Client-Server Caching Revisited

Michael J. Franklin
Michael J. Carey

Technical Report #1089

May 1992

Client-Server Caching Revisited

Michael J. Franklin
Michael J. Carey

Computer Sciences Technical Report #1089
May 1992

Computer Sciences Department
University of Wisconsin - Madison

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

Client-Server Caching Revisited

Michael J. Franklin and *Michael J. Carey*

Computer Sciences Department
University of Wisconsin - Madison
Madison, WI 53706
{mjf,carey}@cs.wisc.edu

ABSTRACT

Most commercial and experimental Object-Oriented Database Systems (OODBMS) are implemented using a client-server software architecture. An effective technique for improving the performance of a client-server database system is to allow client workstations to cache data and/or locks across transaction boundaries. This paper extends an earlier study on the performance of caching in client-server database systems [Care91a] in three ways. The first is a re-examination of heuristics for algorithms that decide dynamically between propagating changes or invalidating remote copies of data pages in order to maintain cache consistency. The second extension is a performance study of a class of caching algorithms known as "callback locking" algorithms. These algorithms are of interest because they provide an alternative to the optimistic techniques used in the earlier study and because they have recently begun to find use in commercial systems. The third way that this paper extends the previous study is to examine the performance of alternative caching algorithms in light of current trends in processor and network speeds and to more closely examine their robustness in the presence of data contention.

1. INTRODUCTION

Object-Oriented Database Systems (OODBMS) are typically designed for use in networks of high-performance workstations and servers. As a result, most OODBMS products and research prototypes have adopted a variant of a *client-server* software architecture known as *data-shipping*. In a data-shipping system, client processes send requests for specific data items to servers, and servers respond with the requested items (and possibly others). Data-shipping systems are particularly well-suited for use in workstation-based environments since they allow a DBMS to perform much of its work on client workstations, thus exploiting the workstations' processing and memory resources and reducing dependence on shared server machines. Data-shipping systems can be categorized as *page servers*, in which clients and servers interact using physical units of data such as pages or segments, and *object servers*, which send logical units of data such as objects or tuples between clients and servers. Many recent systems have adopted the page server approach due its relative simplicity and potential performance advantages compared to an object server approach. These systems include: ObServer [Horn87], O2 [Deux90], Client-Server EXODUS [Exod91, Fran92c], and ObjectStore [Lamb91]. For concreteness, this paper concentrates on cache management algorithms for page server architectures; however, many of the results of this study are also applicable to object server systems.

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

1.1. Client-Server Caching

Caching data and locks at client workstations is an important technique for improving the performance of page server database systems. Data and locks can be cached both within a single transaction (*intra-transaction caching*) and across transaction boundaries (*inter-transaction caching*). Intra-transaction caching is easily implemented; it requires only that a client can manage its own buffer pool and can keep track of which locks it has obtained. All of the algorithms investigated in this study perform intra-transaction caching of both data and locks. On the other hand, inter-transaction caching of data requires additional mechanisms to ensure the consistency of cached data pages, while inter-transaction caching of locks requires full-function lock management on clients and protocols for distributed lock management. This work concentrates on examining the performance tradeoffs that arise when using inter-transaction caching of both data and locks. For the remainder of the paper, we use the term "caching" to refer to inter-transaction caching.

Despite the additional complexity, caching of data and locks has been shown to have the potential to provide substantial performance benefits [Wilk90, Care91a, Wang91]. These benefits include: 1) reducing reliance on the server, thus offloading a potential bottleneck and reducing communication, 2) allowing better utilization of the substantial CPU and memory resources that are available on clients, and 3) increasing the scalability of the system in terms of the impact of adding additional client workstations. However, while the potential benefits of client caching are large, the question of the overhead that is incurred in order to guarantee consistency must be addressed carefully. This overhead can manifest itself in several forms: 1) increased communication, 2) increased load on the server CPU, 3) additional latency (especially at the end of transactions), and 4) extra load placed on clients that have data cached. Also, some algorithms have the property of postponing the discovery of data conflicts, which can result in increased transaction abort rates.

1.2. The Previous Study

In an earlier paper we investigated the performance implications of caching data pages and/or locks at the client workstations of a page server database system [Care91a]. In that paper, five locking-based algorithms were compared: an algorithm that performed no caching, an algorithm that cached data (but not locks) at clients, and three variants of an *Optimistic Two-Phase Locking* (O2PL) algorithm that allowed caching of data pages and an optimistic form of lock caching. In the O2PL schemes, locks are obtained only locally at clients during transaction execution and then, prior to commit, locks on copies of updated pages are obtained at remote sites in order to ensure consistency. This form of lock caching is optimistic in the sense that a local lock at one site does not guarantee the absence of conflicting local locks at other sites. The three O2PL-based cache consistency algorithms examined in [Care91a] differed in the actions that they performed at the remote sites once locks were obtained. One variant, called O2PL-Invalidate (O2PL-I), always removed the copies from the remote site. The second variant, O2PL-Propagate (O2PL-P), always sent new copies of the updated pages to the remote sites, thereby preserving replication. The third variant, O2PL-Dynamic (O2PL-D), was an adaptive algorithm that used a simple heuristic to choose between invalidation and propagation on a copy by copy basis.

In the previous study, a number of conclusions were reached about the relative merits of these alternative caching algorithms. These include the following:

(1) All of the caching algorithms were shown to have much higher performance than the non-caching algorithm in most of the workloads examined.

(2) The optimistic caching of locks in addition to the caching of data was found to provide additional performance benefits except in a workload with extremely high data contention and in some cases where the O2PL-P algorithm performed excessive update propagation.

(3) The invalidation-based version of O2PL (O2PL-I) typically had the highest throughput over the range of client populations and resource parameters studied.

(4) The propagate version of O2PL (O2PL-P) was found to have significant performance advantages for certain types of data sharing, but also displayed a high degree of volatility, especially with regard to client buffer pool sizes. In many workloads, O2PL-P was seen to suffer from wasted work due to propagation of changes to remote copies that were never actually reused.

(5) The dynamic O2PL algorithm (O2PL-D) was seen to approach, but not quite equal, the performance of the better of the static O2PL algorithms over a wide range of workloads.

1.3. Extensions to the Previous Study

In this paper we extend the work of [Care91a] in three ways:

1. *A Better Adaptive Heuristic*: The first extension relates to the heuristics used to choose between invalidation and propagation in the O2PL-Dynamic algorithm. As stated above, the heuristic used in [Care91a] usually approached the performance level of the better of the two static O2PL algorithms for a given workload, but it never exceeded or matched that level. Thus, the first issue addressed here is to find a better heuristic for the adaptive algorithm.

2. *Other Lock Caching Algorithms*: The second extension is an analysis of an alternative approach to maintaining cache consistency known as as *callback locking* [Howa88, Lamb91, Wang91]. As with O2PL-based techniques, callback locking allows clients to cache both data pages and locks, but unlike O2PL, lock caching is not optimistic — that is, sites are not allowed to simultaneously cache conflicting locks. Two algorithms that use callback locking are studied: one that allows caching of both read and write locks, and one that only allows caching of read locks. These algorithms are of interest because they provide an alternative (non-optimistic) implementation of lock caching and because at least one commercial OODBMS has adopted a callback algorithm [Lamb91].

3. *System Parameter and Workload Changes*: This study uses updated system resource parameter values and a new workload. The system parameters have been updated to reflect the changes in hardware technology that have occurred in the two years since the earlier work was initiated. These changes have the potential to shift the performance bottlenecks in the system and thus, may alter the comparative advantages of the caching algorithms. For example, the speed of the processors in workstations and servers has increased dramatically while disk speeds have not; network technology has been improving, but most installed networks have yet to catch up. In this study, we examine the effects of these changes

by increasing the speeds of the client and server CPUs compared to the earlier study, and also by examining the effect of two different network bandwidths. These network speeds roughly correspond to current Ethernet and near-term FDDI networks. Where appropriate, we discuss the differences between the results that were obtained with the previous parameter settings and those obtained in this study with the new settings.

We also use an additional workload to help examine the performance of the caching algorithms in the presence of high data contention. The previous study showed that the lock caching performed by the O2PL algorithms was actually detrimental to performance under a very high data contention workload, but the sensitivity of the algorithms to data contention was not fully explored. In this study, we examine the effects of data contention in more detail and also investigate the effects of data contention on the callback locking algorithms.

1.4. Overview of the Paper

The remainder of the paper is structured as follows: Section 2 describes the three types of cache consistency algorithms that are investigated in this study. Section 3 briefly describes the simulation model and workloads used to perform the study. Section 4 presents the experiments and results. Section 5 discusses related work. Finally, Section 6 presents our conclusions.

2. OVERVIEW OF CACHING ALGORITHMS

The algorithms addressed by the study fall into three basic families: Server-based 2PL, Optimistic 2PL (O2PL), and Callback Locking. These algorithms are described in the following sections.

2.1. Server-based Two Phase Locking

Server-based 2PL schemes are derived from *primary-copy* concurrency control algorithms, with the server's copy of each page being treated as the primary copy of that page. Client transactions must obtain the proper lock from the server before they are allowed to access a data item. Clients are not allowed to cache locks across transaction boundaries, but do cache locks during a transaction so they do not have to check with the server for subsequent accesses within a transaction. In this study we use a variant called Caching 2PL *(C2PL)* which allows *data pages* to be cached at clients across transaction boundaries. Consistency is maintained using a "check-on-access" policy. When a transaction requests a read lock for a page that is cached at its client, it sends the Log Sequence Number (LSN) found on its copy of the page along with the lock request. When the server responds to the lock request, it includes an up-to-date copy of the page along with the response if it determines that the site's copy is no longer current. In C2PL, deadlock detection is performed exclusively at the server. Deadlocks are resolved by aborting the youngest transaction involved in the deadlock. An algorithm similar to C2PL is currently used in the client-server EXODUS storage manager.

2.2. Optimistic Two-Phase locking (O2PL)

The O2PL schemes allow inter-transaction caching of data pages and and an optimistic form of lock caching. They are based on a *read-one, write-all* concurrency control protocol that was developed for replicated data in distributed databases and was studied in [Care91b]. The O2PL algorithms are "optimistic" in the sense that they defer the detection of conflicts among locks cached at multiple sites until transaction commit time. In these algorithms, each client has its own local lock manager from which the proper lock must be obtained before a transaction can access a data item at that client. A non-two-phase read lock (i.e., latch) is obtained briefly at the server when a data item is in the process of being prepared for shipment to a client, thus ensuring that the client is given a transaction-consistent copy of the page. When a transaction wishes to commit, it must obtain exclusive access at the server to all of the data items that it updated. The server then coordinates activities at other clients to ensure that cache consistency is maintained. In order to determine what consistency actions are required, the server keeps track of where pages are cached in the system. Clients inform the server when they drop a page from their buffer by piggybacking that information on the next message that they send to the server. Because of this piggybacking, the server's information is conservative, as there may be some delay before the server learns that a page is no longer cached at a client.

When a transaction is ready to enter its commit phase, it sends a 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 message to all other clients that have cached copies of any of the updated pages. These remote clients obtain update-copy locks on their local copies of the updated pages on behalf of the committing transaction. Once all the required update-copy locks have been obtained, variant-specific O2PL actions are taken. O2PL-Invalidate (*O2PL-I*) invalidates the remote cached copies of data pages after obtaining the update-copy locks, while O2PL-Propagate (*O2PL-P*) propagates the new values of data items to remote caching sites; O2PL-Dynamic (*O2PL-D*) is an adaptive algorithm that chooses dynamically between invalidation and propagation on a per copy basis. If a new page value is propagated to any sites, then a two-phase commit protocol between those sites and the server is used to ensure atomicity. Invalidation does not require a two-phase commit, as it does not result in the updating of any data that remains valid at the site.

Since lock management is distributed in the O2PL algorithms, the clients are necessarily involved in deadlock detection. Locally, clients maintain a waits-for graph which is checked for cycles to detect deadlocks that are local to the client. Global deadlocks are detected using a centralized algorithm a la [Ston79]. The server periodically requests local waits-for graphs from the clients and combines them to build a global waits-for graph. As in the server-based case, deadlocks are resolved by aborting the youngest transaction. The use of update-copy locks allows for quicker detection of certain deadlocks: When a conflict is detected between two update-copy locks or between an update-copy lock and a write lock, it is known that a deadlock has occurred or will shortly occur, and therefore, the deadlock can be resolved immediately [Care91b].

2.3. Callback Locking

The third family of caching algorithms studied is callback locking. Callback locking allows caching of data pages and non-optimistic caching of locks. In contrast to the O2PL algorithms, with callback locking, clients must obtain a lock from the server immediately (rather than at commit time) prior to accessing a data page, if they don't have the proper lock cached locally. When a client requests a lock that conflicts with one or more locks that are currently cached at other clients, the server "calls back" the conflicting locks by sending requests to the sites which have those locks cached. The lock request is granted only when the server has determined that all conflicting locks have been released, so the callback scheme ensures that sites can never concurrently hold conflicting locks. As a result, transactions do not need to perform consistency maintenance actions during the commit phase. In this study, we analyze two callback locking variants: Callback-Read (CB-Read), which allows inter-transaction caching of read locks only, and Callback-All (CB-All), which allows inter-transaction caching of read locks and write locks.

As in the O2PL algorithms, the server keeps track of which sites have copies of pages in their cache. In CB-Read, which caches only read locks, the record that a client has a copy of a page is treated as an implicit read lock on the page. When a request for a write lock on a page arrives at the server, the server issues callback requests to all sites that have a cached copy of the page. At a client, such a callback request is treated as a request for an update-copy lock on the specified page. If the request can not be granted immediately, the client responds to the server saying that the page is currently in use. When the callback request is granted at the client, the page is removed from the client's buffer and an acknowledgement message is sent to the server. When all callbacks have been acknowledged to the server, the server grants a write lock on the page to the requesting client. Any subsequent read or write lock requests for the page will be blocked at the server until the write lock is released by the holding transaction. When a client transaction is done with a write lock (i.e., at the end of a transaction), it sends the new copy of the page to the server and releases the write lock, retaining a copy of the page in its cache.

The CB-All algorithm works similarly to CB-Read, except that write locks are kept at the clients rather than at the server and are not returned to the server at the end of a transaction. In this algorithm, the server's copy information is thus augmented to allow a copy at a site to be designated as an exclusive copy, and clients also keep track of which pages they have exclusive copies of. If a read request for a page arrives at the server and an exclusive copy of the page is currently held at some site, a downgrade request is sent to that site. A downgrade request is similar to a callback request, but rather than removing the page from its buffer, the client simply notes that it no longer has an exclusive copy of the page; in effect it downgrades its cached write lock to a read lock. Non-exclusive copies are treated as in the CB-Read algorithm. As in CB-Read, clients send copies of pages dirtied by a transaction to the server when the transaction commits. However, in CB-All this is done only to to simplify recovery, as no other sites can access a page while it is exclusively cached at another site.

Callback algorithms were originally introduced to maintain cache consistency in distributed file systems such as the Andrew File System [Howa88] and the file system of the Sprite operating system [Nels88]. However, these systems provide a weaker form of consistency than that required by database systems. More recently, a callback locking algorithm that provides serializability has been employed in the ObjectStore OODBMS. A callback locking algorithm which allows the caching of read locks but not write locks was studied in [Wang91]. An important area in which the callback algorithms of this study differ from the algorithm studied in [Wang91] is deadlock detection . In the latter algorithm, all cached locks are considered to be in-use for the purpose of computing the waits-for graph at the server. This may result in an unnecessarily high abort rate due to phantom deadlocks, especially with large client caches. To avoid this problem, the CB-Read and CB-All algorithms adopt a solution that is used in the Object-Store system. In this solution, copy sites must always respond to a callback request immediately, either with an acknowledgment that they have released the lock or with an indication that the lock is currently in use at the site. This allows deadlock detection to be performed at the server with accurate information.

2.4. Summary of the Algorithms

We now briefly summarize the three types of algorithms that are investigated in this study. All of the algorithms allow inter-transaction caching of data pages. The C2PL algorithm does not allow inter-transaction caching of locks — it uses a check-on-access technique for maintaining the consistency of cached data. The O2PL algorithms allow an optimistic form of lock caching: locks are obtained locally at clients during transaction execution, deferring global acquisition of locks until the commit phase. The three O2PL variants use invalidation and/or propagation to maintain consistency. The callback locking algorithms allow true inter-transaction caching of locks, but they require non-cached locks to be obtained at the server immediately during transaction execution. The callback locking algorithms covered in this study both use invalidation to maintain consistency.

3. A CLIENT-SERVER CACHING MODEL

This section presents an overview of the simulation model and workloads used in this study. A more detailed description of both can be found in [Care91a].

3.1. The System Model

Figure 1 shows the structure of the simulation model, which was constructed using the DeNet discrete event simulation language [Livn88]. It consists of components that model diskless client workstations and a server machine (with disks) that are connected over a simple network. Each client site consists of a *Buffer Manager* that uses an LRU page replacement policy, a *Concurrency Control Manager* that is used as either as a simple lock cache or as a full-function lock manager (depending on the cache consistency algorithm in use), a *Resource Manager* that provides CPU service and access to the network, and a *Client Manager* that coordinates the execution of transactions at the client. Each client also has a module called the *Transaction Source* which submits transactions to the client according to the workload model described in the following section. Transactions are represented as page reference strings and are submitted to the client one at a time; upon completion of a transaction, the source waits for a specified think time and then submits a new transaction. When a transaction aborts, it is resubmitted with the same page reference string. The number of client machines is a parameter to the model. The server is modeled similarly to the clients, but with the following differences: the Resource Manager manages disks as well as a CPU, the Concurrency Control Manager has the ability to store information about the location of page

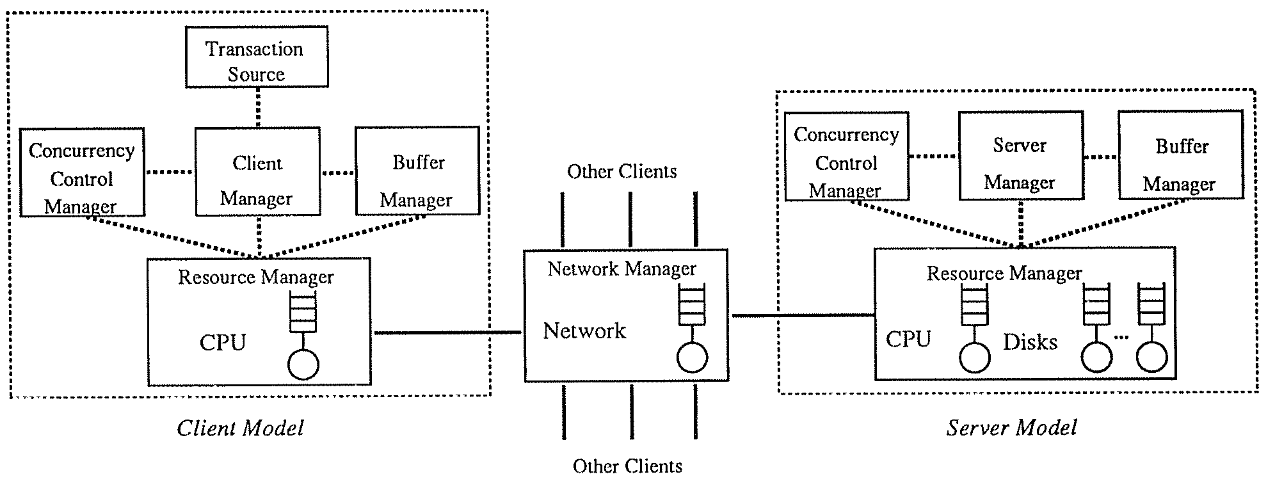

Figure 1: Model of a Client-Server DBMS

copies in the system and also manages locks, there is a *Server Manager* component that coordinates the operation of the server (analogous to the client's Client Manager), and there is no Transaction Source module (since all transactions originate at client workstations).

Table 1 describes the parameters that are used to specify the resources and overheads of the system and shows the settings used in this study. The simulated CPUs of the system are managed using a two-level priority scheme. System CPU requests, such as those for message and disk handling, are given priority over user (transaction) requests. System requests are handled using a FIFO queueing discipline,

Parameter	Meaning	Setting
ClientCPU	Instruction rate of client CPU	15 MIPS
ServerCPU	Instruction rate of server CPU	30 MIPS
ClientBufSize	Per-client buffer size	5% or 25% of database size
ServerBufSize	Server buffer size	50% of database size
ServerDisks	Number of disks at server	2 disks
MinDiskTime	Minimum disk access time	10 millisecond
MaxDiskTime	Maximum disk access time	30 milliseconds
NetworkBandwidth	Network bandwidth	8 or 80 megabits per second
PageSize	Size of a page	4,096 bytes
DatabaseSize	Size of database in pages	1250 (5 MBytes)
NumClients	Number of client workstations	1 to 25
FixedMsgInst	Fixed no. of instructions per message	20,000 instructions
PerByteMsgInst	No. of addl. instructions per msg. byte	10,000 inst. per 4 Kbyte page
ControlMsgSize	Size in bytes of a control message	256 bytes
LockInst	No. of instructions per lock/unlock pair	300 instructions
RegisterCopyInst	No. of inst. to register/unregister a copy	300 instructions
DiskOverheadInst	CPU Overhead for performing disk I/O	5000 instructions
DeadlockInterval	Global deadlock detection frequency	1 second (if needed)

Table 1: System and Overhead Parameters

while a processor-sharing discipline is employed for user requests. Each disk has a FIFO queue of requests; the disk used to service a particular request is chosen uniformly from among all the disks at the server. The disk access time is drawn from a uniform distribution between a specified minimum and maximum. A very simple network model is used in the simulator's *Network Manager* component; the network is modeled as a FIFO server with a specified bandwidth. We did not model the details of the operation of a specific type of network (e.g., Ethernet, token ring, etc.). Rather, the approach we took was to separate the CPU costs of messages from the on-the-wire costs of messages, and to allow the on-the-wire message costs to be adjusted using the bandwidth parameter. The CPU cost for managing the protocol for a send or a receive of a message is modeled as a fixed number of instructions per message plus a charge per message byte.

3.2. Workloads

Our simulation model provides a simple but flexible mechanism for describing workloads. The access pattern for each client can be specified separately using the parameters shown in Table 2. Transactions are represented as a string of page access requests in which some accesses are for reads and others are for writes. Two ranges of database pages can be specified: a hot range and a cold range. The probability of a page access being to a page in the hot range is specified; the remainder of the accesses are directed to cold range pages. For both ranges, the probability that an access to a page in the range will be a write access (in addition to a read access) is specified. The parameters also allow the specification of an average number of instructions to be performed at the client for each page access, once the proper lock has been obtained. This number is doubled for write accesses. By overlapping the ranges across clients and adjusting the write probabilities for each range, it is possible to create workloads with many different locality, data sharing, and data contention properties.

Table 3 summarizes the five workloads that are used in this study. Briefly, the HOTCOLD workload has a high degree of locality per client as well as a moderate amount of sharing and data contention among clients. The PRIVATE workload has high locality per client and only read sharing among clients. There are no read-write or write-write conflicts in this workload — it is intended to model a CAD-type application in which users have a private area of the database that they read and write while also reading

Parameter	Meaning
TransactionSize	Mean number of pages accessed per transaction
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
PerPageInst	Mean no. of CPU instructions per page on read (doubled on write)
ThinkTime	Mean think time between client transactions

Table 2: Workload Parameter Meanings

Parameter	HOTCOLD	PRIVATE	FEED	UNIFORM	HICON
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 250
ColdBounds	rest of DB	626 to 1,250	rest of DB	whole DB	251 to 1250
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	—	0.0, 0.05, 0.10, 0.25, or 0.5
ColdWriteProb	0.2	0.0	0.0/0.0	0.2	0.2
PerPageInst	30,000	30,000	30,000	30,000	30,000
ThinkTime	0	0	0	0	0

Table 3: Workload Parameter Settings for Client n

information from a shared library or a prior version of a design. The FEED workload represents a situation such as a stock quotation system in which one site produces data while the other sites consume it. UNIFORM is a low-locality, moderate write probability workload which is used to examine the properties of the consistency algorithms in a case where caching is not likely to pay off significantly. Finally, HICON is a workload with varying degrees of data contention. It is similar to skewed workloads that are often used to study shared-disk transaction processing systems, and it is introduced here to investigate the effects of data contention on the various algorithms. It should be emphasized that these are all synthetic workloads that are intended to capture important aspects of several classes of real workloads. None was derived from a real application since, at present, well-specified OODBMS workload descriptions are difficult to come by.

4. EXPERIMENTS AND RESULTS

In this section we present the results of the performance studies of the new heuristic for the adaptive O2PL algorithm and of the callback locking algorithms. For most experiments the main metric presented is throughput, measured in transactions per second. Where necessary, auxiliary performance measures are also shown. An additional metric that is used in several of the experiments is the number of messages sent per committed transactions. This metric is computed by taking the total number of messages sent during the simulation runs (by clients and the server) and dividing by the number transaction commits that occur during the simulation run. Results are presented for two different network bandwidths called "slow" (8 Mbits/sec) and "fast" (80 Mbits/sec); these correspond to slightly discounted bandwidths of Ethernet and FDDI technology, respectively.

4.1. A New Adaptive Heuristic

As described earlier, the heuristic for the dynamic O2PL algorithm used in [Care91a] generally performed well, but it was never able to match the performance of the better of the other two O2PL algorithms for a given workload (except for with the PRIVATE workload, where all O2PL algorithms performed identically). During the analysis of the results of the earlier study, it became apparent that there was no intrinsic reason that a dynamic algorithm should not be able to match, or even exceed, the performance of the static O2PL algorithms. For this reason, we re-visited the issue of heuristics for the dynamic algorithm.

The original O2PL-D algorithm uses a simple heuristic to determine whether to use invalidation or propagation in order to ensure that a site does not retain an out-of-date copy of a page. It initially propagates updates, invalidating copies on a subsequent consistency operation if it detects that the preceding page propagation was wasted. Specifically, O2PL-D will propagate a new copy of a page if both: *1) the page is resident at the site when the consistency operation is attempted*, and *2) if the page was previously propagated to the site, then the page has been re-accessed since that propagation.* In the initial study, it was found that the algorithm's willingness to err on the side of propagation resulted in its performance being somewhat lower than that of O2PL-I for the HOTCOLD and UNIFORM workloads. As a result, the approach taken towards developing an improved heuristic was to look at ways of instead erring on the side of invalidation if a mistake was to be made. To achieve this, we propose a new dynamic algorithm, O2PL-ND (for "New Dynamic"), which adds a third condition for propagation. A new copy of a page will be propagated to a site by O2PL-ND only if conditions 1 and 2 of O2PL-D hold plus *3) the page was previously invalidated at that site and that invalidation was a mistake.* The new condition ensures that O2PL-ND will invalidate a page at a site at least once before propagating it. The retention of the previous two conditions ensures that O2PL-ND will cease propagating a page to a site once the page no longer appears to be useful at that site.

In order to implement the new condition, it is necessary to have a mechanism for determining that an invalidation was a mistake. This is not straightforward since invalidating a page removes the page and its buffer control information from the site, leaving no place to store information about the invalidation. To solve this problem we use a structure called the *invalidate window*. The invalidate window is a list of the last *n* distinct pages that have been invalidated at the site. The window size, *n*, is a parameter of the O2PL-ND algorithm that is set when the database system is initialized at each client. When a page is invalidated at a site, it is placed at the front of the invalidate window on that site. If the window is full and the page does not already appear in the window, then the entry at the end is pushed out of the window to make room for the new entry. If the page already appears in the window, however, it is simply moved from its present location to the front. When a transaction page access request results in a page being brought into the site's buffer pool, the invalidate window is checked and if the page appears in the window, the page is marked as having had a mistaken invalidation applied to it.[1] The page remains marked as such as long as it resides in the client's buffer.

Except for the different heuristic, the O2PL-ND algorithm works much like O2PL-D. When a consistency action request arrives at a client, the client checks to see if the page is resident at the site. If not, then the client ignores the page and responds that an updated copy of the page is not needed. If the page

[1] The marking decision is made at this time rather than at the time a consistency action is required in order to limit the algorithm's sensitivity to the window size. If the window was checked at consistency maintenance time, then a small window size could result in useful pages being invalidated because they have been "pushed out of the window" even while they were being used at the client.

is resident at the site, then an exclusive lock is obtained on the site's copy for the consistency action. The client then checks the remaining two conditions for propagation and if both conditions are met the client will decide to receive a new copy of the page. When the site has made a decision for all of the pages affected in the consistency maintenance phase of the transaction, it tells the server which pages it wishes to receive copies of. If the site decided not to propagate any of the changes, then the affected pages are purged from the buffer, the locks are released, and the client informs the server that it does not require any pages. However, if the site decided to propagate some or all of the changed pages, then it retains the locks on those pages (invalidating the other pages, if any) and sends a message to the server that acts as the acknowledgement in the first phase of a two-phase commit protocol and that also informs the server which pages it requires copies of. When the server has heard from all involved clients, it sends copies of the necessary pages to those sites that requested them. This message serves as the second phase of the commit protocol. Upon receipt of the new page copies, the clients install them in their buffer pools and release the exclusive locks on those pages.

To determine the performance of the new heuristic, the algorithm was run for the HOTCOLD, FEED, and UNIFORM workloads described in Section 3.2. The PRIVATE workload was not used since its lack of data contention causes all O2PL algorithms to perform identically. In the following, results are shown for five algorithms: C2PL, O2PL-I, O2PL-P, O2PL-D and O2PL-ND. In all of the experiments described here, the O2PL-ND algorithm was run with a window size of 20 pages. An early concern with the new heuristic was whether it would be too sensitive to the window size. A series of studies using the parameters from [Care91a] found that in most cases, the algorithm was insensitive to the window size within a range of 10 to 100 pages (0.8% to 8% of the database size) [Fran92a]. The only exception was the FEED workload when run with 5% client buffers, in which a slight sensitivity was noticeable for window sizes up to 50 pages. This case and the reasons for the insensitivity of the algorithm in the other cases are discussed in the following sections.

4.1.1. Experiment 1 : The HOTCOLD Workload - Slow Network

Figure 2 shows the throughput for the HOTCOLD workload using the slow network and a client buffer size of 5% of the database size (62 pages). This case is a clear example of how the new heuristic of the O2PL-ND algorithm can improve performance over the heuristic used by the original O2PL-D algorithm. O2PL-ND performs as well as O2PL-I, the better of the static O2PL algorithms in this case, while O2PL-D tracks the lower performance of O2PL-P. All of the algorithms eventually become disk-bound in this case: C2PL hits the disk bottleneck at 25 clients, while the O2PL algorithms become disk-bound at 20 clients. In this experiment, all of the O2PL algorithms outperform the C2PL algorithm prior to reaching the disk bottleneck. This is due to the additional latency incurred by C2PL's heavy reliance on messages, as shown in Figure 3. Once the algorithms become disk-bound, their performance is determined by their disk requirements per commit as shown in Figure 4. The similarity of O2PL-D's performance to that of O2PL-P in this case is due to the fact that O2PL-D will propagate a page once to a site before it detects that it should have invalidated the page. Since the client buffer pool is relatively small in

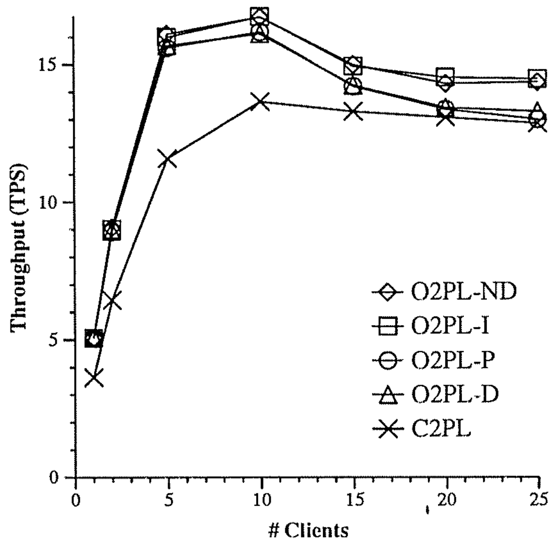

Figure 2: Throughput (HOTCOLD, 5% Cli Buffers, Slow Net)

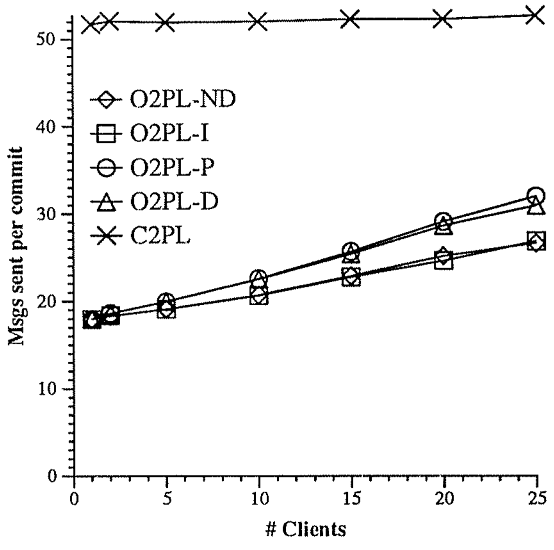

Figure 3: Messages Sent per Commit (HOTCOLD, 5% Cli Buffers, Slow Net)

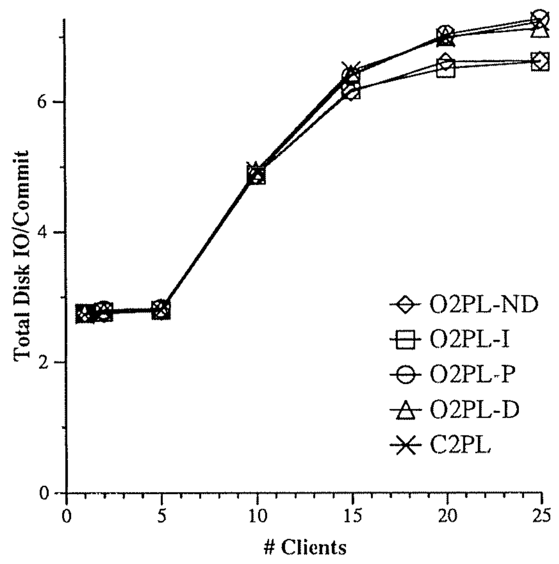

Figure 4: Total Disk I/O per Commit (HOTCOLD, 5% Cli Buffers, Slow Net)

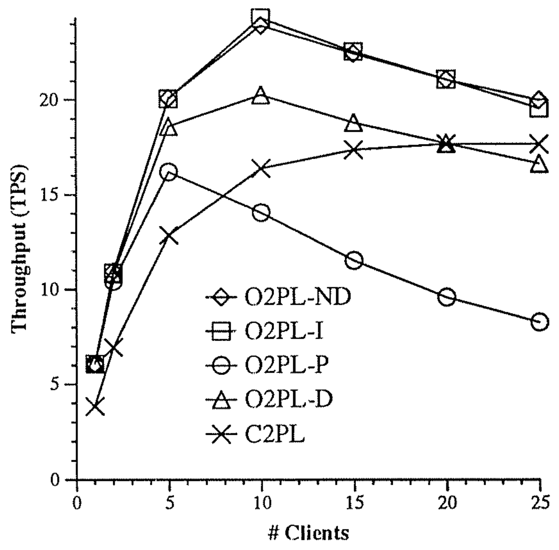

Figure 5: Throughput (HOTCOLD, 25% Cli Buffers, Slow Net)

this experiment, O2PL-P performs only slightly more propagations than O2PL-D, as an unused page is likely to be pushed out of the buffer pool by the LRU page replacement algorithm. About 80% of the propagations go unused for both O2PL-P and O2PL-D (that is, these propagated pages are not accessed before they are forced out of the buffer pool or another consistency action arrives for the page).

The lower performance of O2PL-P and O2PL-D, as compared to O2PL-I and O2PL-ND, is due to both messages and disk I/O. The additional messages are due to the propagations just described. As was seen in [Care91a], the additional disk I/O is caused by slightly lower server and client buffer hit rates. These lower hit rates are the result of the fact that, with propagation of updates, pages are removed from the client buffers only when they age out of the buffer pool. Aged-out pages are less likely to be found in the server buffer than recently invalidated pages if they are needed (thus lowering the server buffer hit rate), and they take up valuable space for too long in the small client buffer pool if they are not needed (thus lowering the hit rate on the client).

The O2PL-ND algorithm performs similarly to O2PL-I mainly because the majority of the consistency operations performed by O2PL-ND are invalidates; at 5 clients and beyond, fewer then 10% of its consistency operations are propagations. Furthermore, of the propagations performed, between 39% and 62% performed prove useful, so their net affect on system performance is minimal. These effects occur because the invalidation window is small relative to the database size. As the invalidate window size approaches the database size, the O2PL-ND algorithm becomes more like the original O2PL-D algorithm. In [Fran92a] it was seen that the number of propagations per committed transaction increases with the window size, but that it remains quite small in the range of window sizes tested (10 to 100). The number of useless propagations grows slowly and smoothly enough with the invalidate window size that there is reasonable room for error in choosing the window size (as long as it is kept well below the size of the database).

Figure 5 shows the throughput results for the HOTCOLD workload and slow network when the client buffer size is increased to 25% of the database size (312 pages). This case shows a clear example of the effect of the new system parameter settings used in this study. In [Care91a], which used slower CPUs and a faster network, the O2PL-D algorithm achieved performance much closer to that of O2PL-I for this case. The parameter settings used in [Care91a] were based on the intuition that the network itself would not ever be a bottleneck. However, the continuing disparity in the performance increases of CPUs and networks has increased the likelihood of a network bottleneck arising. The effect of these technology trends is an increase in the penalty paid for useless propagations and as a result, the relative performance of the O2PL-D algorithm suffers compared to what was observed in [Care91a]. These trends also exacerbate the performance problems incurred by the O2PL-P algorithm due to excessive propagation. In this case, the O2PL-P and O2PL-D algorithms become network-bound, while the O2PL-I and O2PL-ND algorithms approach a disk bottleneck at 25 clients.

In terms of the new heuristic, the results in this case are similar to those for the 5% client buffer case; the O2PL-ND algorithm very closely matches the performance of the O2PL-I algorithm. Once again, O2PL-ND closely matches O2PL-I because the vast majority of the consistency operations that it performs are invalidations. Furthermore, approximately 99% of the invalidations are the result of pages not being in the invalidate window (condition 3). This shows that O2PL-ND's invalidate window is effective in avoiding even the single mistake that is made by O2PL-D, which explains its superior performance.

4.1.2. Experiment 2 : The HOTCOLD Workload - Fast Network

Figure 6 shows the throughput results for the HOTCOLD workload using the fast network and the smaller client buffer pool size. These throughput results are similar to those of the slow network case in that O2PL-I and O2PL-ND have similar performance, while O2PL-P and O2PL-D have similar but lower

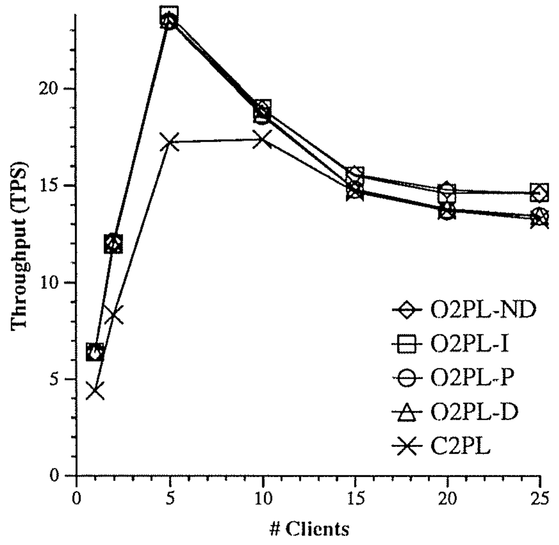

Figure 6: Throughput (HOTCOLD, 5% Cli Buffers, Fast Net)

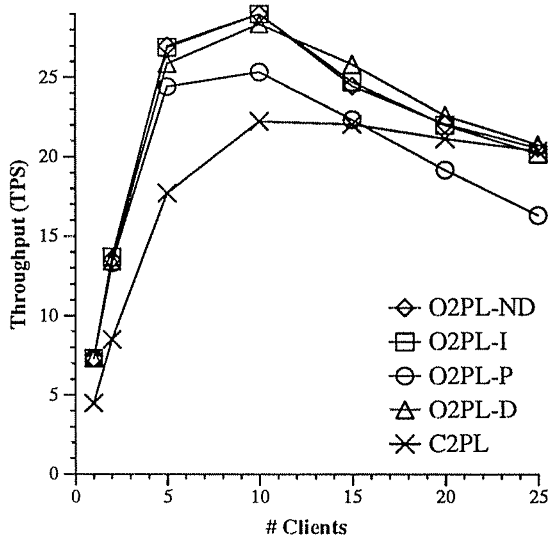

Figure 7: Throughput (HOTCOLD, 25% Cli Buffers, Fast Net)

performance. However, in this case, the performance differences among the O2PL algorithms are primarily driven by disk I/O (all algorithms are eventually disk-bound), and are caused by the algorithms' buffering characteristics described previously. Figure 7 shows the throughput when the larger client buffer size is used. In this case, it can be seen that the original dynamic algorithm performs significantly better than O2PL-P, as it performs many fewer propagations than O2PL-P. In fact, O2PL-P becomes CPU-bound at the server due to message processing overhead, while all other algorithms eventually become disk-bound. It is interesting to note that with these parameter settings, at 15 clients and beyond, O2PL-D matches and even slightly exceeds the performance of the O2PL-I and O2PL-ND algorithms. This is due to the fact that O2PL-D has a better client hit rate than these other algorithms, and thus requires fewer disk I/Os per transaction. Its improved hit rate is due to the fact that O2PL-D performs more than twice as many useful propagations as O2PL-ND. Of course, O2PL-D also performs many more wasted propagations than O2PL-ND, but the overhead of these propagations is not high enough here to cause O2PL-D to become network or server CPU-bound, so it eventually hits the disk bottleneck. In addition, the large buffer-pool size ensures that hot range pages will remain in the client buffer pool despite the wasted propagations.

4.1.3. Summary of the HOTCOLD Experiments

The results of the HOTCOLD experiments show that O2PL-ND performs at about the same level as O2PL-I. This is an improvement over the heuristic used by O2PL-D. However, O2PL-ND does not significantly outperform O2PL-I in any of the HOTCOLD cases tested. This is because the nature of the HOTCOLD workload makes propagation a bad idea in general; *from the view of a particular client*, most updates that are performed at other clients will be on pages that are in the cold region of the database. Therefore, propagation of updates from other clients will typically not be helpful.

4.1.4. Experiment 3: The FEED Workload

In contrast to the HOTCOLD workload, where propagation is generally a bad idea, the FEED workload was shown in [Care91a] to benefit from propagation. This result also holds in general in this study, however, an exception arises in the case with small client buffer sizes and the slow network. The throughput results for this case are shown in Figure 8. In this case, the O2PL algorithms all perform better than C2PL, with O2PL-I having the highest throughput at 15 clients and beyond, while O2PL-P has the lowest of the O2PL algorithms. The dynamic algorithms fall roughly between the two static ones, with O2PL-ND having a slight advantage. This ordering is due to the combination of the slow network, which makes propagations expensive, and the small buffer pool, which causes propagations to be wasted. For the O2PL algorithms that do propagation, only 53% to 60% of the propagations actually prove to be useful. Moreover, due to the small client buffer pools, many of these propagations are used only once before the page is thrown out. A useless propagation costs a page-sized message. On the other hand, a propagation that is used exactly once saves only a smaller control message (the page request message that would be required to obtain the page from the server on the subsequent use had the page not been propagated to the site). Thus, if many "useful" propagations are used only once before being thrown out of the buffer pool or re-propagated to, the net effect of a small majority of useful propagations can be a reduction in the number of messages but an actual increase in the demand for network bandwidth. This is the effect seen in Figure 8.

As mentioned previously, the FEED workload with small client buffer pools was the only situation examined where the invalidate window size of O2PL-ND made a noticeable difference within the range of 10 to 100 pages. In fact, differences are only noticed at window sizes of 50 and smaller, as a window

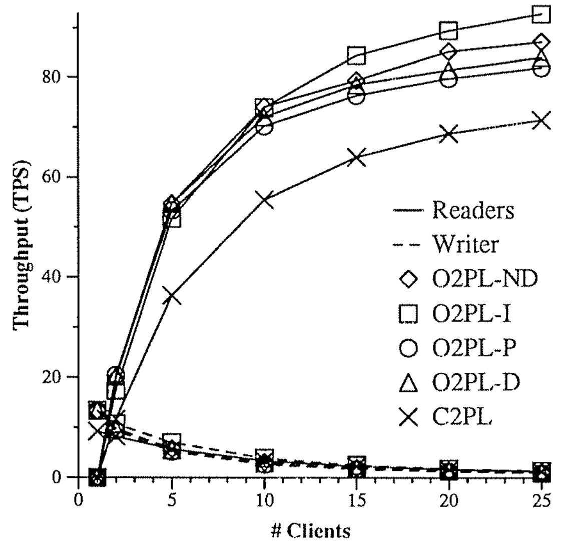

Figure 8: Throughput
(FEED, 5% Cli Buffers, Slow Net)

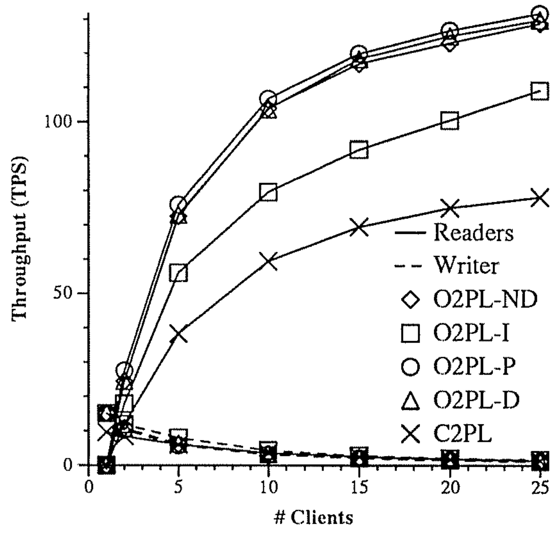

Figure 9: Throughput
(FEED, 25% Cli Buffers, Slow Net)

size of 50 pages allows all pages that are updated by the FEED workload to fit in the window. Therefore, at 50 pages and beyond, O2PL-ND acts exactly like the O2PL-D algorithm since there are no invalidations due to the absence of a page in the invalidation window. The performance of the O2PL-ND algorithm in this case is bracketed by the O2PL-I algorithm (i.e., a window size of 0) and the O2PL-D algorithm. In this test, at 10 clients and beyond, the O2PL-ND algorithm (window size = 20) chooses to invalidate remote copies about 40% of the time, while O2PL-D chooses to invalidate only about 15% of the time.

Figure 9 shows the throughput of the FEED workload with the slow network when the client buffer pool size is increased to 25% of the database size. As in the previous case, all of the algorithms approach a network bottleneck; in this case however, the O2PL-P algorithm significantly outperforms O2PL-I. The larger buffer pools allow all of the clients to keep their 50 hot range pages in memory, so many fewer of O2PL-P's propagations are wasted due to pages being forced out of the buffer pool. As a result, O2PL-I sends more page requests to the server than O2PL-P, which results in increased path length for transactions under O2PL-I. This additional path length includes additional messages (O2PL-I sends about 6 messages per transaction versus about 3 per transaction for O2PL-P), and additional lock requests at the server. The two dynamic algorithms have nearly identical performance in this case; they perform slightly worse than O2PL-P but much better than O2PL-I. The reason that they perform similarly is that the invalidate window has virtually no effect in this case (both algorithms invalidate between 15% and 22% of the time). In this experiment, the percentage of O2PL-ND's invalidations that were due *solely* to a page not being marked as recently invalidated ranged from 2% at two clients to 8% at 25 clients.

The reason that O2PL-ND's invalidate window has such a small impact in this case is due to the fact that, as stated previously, the window is checked when a page is read into memory rather than when a consistency action is required. With the large client buffer size, hot range pages are aged out of the buffer pool very infrequently, and thus hot range pages are removed mainly as the result of invalidations. Invalidated hot pages are likely to be re-referenced quickly, and are likely to still be in the invalidate window when the re-reference occurs. Such pages will then be marked as "recently invalidated" for their entire residency in the buffer pool. This effect was not seen in the smaller buffer case because hot pages were often aged out of the small buffer pool by the LRU mechanism; when an aged out page is reread, it may no longer be in the invalidate window, and thus will not be marked as recently invalidated.

Due to space limitations, the results for the fast network case are not shown here. For the small client buffer case, the O2PL algorithms again all had similar throughput; O2PL-P and O2PL-D performed slightly better than O2PL-ND, which was slightly better than O2PL-I. The increase in network bandwidth removed the advantage of smaller message sizes that was seen earlier for O2PL-I. For the larger client buffer case, the relative performance of the algorithms was similar to that just described for the slow network. In both fast network cases, C2PL becomes server CPU-bound due to its high message volume, so it performs well below the level of the other algorithms.

4.1.5. Experiment 4: The UNIFORM Workload

So far we have investigated one workload where invalidation is advantageous and one in which propagation performs well. We now briefly examine the UNIFORM workload, in which caching is not expected to provide much of a performance benefit. In the case of clients with the 5% buffer pool size, there was very little performance difference among any of the algorithms in the initial study. In this study, the lack of difference among the algorithms holds only for the fast network case. In the slow network case (not shown), even with the small client buffer pools, the O2PL-P and O2PL-D algorithms perform noticeably below the level of the other O2PL algorithms due to the message costs associated with wasted propagations. Once again, O2PL-D performs similarly to O2PL-P because the small buffer pool makes it likely that wasted propagations will be quickly replaced from the client buffer pool (thus limiting O2PL-P's opportunity to make mistakes).

More significant differences become noticeable when the client buffer size is increased to 25%. The throughput results for this case with the slow network are shown in Figure 10. As in the HOTCOLD case, O2PL-ND performs similarly to O2PL-I (and is essentially insensitive to the invalidate window size in the range of 10 to 100 pages). In this workload, propagations do not generally turn out to be useful (e.g., fewer than 15% of O2PL-P's propagations are useful in this case) and this causes algorithms that perform more propagations to suffer. This is because the workload's lack of locality means that pages are likely to be aged out of the buffer pool or updated elsewhere before they are accessed again at a site. Doing fewer propagations thus results in sending fewer messages, and hence, the O2PL-I and O2PL-ND algorithms perform better than the O2PL-P and O2PL-D algorithms. The O2PL-P and O2PL-D algorithms approach a network bottleneck at 25 clients, while the other three algorithms become disk-bound.

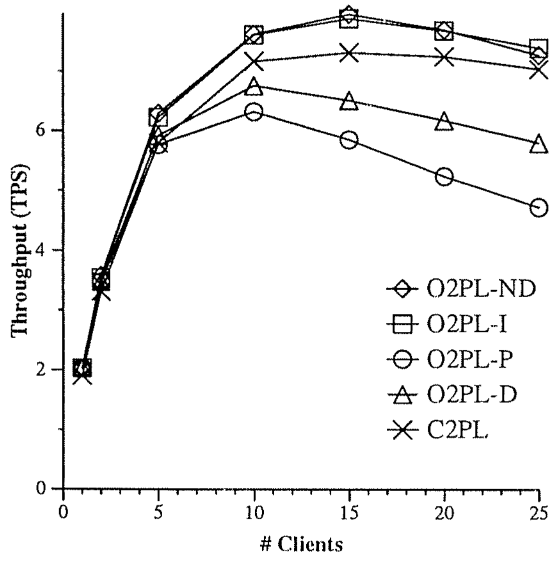

Figure 10: Throughput (UNIFORM, 25% Cli Buffers, Slow Net)

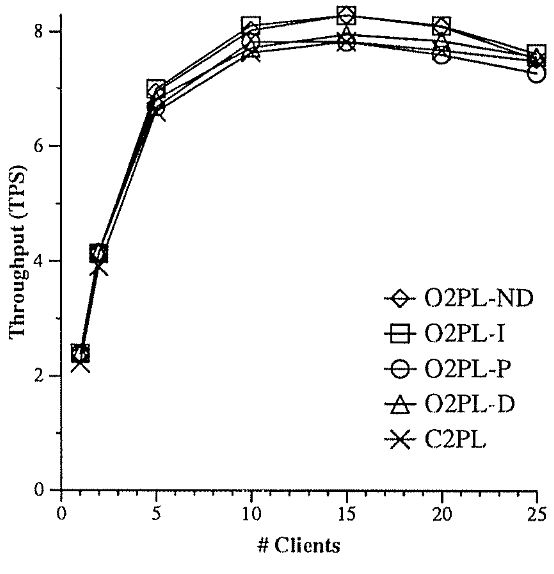

Figure 11: Throughput (UNIFORM, 25% Cli Buffers, Fast Net)

Figure 11 shows the throughput results when the fast network is used, resulting in the disk becoming the main bottleneck. In this case, the results are similar to what was seen in [Care91a] — invalidation has a slight advantage in this case. As a result, the O2PL-ND algorithm is able to fairly closely match the performance of the O2PL-I algorithm.

4.1.6. Summary

The experiments presented in Sections 4.1.1 thru 4.1.4 show that the new heuristic for the dynamic algorithm performs as well as the static O2PL-I algorithm in cases where invalidation is the correct approach, which O2PL-D was unable to do. In addition, it retains the performance advantages of the O2PL-D heuristic in cases where propagation is advantageous. Though the new heuristic never significantly outperformed the better of the static algorithms in any of the cases tested, it was shown to be a reasonable choice even in situations with static locality; as a result, it will certainly have advantages where locality patterns are mixed or dynamic. The effect of the combination of the faster CPUs along with the slow network setting used in this study caused the advantages of O2PL-ND over O2PL-D to be greater than those seen when the parameters of [Care91a] were used. Thus, in situations where the network has a significant impact on performance, the O2PL-ND algorithm is a much better choice than O2PL-D.

4.2. Callback Locking

In this section, we study the performance of the two callback locking algorithms that were described in Section 2: Callback-Read (CB-Read) and Callback-All (CB-All). For comparison purposes, we also show the performance of the O2PL-ND and C2PL algorithms. Again, experiments were run with client buffer pool sizes of 5% and 25% of the database size. Results are shown only for the 25% case since the important aspects of the performance of the CB algorithms are slightly more pronounced (but not qualitatively different) in the large client buffer pool case versus the small buffer case.

4.2.1. Experiment 1: The HOTCOLD Workload

Figure 12 shows the results of the HOTCOLD workload using the slow network and a client buffer pool size of 25%. The C2PL algorithm has the lowest throughput due to the combination of high message requirements (shown in Figure 13) and higher disk requirements (compared the others) due to its lower buffer hit rates. In general, the CB algorithms both perform at a somewhat lower level than O2PL-ND, which has the highest throughput overall. These results are driven primarily by the message requirements of the algorithms as shown in Figure 13 — the disk requirements of the two callback algorithms and the O2PL-ND algorithm are nearly identical, since O2PL-ND chooses to use invalidation for over 90% of its consistency operations. The CB algorithms send significantly more messages per commit than O2PL-ND. The difference between CB-Read and O2PL-ND is due to the fact that CB-Read performs consistency operations on a per page basis, while O2PL-ND performs consistency operations only at the end of the execution phase. By waiting until the end of the execution phase to perform consistency operations, O2PL-ND saves messages by sending fewer requests to the server for consistency actions (one

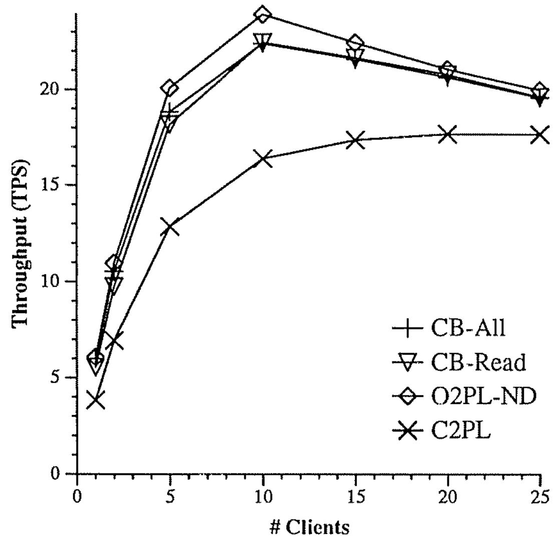

Figure 12: Throughput (HOTCOLD, 25% Cli Buffers, Slow Net)

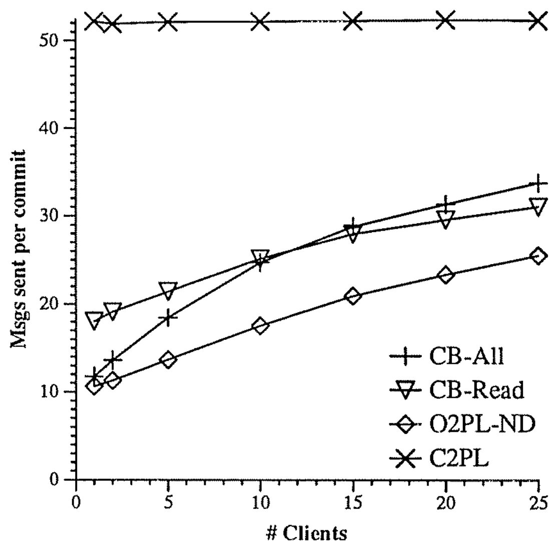

Figure 13: Messages Sent Per Commit (HOTCOLD, 25% Cli Buffers, Slow Net)

per transaction versus as many as one per write for CB-Read) and, to a lesser extent, by grouping multiple consistency requests for the same client in a single message. (In this experiment, O2PL-ND sent an average of between 1.1 and 1.2 consistency operations per consistency message.) In contrast to CB-Read, CB-All is allowed to cache write locks; this is the reason for its sending fewer messages than CB-Read when small numbers of clients are present. However, beyond 10 clients the caching of write locks results in a net increase in messages. For the callback algorithm, the advantage of caching a write lock is that a request for a lock on a site where it is already cached saves one round trip message with the server. However, the penalty for caching a write lock that conflicts with a read request at a different site is also a round trip message with the server. Since the write probability is only 20% in the HOTCOLD workload, caching write locks becomes a losing proposition when the number of clients is sufficient to make it more likely that a page will be read at some remote site before it is re-written at a site with a cached write lock. This result supports the intuition behind the decision in [Wang91] to cache only read locks.

Figure 14 shows the throughput for the HOTCOLD workload with the fast network and large client buffer pools. As shown in Figure 15, all of the algorithms eventually become disk-bound in this case, so the messaging effects discussed in the previous case eventually have no impact on the throughput. As a result, the performance differences between the CB algorithms disappear. Again, O2PL-ND has similar performance to the CB algorithms because it chooses to invalidate pages most of the time, and thus, its buffering behavior is similar to that of the invalidation-based CB algorithms.

4.2.2. Experiment 2: The PRIVATE Workload

The PRIVATE workload has high locality but no data contention. As a result, algorithms which cache data and locks perform very well on this workload. The throughput results for the PRIVATE

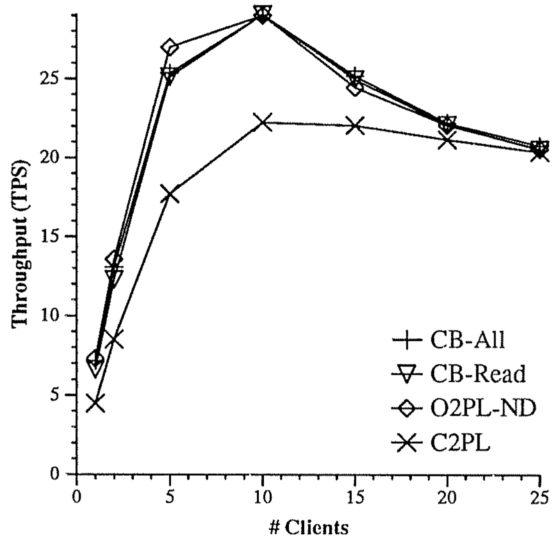

Figure 14: Throughput
(HOTCOLD, 25% Cli Buffers, Fast Net)

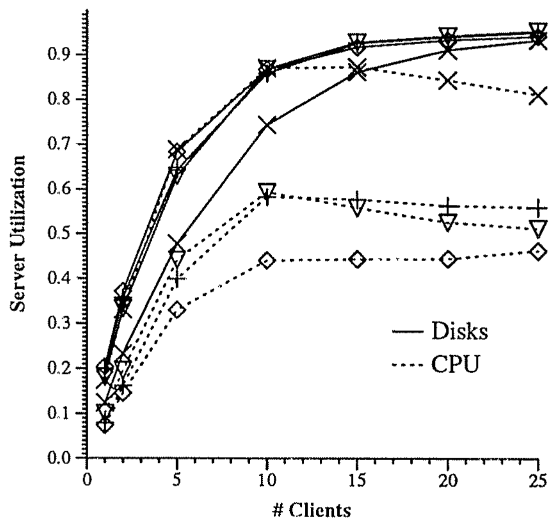

Figure 15: Server Resource Utilizations
(HOTCOLD, 25% Cli Buffers, Fast Net)

workload using the slow network are shown in Figure 16. The results seen here are again due to the message requirements of the algorithms. C2PL requires an average of 39 messages per commit throughout the range of client populations, while CB-Read requires 15 messages and CB-All requires 12 messages.

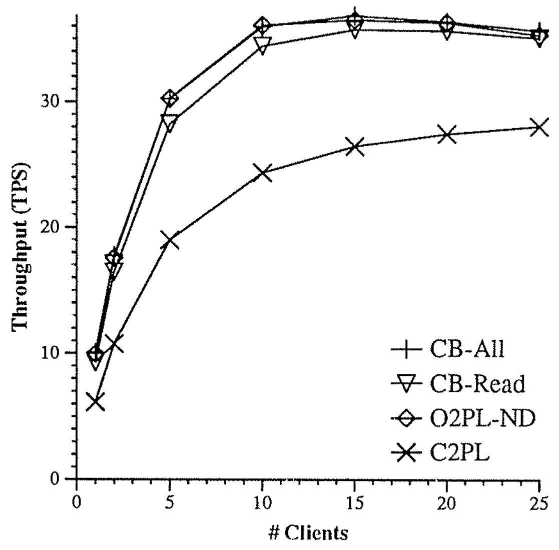

Figure 16: Throughput
(PRIVATE, 25% Cli Buffers, Slow Net)

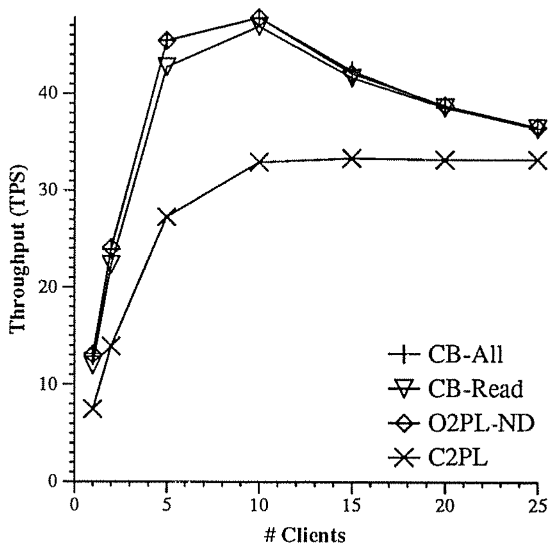

Figure 17: Throughput
(PRIVATE, 25% Cli Buffers, Fast Net)

O2PL-ND's message requirements range from 12 messages per commit at 1 client to 13.4 at 25 clients.[2] All of the algorithms except C2PL become network-bound at 10 clients and beyond. C2PL eventually reaches a network bottleneck at 25 clients. C2PL does not reach the network bottleneck earlier because most of the messages it sends are small control messages (e.g., lock requests and replies). These small messages increase the transaction path length due to CPU processing at the server and the clients. The other algorithms send fewer messages per transaction, but many of these messages contain data pages, which consume more of the network bandwidth. This results in making network bandwidth a larger factor in their path length. C2PL's additional messages in this case are the result of not caching read locks or write locks. The performance of CB-Read also suffers because it does not cache write locks, which leads CB-Read to send a message to the server for every page written by a transaction. When the PRIVATE workload is run with the fast network (Figure 17), the C2PL algorithm becomes CPU-bound at the server due to message overhead, while the other algorithms become disk-bound at 15 clients and beyond. Prior to hitting the disk bottleneck, the CB-Read algorithm pays a slight cost due to added path length for obtaining write locks. Note that in this workload, there is no cost for caching write locks because of the absence of read-write and write-write data sharing among clients. This provides an advantage for CB-All over CB-Read in regions where messages have a significant impact on performance. Once the disk bottleneck is reached, CB-Read, CB-All, and O2PL-ND all perform similarly.

4.2.3. Experiment 3: The FEED Workload

The reader and writer throughput results for the FEED workload with the slow network are shown in Figure 18. O2PL-ND has the best reader throughput prior to 10 clients, while CB-Read and CB-All have the best reader throughput beyond 10 clients. The writer throughput is similar for all of the algorithms at 5 clients and beyond. O2PL-ND and CB-All have better writer throughput when there are 0 or 1 reader sites. Once again, the throughput results are driven by the message requirements of the algorithms. Figure 19 shows the messages sent per commit averaged over the writer and all of the readers for each of the algorithms.[3] O2PL-ND's higher initial reader throughput is due to its lower message requirements, which result from its reduced consistency message requirements (as described previously) and a better client buffer hit rate due to propagations. All of the algorithms except for C2PL approach a network bottleneck at 15 clients and beyond. As the network becomes saturated, the propagations performed by O2PL-ND cause its reader performance to suffer compared to the callback algorithms. In terms of the callback algorithms, the writer site in CB-Read has to request every write lock from the server, while in the CB-All algorithm, the writer has to request write locks that have been called back (virtually all write locks at five clients and beyond). CB-All also requires extra messages for readers to callback the write locks that are cached at the writer site. As a result, CB-All's writer site receives and sends many more messages per commit than CB-Read's writer site. For example, with 15 clients CB-All's writer sends nearly 26

[2] The slight increase in message requirements for O2PL-ND is due to messages for global deadlock detection: during each deadlock detection interval, a round-trip message is sent between the server and each client. In this case, the throughput remains roughly constant as clients are added to the system (beyond 10 clients) so the additional deadlock detection messages increase the per commit total.

[3] Note that since client number 1 is the writer site, the data points for one client in Figure 19 show the messages per writer transaction commit in the absence of conflicting readers.

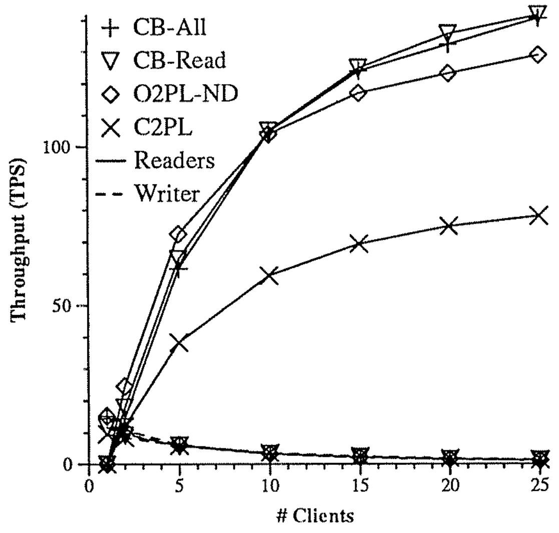

Figure 18: Throughput
(FEED, 25% Cli Buffers, Slow Net)

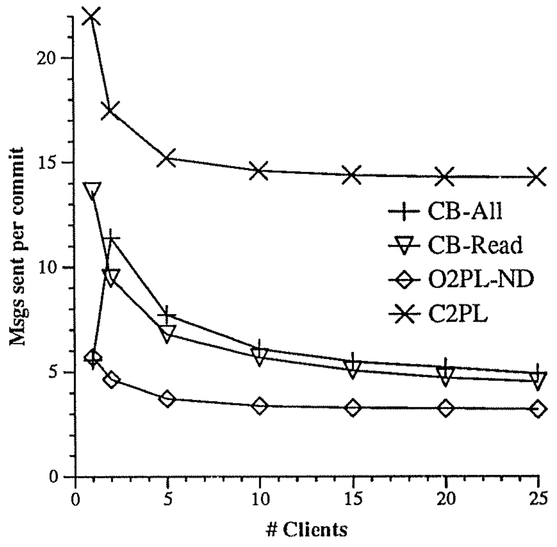

Figure 19: Messages Sent Per Commit
(FEED, 25% Cli Buffers)

messages per writer commit and at 25 clients it sends nearly 43 messages per writer commit. On the other hand, CB-Read's writer site sends about 7 messages per writer commit from 1 to 25 clients. The callback requests for write locks are the cause of the difference in the message requirements for the CB algorithms seen in Figure 19. Furthermore, there is little or no benefit to caching write locks here, as the updated pages are in the hot region of all of the readers and are thus likely to be read at reader clients. The C2PL algorithm has the highest message requirements for reader sites, as such sites must send a message to the server for each page accessed. Once again, many of these messages are small, so C2PL is affected by increased server and client CPU requirements and does not become network-bound.

The throughput results for the FEED workload with the fast network are shown in Figure 20. Again, in this case, all of the algorithms except for C2PL approach (but do not quite reach) a disk bottleneck at 25 clients. C2PL eventually becomes server CPU-bound due to lock requests that are sent to the server. In this case, the propagations performed by O2PL-ND give it a better buffer hit rate at the reader clients. However, this translates to only a slight reduction in disk reads compared to the callback algorithms, as hot pages missed at clients due to invalidations are very likely to be in the server's buffer pool. Thus, in the range of clients studied, the relative performance of the algorithms is still largely dictated by the message characteristics described for the slow network case. With the fast network, the size of the messages has a smaller impact on performance, therefore, the relative performance of the algorithms in this case is more in line with the message requirements shown in Figure 19.

4.2.4. Experiment 4: The UNIFORM Workload

The throughput results for the UNIFORM workload using the slow network are shown in Figure 21. None of the algorithms hits a resource bottleneck in the range of clients, shown, so the throughput results

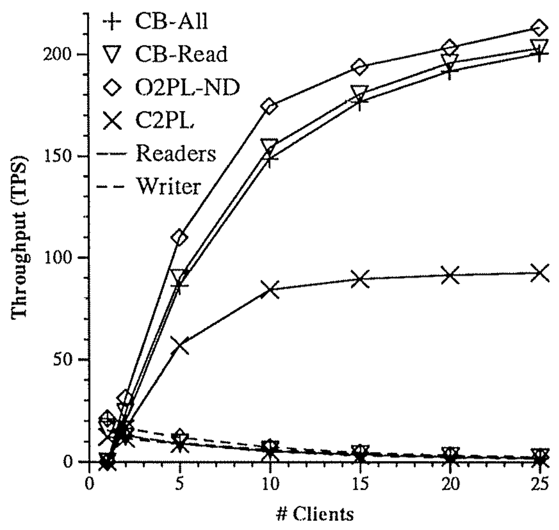

Figure 20: Throughput (FEED, 25% Cli Buffers, Fast Net)

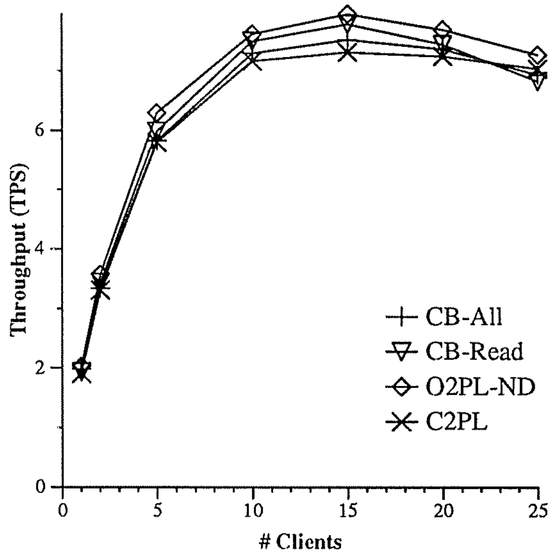

Figure 21: Throughput (UNIFORM, 25% Cli Buffers, Slow Net)

are all latency driven. O2PL-ND achieves the highest throughput across the range of client populations, with the callback algorithms performing below O2PL-ND but better than C2PL through most of the range. CB-Read performs slightly better than CB-All because caching write locks is costly due to the lack of locality and the low write probability. The decline in throughput for the three lock caching algorithms is due to their increased message requirements for consistency operations in this low-locality workload. At 15 clients and beyond, all three of the lock caching algorithms send more messages than C2PL, which does not cache locks. While C2PL's message requirements per commit remain constant as clients are added, the lock caching algorithms are forced to send consistency operations (e.g., invalidations or callback requests) to more sites. C2PL's lower performance through most of the range is due to its higher disk requirements resulting from low client and server buffer hit rates. The reason that the algorithms do not quite reach a bottleneck in this case is the higher level of data contention than was seen in the other workloads. When the fast network is used (not shown), the network effects are removed and all algorithms eventually approach a disk bottleneck. In this case, the lock caching algorithms perform similarly, and slightly better than C2PL, due to the buffer pool effects seen in the preceding experiments.

4.2.5. Experiment 5: The HICON workload

The final workload examined in this paper is the HICON workload. While we do not expect high data contention to be typical for client-server DBMS applications, we use this workload to examine the robustness of the algorithms in the presence of data contention and to gain a better understanding of their different approaches to detecting conflicts. As described in Section 3.2, this workload has a 250 page hot range that is shared by all clients. The write probability for hot range pages is varied from 0% to 50% in order to study different levels of data contention. In this section we briefly describe the HICON results, using the fast network and large client buffer pools. Figure 22 shows the throughput for HICON with a hot write probability of 5%. In the range of 1 to 5 clients, the lock caching algorithms perform similarly and C2PL has lower performance than the others. C2PL's lower performance in this range is due to its significantly higher message requirements here. These results are latency-based, as no bottlenecks develop in this range — in fact, no resource bottlenecks are reached by any of the algorithms in this experiment. At 10 clients and beyond, the effects of increased data contention become apparent; O2PL-ND's performance suffers and it has the lowest utilization of all three major system resources (disk, server CPU, and network). O2PL-ND has a somewhat higher level of blocking for concurrency control than the other algorithms (e.g., at 15 clients, approximately 42% of its transactions are blocked at any given time versus approximately 37% for the callback algorithms and 35% for C2PL). O2PL-ND's higher blocking level is due to the fact that some (global) deadlocks are detected in O2PL-ND using periodic deadlock detection, while in the other algorithms, all deadlocks can be detected immediately at the server. All of the algorithms suffer from increased blocking as clients are added. In addition, the lock caching algorithms suffer from increasing message costs due to consistency messages as clients are added. These two factors account for the thrashing behavior seen in Figure 22. Again, in this workload, the caching of write locks causes CB-All send more messages than CB-Read. C2PL performs best at 25 clients because at that point it sends fewer messages than the other algorithms. The slight downturn in performance seen for the C2PL algorithm at 20 clients and beyond is due to data contention.

Figure 23 shows the throughput of the algorithms when the hot write probability is increased to 10%. Here, the trends seen in the 5% write probability case are even more pronounced. C2PL becomes the highest performing algorithm at 15 clients and beyond, and O2PL-ND performs at a much lower level than the other algorithms. C2PL sends fewer messages per transaction than the other algorithms at fifteen clients and beyond in this case because its message requirements remain constant as clients are added to

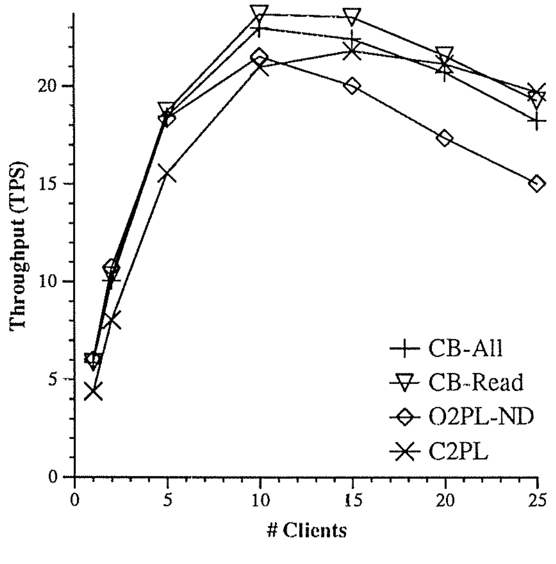

Figure 22: Throughput
(5% HICON, 25% Cli Buffers, Fast Net)

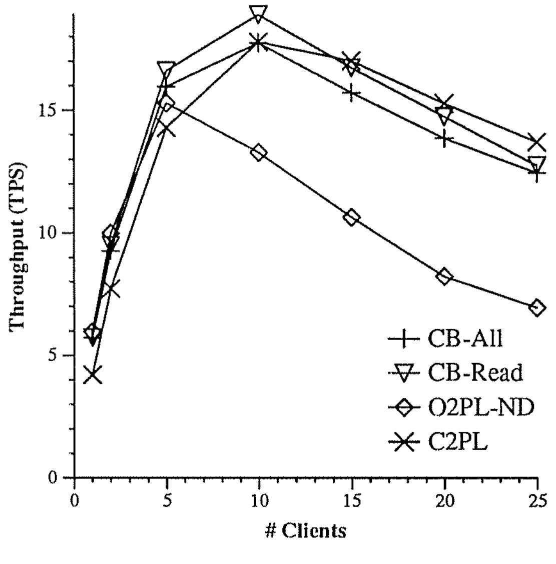

Figure 23: Throughput
(10% HICON, 25% Cli Buffers, Fast Net)

the system. This advantage helps even more in high contention cases since it makes aborts less costly. An increase in data contention causes all of the algorithms to exhibit thrashing behavior beyond 10 clients. In this case, the thrashing is due not only to additional blocking, but also to a significant number of aborts. For example, at 20 clients, O2PL-ND performs nearly 0.6 aborts per committed transaction, while the other algorithms perform about 0.3 aborts per commit. In all of the HICON experiments, O2PL-ND was found to have a significantly higher abort rate than the other three algorithms. This is due to the fact that it resolves write-write conflicts using aborts, whereas the other three algorithms resolve them using blocking. The trends seen in these two cases become more pronounced as the hot write probability is increased beyond the two cases shown here. An interesting effect that appears with higher write probabilities (e.g., HICON with write probabilities of 25% and 50%) is that in some cases, O2PL-ND actually has fewer blocked transactions than the other algorithms. This effect arises because of its higher abort rate, which at high contention levels acts as a throttle on the blocking level. However, the net result is that O2PL-ND performs worse relative to the other algorithms as the hot write probability is increased.

4.2.6. Summary

The results described in the previous sections show that the CB algorithms have slightly lower performance than the O2PL-ND algorithm in situations where network usage plays a large factor in determining performance. This is due to the additional message requirements that the CB algorithms incur because they perform consistency maintenance on a per-page basis. In situations where disk I/O was the dominant factor, they performed at a level similar to that of O2PL-ND since they are invalidation-based. The caching of write locks usually caused a net increase in message traffic, though exceptions arose in cases with few clients or no data contention. The CB algorithms performed better than the non-lock caching C2PL under most workloads, while retaining the lower (compared to O2PL-ND) abort rate of that algorithm. This is due to the fact that the CB algorithms are able to cache locks across transactions but do not have an optimistic component. In the high contention cases, fewer aborts and the ability to perform deadlock detection locally at the server allowed the CB algorithms to be much more robust in the presence of data contention than the O2PL-ND algorithm. C2PL was found to be the best algorithm for high data contention because it has a low abort rate and fast deadlock detection (similar to CB) together with message requirements that remain constant as client sites are added to the system.

5. RELATED WORK

In this section we briefly discuss work related to the specific issues that were addressed in this study. Work related to client-server caching in general is covered in [Care91a].

5.1. Callback Locking

As mentioned earlier, a callback locking algorithm is used to provide cache consistency in the ObjectStore OODBMS and is described briefly in [Lamb91]. A callback locking algorithm for client-server database systems was studied in [Wang91]. That study compared the callback algorithm with a caching 2PL algorithm (similar to C2PL) and two variants of a "no-wait" locking algorithm that allowed clients to access cached objects before they received a lock response from the server. The results of that study showed that if network delay was taken into account, callback locking was the best among the algorithms studied when locality was high or data contention was low, with caching 2PL being better in high conflict situations. These results mostly agree with those presented in Section 4.2. However, [Wang91] did not investigate an algorithm that was comparable to O2PL-ND and did not study the implications of caching write locks. Also, as mentioned earlier, the [Wang91] callback algorithm did not use accurate information for deadlock detection, and would be susceptible to phantom deadlocks in situations with large client buffer pools. The workload model used in [Wang91] provided an interesting notion of locality, one that was more dynamic than that of this study; however, it did not provide a way of controlling the nature of data sharing among clients.

A study that is closely related to client-server caching is the work at Harvard on using Distributed Shared Virtual Memory (DSM) [Bell90] to support database systems. DSM is a technique that implements the abstraction of a system-wide single-level store in a distributed system [Li89]. Three algorithms for maintaining cache consistency were studied in [Bell90], one of which was a 2PL variant that used callbacks. In a DSM system, there is no central server to manage copy information, so each page is statically assigned a "primary" site whose job it is to track the most recent "owner" of the page. The owner site changes dynamically; a site becomes the owner of a page when it obtains a write lock on the page. The owner's write lock may be later downgraded to a read lock as the result of a callback. The callback algorithm was compared to several broadcast-based algorithms using a simulation model that had an inexpensive broadcast facility. Given this facility, the callback-style algorithm typically had lower performance than one of the broadcast algorithms.

More recently, a group at IBM Yorktown has investigated several related algorithms for the shared-disk environment [Dan92]. These algorithms included shared-disk equivalents of the C2PL, CB-Read and CB-Write algorithms. In that study, the performance differences seen among the algorithms were largely due to disk I/Os needed to force pages to stable storage for recovery purposes prior to transferring a page from one node to the next. This is a different problem than what is encountered in a page-server system, since such systems have a central server that is responsible for managing a log, and in the algorithms covered here, all page transfers go through that server.

As mentioned earlier, callback algorithms were initially developed for use in distributed file systems. However, these systems have different correctness criteria and workload characteristics than database systems. The Andrew File System [Howa88] uses a callback scheme to inform sites of pending modifications to files that they have cached. This scheme does not guarantee consistent updates, however. Files that must be kept consistent, such as directories, are handled by simply not allowing them to be updated at cached sites. The Sprite operating system [Nels88] provides consistent updates, but it does so by disallowing caching for files that are open for write access. This is done using a callback mechanism that informs sites that a file is no longer cachable.

5.2. Dynamic Algorithms

We are unaware of any other work investigating dynamic algorithms for choosing among propagation and invalidation to maintain cache consistency in a client-server DBMS environment. Similar problems have been addressed in work on NUMA (Non-Uniform Memory Access) architectures, however. In such systems, an access to a data object that is located in a remote memory can result in migrating the data object to the requesting site (invalidation), replicating the object (propagation), or simply accessing the object remotely using the hardware support of a shared-memory multiprocessor. These issues are addressed in MUNIN [Cart91] by defining various types of sharing properties and allowing the programmer to annotate data declarations with a designation of the type of sharing used for each variable. These annotations serve as hints to MUNIN in deciding among propagation and invalidation, making replication decisions, etc. The DUnX system [LaRo91] uses a parameterized policy which chooses dynamically among page replacement options based on reference histories. The parameters allow a user to adjust the level of dynamism of the algorithms. A performance study in [LaRo91] demonstrated that it was possible to find a set of default parameter settings that provided good performance over a range of NUMA workloads.

6. CONCLUSIONS

In this paper, we have presented several extensions to the earlier client-server caching study of [Care91a]. These are: a better heuristic for the dynamic O2PL algorithm, an investigation of callback locking algorithms, a re-examination of system resource parameters, and a workload with high data contention.

The new heuristic was shown to perform as well as the static invalidation-based O2PL algorithm in cases where invalidation is the correct approach, which is something that the previous heuristic was unable to do. In addition, it retains the performance advantages of the previous heuristic in cases where propagation is advantageous. The heuristic uses a fixed window size parameter, but it was found to be fairly insensitive to the exact size of the window as long as the window size was kept small in proportion to the database size. The advantages of the new heuristic were more significant than were seen when the parameters of [Care91a] were used, as the combination of faster CPUs and a slower network increased the negative effects of bad propagations.

Two variants of callback locking were studied. CB-Read, the variant that caches only read locks, was found to perform as well as or better than CB-All, which caches both read and write locks, in most situations. This was because the caching of write locks was found to cause a net increase in messages except with small client populations or minimal data contention. Both CB algorithms were seen to have slightly lower performance than the O2PL-ND algorithm in situations where network usage plays a large factor in determining performance but their performance was similar to that of the O2PL-ND algorithm in cases where disk I/O was the dominant factor. The CB algorithms were seen to have a lower abort rate than the O2PL-ND algorithm, and were much more robust than O2PL-ND in the presence of data contention. C2PL was found to be the best algorithm for high data contention workloads.

A number of issues remain to be addressed in the area of cache consistency algorithms. In particular, mixed workloads and workloads with dynamic properties should be studied to better demonstrate the effectiveness of the adaptive algorithms. In addition, it should be possible to devise adaptive versions of the callback algorithms that perform better in situations where propagation is appropriate. It should also be possible to devise consistency maintenance algorithms that are adaptive in terms of lock caching as well as data caching. Finally, we are currently investigating ways of more fully exploiting the memory and processing power of the workstations in a client-server database system. A first step in this direction is described in [Fran92b].

REFERENCES

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

[Care91a] Carey, M., Franklin, M., Livny, M., and Shekita, E., "Data Caching Tradeoffs in Client-Server DBMS Architectures", *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Denver, CO, June 1991.

[Care91b] Carey, M. and Livny, M., "Conflict Detection Tradeoffs for Replicated Data", *ACM Transactions on Database Systems*, Vol. 16, No.4, December, 1991.

[Cart91] Carter, J., Bennett, J., and Zwaenepoel, W., "Implementation and Performance of Munin", *Proc. 13th ACM Symposium on Operating System Principles*, Pacific Grove, CA, Oct. 1991.

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

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

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

[Fran92a] Franklin, M., "Architectural Issues in Client-Server Database Systems", *Ph.D. Dissertation Proposal*, University of Wisconsin-Madison, January, 1992.

[Fran92b] Franklin, M., Carey, M., Livny, M., "Global Memory Management in Client-Server DBMS Architectures", to appear *Proc. 18th VLDB Conference*, Vancouver, Canada, August, 1992.

[Fran92c] Franklin, M., Zwilling, M., Tan, C., Carey, M., DeWitt, D., "Crash Recovery in Client-Server EXODUS", to appear *Proc. SIGMOD Int'l Conference on the Management of Data*, San Diego, CA, June, 1992.

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

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

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

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

[Li89] Li, K., Hudak, P., "Memory Coherence in Shared Virtual Memory Systems", *ACM Transactions on Computer Systems*, Vol., 7, No. 4, Nov. 1989.

[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 Transactions on Computer Systems* 6, 1, Feb. 1988.

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

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

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