Global Memory Management in Client-Server DBMS Architectures

Michael J. Franklin
Michael J. Carey
Miron Livny

Technical Report #1094

June 1992

An abridged version of this paper appears in:
The Proceedings of the 18th International Conference on Very Large Data Bases, Vancouver, BC, Canada, August 1992.

Global Memory Management in Client-Server DBMS Architectures

Michael J. Franklin
Michael J. Carey
Miron Livny

Computer Sciences Technical Report #1094
June 1992

Computer Sciences Department
University of Wisconsin-Madison

This work was partially supported by the Defense Advanced Research Projects Agency under contract DAAB07-92-C-Q508, by the National Science Foundation under grant IRI-8657323, and by a research grant from IBM Corporation.

Global Memory Management in Client-Server DBMS Architectures

Michael J. Franklin, Michael J. Carey, Miron Livny

Computer Sciences Department
University of Wisconsin - Madison

ABSTRACT

Earlier performance studies of client-server database systems have investigated algorithms for caching locks and data at client workstations to reduce latency and offload the server. These studies have been restricted to algorithms in which database pages that were not in the local client buffer pool or the server buffer pool were read in from disk. In this paper we investigate a technique that allows client page requests to be serviced by other clients, thus treating the entire system as a single memory hierarchy. We also present techniques for efficiently exploiting this global memory hierarchy by reducing the replication of pages between client and server buffer pools. Global memory management algorithms that employ various combinations of these techniques are then described, and the performance tradeoffs among the algorithms are investigated under a range of workloads and system configurations using a simulation model.

1. INTRODUCTION

Rapid improvement in the price/performance characteristics of workstations, servers, and local-area networks has enabled sophisticated database function to be migrated from machine rooms to desktops. As a result, networks of high-performance workstations and servers have become an important target environment for the current generation of commercial and prototype database systems. The workstation environment provides a new set of performance opportunities and challenges for the design of database systems. One important attribute of such an environment is the presence of a complex memory hierarchy comprising local workstation memory, remote workstation memories, server memory, and disks. Efficient exploitation of the various levels of the hierarchy is necessary in order to attain high performance for large database systems in such an environment.

Database systems intended for a workstation environment are implemented using a *client-server* software architecture. A client-server DBMS is divided into two types of processes. Client processes execute on workstations and provide interaction with user applications. Server processes typically execute on shared server machines and provide access to the database in response to requests from multiple clients. These requests may be high level queries or requests for specific data items. Systems that interact via queries and results are referred to as *query-shipping* systems, while those that interact via lower-level requests for data items are referred to as *data-shipping* systems. Most relational database systems have adopted a query-shipping approach, while object-oriented database systems (OODBMSs) have typically been based on a data-shipping approach. A potential advantage of data-shipping approaches is

This work was partially supported by the Defense Advanced Research Projects Agency under contract DAAB07-92-C-Q508, by the National Science Foundation under grant IRI-8657323, and by a research grant from IBM Corporation.

the ability to exploit the resources of the workstations on which client processes run, since such architectures execute much of the database system functionality on the clients. This is important because, while it is likely that a server machine will have more memory and CPU power than any single workstation, the majority of the total memory and processing power in the network is likely to reside at the workstations. Data-shipping architectures can be further categorized into *page servers*, which transfer physical units (e.g., pages or segments) of data among clients and servers, and *object servers*, which interact using logical units of data such as objects or tuples. Performance tradeoffs among several data-shipping approaches are examined in [DeWi90].

Many recent client-server database systems (e.g. ObServer [Horn87], ObjectStore [Lamb91], O2 [Deux91], and client-server EXODUS [Exod91, Fran92b]) utilize a page server architecture. In these systems, clients interact with servers by sending requests for specific database pages or groups of pages. The server then provides the requested pages back to the client. The server is also responsible for providing transaction support, such as concurrency control and recovery, as well as other shared services for the database. For concreteness, this paper concentrates on client-server database systems that utilize a page server approach. However, many of the results presented are applicable in both the page server and object server contexts. We also focus on systems with a single server, both in our descriptions and our experiments.

1.1. Performance of Page Server Systems

Recently, there have been several studies of the performance aspects of caching algorithms for page server systems [Wilk90, Care91, Wang91, Fran92a]. These studies have shown the advantages and potential pitfalls of attempting to offload servers by caching locks and/or data at client workstations across transaction boundaries. Such caching is referred to as *inter-transaction* caching. Inter-transaction lock caching allows workstations to avoid sending lock request messages to the server; if a workstation already has the proper lock cached for an object, it can access the object without first requesting permission from the server. This reduction in lock requests results in reduced workstation and server CPU requirements for processing messages, reduced access to the lock manager on the server, and reduced competition for the network itself. Thus, lock caching was shown to have two beneficial effects: First, latency was improved due to the reduction in the overall pathlength for transactions. Secondly, the processing load on the server was lessened, thereby alleviating a potential bottleneck in the system. Two basic approaches have been studied for implementing lock caching. Algorithms with an optimistic component allow clients to access data based on cached locks, but require a validation phase prior to commit. Such algorithms were shown to be effective at reducing the number of messages sent by transactions, but were prone to higher abort rates and wasted work in high data contention situations. The other approach is to have the server "call back" cached locks when a conflicting lock request is received. Algorithms using the callback technique were found to incur a slightly higher cost in messages than the optimistic approach, but had fewer aborts and were more robust in the presence of higher data contention.

In addition to showing the potential performance benefits of lock caching in many workloads, the above studies showed that inter-transaction data caching could provide even more significant performance benefits by offloading the server disk in addition to reducing messages and CPU requirements. Data

caching effectively increases the size of the server's buffer pool by extending it to each client's workstation. This extended buffer pool was found to be particularly effective in the presence of locality (i.e., affinity of clients for particular pages). In all of the studies mentioned above, each client used a three-level memory hierarchy consisting of: 1) the local workstation's memory, 2) server memory, and 3) server disk. Thus, each client workstation had access to only a fraction of the total memory in the system. As a result of this, the studies showed that in many cases, significant disk I/O was required even though the aggregate memory of the system was as large or larger than the portion of the database being accessed by the workstations. Furthermore, two additional inefficiencies of this type of data caching were identified in [Care91]. First, when small numbers of workstations were present, there was often a high correlation between the pages resident in the server buffer pool and those resident in the client buffer pools. This correlation reduced the effectiveness of the server buffer pool, as buffer misses at clients often resulted in buffer misses at the server. Secondly, it was shown that with large numbers of clients, each with a fairly large buffer pool, excessive replication of pages in client buffer pools could lead to significant overhead for updates. In some of the cases examined, this overhead even outweighed the performance gains of inter-transaction data caching. Therefore, in most of the workloads examined, algorithms that insured consistency by invalidating remote copies of pages on updates (hence, reducing replication) outperformed algorithms that preserved replication by propagating changes to remote copies.

1.2. Opportunities for Improvement

While the data-caching techniques used in these earlier studies were effective in many situations, they were ultimately limited by their primarily *local* nature. Performance was hindered since clients were unable to exploit a large portion of the memory available in the system and since the memory that was available was not efficiently utilized. However, in all of the algorithms that allowed caching of both data and locks, (which typically performed better than algorithms that cached only data), the server was required to have knowledge of the location of all copies of pages in the system. This information provides an opportunity to improve upon the previous techniques through the use of a *global* approach to memory management. In this paper we investigate the tradeoffs involved with three specific global memory management techniques. First, the clients are allowed to exploit the entire memory of the system by obtaining pages from other clients. Second, buffer replacement policies at the server are modified to reduce the replication of the buffer pool contents of the server and its clients. Third, a simple protocol between clients and servers is used to extend the client buffer pools by moving some of the pages that are forced out of a client's buffer pool into the server's memory.

1.3. Overview of the Paper

In the following sections, we propose a set of algorithms that utilize these three techniques. The implementation of these algorithms requires no additional information at clients and servers beyond what is already required for the lock and data caching algorithms of the studies described previously. In particular, these global memory management algorithms are designed to tolerate imperfect (but conservative) server knowledge of page copy locations. After describing these algorithms and the environment in which they are to be implemented, we present the results of simulation experiments that investigate the performance characteristics and tradeoffs of the three global memory management techniques.

The remainder of the paper is structured as follows: Section 2 details the techniques used for global memory management and a set of algorithms that utilize these techniques. Section 3 describes the simulation model used to investigate the tradeoffs among these techniques over a range of system configurations and workloads. Section 4 presents a series of experiments and their results. Section 5 discusses related work. Finally, Section 6 presents our conclusions and plans for future work.

2. GLOBAL MEMORY MANAGEMENT

In this section, we discuss three techniques for implementing global memory management in a page server environment and outline five specific algorithms that use different combinations of these techniques. We also provide an overview of the expected performance tradeoffs among the different techniques. In order to enable a clear discussion of these techniques and algorithms, we first briefly present a reference architecture for a page server system.

2.1. Page Server Architecture and Assumptions

An example instantiation of a page server DBMS is shown in Figure 1. The system consists of a single server machine and a number of client workstations connected over a local-area network (e.g., an Ethernet). Each client has memory that is available for use as a buffer pool by the database system. The server typically has more available memory and more processing power than any of the clients. The server also has disks on which the permanent copy of the database resides and a (possibly mirrored) disk for the recovery log. In this reference architecture, we assume that the database system does not use client disks for logging or buffering and thus, clients are shown as diskless. The use of client disks, if present, is an interesting area for future work, but it is beyond the scope of the current study.

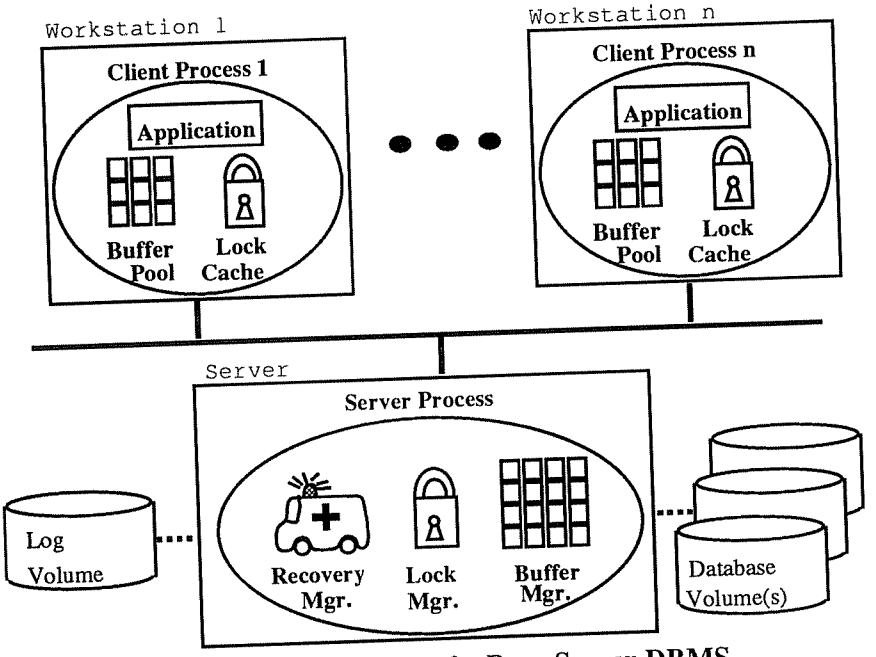

Figure 1: Architecture of a Page Server DBMS

The database system software consists of client database processes that execute at workstations and a server database process that runs at the server machine. A client application accesses the database by making requests to its local client database process. The method of interaction between applications and client database process is left unspecified. They may be linked as a single process or they may be separate processes communicating, for example, via local IPCs or shared memory. A client database process manages a single active transaction for its application at a time, but it is also capable of handling requests from the server and remote transactions. The client database processes communicate with each other and with the server database process by sending messages over the network. Such communication can be initiated asynchronously by either party and is used to handle database access, update propagation, concurrency control, and transaction management functions. For the remainder of the study we assume that single pages are the unit of buffering and locking. Also, except where noted, we assume that an LRU page replacement algorithm is used for each of the buffer pools in the system.

As stated previously, the caching studies described in Section 1 all treated such a system as a three-level memory hierarchy, thus limiting the size of the memory from which to satisfy the page requests of any one client to a fraction of the total memory available to the database system. The goal of the global memory management techniques investigated in this study is to exploit the remaining fraction in an *opportunistic* way. That is, as in the previous caching algorithms, the contents of any one client's memory are dictated by the accesses made by that client, but in addition, those contents (if not exclusively locked) can also be sent to other clients to satisfy their local cache misses. The techniques also attempt to make better use of the server memory in light of this new capability.

The ability to exploit the contents of remote client memory results in a four-level memory hierarchy. The level closest to the client is the *local client memory*, which can be directly accessed by a client database process. The second level of the hierarchy is the *server memory*, which is managed by the server database process. In terms of response time, this memory costs one small message from the client to the server (for the page request) and one large message (containing the page itself) from the server to the requesting client. Messages incur costs not only for their actual on-the-wire time, but also for CPU processing at both the sender and the receiver. The third level of the hierarchy is *remote client memory*. The server is the only site with knowledge of where page copies are cached in the system, so access to this level of the hierarchy must go through the server. Therefore, access to remote client memory costs two small messages and one page-sized message: the client first sends a small message to request the page from the server, the server then forwards that request to another client, and the remote client sends a large message containing the page to the requesting client.[1] Finally, the fourth level of the hierarchy is the server's disk. An access to this level of the hierarchy is the most expensive, costing one small message and one page-sized message as well as one or two disk accesses. (Two disk accesses are required if a dirty page must first be forced from the server's buffer pool in order to make room for the requested page to be read in from disk). In general, the goal of the global memory management techniques studied here

[1] We do not require perfect knowledge of page copy locations at the server, and therefore, there is a small possibility that the remote site will not be able to forward the requested page. Handling of this situation results in an extra message and possibly a disk access, as will be explained in Section 2.3.2.

is to move accesses from the lowest (and most expensive) level of the hierarchy to the higher levels. In particular, the techniques will attempt to convert what would have been disk accesses in a non-global scheme into cheaper accesses to the server memory or to remote client memories.

Up to this point, the discussion of memory management issues has been concerned primarily with the use of memory to avoid disk reads. However, in order to provide durability for the updates of committed transactions, the pages containing these updates must eventually be written to stable storage. In a client-server system, the server is responsible for ensuring the durability of committed updates and also for ensuring that all sites see a transaction-consistent view of the database. The server implementation can be simplified if the server always has the most recent committed copy of a page (either in its memory or on disk). This can be achieved by requiring all pages dirtied by a transaction to be copied to the server before the transaction is allowed to commit.[2] Dirty pages that are copied back to the server have two conflicting characteristics that complicate the buffer replacement policy at the server. On one hand, reclaiming a dirty page's buffer slot requires an I/O to write the page to disk, so keeping a dirty page in the buffer longer can reduce I/O by combining multiple writes to the same page into a single disk write [Chen84]. On the other hand, many of the dirty pages present in the server's buffer pool may not actually be valuable pages, as their placement in the server buffer pool is based on considerations other than their probability of being accessed; the presence of such dirty pages may result in additional disk reads for other pages that are relatively hotter. As will be seen in later sections, these conflicting characteristics will have an effect on the performance of the different global memory management techniques.

2.2. Global Memory Management Techniques

As stated previously, this study concentrates on three related global memory management techniques. These techniques were chosen on the basis of their potential for improving performance versus the complexity of their implementation in a page server environment. The three techniques are outlined below and then presented in more detail in the following section. The techniques are:

Forwarding - The main technique we investigate is to allow a request for a page that is not in the server's buffer pool to be forwarded to a remote client if that client has a copy of the page in its buffer pool. Upon receipt of a forwarded request, the remote client sends a copy of the page directly to the requesting client. The goal of this technique is to reduce disk I/O by extending the amount of memory available to satisfy client page requests. This technique has the highest potential for performance improvement of the three studied, but also requires the most modification to existing data caching algorithms.

Hate Hints - Hate hints are a simple heuristic that can help to keep a larger portion of the database available in memory when the forwarding technique is in use. When the server transfers a page to a client, the server marks that page as hated (i.e., it makes it the "least recently used" page in its buffer pool). The page will then be likely to be replaced when a buffer frame is needed for a new page. This

[2] The relaxation of this restriction requires that the server keep track of the location of any committed updates that are not reflected in either its memory or disk. Also, the server must be able to selectively recover any committed updates that are lost as the result of a client crashing (e.g., using techniques similar to those for handling media recovery).

heuristic is an attempt to reduce page replication between the buffer contents of a server and its clients, thereby allowing a larger number of distinct pages to reside in the global memory. When a page is transferred to a client, it is known that the page is in memory elsewhere in the system, and thus, the copy at the server does not contribute to the percentage of the database available in global memory.

Sending Dropped Pages - This technique attempts to use the server buffer pool to prevent a page from being completely dropped out of the global memory. With this technique, a client informs the server when it intends to drop a particular page from its buffer pool by piggybacking that information on a page request message it sends to the server. If the server determines that the copy to be dropped is the only copy of the page that resides in global memory, it asks the client to send it the page when it is replaced from the client's buffer pool.

2.3. Memory Management Algorithms

In this section, we describe five memory management algorithms that will be used to compare the effectiveness of the global techniques under different workloads and system configurations. One algorithm is a callback algorithm that does not use any of the global techniques. This algorithm is used as a baseline. The other four algorithms are extensions of the baseline algorithm, each of which uses the forwarding technique along with neither, one, or both of the other two global techniques.

2.3.1. Callback Locking (CBL)

CBL is a lock and data caching algorithm based on callback locking [Howa88, Lamb91, Wang91]. In this algorithm, clients initially obtain locks and data by sending requests to the server. Once a page and its corresponding lock are obtained, they can be cached at the client across transaction boundaries. The variant studied here allows caching of read locks but not write locks, as caching write locks was found to be somewhat detrimental to performance for the workloads used in this study [Fran92a]. The caching of a page at a client gives that client an implicit read lock on the page at the server. From the server's point of view, the client then owns the read lock as long as the page is kept in its local buffer.[3] Write locks, on the other hand, are requested explicitly at the server and are released at the end of a transaction. When a client requests a write lock that conflicts with one or more read locks that are currently cached at other clients, the server "calls back" the conflicting locks by sending requests to the sites which have those locks (and page copies) cached. When a client receives a callback request, it checks to see if it is currently using the page. If not, the client removes its copy of the page (if it indeed has one) from its buffer pool and sends an acknowledgement to the server. If the page is currently in use, however, the client queues the callback request and then immediately informs the server that the page is in use. This immediate notification allows the server to perform deadlock detection using accurate information [Lamb91]. The server grants the write lock request only after all conflicting locks have been released.

The fact that the caching of a page at a client grants the client an implicit read lock requires that the server be informed when a page is replaced from a client's buffer pool. Rather than send a message to the server each time it replaces a page, a client simply piggybacks the page numbers of any pages it has

[3] Note that this protocol must be extended if clients are allowed to retain read locks on pages that they do not have in their cache.

dropped on the next message that it sends to the server. As a result of this mechanism, the server's lock table represents a reasonably up-to-date picture of the location of cached data pages throughout the system. This information is slightly conservative in that there is a window during which the server may have an entry for a page copy that has just been dropped from a client. This conservatism does not affect correctness, but may result in an occasional unnecessary callback request.

In the CBL algorithm the server buffer is managed using an LRU policy. Pages become the "most recently used" (i.e., least likely to be replaced) page when they are accessed to be sent to requesting clients. Dirty pages that are copied back to the server by committing transactions are marked as most recently used when they arrive at the server.

2.3.2. The Forwarding Algorithm (FWD)

The first global algorithm, FWD, is simply the callback algorithm extended with the forwarding technique described in Section 2.2. When the server receives a request for a page (and hence, an implicit request for a read lock) from a client, it first it obtains a read lock on the page for the requesting transaction. Once the lock has been obtained, it checks to see if the page is in its local buffer pool and if so, it sends a copy of the page to the requester. If the page is not in the server's buffer, it checks to see if the page is cached at another client and if so, forwards the page request to a remote client that has a copy of the page. When a client receives a forwarded request, it checks to see if it has a copy of the page that it can send to the requesting client, and if so, sends it. A client cannot forward a page if it no longer has that page cached or if it is in the process of trying to obtain a write lock on the page from the server.[4] If the client can not forward the page, it returns the request with a negative acknowledgement to the server. If there are no sites that have copies of the page or if a server receives a negative acknowledgement from a remote client, it reads the page into its buffer pool from disk (as is done for all server buffer misses in the CBL algorithm) and sends it to the requesting client.

2.3.3. Forwarding with Hate Hints (FWD-H)

The FWD-H algorithm is a simple extension of the FWD algorithm that uses the hate hints technique. The algorithm works similarly to FWD except that when the server sends a page to a client the page becomes "hated" (i.e., it is marked as the current "least recently used" page) at the server, making it likely to be replaced from the server's buffer pool. Using the LRU mechanism to implement hate hints has two effects: 1) a non-hated page will never be aged out of the server's buffer pool while the buffer pool contains any hated pages, and 2) hated pages are aged out in a LIFO manner.

As described in Section 2.1, transactions send their dirty pages to the server when they commit. When a dirty page arrives at the server, it is marked as the most recently used page. If a page is present in the buffer pool when a dirty copy of the page arrives at the server, the dirty copy replaces the prior copy in the buffer pool, and it becomes the most recently used page. Conversely, if a page that is marked as dirty in the server buffer pool is sent to a client, it becomes a hated (and still dirty) page.

[4] In the forwarding technique, there is a brief window during which a client may request a write lock on a page at the same time the server is sending it a forwarded request for the page. In this situation, the write lock request will be blocked at the server; the forwarded read request will be rejected by the client and will be satisfied by a disk I/O at the server.

2.3.4. Forwarding with Sending Dropped Pages (FWD-S)

The next algorithm, FWD-S, is an extension of the FWD algorithm in which clients send some of the pages that they drop to the server. This algorithm takes advantage of the message patterns inherent in the baseline CBL algorithm. When a client determines that it needs to request a page from the server, it also checks to see if the new page will force an existing cached page out of the buffer. If so, the client piggybacks the page number of the page it plans to drop on the request message that it sends to the server. When the server receives such a page request, it checks to see if the page to be dropped is the only copy of the page that is currently in the global memory. If so, the server sets a flag in the message that it uses to respond to the page request; this flag informs the client that it should send the page (asynchronously) back to the server rather than simply drop it. When the dropped page arrives at the server, it is marked as the most recently used page.

There are two additional cases that the algorithm must handle. First, if the server forwards the request to a remote client, the remote client must forward the server's send-back decision to the requester along with the page. The second case occurs when the server determines that it will have the only remaining memory-resident copy of the page once the requester drops its copy. In this case, the server marks its copy of the page as most recently used and informs the client that it need not send the dropped page.

2.3.5. Forwarding with Hate Hints and Sending Dropped Pages (FWD-HS)

The final global algorithm is the FWD algorithm extended with both the hate hints and sending dropped pages techniques. It is simply the combination of the FWD-H and FWD-S algorithms.

2.4. Performance Tradeoffs

The previous sections described three techniques for improving performance through global memory management and presented algorithms that use these techniques to extend an algorithm that uses only local memory management. Before presenting the detailed results from our simulation study of these algorithms, it will be useful to consider the expected performance tradeoffs among them. CBL, the baseline algorithm, does not exploit remote client memory and must therefore rely only on the local client memory, the server memory, and disk. The FWD algorithm uses messages and some extra client CPU processing in an attempt to avoid doing disk I/O on server buffer misses. The FWD-H algorithm attempts to further reduce disk I/O by avoiding replication between the contents of the server and its clients, thus increasing the portion of the database that is available in memory. The FWD-S algorithm also tries to replace disk I/O by messages; it attempts to increase the portion of the database retained in memory by sending a copy of a page to the server rather than dropping it, if that copy is the only one resident in global memory. In comparing the FWD-H and FWD-S algorithms, it can be noted that the hate hints and sending techniques have similar goals in that both try to increase the portion of the database that is available in memory. Hate hints is an indirect approach which tries to accomplish its goal by reducing replication. In contrast, the sending technique is a more direct approach, as the system actively tries to keep pages from being dropped from the global memory. Finally, FWD-HS combines all of these techniques and, if the benefits of the hate hints and sending techniques are additive, should keep even more of the database in memory than the other algorithms.

3. MODELING A CLIENT-SERVER DBMS

In order to study the performance of alternative global memory management techniques, we have extended the client-server DBMS simulation model that was used in our earlier studies [Care91, Fran92a]. In this section we describe how the model captures the database, workload, and physical resources of a client-server DBMS that supports the proposed global memory management techniques.

3.1. Database and Workload Models

Table 1 presents the parameters used to model the database and its workload. The database is modeled as a collection of *DatabaseSize* pages of *PageSize* bytes each. The system workload is generated by a collection of *NumClients* client workstations. Each client workstation generates a single stream of transactions, where the arrival of a new transaction is separated from the completion of the previous transaction by an exponential think time with a mean of *ThinkTime*. A client transaction reads between 0.5·*TransactionSize* and 1.5·*TransactionSize* distinct pages from the database. It spends an average of *PerPageInst* CPU instructions processing each page that it reads (this amount is doubled for pages that it writes); the actual per-page CPU requirements are drawn from an exponential distribution.

An important feature of the model is its scheme for defining the page access patterns of workloads, which allows different types of locality at clients and data-sharing among clients to be easily specified. The workload is specified on a per client basis. For each client, two (possibly overlapping) regions of the database can be specified. These ranges are specified by the *HotBounds* and *ColdBounds* parameters. The parameter *HotAccessProb* specifies the probability that a page access will be to a page in the hot region, with the remainder of accesses being to pages in the cold region. Within each region, pages are chosen without replacement using a uniform distribution. The *HotWriteProb* and *ColdWriteProb* parameters specify the region-specific probabilities of writing a page that has been accessed.

System-Wide	
DatabaseSize	Size of database in pages
PageSize	Size of a page
NumClients	Number of client workstations
Per Client	
ThinkTime	Mean think time between client transactions
TransactionSize	Mean no. of pages accessed per transaction
PerPageInst	Mean no. of instructions per page on read (doubled on write)
HotBounds	Page bounds of hot range
ColdBounds	Page bounds of cold range
HotAccessProb	Prob. of accessing a page in the hot range
HotWriteProb	Prob. of writing to a page in the hot range
ColdWriteProb	Prob. of writing to a page in the cold range

Table 1: Database and Workload Parameters

3.2. Physical Resource Model

The model parameters that specify the physical resources of the system and their usage are listed in Table 2. The client and server CPU speeds are specified in MIPS (*ClientCPU* and *ServerCPU*). The service discipline of the client and server CPUs is first-come, first-served (FIFO) for system services such as message and I/O handling. Such system processing preempts other CPU activity. For non-system processing, a processor-sharing discipline is used. The sizes of the buffer pools on the clients and on the server (*ClientBufSize* and *ServerBufSize*) are specified as a percentage of the database size. The client and server buffer pools are both managed using an LRU replacement policy as a default, but facilities such as hate hints are provided to allow the implementation of the policies described in Section 2. Dirty pages are not given preferential treatment by the replacement algorithm but are written to disk when they are selected for replacement. Note that on clients, dirty pages exist only during the course of a transaction. Dirty pages are held on the client until commit time, at which point they are copied back to the server; once the transaction commits, the updated pages are marked as clean on the client.

The parameter *ServerDisks* specifies the number of database disks attached to the server, and each is modeled as having an access time that is uniformly distributed over the range from *MinDiskTime* to *MaxDiskTime*. The disk used to service a given request is chosen at random from among the server disks, so the model assumes that the database is uniformly partitioned across all server disks. The service discipline for each of the disks is FIFO. A CPU charge of *DiskOverheadInst* instructions is incurred for each I/O request. We do not explicitly model logging, as it is not expected to impact the relative performance of the algorithms being studied.

A very simple network model is used in the simulator's *Network Manager* component: the network is modeled as a FIFO server with a service rate of *NetworkBandwidth*. 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 their on-the-wire costs, and to allow the on-the-wire costs of messages to be adjusted using the bandwidth parameter. The CPU cost for managing the protocol for a

Parameter	Meaning
ClientCPU	Instruction rate of client CPU
ServerCPU	Instruction rate of server CPU
ClientBufSize	Per-client buffer size
ServerBufSize	Server buffer size
ServerDisks	Number of disks at server
MinDiskTime	Minimum disk access time
MaxDiskTime	Maximum disk access time
DiskOverheadInst	CPU overhead for performing disk I/O
NetworkBandwidth	Network bandwidth
FixedMsgInst	Fixed no. of instructions per message
PerByteMsgInst	No. of addl. instructions per message byte
ControlMsgSize	Size of a control message (in bytes)
LockInst	No. of instructions per lock/unlock pair
RegisterCopyInst	No. of instructions to register/unregister a copy

Table 2: Resource and Overhead Parameters

message send or receive is modeled as *FixedMsgInst* instructions per message plus *PerByteMsgInst* instructions per message byte.

Finally, the model allows the specification of several other resource-related parameters. The size of a control message (such as a lock request or a commit protocol packet) is given by the parameter *ControlMsgSize*; messages that contain one or more data pages are sized based on Table 1's *PageSize* parameter. Other costs include *LockInst*, the cost involved in a lock/unlock pair on the client or server, and *RegisterCopyInst*, the cost (on the server) to register and unregister (i.e., to track the existence of) a newly cached page copy or to look up the copy sites for a given page.

3.3. Client-Server Execution Model

In the simulator, each client consists of several modules. These include: a *Source*, which generates the workload; a *Client Manager*, which executes the transaction reference strings generated by the Source and processes requests and page receipts from the server and other clients; a *CC Manager*, which is in charge of concurrency control (i.e., locking) on the client; a *Buffer Manager*, which manages the client buffer pool; and a *Resource Manager*, which models the other physical resources of the client workstation. The server is organized similarly, except that it is controlled by the *Server Manager*, which acts in response to the requests sent to it by the clients.

Client transactions execute on the workstations that submit them. When a transaction references a page, the Client Manager must lock the page appropriately and check the local buffer pool for a cached copy of the page; if no such copy exists, the client sends a request for the page to the server. Both locking and buffer management are simulated in detail based on referenced page numbers. Once a local copy of the page exists, the transaction processes the page and and decides whether or not to update it. In the event of an update, the client obtains a write lock on the page locally, and then requests a write lock from the server. The server may be required to callback read locks from other clients before it can grant the write lock request. Once the write lock is obtained, further CPU processing is performed on the page. At commit time, the Client Manager sends a commit request together with copies of any updated pages to the server, which performs the commit processing for the transaction (e.g., placing the copies of the dirty pages in its buffer and releasing locks) and then informs the client that the commit was successful. The server performs deadlock detection based on the information in its lock table and the responses received to its callback requests if a callback is involved in a potential deadlock. If the server decides that it must abort a transaction, it chooses a victim and informs the victim's client manager that the transaction must be aborted. If the victim has an outstanding callback request, the other clients participating in the callback are also informed. When a transaction's client receives an abort request, its Client Manager arranges for the abort, asks the Buffer Manager to purge any updated pages, and then resubmits the same transaction.

4. EXPERIMENTS AND RESULTS

In this section, we present the results of a simulation study of the global memory management algorithms described in Section 2.3. We describe the experiments and results following a discussion of the performance metrics and the parameter settings that were used.

4.1. Metrics and Parameter Settings

The primary performance metric employed in this study is the throughput (i.e., transaction completion rate) of the system.[5] A number of additional metrics are also used to aid in the analysis of the experimental results, including the server buffer hit rate, the client and server resource utilizations, the average number of messages required to execute a transaction, and several others. One special metric that we use is the "database portion available in memory". This is the percentage of the pages of the database that are available to a client without performing a disk I/O. For the forwarding algorithms, this metric is the union of the contents of the server buffer pool and all client buffer pools, whereas for the callback algorithm it is the union of the server buffer pool contents and the contents of only a single client. The various metrics that are presented on a "per commit" basis are computed by dividing the total count for the metric by the number of transaction commits over the duration of a simulation run. To ensure the statistical validity of our results, we verified that the 90% confidence intervals for transaction response times (computed using batch means [Sarg76]) were sufficiently tight. The size of these confidence intervals was within a few percent of the mean in all cases, which is more than sufficient for our purposes. Throughout the paper we discuss only performance differences that were found to be statistically significant.

Tables 3 and 4 present the database and workload parameter settings used in the experiments reported here. Table 3 contains default settings that are common across all of the experiments (except where otherwise noted). The database size is 1,250 pages, with a page size of 4 kilobytes. The number of client workstations is varied from 1 to 25 in order to study how the various algorithms scale, and the think time at the client workstations is zero. The default per-page CPU processing time is 30,000 instructions.

Parameter	Setting
DatabaseSize	1,250 pages (5 megabytes)
PageSize	4,096 bytes
NumClients	1 to 25 client workstations
ThinkTime	0 seconds
PerPageInst	30,000 instructions

Table 3: Database and Workload Parameter Settings

Parameter	RO-HOTCOLD	RW-HOTCOLD	PRIVATE	UNIFORM
TransactionSize	20 pages	20 pages	16 pages	20 pages
HotBounds	p to p+49, p=50(n−1) + 1	p to p+49, p=50(n−1) + 1	p to p+24, p=25(n−1) + 1	—
ColdBounds	rest of DB	rest of DB	626 to 1,250	whole DB
HotAccessProb	0.8	0.8	0.5	—
ColdAccessProb	0.2	0.2	0.5	1.0
HotWriteProb	0.0	0.2	0.2	—
ColdWriteProb	0.0	0.2	0.0	0.2

Table 4: Workload Parameter Values for Client n

[5] We use a closed queuing model, so the inverse relationship between throughput and response time makes either a sufficient metric.

Table 4 describes the workloads considered in this study. These workloads and their motivations will be explained as their corresponding experiments are presented. Briefly, the HOTCOLD workloads have a high degree of locality per client as well as a moderate amount of sharing among clients. Two variants of HOTCOLD are studied: RO-HOTCOLD, a read-only variant, and RW-HOTCOLD, which has a moderate write probability (20%). The PRIVATE workload has high locality per client and only read sharing among clients; there are no read-write or write-write conflicts in this workload. It is intended to model situations such as large, CAD-based engineering projects, in which engineers might work on disjoint portions of an overall design while read-sharing a standard library of components or a previous version of the design. UNIFORM is a moderate write probability workload with no client locality. We used a relatively small database in conjunction with these workloads in order to make simulations involving fractionally large buffer pools and transactions feasible in terms of simulation time; moreover, our intent is to capture that portion of the database which is of relatively current interest to the client workstations, rather than to model the entire database.

Table 5 shows the settings used in our experiments for the system overhead parameters and the resource-related parameters. In setting these parameters we attempted to choose values that are reasonable approximations to what might be expected of systems today or in the near future. The experiments that we describe here were run with 15 MIPS client workstations and a 30 MIPS server. We ran experiments with two network bandwidths, one corresponding roughly to current Ethernet speeds (referred to as the *slow* network in the following sections) and one corresponding roughly to FDDI technology (referred to as the *fast* network). The bandwidth values used (8 Mbits/sec and 80 Mbits/sec respectively) represent slightly discounted values of the stated bandwidths of those networks.

Parameter	Setting
ClientCPU	15 MIPS
ServerCPU	30 MIPS
ClientBufSize	5% or 15% of database size
ServerBufSize	30% of database size
ServerDisks	2 disks
MinDiskTime	10 millisecond
MaxDiskTime	30 milliseconds
DiskOverheadInst	5000 instructions
NetworkBandwidth	8 or 80 megabits per second
FixedMsgInst	20,000 instructions
PerByteMsgInst	10,000 instructions per 4 kilobyte page
ControlMsgSize	256 bytes
LockInst	300 instructions
RegisterCopyInst	300 instructions

Table 5: Resource and Overhead Parameter Settings

4.2. Experiment 1: Read-Only HOTCOLD Workload

The first set of results that we will examine uses a version of the HOTCOLD workload that performs no updates. Although such a read-only workload is not expected to be common, we analyze it first in order to examine the buffering behavior of the various algorithms in the absence of the complications that

are introduced by dirty pages. In the RO-HOTCOLD workload, as shown in Table 4, each client has its own 50 page region of the database to which 80% of its accesses are directed. The hot region of one client is contained in the cold regions of all other clients, so there is substantial sharing of pages in this workload in addition to high per-client locality.

4.2.1. Read-Only HOTCOLD, Small Client Buffer Pools

The aim of each of the global memory management techniques is to reduce the need for disk I/O by increasing the portion of the database that is available in memory. However, there are two reasons why such an increase may not translate into a performance improvement: 1) the resources used to increase the portion of the database available in memory may be more expensive than the resources saved by the increase, and 2) in a skewed workload such as RO-HOTCOLD, some pages are more valuable than others, so a higher portion of the database available in memory does not necessarily imply a reduction in disk I/O. In the following, we first compare the algorithms based on the portion of the database that they keep available in memory, and then examine the resulting resource demands. Finally, we examine how these demands translate into throughput, given the system parameters of Section 4.1.

4.2.1.1. Portion of Database Available In Memory

Figure 2 shows the percentage of the database available in memory for each of the algorithms when running the Read-Only HOTCOLD workload with small client buffer pools (5% of the database size). The dotted line shows the highest in-memory percentage that could be obtained ideally (based on the amount of memory in the system). Algorithms typically have less than the ideal amount of the database in memory due to replication among the contents of the system's buffers pools. There are two types of replication that can arise: server-client correlation, and client-client replication. Server-client correlation can arise when the server and the client buffer managers use the same page replacement policy (LRU). In this situation, when the ratio of the number of pages resident in the server buffer pool to the number of distinct pages that are resident in client buffer pools is high (e.g., greater than one), a page that is in a client's buffer pool is also likely to be in the server's buffer pool. Server-client correlation is most prominent with small client populations. As clients are added to the system, the ratio of pages at the server to pages at the clients becomes smaller, and as this ratio decreases, the server can replicate fewer of the pages that are kept at clients.[6] Client-client replication arises from overlapping client requests. The amount of client-client replication increases as clients are added to the system.

Turning to Figure 2, we first note that CBL has the smallest portion of the database available in memory. CBL does not use forwarding, so the addition of clients does not increase the amount of memory that can be used to service a particular client's requests. CBL has a slight increase the percentage of the database it has available in memory as clients are added to the system, which is due to the reduction in server-client replication. CBL is unaffected by client-client replication, as each client has access only to the contents of the server's buffer pool and its own buffer pool. In contrast to CBL, the forwarding algorithms can capitalize upon the buffer pools brought to the system by additional clients and

[6] In this experiment the aggregate size of the client buffer pools becomes larger than the size of the server buffer pool when more than 6 clients are in the system.

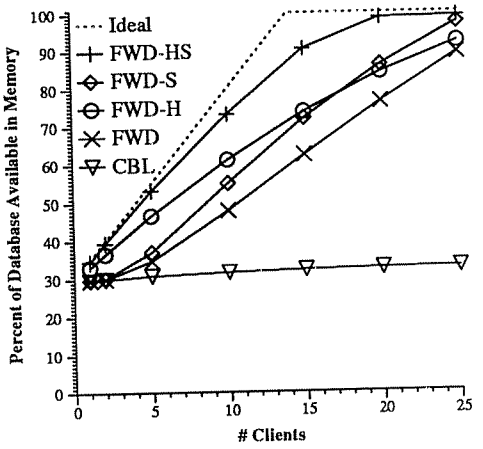

Figure 2: % of DB Available in Memory (RO-HOTCOLD, 5% Client Bufs)

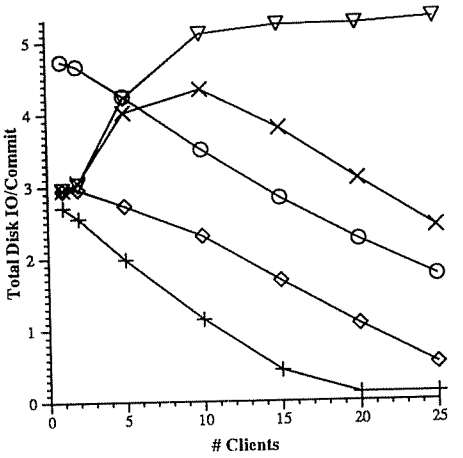

Figure 3: Disk I/O Operations per Commit (RO-HOTCOLD, 5% Client Bufs)

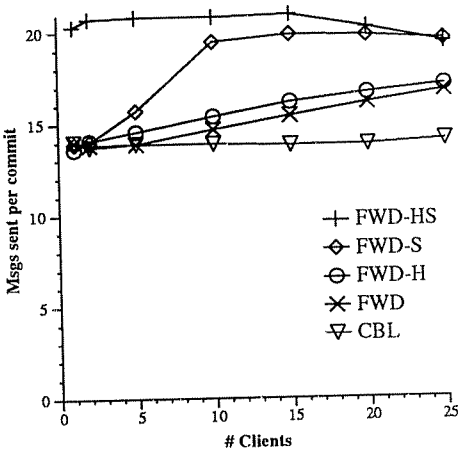

Figure 4: Messages Sent per Commit (RO-HOTCOLD, 5% Client Bufs)

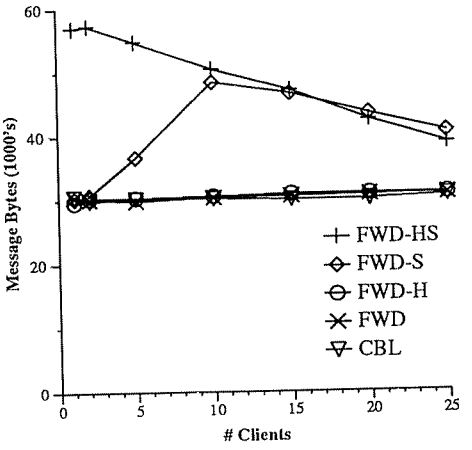

Figure 5: Message Volume per Commit (RO-HOTCOLD, 5% Client Bufs)

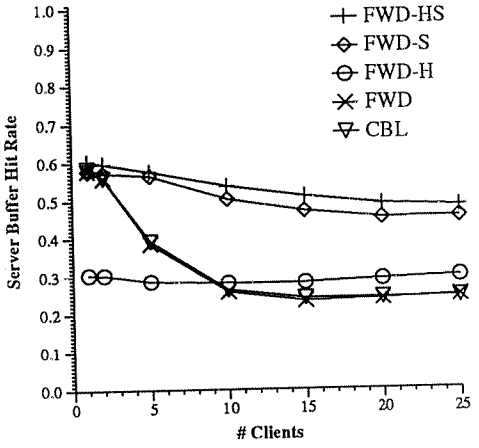

Figure 6: Server Buffer Hit Rate (RO-HOTCOLD, 5% Client Bufs)

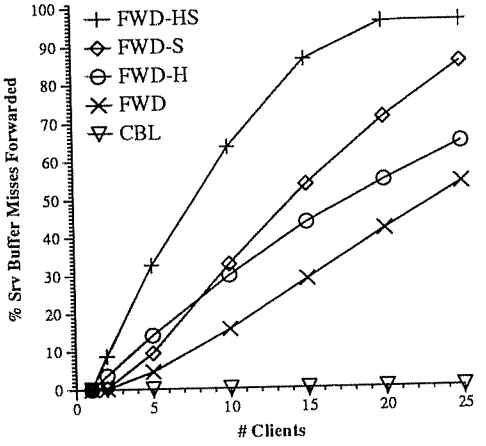

Figure 7: % of Server Misses Forwarded (RO-HOTCOLD, 5% Client Bufs)

thus, they all show a significant increase in the portion of the database available in memory as clients are added. However, as seen in Figure 2, none of the forwarding algorithms are able to attain the ideal in-memory percentage. The forwarding algorithms are affected by both types of replication described above. However, all of the forwarding algorithms incur the same level of client-client replication, as the global memory management techniques they use do not alter the buffering behavior at clients. Therefore, the differences among the forwarding algorithms shown in Figure 2 are the result of differences in their server-client correlation.

FWD, the simplest forwarding algorithm, initially shows very little improvement over CBL. This is due to the correlation between server and client buffers — the contents of the additional client buffers are replicated in the server's buffer pool. As clients are added, the impact of the server-client correlation decreases, until at 25 clients, FWD has access to almost 90% of the database in memory. The other three forwarding algorithms employ techniques that attempt to increase the percentage of the database available in memory. Compared to FWD, the FWD-H algorithm attains a relatively high in-memory percentage with small numbers of clients. This is because hate hints manage to reduce the server-client correlation. However, beyond 15 clients FWD-H's advantage over FWD begins to dissipate; at 25 clients, FWD-H is only slightly better than FWD. This is because as clients are added to the system, the fraction of FWD-H's page requests that are serviced by forwarding increases. Requests serviced by forwarding do not go through the server's buffer pool, so the hate hints become less effective at reducing the server-client correlation.

FWD and FWD-H both have somewhat less than the ideal in-memory database percentage at 25 clients. FWD-S, on the other hand, comes close to having the entire database in memory at this point. The success of FWD-S is due to its effective use of the server buffer pool — it uses the server buffer pool to retain pages that would otherwise have been dropped from the global memory. However, with small client populations, the sending technique has little effect. When server-client buffer correlation is high, pages that are aged out of clients are likely to be in the server buffer pool, so few dropped pages are sent to the server. The sending technique is quite effective for larger client populations but less effective for smaller populations, while the hate hints technique has the opposite characteristics. For this reason, the available in-memory percentages for the two algorithms eventually cross.

FWD-HS, which combines the hate hints and sending techniques, keeps the largest portion of the database available in memory throughout the range of 1 to 25 clients. At 25 clients, FWD-HS has almost 100% of the database available in memory. The interaction of the two techniques is effective throughout the range of client populations, as it tends to keep a copy of a page in memory at either a client or at the server, but not at both.

4.2.1.2. Resource Requirements

We now turn our attention to the resource requirements of the five global memory management algorithms. As expected, the general trend is that in most cases, an increase in the percentage of the database available in memory results in a decrease in disk I/O (since more requested pages are found in memory) and an increase in messages (for serving such requests and for managing the contents of global memory). Figure 3 shows the total number of disk I/Os per committed transaction (in this workload, all disk I/Os

are reads) for the various algorithms. The message requirements are shown in Figure 4, which shows the average number of messages sent per committed transaction, and Figure 5, which shows the total number of message bytes sent per committed transaction. The latter two metrics can differ because some messages are control messages (256 bytes), while other messages contain one or more 4K byte pages. The message and disk I/O requirements for transactions depend on the client buffer hit rate (not shown), the server buffer hit rate (shown in Figure 6), and the percent of server misses that are forwarded to other clients (shown in Figure 7). The sending algorithms also incur additional page-sized messages for sending dropped pages back to the server. All of the algorithms have the same client buffer hit rate (slightly over 65%) because they are all based on CBL and because the global techniques do not affect buffering at the clients. As a result, for all of the algorithms, clients send the same number of page requests to the server. Overall, CBL sends the fewest messages per commit, because it never forwards requests to other clients. For the same reason, its disk I/O requirements are inversely proportional to its server buffer hit rate. CBL initially suffers a steep decrease in the server hit rate (as explained below) as clients are added to the system. Its ultimately low server hit rate and its inability to forward requests cause CBL to have the highest disk I/O requirements at 10 clients and beyond. Note that beyond 10 clients, CBL is the only algorithm for which disk requirements do not decrease as clients (and hence, more buffers) are added to the system.

CBL's server buffer hit rate drops from 58% to 24% (where 30% is what would be expected with a uniform access pattern). This drop is due to the combination of the skewed nature of the RO-HOTCOLD workload and the small client buffer pools. Due to the small client buffer pools (62 pages), the LRU mechanism at each client frequently ages out pages that belong to the client's hot range (at the clients, the hit rate for hot region pages is about 81%). With small numbers of clients in the system, the server buffer pool can hold all of the hot region pages for all of the active clients, and therefore, client misses due to aged-out hot pages are likely to be found at the server (e.g., with two clients, the hit rate for hot pages at the server is nearly 97%). However, as clients are added, the server can hold fewer of the active clients' hot region pages, so the server hit rate for each client's hot region pages drops; in this case, to below 18% at fifteen clients and beyond (compared to over 28% for cold region requests). The lower hot region hit rate is due to another correlation phenomenon: the server tends to keep only the hot region pages that were most recently requested by a client. Unfortunately, these pages are the wrong pages to keep, as hot region pages tend to reside in a client's buffer pool for a long time before they are finally aged-out and subsequently re-requested.[7]

As shown in Figure 6, FWD has a similar overall server hit rate to CBL. Forwarding has only a minimal effect on the server hit rate in this case — slightly lowering the hot region hit rate and raising the cold region hit rate. However, despite this similarity, FWD's resource requirements are different than CBL's. FWD is able to satisfy a significant number of server buffer misses by forwarding requests to other clients (see Figure 7). Therefore, as clients are added, there is an increase in messages but a

[7] In fact, the 18% hit rate obtained for hot region pages is due primarily to requests for hot region pages that were recently accessed as *cold region* pages by other clients. If there were no overlapping cold region accesses, the hot region hit rate would approach zero.

decrease in disk I/O. In contrast to FWD, the other forwarding algorithms take a more active role in affecting the server's buffering behavior. FWD-H has a server buffer hit rate that remains around 30%. The hate hints reduce the impact of the skewed workload on the server buffer hit rate, and thus, the hot region and cold region hit rates both remain close to 30%. Unfortunately, for small numbers of clients, FWD-H has a much lower hit rate than those obtained by the other algorithms. In its attempt to reduce server-client correlation, FWD-H removes hot pages from the server's buffer pool. Many of those pages, while replicated for a brief time, will be eventually aged out of the client's buffer pool and re-referenced at the server. Thus, with small numbers of clients, the reduction in server-client correlation causes a lower server hit rate, and as a result, FWD-H has the highest disk requirements up to 5 clients (Figure 3). However, at 10 clients and beyond, FWD-H's server hit rate becomes better than that of CBL and FWD because it avoids the server-client correlation that causes those algorithms to have a low hit rate for hot region pages. The reduction in server-client correlation also allows FWD-H to be more successful than FWD at forwarding requests missed at the server to other clients (Figure 7). As a result of the server hit rate and forwarding behavior, FWD-H sends more messages than CBL and FWD, but beyond 5 clients, performs fewer disk I/Os.

The sending technique provides a substantial improvement in the server hit rate. As shown in Figure 6, FWD-S has the same initial hit rate as FWD and CBL but it does not suffer as severe a drop in hit rate as clients are added. The high server hit rate is due to the sending technique's ability to keep hot region pages in memory. The influence of the sending technique can be seen in Figure 4, which shows the number of messages sent by FWD-S increasing until 10 clients are in the system. The number of dropped pages sent to the server (not shown) increases due to the reduction in server-client buffer correlation — hot pages that are aged out of client buffers become less likely to be in the server's buffer pool as clients are added. Beyond 10 clients, the number of pages sent by FWD-S begins to decrease, as it becomes more likely that a page dropped by a client is resident in the memory of another client. Despite the reduction in sent pages, FWD-S's message count remains fairly constant due to an increase in forwarded requests. However, its per transaction network bandwidth demands actually decrease (see Figure 5). As shown in Figure 7, at 10 clients and beyond, FWD-S forwards a larger percent of its server misses than the non-sending algorithms. It is important to note that the crossover point of the forwarded percentages of FWD-S and FWD-H occurs with fewer clients than the crossover of their respective in-memory percentages shown in Figure 2. This is because FWD-S does a better job of keeping hot range pages at the server so a miss at the server is likely to be for a cold range page. Such cold range pages are typically in the hot range of another client, and will often be cached in that client's local buffer pool.

The combination of hate hints and the sending technique gives FWD-HS the best server hit rate of the five algorithms. FWD-HS also has the highest forwarded percentage of all of the algorithms. In fact, at 20 clients and beyond, FWD-HS reads a page from disk only once; all subsequent transactions can access the page in memory. As a result, FWD-HS has the lowest disk I/O requirements and the highest message count of the five algorithms. FWD-HS also exhibits an interesting, and potentially expensive, behavior with small client populations: hot region pages are "bounced" between clients and the server. With 2 clients in the system, over 95% of the pages dropped by the clients are sent back to the server. This occurs at a large cost in messages and network bandwidth, and provides only a small savings in disk I/O.

As with FWD-S, the number of dropped pages sent to the server decreases for FWD-HS as clients are added to the system. This reduction, combined with an increase in forwarded requests, results in a slight decrease in message count and a significant drop in network bandwidth requirements.

4.2.1.3. Throughput Results

The last two sub-sections examined the effectiveness of the algorithms in keeping pages available in memory and studied their resource requirements. With these results in mind, we now turn to the resulting performance of the algorithms. Figure 8 shows the throughput results for this experiment with the slow network setting (*NetworkBandwidth* = 1 MByte/sec). All of the forwarding algorithms eventually outperform CBL, showing the potential advantages of avoiding I/O — even at the cost of additional messages. FWD-H has the highest throughput through much of the range, with FWD equaling it at 25 clients. Beyond 10 clients, the sending algorithms perform below the level of FWD and FWD-H. In this case, all of the forwarding algorithms eventually become network-bound and their relative performance becomes inversely proportional to their message bandwidth requirements. With small client populations, however, the relative performance results are somewhat different. The forwarding technique provides no clear performance improvement over the baseline CBL algorithm; in fact, FWD-H and FWD-HS perform slightly worse than CBL up to 5 clients.

The CBL algorithm initially performs well because it has low message requirements and its disk requirements are in line with the other algorithms. However, the other algorithms soon produce a decrease in I/O requirements, while CBL does not. CBL approaches a disk bottleneck at 10 clients and ultimately has the lowest performance. FWD performs similarly to CBL up to 5 clients, but decreasing I/O requirements allow it to eventually perform much better than CBL, approaching a network bottleneck at 25 clients. FWD-H initially suffers due to high I/O requirements with small client populations. However, as clients are added to the system, its I/O requirements diminish and, because it has moderate message requirements, it becomes the best performing algorithm. FWD-S has the best performance at 5

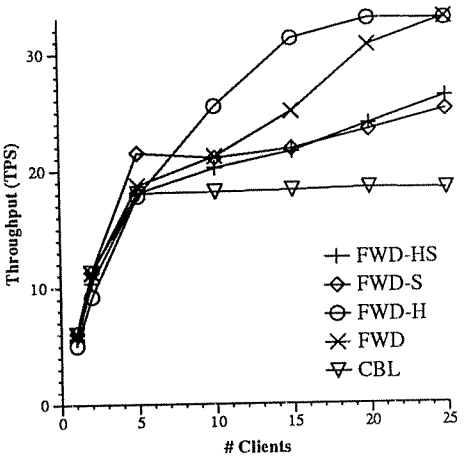

Figure 8: Throughput (RO-HOTCOLD, 5% Cli Bufs, Slow Net)

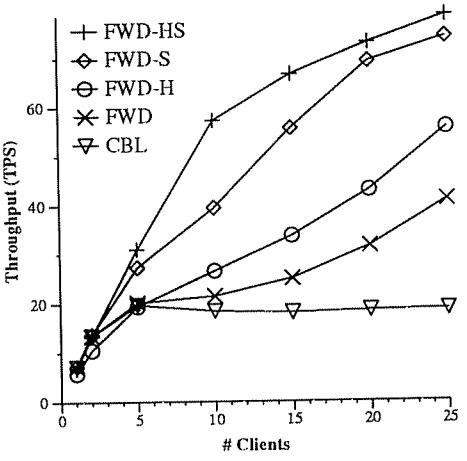

Figure 9: Throughput (RO-HOTCOLD, 5% Cli Bufs, Fast Net)

clients due to its very low I/O requirements. Beyond 5 clients, however, the increase in dropped pages sent by FWD-S causes its performance to suffer relative to the FWD and FWD-H algorithms, which have lower bandwidth requirements. FWD-S is network-bound at 10 clients and beyond; its throughput improvement beyond this point is due to the reduction in network bandwidth requirements as fewer dropped pages are sent to the server. FWD-HS has the lowest I/O requirements throughout the range of client populations, but due to its high message requirements and the slow network, it performs poorly compared to FWD and FWD-H. FWD-HS is the first algorithm to hit the network bottleneck; it becomes network-bound at 5 clients. Its network bandwidth requirements cause it to perform below FWD-S prior to 15 clients, and only slightly better than FWD-S thereafter.

Figure 9 shows the throughput results for the same workload and buffer pool sizes as the previous case, but with a faster (e.g., FDDI) network. The faster network has the effect of reducing the cost of using network bandwidth, and thus, the trade-off of messages for disk I/O becomes a better bargain. In this case, therefore, FWD-HS has the best performance, followed by FWD-S, FWD-H, and FWD. CBL, the non-forwarding algorithm, has the lowest performance. The sending algorithms both become CPU-bound at the server due to the cost of processing large messages, while the other algorithms are negatively impacted by their greater I/O requirements.

4.2.2. Read-Only HOTCOLD, Large Client Buffer Pools

We now turn our attention to the Read-Only HOTCOLD workload with the client buffer pool size increased to 15% of the database size. The larger client buffers have two important effects for this workload: 1) more of the database is available in memory with smaller numbers of clients than in the cases previously studied, and 2) the client buffers are large enough so that hot region pages are very rarely dropped from a client's buffer pool. Figure 10 shows the throughput results for this case using the slow network. As can be seen in the figure, the forwarding algorithms all converge at 20 clients and beyond, while the CBL algorithm has lower performance than the forwarding algorithms at five clients and

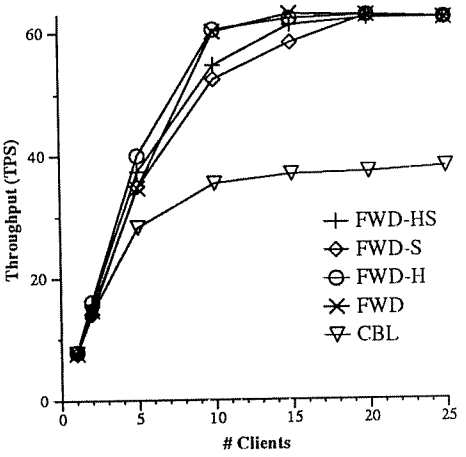

Figure 10: Throughput (RO-HOTCOLD, 15% Cli Bufs, Slow Net)

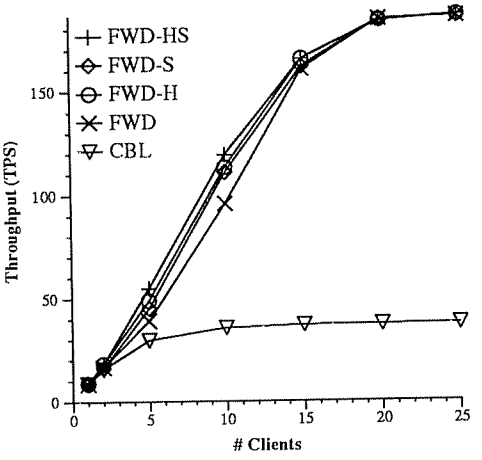

Figure 11: Throughput (RO-HOTCOLD, 15% Cli Bufs, Fast Net)

beyond. The CBL algorithm is once again disk-bound (although at a higher performance level than in the previous cases), and the forwarding algorithms all become network-bound at 15 clients and beyond. Prior to converging, the sending algorithms perform somewhat worse than the other forwarding algorithms. This is because they incur large message costs for sending dropped pages back to the server, and in this case most dropped pages are cold region pages, so there is little benefit to having these pages at the server. FWD-H has a slight performance advantage up to 10 clients due to its effectiveness at reducing server-client correlation with small client populations. Unlike in the small buffer case, this reduction provides an improvement in the server buffer hit rate, as clients tend to request only cold region pages from the server. FWD-HS has a slightly better initial server hit rate than FWD-H, but its performance is penalized by its high message requirements.

The forwarding algorithms eventually converge in Figure 10 for three reasons: First, all of the forwarding algorithms have access to the entire database in global memory at 20 clients and beyond, so they perform no disk I/O at that point. Second, the large buffer pools cause the sending algorithms to send fewer pages back to the server as clients are added to the system — it becomes less likely that a client has the only copy of a cold region page. Third, at 20 clients and beyond, all of the forwarding algorithms have the same server hit rate (around 30%). This is because with a large number of clients, the potential overlap of each client's buffer contents with the server is small, and since most page requests sent to the server are for cold region pages, the server sees a non-skewed access pattern.

The performance results for the faster network case (shown in Figure 11) are similar to those seen with the slower network, in that all of the forwarding algorithms converge at a performance level much higher than the CBL algorithm (in this case, by a factor of about 5 at 25 clients). However, in this case, the fast network allows FWD-HS and FWD-H to capitalize on their high server hit rates prior to the convergence.

4.2.3. Summary of the Read-Only HOTCOLD Results

The study of the HOTCOLD workload in the absence of updates revealed a number of important aspects of the performance of the global memory management techniques. Most importantly, it was shown that forwarding page requests to remote clients can provide significant performance improvements. Forwarding a request saves a disk I/O while requiring no extra CPU work at the server, thus offloading an important shared resource. The cost of forwarding is an extra control message plus the latency due to message handling at the remote client. Even in network constrained situations, forwarding was found to be beneficial.

The hate hints technique was found to improve the performance of forwarding by reducing the correlation of buffer contents between clients and the server. An important exception to this was in cases with small client buffer pools and small numbers of clients. In such cases, the hate hints were found to hurt the server hit rate by removing valuable hot region pages from the server buffer pool. As a result, there were cases in which the FWD-H algorithm had a larger portion of the database available in memory, but had higher disk requirements than other algorithms. In contrast, the sending technique was found to be effective at keeping hot region pages in memory. However, the sending technique was found to pay a large price in message bandwidth to avoid disk I/Os, often causing the slow network to become a bottleneck

while the disk became underutilized. Furthermore, when the size of the client buffer pools was large enough to keep hot region pages from being replaced at the clients, the sending technique was detrimental to performance because it used valuable network bandwidth to keep cold region pages in memory.

The FWD-HS algorithm, which uses a combination of the hate hints and sending techniques, was able to keep more of the database available in memory than any of the other algorithms studied due to its ability to make effective use of the server buffer pool. However, this did not always translate into better performance. With small buffer pools, the combination of the techniques resulted in hot region pages being bounced between clients and the server across the network, resulting in heavy network traffic. With larger buffer pools, its performance was negatively affected by the tendency of the sending technique to waste network bandwidth on cold region pages. Message costs hurt the throughput of FWD-HS in the slow network cases, but when the fast network was used, FWD-HS was the best performing algorithm.

In general, the experiments showed that the global memory management techniques were effective in offloading the server's disks by increasing the amount of the database available in memory. However, it was also seen that while offloading the disk in this manner can provide substantial performance gains, doing so is not a guarantee of improved performance. In particular, if the wrong pages are kept in memory, or if the price paid to keep pages available in memory is too high, performance can suffer.

4.3. Experiment 2: Read-Write HOTCOLD Workload

The previous section analyzed the five memory management algorithms in the absence of writes in order to examine their behavior without the complications introduced by dirty pages. In this section, we investigate the performance of the algorithms using the HOTCOLD workload with a write probability of 20% (*HotWriteProb* = *ColdWriteProb* = 0.20). In the following discussion, we concentrate on those aspects of performance that are caused by the introduction of writes.

4.3.1. Read-Write HOTCOLD, Small Client Buffer Pools

Figure 12 shows the percentage of the database available in memory for the Read-Write HOTCOLD workload with small client buffer pools and the slow network. Compared to the read-only case (Figure 2) CBL remains largely unchanged, FWD and FWD-H initially have a slight degradation but eventually have an improvement at larger client populations, and FWD-HS suffers degradation throughout the range of clients. Most strikingly, the large benefit of FWD-HS observed in the earlier experiment is not present here. With small client populations, the server-client buffer correlation is increased by the dirty pages that are sent to the server by committing transactions. This particularly hurts the FWD-H and FWD-HS algorithms, which gained by reducing this correlation in the read-only case. However, as more clients are added, callback requests begin to reduce the replication among client buffer contents; before a page is updated at one client, it is removed from the buffers at any other clients that have it cached. Therefore, the result of the addition of writes is to increase server-client correlation while slightly decreasing client-client replication.

Figure 13 shows the total number of disk I/Os (shown as solid lines) and the number of disk writes (shown as dotted lines) performed per committed transaction. The most noticeable change in overall disk

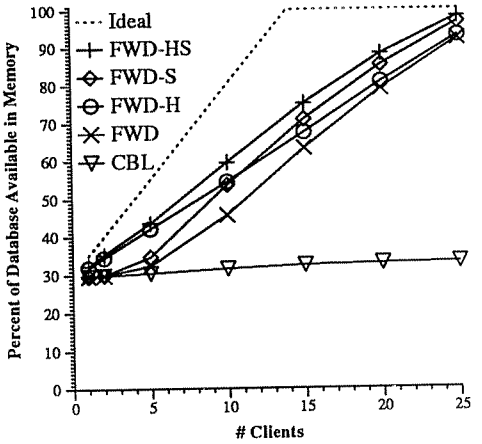

Figure 12: % of DB Available in Memory (RW-HOTCOLD, 5% Cli Bufs, Slow Net)

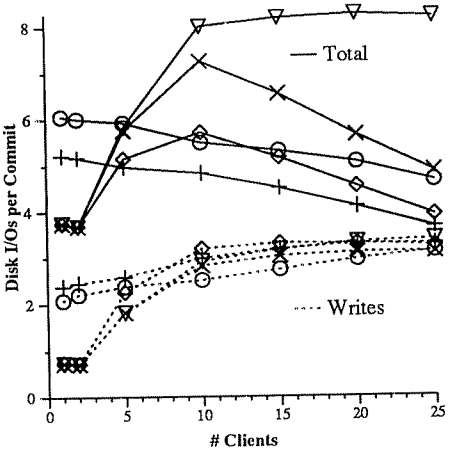

Figure 13: Disk Writes and Total I/O (RW-HOTCOLD, 5% Cli Bufs, Slow Net)

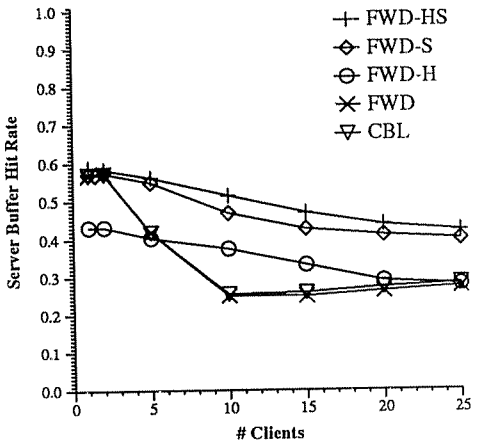

Figure 14: Server Buffer Hit Rate (RW-HOTCOLD, 5% Cli Bufs, Slow Net)

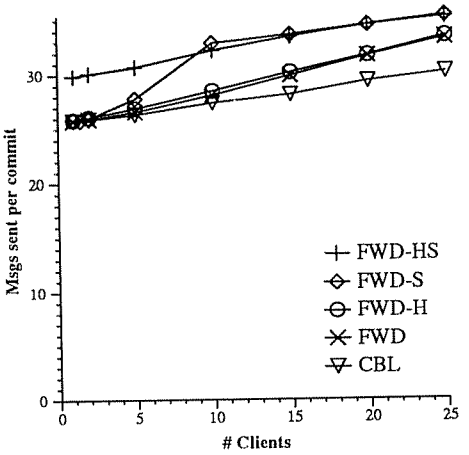

Figure 15: Messages Sent per Commit (RW-HOTCOLD, 5% Cli Bufs, Slow Net)

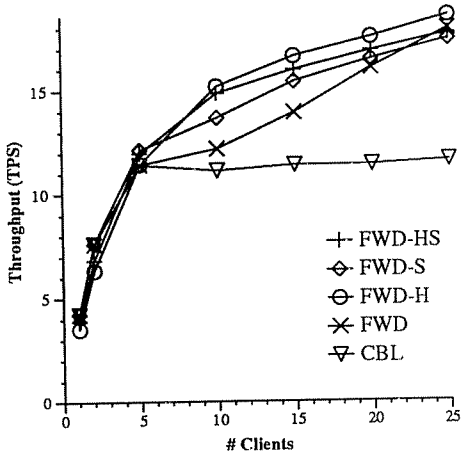

Figure 16: Throughput (RW-HOTCOLD, 5% Cli Bufs, Slow Net)

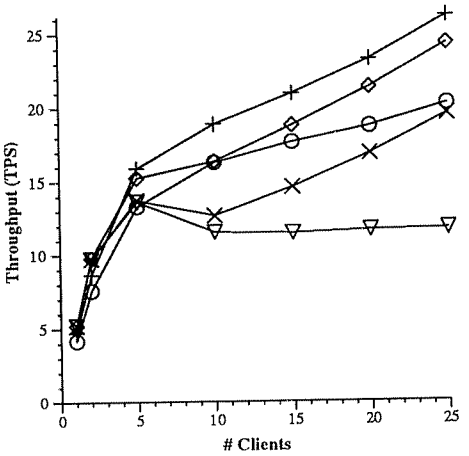

Figure 17: Throughput (RW-HOTCOLD, 5% Cli Bufs, Fast Net)

requirements is that the FWD-HS and FWD-S algorithms no longer enjoy the large advantage that they had in the read-only case (shown in Figure 3). The disk requirements of the algorithms are affected by the need to perform disk writes for dirty pages that are aged out of the server's buffer pool and by the changes in the client and server buffer contents caused by the handling of dirty pages. Disk writes are an additional cost that is incurred by all of the algorithms, however, with 1 to 5 clients the FWD-H and FWD-HS algorithms write more pages than the other algorithms. This is because the hate hints cause requested dirty pages to be moved to the head of the LRU chain, reducing their residence time in the server's buffer pool and thereby reducing the opportunity for combining multiple client writes into a single disk write. As more clients are added, the other algorithms also incur an increase in disk writes due to additional traffic through the server buffer pool. This traffic is caused by disk reads for the FWD and CBL algorithms, and by dropped pages that are sent to the server for FWD-S.

The disk read requirements of the algorithms are changed by the introduction of writes in two ways. First, dirty pages sent to the server by committing transactions impact the server hit rate (shown in Figure 14) in an algorithm-dependent manner. For FWD-H, these pages improve the server hit rate significantly for small client populations (e.g., with 2 clients the server hit rate is 43%, as compared to 30% in the read-only case). The pages dirtied by a client are likely to be hot region pages for that client, so sending those pages to the server helps overcome FWD-H's tendency to remove hot pages from the server's buffer pool. The beneficial effect of the dirty pages is reduced as clients are added to the system. In contrast, FWD-S and FWD-HS suffer a reduction in server hit rate due to dirty pages. A page that is sent to the server because it is dirtied is not necessarily the only copy of the page in the system. Such dirty pages therefore reduce the effective use of the server buffer pool that was exhibited by the sending algorithms in the read-only case. The second way that writes affect the disk read requirements is via callbacks. Callbacks reduce client-client buffer replication, which increases the effectiveness of forwarding, thereby reducing disk read requirements. Therefore the disk read requirements for the FWD algorithm at 15 clients and beyond are somewhat lower than in the read-only case.

Figure 15 shows the number of messages sent per committed transaction. The differences among the algorithms are much smaller here than those seen in read-only case (shown in Figure 4). This occurs for two reasons. First, writes increase the message requirements of all of the algorithms by introducing four new kinds of messages: 1) write lock requests, 2) callback requests for cached read locks, 3) subsequent re-requests for pages and locks that were called-back and 4) messages for dirty pages sent to the server prior to commit. Secondly, the relative message requirements of the FWD-HS algorithm are reduced for small client populations because the dirty pages sent to the server (which are sent by all algorithms) reduce the number of dropped pages that FWD-HS must send to the server.

Figure 16 shows the throughput for the RW-HOTCOLD workload with small client buffer pools and the slow network. The forwarding algorithms all outperform CBL at 10 clients and beyond. This is due once again to CBL's high disk requirements. All of the forwarding algorithms have similar throughput at 25 clients. FWD-H attains the highest throughput at 10 clients and beyond, but has poor throughput with small client populations due to high disk read and write requirements. The sending algorithms, which reach a network bottleneck at 20 clients, perform close to FWD-H due to the reduction in differences in message requirements. FWD-H and FWD both approach, but do not quite reach, a network bottleneck at

25 clients. FWD-HS initially performs poorly due to message costs and disk writes caused by the sending of dropped pages that force dirty "hated" pages out of the server's buffer pool. When the faster network is used, the throughput results (shown in Figure 17) display similar trends but smaller differences compared to what was observed in the read-only case (Figure 9). One difference is that here, all of the algorithms become disk-bound so FWD-H and FWD-HS are impacted by their increased disk requirements (compared to Figure 9).

4.3.2. Read-Write HOTCOLD, Large Client Buffer Pools

Figure 18 shows the throughput results for the Read-Write HOTCOLD workload run with larger client buffer pools and the slow network. Once again, CBL performs at a much lower level than the forwarding algorithms due to its high disk requirements. The forwarding algorithms all have similar throughput; their performance is primarily dictated by their message behavior (not shown), which becomes nearly identical at 15 clients and beyond. This convergence is similar to what was observed in the read-only case (Figure 10) and occurs for the same reasons. The sending algorithms perform closer to the others in this case because the sending of dirty pages and the effect of callbacks result in fewer dropped pages being sent to the server. When the fast network is used in conjunction with the larger client buffer pools (Figure 19), the relative performance of the algorithms is driven by disk demands, and an interesting effect occurs: at 10 clients and beyond, the forwarding algorithms separate into two distinct classes — the algorithms that use hate hints, and those that do not. FWD and FWD-S outperform the other forwarding algorithms because hate hints lead to more disk writes by reducing the amount of time that dirty pages are retained in the server buffer pool. The sending technique has no differentiating effect in this case because few pages are sent back to the server beyond 10 clients.

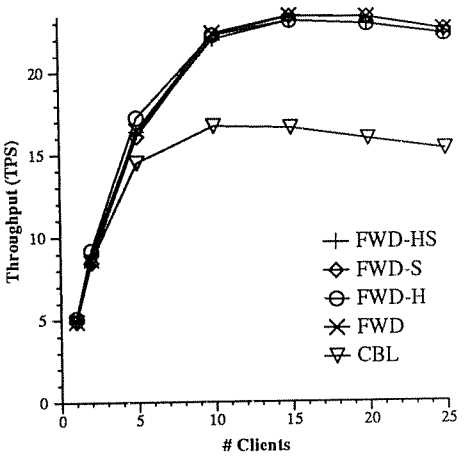

Figure 18: Throughput (RW-HOTCOLD, 15% Cli Bufs, Slow Net)

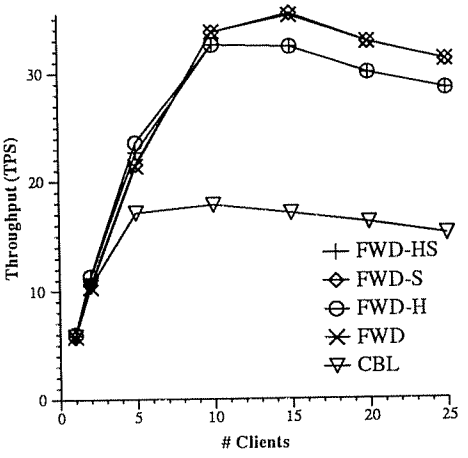

Figure 19: Throughput (RW-HOTCOLD, 15% Cli Bufs, Fast Net)

4.3.3. Summary of the Read-Write HOTCOLD Results

The introduction of writes to the HOTCOLD workload was found to have a number of complex effects on the message and disk requirements of the global algorithms; however, the relative performance

of the algorithms was not greatly affected in most of the cases studied here. When the slow network was used, the most important impact of the writes was a reduction in the differences among the message requirements of the algorithms. This reduction was the result of additional messages incurred by all algorithms and a decrease in the number of dropped pages sent to the server by the sending algorithms. When the fast network was used, the effects of updates on disk requirements played a greater role in determining the relative performance of the algorithms. These effects varied greatly among the algorithms. All algorithms had increased disk requirements due to disk writes. The sending algorithms incurred an additional increase in disk requirements due to a reduction in server buffer hit rates caused by dirty pages sent to the server by committing transactions, which increased the server-client buffer correlation. FWD-H and FWD-HS paid a high price in disk writes when few clients were present because the hate hints reduced the residency time of dirty hot region pages in the server buffer pool. The FWD-S, FWD, and CBL algorithms all saw a slight decrease in disk reads because the dirty pages sent to the server increased their server buffer hit rates. (This was due to dirty pages being re-referenced by clients from which they were previously called-back). Also, the utility of forwarding was slightly improved by the callback mechanism, which reduces client-client buffer replication.

As the preceding discussion indicated, the impact of sending dirty pages to the server on commit varies depending on the memory management algorithms, workload characteristics, and system resources used. In general, however, the sending of dirty pages to the server at commit had a negative impact on performance — requiring additional messages, increasing the cost of hate hints, and interfering with the strategies used for managing the server's buffer pool. The sending of dirty pages to the server at commit time helps to simplify the recovery system in a client-server DBMS [Fran92b], but it could be avoided or reduced at the expense of increasing recovery complexity. As will be discussed in Section 5.2, a similar issue (forcing dirty pages to disk) has been investigated for shared-disk environments [Moha91, Dan92]. As future work, we plan to further investigate policies for handling dirty pages in the client-server environment.

4.4. Experiment 3: PRIVATE Workload

The third experiment that we report here uses the PRIVATE workload. In this workload (see Table 4), each client has a 25-page hot region of the database to which 50% of its accesses are directed; the other 50% of its accesses are directed to a 625-page read-only area of the database. Clients do not access pages in each other's hot regions, so there is no read/write sharing of data in this workload. Thus, the PRIVATE workload has updates and high locality but has no data contention. Figure 20 shows the throughput results for the PRIVATE workload with small client buffer pools and the slow network. In this case, the relative performance of the algorithms is similar to what was seen for the RW-HOTCOLD workload (Figure 16). There is however, an important difference — the relative performance of the sending algorithms is worse here. This is due to higher message requirements resulting from sending more dropped pages to the server. Each client has a private hot region, and thus, when a client drops a hot region page, it will be sent back to the server unless the server already has a cached copy of the page; there is no chance that a copy of such a page will exist at another client. In the fast network case (Figure

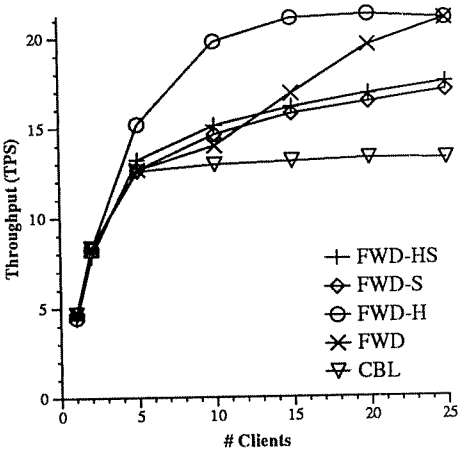

Figure 20: Throughput (PRIVATE, 5% Cli Bufs, Slow Net)

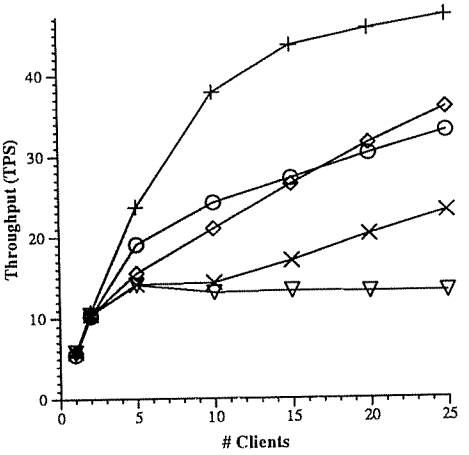

Figure 21: Throughput (PRIVATE, 5% Cli Bufs, Fast Net)

21), the FWD-HS algorithm performs much better than the others because it is effective at keeping more of the database available in memory and thus has lower disk requirements.

Figure 22 shows the throughput results when the client buffer size is increased to 15%. In this case, the client buffer pools are large enough to contain the entire hot region for each client. As a result, any hot region pages that are kept in the server's buffer pool are detrimental to performance because they will not be accessed by any clients and thus, they waste space that could be used for cold region pages. In this situation, the hate hints will tend to remove cold region pages from the server's buffer pool while retaining dirty hot region pages, resulting in the poor server hit rates shown in Figure 23. The other forwarding algorithms (FWD and FWD-S) also have poor server hit rates due to the presence of dirty hot region pages in the server's buffer pool. The sending technique gives FWD-S a slightly better server hit rate than FWD with small numbers of clients, as it causes some cold region pages to be sent back to the server. The CBL algorithm has the best server hit rate at 10 clients and beyond due to the presence of

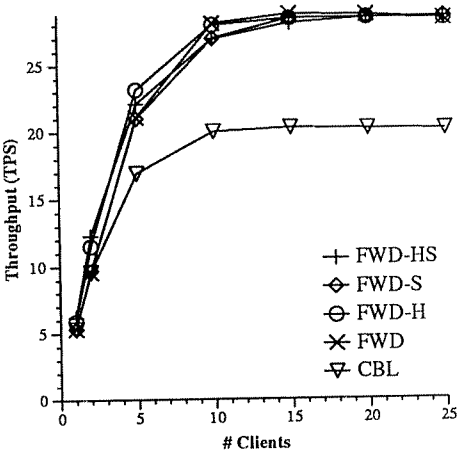

Figure 22: Throughput (PRIVATE, 15% Cli Bufs, Slow Net)

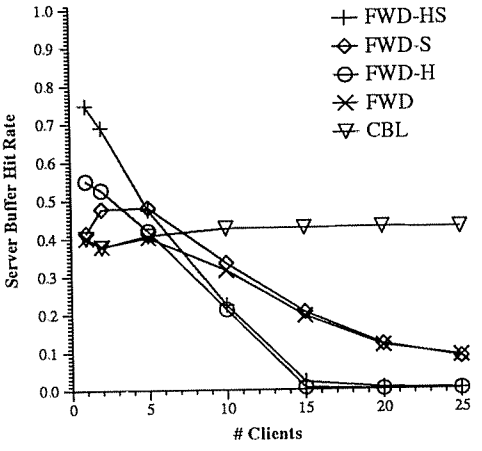

Figure 23: Server Buffer Hit Rate (PRIVATE, 15% Cli Bufs, Slow Net)

cold region pages that are read in from disk. The other algorithms do not read as many cold region pages from disk since they are able to satisfy many server misses by forwarding requests.

While the interaction of the workload with the handling of dirty pages results in the interesting buffering effects just described, the net effects on performance are not as great as might be expected. In the slow network case (Figure 22), there are only small differences among the performance of the algorithms. In this case the performance results for the forwarding algorithms are primarily dependent on the message behavior of those algorithms, and they all eventually have similar message behavior. The CBL algorithm becomes disk-bound despite its superior server hit rate, while the forwarding algorithms perform virtually no disk reads at 15 clients and beyond. In the fast network case (not shown), the forwarding algorithms become server CPU bound at 15 clients and converge shortly thereafter, at about 3.5 times the throughput of the disk-bound CBL algorithm.

4.5. Experiment 4: UNIFORM Workload

As described in Table 4, the UNIFORM workload has no locality and has a write probability of 20%. With small client buffers and the slow network (shown in Figure 24), FWD-H performs the best until it is matched by the FWD algorithm at 25 clients. Initially, FWD-H's advantage is due to a high server buffer hit rate. In this case, the hate hints improve the server buffer hit rate for small client populations by reducing server-client correlation. In contrast to what was observed in the skewed workloads, decreasing server-client correlation actually improves the server buffer hit rate here, since all pages are equally likely to be accessed. As more clients are added, FWD-H's advantage is the result of it having lower message requirements than the sending algorithms. The lack of locality reduces the importance of the sending technique, so FWD-S keeps a smaller portion of the database in memory than FWD-H until 25 clients (at which point they are equal). FWD keeps the smallest portion of the database in memory among the forwarding algorithms in this case, but its lower message requirements allow it to perform relatively well with larger numbers of clients.

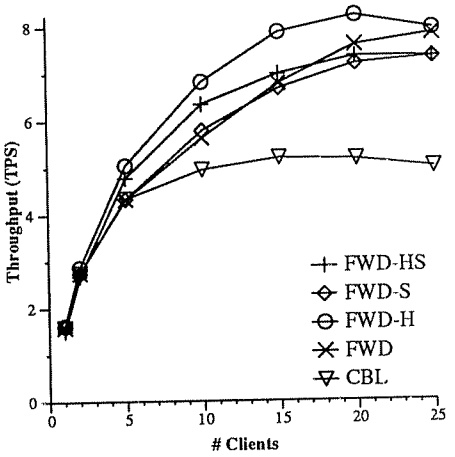

Figure 24: Throughput (UNIFORM, 5% Cli Bufs, Slow Net)

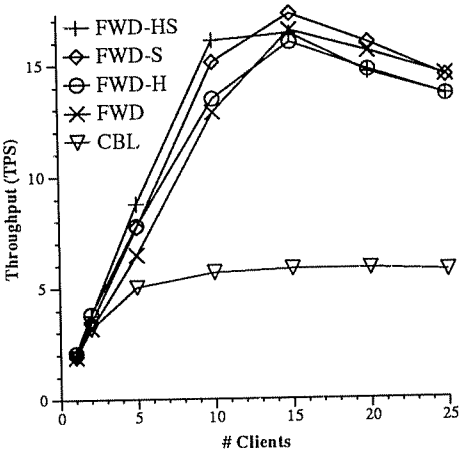

Figure 25: Throughput (UNIFORM, 15% Cli Bufs, Fast Net)

With larger client buffer pools and the fast network (shown in Figure 25), the forwarding algorithms begin to approach the bifurcated state that they reached in the RW-HOTCOLD case (Figure 19). However, there are two noticeable differences in the trends compared to the RW-HOTCOLD case. First, prior to 20 clients, the FWD-HS algorithm performs better than FWD-H, and second, prior to 25 clients, the FWD-S algorithm performs better than FWD. At 25 clients, all of the forwarding algorithms have access to the entire database in memory. However, prior to this point, FWD-HS keeps a larger portion of the database available in memory than FWD-H, and FWD-S keeps a larger portion of the database available in memory than FWD. In this case, the ability to keep more of the database available in memory results in noticeably better performance. Once all of the forwarding algorithms have the entire database available in memory, FWD-H and FWD-HS pay a slight penalty due to additional disk writes caused by hate hints.

5. RELATED WORK

In this section we briefly discuss work related to global memory management in client-server DBMSs, data sharing DBMSs, and other distributed systems.

5.1. Workstation-Server Database Systems

Several recent papers have investigated issues of global memory management for DBMSs in a workstation-server environment. In [Leff91], the problem of replica management for efficient use of the global memory resources of a distributed system was addressed. In particular, an analytical model was used to investigate the tradeoffs between a "greedy" algorithm, where each site makes its own caching decisions to maximize its own performance, and two algorithms where caching decisions are made (statically) in order to maximize overall system performance. The latter two algorithms were intended to provide an upper bound on the potential performance improvement that could be expected due to global memory management, and not as practical algorithms for implementation. The key tradeoff found was that without coordination, all sites chose to cache the hottest objects, while the coordinated strategies were able to keep many different pages in memory, thereby replacing disk I/Os with (cheaper) messages. The study did not consider updates and all sites had the same database reference patterns. Our algorithms also attempt to keep more pages available in memory, but they focus on using the server buffer as a mechanism for achieving this rather than modifying client buffer replacement policies. Algorithms for using the memory of underutilized workstations (called *mutual-servers*) to keep more of the database in memory were proposed and studied in [Pu91]. The algorithms included variations in which the sender and/or the receiver played active roles in initiating the caching of a page at a mutual-server. All of the algorithms were broadcast-based, and the study did not consider concurrency control or data contention issues.

5.2. Transaction Processing Systems

Issues related to some of the global memory management techniques discussed here are addressed in a data-sharing context in several papers from IBM Yorktown. In one paper [Dan91] an analytical model was used to study a two-level buffer hierarchy. The paper investigated policies for placing pages in a shared buffer based on when pages are updated, read in from disk, and/or aged-out of a private buffer.

The study identified a number of buffer correlation effects similar to those discussed in Section 4, especially those dealing with the relative sizes of the shared (server) and private (client) buffers. There are several important differences between the two studies, however. First, the IBM study did not investigate global algorithms that allowed shared-buffer misses to be serviced by private buffers. Second, it compared the techniques based on their buffer hit rates but did not investigate the impact of the algorithms on actual transaction throughput. Many of the experiments discussed in Section 4 showed that the buffer hit rates often did not determine the performance of the algorithms. Third, due to the nature of the shared-disk environment, the IBM study investigated workloads in which there was no difference among the access patterns at the various private buffers. Therefore, the complex interaction of server-client correlation and server hit rates that arose here in the HOTCOLD workloads were not identified.

A more recent paper [Dan92] studies callback-style shared-disk caching algorithms and investigates the performance gains that are available by avoiding disk writes for dirty pages when transferring a page between sites. Adding this optimization to a shared-disk system results in more complex recovery schemes, as described in [Moha91]. An algorithm which avoids replicating copies at multiple sites was studied and was found to have tradeoffs similar to some of those seen for our forwarding techniques. However, there are significant differences between the client-server environment and the data-sharing environment (e.g, the use of the server for logging and recovery [Fran92b], and the expense of messages in a client-server system). Thus, many of the tradeoffs that we have identified in a client-server context differ from those observed in data-sharing systems.

Another recent paper [Rahm92] studies the use of several types of extended memory to improve the performance of transaction processing systems. Extended memory adds a new memory hierarchy level between main memory and disk. An added dimension is the use of non-volatile memory to avoid data and log disk writes. The paper also investigates other types of extended memory such as solid-state disks and disk caches. Many of the correlation issues found in our study (and in [Dan91]) also arise in this environment. In particular, the correlation caused by forcing dirty pages to the extended memory (similar to our copying of dirty pages back to the server), was shown to reduce the effectiveness of the extended memory. This study differs from ours because of fundamental differences in the system architecture; these include the use a single main memory buffer (versus multiple clients), lower communication costs between the levels of the hierarchy, and differences in the types of workloads studied.

5.3. Non-DBMS Approaches

Issues related to global memory management have also been addressed in distributed object systems such as Emerald [Jul88], where methods for allowing objects to migrate among sites were addressed. Migration in this case was intended to improve performance by bringing objects to the sites where they were being accessed, and also to simplify the programming of distributed applications, rather than to avoid disk I/O. The idea of using the memory of idle workstations as a backing store for virtual memory was investigated in [Felt91]. This paper raised several policy issues and presented a simple queueing model study that indicated that the approach has potential for significant performance gains over swapping to disk, even with current networking and OS technology. Finally, work in Non-Uniform Memory Access architectures, in which the memories of nodes in a multiprocessor system are viewed as a single

memory hierarchy, is relevant as well. This work includes [LaRo90], which studied the performance of a wide range of proposed memory management policies and [LaRo91], which investigated dynamic policies that can be adapted to particular page reference behaviors.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have studied performance tradeoffs for global memory management in page-server database systems. Three different memory management techniques were presented. Each of the proposed techniques can be implemented within the context of existing client-server DBMS data and lock caching algorithms, and they require no information to be kept at the server or at the clients beyond what is already required by such algorithms. The primary technique, forwarding, attempts to avoid disk I/O at the server by forwarding page requests to remote clients that have a requested page in their buffer pool. Forwarding allows the buffer pools of remote clients to be treated as an additional level in a global memory hierarchy. Accesses to this additional memory level incur a cost of one control message and some additional client processing beyond the cost of a request made to the server, so it lies between the server memory and the disks in the hierarchy.

The two other techniques are intended to increase the utility of forwarding by more efficiently managing the global memory of the system. These techniques attempt to exploit the server's buffer pool in order to keep a larger portion of the database available in the global memory. One technique, called hate hints, is a simple heuristic which tries to reduce replication between the buffer pool contents of the server and its clients. In this technique, the server's buffer replacement policy is modified to "hate" a page when it sends a copy of that page to a client. The other technique, called sending dropped pages, attempts to retain pages in memory by keeping a client from simply dropping a valuable page from its buffer pool. By piggybacking information on other messages, a client informs the server of its intention to drop a page. If the server then determines that the page to be dropped is the only copy of that page in global memory, it asks the client to send the page back rather than drop it.

These three techniques were compared under a range of workloads and system configurations using a simulation model. The results of the performance study show that, as expected, forwarding can provide significant performance gains over a non-forwarding cache management algorithm. The study also showed that the hate hints and sending techniques were indeed effective in keeping a larger portion of the database in memory. However, while these techniques achieved their objectives, they did not always yield performance improvements and in some cases were even detrimental to performance. For example, the hate hints technique was successful at reducing replication, but in some situations it removed valuable pages from the server buffer pool — thereby increasing I/O demands. The sending technique was found to be expensive in network-constrained situations in which using messages to avoid disk I/Os is the wrong approach to take. However, in many situations the sending and hate hints techniques were both shown to provide substantial performance gains. The study also investigated the impact of updates on the buffering behavior and performance of the algorithms, and it identified issues in the interaction of global memory management and the management of dirty pages for supporting transaction durability.

A number of areas for future work were raised by this study. First, the experiments identified situations where the global techniques caused the system to become unbalanced or perform extra work. For

example, there were cases in which the disk became underutilized while the network became saturated due to the effectiveness of the forwarding technique. Another example was the interaction of hate hints and dirty pages in the server buffer, which caused an increase in the number of disk writes performed per transaction. These situations demonstrate the need for algorithms that can adapt to the resource and memory usage patterns of the system. Several extensions along these lines could be easily added to the existing techniques. Another important area for future study is the recovery and performance implications of techniques that would avoid having clients send dirty pages to the server prior to committing a transaction. Finally, we also plan to investigate global memory management algorithms that take a more active role in determining where pages should reside in the system. Such algorithms could control issues such as the amount of replication allowed at various levels in the system and the placement of page copies in the memories of idle or under-utilized workstations.

REFERENCES

[Care91] Carey, M., Franklin, M., Livny, M., Shekita, E., "Data Caching Tradeoffs in Client-Server DBMS Architectures", *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Denver, CO, June, 1991.

[Chen84] Cheng, J., Loosley, C., Shibamiya, A., Worthington, P., "IBM Database 2 performance: Design, Implementation, and Tuning", *IBM Systems Journal*, Vol. 23, No. 2., 1984.

[Dan91] Dan, A., Dias, D., Yu, P., "Analytical Modelling of a Hierarchical Buffer for a Data Sharing Environment", *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, San Diego, CA, May, 1991.

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

[Deux91] Deux, O., *et al.*, "The O2 System", *Communications of the ACM*, Vol. 34, No. 10, Oct., 1991.

[DeWi90] DeWitt, D., Futtersack, P., Maier, D., Velez, F., "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems," *Proc. 16th Int'l Conf. on Very Large Data Bases*, Brisbane, Aug., 1990.

[Exod91] EXODUS Project Group, *EXODUS Storage Manager Architectural Overview*, EXODUS Project Document, University of Wisconsin - Madison, Nov., 1991.

[Felt91] Felten, E., Zahorjan, J., "Issues in the Implementation of a Remote Memory Paging System", *Tech. Rept. 91-03-09*, University of Washington, March, 1991.

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

[Fran92b] Franklin, M., Zwilling, M., Tan, C., Carey, M., DeWitt, D., "Crash Recovery in Client-Server EXODUS", *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, San Diego, CA, June, 1992.

[Horn87] M. Hornick and S. Zdonik, "A Shared, Segmented Memory System for an Object-Oriented Database," *ACM Trans. on Office Information Systems* 5, 1, Jan., 1987.

[Howa88] Howard, J., *et al*, "Scale and Performance in a Distributed File System," *ACM Trans. on Computer Systems* 6, 1, Feb., 1988.

[Jul88] Jul, E., Levy, H., Hutchinson, N., Black, A., "Fine Grained Mobility in the Emerald System", *ACM Trans. on Computer Systems*, Vol. 6, No. 1, February, 1988.

[Lamb91] Lamb, C., Landis, G., Orenstein, J. Weinreb, D., "The ObjectStore Database System", *Communications of the ACM*, Vol. 34, No. 10, Oct., 1991.

[LaRo90] LaRowe, P., Ellis, C., "Experimental Comparison of Memory Management Policies for NUMA Multiprocessors", *Tech. Rep. CS-1990-10*, Duke University, April, 1990.

[LaRo91] LaRowe, P., Ellis, C., Kaplan, L, "The Robustness of NUMA Memory Management", *Proc. 13th ACM Symp. on Operating Systems Principles*, Pacific Grove, CA, Oct., 1991.

[Leff91] Leff, A., Yu, P., Wolf, J., "Policies for Efficient Memory Utilization in a Remote Caching Architecture", *Proc. 1st Int'l Conf. on Parallel and Distributed Information Systems*, Miami Beach, FL, Dec., 1991.

[Moha91] Mohan, C., Narang, I., "Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment", *Proc. 17th Int'l Conf. on Very Large Data Bases*, Barcelona, Sept., 1991.

[Pu91] Pu, C., Florissi, D., Soares, P., Yu, P., Wu, K., "Performance Comparison of Sender-Active and Receiver-Active Mutual Data Serving", *Tech. Rept. CUCS-014-090*, Columbia University, 1991.

[Rahm92] Rahm, E., "Performance Evaluation of Extended Storage Architectures for Transaction Processing", *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, San Diego, CA, June, 1992.

[Sarg76] Sargent, R., "Statistical Analysis of Simulation Output Data," *Proc. 4th Annual Symp. on the Simulation of Computer Systems*,1976.

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

[Wilk90] Wilkinson, W., and Neimat, M.-A., "Maintaining Consistency of Client Cached Data," *Proc. 16th Int'l Conf. on Very Large Data Bases*, Brisbane, Aug., 1990.