Local Disk Caching for Client-Server Database Systems*

Michael J. Franklin† Michael J. Carey Miron Livny

Computer Sciences Department
University of Wisconsin - Madison
{mjf,carey,miron}@cs.wisc.edu

Abstract

Client disks are a valuable resource that are not adequately exploited by current client-server database systems. In this paper, we propose the use of client disks for caching database pages in an extended cache architecture. We describe four algorithms for managing disk caches and investigate the tradeoffs inherent in keeping a large volume of disk-cached data consistent using a detailed simulation model. The study shows that significant performance gains can be obtained through client disk caching; particularly if the client disk caches are kept consistent. We also address two extensions to the algorithms that arise due to the performance characteristics of large disk caches: 1) methods to reduce the work performed by the server to ensure transaction durability, and 2) techniques for bringing a large disk-resident cache up-to-date after an extended off-line period.

1 Introduction

Object-Oriented Database Management Systems (OODBMS) are typically constructed using client-server software architectures. Examples include products such as O2 [Deux91], Objectivity [Obje91], ObjectStore [Lamb91], Ontos [Onto92], and Versant [Vers91], as well as research systems such as ORION [Kim90] and EXODUS [Fran92c]. A primary motivation for the use of client-server architectures is the desire to exploit the plentiful and relatively inexpensive resources provided by current workstation technology. Moving functionality to the clients provides both performance and scalability benefits. Performance is improved, for example, by reducing interaction with servers and by avoiding the costs of obtaining data from remote sites. Scalability can be improved through the offloading of shared resources such as server machines and the network, which are potential bottlenecks.

Most current OODBMS exploit client *processor* resources through the use of a *data-shipping* architecture. In data-shipping systems, clients send requests for specific data items to the server, which obtains the items and sends them back to the client. This approach enables much of the work of data manipulation to be performed at clients. Existing systems exploit client *memory* resources through the use of intra- and inter-transaction caching. In contrast, existing OODBMS provide only limited support for exploiting client *disk* resources. This omission is potentially costly, as client disks represent a valuable addition to the storage hierarchy of a client-server OODBMS due to their capacity and non-volatility. Inter-transaction memory caching and other client memory management techniques have been shown to provide substantial performance benefits for client-server database systems [Wilk90, Care91, Wang91, Fran92a, Fran92b]. In this paper, we investigate the extension of client-server caching to the use of client disks. We focus on data-shipping systems in which pages serve as the unit of interaction between clients and servers. Such systems are referred to as *page servers* [DeWi90].

1.1 Alternatives for Integrating Client Disks

There are several ways in which existing OODBMS typically allow client disks to be used. The first, and most common, is to use local disks to make each client a *server* for part of the database. The data to be placed on client disks is determined statically by partitioning the database among the clients. Clients are given ownership of the data pages for which they act as server, meaning that they are responsible for maintaining the consistency of the data, and must always be capable of providing its most recent committed value. Another way that existing systems allow the use of client disks is indirectly, through virtual memory swapping. Most OODBMS keep their client caches in virtual memory. If the cache is larger than the allocated physical memory, the operating system will swap parts of it to disk. The Versant system provides an additional way of using client disks, called the "Personal Database" [Vers91]. Users can check objects out from the shared database and place them in a personal database, which can reside on the client disk. Objects that are checked out cannot be accessed by other clients.

The first approach, making clients act as servers, has implications for data availability; as the crash of a client causes the data owned by that client to become unavailable. Allowing clients to own data is problematic due to inherent asymmetries in workstation-server environments. For example, clients are typically managed by users and placed in individual offices (or briefcases), while in contrast, servers are managed by an operations staff and kept in protected machine rooms. Also, it is more cost effective to place duplexed log disk storage or non-volatile memory at server machines than to place such functionality at each client machine. Consequently, we expect servers to be in-

*This work was partially supported by DARPA under contract DAAB07-92-C-Q508, by the National Science Foundation under grant IRI-8657323, and by a research grant from IBM.

†Current Address: Dept. of Computer Science, University of Maryland, College Park, MD 20742.

Proceedings of the 19th VLDB Conference
Dublin, Ireland, 1993

herently more reliable and available than clients. These concerns limit the applicability of client ownership policies to data that is private and/or of limited value, or to systems in which substantial expense is incurred to make clients more reliable. The problems of the second approach, using operating system virtual memory for buffer management, are well known (e.g., [Ston81, Trai82]), and stem from (among other things) the operating system's lack of knowledge about database access patterns and differences in disk management policies. Relying on the operating system to manage the client disk places an important performance issue beyond the control of the database system. Furthermore, virtual memory swapping makes only transient use of client disks, and thus, the disks can cache data pages only while a caching process is active at a client.

In this paper, we study a different approach, namely, we extend our previous work on memory caching [Care91, Fran92a, Fran92b] to take advantage of client disks. Caching enables the use of client disks without incurring the problems associated with giving ownership of data to clients. We refer to this approach as the *extended cache* architecture. The use of client disks as an extended cache provides a *qualitative* change in the utility of caching at client workstations compared to memory-based caching strategies. The lower cost per byte of disk storage increases the amount of data that can be cached at a client, possibly enabling the caching of the entire portion of the database that is of interest at that site. This has the potential to affect the basic trade-offs in cache management (as compared to memory-only caching), and may change the role of the server from that of a data provider to that of an arbiter of data conflicts and a guarantor of transaction semantics.

In terms of related work, we are aware of only one other study of client disk caching for DBMSs [Deli92]. This work investigates a system in which relational query results are cached on client disks but all updates are performed at the server. Prior to executing a query, a client sends a message to the server requesting any updates that have been applied to tuples cached at the client. In response, the clients are sent logs containing relevant updates, which are then applied to the cached query results. As will be seen in Section 2, the extended cache architecture we study is quite different; it allows updates to be performed at clients, and uses the disk as a page cache that is largely an extension of the LRU-managed memory cache. Also related is work on distributed file systems. Unlike existing DBMSs, some distributed file systems do use client disks for caching (e.g., Andrew [Howa88] and its follow-on project CODA [Kist91]). Distributed file systems, however, differ from client-server DBMS in significant ways, including: 1) they support caching at a coarse (e.g., file) granularity, rather than at a page or object granularity, 2) they do not support serializable transactions on the cached files, and 3) they are typically designed under the assumption that sharing is an infrequent occurrence.

1.2 Extended Cache Design Issues

In this paper we address three specific issues in the design of an extended cache. First, we develop algorithms for accessing and ensuring the consistency of data kept in client disk caches, and we study their performance. As will be shown in the study, the success of disk-caching in offloading the server for certain workloads can result in the server disk *writes* becoming the next bottleneck. Therefore, the second issue we study is how to reduce the demands on the server disks that result from transaction updates. Thirdly, we discuss several techniques for managing the transition of client disk caches from an off-line state to an on-line state. These techniques build on the methods that are developed in the first part of the paper for ensuring the consistency of disk-cached data. The remainder of the paper is structured as follows: Section 2 addresses the problem of disk cache management and outlines a group of prospective algorithms. Section 3 briefly describes our client-server simulation model and workloads. Section 4 presents a performance comparison of the algorithms. Section 5 discusses extensions of the algorithms. Finally, Section 6 presents our conclusions and plans for future work.

2 Extended Cache Implementation

In this section we propose algorithms for managing and utilizing client disks in the extended cache architecture. Before describing these algorithms, however, we first elaborate on our model of how the client disk is employed by the database system, and we describe the algorithm we have chosen for maintaining *memory* cache consistency. This algorithm serves as a foundation for all of our extended cache management algorithms.

2.1 Disk Cache Management Pragmatics

We assume that each client has a fixed amount of disk space that is allocated for use as a cache for database pages; this area is managed by the database system. We refer to this space as the *disk cache*, and likewise, we refer to the area of main memory that is used to cache database pages as the *memory cache*. We also assume that each page is tagged with a version number that uniquely identifies the state of the page with respect to the updates applied to it. Such version numbers are typically maintained by DBMS systems to support recovery (e.g., Log Sequence Numbers (LSNs)) [Gray93].

The memory cache is managed through the use of a data structure containing an entry for each resident page and a list of available memory cache slots. The LRU chain is threaded through this structure. The disk cache is managed as a FIFO queue based on when pages are added to the disk cache (rather than LRU). To implement the disk cache efficiently, a structure analogous to that used to manage the memory cache must also be maintained in memory. This structure contains an entry for every page resident in the disk cache and a list of available disk cache slots. At a minimum, the entry for each page contains the position of the page in the disk cache page queue, its location on disk, and its version number (LSN).

Pages flow between the memory and disk caches of a client as shown in Figure 1. A new page is first brought into the memory cache as the result of a cache miss. Pages can be faulted in from the server or from the local disk[1]. To bring a new page into the memory cache, a cache slot must be made available for the page (unless a free slot already exists). To open a memory cache slot, the least recently used page in the memory cache is chosen for replacement. When a page is replaced from the memory cache it is *demoted* to the local disk cache.

[1] Pages copied from the local disk also remain in the disk cache at their current position in the disk cache page queue.

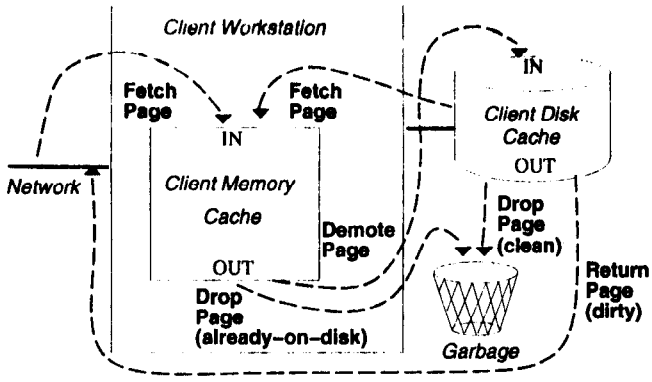

Figure 1: Client Page Flow

There are three cases to consider when demoting a page from the memory cache to the disk cache. The first two cases arise if a copy of the demoted page is already resident in the disk cache. If the copy in the memory cache has not been updated, then its disk-resident copy is simply made the most recently added page in the disk cache by adjusting the disk cache control information (which is memory-resident). Thus, for the first case, no disk access is required. The second case arises when an out-of-date copy of the demoted page is present in the disk cache. In this case, the disk cache copy of the page is overwritten and it becomes the most recently added page in the disk cache. The third case arises when no copy of the demoted page exists in the disk cache. In this case a slot in the disk cache must be made available for the demoted page. The process of opening a disk slot is analogous to the replacement process in the memory cache. If no slot is available, then the least recently added page is removed from the disk cache. The fate of the page chosen for replacement depends on the status of its contents. If the page contains updates that are not reflected in any other copies of the page (e.g., at the server), then a copy of the page is sent to the server. Otherwise, the chosen page is simply overwritten.

2.2 A Memory Cache Management Algorithm

Based on the results of our earlier investigations of memory cache management algorithms we have adopted the Callback-Read (CB-R) algorithm to serve as the basis for our ongoing work. As is shown in [Fran92a], CB-R provides good performance and is robust with respect to many system and workload parameters. Callback-Read is derived from techniques that were originally used to maintain cache consistency in distributed file systems such as Andrew [Howa88] and Sprite [Nels88]; however, these algorithms did not support serializable transactions. Transactional callback locking algorithms were later employed in the ObjectStore OODBMS [Lamb91] and have been studied in [Wang91] and later in [Fran92a].

Under CB-R, all pages in a client's memory cache are guaranteed to be valid. CB-R grants clients authority to read objects in their memory caches, but they must obtain permission from the server to write objects . If a client wishes to read a page that is in its memory cache, it simply acquires a local (to the client) read lock on the page. To write a page, however, the client must first send a request for a write lock to the server. If the page on which the write lock is requested is cached at other sites, the server "calls back" the conflicting permissions by sending requests to the sites which have the page cached. The server tracks the contents of the memory caches at each site, so clients must inform the server when they drop a page from their memory cache. To save messages, this information is piggybacked on other messages sent from the clients to the server, rather than sent immediately. As a result, the server may temporarily believe that page copies are present in caches where they are no longer resident.

At a client, a callback request is treated as a request for an exclusive lock on the specified page. If the request can not be granted immediately (due to a lock conflict), the client responds to the server saying that the page is currently in use. When the callback request is eventually granted at the client, the page is removed from the client's memory cache (i.e., it is *invalidated*) and an acknowledgement message is sent to the server. When all callbacks have been acknowledged, the server grants a write lock on the page to the requesting client. Any subsequent requests by other clients to obtain a copy of the page will be blocked at the server until the write lock is released. At the end of the transaction, the client sends copies of the updated pages to the server and releases its write locks, retaining copies of the pages in its memory cache.

2.3 Extended Cache Management Algorithms

We now develop algorithms that build upon the CB-R memory cache management algorithm to support the use of client disks in the extended cache architecture. Two important aspects of such algorithms are *hierarchy search order* and *consistency maintenance*. The hierarchy search order dictates the process through which a particular data item is found in the storage hierarchy. Consistency maintenance ensures that the caching of data in the storage hierarchy does not cause a violation of transaction execution serializability. These two dimensions define a design space for cache management algorithms incorporating client disks. We first address each dimension separately, and then describe algorithms that integrate them.

2.3.1 Hierarchy Search Order

The goal of the hierarchy search order is to allow transactions to obtain the *lowest cost* copy of a page among the copies present in the system. In this study, there are four potential locations from which a client can obtain a page copy: *1) local client memory, 2) local client disk, 3) server memory, and 4) server disk* [2]. The cost of obtaining a page from a location in the storage hierarchy is dependent on several factors:

1. The path length of accessing the location (e.g., the cost of sending and receiving messages, the cost of performing disk I/O, etc.)

2. Contention for those shared resources required to access the location (e.g., the network, server or client disk, server or client CPU, etc.)

3. The probability of finding the page at the location (e.g., server or client buffer hit rate).

[2] As described in [Fran92b], clients can also be allowed to obtain pages from other clients. The additional resources represented by remote clients can be fit into the framework used in this paper; however, for simplicity this issue is not addressed in this study.

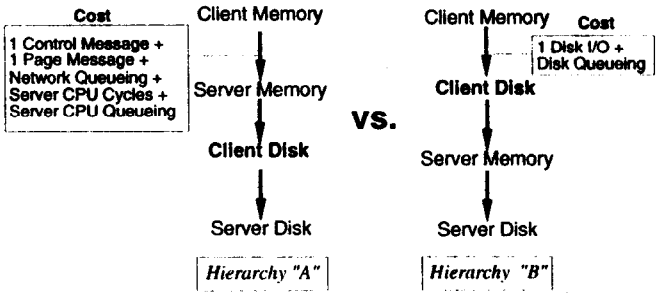

Figure 2: Two Possible Hierarchies

It is important to note that while the path length is relatively fixed for a given configuration, the other two components are dependent on dynamic aspects of the workload such as application mix and intensity. Therefore, in general it is not possible to determine a fixed cost hierarchy for the DBMS to traverse.

Figure 2 shows two possible cost hierarchies for a client-server DBMS if the issue of consistency is ignored (i.e., assuming all copies are valid). In hierarchy "A", the server's memory is assumed to be cheaper to access than the client's local disk cache, while in hierarchy "B", these two levels are inverted. As the client population changes, this inversion can (and does, as will be shown in Section 4) take place. In a system with few clients, there will be low contention for the server and network resources, so the tradeoff will be between the cost of a random disk I/O and the cost of a round-trip RPC to the server. With current technology, the access to the server memory will likely be less expensive than the disk I/O. However, as clients are added, contention for the network and the server will increase. Eventually, the cost of a local disk I/O will fall below the expected cost of a remote memory access.

Despite the dynamic nature of the hierarchy costs, there are some relationships that remain fixed for a given configuration. For example, the local client memory is always the least expensive level, and it is always checked first to determine if the page is already available to the transaction. Likewise, the server memory is always cheaper to access than the server disk. In addition, if the server disks and client disks have the same performance characteristics, then the server disk is always the most expensive location.

2.3.2 Consistency Maintenance

Consistency maintenance ensures that transactions execute in a serializable manner despite the presence of replicated copies of data pages. Two basic mechanisms for performing consistency maintenance are *detection and avoidance*. Detection schemes allow stale copies of data to remain in the system; the validity of a page copy must be confirmed before that copy can be accessed by a transaction, and thus, attempted access to invalid pages is detected and disallowed. Detection-based schemes are often referred to as *check-on-access* algorithms [Dan92].

In contrast, avoidance-based schemes ensure that all accessible copies of a data page are valid so that access to invalid data is avoided. This can be done by *invalidating* other replicas of an updated item (as is done by CB-R for memory cache contents), or by *propagating* the new data value to the other replicas. Our previous studies of memory-based caching showed that invalidation performs better than propagation under many workloads. Propagation preserves replication, thereby increasing the cost of updates. In contrast, invalidation destroys replication of read-write shared data so that subsequent updates incur less consistency overhead. In this study we consider relatively large disk caches and large client populations, which would exacerbate the problems of propagation. Furthermore, propagation to a page copy in the disk cache would require disk I/O, whereas invalidation can be performed by simply modifying memory-resident structures. For these reasons, we restrict our study of avoidance-based mechanisms for disk cache consistency maintenance to algorithms that use invalidation.

2.3.3 Integrated Algorithms

Efficient cache management algorithms must address the interaction of hierarchy search order and consistency maintenance considerations. For example, if the server must be contacted to determine the validity of a page copy, then the cost of accessing the server memory is reduced because the required consistency checking message can also serve as an implicit request for the page. As described in the previous two sections, we consider two options for each of the dimensions in our design space. The four resulting algorithms are shown in Figure 3. For the hierarchy

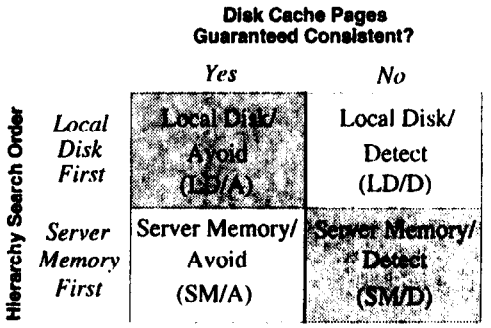

Figure 3: Simplified Algorithm Design Space

search order dimension, we investigate the tradeoffs between algorithms that differ in whether they favor accessing the local disk cache (called LD algorithms) or the server memory (called SM algorithms). In terms of consistency maintenance, all of the algorithms that we develop use CB-R (which is avoidance-based) to manage the client *memory* caches, ensuring that the memory cache contents are always valid. This is because our previous studies showed that avoidance generally performs better than detection for memory caches. For *disk* caches, however, we re-examine the tradeoffs between avoidance and detection. This re-examination is undertaken because disk caches are much larger than memory caches and typically hold colder data, resulting in different tradeoffs than for memory caches.

The two avoidance-based algorithms that we study are called Local Disk/Avoid (LD/A) and Server Memory/Avoid (SM/A). Both of these algorithms extend the CB-R algorithm to ensure that all pages resident in the disk caches (in addition to those in the memory caches) are valid. Consequently, clients can read pages from their local disk cache without contacting the server. Likewise, clients must invalidate page copies from both their memory and disk caches in order to service a callback request. These algorithms extend CB-R to the disk cache contents, so the server must track the contents of both the memory and disk caches at the clients. Clients inform the server when they no

longer have a copy of a page in either cache, rather than when the page is removed from the memory cache as in standard CB-R.

Using LD/A, clients request a page from the server only if that page is absent from both local caches. In contrast, SM/A assumes that the server memory is cheaper to access than the local disk; therefore, when an SM/A client fails to find a page in its memory cache, it sends a request for the page to the server. In the request, it includes an indication of whether or not it has a copy of the page in its local disk cache. If the page is resident in the server's memory, the server will send a copy of the page to the client. If the page is not in the server's memory, but a copy is resident in the client's disk cache, then the server simply tells the client to read the page from its local disk cache. LD/A and SM/A access the server's disk only as a last resort.

In contrast, the detection-based algorithms, Local Disk/Detect (LD/D) and Server Memory/Detect (SM/D), allow pages in the disk caches to be out-of-date. Therefore, clients must contact the server to determine the validity of page copies in their local disk cache. The client is allowed to fault a disk-cached page into its memory cache only if the server replies that the disk-cached page copy is valid[3]. For these algorithms, the server tracks only the contents of the memory caches at the clients. As with standard CB-R, clients inform the server when they drop a page from their memory cache, even if the a copy of the page resides in the client disk cache. As a result, a page in the client disk cache at a site is invalidated by the callback mechanism only if a copy of the page also resides in the memory cache at that site.

When LD/D incurs a buffer miss in a client memory cache, it checks the disk cache information to see if the disk cache holds a copy of the desired page. If so, it obtains the version number of the disk-resident copy (from the in-memory disk cache control table) and sends it to the server. The server checks to see if the client's copy is valid and if so, informs the client that it can access the cached page copy. If the client does not have a valid copy of the page, then the server obtains one (either from its memory or from its disk) and returns it to the client. SM/D works similarly, but takes advantage of its communication with the server to access the server memory first. When the server receives a validity check request for a page that is resident in the server memory cache it returns the page regardless of the status of the client's disk-resident copy. Therefore, the SM/D algorithm uses a larger return message to avoid performing a disk read at the client. As with the avoidance-based algorithms, the server's disk is the last place accessed.

2.4 Algorithm Tradeoffs

In this section we briefly summarize the performance-related tradeoffs among the cache management algorithms. The most intricate tradeoffs that arise in this study are those between using detection or avoidance to maintain disk cache consistency. These tradeoffs are similar to those that arise in memory-only caching, but they differ qualitatively due to the higher capacity of disk caches and the potential shifting of bottlenecks when client disks are employed. There are three main tradeoffs to consider: 1) message requirements, 2) server memory requirements, and 3) effective disk cache size. In terms of messages, avoidance is most efficient when read-write data sharing is rare; consistency is maintained virtually for free in the absence of read-write sharing but requires communication when sharing arises. In contrast, detection incurs a constant overhead regardless of the amount of contention. This overhead is higher than that of avoidance under light read-write sharing, but can be lower than for avoidance when the sharing level is increased. The second tradeoff is the size of the page copy information that the server must maintain under each scheme. This information is kept memory resident, so it consumes space that could otherwise be used to buffer pages. For avoidance, the server maintains a record of each page copy residing in a client disk cache, while for detection, the server needs only to keep a list of LSNs for pages that may be in one or more disk caches.

The third tradeoff relates to a metric we call the *effective disk cache size*. It has been observed that for memory caches, the amount of useful (valid) data that can be kept in client caches is lower for detection than for avoidance [Dan90, Care91]. This is because detection allows pages to remain in the cache after they become out-of-date. Such pages are invalid, so they provide no benefit, but take up cache slots that could be used to store valid pages. Avoidance on the other hand, uses invalidation to remove out-of-date pages, thus opening more cache slots for valid pages. As a result, in the presence of read-write sharing we should expect that the detection based algorithms (LD/D and SM/D) will have a smaller effective disk cache size than the avoidance-based ones (LD/A and SM/A).

There are several other tradeoffs that affect the relative performance of the algorithms. A small tradeoff arises when choosing between using client disks for caching or not. Client memory space is required to store the information used to manage the local disk cache and this information reduces the size of the client memory cache. There are also the obvious tradeoffs between the hierarchy search orders. The SM algorithms both attempt to avoid accessing local disks. SM/D does this by using a larger response message to a validity check request when the requested page is in the server memory, while SM/A uses a separate round-trip communication with the server to obtain the page. In contrast, the LD algorithms will use a local disk first, even if there is low contention for shared resources. In particular, LD/D will ignore the presence of a page copy in the server's memory even though it has to contact the server to check the validity of the copy of the page in its disk cache.

3 A Client-Server Caching Model

3.1 The System Model

In order to study the performance of the disk cache management algorithms, we have extended an existing page server DBMS simulation model to include client disk caches. In this section, we briefly describe the model; a more detailed description of the basic model can be found in [Care91]. The model was constructed using the DeNet discrete event simulation language [Livn88]. It consists of components that model a server machine and a varying number of client workstations that are connected over a simple network. Each client site consists of a *Cache Manager* that manages the contents of memory and disk caches using an LRU page replacement policy, a *Concurrency*

[3]Recall that once a page is placed in the memory cache, it is guaranteed to be valid for the duration of its residency.

Parameter	Setting	Parameter	Setting
Instruction rate of client CPU	15 MIPS	Size of a page	4,096 bytes
Instruction rate of server CPU	30 MIPS	Size of database in pages	2500 (10 MBytes)
Per-client memory size	3% of DB size	Number of client workstations	1 to 50
Per-client disk cache size	50% of DB size	Fixed no. of instructions per message	20,000 instructions
Number of disks per client	1 disk	Addl. instructions per msg. size	10,000 inst./4Kb page
Server memory size	30% of DB size	Size in bytes of a control message	256 bytes
Number of disks at server	3 disks	No. of instructions per lock/unlock pair	300 instructions
Minimum disk access time	10 milliseconds	No. of inst. to register/unregister a copy	300 instructions
Maximum disk access time	30 milliseconds	CPU Overhead for performing disk I/O	5000 instructions
Network bandwidth	8 or 80 Mbits per sec		

Table 1: System and Overhead Parameters

Parameter	UNIFORM-WH	HOTCOLD	PRIVATE
Mean number of pages accessed per transaction	20 pages	20 pages	16 pages
Page bounds of hot range	1 to 1250	p to p+49, $p = 50(n-1)+1$	p to p+24 $p = 25(n-1)+1$
Page bounds of cold range	1251 to 2500	rest of DB	1251 to 2500
Probability of accessing a page in the hot range	0.5	0.8	0.5
Probability that a hot range access is a write	0.1	0.1	0.1
Probability that a cold range access is a write	0.0	0.1	0.0
Mean no. of CPU instructions per page on read (doubled on write)	30,000	30,000	30,000
Mean think time between client transactions	0	0	0

Table 2: Workload Parameter Meanings and Settings for Client n

Control Manager that provides locking and consistency management support, a *Resource Manager* that models CPU and disk service and provides access to the network, and a *Client Manager* that coordinates the execution of transactions at the client. Each client also has a module called the *Transaction Source* which submits transactions to the client according to the workload model described in the following section. Transactions are represented as page reference strings and are submitted to the client one-at-a-time; upon completion of a transaction, the source waits for a specified think time and then submits a new transaction. When a transaction aborts, it is resubmitted with the same page reference string. The number of client machines is a parameter to the model. The server is modeled similarly to the clients, but with the following differences: the Concurrency Control Manager has the ability to store information about the location of page copies in the system and also manages locks, there is a *Server Manager* component that coordinates the operation of the server (analogous to the client's Client Manager), and there is no Transaction Source module (since all transactions originate at client workstations).

Table 1 describes the parameters that are used to specify the resources and overheads of the system and shows the settings used in this study. We use a relatively small database in order to make the detailed modelling of the cache behavior of a large distributed system feasible in terms of simulation time. The memory and disk sizes of the server and clients are specified as a percentage of the database size, and it is these ratios that are the important factor here, not the absolute database size. The simulated CPUs of the system are managed using a two-level priority scheme. System CPU requests, such as those for message and disk handling, are given priority over user (transaction) requests. System requests are handled using a FIFO queueing discipline, while a processor-sharing discipline is employed for user requests. At the server, each disk has a FIFO queue of requests; the disk used to service a particular request is chosen uniformly from among all the server's disks. In the current study, each client has at most one disk. For disks at the server and at the clients, the disk access time is drawn from a uniform distribution between a specified minimum and maximum. In order to make client disk caching cost-effective, writes to the client disks must be done asynchronously; thus, two memory pages are reserved for use as I/O write buffers at each client. Likewise, at the server, one memory page per active client is reserved for use as an I/O write buffer.

Due to the large disk cache sizes that we use in this study, we preload each client disk in order to reduce the impact of a long warm-up phase on the statistics produced by a simulation run. The initial pages placed on a client's disk are chosen randomly from among the pages that may be accessed at that client according to the workload specification (see Section 3.2). The warm-up phase also affects the amount of memory that needs to be reserved for disk management information at clients. At the start of the simulation we initially reserve the maximum amount required based on the disk cache size (assuming 20 bytes of descriptive data per disk page) and then reduce the allocation (if necessary) after running the simulation for a brief period. After the I/O write buffers and the disk information pages are subtracted, the remainder of the client's memory is available for use as a memory cache. As stated in Section 2.4, the server also uses memory for the structures that it uses to track page copy locations. The amount of memory reserved for this structure is changed dynamically during the simulation run, based on the number of outstanding page copies that the server must track.

Finally, we use a very simple network model for the simulator; the network is modeled as a FIFO server with a specified bandwidth. We did not model the details of the operation of a

specific type of network (e.g., Ethernet, token ring, etc.). Rather, the approach we took was to separate the CPU costs of messages from the on-the-wire costs, and to allow the on-the-wire costs to be adjusted using the bandwidth parameter. In this study, we use two network bandwidths that correspond roughly to current Ethernet (referred to as the *slow* network in the following sections) and FDDI (referred to as the *fast* network) speeds. The bandwidth values used (8 Mbits/sec and 80 Mbits/sec respectively) represent slightly discounted values of the stated bandwidths of those networks. The CPU cost of managing the protocol for a send or a receive is modeled as a fixed number of instructions per message plus a charge per byte.

3.2 Workloads

We use the simulation model to study the performance of the disk cache management algorithms under a variety of system configurations and workloads. The relative benefits of a particular algorithm can vary depending on many workload factors such as access locality, update intensity, amount of read sharing and read-write sharing among clients, etc. Our simulation model provides a simple but flexible mechanism for describing a variety of workloads. The access pattern for each client can be specified separately using the parameters shown in Table 2. Transactions are represented as a string of page access requests in which some accesses are for reads and others are for writes. Two ranges of database pages can be specified: a hot range and a cold range. The probability of an access to a page in the hot range is specified; the remainder of the accesses are directed to cold range pages. For both ranges, the probability that an access to a page in the range will be a write access (in addition to a read access) is specified. The parameters also allow the specification of an average number of instructions to be performed at the client for each page access, once the proper lock has been obtained. This number is doubled for write accesses.

We present results for three workloads in this paper. The characteristics of these workloads are summarized in Table 2. UNIFORM-WH is a low-locality workload in which half of the database is read-write shared while the other half is shared in a read-only manner. Low per-client locality and the presence of read-write sharing both negatively impact the performance of caching at clients. The HOTCOLD workload has a high degree of locality per client and a moderate amount of read-write sharing among clients. The high locality and read-write sharing of this workload provide a test of efficiency of the consistency maintenance mechanisms used by the algorithms. Finally, the PRIVATE workload has high per-client locality and no read-write sharing. We expect this type of access to be typical in applications such as large CAD systems or software development environments in which users access and modify their own portion of a design while reading from a library of shared components. The PRIVATE workload represents a very favorable environment for client caching.

4 Performance of the Extended Cache

We ran simulation experiments using all four algorithms described in Section 2.3.3. To simplify the presentation we sometimes focus on the results for the Local Disk/Avoid (LD/A) and Server Memory/Detect (SM/D) algorithms, as these two algorithms are often sufficient to show the tradeoffs among the different approaches to consistency maintenance and hierarchy search order. In addition to the disk caching algorithms, we also show results for the Callback-Read (CB-R) algorithm, which does not utilize the client disk caches. CB-R is used as a baseline to help gauge the magnitude of the performance gain (or in a few cases, loss) resulting from the use of client disk caches.

4.1 Experiment 1: UNIFORM-WH Workload

We now turn to our first set of results, which were obtained using the UNIFORM-WH workload. In this workload (as shown in Table 2) all clients uniformly choose pages to access from the entire database. The pages in the first half of the database are accessed with a 10% write probability, while the pages in the other half are accessed read only. Figure 4 shows the distribution of page accesses among the four levels of the storage hierarchy for the LD/A and SM/D algorithms. Several trends can be seen in the figure. First, LD/A and SM/D obtain a similar (small) percentage of their pages from the client memory caches. Because the algorithms all use CB-R to manage memory caches and search the memory cache before looking elsewhere, they have similar memory cache hit rates in this experiment and in the ones that follow. Second, as would be expected, SM/D obtains more pages from the server memory and fewer pages from the local disk cache than LD/A throughout the range of client populations. Therefore, LD/A does more total (server and client) disk reads to get its pages than SM/D. Third, with one client, LD/A does not access any pages from the server memory, but then reaches a fairly stable level of access. As seen in previous studies [Dan90, Care91, Fran92a], the initial low server memory hit rate is due to correlation between the contents of the client's caches and the server memory — with few clients, the server memory largely contains copies of the pages that are in client memories, so the server memory is less effective for serving client misses than would otherwise be expected given its size. This correlation is damped out as more clients are added to the system. Fourth, and most importantly for this experiment, both algorithms initially obtain a similar number of pages from the server disk, but as clients are added SM/D obtains more pages from the server disk than LD/A. This is due to the differences in effective disk cache size discussed in Section 2.4.

The performance impact of these page distributions can be seen in Figure 5, which shows the throughput results for this workload when run with the fast network, and in Figure 6, which shows the corresponding transaction response times up to 20 clients[4]. First, comparing the performance of CB-R to that of the disk caching algorithms shows the magnitude of the performance gains obtained by introducing client disk caches. As would be expected, CB-R is hurt by its relatively high server disk demands as clients are added[5]. For the disk caching algorithms, the performance results show how the dominant algorithm characteristic changes as the number of clients in the system varies. In general, at 20 clients and beyond, the avoidance-based algorithms (LD/A and SM/A) perform similarly, and they are significantly better than the detection-based SM/D and LD/D. However, with small client populations the dominant characteristic is the hierarchy search order, with the server memory al-

[4] Note that we use a closed system model, so throughput and response time are equivalent metrics.

[5] With one client, CB-R actually has a slight performance advantage because its client memory cache does not have to store disk cache information.

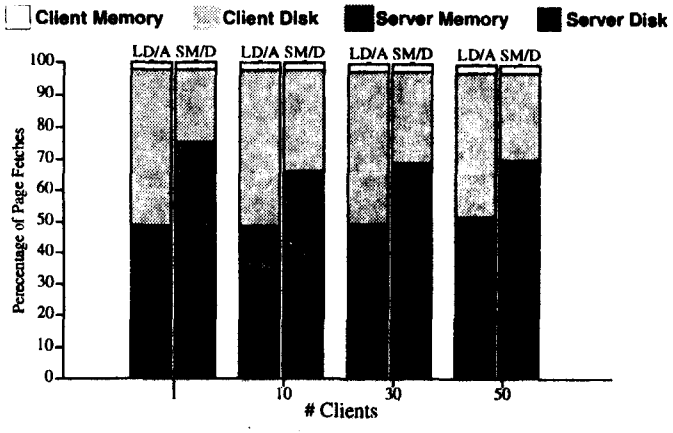

Figure 4: Page Access Distribution
(UNIFORM-WH Workload)

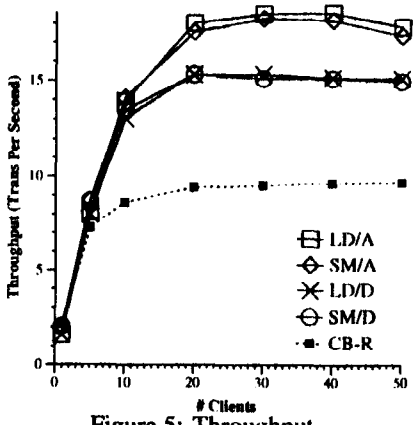

Figure 5: Throughput
(UNIFORM-WH, Fast Network)

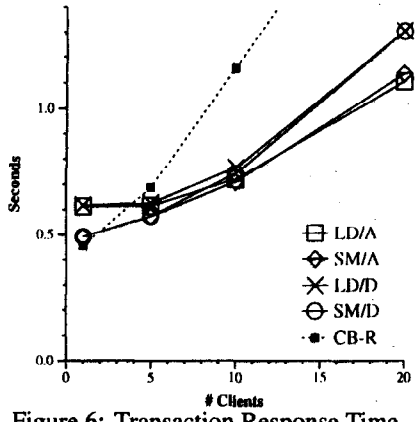

Figure 6: Transaction Response Time
(UNIFORM-WH, Fast Network)

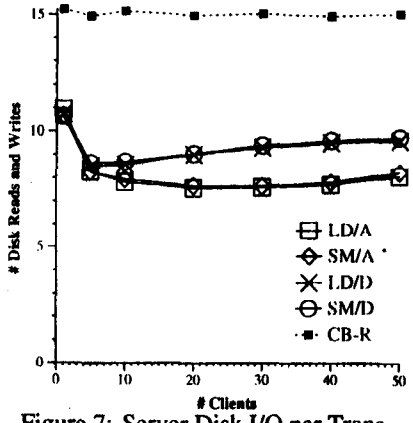

Figure 7: Server Disk I/O per Trans
(UNIFORM-WH, Fast Network)

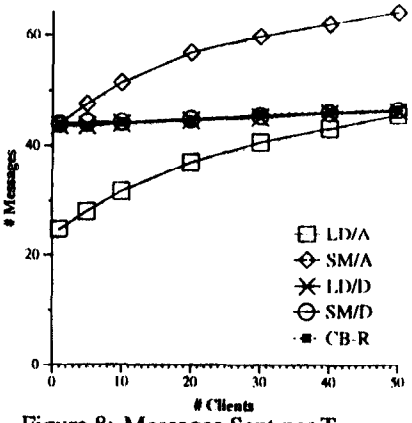

Figure 8: Messages Sent per Trans
(UNIFORM-WH, Fast Network)

gorithms (SM/D and SM/A) having slightly better performance than LD/A and LD/D.

The impact of the search order can be seen most clearly in Figure 6. In the range of 1 to 5 clients the Server Memory algorithms perform best. In this region, the server memory is less costly to access than a local disk cache because with small client populations, the (fast) network and the server are lightly loaded, and the fast network involves low on-the-wire costs for sending pages from the server to clients. The relative costs of accessing the server memory and accessing the local disk can be seen by comparing the performance of the two avoidance-based algorithms (LD/A and SM/A). The two algorithms perform the same amount of work to maintain the consistency of the disk caches and obtain the same number of pages from the client memory caches and the server disk. They differ only in the proportion of pages that they obtain from the local disk caches versus the server memory. With the fast network (Figures 5 and 6), the server memory is slightly less expensive up to 10 clients, beyond which the local disk caches are cheaper to access. When the slow network is used, (not shown) the network eventually becomes a bottleneck for the SM algorithms due to the volume of pages being sent from the server to the clients and thus, the server memory is even more expensive to access.

At 20 clients and beyond, the performance of the algorithms is dictated by the disk cache consistency approach — the avoidance-based algorithms dominate the detection-based ones. The reason for this can be seen in the number of server disk I/Os the algorithms perform per transaction (Figure 7). In this experiment, the server disk becomes the dominant resource, as all of the algorithms approach a disk bottleneck. As can be seen in the figure, the detection-based SM/D and LD/D algorithms lead to over 20% more disk I/Os than the avoidance-based algorithms beyond 20 clients. These additional I/Os are reads (as disk writes account for less than one I/O per transaction for all of the algorithms here). The extra server disk reads occur because the effective size of the client disk caches is substantially lower for the detection-based algorithms than for the avoidance-based ones. The reason that the effective disk cache size differences are so significant here is due to the uniform access pattern. With a uniform workload, all cache slots are equally valuable — there is no "working set" that can be kept cache-resident. Also, the uniformity of the workload results in a high degree of read-write sharing, so disk caches managed by the two detection-based algorithms will contain a large number of invalid pages.

Finally, it is important to note the message behavior of the algorithms. Figure 8 shows the number of messages sent per committed transaction. The message counts of the detection-based algorithms (and CB-R) remain fairly constant as clients are added. They each need to send a round trip message for each page accessed by a transaction. Some of these messages

are short control messages, while others (especially for CB-R) are large messages containing page values. In contrast, LD/A initially requires fewer messages, as it contacts the server only for pages that are absent from both of the local caches on a client. However, LD/A also requires invalidation messages to be sent to remote sites when a page is updated. The number of invalidation messages that must be sent increases with the number of clients in the system for this workload. The increase in invalidations also results in an increase in the number of pages that LD/A must request from the server. Finally, SM/A incurs the combined message costs of contacting the server on each page access and of invalidating remote pages; thus, it has the highest message costs among the algorithms studied.

4.2 Experiment 2: HOTCOLD Workload

The next workload that we investigate is the HOTCOLD workload, in which (as shown in Table 2) each client has a distinct 50-page read/write "hot range" of the database that it prefers, and the hot range of each client is accessed by all other clients as part of their cold range. This workload exhibits a high degree of locality, but also has a significant amount of read-write sharing as the client population is increased. As shown in Figure 9, the high locality of this workload allows the majority of pages to be obtained from each client's memory cache. The figure also shows that LD/A and SM/D obtain a similar, but small, percentage of their pages from the server disk. The differences between the two algorithms are evident in the way that they split the remaining accesses. With one client, LD/A obtains the remainder of its pages from client disk caches. As clients are added, the proportion of pages that LD/A obtains from the server memory increases somewhat. As discussed below, this occurs because adding clients increases read-write data sharing for this workload, and the effectiveness of the client disk caches decreases in the presence of such data sharing. SM/D shows different trends, with the portion of its pages coming from the server memory decreasing as clients are added. This decrease occurs because, in this type of workload, the server memory can hold fewer of the active hot set pages as the client population increases. Note that at a population of fifty clients, both algorithms have very similar page access distributions.

These distributions show the effects of the high level of read-write sharing in this workload, which increases as clients are added to the system. Regardless of the algorithm used, the update of a page at one site causes all copies of the page at other sites to become unusable. Thus, as the client population increases for this workload, the utility of the clients' disk caches decreases. For example, at 50 clients the LD/A algorithm has on average, over 1000 disk cache slots (out of 1250) empty as a result of invalidations — a disk cache larger than 250 pages will simply not be used by the avoidance-based algorithms in this experiment.

Turning to the throughput results for the fast network (shown in Figure 10) it can be seen that the advantages of using disk caches are lower in this experiment than in the previous one, particularly with large client populations. As in the UNIFORM-WH experiments, the Local Disk algorithms have the lowest performance for small client populations. At 20 clients, all four of the disk caching algorithms have roughly the same throughput, which is about 50% higher than that of CB-R. Beyond this

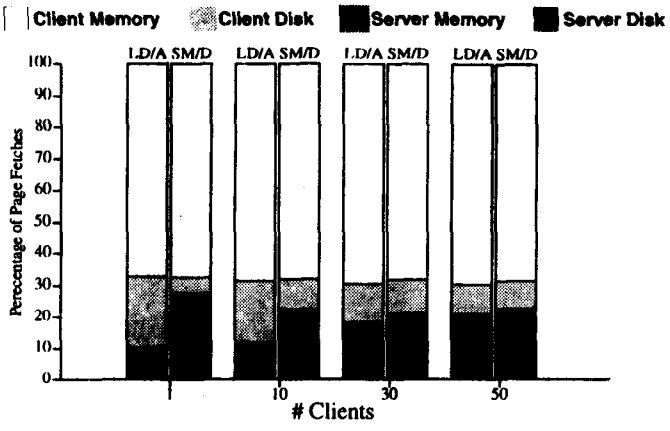

Figure 9: Page Access Distribution (HOTCOLD Workload)

point, however, the disk caching algorithms separate into two classes — and the detection-based algorithms out-perform the avoidance-based ones. This is the opposite of the ordering that was seen in the UNIFORM-WH case, and occurs despite the fact that at 20 clients and beyond, all four of the algorithms perform a similar amount of server disk I/O (Figure 11)[6]. The reason for this behavior is due to the extra work that LD/A and SM/A perform for invalidations. As shown in Figure 12, the avoidance-based algorithms send significantly more messages per transaction than the other algorithms. These additional messages are largely due to the invalidation of remote disk cache pages, the impact of which can be seen, for example, in the difference between the SM/A and SM/D lines in Figure 12. The additional invalidation activity results in increased transaction path length. Consequently, while the detection-based algorithms both eventually become server disk-bound, the avoidance-based algorithms never reach the disk bottleneck and their performance falls off at a faster rate. The results for the Slow network (not shown) are similar to these, although the Server Memory algorithms perform below the level of the Local Disk algorithms (but still above CB-R) in the range of 10 to 30 clients, due to the network cost of sending pages from the server to clients.

This experiment demonstrates the effect of a high degree of read-write sharing on the performance of the disk cache management algorithms and on the usefulness of client disk caching in general. This is demonstrated by the fact that the throughput for all four disk caching algorithms has a downward slope beyond 10 clients. As the level of read-write sharing is increased the disk caches become less effective. If the sharing level were high enough, then the client disk caches could actually harm performance — because they would not contribute any useful pages yet they require maintenance overhead. Therefore, it is clear that client disk caching will be most appropriate for environments in which there is a substantial amount of data that is not subject to a high degree of read-write sharing.

4.3 Experiment 3: PRIVATE Workload

We expect that many application environments for OODBMS will have a substantial amount of data that is private or has low data contention. In this section, we investigate the performance of the alternative disk caching algorithms using the PRIVATE

[6]Note that, unlike the previous case, server disk writes are an important component of the overall server I/O here.

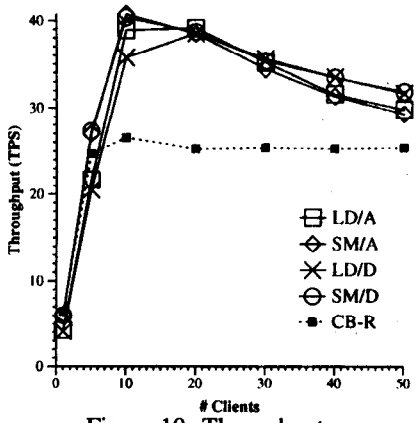

Figure 10: Throughput
(HOTCOLD, Fast Network)

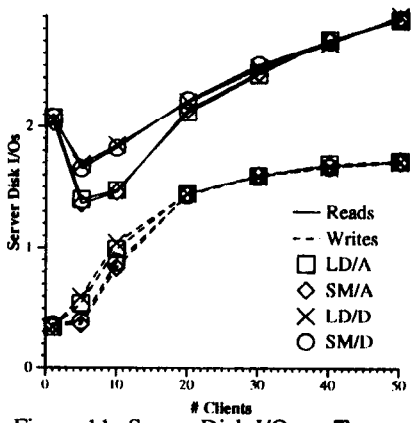

Figure 11: Server Disk I/O per Trans
(HOTCOLD, Fast Network)

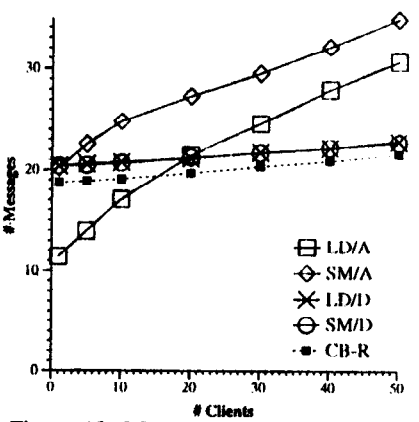

Figure 12: Messages per Transaction
(HOTCOLD, Fast Network)

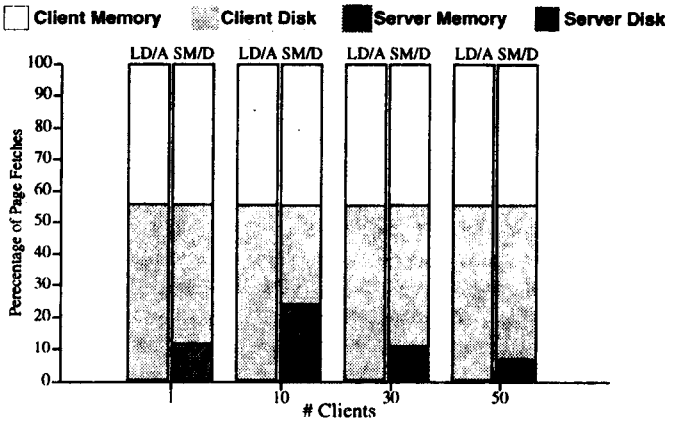

Figure 13: Page Access Distribution
(PRIVATE Workload)

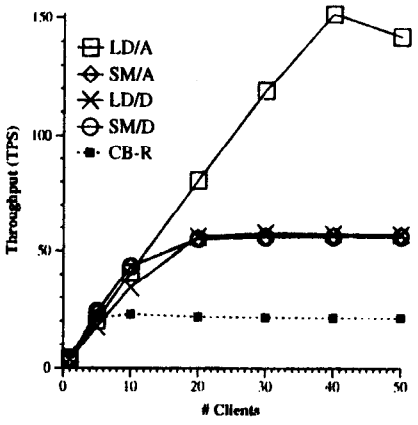

Figure 14: Throughput
(PRIVATE, Fast Network)

workload, which is intended to model a CAD or software engineering environment in which each client works on a separate part of the database while reading from a shared library. In this workload (as shown in Table 2), each client has exclusive read-write access to a 25-page region of the database, and all clients share the other half of the database in a read-only fashion; thus, none of the database is read-write shared.

As can be seen in Figure 13, this workload presents an excellent environment for client disk caching. The most notable aspect of this graph is that (after system start-up) no server disk reads are required. Moreover, LD/A accesses all of its pages locally at the clients. As would be expected, SM/D behaves differently. It has an initial decrease in its locally-accessed portion; but, beyond 10 clients the locally-accessed portion increases — at 50 clients over 90% of the SM/D page accesses are satisfied locally. This is due to server-client memory correlation effects as discussed for the UNIFORM-WH workload in Section 4.1. As clients are added to the system, the correlation dissipates and the server memory hit rate improves. Due to the skewed access pattern of the PRIVATE workload, however, when enough clients are added that their hot ranges no longer fit in the server memory, the hit rate at the server once again decreases.

The PRIVATE workload throughput results (Figure 14) show that the LD/A algorithm has substantial performance benefits over the other algorithms in this case. This is due to the effectiveness of the local disk caches, as described above, and because of LD/A's use of avoidance-based consistency management. In this workload, there is no read-write sharing, so no invalidation requests are required. Consequently, the avoidance-based algorithms get consistency virtually for free, while the detection-based algorithms must still check with the server on every initial access. As a result, LD/A sends only 6.5 messages per committed transaction, while the other algorithms send over 23 messages per transaction. The combination of local access and cheap consistency maintenance allows LD/A to scale almost linearly up to 40 clients here, while the other disk caching algorithms all bottleneck at 20 clients. Also, note that for this workload, CB-R flattens out at only 5 clients, at a throughput level that is less than 40% of the peak LD/D, SM/A, and SM/D throughput and less than 15% of the peak throughput of LD/A.

In this experiment, all of the disk caching algorithms except for LD/A become bottlenecked at the server CPU due to messages, while LD/A is ultimately bottlenecked at the server disk. Recall that in this experiment, LD/A does not perform any reads from the server disk; the disk bottleneck is caused by *write* I/Os. The server write I/Os occur because pages that are dirtied by transactions are copied back to the server at commit time. These pages must eventually be written to the server disk when they are aged out of the server's memory. When the slow network is used (not shown), the copying of dirty pages back to the server also hurts the scalability of LD/A. In this case, however, the dirty

pages cause the network (rather than the server disk) to become the bottleneck. In Section 5.1 we study ways to reduce this cost.

4.4 Result Summary

In this section we review the main results of the preceding performance study. First, it should be noted that in all but a few cases, client disk caching provided performance benefits over the memory-only caching CB-R algorithm. The three workloads used in the study brought out different tradeoffs among the algorithms for managing client disks. In the workloads with read-write sharing (UNIFORM-WH and HOTCOLD), we saw that with small client populations, the dominant algorithm characteristic was the hierarchy search order, with the server memory first algorithms having a slight advantage over the local disk first algorithms. With larger populations, however, the disk cache consistency maintenance approach was dominant for these workloads. For UNIFORM-WH, the avoidance-based algorithms performed best because they resulted in a larger effective disk cache size. With the fast network, there was little difference between the two avoidance-based algorithms (LD/A and SM/A), but when the slower network was used, LD/A outperformed SM/A because the local disk caches were less expensive to access than the server memory. For the HOTCOLD workload, the high level of read-write sharing in the presence of large client populations reduced the usefulness of the client disk caches, and it caused the avoidance-based algorithms to perform slightly worse than the detection-based ones because of higher message requirements. The PRIVATE workload, which has high per-client locality and no read-write sharing was seen to be an excellent workload for client disk caching. For this workload, the LD/A algorithm performed far better than the others when 20 or more clients were present in the system. This is because its bias towards using the local disk first allowed it to scale and its use of avoidance for disk cache consistency maintenance allowed it to ensure consistency virtually for free (due to the absence of read-write sharing). In fact, LD/A scaled nearly linearly with the number of clients until the server disk became a bottleneck due to writes.

5 Algorithm Extensions

In this section, we address two additional performance enhancements for client disk caching: 1) reducing the overhead caused by copying updated pages to the server at commit-time, and 2) ways to reduce the expense of maintaining large caches that are not currently in use.

5.1 Reducing Server Overhead

As stated in Section 1.1, the server is the natural location at which to guarantee transaction semantics. The policy of copying dirty pages to the server at commit time simplifies the implementation of the server's ownership responsibilities. For example, it allows the server to easily produce the most recent committed copy of a page when it is requested by a client. However, the results of the previous experiments (particularly for the PRIVATE workload) show that for certain workloads this simplicity comes at a cost in performance and scalability. In this section, we examine the complexity and the potential performance gains that result from relaxing the commit-time page send policy. While we are unaware of any work in that has addressed this issue for client-server DBMS, it should be noted that similar issues can arise when transferring pages among the processing nodes of shared-disk transaction processing systems [Moha91, Dan92].

In the following discussion, we assume a system that uses a write-ahead-logging (WAL) protocol [Gray93] between clients and the server. Therefore the server is always guaranteed to have all of the log records required to reconstruct the most recently committed state of all database pages. A description of the implementation of such a protocol (i.e., ARIES [Moha92]) for a client-server DBMS can be found in [Fran92c].

5.1.1 Consequences of Retaining Dirty Pages

Relaxing the commit-time page send policy places certain constraints on the operation of clients. If clients are allowed to commit transactions without copying updated pages to the server, then a client may have the only copy of the most recent committed value of a page. Clients can not freely dispose of such pages. In contrast, under the commit-time page send policy clients are free to retain or drop updated pages after a transaction commits. Furthermore, allowing a client to overwrite the only valid copy of a page complicates the implementation of transaction abort; either clients will have to perform undo (which implies that they need sophisticated log management), or affected pages will have to be sent to the server to be undone.

Relaxing the policy also has implications for the operation of the server. The server must keep track of which client (if any) has the current copy of each page, and to satisfy a request for a page the server may have to obtain the most recent copy of a page from a client. This facility can be added to our existing algorithms by using the Callback-Write (CB-W) algorithm as a basis rather than Callback-Read. As described in [Fran92a], CB-W is a callback locking algorithm that allows clients to retain *write* (as well as read) permission on pages across transaction boundaries. When the server receives a request for a page that is cached with write permission at a client, the server sends a request to the client, asking it to downgrade its cached write permission to read permission. If the commit-time page send policy is enforced, the client needs only to acknowledge the downgrade to the server, the server can then send its own copy of the page to the requester. If the policy is relaxed, CB-W must be changed so that in response to a downgrade request, a client will send a copy of the page to the server along with the acknowledgement. When the server receives the new page copy it installs it in its memory cache and sends a copy to the requester.

This scheme works, of course, only if clients are always available to respond to downgrade requests from the server. Under the commit-time page send policy, the server has the option of unilaterally deciding that a client's cache contents are invalid and deciding to abort outstanding transactions from a non-responsive client. Without the commit-time page send policy, however, this could result in the loss of committed updates. In a system that uses write-ahead-logging this problem can be solved by taking advantage of the server's log. To do so, however, requires an efficient way of performing redo on individual pages while normal processing is underway. One possible implementation would be to link all of the log records pertaining to a particular page backwards through the log, and to have the server keep track of the most recent log record for each outstanding page. If a page is needed from an unresponsive client, the

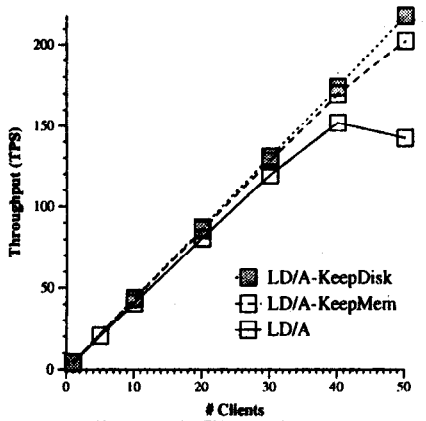

Figure 15: Throughput (PRIVATE, Fast Network)

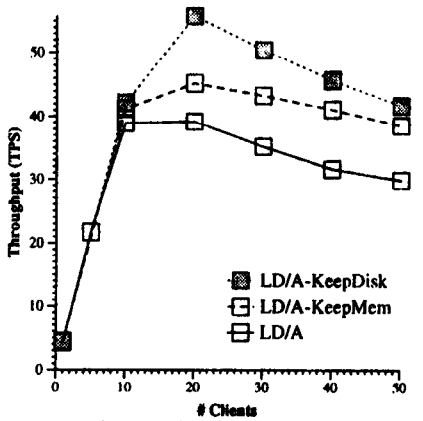

Figure 16: Throughput (HOTCOLD, Fast Network)

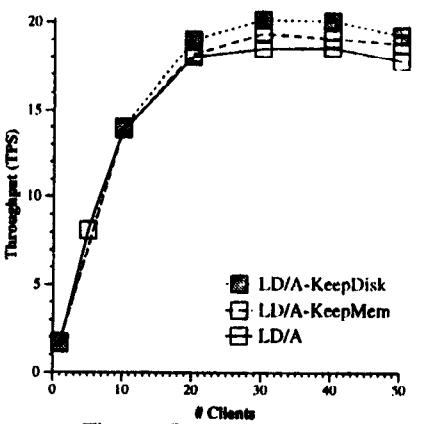

Figure 17: Throughput (UNIFORM-WH, Fast Network)

server can then perform redo processing on its copy of the page by scanning backwards through the linked list of log records for the page to find the first missing update. It can then process those records in the forward direction, redoing the missed updates.

A related problem has to do with log space management. The log is typically implemented as a circular queue in which new records are appended to the tail and records are removed from the head when they are no longer needed for recovery. A log record can be removed from the head of the log if the transaction that wrote the record completed (committed or aborted) and the copy of the corresponding page in stable storage is correct with respect to the logged update and the outcome of the transaction[7]. When executing transactions, there is a minimum amount of free log space that is required in case recovery needs to be performed. If the required free space is not available, then transactions must be aborted. If clients are allowed to retain dirty pages indefinitely, then the server may be unable to garbage collect its log, resulting in transaction aborts. Therefore, dirty pages must still be copied back to the server periodically. This can be done by having clients send back updated copies of pages that exceed a certain age threshold, or by having the server send requests for copies of pages that will soon impact its ability to garbage collect the log. Regardless of which method is employed, as a last resort the server can always apply selective redo to particular pages in order to free up log space.

5.1.2 Potential Performance Gains

It should be clear that substantial complexity must be incurred in order to relax the commit-time page send policy. In this section we examine the performance improvements that could result from relaxing this policy. We extended the Local Disk/Avoid algorithm to allow pages updated by transactions to remain dirty at clients. We consider two extensions: *LD/A-KeepMem*, which allows updated pages to remain dirty in a client's memory cache, but sends copies to the server when the page is demoted to its disk cache, and *LD/A-KeepDisk*, which allows dirty pages to reside in the disk cache as well as in memory. In both extended algorithms, a copy of a page is sent to the server in response to a downgrade request for the page, after which the client's copy of the page is no longer considered dirty.

[7]For a committed transaction the stable copy of the page must reflect the logged update. For an aborted one, the stable copy must *not* reflect the logged update.

Figures 15- 17 show the throughput of the original and extended LD/A algorithms for the three workloads studied in Section 4. In these experiments we did not model the periodic copying of dirty pages to the server for log space reclamation, and we did not attempt to study the impact of client failures on the performance of the server. As a result, the performance gains shown in the figures are upper bounds for what could be expected (this is particularly true for the LD/A-KeepDisk algorithm). The throughput results for the PRIVATE workload show that, as expected, relaxing the commit-time page send policy can avoid the performance bottleneck that LD/A hits beyond 40 clients. LD/A-KeepMem performs about 33% fewer disk writes with 50 clients than does the original LD/A algorithm. The LD/A-KeepDisk algorithm performs no disk writes in this experiment, and thus, it scales linearly within the range of client populations studied here. Unfortunately, the results for LD/A-KeepDisk are unrealistic, because as mentioned above, pages do eventually have to be sent back to the server to allow log space to be reclaimed (and to minimize the performance impact of a client failure). However, a reasonable implementation of LD/A-KeepDisk should perform fewer server disk writes than LD/A-KeepMem does here, and thus, could still approach linear scale-up within this range of client populations.

The throughput results for the HOTCOLD workload (Figure 16) also show a performance gain from relaxing the commit-time page send policy. LD/A-KeepMem and LD/A-KeepDisk each benefit from a reduction in both server disk writes and server disk reads in this case. The reduction in server disk reads is due to an increase in the server memory hit rate. As it turns out, the commit-time page send policy hurts the server memory hit rate because the server memory becomes filled with copies of dirty pages that are also cached at clients (this phenomenon was also observed in [Fran92b]). In this experiment, LD/A-KeepDisk reaches a bottleneck at the server CPU due to messages sent for invalidations, and thus, its performance falls steeply beyond 20 clients. LD/A-KeepMem has similar message requirements, but it also has somewhat higher server disk I/O requirements, and thus it performs at a lower level than LD/A-KeepMem. Similar effects on server disk reads and writes also occur for the UNIFORM-WH workload (Figure 17). In this case, however, there is a smaller performance benefit to relaxing the policy. In this workload, the probability of amortizing writes due to relaxing the policy is low. This is due to the lack

of per-client locality, which makes it likely that a page will be accessed at another client or dropped from memory (in the case of the KeepMem algorithm) before it is rewritten at a client. As a result, the LD/A-KeepMem and LD/A-KeepDisk algorithms do not provide a reduction in disk writes for this workload; the gains are the result of a slight reduction in server disk reads.

5.2 On-line vs. Off-line Caches

Up to now, we have focused on systems in which all of the clients actively use the database. In an actual environment, however, we would expect clients to go through periods of activity and inactivity with respect to the database. In other words, clients may be "on-line" or "off-line". One aspect of client disk caching that we have not yet addressed is how to treat the data cached at a client when the client is off-line. If the cache contents are retained across an off-line period, then when a client comes back on-line, the cache will already be in a "warm" state. The retention of client cache contents across off-line periods is particularly important for disk caches, as large disk cache contents would be very expensive to re-establish from scratch when reactivating the database system. Also, due to the low cost and non-volatility of disk storage, it is inexpensive and simple to allow the disk cache contents to persist through an off-line period. One problem that must be addressed, however, is to ensure that upon re-activation the client will not access cached data that has gone stale during the off-line period. If a detection-based approach to cache consistency maintenance is used, then the retained cache can be used as is. But, as shown in the performance results of Section 4, avoidance-based protocols are the recommended approach, and under such protocols, steps must be taken to ensure the validity of the cache contents after an off-line period.

The simplest approach is to have each client field invalidation requests, even during "off-line" periods. This solution requires, of course, that clients stay connected to the server. The cost of this approach is that a client process with access to the cache management data structures must be active during "off-line" periods, and invalidation messages for the pages in the inactive caches will still have to be sent by the server and processed by the off-line clients. Such overhead is likely to be acceptable in many environments. If, however, disconnection is likely during the off-line period, or if database system overhead during off-line periods is undesirable, then there are several ways to extend the consistency maintenance techniques of Section 2.3.2 to allow cache consistency to be re-established when a client re-activates its local database system.

When a client wishes to go off-line with respect to the database, it must first return any dirty page copies to the server, save its disk cache control information on disk, and then inform the server that it is going off-line. An *incremental* approach to re-establishing cache consistency can be easily implemented by combining detection and avoidance techniques. As part of the process of going off-line, a client marks all of its disk cache pages as "unprotected". When a client is on-line, it must treat any unprotected cached page as if a detection-based consistency maintenance algorithm was being used, checking validity of the page with the server. Once the validity of a page is established, the page is marked as "protected" and thereafter, can be accessed using the normal avoidance-based protocol.

An alternative approach to re-establishing cache consistency is to have the server keep track of updates to the pages that are resident in off-line caches. When an off-line client wishes to come back on-line, it sends a message to the server, and the server responds with information that allows the client to re-establish the validity of its cache contents. This information can be: 1) a list indicating the pages to invalidate, 2) actual log records for the updates that were applied to the off-line pages by other clients, or 3) copies of the pages that changed during the off-line period. The tradeoffs between option 1 and the others are similar to those that arise when deciding between invalidation or propagation of updates for on-line caches (as discussed in Section 2.3.2). Therefore, we expect that sending a list of invalidations will typically have the best performance.

The incremental approach has the advantage that clients can begin processing immediately after coming on-line; however, it has the disadvantage of having to contact the server for validity checks (assuming the use of the LD/A algorithm, for example), so performance may be reduced during the initial period after coming back on-line. In contrast, the all-at-once approach has the advantage that the disk caches are quickly cleared of invalid data, resulting in a larger effective disk cache. Also, less communication with the server is required to re-establish to cache consistency than with the incremental approach. The disadvantages of the all-at-once approach are that the server needs to perform bookkeeping for off-line clients, and that there will be a delay between the time that a client comes back on-line and the time that it can begin processing database transactions.

Several projects have performed work on deferred consistency maintenance that is related to the work here. The ADMS± system [Rous86] uses an incremental method to update query results that are cached at clients. Updates are performed at clients prior to executing a query at a client. As discussed in Section 1.1, the performance of a similar scheme is studied in [Deli92]. Techniques to reduce update overhead for caches in information retrieval systems are addressed in [Alon90]. These techniques are based on *quasi-copies*; i.e., copies whose values are allowed to diverge from the value of the primary copy. Finally, distributed file systems that support disconnected operation, such as CODA [Kist91] and Ficus [Guy91], must address issues of validating local caches after a disconnected period. Such systems do not support transaction semantics, however, and require user-interaction to resolve some classes of conflicts.

6 Conclusions and Future Work

In this paper we have demonstrated that client disks can provide substantial performance benefits when employed in an extended cache architecture. We described and studied four alternative algorithms. The performance study showed that for small client populations, the server memory was typically less expensive to access than the local client disk caches, but that this order inverted as clients were added to the system. In terms of disk cache consistency maintenance, algorithms that avoid access to stale data through the use of invalidations were found, in most cases, to perform better than algorithms that detect access to stale pages. This was due primarily to a larger effective disk cache size for avoidance-based algorithms. As expected, however, under high levels of read-write sharing, the larger number of messages due to consistency maintenance caused the performance of the avoidance-based algorithms to suffer somewhat.

However, in the cases where avoidance was seen to perform worse than detection, the relative advantage of disk caching in general was low for all of the algorithms. We expect that disk caching will have the greatest benefits in configurations with large client populations and low levels of read-write sharing. For such envrionments, Local Disk/Avoid (LD/A) appears to be the most promising of the four algorithms. Dynamic extensions of LD/A can be developed to better handle cases with small client populations.

The effectiveness of client disk caching in reducing the demand for server disk reads resulted in an increase in the relative impact of server disk writes on performance. To address this, we investigated ways of relaxing the policy of sending copies of updated pages to the server at commit-time. Although relaxing the policy has serious complexity implications for database system implementation, it appears that many of the problems can be solved by extending standard write-ahead-logging techniques. Using simple extensions of the LD/A algorithm, we found that the potential performance and scalability benefits of allowing updated pages to remain dirty in client memory caches and/or disk caches can be quite high. Another issue raised by the large capacity of client disk caches is the importance of preserving the cache contents across periods of database system inactivity. We described several approaches that allow clients to re-establish the validity of their cache contents after an "off-line" period.

The use of client disk caches is related to our earlier work on global memory management [Fran92b], as both techniques utilize client resources to offload the server disk. Client disk caching is likely to be easier to add to an existing system because the disk is treated as an extension of the memory cache, while global memory management requires new communication paths. Also, client disk caching is likely to be more effective than global memory in situations where clients access primarily private data. In contrast, global memory may be more useful for situations in which much data is shared among the clients — particularly if clients often read data that is written by other clients. The two techniques are complementary and could be integrated in a single system. To do so, however, several interesting performance issues will need to be addressed. For example, it is not obvious where to place the various remote client resources in the hierarchy search order. This is one avenue for future work.

For longer-term future work, we plan to investigate the opportunities raised by the non-volatility of client disks. In particular, we plan to investigate disconnected operation, as the off-line/on-line issues raised in Section 5.2 will be of prime importance in an environment where clients can disconnect from the rest of the database system. We also plan to study the implementation of logging and crash recovery algorithms that enable the relaxation of the commit-time page send policy.

References

[Alon90] R. Alonso, D. Barbara, H. Garcia-Molina, "Data Caching Issues in an Information Retrieval System", *ACM TODS*, 15(3), 1990.

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

[Dan90] A. Dan, D. Dias, P. Yu, "The Effect of Skewed Data Access on Buffer Hits and Data Contention in a Data Sharing Environment", *Proc. 16th VLDB Conf.*, Brisbane, Australia, Aug., 1990.

[Dan92] A. Dan, P. Yu, "Performance Analysis of Coherency Control Policies through Lock Retention", *Proc. ACM SIGMOD Conf.*, San Diego, June, 1992.

[Deli92] A. Delis, N. Roussopoulos, "Performance and Scalability of Client-Server Database Architectures", *Proc. 18th VLDB Conf.*, Vancouver, Canada, Aug., 1992.

[Deux91] O. Deux *et al.*, "The O2 System", *Communications of the ACM*, 34(10), Oct., 1991.

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

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

[Fran92b] M. Franklin, M. Carey, and M. Livny, "Global Memory Management in Client-Server DBMS Architectures", *Proc. 18th VLDB Conf.*, Vancouver, B.C., Canada, Aug., 1992.

[Fran92c] M. Franklin, M. Zwilling, C. Tan, M. Carey, and D. DeWitt, "Crash Recovery in Client-Server EXODUS", *Proc. ACM SIGMOD Conf.*, San Diego, June, 1992.

[Gray93] J. Gray, A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Mateo, CA, 1993.

[Guy91] R. Guy, "Ficus: A Very Large Scale Reliable Distributed File System", *Ph.D. Dissertation, UCLA TR CSD-910018*, June, 1991.

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

[Kim90] W. Kim, *et al.*, "The Architecture of the ORION Next-Generation Database System," *IEEE TKDE*, 2(1), Mar., 1990.

[Kist91] J. Kistler, M. Satyanarayanan, "Disconnected Operation in the Coda File System", Proc. 13th SOSP Conf., Oct., 1991.

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

[Livn88] M. Livny, *DeNet User's Guide*, Version 1.0, Comp. Sci. Dept., Univ. of Wisconsin-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", *Proc. 17th VLDB Conf.*, Barcelona, Sept., 1991.

[Moha92] C. Mohan, *et al.*, "ARIES: A Transaction Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", *ACM TODS*, 17(1), Mar., 1992.

[Nels88] M. Nelson, B. Welch, J. Ousterhout, "Caching in the Sprite Network File System", *ACM TOCS* 6(1), Feb., 1988.

[Obje91] Objectivity Inc., *Objectivity/DB Documentation V 1*, 1991.

[Onto92] ONTOS Inc., *ONTOS DB 2.2 Reference Manual*, 1992.

[Ston81] M. Stonebraker, "Operating System Support for Database Management", *CACM*, 24(7), 1981.

[Rous86] N. Roussopoulos, H. Kang, "Principles and Techniques in the Design of ADMS+-", *IEEE Computer*, Dec., 1986.

[Trai82] I. Traiger, "Virtual Memory Management for Database Systems", *Operating Systems Review*, 16(4), Oct., 1982.

[Vers91] Versant Object Technology, *VERSANT System Reference Manual, Release 1.6*, Menlo Park, CA, 1991.

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

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