Data Caching Tradeoffs in Client-Server DBMS Architectures

by

Michael J. Carey
Michael J. Franklin
Miron Livny
Eugene J. Shekita

Computer Sciences Technical Report #994
January 1991

Data Caching Tradeoffs in Client-Server DBMS Architectures

Michael J. Carey
Michael J. Franklin
Miron Livny
Eugene J. Shekita

Computer Sciences Department
University of Wisconsin

This research was partially supported by the Defense Advanced Research Projects Agency under contracts N00014-88-K-0303 and NAG-2-618 and by the National Science Foundation under grant IRI-8657323.

Data Caching Tradeoffs in Client-Server DBMS Architectures

Michael J. Carey, Michael J. Franklin,
Miron Livny, Eugene J. Shekita
Computer Sciences Department
University of Wisconsin

ABSTRACT

In this paper, we examine the performance tradeoffs that are raised by caching data in the client workstations of a client-server DBMS. We begin by presenting a range of lock-based cache consistency algorithms; these algorithms arise by viewing cache consistency as a variant of the well-understood problem of replicated data management. We then use a detailed simulation model to study the performance of these algorithms over a wide range of workloads and system resource configurations. The results illustrate key performance tradeoffs related to client-server cache consistency, and should be of use to designers of next-generation DBMS prototypes and products.

1. INTRODUCTION

With networks of powerful workstations becoming commonplace in scientific, engineering, and even office computing environments, *client-server* software architectures have become a common approach to providing access to shared services and resources. Most commercial relational database management systems today are based on client-server architectures, with SQL queries and their results serving as the basis for client-server interactions [Ston90a]. In the past few years, a number of object-oriented DBMS (OODBMS) prototypes and products have appeared, virtually all of which are based on client-server architectures. Compared to relational database systems, these systems generally take a different approach when it comes to client-server interactions. In order to offload processing to client workstations, it is common for client-server interactions in an OODBMS to take place at the level of individual objects or pages of objects rather than queries [DeWi90]. Prototypes based on object-level interactions include Orion [Kim90] and O2 [Deux90], among others. Among the prototypes based on page-level (or multi-page block) interactions are ObServer [Horn87] and the current version of the EXODUS storage manager [Care89a, Zwil90]; the ObjectStore system from Object Design [ODI90] is an example of a commercial OODBMS product based on page-level interactions.

In architectures where data pages or objects are the basis for client-server interactions, it is possible to cache data in the local memories of client workstations for later reuse.[1] Such caching can reduce the need for client-

This research was partially supported by the Defense Advanced Research Projects Agency under contracts N00014-88-K-0303 and NAG-2-618 and by the National Science Foundation under grant IRI-8657323.

[1] If the client workstations have local disks, it is also possible to cache data on secondary storage, as in the Andrew file system [Howa88]. Local disk caching is beyond the scope of this paper, however, and will not be discussed further.

server interaction, lessening the network traffic and message processing overhead for both the server and its clients. It also enables client resources to be used by the DBMS, thus increasing both the aggregate memory and the aggregate CPU power available for database-related processing. An increase in the aggregate memory of the DBMS can reduce the I/O load on the server, while an increase in the aggregate CPU power available to the DBMS can reduce the load on the server CPU(s). Depending on the nature of the applications — including their balance of I/O and CPU demands, their locality of access, and the proportional cost of their DBMS accesses relative to their overall computational requirements — increasing the aggregate resources of the DBMS can result in significant performance improvements. Of course, in applications where the response time is dominated by the time spent at the client CPU, or where a large fraction of the DBMS is accessed relative to the size of the client buffer pool, the performance benefits of caching would be negligible.

Despite its potential, caching is not a performance panacea. In order to incorporate caching, the DBMS must include a protocol that ensures cache consistency. Such a protocol may be complex to implement, it may entail a significant amount of processing overhead, and its impact on system performance may be workload-dependent. Depending on how the protocol and the workload interact, the cache consistency protocol might actually increase the load on the server and/or the client workstations due to its overhead, particularly when there are a large number of client workstations. Another potential pitfall, which depends on the concurrency control scheme used by the protocol, is the late discovery of data conflicts. Thus, the potential consequences of adding caching to a client-server DBMS range from a significant improvement in performance to a notable degradation in performance. An example of a workload where caching can be highly beneficial is the Sun Engineering Database Benchmark [Catt90a].

In this paper, we examine the data caching performance tradeoffs discussed above. We begin by presenting a range of lock-based cache consistency algorithms that result from recognizing that cache consistency is simply a variant of the replicated data management problem studied by distributed DBMS researchers. For concreteness, we focus our attention on systems where client-server interactions are page-based. This approach, also referred to as the *block server* approach [Ston90b], was shown to perform well for CAD-style data access patterns in a recent performance study [DeWi90]. Also, our work was motivated by a desire to understand performance tradeoffs in our own page-based, client-server storage managers [Care89a, Shek90, Zwil90]. Given this set of cache consistency algorithms, we then describe a detailed simulation model that was developed to study their performance over a wide range of workloads and system resource configurations.

The performance of transaction-oriented cache consistency algorithms has been examined in several related contexts. The only other client-server data caching study that we know of is a recent simulation study at HP Laboratories [Wilk90]. Our work differs from their work in several ways. First, we employ a much more detailed model of buffering, the importance of which will be clear from our results. Second, we study a broader range of DBMS workloads. Our work is also related to studies of shared-disk architectures, including the work by Bhide [Bhid88] and by a DBMS performance group at IBM Yorktown (reported in [Yu87, Dan90] and other related papers). Again, our work differs from these efforts in several ways. First, shared-disk and client-server DBMS architectures are qualitatively different. In a client-server DBMS, clients must interact with the server in any case,

so the server is a natural center of activity that is also available to assist in cache management. Also, shared-disk DBMS configurations tend to involve relatively few machines. Second, we study a range of cache consistency algorithms that goes beyond those found in the shared-disk literature, including algorithms that propagate changes to other caches rather than invalidating other cached copies. The final example of closely related work is the recent Harvard work on transaction-oriented distributed memory hierarchies [Bell90]. In contrast to our work, their work assumes a decentralized, shared-nothing architecture and a communications network with hardware broadcast support, yielding a very different set of resource-related performance tradeoffs. In addition, it should be noted that our work is also loosely related to studies of multiprocessor cache coherency algorithms (e.g., [Arch86]) and to work on caching in distributed file systems (e.g., [Howa88, Nels88, Gray89]). We will return to the topic of related work again once we have presented our performance results. To summarize, the main contributions of this paper include viewing client-server cache consistency as a replica management problem, studying a range of algorithms that naturally arise from this view, using a detailed simulation model to do so, and exploring a wide range of workloads and system resource configurations.

The remainder of the paper is organized as follows: Section 2 describes our architectural assumptions and presents the cache consistency algorithms that form the basis for our performance study. Section 3 describes our simulation model. Section 4 describes our simulation experiments and results. Finally, Section 5 summarizes our results and discusses our plans for future work on client-server DBMS architectures.

2. CACHE CONSISTENCY ALGORITHMS

In this section, we present the cache consistency algorithms that are the focus of this paper. Before doing so, however, we review the page (or block) server approach to a client-server DBMS and explain how cache consistency is related to replica management in a distributed DBMS.

2.1. Page Server Architecture

The general architecture of a caching client-server DBMS is depicted in Figure 1. The system consists of a database server which is connected to N client workstations via a local area network. The system's secondary storage, which includes one or more disks on which the database is stored and a (mirrored) disk for the log, is connected to the server. The software of the DBMS includes components that reside on both the server and the clients, so applications running on client workstations can view the DBMS essentially as a locally available service. The DBMS has buffer pool space available on both the server and on the client workstations, and it is free to manage this space as it sees fit in order to optimize the overall performance of the DBMS. In the page server architecture, pages (or multi-page blocks) are the unit of client-server data transfer and also serve as the unit of data caching.

Process-wise, we assume that each client application (CA) runs as a process (i.e., in an address space) that is separate from the DBMS. We further assume that the DBMS itself consists of a multi-threaded database server (DS) process and a collection of multi-threaded client database (CD) processes. It is also assumed that there are one or more client application processes and exactly one database client process per client workstation. We will not

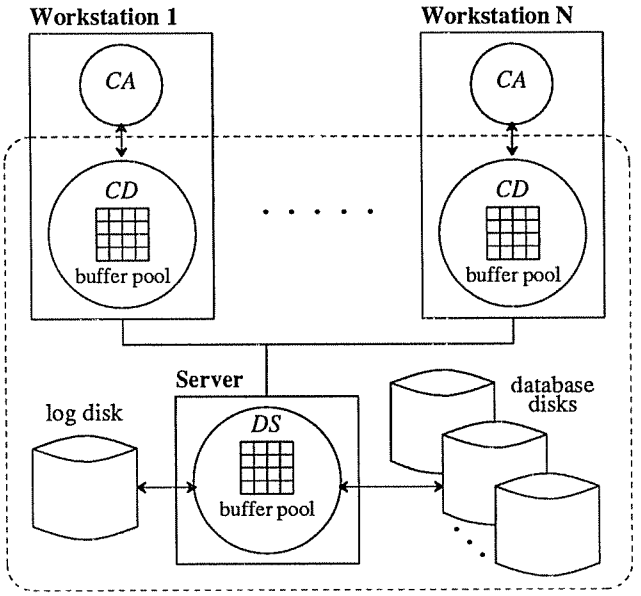

Figure 1: Cache Architecture of a Client-Server DBMS

concern ourselves with the details of how the client application and database processes interact within a client workstation; this is system-dependent, and may be handled through database library calls (based on local IPC or shared memory) or through virtual memory assisted page-faulting (a la [Shek90]). All that matters here is that client applications can somehow submit requests to the client database process in order to control (i.e., begin, commit, and abort) transactions and to read and write objects in the database. In addition, the database client and server processes can communicate both synchronously and asynchronously with respect to client application processes in order to handle cache misses, updates, and cache consistency. Finally, since the database client processes are separate from client application processes, their state outlives client transactions. Thus, they are free to cache data both within and across transaction boundaries as long as system-wide cache consistency is maintained.

Other recent work on client-server cache consistency [Wilk90] has approached the problem from first principles. In that work, two algorithms were developed and analyzed. One of the algorithms was developed by viewing cached pages as snapshots of server pages and characterizing them according to their current state relative to the server's version; this is similar to approaches found in multiprocessor cache coherency algorithms [Arch86]. The other algorithm is based on an analogy with notification ideas from the active database area. Both of these algorithms required that certain conditions be checked and later rechecked in order to avoid potential race conditions between transactions. Given the general system model described above, a cleaner approach can be arrived at by recognizing that cached data are just like replicated data in a distributed DBMS, though they reside in main memory rather than on disk. Thus, all of the well-known results on replica management theory and algorithms [Bern87] can be applied to the client-server cache consistency problem.

Given this observation, we now present five candidate algorithms for maintaining client-server cache con-

sistency.[2] The first algorithm is a basic two-phase locking scheme, based on the primary copy approach to replica management [Bern87], in which data is not cached between client transactions. The second algorithm extends the first to allow for inter-transaction data caching. The other three algorithms are each based on an optimistic variant of two-phase locking, studied in [Care89b], where updates to remote copies of replicated data are deferred until end-of-transaction. One of the algorithms handles updates by invalidating all remotely cached copies, while another handles updates by propagating new data values to the remote caches (as is done for replicated data in a distributed DBMS); the final algorithm takes a dynamic approach, choosing between invalidation and propagation on a per-page basis.

In all five algorithms, the client database process must request data from the database server if a cache miss occurs, and data can be safely cached within the context of a single transaction; it should not be necessary for a transaction to re-fetch the same data again unless its working set is large relative to the available client buffer space. Also, in all algorithms, committing a transaction involves sending a commit message with copies of all updated pages and their associated log records back to the server. This allows the server to handle future requests for the modified data directly — even if the client workstation is turned off or crashes in the meantime.

2.2. Basic Two-Phase Locking (B2PL)

The first algorithm is a primary copy locking algorithm [Bern87] in which the client database process discards cached data between transactions. Transactions set read locks on the data pages that they access, upgrading their read locks to write locks if an item is to be updated. All first-time lock requests are sent to the server, which serves as the primary copy site. Subsequent requests for locks on pages that the transaction has already locked do not require any interaction with the server as long as the mode of the new request is same as that of the existing lock. Read lock requests and page access requests are combined, and the server's response is to return the requested page after obtaining a read lock on the page for the requesting client transaction. All locks are held until the transaction either commits or aborts. As usual, it is possible for deadlocks to arise, and these are handled through centralized deadlock detection on the server when lock waits occur. Deadlock resolution involves aborting the transaction with the most recent initial startup time among those involved in the deadlock. This algorithm is of interest because it is a simple starting point, it is currently in use in the client-server EXODUS storage manager [Zwil90], and it was also used as a baseline algorithm in [Wilk90].

2.3. Caching Two-Phase Locking (C2PL)

The second algorithm is a refinement of B2PL in which inter-transaction data caching is permitted. As in B2PL, all locking and deadlock detection duties are the responsibility of the server. Thus, all first-time lock requests require a round-trip message interchange between client and server database processes. Unlike B2PL, however, the contents of the client buffer pool are retained across transactions. Despite this retention, C2PL

[2] Note that, for the remainder of this paper, we shall assume that pages form the unit of buffering and cache consistency. The algorithms that will be described, however, easily extend to the case of multi-page blocks. We will also focus our attention strictly on the caching of data; index pages must be handled via a separate mechanism in order to support the necessary level of concurrent activity.

guarantees that transactions always read valid data by having the server piggyback updated copies of pages, when necessary, on reply messages to lock requests. In order for the server to know when an updated page must be supplied, a lock request includes the locally known log sequence number (LSN) of the page if the page is cached at the requesting client. The server compares this LSN with the page's true LSN to determine if the cached copy is still valid.

To facilitate the LSN check, the server maintains a table containing the LSNs of all pages that are currently cached on one or more client workstations. It does so by recording the LSN when it provides a page to a client, and clients inform the server when they no longer have a copy of a given page. For example, whenever a client selects a clean cached page for replacement, it simply discards the page and then notifies the server as soon as it is conveniently possible to do so. This is accomplished by piggybacking a list of recently discarded pages on the next message that it sends to the server.

2.4. Optimistic Two-Phase Locking (O2PL)

The next three algorithms, referred to collectively as the O2PL family of cache consistency algorithms, are all based on a read-one/write-all [Bern87] optimistic locking scheme studied in [Care89b] for distributed replica management. These algorithms differ from C2PL in that, prior to transaction commit time, clients set read and write locks locally without obtaining locks at the server. Moreover, a client cache miss causes the client to request the page from the server, as usual, but a read lock is acquired and held on the server only long enough for the server to obtain a stable copy of the page to send back to the client. The server keeps track of which client caches have current copies of which pages. Note that optimistic locking schemes have also been proposed for shared-disk systems, for example, the semi-optimistic "pass-the-buck" locking scheme of [Yu87].

In the O2PL algorithms, client updates are performed locally, but they are are not permitted to migrate back to the server's buffer pool until the associated update transaction enters its commit phase. At that time, the client database process associated with the update transaction sends a commit message to the server containing a copy of each page that has been updated by the transaction; the server then acquires update-copy locks (similar to write locks) on these pages on behalf of the update transaction. Once these locks have been acquired, the server sends a prepare-to-commit message to all other client database processes that contain cached copies of any of the updated pages. These client database processes request update-copy locks on the updated data on behalf of the committing transaction.[3] Once all of the relevant update-copy locks have been obtained, variant-specific O2PL actions are taken.

Update-copy locks are exclusive locks that enable certain deadlocks to be detected early. Being exclusive locks, if another transaction holds a read lock on a page when an update-copy lock is requested for it, the committing update transaction will wait until the reader completes. A conflict between an update-copy lock and a write lock indicates an impending distributed deadlock, and it can be resolved as such without further delay [Care89b]. Other deadlocks, including distributed deadlocks, are possible; they are dealt with by having the client and server

[3] If an update-copy lock request is made for a page that is no longer cached at a given client site, the site simply ignores this lock request.

database processes check for local deadlocks, and by having the server periodically check for distributed deadlocks a la [Ston79].

2.4.1. Update Invalidation (O2PL-I)

In the invalidation variant of O2PL, the variant-specific action is the invalidation of other cached copies of updated pages. That is, a committing update transaction acquires update-copy locks on all copies (i.e., at the server and at any clients) of the updated pages. At the server, these locks enable the committing transaction to safely install its updates. On other clients, however, they enable it to safely invalidate cached copies of the page. Once all updated pages have been invalidated, these other clients send a prepared-to-commit message back to the server, release their update-copy locks, and then drop out of the commit protocol. The server can commit the update transaction when all sites containing cached copies of the updated data have responded, at which point only the server and the client that originated the update have copies of the updated data.

2.4.2. Update Propagation (O2PL-P)

In the propagation variant of O2PL, the variant-specific action is the propagation of updates to other cached copies of updated pages. Thus, the O2PL-P algorithm keeps all clients informed of any changes made to the data resident in their local caches. As in O2PL-I, a committing update transaction acquires update-copy locks on all copies of pages to be updated. In this case, however, these locks are used to enable the committing transaction to safely install its updates on every machine that holds a copy of the updated data. Since updates must be installed on the server and all clients atomically, O2PL-P employs a two-phase commit protocol rather than the one-phase commit that suffices for O2PL-I. Also, the prepare-to-commit messages that the server sends to clients in this case must include copies of the relevant updates; these updates are installed during the second phase of the commit protocol to avoid overwriting valid cached pages before the outcome of the update transaction is certain.[4]

2.4.3. A Dynamic Algorithm (O2PL-D)

The O2PL-I and O2PL-P algorithms were motivated by different workloads. As we will show in Section 4, each algorithm provides significant performance benefits under the right conditions. The dynamic variant of O2PL attempts to invalidate cached copies of data when invalidation is appropriate and to propagate changes when doing so seems more beneficial. This dynamic algorithm, O2PL-D, propagates updates like O2PL-P unless it detects that it is doing so too frequently. In O2PL-D, an update to a page will lead to an invalidation of the page instead of a change propagation if a caching client notices that (i) it has already propagated a change to this page, and (ii) the page has not been re-referenced by the client since that time. Clients that do no propagation in response to a prepare-to-commit message from the server can drop out of the commit protocol at the end of the first phase, as in O2PL-I.

[4] Note: The installation of these updates has no affect on the position of the updated pages in the clients' LRU chains.

2.5. Performance Tradeoffs

We have presented five algorithms for client-server cache consistency, all based (in one way or another) on viewing cached pages as replicated data. While Section 4 will present results from a quantitative study of the tradeoffs between the various algorithms, it is worthwhile to consider some of their qualitative differences. B2PL is the simplest approach, and will serve as a baseline against which to evaluate the other approaches. C2PL, which extends B2PL to support caching across transaction boundaries, extends the aggregate memory of the DBMS to include the buffer space on the client workstations. In contrast to B2PL and C2PL, which require the server to handle all lock requests, the O2PL algorithms extend C2PL by taking a more optimistic approach. The O2PL algorithms allow client transactions to execute entirely locally between cache misses, communicating with the server and with other client workstations only at commit time (to handle updated pages); this implies that transactions that manage to run without cache misses can execute with no server interactions until they reach their commit point. Among the O2PL algorithms, O2PL-I invalidates other cached copies of updated pages at this point, whereas O2PL-P propagates changes to these copies. O2PL-D is a simple dynamic algorithm that attempts to combine the best features of these two static O2PL variants. Finally, compared with B2PL and C2PL, all three O2PL variants allow additional concurrency since they detect data conflicts later; of course, this can lead to more aborts.

3. MODELING A CLIENT-SERVER DBMS

In order to study client-server caching, we have constructed a detailed simulation model of a client-server DBMS. The structure of our simulation model is based on the system architecture that was shown earlier in Figure 1. In this section, we describe how the model captures the database, workload, and various physical resources of a client-server DBMS. Certain aspects of the system, such as consistency control and buffer management, are modeled in their full detail; other aspects, such as the database and the workload, are modeled more abstractly. The model has been implemented using the DeNet simulation language [Livn88].

Parameter	Meaning
DatabaseSize	Size of database in pages
PageSize	Size of a page
NumClients	Number of client workstations
ThinkTime	Mean think time between client transactions
TransactionSize	Mean number of pages accessed per transaction
PerPageInst	Mean number of instructions per page on read (doubled on write)
HotBounds	Lower and upper page bounds of hot range
ColdBounds	Lower and upper page bounds of cold range
HotAccessProb	Probability of accessing a page in the hot range
HotWriteProb	Probability of writing to a page in the hot range
ColdWriteProb	Probability of writing to a page in the cold range

Table 1: Database and Workload Parameters

3.1. Database and Workload Models

Table 1 presents the parameters used to model the database and its workload. Since the cache consistency algorithms of interest here are page-oriented, the database and client transaction behavior are modeled at the page level. 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. For simplicity, each client workstation in the model is assumed to submit only one transaction at a time. Each workstation has its own set of values for the remaining workload parameters, which have been designed to allow a wide range of workloads to be modeled.

Each client workstation generates a stream of transactions, with adjacent transactions being separated 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, and this amount is doubled for pages that it writes; the actual per-page CPU requirement is drawn from an exponential distribution. To allow locality to be modeled, each client workstation has *hot* and *cold* regions of the database that are associated with it. *HotBounds* and *ColdBounds* specify the (possibly overlapping) ranges of pages in the client's hot and cold regions, respectively. When randomly generating page references for a new transaction, a page is drawn uniformly from among those in the client's hot region with probability *HotAccessProb*; otherwise the page is drawn from its cold region. *HotWriteProb* and *ColdWriteProb* specify the region-specific probabilities of writing a page that has been read.

3.2. Client-Server Execution Model

The simulator works by having each client workstation use the parameters just described to submit a sequence of client transactions. A transaction makes requests to begin and end its execution and to read and write pages of the database. The basic structures of the client and server components of the simulation model are indicated in Figures 2-3. The client component includes a *Source*, which generates the workload in the manner described in the previous section; a *Client Manager*, which executes transaction reference strings generated by the Source in accordance with the chosen cache consistency protocol; a *CC Manager*, which is in charge of concurrency control (i.e., locking); a *Buffer Manager*, which is responsible for managing the client buffer pool; and a *Resource Manager*, which models the other physical resources of the client workstation. The server component is organized similarly, except that its workload arrives via the network rather than from a local transaction Source. One portion of both the Client and Server Managers encapsulates the details of the cache consistency algorithms of interest.

Client transactions execute on the workstations that submit them. Details of their execution depend on the cache consistency algorithm in use, as covered in the architectural model and algorithm descriptions of Section 2. 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, algorithm-dependent steps are taken in reaction to the buffer miss. 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 decides whether or not to update it. In the event of an update, further processing is followed by algorithm-dependent update-handling actions. At commit

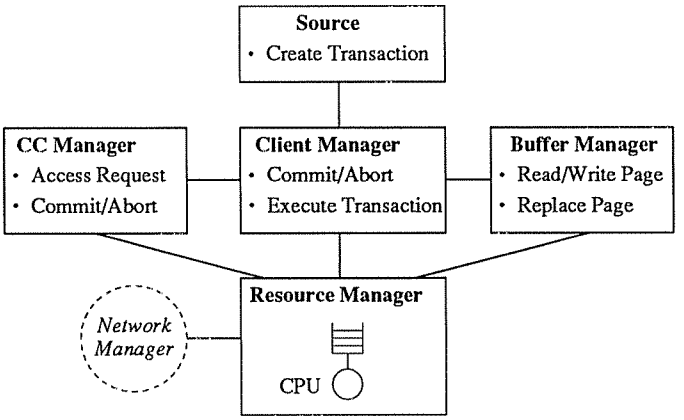

Figure 2: Client Component of Simulation Model

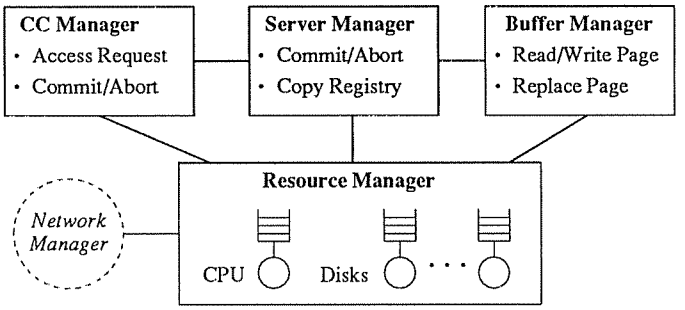

Figure 3: Server Component of Simulation Model

time, the Client Manager sends a commit request together with any updates to the server, which then takes the appropriate algorithm-dependent commit actions; an exception is that, in the O2PL algorithms, read-only transactions can commit without any commit-time server interaction. In the event of a transaction abort, which can occur due to a deadlock, the Client Manager arranges the abort, asks the Buffer Manager to purge any updated pages, and then resubmits the same transaction. Activity at the server is controlled by the Server Manager component of the simulator, which acts in response to the requests sent to it by the Client Managers.

In addition to the transaction-induced processing costs mentioned in Table 1, the simulation model includes the system-related costs given in Table 2. One such cost is the overhead to send or receive a message, which is

Parameter	Meaning
FixedMsgInst	Fixed number of instructions per message
PerByteMsgInst	Number of additional instructions per message byte
ControlMsgSize	Size in bytes of a control message
LockInst	Number of instructions per lock/unlock pair
RegisterCopyInst	Number of instructions to register/unregister a page copy
DeadlockInterval	Global deadlock detection interval

Table 2: Various System Overhead Parameters

modeled as *FixedMsgInst* instructions per message plus *PerByteMsgInst* instructions per message byte. The size of a control message (e.g., 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 overheads include *LockInst*, the cost involved in a lock/unlock pair on the client or server, and *RegisterCopyInst*, the cost to register and unregister (i.e., to track the existence of) a new cached page copy on the server or to look up the copies (if any) of a given page. The parameter *DeadlockInterval* indicates the frequency with which the server performs global deadlock detection in the O2PL algorithms, at which time it exchanges messages with all of the client workstations in order to obtain copies of their waits-for graphs.

3.3. Physical Resource Models

Table 3 lists the model parameters that specify the physical resources of the client workstations, the server, and the local area communications network. Included are the client and server MIPS ratings (*ClientCPU* and *ServerCPU*) and their respective buffer pool sizes (*ClientBufSize* and *ServerBufSize*). The service discipline of the client and server CPUs is first-come, first-served (FIFO) for message processing and processor-sharing for all other services; message processing preempts other CPU activity. The client and server buffer pools are both managed via an LRU replacement policy, and the server only writes dirty pages back to disk once they are actually selected for replacement. No preference is given to dirty pages. 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.

Turning to the physical I/O model, the parameter *ServerDisk* 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 of the disks. The service discipline for the disks is modeled as being FIFO.

Finally, 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*. A simple model is sufficient because our experiments assume a local area network, where the actual time on the wire for messages tends to be negligible and

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
NetworkBandwidth	Network bandwidth

Table 3: Resource-Related Parameters

the main cost issue is the CPU time for sending and receiving messages. This cost assumption has been found to provide reasonably accurate performance results despite its simplicity [Lazo86].

4. EXPERIMENTS AND RESULTS

In this section, we present performance results for the various client-server cache consistency algorithms and discuss their associated tradeoffs. We describe the experiments that were performed and the results that were obtained 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 average transaction response time, the client and server buffer hit ratios, client and server resource utilizations, the average number of messages required at the server to execute a transaction, and several others. The counts 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 almost 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 4-5 present the database and workload parameter settings used in the experiments reported here. Table 4 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 cache consistency algorithms scale, and the think time at the workstations is zero. The default per-page CPU processing time is 30,000 instructions.

Table 5 describes the range of workloads considered in this study. These workloads and their motivations will be described as their corresponding experiments are presented. Note that these workloads were designed to allow the exploration of the performance tradeoffs for client-server cache consistency algorithms; it is not the intent

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 4: Database and Workload Parameter Settings

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

Parameter	HOTCOLD	PRIVATE	FEED	UNIFORM	SHAREDHOT
TransactionSize	20 pages	16 pages	5 pages	20 pages	20 pages
HotBounds	p to p+49, p=50(n−1) + 1	p to p+24, p=25(n−1) + 1	1 to 50	—	1 to 50
ColdBounds	rest of DB	626 to 1,250	rest of DB	whole DB	rest of DB
HotAccessProb	0.8	0.5	0.8	—	0.8
ColdAccessProb	0.2	0.5	0.2	1.0	0.2
HotWriteProb	0.2	0.2	1.0/0.0	—	0.2
ColdWriteProb	0.2	0.0	0.0/0.0	0.2	0.2

Table 5: Workload Parameter Values for Client n

of this study to predict absolute performance for any particular system or application. Moreover, none of these workloads was derived from a real OODBMS application, as such applications are difficult to come by. 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 not to model the entire database, but rather to capture that portion which is of relatively current interest to the client workstations. Note that the most important factor here is the ratio of the transaction and client-server buffer pool sizes to the database size, not the absolute database size itself.

Table 6 shows the settings used in our experiments for the system overhead parameters, and Table 7 presents our resource-related parameter settings. In setting the various instructions counts and other parameters, we attempted to choose values that are reasonable approximations to what might be expected of systems today or in the near future. As indicated, the experiments that we will present were run with 5 MIPS client workstations and a 10 MIPS server. We also ran experiments with 15 MIPS clients and a 50 MIPS server; the absolute performance results were different, but the basic lessons were the same, so we do not present those results here. In addition, we conducted experiments with a range of client and server buffer pool sizes in order to understand how these important system parameters influence caching-related performance. Space will only permit the presentation of a representative subset of our full set of results.

Finally, before presenting the results for the various workloads, it will be helpful to briefly review our performance expectations. In most of our experiments, we will show how the various cache consistency algorithms perform as a function of the number of client workstations. Each client workstation adds both additional work (i.e.,

Parameter	Setting
FixedMsgInst	10,000 instructions
PerByteMsgInst	5,000 instructions per 4 kilobyte page
ControlMsgSize	256 bytes
LockInst	300 instructions
RegisterCopyInst	300 instructions
DeadlockInterval	1 second

Table 6: System Overhead Parameter Settings

Parameter	Setting
ClientCPU	5 (or 15) MIPS
ServerCPU	10 (or 50) MIPS
ClientBufSize	5%, 10%, 25%, or 50% of database size
ServerBufSize	10%, 25% or 50% of database size
ServerDisks	2 disks
MinDiskTime	10 millisecond
MaxDiskTime	30 milliseconds
NetworkBandwidth	32 megabits per second

Table 7: Resource-Related Parameter Settings

another transaction stream) and additional resources (i.e., another CPU and more memory) to the system. Ideally, then, we would like to see the system throughput increase linearly as the number of clients increases, with the average transaction response time remaining constant. In practice, of course, there are several possible impediments to linear system scaleup. These include (i) the formation of a bottleneck at the server CPU or disks, (ii) the formation of a data contention bottleneck, or (iii) an increase in the overall pathlength of transactions. Item (iii) can occur if the effect of adding a client is that additional messages or more disk accesses are required of all transactions; in this case, it is possible for thrashing to be observed, i.e., increasing the number of clients results in a decrease in overall system throughput.

4.2. Experiment 1: HOTCOLD Workload

The first performance results that we will examine are those for the HOTCOLD workload. In this workload, as indicated in Table 5, each client has its own 50-page hot region of the database to which 80% of its accesses are directed; the remaining accesses go elsewhere in the database. Client transactions each read an average of 20 pages, updating pages with a probability of 20%. Thus, this workload represents a situation where client transactions favor disjoint regions of the database, but where some read/write overlap exists in the data accessed by different clients (since the hot range of each client overlaps the cold range of all other clients). This situation is of interest because some OODBMS developers expect skewed client access distributions to be common in OODBMS applications [Catt90b, Wein90].

4.2.1. HOTCOLD Workload, Small Client Buffer Pool

Figure 4 shows the overall system throughput as a function of the number of clients for the HOTCOLD workload with a relatively large server buffer (*ServerBufSize* =50% of the database size) and small client buffers (*ClientBufSize* = 5% of the database size). Figure 5 shows the corresponding average transaction response time results. As shown, the three optimistic 2PL (O2PL) algorithms perform the best here, followed by caching 2PL (C2PL), with the basic 2PL scheme (B2PL) performing the worst among the algorithms studied. Initially, all three O2PL algorithms perform alike, as do the pair of server-locking algorithms (C2PL and B2PL). All provide near-linear scaleup in the range from 1 to 5 clients, as shown by their near-linear throughput increases and fairly flat response time curves in this range. Note that throughput also increases more rapidly for the O2PL algorithms, as the

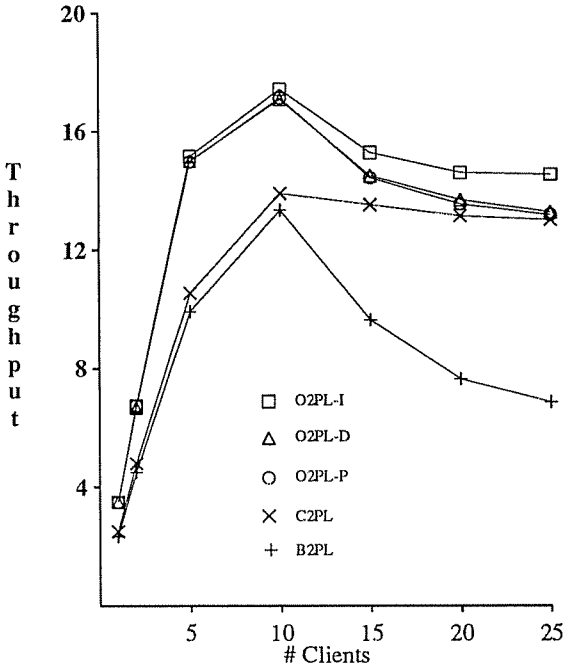

Figure 4: Throughput (Transaction/sec)
(HOTCOLD, Buffers: 50% server, 5% client)

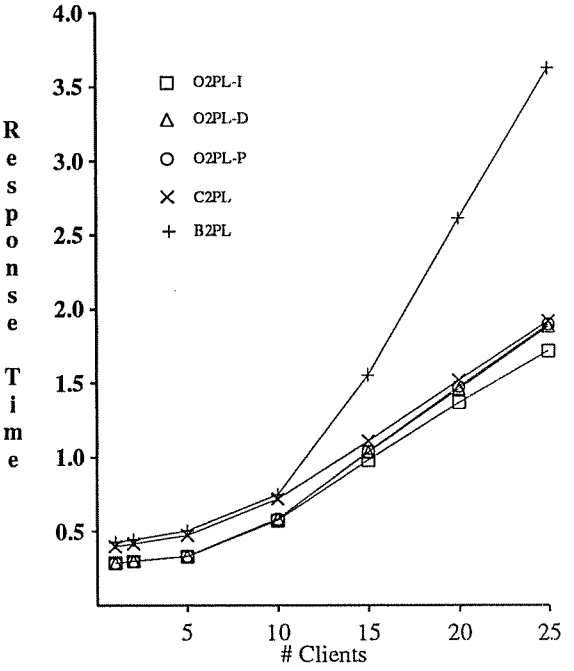

Figure 5: Response Time (sec)
(HOTCOLD, Buffers: 50% server, 5% client)

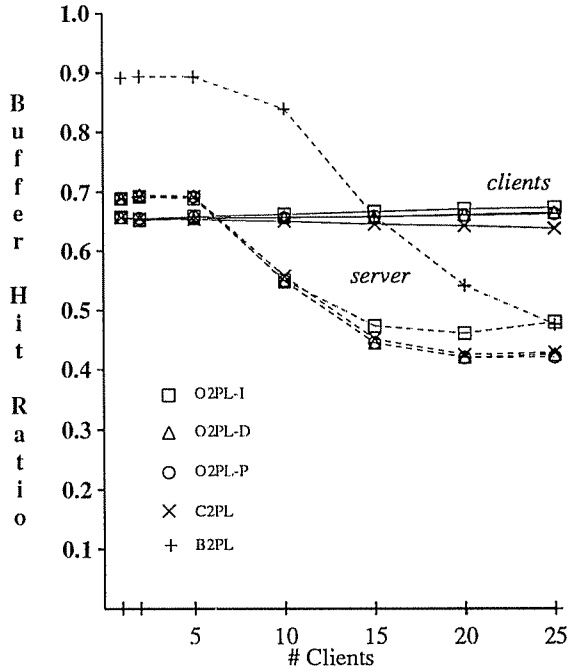

Figure 6: Client and Server Buffer Hit Rates
(HOTCOLD, Buffers: 50% server, 5% client)

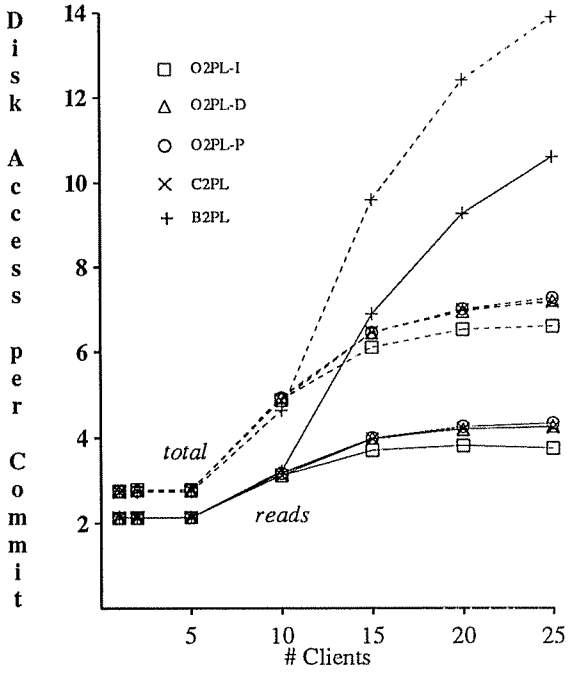

Figure 7: Disk Reads and Total I/O per Commit
(HOTCOLD, Buffers: 50% server, 5% client)

initial slope of the throughput curves is determined by the algorithms' 1-client throughput values (i.e., adding another client adds this much additional throughput when scaleup is linear).

The superior performance of the O2PL algorithms in the 1-5 client range is due to the considerable message savings that they offer. In this range, the O2PL algorithms require 18-19 messages to be processed per transaction on the server (counting both message sends and receives), on the average, as opposed to 52 messages per transaction for the server-locking algorithms. Under this workload, each client has its own 50-page hot range, covering 4% of the database, while client buffers are sized at 5% of the database. Thus, in the O2PL algorithms, most requests for hot pages can be satisfied without server interaction, whereas C2PL and B2PL have to request locks from the server for every new page accessed by a transaction. Indeed, we can see from Figure 6 (which gives the client and server hit rates) that 65% of the O2PL read requests can be processed without a server message since the O2PL algorithms send messages to the server only on cache misses (and at commit time for update transactions). The fact that this savings in messages is indeed the cause of the superior performance of the O2PL algorithms is confirmed by Figure 7, which shows that both the number of disk reads and and the total number of disk I/Os (including writes) per transaction are the same for all five algorithms in the 1-5 client range.

Looking out beyond 5 clients in Figures 4-5, we see that the O2PL algorithms retain their performance advantage, but that all of the algorithms lose their linear scaleup behavior. The throughput of each algorithm improves in a sublinear fashion in the 5-10 client range, and then throughput actually decreases for awhile before starting to level off again at 25 clients. This behavior is explained by Figures 6-7 and by the server CPU and disk utilizations. Since each client has a 4% hot range, but also accesses pages outside of this range, the server buffer pool size being 50% of the database causes the server hit rate to suffer at 10 clients and beyond; the server is no longer able to retain all of the clients' hot range pages in order to handle hot range cache misses without disk I/O. This is evident in the server hit rates of Figure 6 and the disk I/Os of Figure 7.[6] Eventually, the server hit rate decreases to 50% or slightly less, where 50% is what would be expected for a uniform (instead of skewed) reference stream. Moreover, once the server becomes I/O-bound, which occurs in the 10-15 client range, the increased server I/O pathlength for transactions (caused by this hit rate dropoff) is sufficient to induce the thrashing that is evident in Figure 4.

Focusing on the individual algorithms, we see that B2PL suffers the most from the dropoff in server hits as clients are added. As indicated in Figure 7, B2PL's dependence on server buffering leads to a significant I/O increase when the server can no longer retain the hot pages for all clients. C2PL does not suffer in this manner since it retains client buffer contents across transactions, giving it a client hit rate comparable to the O2PL algorithms. The slight performance advantage of C2PL prior to this region is due to the fact that its messages tend to be shorter than those of B2PL, as not all C2PL lock grant messages carry data pages. The O2PL algorithms thrash due to I/O activity beyond 10 clients, as discussed above, but they still perform the best since caching pays off and these algorithms have a significantly smaller CPU pathlength due to their message savings. O2PL-I, the invalidate-based

[6] Aside from B2PL, which does not cache data between transactions, all of the algorithms have a client buffer hit rate of about 65% or so despite their 80% hot range access frequency. The difference of 15% indicates the level of hot page misses due to the small client buffer size.

variant of O2PL, performs a bit better in Figures 4-5 due to a slightly higher buffer hit rate at the server (Figure 6) and a small I/O savings that results (Figure 7).

As compared to C2PL, the I/O savings provided by O2PL-I is due to the fact that O2PL-I provides a slightly larger effective client buffer pool — in C2PL, outdated pages are retained in remote client buffers, whereas they are invalidated immediately in O2PL-I. These outdated pages take up space in C2PL's LRU stack instead of becoming immediately reusable as they do in O2PL-I.[7] As compared to the other O2PL algorithms, O2PL-I's better server hit rate is due to the fact that hot page misses on the client are likely to lead to hot page misses on the server for O2PL-P and O2PL-D, as such hot pages have not been accessed very recently. In O2PL-I, however, a client's hot page misses are more likely to lead to hits in the server buffer pool, as cold page updates that invalidate its hot pages will likely lead to their being re-referenced rather quickly by the client where they are hot.

4.2.2. HOTCOLD Workload, Larger Client Buffer Pool

Figure 8 shows the overall throughput results for the HOTCOLD workload when the client buffer size is increased to 25% of the database. Comparing these results with Figure 4, it can be seen that the additional aggregate memory is strictly beneficial for all algorithms except B2PL and O2PL-P. Since B2PL does not retain its buffer contents between transactions, it performs exactly as before.[8] O2PL-P, the propagate-based member of the O2PL algorithm family, benefits from the increased memory for awhile, but as the system becomes large its performance actually suffers; moreover, it is outperformed significantly by the other two O2PL algorithms. Shape-wise, the curves in Figure 8 are roughly similar to those of Figure 4, and they can be similarly explained.

As noted above, the B2PL algorithm performs here just as it did with the 5% client buffer size. As before, it becomes I/O-bound due to the I/O activity caused by the inability of the server buffer pool to retain the hot regions of all clients beyond 10 clients or so. This is evident from Figure 9, which shows how the B2PL server hit rate declines at this point. In contrast, the C2PL algorithm clearly benefits from the added memory here. The reason is shown in Figure 9 — its client hit rate is significantly higher in this case since all of a client's hot region (4% of the database) fits in its buffer pool with room (21% of the database) to spare for some cold pages as well. In fact, C2PL turns out to be strictly CPU-bound at the server in this case due to frequent lock request messages.

Turning to the O2PL algorithms, O2PL-I performs the best, followed closely by O2PL-D. Both thrash somewhat at high loads for reasons similar to those in the 5% client buffer case. Specifically, this thrashing is caused by an increase in the average number of disk I/Os per transaction, as before. In this case, however, the additional I/Os turn out to be due to writes: When the server becomes unable to retain the hot pages for all clients in memory, it has to replace some of them, and when they are dirty, it must write them back to disk; since they are retained in the

[7] We instrumented the client buffer pool code to keep track of the average number of outdated pages in C2PL and the average free page count for both algorithms. The results showed that, in this experiment, C2PL's effective buffer size is about 5% smaller due to the presence of outdated pages in the 25-client case. We then ran an experiment in which C2PL's buffer size was increased by this amount, and its resulting I/O activity indeed matched that of O2PL-I.

[8] Recall that our workload model captures the first-time page references of transactions, not their page re-reference behavior, as caching within the scope of a single transaction is obviously beneficial and will aid all algorithms identically.

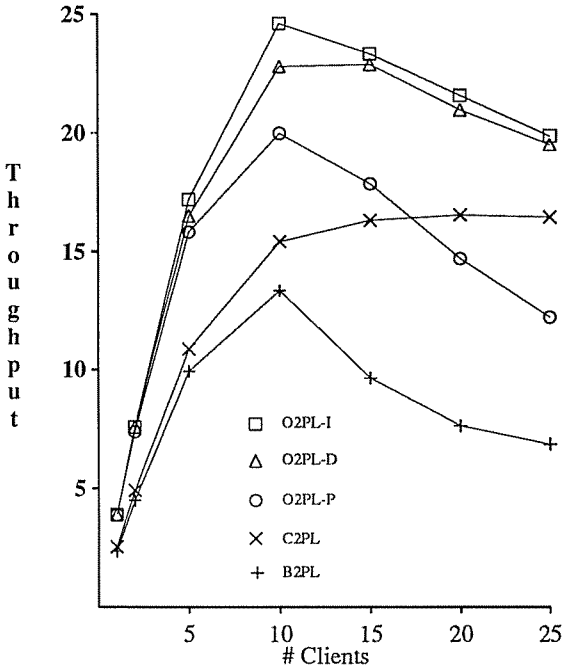

Figure 8: Throughput (Transaction/sec)
(HOTCOLD, Buffers: 50% server, 25% client)

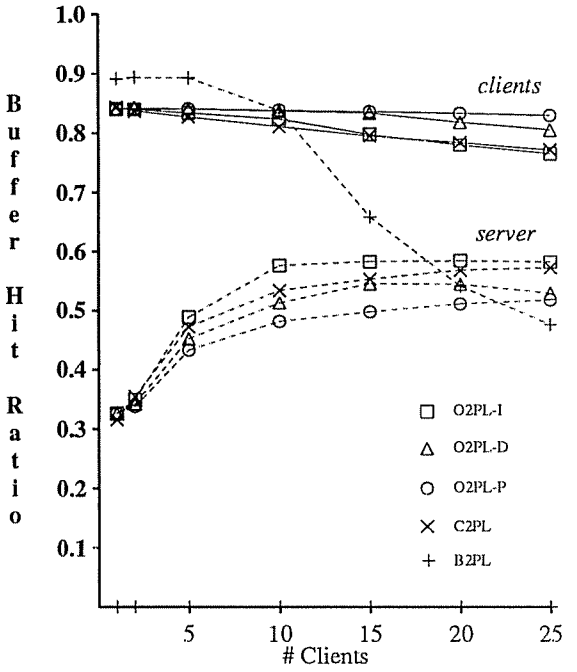

Figure 9: Client and Server Buffer Hit Rates
(HOTCOLD, Buffers: 50% server, 25% client)

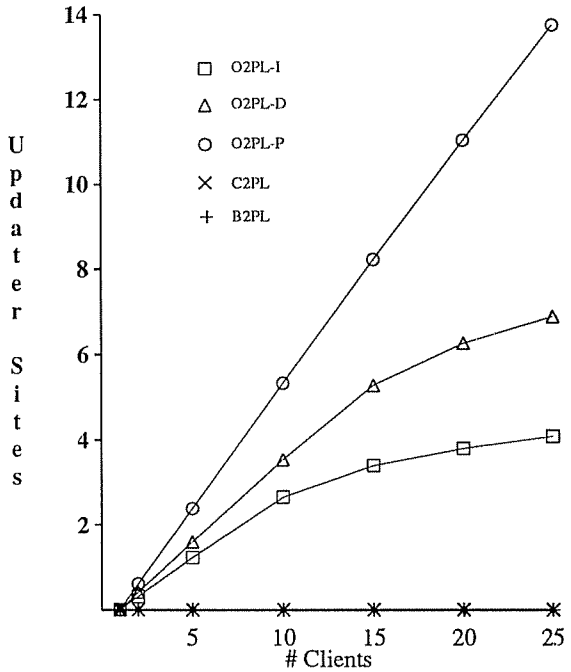

Figure 10: Avg. Number of Updaters per Trans
(HOTCOLD, Buffers: 50% server, 25% client)

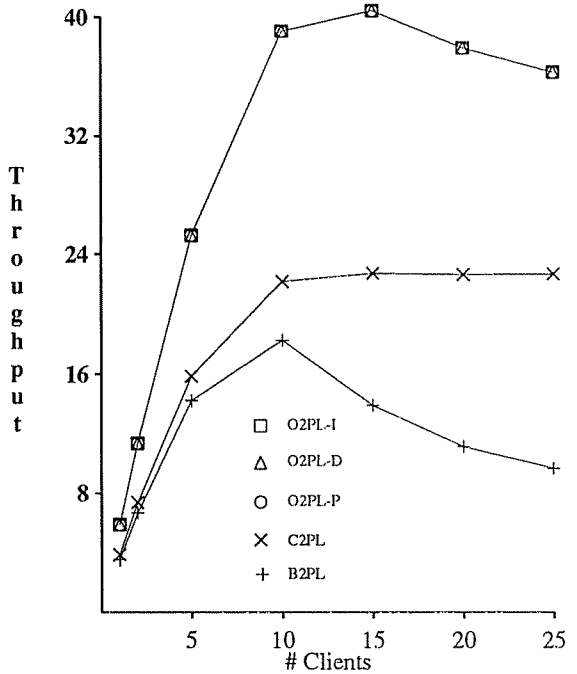

Figure 11: Throughput (Transaction/sec)
(PRIVATE, Buffers: 50% server, 25% client)

corresponding client buffer pool, removing them at the server does not significantly impact the number of disk reads. O2PL-P performs significantly worse here due to the fact that it will repeatedly propagate a hot page's updates to all clients that recently read it as a cold page (and therefore still have a copy in their buffer pool). This is evident in Figure 10, which shows the average number of remote clients that the server must contact in order to commit a transaction. For large system configurations, this leads O2PL-P to become CPU-bound, and it thrashes due to the increased server CPU pathlength caused by its propagation messages. This effect was not observed in the previous case because, with a small client buffer, LRU replacement kept the number of cached cold pages to a minimum. Since O2PL-I invalidates such counterproductive, remotely-cached copies of hot pages, this is less of a problem for O2PL-I (see Figure 10), and thus it manages to remain I/O-bound. O2PL-D propagates updates once to remote copies before recognizing that propagating changes to them is counterproductive, leading it to perform slightly worse than O2PL-I but much better than O2PL-P.

There is one additional interesting effect to be noted here, though it has little to do with the relative performance of the various cache consistency schemes. In Figure 9, the initial server hit rate for each of the schemes is initially lower than one might expect; with the server buffer size being 50% of the database size, the observant reader may have wondered why the hit rate is significantly lower in the 1-5 client range. This is because the client and server buffer contents are highly correlated in this range. For example, in the 1-client case, when the client misses on a cold region page, it has only a 25% chance of finding the page in the server buffer pool — because one half of the server's buffer pool is essentially a mirror-image of the client's own buffer pool. This effect disappears once the number of clients becomes sufficient to randomize the server buffer contents.

4.3. Experiment 2: PRIVATE Workload

The next results that we will discuss are those for the PRIVATE workload. In this workload (see Table 5), 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 portion of the database. Thus, there is no read/write sharing of data in this workload. This workload is intended to represent situations such as large, CAD-based engineering projects in which each engineer might work on disjoint portions of an overall design while read-sharing a standard library of components [Wein90]; it can also be viewed as a more extreme version of the previous workload.

Figure 11 presents the overall throughput results for this workload with *ServerBufSize* = 50% and *ClientBufSize* = 25%, as in the experiment just discussed. The general shapes of the curves are very similar to those of Figure 8 since the nature of the workloads is similar. B2PL and C2PL perform very much like in Figure 8, for virtually identical reasons. B2PL is again I/O-bound due to server buffer misses, while C2PL is CPU-bound due to its lock request messages. With this workload, all three O2PL algorithms perform identically; this is because it is never the case that an updated page is present in a remote buffer pool, so their different approaches to cache consistency maintenance are never exercised. The explanation for the shape of their curves is the same as for O2PL-I in Figure 8 of the preceding experiment. The main thing to notice here is that all three O2PL algorithms offer significant performance improvements over C2PL and B2PL. Caching is very beneficial for this workload, allowing the O2PL algorithms to execute transactions with many fewer messages, so O2PL's performance is server I/O-limited. In

fact, the average transaction required about 40 messages to be processed (i.e., sent or received) by the server for C2PL and B2PL, versus about 12 messages for the O2PL algorithms, independent of the system size.

4.4. Experiment 3: FEED Workload

We now turn our attention to a very different workload, an "information feed" workload where client #1 produces data that all other clients consume. This is intended to approximate an environment like stock trading, where a database of stock prices might be maintained by an information feed and then accessed heavily by other workstations. As indicated in the workload description of Table 5, 80% of client #1's accesses, including all of its updates, are directed to database pages 1-50; 80% of the accesses of the other clients, which are read-only, go to this region as well.

Figure 12 presents the throughput results for this workload, again for *ServerBufSize* = 50% and *ClientBufSize* = 25%. The throughput results for the writer (client #1) and readers (remaining clients) are separated in order to provide a clear picture of the impact of this heterogeneous workload. As shown in the figure, the O2PL algorithms outperform the two server-locking algorithms. Among the O2PL algorithms, O2PL-P provides the best performance, O2PL-D is next, and O2PL-I performs quite a bit worse than these two. Since the pages in the hot region of this workload are updated by client #1, which provides the information feed, and are used heavily by all of the other clients, these results are not surprising; propagation is clearly the right approach for such a workload, as expected. For all algorithms, the addition of clients (readers) leads to an increase in the overall reader throughput, as each one adds a new transaction stream. Adding clients also leads to a decrease in the writer throughput, as each new client is a source of additional server loading (and data contention) for transferring information from client #1 to the other clients.

The results of this experiment are explained by Figure 13, which shows the average number of messages per transaction processed by the server (averaged over both the writer and readers together). Due to the message intensity of the two server-locking algorithms, these algorithms become CPU-bound at the server with 10-15 clients (i.e., 1 writer and 9-14 readers). They are therefore unable to provide additional reader throughput beyond this point. O2PL-I suffers from a similar, but much less extreme, fate; each update by client #1 leads to the invalidation, and subsequent re-access at the server, of the updated data. O2PL-P performs the best, and actually achieves a high I/O utilization because it requires few enough messages that the server CPU does not become a bottleneck in the 1-25 client range. 2PL-D performs almost as well, but does enough invalidation to cause some loss of performance. At this point, it should be noted that, though 2PL-D has generally performed a bit worse than the better of the two static O2PL algorithms for each of the workloads examined here, it has also tended to perform quite a bit better than the lesser O2PL algorithm in cases involving significant performance differences.

4.5. Experiment 4: UNIFORM Workload

Up to now, we have explored various workloads for which it is intuitive that some form of caching should be beneficial. In this experiment, we turn our attention to the UNIFORM workload of Table 5. In this workload, each

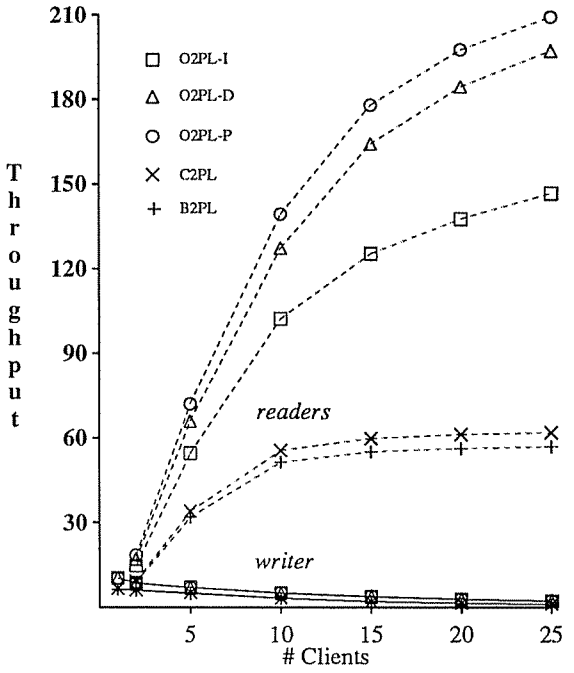

Figure 12: Throughput (Transaction/sec)
(FEED, Buffers: 50% server, 25% client)

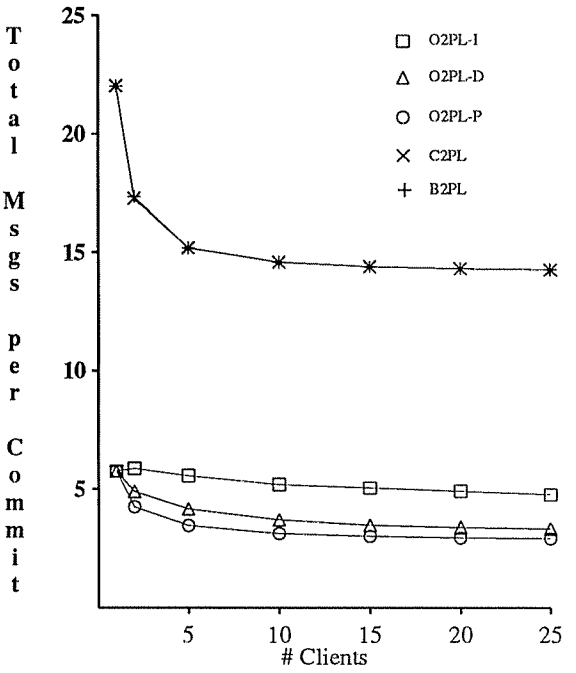

Figure 13: Messages Processed per Commit
(FEED, Buffers: 50% server, 25% client)

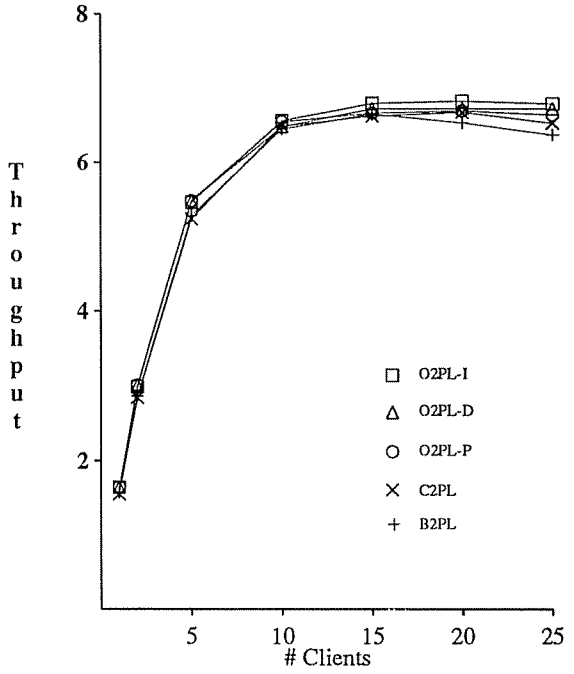

Figure 14: Throughput (Transaction/sec)
(UNIFORM, Buffers: 50% server, 5% client)

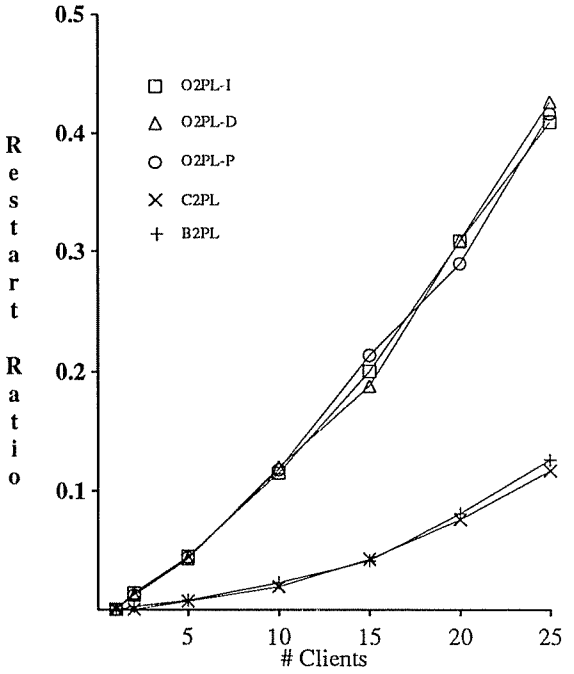

Figure 15: Aborts per Commit
(UNIFORM, Buffers: 50% server, 5% client)

client transaction reads an average of 20 pages, chosen uniformly from among all of the pages in the database, updating an average of 20% of these pages. In this case, then, client workstations display no locality of access, placing the utility of inter-transaction data caching in doubt.

4.5.1. UNIFORM Workload, Small Client Buffer Pool

Figure 14 presents the throughput results for the UNIFORM workload with *ServerBufSize* = 50% and *ClientBufSize* = 5%. All five of the cache consistency algorithms provide essentially the same level of performance in this case due to the small client buffer size and the lack of locality. When we examined the relevant underlying performance data, we found that all algorithms had the same level of server disk activity. We did observe server CPU utilization differences due to messages; the O2PL algorithms sent more messages per transaction than the two server-locking algorithms beyond the 10-client point. However, since the server disk utilization was approximately 90%, while the server CPU utilization was in the 40-60% range, these message differences had no impact on throughput. In fact, it turns out that system performance is actually limited somewhat by data contention here; the server disk is only 90% utilized when the throughput curves level off. The level of data contention can be seen in Figure 15, which shows the number of aborts per commit. Aborts become quite common for the O2PL algorithms as the number of clients becomes large. However, since all algorithms did the same amount of I/O, it is clear that the data needed by restarted transactions was always available in the client buffer (for data read) or the server buffer (for updates, since updated client pages are discarded on abort). This was further borne out by the client hit rate, which was seen to increase in proportion to the abort rate. As a result, restarts are essentially free, in terms of throughput, because client and server cache hits involve only CPU usage (and CPU was not a bottleneck).

4.5.2. UNIFORM Workload, Larger Client Buffer Pool

Figure 16 presents the throughput results for the UNIFORM workload with *ServerBufSize* = 50% and *ClientBufSize* = 25%. With this significantly larger client buffer size, performance differences now exist between the various algorithms. O2PL-I performs the best, followed by C2PL and O2PL-D (which have essentially the same performance), with O2PL-P and then B2PL providing the worst overall throughput. As usual, the performance of B2PL is unchanged from the small client buffer results, whereas each of the other algorithms benefits to some extent from the additional client buffer space (except for O2PL-P at 25 clients). The results in Figure 16 are due to a combination of factors that can be understood by examining the buffer hit rates (Figure 17), the number of I/Os per transaction (Figure 18), the server resource utilizations (Figure 19), and the data contention level (which is essentially the same here as with 5% client buffers, i.e., see Figure 15).

The difference between C2PL and B2PL indicates the performance increase due to the availability of client buffers; due to the uniform access pattern, and the fact that the server hit rate is already 50% (once the correlation effect discussed earlier is damped out), the additional client buffer space does not improve performance as dramatically here as in Experiments 1-3. Comparing C2PL with O2PL-I, which is the best performer, O2PL-I provides a modest performance improvement due to an effect that was also encountered in the HOTCOLD workload — O2PL-I provides a larger effective client buffer pool than C2PL due to the fact that invalidated pages are

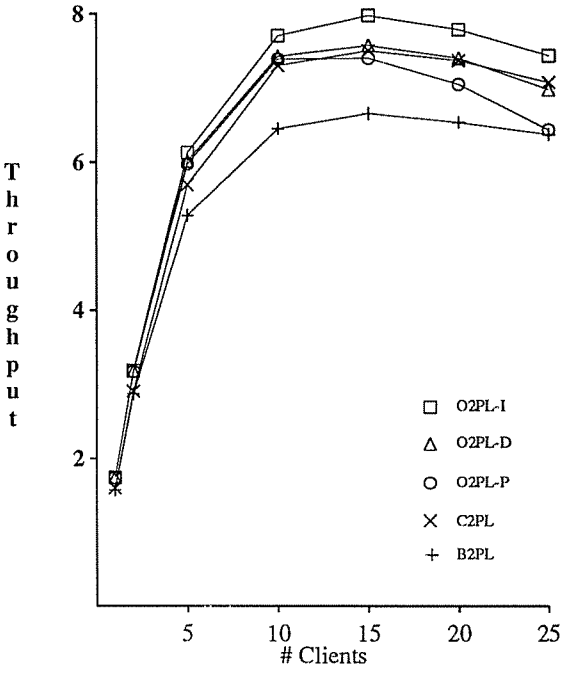

Figure 16: Throughput (Transaction/sec)
(UNIFORM, Buffers: 50% server, 25% client)

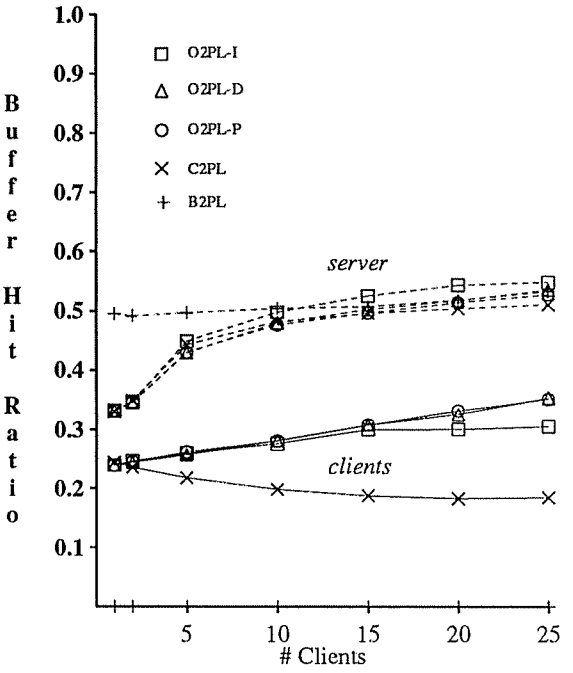

Figure 17: Client and Server Buffer Hit Rates
(UNIFORM, Buffers: 50% server, 25% client)

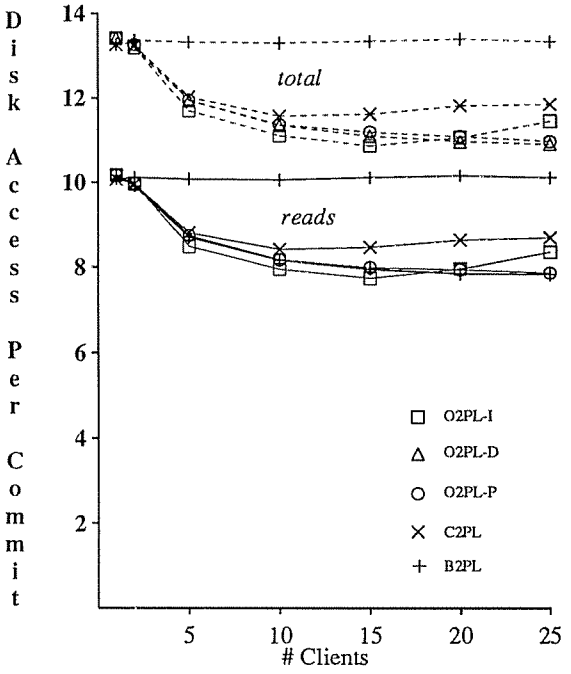

Figure 18: Disk Reads and Total I/O per Commit
(UNIFORM, Buffers: 50% server, 25% client)

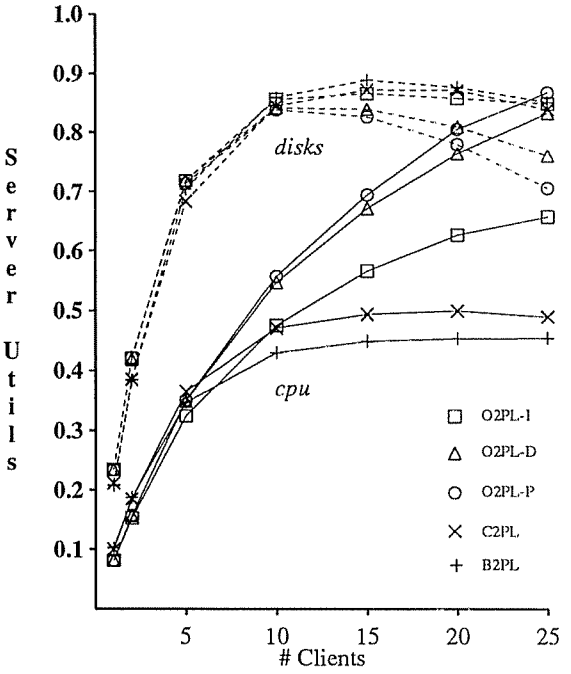

Figure 19: Server Resource Utilizations
(UNIFORM, Buffers: 50% server, 25% client)

immediately freed, rather than taking up buffer pool space as they do in of C2PL. This is evident in Figure 17, though some of the difference there is due to restart-induced buffer hits, and in Figure 18, where O2PL-I is seen to require fewer I/Os per transaction than C2PL. The other two O2PL algorithms are unable to benefit similarly from the lack of outdated pages because of their high CPU overheads. As is evident in Figure 19, they utilize the server CPU much more heavily than O2PL-I, propagating updates (repeatedly) to other clients instead of simply invalidating the updated pages. Moreover, propagation is simply not beneficial here.[9] O2PL-P suffers slightly more due to wasted propagations because O2PL-D only propagates once to a page before invalidating it. Finally, all of the caching algorithms can be seen to thrash somewhat in Figure 16; this is due largely to data contention (i.e., transaction restarts).

To explore the impact of updates (and associated data contention) here, Figure 20 shows how throughput is affected as the write probability for transactions is varied in the 10-client case. As the write probability goes to zero, performance converges for all of the algorithms except B2PL. This is because all of the other algorithms benefit from caching, and their effective buffer pool size and propagation-related differences disappear in the absence of updates. Conversely, when the write probability becomes very large, the optimistic locking approach of the O2PL algorithms causes their performance to suffer; we will see more of this effect in the next experiment.

4.6. Experiment 5: SHAREDHOT Workload

In this experiment, we examined the performance of the various cache consistency algorithms for the SHAREDHOT workload, a workload that generates an extremely high level of data contention. As indicated in Table 5, the transactions in this workload have characteristics similar to those in the HOTCOLD workload, reading an average of 20 pages, updating an average of 20% of them, and directing 80% of their accesses to a 50-page hot region of the database. In this case, however, all clients have the *same* hot region. It is not expected that OODBMS workloads will involve this level of data contention, but we include one such experiment for completeness.

Figure 21 shows the throughput results for this workload in the case where *ServerBufSize* = 50% and *ClientBufSize* = 10%. Due to the high level of data contention here, the two server-locking algorithms actually outperform the three O2PL algorithms by a very significant margin. An examination of the server resource utilizations revealed that that the system is highly "data-bound" with this workload, with server disk utilizations of only 10% and server CPU utilizations in the 10-60% range (depending on the algorithm). Here, the optimism of the O2PL approach, with its more permissive handling of conflict detection and its corresponding tendency to resolve conflicting data accesses later, more than counteracts the benefits of caching. Moreover, as shown by the small difference between C2PL (which performs the best with two or more clients) and B2PL, caching does not improve performance much in this case anyway; this is because the server buffer pool is able to be very effective, retaining 50% of the database (including the hot region, which is common to all clients).

[9] We instrumented our simulator to keep track of the fraction of all propagated pages that are actually used by the client subsequent to the propagation, i.e., before the page is replaced or overwritten by another propagation. Our measurements indicate that, here, only 10-15% of the pages propagated by O2PL-P actually proved useful in this sense.

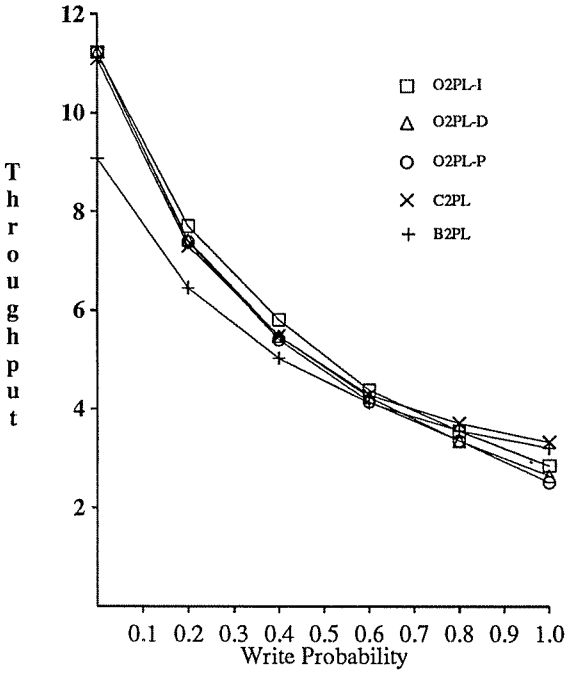

Figure 20: Throughput (TPS) vs. Write Prob.
(UNIFORM, 10 Clients, Buffers: 50% srv, 25% cli)

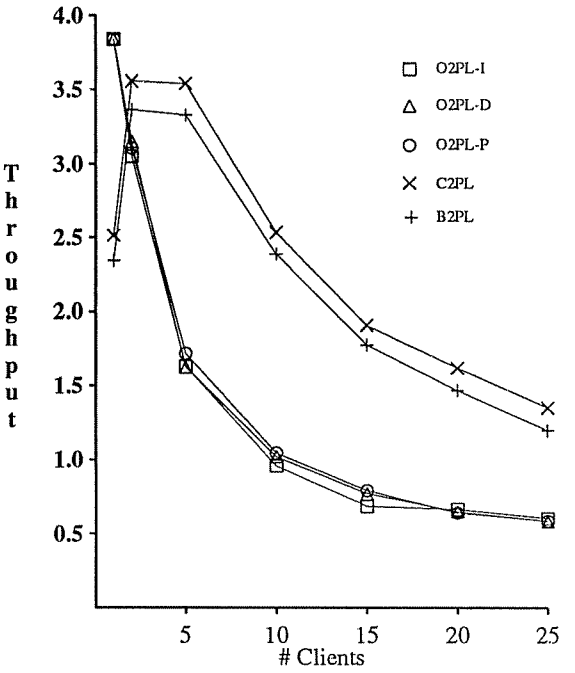

Figure 21: Throughput (Transactions/sec)
(SHAREDHOT, Buffers: 50% server, 10% client)

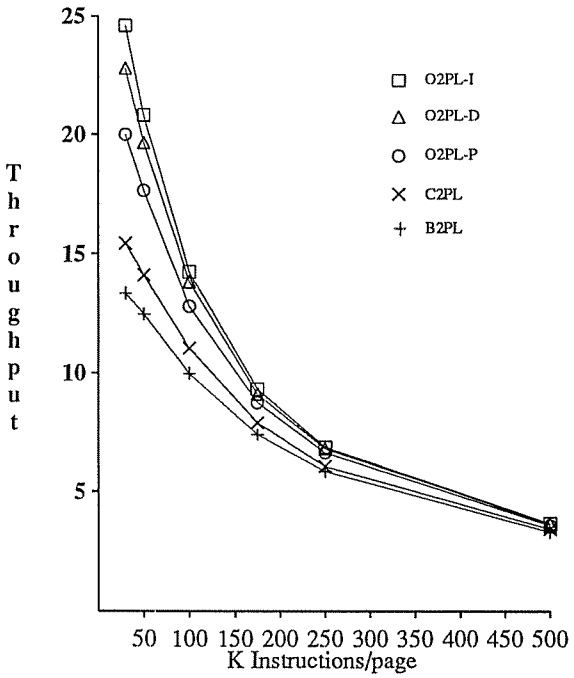

Figure 22: Throughput (TPS) vs. PerPageInst
(HOTCOLD, 10 Clients, Buffers: 50% srv, 25% cli)

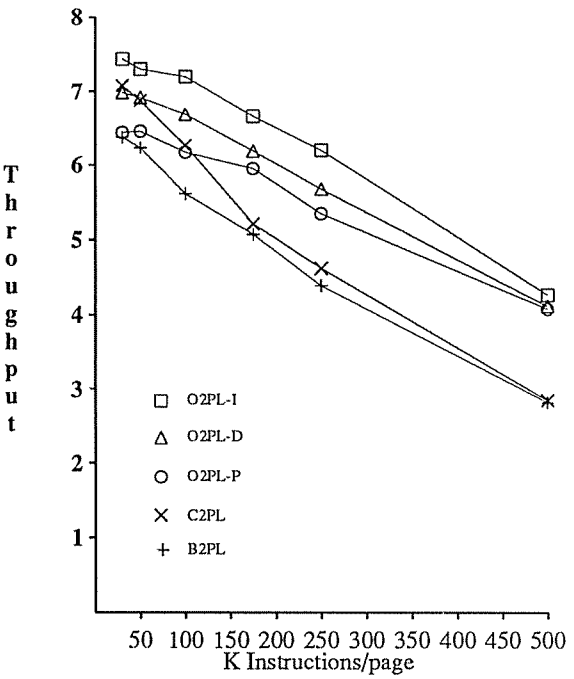

Figure 23: Throughput (TPS) vs. PerPageInst
(UNIFORM, 25 Clients, Buffers: 50% srv, 25% cli)

4.7. Experiment 6: Impact of Application Pathlength

An important aspect of the workload that we have not yet varied is the amount of client CPU processing involved in transaction execution. Up to now, the application pathlength has been fixed at a *PerPageInst* setting of 30,000 instructions. Figure 22 shows how the performance results for the HOTCOLD workload of Experiment 1 vary as a function of this parameter in the 10-client case for the larger client buffer size setting (i.e., with *ServerBufSize* = 50% and *ClientBufSize* = 25%). As shown, the algorithm differences that were observed earlier decrease as the application pathlength increases, eventually reaching a point where there is simply no difference between the algorithms. This is to be expected — it is obvious that once the per-page client CPU time becomes totally dominant, and cache-related overheads are reduced to noise, the choice of a caching policy will have no impact on overall performance.

Figure 23 shows how the UNIFORM workload results vary with the application pathlength in the case where *ServerBufSize* = 50% and *ClientBufSize* = 25% in the 25-client case. In contrast to what we saw above, significant performance differences remain for the UNIFORM workload even when the per-page client CPU pathlength is 500,000 instructions. The throughputs of the three O2PL algorithms converge, and likewise for the two server-locking algorithms, but the O2PL algorithms perform significantly better here. This is strictly a concurrency control effect, and is also to be expected. As seen in Experiment 4, there is a significant amount of data contention in the UNIFORM workload. Moreover, when the client portion of the CPU pathlength becomes dominant, the system moves into an "infinite resource" region of operation — that is, the critical resource for transactions is the client CPU, a resource that transactions do not share. It is well-known that optimistic concurrency control algorithms outperform traditional locking algorithms in the presence of data contention when resources are plentiful [Fran85, Agra87], and this is precisely the difference between the algorithms here.

4.8. Comparison With Related Work

As described in the introduction, the studies most closely related to this one are shared-disk performance studies [Bhid88, Yu87, Dan90], the client-server data caching study that was described in [Wilk90], and the transaction-oriented distributed memory hierarchy work of [Bell90]. As explained earlier, client-server DBMS architectures tend to be larger than shared-disk DBMS configurations, and the existence of a central server changes the nature of the performance problem somewhat (e.g., consider the various client-server buffer interactions, and server CPU limitations, that have been important here). Also, we have not encountered propagation-based algorithms in the shared-disk literature. Despite these differences, our results on the good performance of O2PL-I as compared with server-locking agree qualitatively with related findings for shared-disk algorithms, such as the success of a semi-optimistic locking scheme in [Yu87]. As for the related client-server caching study in [Wilk90], our study employed a much more detailed buffering model and also covered a wider range of workloads and parameter settings than the work reported there; the importance of both differences should be clear from the results. However, again there is agreement to be found at a qualitative level. For example, [Wilk90] also found that the performance of a propagation-based cache consistency algorithm, roughly similar to our O2PL-P algorithm, performed

badly when propagated data was infrequently referenced. We have gone further in this regard, proposing and investigating the O2PL-D algorithm as a solution to the performance sensitivity of the propagation approach. Lastly, in comparison to the work of [Bell90], where consistency control algorithms based on both invalidation and (periodic) propagation were also considered, we have observed a very different set of resource-related performance tradeoffs due to major architectural differences between the systems studied.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have examined the performance tradeoffs associated with caching data on client workstations in a client-server DBMS architecture. We began by presenting five lock-based cache consistency algorithms that arose by viewing the cache consistency problem as a variant of the problem of replicated data management in a distributed DBMS. Two of the algorithms that were presented always set locks at the server, while the other three are more optimistic in their approach to locking. Among the latter group, one uses invalidation to maintain consistency in the face of updates, another bases its approach on propagation of updated values, and the third algorithm is a dynamic scheme that attempts to combine both approaches. We used a detailed simulation model to study the performance of these algorithms over a wide range of workloads and system configurations.

The results of our performance study indicate that caching can improve performance significantly for some workloads, although we also studied workloads where the performance improvement due to caching was marginal or even nonexistent. We found that the invalidation-based optimistic algorithm (O2PL-I) performed quite well for many workloads, while its propagation-based counterpart (O2PL-P) performed better in a workload designed to capture an "information feed" application. O2PL-P was also found to be rather workload-sensitive, however, and had problems when scaled to large system configurations with some of the workloads that were investigated. The dynamic algorithm (O2PL-D) managed to track the performance of the better O2PL algorithm for each workload studied, performing close to (but never quite as well as) the better of the static O2PL algorithms. Lastly, the caching server-locking algorithm (C2PL) was generally outperformed by the better of the O2PL algorithms, except in the case of a workload that had an extremely high level of data contention. In addition to these algorithm-oriented results, our study has indicated the importance of using a detailed model of buffering when investigating client-server cache consistency tradeoffs.

We plan to continue this work in several ways. First, we would like to find a more satisfactory dynamic algorithm than O2PL-D, one that can more closely match or even exceed the performance of the better static O2PL algorithm over a wide range of workloads. Second, we plan to validate our simulation model against the EXODUS storage manager once its recovery implementation is complete (so that transactions can be aborted and restarted), at least for the C2PL and B2PL algorithms. Finally, we plan to turn our attention to other interesting and related issues, including data caching on client disks and the evaluation of alternative approaches to client-server crash recovery.

ACKNOWLEDGEMENTS

The authors would like to thank David DeWitt for many lively discussions regarding workload modeling and other aspects of this work. We would also like to thank Rick Cattell of Sun Microsystems and Dan Weinreb of Object Design for their input regarding OODB workloads.

REFERENCES

[Agra87] Agrawal, R., Carey, M., and Livny, M., "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Trans. on Database Sys.* 12, 4, Dec. 1987.

[Arch86] Archibald, J., and Baer, J.-L., "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. on Comp. Sys.* 4, 4, Nov. 1986.

[Bell90] Bellew, M., Hsu, M., and Tam, V.-O., "Update Propagation in Distributed Memory Hierarchy," *Proc. 6th Int'l. Conf. on Data Eng.*, Los Angeles, CA, Feb. 1990.

[Bern87] Bernstein, P., Hadzilacos, V., and Goodman, N., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[Bhid88] Bhide, A., and Stonebraker, M., "An Analysis of Three Transaction Processing Architectures," *Proc. 14th VLDB Conf.*, Los Angeles, CA, Aug. 1988.

[Care84] Carey, M., and Stonebraker, M., "The Performance of Concurrency Control Algorithms for Database Management Systems," *Proc. 10th VLDB Conf.*, Singapore, Aug. 1984.

[Care89a] Carey, M., *et al*, "Storage Management for Objects in EXODUS," in *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. Lochovsky, eds., Addison-Wesley, 1989.

[Care89b] Carey, M., and Livny, M., "Conflict Detection Tradeoffs for Replicated Data," submitted to *ACM Trans. on Database Sys.*. (Available as Comp. Sci. Tech. Report No. 826, University of Wisconsin, March 1989.)

[Catt90a] Cattell, R., and Skeen, J., *Engineering Database Benchmark*, Tech. Rep., Database Eng. Group, Sun Microsystems, April 1990.

[Catt90b] Cattell, R., personal communication, Nov. 1990.

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

[Deux90] Deux, O., *et al*, "The Story of O_2," *IEEE Trans. on Knowledge and Data Eng.* 2, 1, March 1990.

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

[Fran85] Franaszek, P., and Robinson, J., "Limitations of Concurrency in Transaction Processing," *ACM Trans. on Database Sys.* 10, 1, March 1985.

[Gray89] Gray, C., and Cheriton, D., "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency," in *Proc. 12th ACM Symp. on Op. Sys. Prin.*, Litchfield Park, AZ, Dec. 1989.

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

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

[Kim90] Kim, W., *et al*, "The Architecture of the ORION Next-Generation Database System," *IEEE Trans. on Knowledge and Data Eng.* 2, 1, March 1990.

[Lazo86] Lazowska, E., *et al*, "File Access Performance of Diskless Workstations," *ACM Trans. on Comp. Sys.* 4, 3, Aug. 1986.

[Livn88] Livny, M., *DeNet User's Guide*, Version 1.0, Comp. Sci. Dept., Univ. of Wisconsin, Madison, 1988.

[Nels88] Nelson, M., Welch, B., and Ousterhout, J., "Caching in the Sprite Network File System," *ACM Trans. on Comp. Sys.* 6, 1, Feb. 1988.

[ODI90] Object Design, Inc., *ObjectStore Technical Overview*, Aug. 1990.

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

[Shek90] Shekita, E., and Zwilling, M., "Cricket: A Mapped Persistent Object Store," *Proc. 4th Int'l. Workshop on Pers. Obj. Sys.*, Martha's Vineyard, MA, Sept. 1990.

[Ston79] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," *IEEE Trans. on Softw. Eng.* SE-5, 3, May 1979.

[Ston90a] Stonebraker, M., *et al*, "Third-Generation Data Base System Manifesto," *SIGMOD Record* 19, 3, Sept. 1990.

[Ston90b] Stonebraker, M., "Architecture of Future Database Systems, *Data Eng.* 13, 4, Dec. 1990.

[Wein90] Weinreb, D., personal communication, Nov. 1990.

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

[Yu87] Yu, P., *et al*, "Analysis of Affinity Based Routing in Multi-System Data Sharing, *Perf. Evaluation* 7, 2, June 1987.

[Zwil90] Zwilling, M., *Using the EXODUS Storage Manager V2.0 (alpha)*, EXODUS Project Document, Dec. 1990.