Fine-Grained Sharing in a Page Server OODBMS

Michael J. Carey
Michael J. Franklin
Markos Zaharioudakis

Technical Report #1224

April 1994

Fine-Grained Sharing in a Page Server OODBMS*

Michael J. Carey
Computer Sciences Department
University of Wisconsin
Madison, WI 53706
carey@cs.wisc.edu

Michael J. Franklin
Dept. of Computer Science
University of Maryland
College Park, MD 20742
franklin@cs.umd.edu

Markos Zaharioudakis
Computer Sciences Department
University of Wisconsin
Madison WI 53706
markos@cs.wisc.edu

Abstract

For reasons of simplicity and communication efficiency, a number of existing object-oriented database management systems are based on page server architectures; data pages are their minimum unit of transfer and client caching. Despite their efficiency, page servers are often criticized as being too restrictive when it comes to concurrency, as existing systems use pages as the minimum locking unit as well. In this paper we show how to support object-level locking in a page server context. Several approaches are described, including an adaptive granularity approach that uses page-level locking for most pages but switches to object-level locking when finer-grained sharing is demanded. We study the performance of these approaches, comparing them to both a pure page server and a pure object server. For the range of workloads that we have examined, our results indicate that a page server is clearly preferable to an object server. Moreover, the adaptive page server is shown to provide very good performance, generally outperforming the pure page server, the pure object server, and the other alternatives as well.

1 Introduction

Recent years have seen dramatic research and development progress in the area of object-oriented database management systems (OODBMS). There are now a number of commercial offerings in this area, and these systems are beginning to gain real acceptance for certain classes of commercial applications (e.g., CAD/CAM and CASE). This emerging generation of DBMSs is being deployed primarily in distributed, workstation/server-based environments. In contrast to traditional relational systems, distributed OODBMS architectures are typically based on a *data-shipping* approach. Data items are shipped from servers to clients so that query processing (as well as application processing) can be performed at the client workstations, providing two advantages. First, OODBMSs usually offer programmatic interfaces that support navigation through complex persistent data structures. Data-shipping moves the data closer to the applications, allowing efficient fine-grained interaction between the application and the DBMS. Second, data-shipping offloads DBMS function from the server to the client workstations, enabling the plentiful and relatively inexpensive resources of the workstations to be exploited.

*This research was sponsored in part by the Advanced Research Projects Agency, ARPA order number 018 (formerly 8230), monitored by the U.S. Army Research Laboratory under contract DAAB07-91-C-Q518.

1.1 OODBMS Architectures

OODBMS applications phrase their requests logically, in terms of objects. The objects themselves, however, are stored on disk in units of pages; therefore, an OODBMS must (at some level) manage storage in terms of data pages. The data shipping approach does not dictate where in the system the mapping between objects and pages resides. As a result, existing OODBMSs differ in the granularity at which their clients and servers interact. There are two basic approaches: *page servers*, where clients and servers interact using physical units of data (e.g., individual pages or groups of pages), and *object servers*, where client/server interactions involve logical units of data (e.g., individual objects). The choice between these two approaches has a significant impact on the design of many OODBMS functions, including concurrency control, crash recovery, and query processing.

Examples of both page and object servers can be found among current OODBMSs. Current systems that are based on page server architectures include EXODUS, GemStone, O2, ObjectStore, and SHORE. Systems based on the object server approach include Itasca (a commercial version of the MCC ORION prototype) and Versant. The Ontos system allows application programmers to choose between the object and page server approach statically, on a per-collection basis. Finally, Objectivity is implemented on top of the NFS remote file access protocol – it can be viewed as a page server that uses NFS as its page transport mechanism. This wide variety of approaches is due, at least in part, to the lack of a common understanding of the performance tradeoffs between page servers and object servers. While benchmarks for OODBMS have been developed (e.g., 001 [Catt92] and 007 [Care93]), they do not isolate the effect of the transfer granularity from the effects of other system implementation decisions.

In addition to transferring data efficiently, OODBMS try to improve performance by reducing the need to obtain data from the server in the first place. In order to minimize communication and server disk accesses, both page servers and object servers typically allow clients to cache data items in their local memories once the items have been obtained from the server. To more fully exploit the use of client memory, some systems support *intertransaction caching*, where clients are permitted to retain their cache contents even across transaction boundaries. Intertransaction caching requires the use of a *cache consistency maintenance* protocol to ensure that all clients see a consistent (serializable) view of the database. Despite the potential overhead associated with such a protocol, intertransaction caching has been shown to offer significant performance gains [Wilk90, Care91, Wang91, Fran92a, Fran93]. As for data transfers, the granularity at

which cache consistency is maintained can have a large impact on the performance of an OODBMS; this is another area where the tradeoffs are not yet well-understood.

1.2 Our Focus

Given the lack of concrete data on the tradeoffs between object and page servers, the debate has been largely based on qualitative arguments. One commonly voiced concern about page servers is the potential for increased data contention due to the use of page-level cache consistency protocols and the difficulty of implementing fine-grained concurrency control in the context of a page server (e.g., [DeWi90, Catt91]).

In this paper we study the performance implications of ODDBMS granularity in a multiple client context. While virtually all existing page servers support cache consistency at only a page (or coarser) granularity, we demonstrate that this restriction is neither necessary nor desirable. We describe three approaches for extending a page server OODBMS to support fine-grained cache consistency maintenance, studying their relative performance and comparing them to both a basic object server and a basic page server. One of the three approaches simply extends a page server to perform static, object-level cache consistency maintenance, while the other two are adaptive in nature.

The remainder of this paper is structured as follows: Section 2 surveys related work. Section 3 discusses granularity issues in data-shipping OODBMS in more detail and describes five alternative approaches to OODBMS granularity. Section 4 describes a simulation model, and Section 5 uses the model to analyze the performance of the five alternatives. Section 6 covers advanced issues. Finally, Section 7 summarizes our results and discusses future work.

2 Related Work

As implied in the Introduction, the current state of knowledge regarding the performance of alternative OODBMS architectures is rather primitive. The only previous study that has attempted to answer some of the relevant questions is [DeWi90]. That study compared the performance of an object server and a page server (and also an NFS-based file server), but only in the case of a single client and a single server. The main conclusion was that page servers perform well when access to multiple objects on a page is likely (i.e., when data is well-clustered), whereas object servers are preferable when clustering on pages is poor or client memory is scarce — due to communications and client memory savings gained by shipping only requested objects to clients. In general, the performance of the page server was somewhat more robust because there is only a small incremental CPU cost for sending a message containing a page of objects compared to that for sending only a single object. While this study was informative as far as it went, multi-user issues such as concurrency control, cache consistency, and potential resource bottlenecks were not investigated.

Several papers related to granularity in the area of multiprocessor shared disk (or "data sharing") systems are relevant to our work. The most relevant, which inspired the best of the algorithms that we will be proposing, is the design of an adaptive locking algorithm for Rdb/VMS for use on

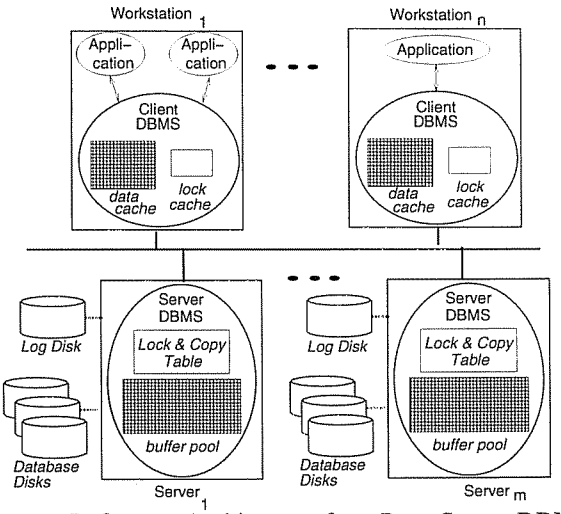

Figure 1: Reference Architecture for a Page Server DBMS

VAXClusters [Josh91]. This algorithm uses a technique called *lock de-escalation* and works in the context of a hierarchy of lock granularities [Gray79]. The basic idea was to obtain locks at the coarsest granularity (e.g., file) for which no concurrency control conflict exists in order to minimize interaction with the VMS distributed lock manager; however, to avoid undue data contention, locks can later be "de-escalated" into finer-grained locks (e.g. pages or records) if subsequent conflicts arise. Independently, an almost identical notion of lock de-escalation was proposed for use in the context of main memory database systems [Lehm89]. While [Josh91] presented some brief preliminary performance results, neither of these papers examined the performance of lock de-escalation in any detail.

Related work on shared disk systems has also been done at IBM Almaden [Moha91]. This paper proposed using strict two-phase locking on *objects* to ensure serializability, while using physical locks on *pages* to ensure cache consistency. These physical locks can either be released during a transaction or they can be held across transactions. The basic idea is to allow a given page to be held in shared (read-only) mode simultaneously at multiple sites, but to ensure that only one site at a time can gain update access to the page. To update a page, a site is required to become the "owner" of that page; when a different site wishes to update the page, it must be sent the most recent copy of the page by the current owner. This approach was designed to exploit the relatively inexpensive inter-node communication paths usually found in tightly-coupled data sharing architectures, and several alternative algorithms (differing in their crash recovery implications) were proposed. The algorithms were ordered based on their designers' performance expectations [Moha91], but no performance analysis was attempted.

3 Alternative Approaches

The general architecture of a data-shipping OODBMS is shown in Figure 1; it consists of two types of processes that communicate via a local area network. Each client workstation runs a Client DBMS process which provides access to the database for the application(s) running at its local workstation.

Applications make requests of their local Client DBMS process, which executes the request, sometimes by sending requests for transaction support or for data items to a Server DBMS process. Server DBMS processes are the actual owners of data; they are ultimately responsible for preserving the integrity of the data and for enforcing transaction semantics. These processes manage the stable storage on which the permanent version of the database and the log reside; they also provide locking and copy management for the data that they own.

At a conceptual level, the database consists of *objects*, while at a physical level, the database is divided into fixed-length (on the order of 4K or 8K bytes) *pages* A page is the minimum granularity of data that can be transferred between memory and disk. In contrast to pages, objects are a logical unit; their size is independent of the physical page size. In general, multiple objects can reside in a single page or a single object can span multiple pages. The Client DBMS process at a workstation is responsible for accepting local application requests for objects and for bringing the relevant data items (either objects or pages) into memory at the client. As a result, all of the data items required by the application are ultimately brought from the server(s) to the clients. As described earlier, intertransaction caching can be used to permit clients to retain the data items that they receive from the server across transaction boundaries. Server DBMS processes are responsible for providing the most recent committed values for the data items that they own in response to client requests; of course, due to concurrency control conflicts, it may not be possible for a server to provide the requested items immediately.

3.1 Granularity Issues

The issue of data granularity arises in many aspects of an OODBMS. In this paper, we will focus on the choice of an appropriate data granularity for three system functions:

1. *Client-server data transfer* - The data items that clients request from servers can be either objects (for object servers) or pages (for page servers). If pages are used, then the server can be simplified, as clients can be made responsible for mapping objects to pages.

2. *Concurrency control (locking)* - To support multi-client (i.e., multi-user) access to data, the DBMS must enforce serializability for transactions. Concurrency conflicts can be detected at various granularities. Locking at a coarse granularity usually involves less overhead, but it raises the potential for conflicts due to *false sharing*, i.e., conflicts arising at a coarse granularity due to concurrent access to distinct, but co-located, finer-grained items.

3. *Replica management (callbacks)* - Intertransaction caching allows multiple copies of data items to reside in different client caches. As a result, cache consistency maintenance requires replica management in addition to concurrency control. The granularity at which replication is managed is therefore a third parameter in the design of a data-shipping OODBMS.

The choices of granularity for these three functions are orthogonal. That is, it is possible to devise working algorithms involving the use of objects or pages in each case, though care must be taken to choose combinations that can be implemented efficiently. In this paper we argue that existing systems – which typically use their data transfer granularity (be it objects or pages) as the finest granularity for concurrency control and replica management as well – have really been *too* careful in this regard. We describe and analyze the performance of several approaches where this restriction is not made. Furthermore, we show that significant performance gains can be obtained by allowing these granularity decisions to be made dynamically, in an adaptive fashion, rather than statically.

In the remainder of this section, we describe five different approaches for treating granularity in an OODBMS. First, we present two "basic" approaches in which the granularity chosen for all three functions is the same. The remaining techniques operate at multiple granularities. All five approaches are extensions of a pessimistic, locking-based cache consistency protocol known as *Callback Locking* [Howa88, Lamb91]. Transactional Callback Locking algorithms have previously been shown to have good performance over a wide range of system configurations and workload characteristics [Wang91, Fran92a] and are used by several OODBMSs, including ObjectStore and SHORE.

In the approaches described here, we assume an underlying steal/no-force recovery scheme based on write-ahead logging and a purge-pages-at-client, undo-at-server approach to handling transaction aborts (a la [Fran92b]). When a transaction commits, copies of all updated data pages that are still in the client's cache are sent back to the server that owns the data; this simplifies the server's job of ensuring durability for committed updates. However, the server is not required to force the updated pages to disk. To simplify the descriptions of the approaches, we will assume that objects are smaller than a page, as large objects (i.e., those that span multiple pages) can be handled in a page-at-a-time fashion, as is done in EXODUS and other systems. We will also initially assume that updates do not change the sizes of objects, deferring the discussion of size-changing updates until Section 6. Finally, our descriptions assume a system with a single server, and the algorithms are presented as though a given client workstation can have only one transaction active at a time. Extensions to multiple servers with partitioned data are straightforward, and local lock management can be used to allow multiple transactions on a client to safely share a common cache.

3.2 Basic Approaches

In this section we describe two basic approaches where the same, statically-chosen granularity is used for data transfer, for concurrency control, and for replica management. These approaches correspond to the traditional notions of a page server and an object server, respectively.

3.2.1 Page Server (PS)

The first approach that we discuss is the basic page server (PS), which transfers pages between clients and servers. The PS approach uses a Callback Locking algorithm to maintain cache consistency; it uses the page-level Callback-Read (CB-Read) algorithm studied in [Fran92a, Fran93]. CB-Read guarantees

that copies of pages in client caches are always valid, so client transactions can safely read-lock and read cached pages without server intervention. When a client needs access to a page that is not resident in its cache, it sends a request for the page to the server. If no other client is holding a write lock on the page, then the server returns a copy of the page to the requestor; otherwise, the server waits until the conflicting lock is released before sending the page. The server manages write locks and keeps track of the locations of cached pages throughout the system, while read locks are managed by the clients.[1]

In order to update a page, a client must first obtain a write lock from the server. When a write lock request arrives for a page that is not locked at the server, the server issues *callback* requests to all sites (except the requester) that hold a cached copy of the page. At a client, such a callback request is treated as a request for an exclusive lock on the specified page. If the callback request cannot be granted immediately (due to a local lock conflict with an active transaction), the client responds to the server saying that the page is currently in use. When the callback request gains exclusive access to the page, the page is purged from the client's cache and an acknowledgement is sent to the server. Once all callbacks have been acknowledged to the server, the server registers a write lock on the page for the requesting client and informs the client that its write lock request has been granted. Subsequent read or write requests for the page by transactions from other clients will then block at the server until the write lock is released by the holding transaction. At the end of the transaction, the client ensures that copies of all updated pages exist at the server and then releases its write locks, retaining copies of all cached pages (and thus implicit permission to read those pages).

3.2.2 Object Server (OS)

The object server (OS) approach is analogous to PS, except that it performs data transfer, concurrency control, and replica management all at an object granularity rather than at a page granularity. The basic object server is the most obvious approach towards avoiding the potential communication, memory usage, and false sharing problems of the coarser-grained PS approach. This approach is similar to the object server that was studied in [DeWi90], although consistency issues were not explicitly considered there.

3.3 Hybrid Approaches

In contrast to PS and OS, the remaining approaches all remove the restriction of having to statically choose a single granularity to be used for all granularity decisions. In this section, we describe three modifications of the basic PS approach. We chose a page server over an object server as the basis for these modifications based on efficiency and simplicity considerations (this decision is revisited in Section 6). The three proposed approaches are each based on adding one or both of the following extensions to PS:

1. *Fine Granularity* - Although pages are the granularity of data transfer, concurrency control and/or replica management can be performed at a finer (in this case object) granularity.

2. *Adaptive Granularity* - In some cases, the granularity at which an action (either locking or callback) is performed can be determined dynamically.

The first extension, fine-granularity, aims to mitigate the potential inefficiencies due to false sharing in a pure page server while retaining the communication efficiency associated with shipping pages rather than objects. However, as has been noted elsewhere, the maintenance of cache consistency at a granularity finer than the transfer granularity increases the complexity of maintaining transaction semantics. For example, as discussed in [DeWi90], if multiple clients simultaneously have permission to update different objects on the same page, then the resulting two copies of the page must be carefully *merged* to avoid losing one of the updates. A way to avoid this problem is to disallow simultaneous updates by using a single *write token* per page [Li89], as is proposed in [Moha91]. The write token approach can be communication-intensive, however — the entire page must often be sent when the write token is transferred from one client to the other. For this reason, we have chosen to merge updates in the approaches studied here. Merging is a relatively simple process if objects do not change size; size-changing updates can also be handled and will be addressed in Section 6.

The second extension, adaptive granularity adjustment, aims to avoid the potentially high communication costs that fine-grained cache consistency maintenance can involve. In a low contention environment, the use of object level locking or callbacks can greatly increase the number of messages required to maintain cache consistency (as compared to the basic PS approach) with no accompanying benefit. By allowing the granularity to adapt to the current level of contention, we hope to enable the system to adapt appropriately to different workloads.

The following subsections describe our hybrid approaches. Because they are based on the PS architecture, we denote them with acronyms of the form PS-*xy*, where *x* indicates the granularity used for concurrency control and *y* indicates the granularity used for replica management. An "O" is used to denote a static object granularity, and an "A" indicates that the granularity is determined adaptively. Thus, the three hybrid approaches are (in order of increasing dynamism): PS-OO, PS-OA, and PS-AA. In all three approaches, while finer granularity techniques are used for concurrency control and/or callbacks, client cache management still takes place at the page level.

3.3.1 Object Locking w/Object Callbacks (PS-OO)

The first hybrid approach is basic PS extended with static object locking and object callbacks. This scheme attempts to combine the communication advantages of page transfers with the increased concurrency allowed by object-level locking and callbacks. In PS-OO, objects that are cached at a client can be read by that client without server intervention. When a requested object is not cached locally, the client determines the

[1] Read locks for pages that have been accessed by an active client transaction and then dropped from that client's cache are maintained at the server as well as at the client; such read locks are recorded at the server before it deletes the relevant page copy from its table of cached copy locations.

page in which the requested object resides, and requests that page from the server. The server reads the page from disk (if necessary) and ensures that no other clients have a write lock on the requested object. Before sending the page to the client, it marks any objects in the page that are write-locked by transactions running at other clients as "unavailable". The server then sends the page to the requesting client.

When the client receives a page from the server, it places the page in its cache. If the client already has the page cached, however, and there are uncommitted updates on the page, it first merges the incoming page with its own copy of the page (being sure not to overwrite the objects that have already been updated). All of the objects on the page, except for those marked as unavailable, are then considered to be cache-resident at the client. When the client wishes to update an object, it sends a request for a write lock on the object to the server. This request is handled similarly to a write request in OS, namely, object-level callbacks are sent to remote clients that have cached copies of the object. In this case, however, a client responding to a callback marks the object as "unavailable" rather than purging it from the cache. The callback therefore does not affect the availability of other objects on the page.[2] As in the basic OS approach, the server maintains information on the location of cached copies at an object granularity.

3.3.2 Object Locking w/Adaptive Callbacks (PS-OA)

PS-OA, our second hybrid approach, uses the same locking protocol as PS-OO, but differs in how callbacks are handled.[3] Due to its object-level callbacks, PS-OO is susceptible to the following inefficiency: Suppose that client A reads a given page P once, and that P then remains in its cache unused. If a different client, B, wishes to later update multiple objects on page P, a separate callback request will be sent to client A for *each individual object* that B wishes to update. Client A will have to service all of these callbacks, even though none of the objects on page P are now being used by client A. PS-OA attempts to avoid this problem by performing callbacks in an adaptive (de-escalating) manner.

At the server, information on the location of copies is kept by PS-OA at a page granularity (rather than at an object granularity like PS-OO). When the server receives a write lock request for an object on a given page, it sends callbacks to all other clients that have copies of that page cached. When a client receives a callback request, it checks to see if any of the objects on that page are currently locked by an executing transaction. If so, the callback behaves like a callback request in PS-OO, and only the specific object is called back. However, if none of the other objects on the page are in use, then the entire page is purged from the cache instead. The server is informed of the action taken by the client that received the callback, and it updates its copy information accordingly.

[2]Transaction aborts are handled similarly, with affected objects being marked as "unavailable" in the aborting transaction's client cache if the page cannot be purged due to other active transactions at that client.

[3]A related hybrid approach, based on hardware support for locking of "minipages," was independently proposed in [Chu94]. Write-write conflicts in that proposal are always handled at the page level, however.

3.3.3 Adaptive Locking w/Adaptive Callbacks (PS-AA)

The final approach that we describe is PS-AA, which performs both locking and callbacks in an adaptive fashion. PS-AA behaves like the basic PS scheme in the absence of data conflicts, de-escalating to finer-grained operation only for data items on which conflicts arise. Furthermore, PS-AA has the ability to "re-escalate" if it determines that the contention that caused a de-escalation has dissipated. In PS-AA, clients record their reads and their writes locally at both a page and an object granularity, essentially obtaining local locks (i.e., at the client) at both levels. The server tracks cached copies of data at a page granularity, as in PS-OA. When a client wishes to read an object that is not present (or is marked as "unavailable") in its cache, it sends a read request to the server. When the server receives an object read request, it checks for conflicts at the object and/or page granularity. Three cases can arise:

No Conflict - If no other client has a write lock on the page, and no other client has an object-level write lock on the requested object, then PS-AA behaves like the other two hybrid PS algorithms. That is, the server returns the entire page to the client with any objects that are write-locked by other clients being marked as "unavailable".

Object-level Conflict - If another client holds an object-level write lock on the requested object, then the read request is blocked until the other client terminates.

Page-level Conflict - Lastly, if the read request conflicts with a *page-level* write lock held by another client, then the client holding the page-level write lock is asked to de-escalate its lock. To do so, it obtains object-level write locks at the server for any objects that it has updated while holding the page write lock. After de-escalation is done, the server checks for object-level conflicts and then proceeds according to one of the two cases described above.

When a client wishes to update an object for which it does not have write permission (i.e., for which it has neither a write lock on the object nor on its containing page), it sends a write request to the server. Initially, the write request is served as in PS-OA – if no object-level write-write conflict is found, then callbacks are sent to clients caching the page (if any). At each client, the callback will invalidate the entire cached page if possible. After the server has collected the callback acknowledgements, there are two possible outcomes: If the page has been successfully invalidated everywhere (i.e. nobody was using it), then a page write lock is granted to the requestor. Otherwise, if one or more clients were using the page, then no page write lock is possible and the requestor is granted permission only to update the specific object.

4 Page-Server OODBMS Model

To study the performance of the alternatives described in Section 3, we extended an existing page server OODBMS simulator to model transactions at the level of object (rather than page) references and processing. We also developed a variant of the simulator that models an object server rather than a page server. In this section, we briefly describe these extended models; readers interested in more information about the basic model can find more complete descriptions in [Care91, Fran93].

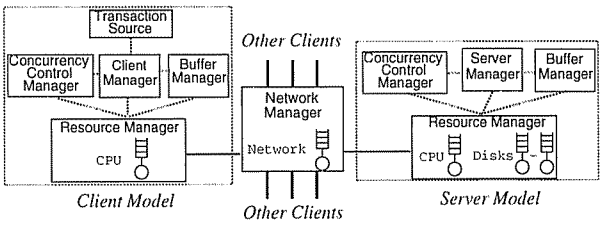

Figure 2: Model of a Page-Server OODBMS

4.1 System Model

The structure of the page server OODBMS model, which we built using the DeNet simulation language [Livn88], is depicted in Figure 2. We model a system consisting of a single server plus a varying number of client workstations, all of which are connected via a local area network. The number of client machines in the system is a parameter of the model. The model for each client node consists of a *Buffer Manager*, which manages the contents of the client buffer pool (i.e., client cache) using an LRU page replacement policy, a *Concurrency Control Manager*, which performs locking and consistency management functions, a *Resource Manager*, which models CPU activity and provides access to the network, and a *Client Manager*, which coordinates the execution of client transactions. Each client also has a module called the *Transaction Source* that submits transactions to the client workstation one after another; upon completion of one transaction, the source waits for a specified thinking period and then submits a new transaction. Client transactions themselves are each modeled as a string of object references (i.e., object reads and writes). When a client transaction aborts, it is resubmitted with the same object reference string.

The server model is somewhat different than that of the clients. One difference is that its Concurrency Control Manager, in addition to managing lock information, stores information about the location of cached data copies (either pages or objects, depending upon the approach). Another difference is that there is no Transaction Source module; work for the server always arrives via the network. A third difference has to do with the server's *Resource Manager*, which must model disk activity as well as CPU activity and network access. Lastly, it has a *Server Manager* component that coordinates server operation based on the stream of incoming client requests.

Table 1 shows the parameters for specifying the resources and overheads of the system and the settings that will be used for most of this study. The simulated CPUs of the system are managed using a two-level priority scheme. System CPU requests, such as lock, message, and I/O initiation, are given priority over user-level requests and are scheduled using a FIFO queueing discipline; user requests are managed using processor-sharing. Each disk has a FIFO queue of I/O requests, and the disk for each request is chosen uniformly from among all of the server's disks. Disk access times are drawn from a uniform distribution between a specified minimum and maximum. The simulator's *Network Manager* component is very simple, consisting of a FIFO server with a specified bandwidth, as protocol processing (i.e., CPU overhead) dominates the on-the-wire time for messages in modern local area networks. The CPU cost to send or receive a message via the network is modeled as a fixed per-message instruction count plus an additional per-byte instruction increment. In addition to the main settings shown in Table 1, we have also varied many of the model's size parameters (for example, scaling the database and buffers by an order of magnitude) in order to gain additional insight into our results and to verify that our performance results scale. These results are discussed where appropriate in Section 5.

To extend the page server model of [Care91] to model object-level processing, two parameters were added. The first new parameter is *ObjectsPerPage*, which specifies the number of objects per page in the database. The other new parameter is *CopyMergeInst*, which specifies the per-different-object cost incurred to merge two copies of a page that differ due to concurrent updates to some of their objects. In developing the object server variant of the model, the client buffer manager was modified to manage an LRU cache of objects rather than pages, and the client and server models were modified to exchange objects rather than pages when data transfers occur. The parameters of Table 1 still apply in the object server case; the capacity of the client cache (in objects) in this case is *ClientBufSize* pages times *ObjectsPerPage* objects per page.

4.2 Workload Model

Our simulation model provides a simple yet flexible mechanism for describing OODBMS workloads. The access pattern for each client workstation can be individually specified using the parameters shown in Table 2. The size of the transactions submitted by a given client is controlled by two parameters: *TransSize*, which specifies the average number of pages accessed by a transaction, and *PageLocality*, which specifies a range of values for the number of objects to be accessed per page by a transaction. In addition, transactions can have an *AccessPattern* that is either clustered (in which case all of the referenced objects on any given page will be referenced together) or unclustered (in which case references to objects on different pages may be interleaved). To model different sharing patterns, two ranges of database pages can be specified for each client: a hot range and a cold range. The probability of a given page access being directed to a page in the hot range is explicitly specified; other accesses go to cold range pages. For both ranges, the probability that an object read access within the range will lead to an update of the object is also specified. The workload parameters also include the average CPU cost for processing an object at the client once the proper lock has been obtained, and this cost is doubled for write accesses.

Table 2 summarizes the workload parameters to be used throughout most of this study. Other values for these parameters, particularly transaction size and page locality, have also been explored and will be discussed briefly later. The data sharing patterns inherent in the workloads of Table 2 were chosen due to their previously established effectiveness as performance discriminators for client caching alternatives [Care91, Fran93]. The HOTCOLD workload has a high degree of locality per client and a moderate amount of sharing and data contention among clients. The UNIFORM workload

ClientCPU	Instruction rate of client CPU	15 MIPS
ServerCPU	Instruction rate of server CPU	30 MIPS
ClientBufSize	Per-client buffer size	25% of DB size
ServerBufSize	Server buffer size	50% of DB size
ServerDisks	Number of disks at server	2 disks
MinDiskTime	Minimum disk access time	10 milliseconds
MaxDiskTime	Maximum disk access time	30 milliseconds
NetworkBandwidth	Network bandwidth	80 Mbits per second
NumClients	Number of client workstations	10
PageSize	Size of a page	4,096 bytes
DatabaseSize	Size of database in pages	1250 (5 MB)
ObjectsPerPage	Number of objects per page	20 objects
FixedMsgInst	Fixed number of instructions per message	20,000 instructions
PerByteMsgInst	Number of additional instructions per message byte	10,000 per 4KB page
ControlMsgSize	Size in bytes of a control message	256 bytes
LockInst	Number of instructions per lock/unlock pair	300 instructions
RegisterCopyInst	Number of inst. to register/unregister a copy	300 instructions
DiskOverheadInst	CPU cost for performing a disk I/O	5000 instructions
CopyMergeInst	Number of instructions to merge two copies of a page	300 instructions per object

Table 1: System and Overhead Parameters

is a low-locality workload with no particular per-client data affinity; its level of inter-client data contention is therefore somewhat higher than in the HOTCOLD workload. The HICON workload is similar to the skewed workloads that are often used to study shared-disk transaction processing systems, and it is used here to expose the performance tradeoffs that would be seen in the (unlikely) event of very high OODBMS data contention. Finally, PRIVATE is a CAD-like workload with no data contention whatsoever; the only inter-client sharing in the workload involves read-only data.

As indicated in Table 2, our study will center primarily around two different settings for the transaction size and page locality parameters. We will use a transaction size setting of 30 pages together with a page locality range of 1-7 (averaging 4) objects, and we will also use a transaction size setting of 10 pages with a page locality range of 8-16 (averaging 12) objects. Both settings yield overall average transaction lengths of 120 objects. By varying these two parameters together in this way for each workload, we can study the impact of a higher or lower page locality while keeping the average transaction length constant in terms of the number of objects accessed.[4]

5 Experiments and Results

In this section, our OODBMS simulation model is used to explore the relative performance of the five main alternatives covered in Section 3, all of which use the callback-read paradigm for caching data on clients across transactions. Briefly, the five alternatives include the basic object server (OS), the basic page server (PS), and the three schemes for fine-grained sharing in a page server – one that does both locking and callbacks at the object level (PS-OO), one that does object-level locking but adaptively switches to page-level callbacks when it can (PS-OA), and one that adaptively switches from object-level to page-level operation for doing locking as well as callbacks (PS-AA) whenever possible.

[4]Due to the size of the client hot regions in the PRIVATE workload, i.e., 25 pages, and to the fact that pages are chosen without replacement to create transaction reference strings, the 30-page transaction size setting is not compatible with the PRIVATE workload.

5.1 Plan of Attack

To explore the tradeoffs between these five alternatives, their performance will be examined under each of the four workloads specified in Section 4.2. As mentioned there, two cases will be considered within each workload, one where the transaction page locality is relatively low and one where it is a factor of three higher; locality is a key factor in determining the tradeoffs between object servers and page servers [DeWi90], and will also be seen to play a significant role in terms of determining the appropriate locking granularity. Within each case, the system size is fixed at 10 clients and the object update probability is varied to obtain different levels of contention and write-read data sharing. Following this series of experiments, some of the other experiments that we have conducted are summarized.

The primary performance metric for the study is the throughput (transaction completion rate) of the system.[5] We also examined a number of additional metrics in carrying out our analysis of the results (e.g., client and server resource utilizations, average per-transaction message counts, average lock waits, and transaction restart rates), but will not be specifically showing that data here. To ensure the statistical validity of our results, we verified that the 90% confidence intervals for transaction response times (computed using batch means) were sufficiently tight. The confidence interval sizes were within a few percent of the mean in most cases, which is more than sufficient for our purposes.

5.2 HOTCOLD Workload

The first workload to be examined is the HOTCOLD workload. In this workload, each client has an affinity for its own preferred region of the database, directing 80% of its accesses to that specific region and only 20% to the database as a whole. Figures 3 and 4 show the throughput results for the relatively low and high page localities, respectively, for this workload. To aid in the interpretation of the results, Figure 5 shows how the per-page update probability behaves as a function of these figures' x-axis, i.e., as a function of the per-object update probability,

[5]Since we are using a closed queuing model, the inverse relationship between throughput and response time makes either metric sufficient.

Parameter	Meaning	HOTCOLD	UNIFORM	HICON	PRIVATE
TransSize	Mean no. of pages accessed per trans.	30 or 10	30 or 10	30 or 10	10
PageLocality	No. of objects accessed per page (min-max)	1-7 or 8-16	1-7 or 8-16	1-7 or 8-16	8-16
AccessPattern	Object access pattern	unclustered	unclustered	unclustered	unclustered
HotBounds	Page bounds of hot range	p to p+49, $p = 50(n-1)+1$	-	1 to 250	p to p+24, $p = 25(n-1)+1$
ColdBounds	Page bounds of cold range	rest of DB	whole DB	rest of DB	626 to 1250
HotAccProb	Prob. of accessing a page in the hot range	0.8	-	0.8	0.5
HotWrtProb	Prob. of updating an accessed hot object	0.02 to 0.5	-	0.02 to 0.5	0.02 to 0.5
ColdWrtProb	Prob. of updating an accessed cold object	0.02 to 0.5	0.02 to 0.5	0.02 to 0.5	0
PerObjInst	Mean no. of CPU inst. per object on read (doubled on write)	5000	5000	5000	5000
ThinkTime	Mean think time between transactions	0	0	0	0

Table 2: Workload Parameter Definitions and Settings for Client n

for several different page localities. (The behavior shown in Figure 5 is general, not workload-dependent, so it will help us to interpret the results for other workloads as well.)

We begin by considering the results shown in Figure 3 for the case with a relatively low page locality. Initially, when the write probability is low, object updates are few and data contention is negligible. In this region, the fully adaptive page server (PS-AA), the adaptive callback page server (PS-OA), and the basic page server (PS) all perform equally well; the vast majority of their messages in this case are page-level reads and page-level callbacks. The page server with object locking and callbacks (PS-OO) performs slightly worse due to its object-level callbacks (which are the only way that it differs from PS-OA). In PS-OO, when a client tries to update objects in its hot region that other clients have read and therefore cached, the cached copies are called back one object at a time; in contrast, the three best performers call back a whole page of objects at a time, thereby saving significantly on callback messages. These extra messages in PS-OO hurt performance due to the CPU burden for message handling that they impose on the server. Finally, the basic object server (OS) performs the worst here due to the many additional messages implied by its object-at-a-time approach to requesting data from the server.

As the write probability is increased in Figure 3, throughput goes down because more updates bring more work (both CPU processing and I/Os) and also more data contention. As this happens, significant differences appear among the page server (PS) alternatives. PS-AA, the fully adaptive approach, stands out as the best; PS-OA is next, then basic PS, and PS-OO performs the worst. PS-OA worsens with write probability because, though it performs page-level callbacks if it can, it requests individual object-level write locks, and these requests involve messages. PS-AA wins here because only about 10% of the pages that a given transaction updates turn out to require object locks; the rest are page-locked, saving many write lock messages over PS-OA. Basic PS suffers because of its page-level lock granularity – as the object write probability is increased, page-level data contention grows rather rapidly (as indicated by the middle curve in Figure 5). Thus, PS experiences contention due to false sharing that PS-AA manages to head off by de-escalating locks when necessary. Finally, PS-OO and OS perform poorly for the reasons discussed in the preceding paragraph.

We now turn to Figure 4, the case with much higher page locality. The relative ordering of the alternatives is nearly the same as before, with one major exception: basic PS does very well here. This is due to the much higher page locality, which enables transactions to process the same number of objects with fewer page accesses, yielding much less data contention for PS than before. With its contention problems largely swept aside, PS is now able to benefit significantly from locking and calling back a number of objects at once due to its page-level nature. Put differently, the alternatives that request locks and/or data on a per-object basis suffer more here, as there is now a higher relative overhead for operating at the object level. In fact, this higher relative overhead causes the server to become CPU-bound due to message overhead for these alternatives, which explains why they perform so much worse than PS does as the write probability is increased. In contrast, because of its adaptive nature, PS-AA still performs quite well. Unlike the other alternatives, PS-AA almost matches the performance of PS, being just a little worse due to message overhead (because it still occasionally handles objects at the object level).

5.3 UNIFORM Workload

The next workload to be considered is the UNIFORM workload, where all clients access data uniformly throughout the entire database. Due to the absence of per-client data skew, inter-client sharing of pages is more likely, therefore giving the alternatives that allow object-level sharing potentially more to gain. Figures 6 and 7 show the low and high page locality results for this workload, respectively. Due to the lack of locality in the workload, the server disks turn out to be more of a bottleneck here; this makes the performance differences between the alternative page servers a bit smaller, as message differences are not quite as important as they were in the HOTCOLD case (although they are certainly still important, particularly in the high page locality case).

As before, let us first look at the low locality case (Figure 6). The performance of the basic page server (PS) is relatively worse here than in the HOTCOLD workload due to the higher level of data contention in the UNIFORM workload. In fact, the performance of PS is even worse than that of the basic object server (OS) for write probabilities beyond 0.1. The relative ordering of the remaining three page server approaches is the same as before, with PS-AA beating PS-OA, which beats PS-OO, and for the same basic reasons. PS-AA outperforms PS-OA only slightly; PS-AA has a bit more overhead than it

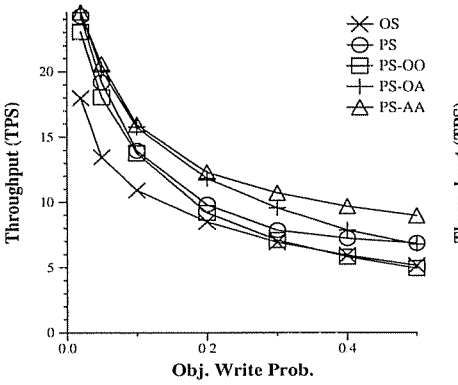

Figure 3: HOTCOLD, 10 Clients
transSize = 30, *pageLocality* = 4 (*avg*)

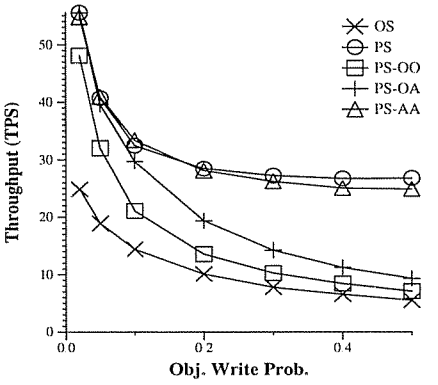

Figure 4: HOTCOLD, 10 Clients
transSize = 10, *pageLocality* = 12 (*avg*)

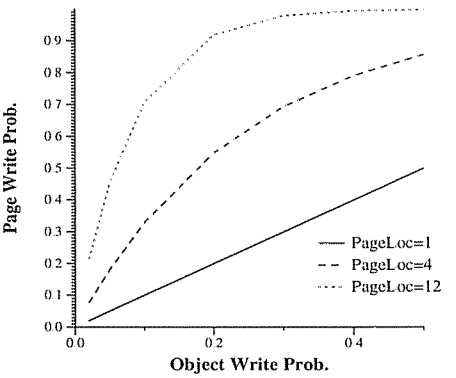

Figure 5: Page Write Probability

did before because a larger percentage of its updated pages are locked at the object level (20% as opposed to 10%), and PS-OA has less to lose here due to its message overhead (as explained above). PS-OO does somewhat better here due to the lack of a hot/cold page distinction in the UNIFORM workload. (In the HOTCOLD workload, performance suffered under PS-OO due to cases where a client accessed another client's hot page and would then have to endure a series of object-at-a-time callbacks by the client for whom the page is hot.)

We now turn to the high locality case (Figure 7). As before, the performance of PS is now much better because the high locality significantly reduces data contention. As Tay has shown [Tay85], contention grows as the square of transaction size, so reducing the transaction size for PS by a factor of three (in pages) implies a nine-fold decrease in inter-client page contention. Moreover, because of the high page locality, the alternatives that lock objects instead of pages all now suffer a much greater relative overhead, leading the right-hand side of Figure 7 to tell a story similar to that of Figure 4. As we saw there, only PS-AA can manage to match the performance of the basic page server given the high page locality.

5.4 HICON Workload

Figures 8 and 9 show the results for the HICON workload, where all clients have the *same* data access skew and the degree of data contention is very high as a consequence. While we would never expect anything close to this degree of data contention in an OODBMS, we were interested in seeing how well PS-AA manages to adapt under such conditions.

In most respects, Figures 8 and 9 are fairly similar to the UNIFORM figures (Figures 6-7). This is to be expected, as HICON is also a uniform workload in the sense that all clients have the same data affinity; HICON differs only in that it involves much more data contention as the write probability is increased. Thus, most of the analysis in the preceding section carries over to this experiment as well. The main difference lies in PS-AA's performance in the high page locality case (Figure 9). As the write probability is increased, PS-AA is not able to track the performance of the basic page server in this case, making PS the clear performance leader at high write probabilities in Figure 9. At first, this seems counter-intuitive: How can PS possibly beat PS-AA under such high contention?

To understand the reason for these results, it is useful to consider the meaning of the topmost curve of Figure 5. This curve shows that for the object write probability range where PS performs the best in Figure 9, the page write probability is very close to 1.0. This means that contention for PS stops growing for object write probabilities beyond 0.2. In fact, contention for PS actually drops a bit for the higher write probabilities because pages are updated earlier; since PS write-locks pages as soon as any of their objects are updated, it obtains locks sooner and the likelihood of deadlocks is reduced. In contrast, PS-AA asks for write locks on objects as they are individually updated. As a result, many more deadlocks and transaction aborts occur under PS-AA here, which slows down transactions and further increases contention. The lesson from this experiment is thus not surprising after all – under a combination of high page locality and object write probability, the majority of page-level conflicts also imply object-level conflicts, making finer-granularity locking harmful rather than helpful. Thus, because PS-AA is doing its job – locking at the object level when page-level contention is present – the basic page server outperforms it here. Fortunately, such extreme data contention is unlikely to arise in realistic OODBMS workloads, so this is not an indictment against PS-AA.

5.5 PRIVATE Workload

Finally, we turn to the PRIVATE workload. As compared to the previous workloads, PRIVATE is most similar to HOTCOLD – but with a read-only cold region, resulting in a workload with no read/write or write/write data sharing (i.e., with no data contention whatsoever). As shown in Table 2, the database is split into two regions. The first half of the database is a set of completely private, per-client hot regions, and the other half is the shared, read-only cold region. The only updates in the workload occur in the private hot regions. Some OODBMS researchers expect such conflict-free workloads to be the norm for engineering design applications, making it interesting to ask how the various alternatives handle them.

The PRIVATE workload results for the high page locality case[6] are shown in Figure 10. The results are easily explained; client caching is very effective in keeping the hot regions

[6]Recall that smaller client hot regions rule out using our low locality workload with PRIVATE. We ran a case with *transSize* = 15 and *pageLocality* = 8 (*avg*) to ensure that the results shown here are indeed indicative of PRIVATE performance.

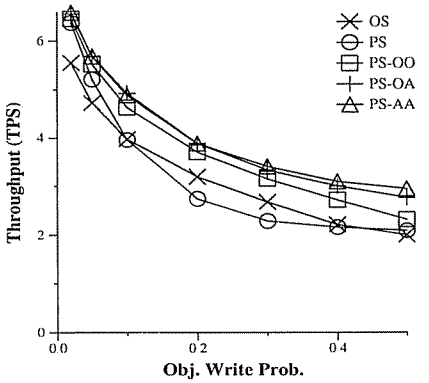

Figure 6: UNIFORM, 10 Clients
transSize = 30, *pageLocality* = 4 (*avg*)

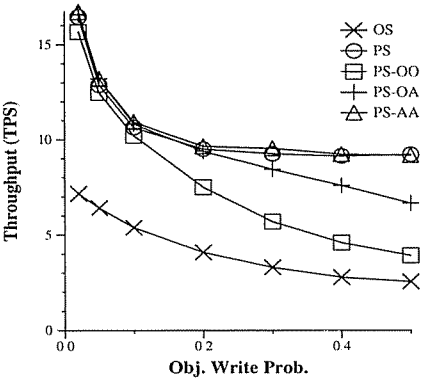

Figure 7: UNIFORM, 10 Clients
transSize = 10, *pageLocality* = 12 (*avg*)

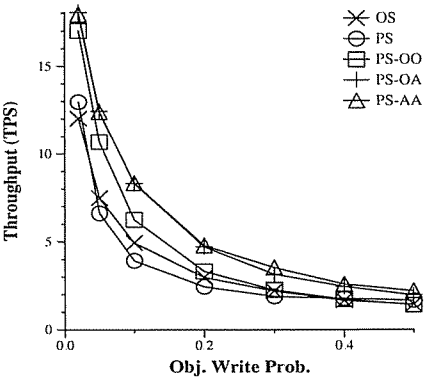

Figure 8: HICON, 10 Clients
transSize = 30, *pageLocality* = 4 (*avg*)

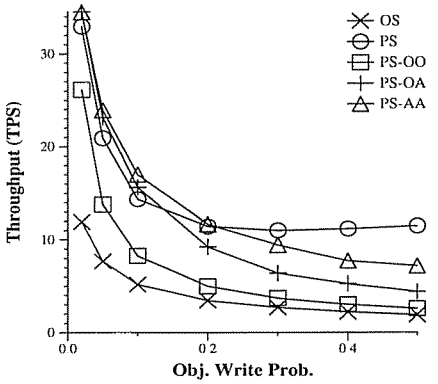

Figure 9: HICON, 10 Clients
transSize = 10, *pageLocality* = 12 (*avg*)

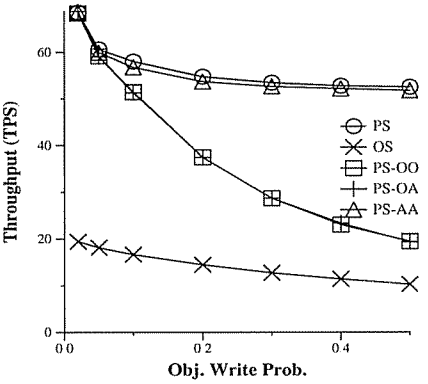

Figure 10: PRIVATE, 10 Clients
transSize = 10, *pageLocality* = 12 (*avg*)

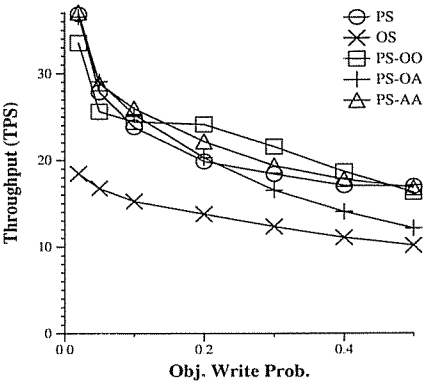

Figure 11: Interleaved-PRIVATE
transSize = 20, *pageLocality* = 6 (*avg*)

cached in all cases, making the system server CPU-bound for alternatives with heavy message requirements. Messages are thus the primary PRIVATE performance determinant. Since there is no data contention, both PS and PS-AA request page-level write locks in this case, thereby providing good performance. Performance degrades significantly with increasing write probability for PS-OA and PS-OO due to the fact that they send many messages to ask for object-level write locks; the two are indistinguishable because no callbacks occur under the PRIVATE workload. As expected, since OS does everything at the object-level, it performs the worst.

When considering the performance issues for contention-free workloads, one other question came to mind: How would the alternatives perform for workloads where there is no contention for objects, but where significant page-level contention exists (i.e., heavy false sharing)? To answer this question, we developed a variant of the PRIVATE workload called *Interleaved PRIVATE*. This workload models an extreme case of false sharing; it was arrived at by interchanging objects between pairs of database pages spaced at 25-page intervals so that the hot regions of clients are combined in a pairwise fashion – the hot objects for client 1 now reside in the top half of each page of their combined (50-page) hot region, and client 2's hot objects are on the bottom half of each page, with clients 3 and 4, 5 and 6, and so on having their hot regions interleaved in the same manner. We then took the PRIVATE workload transactions from before and had them access the same objects, but in their new locations, yielding a workload with transactions that can be roughly described as having *transSize* = 20 and *pageLocality* = 6 (*avg*).

Figure 11 shows the results for the Interleaved Private workload. Compared to Figure 10, several differences are apparent. First, while OS is still the worst alternative, it is relatively better due to the lower page locality. Second, also due to the lower page locality, the performance differences are smaller here than in the original PRIVATE workload. Third, and most importantly, the relative performance of the alternatives is somewhat different here. Over some of the range of write probabilities, in fact, PS-OO, which locks and calls back objects, offers the best performance, even outperforming PS-AA to some extent. This is due to the extreme nature of the Interleaved PRIVATE workload – since the hot objects for a given client occupy half of a page that also contains hot objects from its corresponding neighbor, the alternatives that use page-level callbacks create a significant "ping-pong" effect, with pages moving back and forth between the clients (via the server, of course). This can happen frequently in PS-AA and PS-OA because a client transaction of average size 20 will access about 40% of the pages in the client's hot region; thus, a neighboring transaction that updates objects residing in any of the other 60% of the hot region pages will call them back in their entirety. Since PS-OO does object-level callbacks, it allows each client to simply cache and retain their hot objects, avoiding the "ping-pong" problem. (OS actually avoids it too, but this advantage is overwhelmed by the other message costs of OS.) However, the performance of PS-OO degrades as the write probability is

increased due to increasing message overhead associated with object-level write lock requests.

5.6 Scalability and Completeness

In this section, we discuss several other experiments. We first address database-size scaling, and then briefly summarize key lessons from our other parametric explorations.

5.6.1 Scalability of Results

In reading through the analyses of the various workloads, the reader will hopefully have noticed two things – that the reasons for the results make a good deal of sense independent of the workload particulars, and that they depend on relative rather than absolute conditions. Thus, while we have used a small database and small caches to make our simulations less costly, there is nothing in particular about the results that should keep them from being equally applicable given much larger databases and memories. To verify this, we re-ran our HOTCOLD, UNIFORM, and HICON experiments with the database and client and server buffer pool sizes all scaled up by a factor of nine. To reestablish similar operating conditions following this order of magnitude increase, we also scaled up the transaction size in pages. Since contention decreases linearly in the database size, but increases as the square of the transaction length [Tay85], the transactions were scaled up by a factor of three.

Figures 12-14 show the results of the scaled-up case for the low page locality. To make it easy to compare trends with the experiments with the small database and buffer pools, the results are shown in a normalized fashion. Specifically, since PS-AA was generally the best performer throughout, the figures plot the throughput of the other alternatives as a fraction of PS-AA's throughput. The solid lines show the results for the scaled-up case, with the corresponding dotted lines showing the previous results. It is clear from the curves that the results scale as expected; the trends are the same, and lead to the same conclusions regarding the relative performance of the alternatives. The only arguably significant difference appears in Figure 12, where OS performs even worse for the HOTCOLD workload relative to the other alternatives.

5.6.2 Parameter Space Coverage

During the process of trying to understand the tradeoffs among the alternatives, we also conducted a number of other experiments. Though space prevents us from presenting those results in detail, we summarize them briefly here. Our other experiments included varying the number of client workstations a la [Care91, Fran92a], using the clustered object access pattern, and reducing the network bandwidth by a factor of ten. In all cases, while the numbers were different, the qualititative results told the same basic story regarding the algorithm tradeoffs and the relative superiority of PS-AA. We also tried experiments with other transaction sizes and localities; the only results that differed from those of the experiments covered here were results with an extreme page locality of one. That extreme case yielded the only cases that we found where the object server approach was competitive; under the UNIFORM workload, OS won slightly (but only very briefly), and under the HOTCOLD workload, OS was somewhat better than PS-AA over the entire range of write probabilities. In both cases, the communication cost saved by not sending a whole page to get a single object was the main reason for the improvement, a la [DeWi90].

6 Other Granularity Issues

In this section we discuss two additional granularity issues: concurrent page updates and "grouped-object" servers.

6.1 Concurrent Page Updates

As discussed in Section 3.3, combining object-level locking with page-granularity data transfers raises the issue of managing concurrent updates to pages. In general, this issue can be resolved by either merging the updates of multiple clients when necessary, or by disallowing concurrent updates using a write-token approach. Merging can be CPU intensive and may also require additional disk I/O at the server. On the other hand, a write-token approach has the potential to be more communication-intensive, as pages must often be sent along with the transfer of the write-token. In a workstation/server OODBMS, the cost of bouncing pages among multiple updaters could be significant due to the relatively high communication overheads of a local area network.

For this reason, we have focused our work on approaches that allow concurrent page updates to occur.[7] As stated previously, with fixed-length objects, the merging of concurrent updates is fairly straightforward. When objects are allowed to change size, however, the merge operation can become more complicated. For example, if two objects on a given page are increased in size by concurrent updaters, it is possible that a subsequent merging of the two updates will cause the page to overflow. Overflow can be handled using a standard forwarding technique a la [Astr76], but this requires additional mechanism at the server and could entail extra disk I/Os to update the anchor pages of forwarded objects. Another approach would be to use a space-reservation protocol; e.g., potential updaters could be allocated a maximum update size beyond which they would have to perform the forwarding themselves, as normal.

An alternative to merging that still allows concurrent page updates can be applied in systems that use write-ahead-logging (WAL). The WAL protocol ensures that all relevant log records are sent to the server prior to a transaction commit. These log records could be replayed at the server in order to update the pages at the server, obviating the need to merge already-updated pages that have been sent from clients. This "redo-at-server" scheme is simple to implement (and was thus chosen for the initial version of SHORE [Care94]). However, it has the drawback of shifting a significant burden to the server, as the server must repeat the updates performed by all clients. This could negate one of the primary advantages of data-shipping, namely the offloading of the server by utilizing client resources.

6.2 Grouped-Object Servers

A second issue that we wish to address is our choice of a page server as the foundation for our hybrid approaches. In particular, it can be argued that it is possible to group

[7] As future work, we do also plan to explore the write token approach.

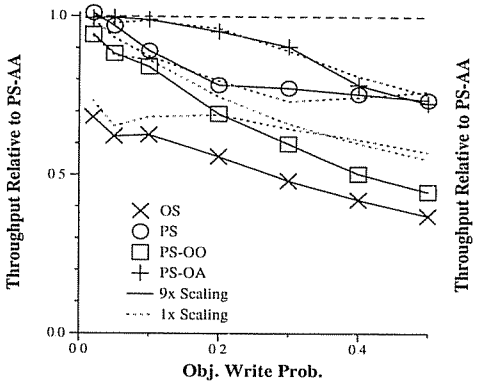

Figure 12: HOTCOLD Scaling, 10 Cli
transSize = 90, *pageLocality* = 4 (*avg*)

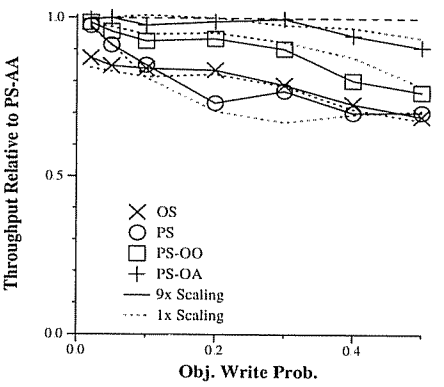

Figure 13: UNIFORM Scaling, 10 Cli
transSize = 90, *pageLocality* = 4 (*avg*)

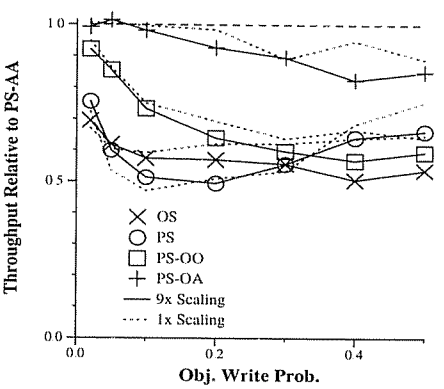

Figure 14: HICON Scaling, 10 Clients
transSize = 30, *pageLocality* = 4 (*avg*)

objects into coarser units – independent of pages – in order to obtain the benefits of large data transfers. A variant of this approach, which we call a "grouped-object" server, is provided by the Versant OODBMS. While grouped-objects can easily be used for data transfer, they can only be used for cache consistency if the grouping is done in a static fashion. This is because all clients must agree as to which objects belong to a particular granule in order for consistency to be maintained; furthermore, all clients must follow the same hierarchical locking protocol over the grouped objects. Thus, if grouping is done dynamically, grouping will only be usable for data transfers; in this case, a grouped-object server will have performance characteristics similar to our PS-OO approach, which typically had lower performance than PS-AA.

7 Conclusions

In this paper, we have studied the granularity choices that arise in the areas of client-server data transfers and cache consistency for data-shipping OODBMS architectures. While existing OODBMSs tend to favor the page server approach, this approach has been criticized for being too restrictive when it comes to concurrency; this is due to the fact that existing implementations have used pages as the granularity for locking and callbacks as well as for data transfers. We showed that this criticism is unwarranted by presenting three page server variants that allow for concurrent data sharing at the object level while retaining the performance advantages of shipping pages in response to client data access requests. The first was a static approach that performs locking and callbacks at the object level; the second approach is similar, but it uses page-level callbacks when it can; the third approach also uses page-level locking when it can, switching to object-level locking only for pages where finer-grained sharing occurs. We studied the performance of these approaches for a wide range of client data sharing patterns, comparing them to each other and to pure page and object servers. For the range of workloads examined, our results indicate that a page server is clearly preferable to an object server. Moreover, the adaptive page server was shown to provide very good performance; it outperformed the pure page server, the pure object server, and the other two alternatives. As a result, we plan to use this approach in the context of the SHORE system that we are now building [Care94].

References

[Astr76] M. Astrahan, *et al.*, "System R: Relational Approach to Database Management", *ACM TODS* 1(2), 1976.

[Care91] M. Carey, M. Franklin, M. Livny, E. Shekita, "Data Caching Tradeoffs in Client-Server DBMS Architectures", *ACM SIGMOD Conf.*, Denver, June 1991.

[Care93] M. Carey, D. DeWitt, J. Naughton. "The OO7 Benchmark" *ACM SIGMOD Conf.*, Washington D.C., 1993.

[Care94] M. Carey, *et al.* "Shoring up Persistent Applications", ACM SIGMOD Conf., Minneapolis, May, 1994.

[Catt91] R. Cattell, *Object Data Management*, Addison Wesley, Reading, MA, 1991.

[Catt92] R. Cattell, J. Skeen, "Object Operations Benchmark", *ACM TODS*, 17(1), March 1992.

[Chu94] S. Chu, M. Winslett, "Minipage Locking Support for Page-Server Database Management Systems", *submitted for publication*, Feb. 1994.

[DeWi90] D. DeWitt, P. Futtersack, D. Maier, F. Velez, "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems", *16th VLDB Conf.*, Brisbane, Australia, Aug. 1990.

[Fran92a] M. Franklin, M. Carey, "Client-Server Caching Revisited", *Proc. Int'l Workshop on Distributed Object Mgmt.*, Edmonton, Canada, Aug. 1992.

[Fran92b] M. Franklin, *et al.* "Crash Recovery in Client-Server EXODUS", ACM SIGMOD Conf. San Diego, June, 1992.

[Fran93] M. Franklin, *Caching and Memory Management in Client-Server Database Systems*, Ph.D. Thesis, TR #1168, Computer Sciences Dept., Univ. of WI, Madison, July 1993.

[Gray79] J. Gray "Notes on Database Operating Systems" *Operating Systems: An Advanced Course*, Springer-Verlag, 1979.

[Howa88] J. Howard, *et al*, "Scale and Performance in a Distributed File System", *ACM TOCS* 6(1), Feb. 1988.

[Josh91] A. Joshi, "Adaptive Locking Strategies in a Multi-Node Data Sharing System", *17th VLDB Conf.*, Barcelona, 1991.

[Lamb91] C. Lamb, G. Landis, J. Orenstein, D. Weinreb, "The ObjectStore Database System", *CACM* 34(10), Oct. 1991.

[Lehm89] T. Lehman, M. Carey, "A Concurrency Control Algorithm for Memory-Resident Database Systems", *3rd Int'l. FODO Conf.*, Paris, France, June 1989.

[Li89] K. Li, P. Hudak, "Memory Coherence in Shared Virtual Memory Systems", *ACM TOCS*,7(4) Nov., 1989.

[Livn88] M. Livny, *DeNet User's Guide*, Version 1.0, Computer Sciences Dept., Univ. of WI-Madison, 1988.

[Moha91] C. Mohan, I. Narang, "Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment", *17th VLDB Conf.*, Barcelona, Sept. 1991.

[Tay85] Y. Tay, N. Goodman, R. Suri, "Locking Performance in Centralized Databases", *ACM TODS* 10(4), Dec. 1985.

[Wang91] Y. Wang, L. Rowe, "Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture", *ACM SIGMOD Conf.*, Denver, June 1991.

[Wilk90] W. Wilkinson, M. Neimat, "Maintaining Consistency of Client Cached Data", *16th VLDB Conf.*, Brisbane, 1990.