Object-Oriented Databases*. PhD thesis, MIT, 1995.

[GK94] C. Gerlhof and A. Kemper. A Multi-Threaded Architecture for Prefetching in Object Bases. In *Proceedings of EDBT Conference*, 1994.

[KGBW90] W. Kim, J. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE TKDE*, 2(1), 1990.

[KJF96] D. Kossmann, B.T. Jonsson, and M. Franklin. A Study of Query Execution Strategies for Client-Server Database Systems. In *Proceedings of ACM SIGMOD Conference*, 1996.

[KK94] A. Kemper and D. Kossmann. Dual-Buffering Strategies in Object Bases. In *Proceedings of VLDB Conference*, 1994.

[LAC+96] B. Liskov, A. Adya, M. Castro, M. Day, and et al. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proceedings of ACM SIGMOD Conference*, 1996.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10), 1991.

[MN94] C. Mohan and I. Narang. ARIES/CSA: A Method for Database Recovery in Client-Server Architectures. In *Proceedings of ACM SIGMOD Conference*, 1994.

[Obj98] Objectivity. White Paper: Choosing an Object Database. In *www.objectivity.com/ObjectDatabase/WP/Choosing/Choosing.html*, 1998.

[OS94] J. O'Toole and L. Shrira. Hybrid caching for large scale object systems. In *Proceedings of Workshop on Persistent Object Systems*, 1994.

[OVU98] M.T. Ozsu, K. Voruganti, and R. Unrau. An Asynchronous Avoidance-based Cache Consistency Algorithm for Client Caching DBMSs. In *Proceedings of VLDB Conference*, 1998.

[PBJR96] E. Panagos, A. Biliris, H. Jagadish, and R. Rastogi. Fine-granularity Locking and Client-Based Logging for Distributed Architectures. In *Proceedings of EDBT Conference*, 1996.

[TN92] M. Tsangaris and J. Naughton. On the performance of object clustering techniques. In *Proceedings of ACM SIGMOD Conference*, 1992.

[Ver98] Versant. ODBMS. In *http://www.versant.com*, 1998.

[WD94] S. White and D. DeWitt. QuickStore: A high performance mapped object store. In *Proceedings of ACM SIGMOD Conference*, 1994.

[WD95] S. White and D. DeWitt. Implementing Crash Recovery in QuickStore: A Performance Study. In *Proceedings of ACM SIGMOD Conference*, 1995.