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-Madison

[**]IBM Almaden Research Center

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 that 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 the 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. Unlike relational database systems, client-server interactions in an OODBMS 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]. Prototypes based on page-level (or multi-page block) interactions include ObServer [Horn87] and the EXODUS storage manager [Care89a]. ObjectStore [ODI90] is a commercial OODBMS based on page-level interactions.

In such architectures, it is possible to cache data in the local memories of client workstations for later reuse. Caching can reduce the need for client-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 example of a workload where caching can be highly beneficial is the Sun Engineering Database Benchmark [Catt90a].

[†]This work was done while the author was with the Computer Sciences Department, University of Wisconsin-Madison.

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.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1991 ACM 0-89791-425-2/91/0005/0357...$1.50

Caching is not a performance panacea, however. To incorporate caching, the DBMS must include a cache consistency protocol. Such a protocol may be complex, it may entail substantial processing overhead, and its impact on system performance may be workload-dependent. 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 many client workstations. Another potential pitfall, which depends on the concurrency control scheme used, is the late discovery of data conflicts. Thus, the potential consequences of adding caching to a client-server DBMS deserve investigation.

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. We then describe a detailed simulation model that was developed to study their performance over a wide range of workloads and system resource configurations. 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, this choice was motivated by a desire to understand performance tradeoffs in our own page-based, client-server storage managers [Care89a, Shek90].

The performance of transaction-oriented cache consistency algorithms has been examined in several related contexts. The one previous client-server data caching study is the HP Laboratories work [Wilk90]. Our work differs 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 work by Bhide [Bhid88] and by a DBMS performance group at IBM Yorktown [Dan90]. 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, so the server is a natural center of activity that can be used to assist in cache management. Also, shared-disk DBMS configurations tend to involve relatively few machines. Second, our range of cache consistency algorithms goes beyond those found in the shared-disk literature, including algorithms that propagate changes rather than invalidating other cached copies. Another closely related effort is the Harvard work on transaction-oriented distributed memory hierarchies [Bell90]. This 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. Also, our work is loosely related to studies of multiprocessor cache coherency algorithms (e.g., [Arch86]) and of caching in distributed file systems (e.g., [Howa88, Nels88]).

The remainder of the paper is as follows: Section 2 describes our architectural assumptions and cache consistency algorithms. Section 3 describes our simulation model. Section 4 describes experiments and results. Section 5 presents our conclusions.

2. CACHE CONSISTENCY ALGORITHMS

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 is connected to the server, include the disks on which the database is stored and a (mirrored) disk for the log. The software of the DBMS has components that reside on the server and on the clients, so applications running on client workstations can view the DBMS essentially as a local service. The DBMS has buffer pool space available on both the server and the clients, and it is free to manage this space as it sees fit in order to optimize its overall performance. In the page server architecture, pages are the unit of client-server data transfer and also serve as the unit of data caching and cache consistency.[1]

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 workstation. We will not concern ourselves with the details of how the client application and database processes interact within a client workstation. All we require 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. Also, 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 state of database client processes outlives client transactions, they are free to cache data both within and across transaction boundaries as long as system-wide cache consistency is maintained.

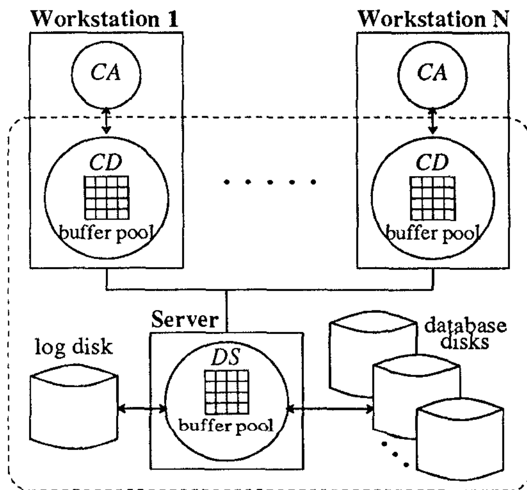

Figure 1: Cache Architecture of a Client-Server DBMS

[1] In this study, we focus our attention on the caching of data; index pages must be handled via a separate mechanism in order to support the necessary level of concurrent activity.

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 obtained by recognizing that cached data are just like replicated data in a distributed DBMS, though they reside in main memory rather than on disk.[2] Thus, all of the well-known results on replica management [Bern87] can be applied.

Given this observation, we now present five candidate algorithms for maintaining client-server cache consistency. 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 based on an optimistic variant of two-phase locking [Care89b], in which updates to remote copies of replicated data are deferred until end-of-transaction.

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 larger than the available client buffer space. Also, 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 the same as that of the existing lock. Read lock requests and page access requests are combined, and the server returns 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. Deadlocks are handled through centralized deadlock detection on the server when lock waits occur. Deadlock resolution aborts the transaction with the most recent initial startup time among those involved in the deadlock.

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

[2] If the workstations have local disks, it is possible to cache data on secondary storage. Local disk caching is beyond the scope of this paper, however, and will be addressed in future work.

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 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. Clients inform the server when they no longer have a copy of a given page. For example, when a client elects to replace a clean cached page, it discards that page and then notifies the server 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. When a client requests a page from the server, a read lock is acquired and held on the server only long enough 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 not permitted to migrate back to the server until the update transaction enters its commit phase. At that time, the client 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 clients that contain cached copies of any of the updated pages. These clients request update-copy locks on the updated data on behalf of the committing transaction.[3] Once all 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. When a committing transaction requests an update-copy lock for an item that is locked with a read lock by another transaction, 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. Other deadlocks, including distributed deadlocks, are dealt with by having the client and server database processes check for local deadlocks, and by having the server periodically check for distributed deadlocks a la [Ston79].

[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.

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 other 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 updated pages. Once all updated pages have been invalidated at a client, the client sends a prepared-to-commit message back to the server, releases its update-copy locks, and then drops 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 this 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 all cached copies of the 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.

2.5. Performance Tradeoffs

We have presented five algorithms for client-server cache consistency, all based on viewing cached pages as replicated data. We now 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,

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

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 only at commit time (to handle updated pages); therefore, transactions that 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

3.1. Database and Workload Models

We now describe our client-server DBMS simulation model, which has been constructed using the DeNet simulation language [Livn88]. 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 transaciton 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 requirement is drawn from an exponential distribution. To model locality, each client workstation has *hot* and *cold* regions of the database associated with it. *HotBounds* and *ColdBounds* specify the (possibly overlapping) ranges of pages in the client's hot and cold regions, espectively. Pages to be referenced are drawn uniformly from 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.

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 xactions
TransactionSize	Mean no. of pages accessed per xaction
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. Client-Server Execution Model

In the simulator, the client component includes a *Source*, which generates the workload; 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 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, 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 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.

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 cpu cost to send or receive a message, which is 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 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. The *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

FixedMsgInst	Fixed no. of instructions per message
PerByteMsgInst	No. of addl. instructions per msg. byte
ControlMsgSize	Size in bytes of a control message
LockInst	No. of instructions per lock/unlock pair
RegisterCopyInst	No. of instructions to register/unregister a page copy
DeadlockInterval	Global deadlock detection interval
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 2: System and Resource Parameters

clients in order to obtain copies of their waits-for graphs.

3.3. Physical Resource Models

The model parameters that specify the physical resources of the system are also listed in Table 2. 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 writes dirty pages back to disk only when they are actually 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 *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 disks. The service discipline for the disks is FIFO.[5]

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

4. EXPERIMENTS AND RESULTS

4.1. Metrics and Parameter Settings

The primary performance metric employed in this study is system throughput (i.e., transaction completion rate). Additional metrics are used in the analysis of the experimental results. Metrics 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. 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 used in all of the experiments (except where otherwise noted). The number of client workstations is varied from 1 to 25 in order to study how the cache consistency algorithms scale, and the think time at the workstations is zero. The default per-page CPU processing time is 30,000 instructions. The database size is 1,250 pages, with a page size of 4 kilobytes. We used a relatively small database in order to make simulations involving fractionally large buffer pools feasible in terms of simulation time; moreover, our intent is to capture that portion which is of 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.

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

Table 4 describes the workloads considered in this study. These workloads and the motivation for them will be described as the experiments are presented. These workloads were designed to allow the exploration of the performance tradeoffs for client-server cache consistency algorithms. None of these workloads was derived from a real OODBMS application, as such applications are difficult to come by.

Table 5 shows the settings used for the system overhead and resource-related parameters. The experimental results presented

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

Table 4: Workload Parameter Values for Client n

FixedMsgInst	10,000 instructions
PerByteMsgInst	5,000 instructions per 4 kilobyte page
ControlMsgSize	256 bytes
LockInst	300 instructions
RegisterCopyInst	300 instructions
DeadlockInterval	1 second
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 5: System and Resource Parameter Settings

[5] Logging is not explicitly modeled as it was not expected to impact the relative performance of the algorithms based on experience with an earlier study [Care89b].

here were obtained with 5 MIPS client CPUs and a 10 MIPS server CPU. 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. We also 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. Due to space limiations, only a representative subset of our results is presented here.

In most of our experiments, we show how the cache consistency algorithms scale with the number of client workstations. Each client workstation adds both additional work and additional resources to the system. Ideally, we would like the system throughput to scale linearly with the number of clients. In practice, 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 (i.e., if additional messages or more disk accesses are required of all transactions).

4.2. Experiment 1: HOTCOLD Workload

In the HOTCOLD workload, 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. Transactions each read an average of 20 pages, updating pages with a probability of 20%. This models a situation where different clients favor disjoint regions of the database, but where some read/write overlap exists (since the hot range of each client overlaps the cold range of all other clients). Skewed client access distributions are important to study since some OODBMS developers expect them to be common in OODBMS applications [Catt90b, Wein90].

4.2.1. HOTCOLD - Small Client Buffer Pool

Figure 2 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 3 shows the average transaction response times. The three optimistic 2PL (O2PL) algorithms perform the best, followed by caching 2PL (C2PL). The basic 2PL scheme (B2PL) performs the worst. 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.

The superior performance of the O2PL algorithms in the 1-5 client range is due to their considerable message savings. In this range, the O2PL algorithms require an average of 18-19 messages to be processed per transaction on the server (counting both message sends and receives), as opposed to 52 messages per transaction for the server-locking algorithms. 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 4 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 5, 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 2-3, 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 4-5 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 pages in order to handle hot cache misses without disk I/O. This is evident in the server hit rates of Figure 4 and the disk I/Os of Figure 5. 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 2.

B2PL suffers the most from the dropoff in server hits as clients are added. As shown in Figure 5, 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. 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 much smaller CPU pathlength due to their message savings. O2PL-I, the invalidate-based variant of O2PL, performs a bit better in Figures 2-3 due to a slightly higher buffer hit rate at the server (Figure 4) and a small I/O savings that results (Figure 5).

As compared to C2PL, the I/O savings provided by O2PL-I is due to the fact that O2PL-I has a slightly larger effective client buffer pool — in C2PL, outdated pages are retained in 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.[6] As compared to the other O2PL algorithms, O2PL-I's better server hit rate is due to the fact that in O2PL-P and O2PL-D, hot pages are removed from the client buffer only when they become the least recently used page in the buffer. In O2PL-I however, hot pages are also removed from the client buffer by the invalidation process. Such hot pages are likely to be in the server's buffer when re-referenced by the client, since the update that caused the invalidation places an up-to-date copy of the page at the server.

4.2.2. HOTCOLD - Larger Client Buffer Pool

Figure 6 shows the overall throughput results for the HOTCOLD workload when the client buffer size is increased to 25% of the database. In comparison with Figure 2, it can be seen that the additional aggregate memory is strictly beneficial for all algorithms except B2PL and O2PL-P. Since B2PL does not

[6] We instrumented the client buffer pool code and found 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.

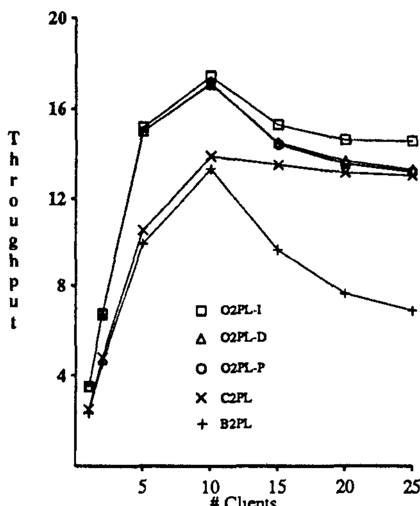

Figure 2: Throughput (Transaction/sec)
(HOTCOLD, Buffers: 50% server, 5% client)

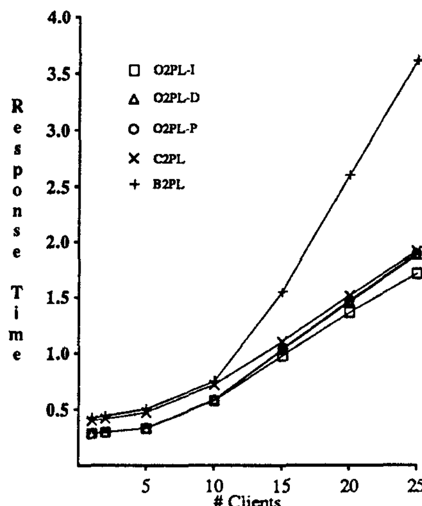

Figure 3: Response Time (sec)
(HOTCOLD, Buffers: 50% server, 5% client)

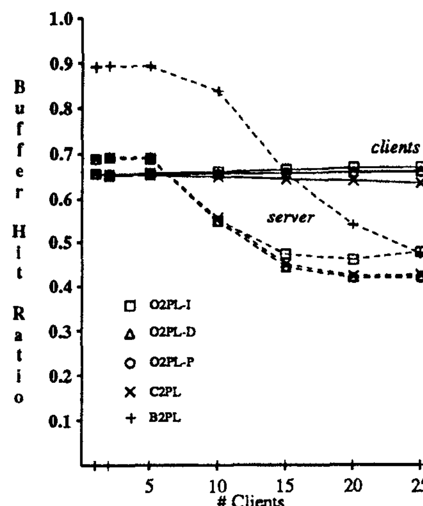

Figure 4: Client and Server Buffer Hit Rates
(HOTCOLD, Buffers: 50% server, 5% client)

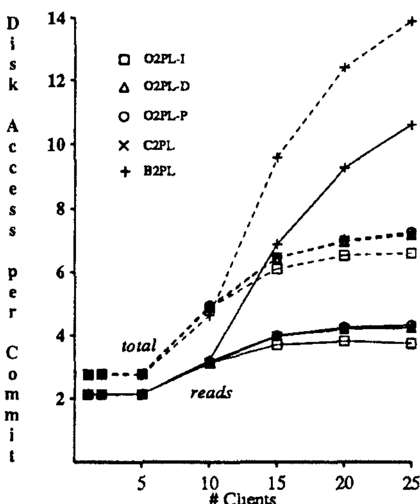

Figure 5: Disk Reads and Total I/O per Commit
(HOTCOLD, Buffers: 50% server, 5% client)

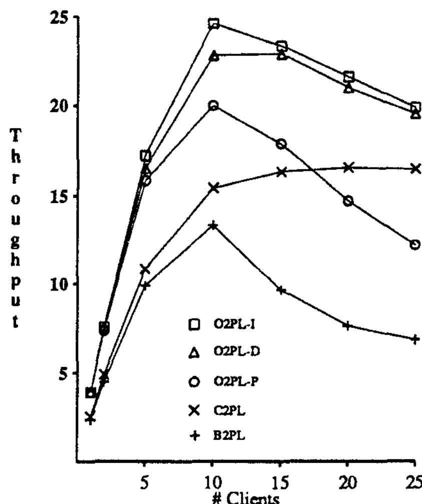

Figure 6: Throughput (Transaction/sec)
(HOTCOLD, Buffers: 50% server, 25% client)

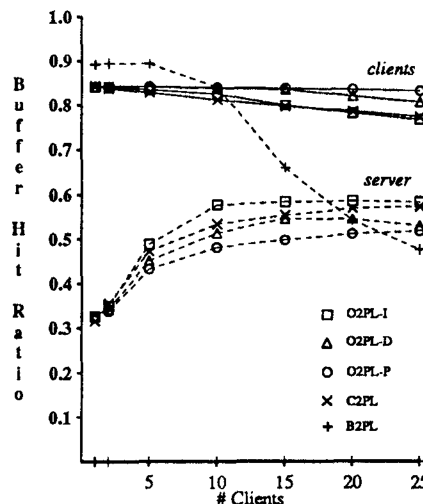

Figure 7: Client and Server Buffer Hit Rates
(HOTCOLD, Buffers: 50% server, 25% client)

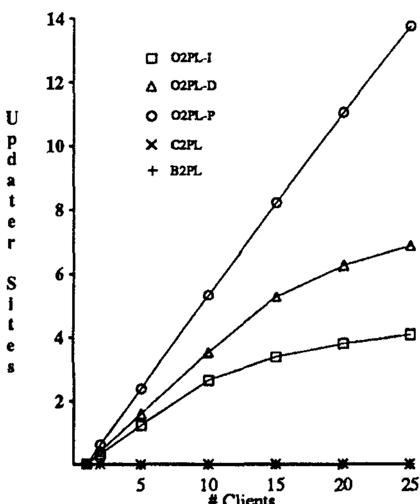

Figure 8: Avg. Number of Updaters per Trans.
(HOTCOLD, Buffers: 50% server, 25% client)

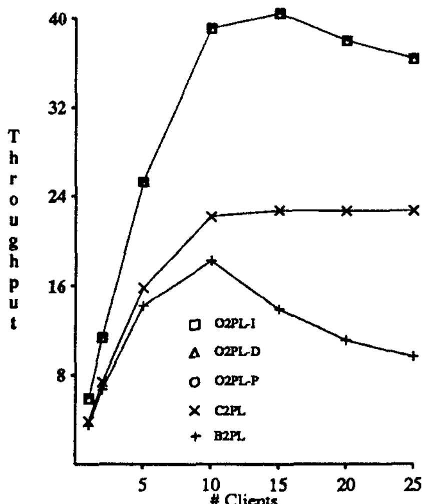

Figure 9: Throughput (Transaction/sec)
(PRIVATE, Buffers: 50% server, 25% client)

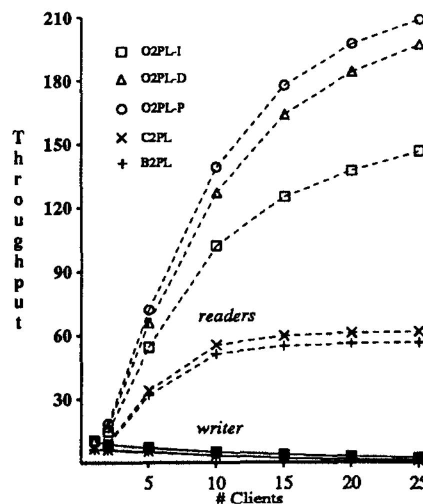

Figure 10: Throughput (Transaction/sec)
(FEED, Buffers: 50% server, 25% client)

retain its buffer contents between transactions, it performs exactly as before (recall that we model only the first-time page references of transactions, as caching within the scope of a single transaction will aid all algorithms identically). O2PL-P, the propagate-based O2PL algorithm, benefits from the increased memory for awhile, but as the system grows its performance suffers; also, it is outperformed significantly by the other two O2PL algorithms. The curves in Figure 6 are roughly similar to those of Figure 2, and they can be similarly explained.

The B2PL algorithm performs here just as it did with the 5% client buffer size. In contrast, the C2PL algorithm clearly benefits from the added memory. The reason is shown in Figure 7. The client hit rate of C2PL is significantly higher in this case since the client's hot region (4% of the database) fits in its buffer pool with room (21% of the database) for cold pages as well. C2PL is strictly CPU-bound at the server in this case due to frequent lock request messages.

Of the O2PL algorithms, O2PL-I performs the best, followed closely by O2PL-D. Both thrash at high loads due to an increase in the average number of disk I/Os per transaction. Unlike the previous case, however, the additional I/Os are due to writes: When the server is 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 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 accessed it as a cold page (and therefore still have a copy in their buffer pool). This is evident in Figure 8, 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 8), 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.

Figure 7 shows an interesting effect (that only slightly affects the relative performance of the cache consistency schemes). The server hit rate for each of the schemes is initially (in the 1-5 client range) lower than one might expect; 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

In the PRIVATE workload, 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 region. 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 works on disjoint portions of an overall design while read-sharing a standard library of components [Wein90].

Figure 9 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 6 since the nature of the workloads is similar. B2PL and C2PL perform very much like in Figure 6, 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. The explanation for the shape of their curves is the same as for O2PL-I in Figure 6 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.

4.4. Experiment 3: FEED Workload

We now turn our attention to 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. 80% of client #1's accesses (reads and 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 10 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. 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 and are used heavily by all the readers, these results are not surprising; propagation is clearly the right approach for such a workload. 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.

The results of this experiment are explained by the message requirements of the algorithms. 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 because it requires few messages; the server CPU does not become a bottleneck in the 1-25 client range. O2PL-D performs almost as well, but does enough invalidation to cause a loss of performance. At this point, it should be noted that, though O2PL-D has generally performed a bit worse than the better of the two static O2PL algorithms for each of the workloads, it has also tended to perform significantly better than the lesser O2PL algorithm in cases involving significant performance differences.

4.5. Experiment 4: UNIFORM Workload

Up to now, we have explored skewed workloads for which it is intuitive that some form of caching should be beneficial. In this experiment, we examine the UNIFORM workload in which

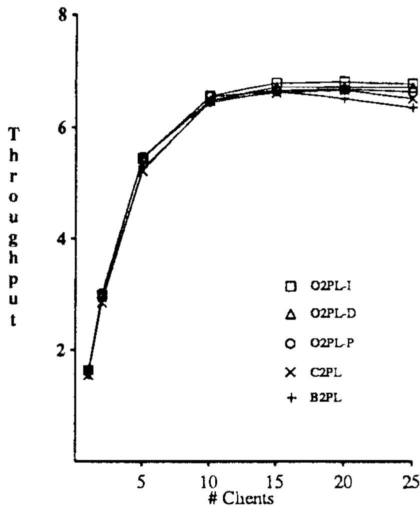

Figure 11: Throughput (Transaction/sec) (UNIFORM, Buffers: 50% server, 5% client)

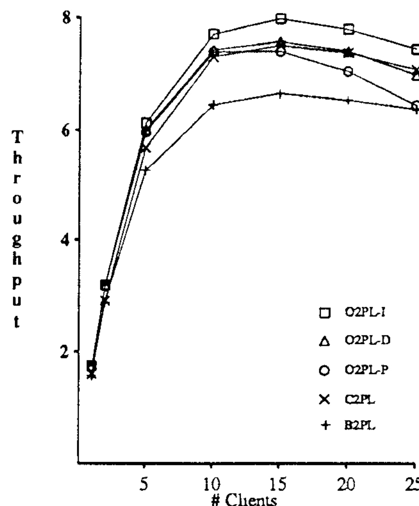

Figure 12: Throughput (Transaction/sec) (UNIFORM, Buffers: 50% server, 25% client)

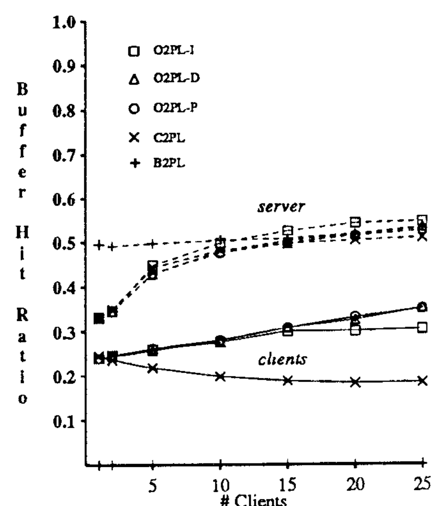

Figure 13: Client and Server Buffer Hit Rates (UNIFORM, Buffers: 50% server, 25% client)

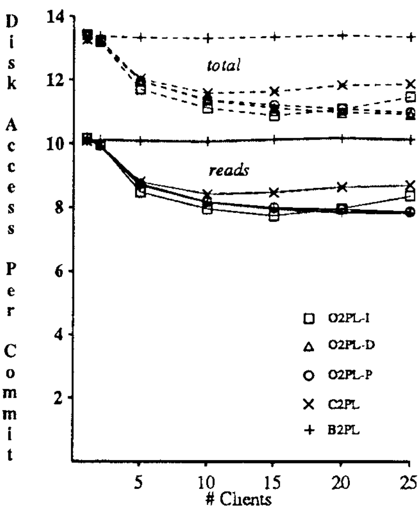

Figure 14: Disk Reads and Total I/O per Commit (UNIFORM, Buffers: 50% server, 25% client)

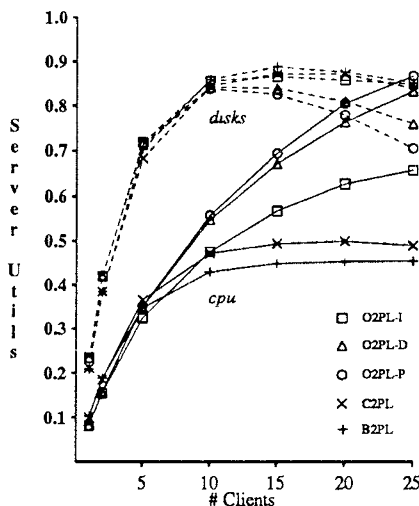

Figure 15: Server Resource Utilizations (UNIFORM, Buffers: 50% server, 25% client)

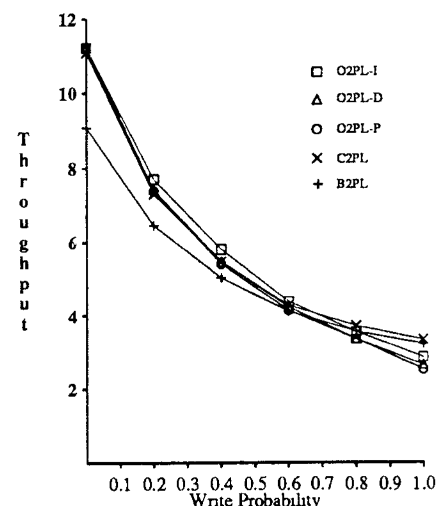

Figure 16: Throuput (TPS) vs. Write Prob. (UNIFORM, 10 Clients, Buffers: 50% srv, 25% cli)

each 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 the pages read.

Figure 11 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. Figure 12 shows the throughput for the UNIFORM workload with *ServerBufSize* = 50% and *ClientBufSize* = 25%. With this much larger client buffer size, performance differences now exist between the various algorithms. O2PL-I performs the best, followed by C2PL and O2PL-D 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 12 are due to a combination of factors that can be understood by examining the buffer hit rates (Figure 13), the number of I/Os per transaction (Figure 14), the server resource utilizations (Figure 15), and the data contention level.

The difference between C2PL and B2PL indicates the performance increase due to the availability of client buffers; because of 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. O2PL-I provides a modest performance improvement over C2PL due to an effect that was also encountered in the HOTCOLD workload; that is, O2PL-I provides a larger effective client buffer pool than C2PL because invalidated pages are immediately freed, rather than taking up buffer pool space as they do in C2PL. This is evident in Figure 13, though some of the difference there is due to restart-induced buffer hits, and in Figure 14, 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. They utilize the server CPU much more heavily than O2PL-I, propagating updates to other clients instead of simply invalidating the updated pages. Moreover, propagation is simply not beneficial here.[7] O2PL-P suffers slightly more due to

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

propagations because O2PL-D only propagates once to a page before invalidating it. Finally, all of the caching algorithms can be seen to thrash in Figure 12; this is due largely to data contention (i.e., transaction restarts).

Figure 16 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 algorithms except B2PL. This is because all 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.

4.6. Additional Experiments

The results from two other exeriments are worth noting here (see [Care90] for details). One experiment studied an extremely high contention workload in which the C2PL and B2PL algorithms out performed the O2PL algorithms due to O2PL's late resolution of conflicts. This result could be anticipated from Figure 16. The other experment examined the impact of longer application pathlength (i.e, more CPU processing per page) and found that with low data contention, the difference among the algorithms was negligible (as expected). However, in a higher contention case, the O2PL algorithms outperformed the others.

5. CONCLUSIONS

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.

ACKNOWLEDGEMENTS

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

REFERENCES

[Arch86] Archibald, J., and Baer, J.-L., "Cache Coherence Protocols: Evaluation Using a Multilprocessor Simulation Model," *ACM TOCS* 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.*, 1988.

[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", to appear *ACM TODS.*

[Care90] Carey, M.J., *et al*,"Data Caching Tradeoffs in Client-Server DBMS Architectures",*Comp. Sci. TR #994, University of Wisconsin-Madison*, January 1991.

[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.*, Aug. 1990.

[Deux90] Deux, O., *et al*, "The Story of O_2," *IEEE TKDE* Mar. 1990.

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

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

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

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

[Lazo86] Lazowska, E., *et al*, "File Access Performance of Diskless Workstations," *ACM TOCS* 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 TOCS* 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*,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 Multilple 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.*, Aug. 1990.

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