Maintaining Consistency of Client-Cached Data

Kevin Wilkinson and Marie-Anne Neimat
Hewlett-Packard Laboratories, 1501 Page Mill Rd., Palo Alto, CA 94304

Abstract

This paper addresses the problem of cache consistency in a client-server database environment. We assume the server provides shared database access for multiple client workstations and that client workstations may cache a portion of the database. Our primary goal is to investigate techniques to maintain the consistency of the client cache and to improve server throughput. We propose a new cache consistency algorithm for client caches. The algorithm is a simple extension to two-phase locking and consists of three additional lock modes that must be supported by the server lock manager. For comparison, we devised a second cache consistency algorithm based on notify locks. A simulation model was developed to analyze the performance of the server under the two cache consistency algorithms and under non-caching two-phase locking. The results show that both consistency algorithms can significantly improve server performance over basic two-phase locking. The notify locks algorithm, at times, out-performs the cache locks algorithm. But, it is very sensitive to data contention and server load. Cache locks is always better than two-phase locking and is much more stable than notify locks under all conditions.

1 Introduction

This work was motivated by recent research in database servers and active databases. The use of database servers is emerging as a common paradigm to provide shared data access over computer networks. Typically, the application program runs as a client process and communicates with the database server through messages. This increases the cost of each data request. One solution is to reduce the number of requests by caching a portion of the database on the client. When a client cache is used, there must be a protocol between the client and server to ensure that the client cache remains consistent with the shared database. In this sense, the client cache may be viewed as active data since updates should trigger a cache-refresh operation.

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

Active databases allow applications to be informed of changes to some portion of a shared database by other transactions. In practice, this has meant that all updates to the database must be monitored by the database management system to determine if the updates affect the *active* data [4, 12]. When the active data is updated, the database system must inform the affected clients that a change has occurred. Thus, all transactions incur additional overhead to support a service that they may never use (i.e. detection and notification of updates). The challenge is to make this overhead as low as possible since this translates directly into lower response time and higher throughput. This is especially important when the database is on a central server, and therefore, a potential bottleneck.

In our view, the "active database problem" may be treated as a special case of concurrency control synchronization. A transaction with active data may be considered to have a lock on the active data. Other transactions may update the active data which, in effect, breaks the lock. The problem, then, is how to inform the lock holder of the update.

Our approach to maintaining cache consistency is to integrate the cache consistency algorithm with the lock manager of the database management system. There are several advantages in doing this. First, the impact on the database management system is minimized in that no new modules need be written or incorporated. Second, the path length for request processing is not significantly increased since the lock manager is already called for most requests. Third, it is transparent to non-caching transactions and requires few changes to caching transactions.

We present two algorithms. The first is an extension to two-phase locking and consists of adding new lock modes (*cache locks*) to the lock manager. The second algorithm is based on notify locks.

Through simulations, we compare the relative performance of the two cache consistency algorithms and contrast it with non-caching two-phase locking. Our simulation model was derived from the model developed in [1]. We extended that model to a distributed environment with one server and multiple clients and with caching on the clients.

This work differs from previous research in cooperating transactions [5] and active databases with real time requirements [12, 13]. Our primary goal is to support client caching and improve server performance. Thus, we are willing to sacrifice some capabilities of active databases such as immediate notification of updates.

The notion of materialized views (or snapshots) is loosely-related to our work in that a snapshot may be considered a form of cached data. The emphasis of this work has been on efficient mechanisms for refreshing the snapshot after an update to the base data. A refresh algorithm for views over a single base table was described in [9]. To detect view updates, it requires base tables to be augmented with a timestamp and link field for each tuple. An algorithm for arbitrary select-project-join views was described in [2]. Each update transaction, as a side-effect, builds additional sets of tuples to add or remove from each affected view.

A system that supports read-only caching for workstations accessing a centralized database is described in [6]. However it requires some changes to the database management system to timestamp base data. Also, it does not support ad-hoc queries or ad-hoc updates.

The next section describes our cache consistency algorithm based on cache locks. Section 3 contains a description of notify locks as we have adapted them to maintain the consistency of cached data. The two algorithms are compared in Section 4. Section 5 describes the queueing model we have used for our performance studies. Our simulation experiments and their results are described in Section 6. Section 7 contains a summary of the results of our study. Finally, we note that this work is being done as part of the Papyrus project [11]. This is a larger effort to construct cooperative, high-performance data servers.

2 Cache Locks

Consider a client process in which a portion of a database has been cached to reduce access time to a shared database server. We assume that the database server uses two-phase locking as its concurrency control algorithm. We also assume that client transactions have a read-phase and a commit phase and updates are not sent to the server until the commit phase.

We observe that a cached object is merely a snapshot of an object on the server. A cached object may be in one of four states relative to the *current* version on the server. It may be in the process of being copied from the server into the client cache, identical to the current version on the server, out-of-date due to an update by an uncommitted transaction, or out-of-date due to an update by a committed transaction.

The idea behind the cache locks algorithm is to use lock modes in the lock manager to correspond to the states of a cached object. Every object in a client cache must have a lock on the server. The first state (being copied) can be modeled with a conventional share lock. For the remaining three states, we add three new lock modes to the lock manager. A client can easily detect if a cached object is out-of-date by checking the mode of its corresponding lock on the server. If the cached data is no longer current, transactions using that data may not commit.

In conventional two-phase locking, the locks of a transaction are discarded when the transaction terminates. However, a client database cache should survive across transactions in order to benefit multiple transactions (so they avoid the cache-load overhead). To support this, we must permit cache locks to survive across transactions so that updates to cached data can be tracked even when they occur outside the boundaries of client transactions. For this purpose, we introduce a new type of transaction, the ***envelope transaction***. The envelope transaction is a long-term, read-only transaction that is created on behalf of the client's cache manager and that has slightly different semantics from normal transactions as described, below. It begins when the cache is created, and is responsible for loading objects from the server into the cache and for maintaining cache consistency. Note that to simplify the discussion, we assume there is no more than one active transaction using a client cache at any time. There is no difficulty in supporting simultaneous caching transactions on a client but it requires concurrency control on the client cache which is beyond the scope of this paper.

In order to load an object from the server into its local cache, a client's envelope transaction sets a conventional S lock (*Share* lock) on the object for the duration of the load phase. This ensures that a consistent snapshot of the object is loaded into the cache. Once the object has been loaded, the envelope transaction requests the server to *demote* the S lock to a C lock (*Cache* lock). A cache lock indicates that the lock holder is caching the locked object. However, unlike share locks, C locks do not block other transactions. A request from another transaction for an X lock (*Exclusive* lock) will break a C lock and, as a side-effect, change the C lock to a P lock (*Pending update* lock). If the updating transaction aborts, the P lock is converted back to a C lock. If the updating transaction commits, the P lock is converted to an O lock (*Out-of-date* lock). The lock compatibility matrix appears in Figure 1.

T_j*(lock)* / T_i*(request)*	S	X	C	P	O
S	√	*No*	√	√	√
X	*No*	*No*	√	√	√

Figure 1: *Lock Conflict Matrix*

Envelope transactions typically hold only C, P and O locks. Their S locks are only held for a short duration. Thus, the presence of envelope transactions does not reduce the throughput of the system since their locks do not block S and X locks.

User transactions that run on a caching client interact with the envelope transaction in the following way. For a non-cached object, user transactions request S or X locks from the server, as before. For a cached object, the user transaction must still acquire an S or X lock from the server. However, rather than acquiring a new lock, the user transaction merely inherits the lock from the envelope transaction and later upgrades the lock to an S or X lock.

We assume that the client will not attempt to cache objects that are database hotspots.[1] In order to reduce message traffic, we use an optimistic algorithm that delays acquiring locks for cached objects until the transaction tries to commit. Under this scheme, caching clients keep track of their transactions' read/write set. To access a cached object, the user transaction merely copies the object from the cache. When the user transaction is ready to commit, a commit request is sent to the server along with the read/write set. At this point, locks must be acquired for objects that were accessed from the cache. For each such object, the user transaction first inherits the lock mode of its corresponding envelope transaction. Then the server attempts to upgrade the lock to an S or X lock.

The lock upgrade matrix appears in Figure 2. P locks were introduced to take advantage of the repeatability of reads from the cache. User transactions with P locks in their read set may commit without violating serializability. On the other hand, transactions with P locks in their write set must block until the P lock is converted to a C or O lock. Attempts to upgrade O locks cause the user transaction to abort.

T_j(*lock*) T_i(*upgrade*)	S	X	C	P	O
S	S	X	S	P	*No*
X	X	X	X	*Block*	*No*

Figure 2: *Lock Upgrade Matrix*

Once the user transaction terminates, all S and X locks that it held on cached data are dropped but the corresponding C locks remain held by the envelope transaction. The lock inheritance introduced here is different from that of nested transactions [10]. A caching client may be viewed as having a parent transaction that runs forever. Child transactions inherit the locks of the parent but they are allowed to commit even though the parent will never commit. Envelope transactions impose little overhead on a lock manager. They require the bookkeeping of the C, P, and O locks, but do not necessitate any automatic message sending on the part of the server.

Optimization For high conflict rates, the algorithm will have a high number of restarts if conflicts on cached objects are not detected until the commit point. We can reduce the number of restarts if the cache manager can discover its broken C locks before they are accessed. One solution is to send a notification message whenever a lock is broken. This is basically the notify algorithm described in Section 3. Another solution is to add a new lock manager call that returns a list of all O locks held by a transaction. So, we assume that every request by a caching transaction includes an additional request to check for O locks for the corresponding envelope transaction. The envelope transaction may mark out-of-date objects in its cache or refresh the cache by immediately re-reading the objects. The choice depends on whether one is optimizing response time or throughput. In our studies, we are optimizing for throughput so we chose to mark the objects in the cache and reload them upon demand.

Another potential optimization is to piggyback lock requests for previously accessed cached objects whenever a client calls the server. However, our purpose was to compare different algorithms rather than devise one completely optimal algorithm. This optimization would have made both the cache locks and notify locks algorithms very similar to two-phase locking. Thus, we leave it for future study.

We now summarize the cache locks algorithm for both the client and server.

Client Cache Locks Algorithm

- Whenever a user transaction requests an out-of-date item from the cache, send a refresh request to the server for that item.
- Whenever a user transaction tries to commit, send a commit request to the server. Accompany the message with the identifiers of objects in the transaction's read/write set and with the new values of objects it updated.
- Whenever the server accompanies a response with a list of identifiers for cached objects with O locks, mark these objects as out-of-date, and, if one of these objects is in the read/write set of an active transaction, abort the transaction.

Server Cache Locks Algorithm

- Whenever a commit request is received from a caching client, do the following:
    1. For each object in the transaction's read set, inherit the corresponding lock of the envelope transaction and upgrade it to an S lock. The upgrade from a C lock is immediately granted, the upgrade from a P lock is postponed until step 3, and the upgrade from an O lock is denied causing the requesting transaction to abort. If the transaction is aborted, go to step 4.
    2. For each object in the transaction's write set, inherit the corresponding lock of the envelope transaction and upgrade it to an X lock. The upgrade from a C lock is immediately granted, the upgrade from a P lock is blocked until the P lock turns into a C or O lock, and the upgrade from an O lock is denied causing the

[1] Our algorithm will work for cached hotspots but with reduced performance. Cached hotspots are best handled by acquiring an explicit lock as soon as they are accessed.

requesting transaction to abort. If the transaction is aborted, go to step 4. If an X lock is acquired, convert C locks of other transactions to P locks and post the update to the server's database.

3. Recheck the status of read objects with P locks. The transaction can commit only if none of its P locks has been converted to an O lock.
4. Release the locks held by the transaction. Releasing X locks has the side effect of turning other transactions' P locks into C or O locks depending on whether the transaction aborts or commits.
5. Send a response to the client indicating whether the transaction has committed or aborted. Accompany the response with the identifiers of objects whose C locks have turned into O locks.

- Whenever a request to refresh an item is received from a client, drop the O lock held by the client's envelope transaction, if any, acquire a C lock on the object on behalf of the envelope transaction, an S lock on behalf of the user transaction, and send a copy of the object to the client. Note that it is not necessary to immediately acquire the S lock for the user transaction; it may wait until the commit phase. It is acquired at this point as a small optimization since the user transaction needs the S lock eventually. Accompany the response with the identifiers of objects whose C locks have turned into O locks.

3 Notify Locks

The concept of *notify locks* was used in ObServer [7] to allow a group of clients to share uncommitted results. Holders of notify locks could specify whether they wanted to be informed of reads or writes of an object. The server would send them a notification message whenever such an action was applied to the object. Although notify locks were designed to support cooperating transactions, the concept behind them can be used to keep client caches up-to-date. We adapted notify locks to fit a client cache model. Applying the same reasoning we used in designing the cache locks algorithm, we designed an algorithm based on notify locks that generates as few messages as possible.

Notifications of updates can be sent to clients at various times. For this study, we have chosen to send them when the updates are actually committed. This represents the minimum number of messages sent while still providing timely information to prevent transactions destined for an abort from performing useless work. Furthermore, by only taking into account committed updates, the client can take advantage of the repeatability of reads from its cache.

When the client receives a notification from the server, it checks the list of updated objects against the read/write sets of its active transactions. Any transaction with an updated object in its read/write set must abort. When the server receives a request for commit from a client, it assumes that the client has enforced consistency correctly. Because notification messages are sent asynchronously, there is a window of vulnerability caused by the delay in message transfers. For example, suppose a client decides that a transaction can commit and sends a commit request to the server. Until the server receives that commit request, it may still be sending notification messages to the client that could abort the transaction. To protect against this exposure, a handshake is required between the client and server to ensure that the client has seen the most recent notification message. We chose to implement this handshake by assigning a sequence number to every message sent from the server to a client. The client, in turn, accompanies requests to the server with the most recent message sequence number that it has seen. If a commit request contains a sequence number that is too low, the server rejects the request, and asks the client to verify if the transaction should be committed and then resend the request.

As with the cache locks algorithm, clients send their updates to the server with their commit requests. If a commit request is accepted, the server applies the updates to the database. The server then sends notifications to all the clients whose caches have been invalidated by the committing transaction. One can observe that when the server is ready to send the notifications, most of the updated data is in the main memory of the server since the updates were just posted to the database. Thus, to reduce overall I/Os and message passing, we assume the server sends to each affected client, the new values of the updated objects. In this way, clients' caches are automatically refreshed with each notification message, and explicit refresh requests are never required by the clients.

One can notice another period of exposure to multiple concurrent updates. It takes place after a transaction has been accepted for commit by the server, while the deferred updates are being applied to the central database. Other transactions can invalidate the read/write set of the committing transaction during this period. For this reason, another check of message sequence numbers is required after all deferred updates have been posted.

We now summarize the notify locks algorithm for cached data.

Client Notify Locks Algorithm

- Whenever a user transaction tries to commit, send a commit request to the server. Accompany the message with the new values of objects it updated and with the last message sequence number received from the server.

- Whenever a notify message is received from the server, refresh the cache with the new values sent by the server, remember the message sequence number sent by the server, and if any of the updated objects is in the read/write set of an active transaction, abort the transaction.

- Whenever a message is received from the server to verify if a commit request with a low sequence number should still commit, check if the transaction was aborted as the result of a notify message. Respond with a message to the server to abort the transaction or proceed with the commit.

Server Notify Locks Algorithm

- Whenever a commit request is received from a caching client, do the following:

  1. Verify that the received sequence number is equal to the last sequence number sent to the client. If not, ask the client to verify if the transaction should still commit. If the transaction must abort, go to step 4.
  2. For each object in the write set of the transaction, acquire an X lock, and post the deferred update to the central database. The X lock is a standard two-phase locking X lock and is subject to the usual blocking and deadlocks. If the transaction is forced to abort, go to step 4.
  3. Again, compare the received sequence number with the last sequence number sent to the client. If the client's number has fallen behind the server's number, ask the client to verify if the transaction should still commit. (Note that the server will not release the X locks held by the transaction while the verification takes place. This will allow the transaction to commit rapidly once it returns to the server.)
  4. Release the locks held by the transaction and send a response to the client indicating whether the transaction has committed or aborted.
  5. If the transaction has committed, go over its write set, and for each client with one or more cached objects in the write set, send a message notifying it of updates to its cache. Accompany the message with the new object values and with a newly generated message sequence number.

4 Comparison and Discussion

The cache locks algorithm and the notify locks algorithm implement different concurrency control policies for their cache hits and cache misses. Whenever they access data that is not in their cache or that is out-of-date in the cache, both algorithms implement two-phase locking. For cache hits, both algorithms are optimistic [8] in that locks are never required for up-to-date cached objects. However, unlike a pure optimistic algorithm, a cache locks or notify locks transaction may abort during its read phase before it reaches its commit point. This will occur when the client is informed that an object already read or updated (locally) by the transaction has been permanently modified. In this sense, both algorithms may be termed ***semi-optimistic***. Thus, in effect our scheme supports two-phase locking and semi-optimistic transactions, simultaneously.

Boral and Gold [3] have shown a framework under which different synchronization techniques can be combined into one concurrency control algorithm. In particular, they prove the correctness of an algorithm that combines two-phase locking with optimistic concurrency control. Using their framework, it is straightforward to prove the correctness of the combined two-phase locking / cache locks algorithm and the two-phase locking / notify locks algorithm.

Although the cache locks algorithm is semi-optimistic for cache hits, under periods of high data contention or for database hotspots, it behaves more like a two-phase locking algorithm. Under these conditions, the cache will be usually out-of-date. This causes the effective cache size to shrink by reducing the number of cache hits so that the cache-locks algorithm becomes more like two-phase locking. Unlike the cache locks algorithm, the cache for the notify locks algorithm is always up-to-date. Thus, it makes more effective use of its cache than the cache locks algorithm.

It may seem unfair to compare the two algorithms since they use their cache so differently. But, there are many possible variations on these algorithms and we could not explore the entire space. Rather, we tried to incorporate what we thought would be reasonable optimizations in a real implementation. The comparison between the resulting two algorithms is interesting because both algorithms provide client cache consistency but make different trade-offs with respect to communication and server load.

There are some potential problems with integrating cache consistency with concurrency control. First, the server is required to maintain many more locks for caching clients than for non-caching clients. In principle, this is not a problem. However, a lock manager may be implemented with fixed limits on the number or duration of locks that may be held. The number of locks may be reduced by using higher granularity locks.

A second problem is that, in some systems, locks are transparent to higher levels of software (e.g. if locks are implicitly obtained as a side-effect of a storage manager request). Both the cache locks and notify locks algorithms require logical (object) locks. Thus, a translation may be necessary from lock manager identifiers to client cache manager identifiers.

5 Simulation Model

Our goal in the performance analysis was to compare the effect of the two cache consistency algorithms on the performance of the database server. For a baseline comparison, we also studied the server performance under non-caching two-phase locking. For the analysis, we developed a closed queueing simulation model. Our model is heavily based on the model in [1] but it was extended for a client-server environment and for client caching. It consists of a single database server and a fixed number of clients. Each client has one active terminal so the number of clients, *num_clients*, also represents the number of terminals. The logical queueing model is illustrated in Figure 3. To simplify the figure, only one client is illustrated.

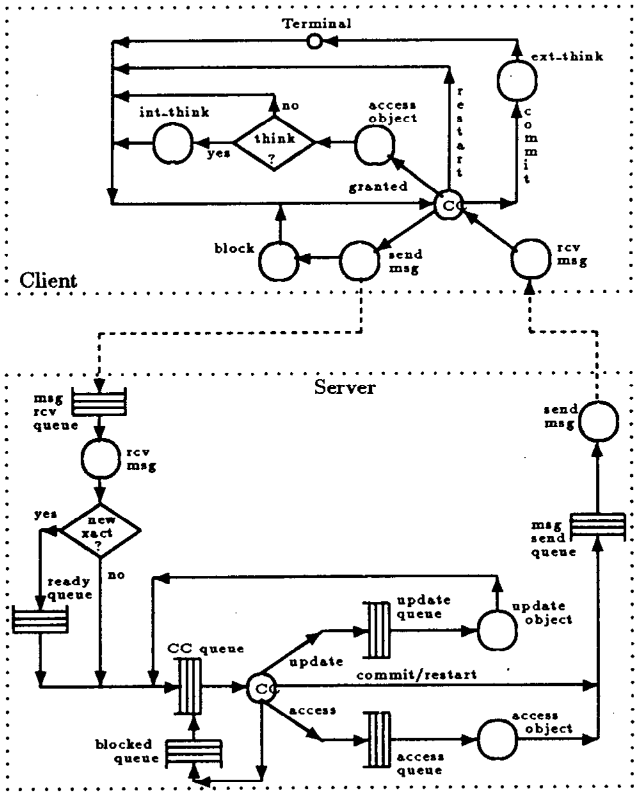

Figure 3: *Logical Queueing Model*

We assumed infinite resources for each client and finite resources for the server. Some clients are designated as caching clients and others are designated as non-caching. The number of caching clients, *ccl*, is a simulation parameter.

Transactions originate on the clients. As mentioned earlier, transactions have a read phase and a commit phase and writes are deferred until the commit phase. The write set is a subset of the read set (i.e. an object must be read before being written). The external thinking delay on a client after committing a transaction is assumed to have an exponential distribution with mean *ext_think_time*.

The number of objects accessed by a transaction was drawn from a uniform distribution between *max_size* and *min_size* and with mean *tran_size*. The probability that an object in the read set was updated was a constant, *write_prob*. The number of objects in the database, *db_size*, was chosen to be relatively small in order to model contention.

For caching clients, the cache size, *cache_size*, was a constant. The content of each client's cache is fixed at the start of the simulation by choosing objects uniformly from the database. *Cache_hit_prob* is the probability that an item in the read set of a transaction will be directed towards the client's cache. Note that for the cache locks algorithm, this does not guarantee a cache hit since the cached object may be out-of-date necessitating a server call to refresh the cache. As in [1], objects in the write set of a transaction were uniformly chosen from the entire database.

The client read phase proceeds as follows. For every object in the read set, the client concurrency control manager is called. The client concurrency controller implements two-phase locking for non-caching clients and the *client cache locks algorithm* or the *client notify locks algorithm* for caching clients. For cache misses (and for non-caching clients), the client concurrency controller reads the object from the server. For the commit phase, the client concurrency control manager sends the read/write sets and a commit request to the server. To model interactive transactions, there was an internal thinking delay between the read and commit phases that was exponentially distributed with mean *int_think_time*.

For a read request, the outcome of a client concurrency control request will be that the request was granted or that the transaction was aborted and must be restarted. If the request is granted, the client access time for the object was simulated with a delay of *obj_client_cpu*. No I/O time is ever incurred on the client (message processing is discussed later). If restarted, the transaction is rerun with the same read/write set. The outcome of a commit request is either commit or restart.

On the server side, the multiprogramming level, *mpl*, is used to limit the number of active transactions. A transaction joins the "ready queue" on its first request to the server and waits until it is allowed to become active. Subsequent requests by that transaction bypass the "ready queue." A transaction is considered active until it commits, i.e. a restarted transaction does not wait again in the ready queue.

Requests made by the cache locks algorithm include the envelope transaction identifier and the user transaction identifier. The user transaction identifier is used to determine if the transaction is active on the server. The server concurrency control manager implements two-phase locking for non-caching transactions and the *server cache consistency algorithm* or the *server notify locks algorithm* for caching transactions.

For read requests, the outcome of the server concurrency controller will be either to grant access, to block (waiting for a lock) or to abort and restart the transaction. On the server, objects are read in *obj_io* time followed by *obj_server_cpu* time.

Deferred updates are performed as part of a commit request. The server concurrency controller is called for each object in the write set. The outcome will either be to grant the update request, to block or to restart the transaction. If all updates succeed, the transaction is committed.

Objects are written in *obj_server_cpu* time followed by *obj_io* time. Although not shown in Figure 3, the notify locks algorithm has the side effect of sending, on a successful commit, messages to all the clients whose caches have been invalidated by the committing transaction.

Resource contention on the server takes place over the CPUs and disks. As in [1], the resource usage times are constants rather than stochastic values. The number of CPU and I/O servers, *num_cpus* and *num_disks*, were fixed. CPU requests are serviced first-come, first-served by any available CPU. I/O requests are randomly assigned to one of the disks where they are also serviced in a first-come, first-served manner.

Sending and receiving messages imposes an overhead of *msg_cpu* on the CPU. Message requests are given higher priority than other CPU requests. In our experiments, we have not explicitly accounted for network delays. Instead we have bundled them in the *msg_cpu* time. We have assumed that contention on the network interface is small relative to the CPU contention.

Table 1 summarizes the fixed simulation parameters. Table 2 displays the values of the experimental simulation parameters.

Parameter	Meaning	Value
db_size	number of database objects	1000
tran_size	mean number of objects accessed per trans.	8
max_size	largest trans. size	12
min_size	smallest trans. size	4
write_prob	Pr (write X | read X)	0.25
num_clients	number of clients	200
ext_think_time	mean time between trans.	1 s
obj_io	I/O time per object	35 ms
obj_client_cpu	client CPU time per object	10 ms
obj_server_cpu	server CPU time per object	5 ms
msg_cpu	CPU time to send or receive a message	5 ms
num_cpus	number of CPUs on server	1
num_disks	number of disks on server	2

Table 1: *Fixed Simulation Parameters*

We emphasize several aspects of the simulation model. First, as in [1], the multiprogramming level effectively controls the conflict rate (albeit, indirectly). Second, even though in a real system, the request and response message sizes might differ by an order of magnitude, we chose to make the message transmission time constant. This is a reasonable approximation because the communication time is typically dominated by a high fixed overhead. A third point is that we do not model an LRU cache but, instead, fix the contents of each client's cache. Since we are assuming a uniform reference pattern, it made no sense to implement an LRU scheme. In effect, the cache hit probability models locality of reference.

Parameter	Meaning	Value
ccl	number caching clients	0, 10, 25, 50, 100, 190, 200
cache_size	client cache size	15, 30, 45, 60
mpl	multiprogramming level	10, 25, 50, 100, 200
cache_hit_prob	Pr (object is cached)	0.1, 0.5, 1.0
int_think_time	mean internal think time per trans.	0, 1, 5, 10 s

Table 2: *Experimental Simulation Parameters*

To validate our implementation of the model, we ran a number of two-phase locking experiments with the message time set to 0 and other parameter settings equivalent to those in [1]. Our results matched the two-phase locking results in [1].

6 Experiments

We ran a number of simulations to study the behavior of the two cache consistency algorithms under different conditions and to compare their performance with non-caching two-phase locking. With the client caching level (*ccl*) fixed at 200, we ran simulations for all combinations of the other parameters. We then ran simulations that varied *ccl* for subsets of the other parameters. In order to stress the algorithms and investigate their differences under periods of high load, we used extreme values for some of the simulation parameters. Our goal was to compare the algorithms, not to model an actual system.

In this section, we describe three sets of experiments that summarize what we feel are the most interesting results. The first set of experiments studied the effect of cache size and cache hit probability on performance. In the second set of experiments, we studied the behavior of interactive transactions. Finally, the last set of experiments investigates the effect of varying the number of caching clients on overall server throughput. All simulations were run on an HP 9000/370 (MC68030-based machine) running HP-UX (HP's version of Unix) and using the CSIM simulation package [14].

6.1 Variable Cache Size and Cache Hit Probability

In the first set of experiments, we examined the effect of resource contention on server throughput (measured in transactions per second) for each algorithm. The number of caching clients was fixed at 200 and the cache size and cache hit probability were varied. The internal think time was set to 0 which causes the heaviest CPU and I/O load. Figures 4 through 9 show the effects of cache size and cache hit probability on throughput for cache sizes of 15 and 60. The two algorithms are solid - two-phase locking   dotted - cache locking   dashed - notify locking

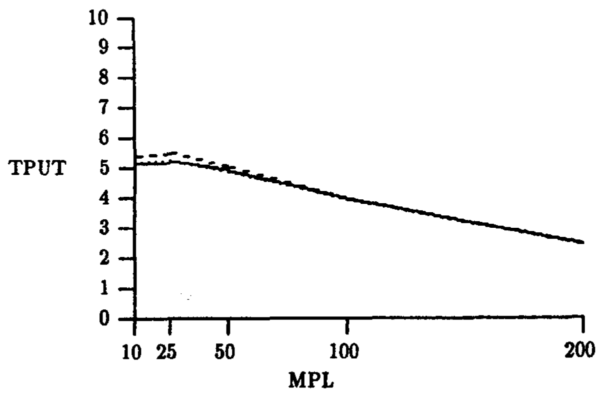

Fig 4: Thruput for Cache Size 15, Hit Prob 0.1, Int Think 0

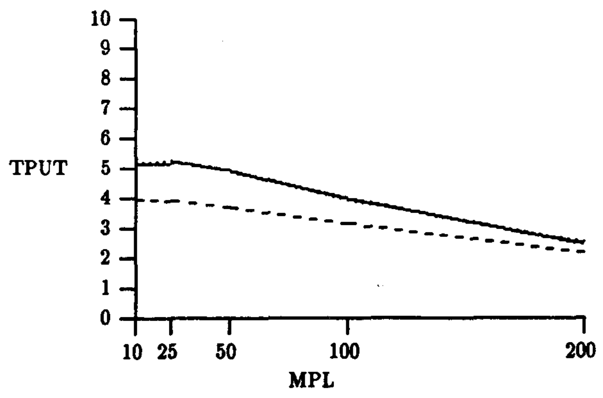

Fig 5: Thruput for Cache Size 60, Hit Prob 0.1, Int Think 0

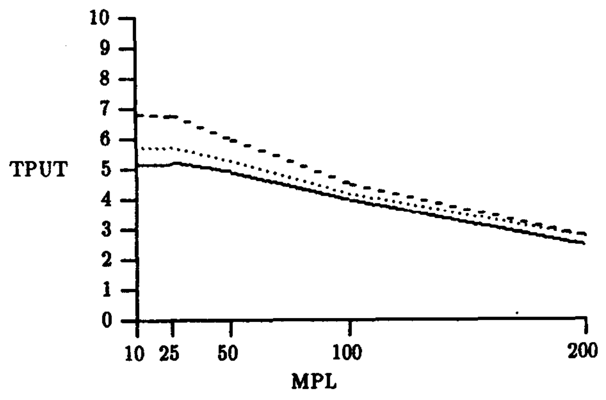

Fig 6: Thruput for Cache Size 15, Hit Prob 0.5, Int Think 0

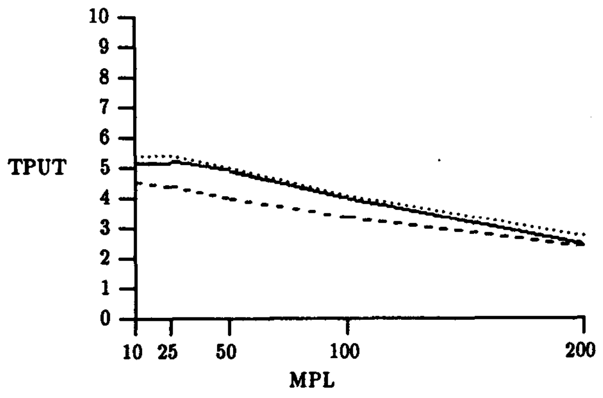

Fig 7: Thruput for Cache Size 60, Hit Prob 0.5, Int Think 0

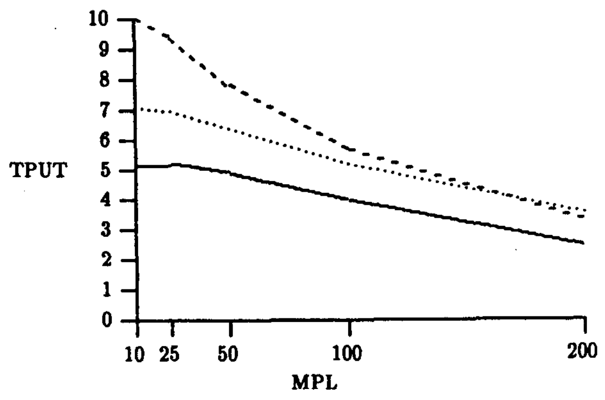

Fig 8: Thruput for Cache Size 15, Hit Prob 1.0, Int Think 0

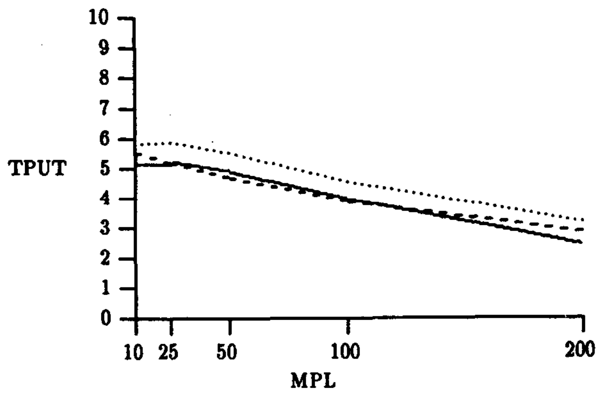

Fig 9: Thruput for Cache Size 60, Hit Prob 1.0, Int Think 0

compared to non-caching two-phase locking to evaluate the relative benefit of caching on server throughput.

With the cache size fixed at 15 objects, we observe that the performance of both consistency control algorithms improves as the cache hit probability increases. At low cache hit probability, the cache locks algorithm is practically identical to two-phase locking. A cache locks transaction must request most objects in its read set from the server. As the cache hit probability increases, the cache locks algorithm is able to take more advantage of its cache which reduces the server load and increases throughput.

The notify locks algorithms does remarkably well for low multi-programming levels. Its cache is always up-to-date, and at low multi-programming levels, it is invalidated infrequently. Under these circumstances, it utilizes its cache very efficiently. For higher levels of multi-programming, the cache is invalidated more frequently thus causing some of the notify lock transactions to abort. At high multi-programming levels, the deterioration in performance is more rapid for notify locks than for cache locks because the notify locks algorithm loses its advantage of always having the cache up-to-date.

Surprisingly, as the cache size increases from 15 objects to 60 objects, the overall benefit of both algorithms is reduced considerably. With a larger cache size, the probability that a random update will invalidate some cached object increases. Since the cache hit probabilities are not a function of the cache size, larger cache sizes dilute the benefits of caching by becoming more vulnerable to updates. The notify locks algorithm does very poorly under these circumstances. At low cache hit probabilities, it pays a high penalty in notify messages and gains very little benefit from the cache (Figure 5, Figure 7).

There is a major difference in behavior between the notify locks algorithm on one hand and the cache locks and two-phase locking algorithms on the other hand. As in [1] and for the simulation parameters we have chosen, two-phase locking and cache locks are I/O bound. With notify locks, the cache is always up-to-date. Consequently, I/O requests are submitted to the server only for cache misses and for deferred updates. This reduces disk utilization and causes the notify locks algorithm to be, in general, CPU bound. Any experimental setting that increases the number of notification messages to clients aggravates this problem and causes the performance of the notify locks algorithm to deteriorate.

Figures 10, 12 and 14 show CPU utilization, disk utilization, and average number of retries per committed transaction for all three algorithms with the cache hit probability fixed at 1.0 and the cache size fixed at 15 objects. These figures correspond to the throughput shown in Figure 8. Figures 11, 13 and 15 show the same data but for a cache size of 60 objects. They correspond to the throughput shown in Figure 9.

One can see that as the cache size increases, the severe decline in throughput for the notify locks algorithm is not due to a larger number of retries but instead to the saturation of the CPU. In fact, both the number of retries and disk utilization actually decrease as cache size increases. This is because of the increased message processing necessary for larger cache sizes. For a cache size of 60 objects, the server must send, on average, 4 times as many notify messages per update as for a cache size of 15 objects. Thus, throughput for the notify locks algorithm is dominated by the message delay for large cache sizes so fewer transactions are actually submitted.

We note that the CPU and disk utilization for the cache locks algorithm is very similar to the utilizations for two-phase locking. However, the cache locks algorithm achieves higher throughput. This is because the cache locks algorithm is able to offload I/O processing from the server which reduces the transaction response time. We also note that as the cache size increases, the average number of retries also decreases for the cache locks algorithm. This is because the number of cache hits actually decreases as cache size grows. Thus, the cache locks algorithm takes less advantage of the cache and tends to perform more like two-phase locking for large cache sizes.

In comparing the overall performance of cache locks and notify locks, one finds that notify locks can provide significant increases in throughput under some circumstances but has disastrous effects under others. In particular, it is very sensitive to the server load. The cache locks algorithm, on the other hand, does well under some circumstances and is more stable. It never does worse than two-phase locking.

6.2 Interactive Transactions

In the second set of experiments, we wanted to compare the algorithms on the basis of pure data contention. Thus, we reduced the resource bottleneck by increasing the internal think time. As in the first experiments, the number of caching clients was fixed at 200 and the cache size and cache hit probability were varied. Note that longer think times result in longer response times and reduced throughput.

For these experiments, the throughputs had interesting characteristics. Figures 16 and 17 display the throughputs for a cache size of 15 objects, cache hit probabilities of 0.5 and 1.0 and an internal think time of 5 seconds. Again, the results of non-caching two-phase locking are included for comparison.

We notice that throughput first increases and then drops off rapidly. As the multi-programming level increases, transactions do not have to wait as long in the ready queue. This improves their response time. But, as the multi-programming level increases further, the number of retries increases dramatically causing the response time to deteriorate[2]. The increase in response time is due only to contention on database objects and not to contention on the CPU and disks. This pattern was repeated for other internal think times although the magnitude of the throughput varied.

Comparing Figures 16 and 17, we notice two effects of increasing the cache hit probability. First, there is only a modest increase in throughput as cache hit probability is increased. The high think time somewhat masks

[2]Although not shown, the plot of the number of retries was similar to that shown in Figure 14

solid - two-phase locking dotted - cache locking dashed - notify locking

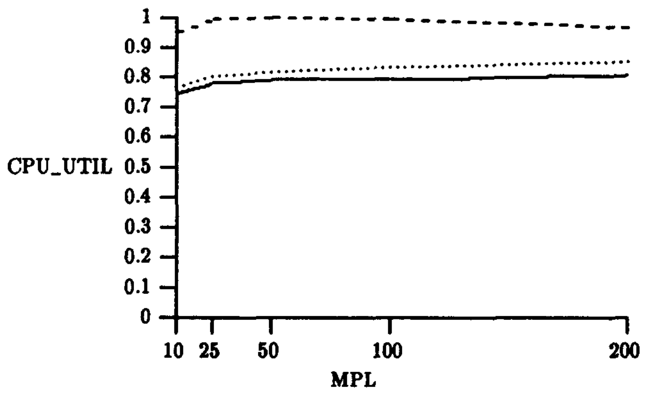

Fig 10: CPU Util for Cache Size 15, Hit Prob 1.0, Int Think 0

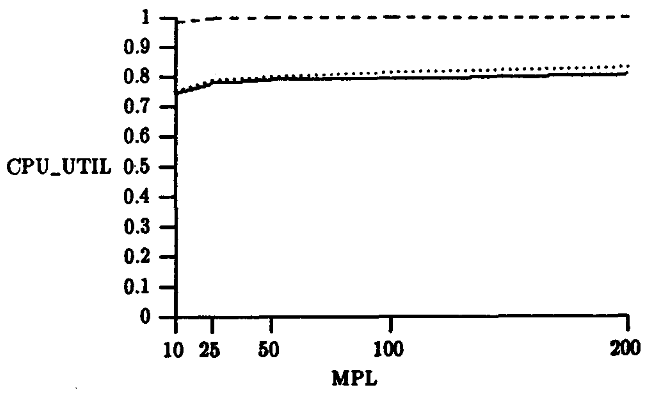

Fig 11: CPU Util for Cache Size 60, Hit Prob 1.0, Int Think 0

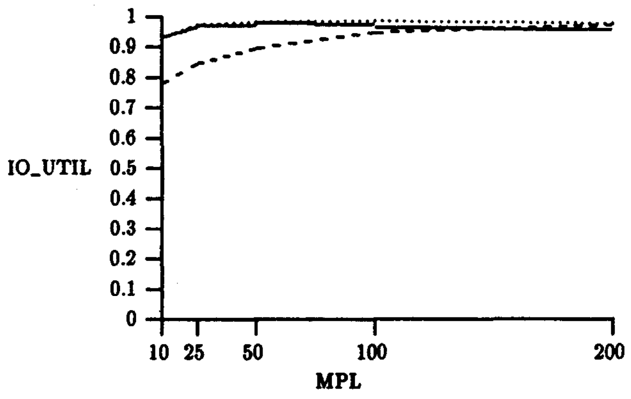

Fig 12: I/O Util for Cache Size 15, Hit Prob 1.0, Int Think 0

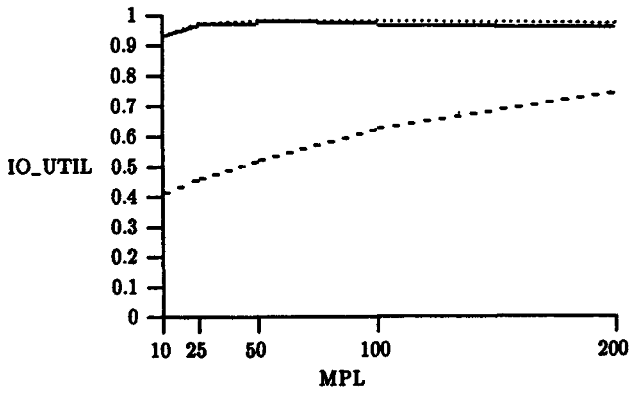

Fig 13: I/O Util for Cache Size 60, Hit Prob 1.0, Int Think 0

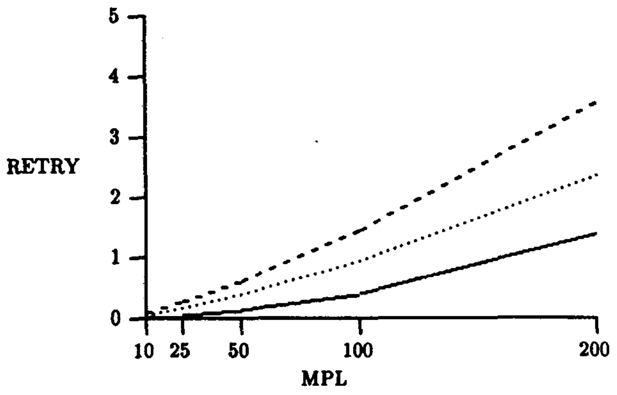

Fig 14: Retries for Cache Size 15, Hit Prob 1.0, Int Think 0

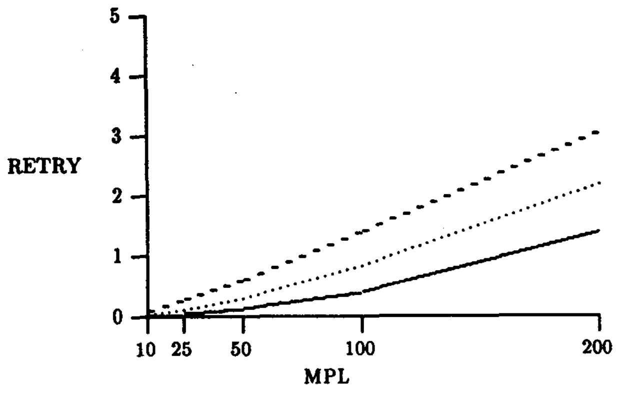

Fig 15: Retries for Cache Size 60, Hit Prob 1.0, Int Think 0

the benefit of the higher number of cache hits. This can be seen by comparison with Figures 6 and 8 which show a large increase in throughput. Second, the cross-over point at which cache locks algorithm performs better than the notify locks algorithm occurs earlier as the cache hit probability is increased. This is a consequence of the larger number of restarts suffered by the notify locks algorithm. When the cache hit probability is increased, the notify locks algorithm has more cache hits than the cache locks algorithm. This makes it more optimistic than cache locks and causes it more restarts. The number of restarts for the cache locks algorithm also increases with higher cache hit probability but not as fast.

6.3 Co-Existence of Caching and Non-Caching Transactions

In the final set of experiments, we were interested in the effect of caching transactions on non-caching transactions and on overall throughput. We ran a set of experiments in which we fixed the multiprogramming level but varied the number of caching clients.

At a multiprogramming level of 200, increasing the number of caching clients had little effect on overall throughput because the cache was so frequently invalid. This can actually be verified in Figures 4 through 9 where the variation between non-caching two-phase locking and the two caching algorithms is very modest for high levels of multiprogramming. We also observe from this set of figures that some of the best throughput numbers were obtained with an multiprogramming level of 25. Thus, we decided to fix the multiprogramming level at 25 and vary the number of caching clients.

Figure 18 shows the overall server throughput when caching clients use the cache locks algorithm or the notify locks algorithm at a multiprogramming level of 25, a cache hit probability of 0.5 and a cache size of 15. The figure also displays the server throughput when all clients use non-caching, two-phase locking. Figure 19 shows the same data but for a cache size of 60.

The results of Figure 18 are intuitive. They show that increasing the number of caching clients improves

solid - two-phase locking dotted - cache locking dashed - notify locking

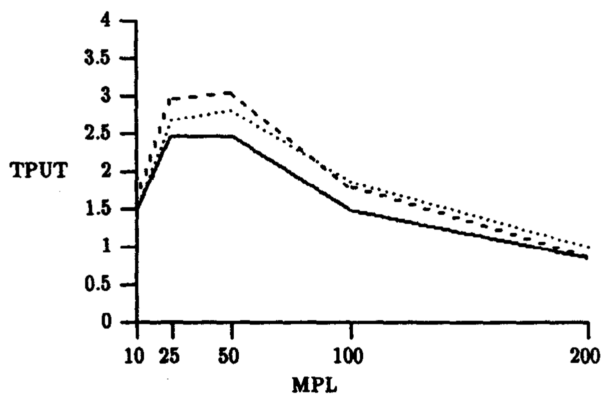

Fig 16: Thruput for Cache Size 15, Hit Prob 0.5, Int Think 5

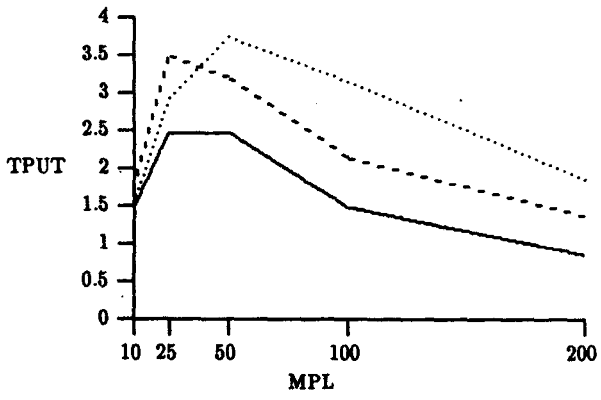

Fig 17: Thruput for Cache Size 15, Hit Prob 1.0, Int Think 5

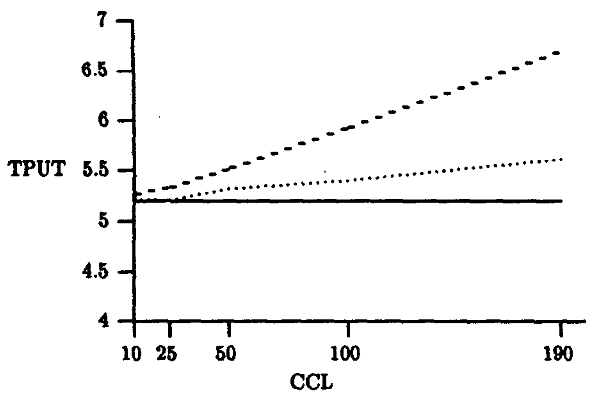

Fig 18: Thruput for Cache Size 15, Hit Prob 0.5, Int Think 0, MPL 25

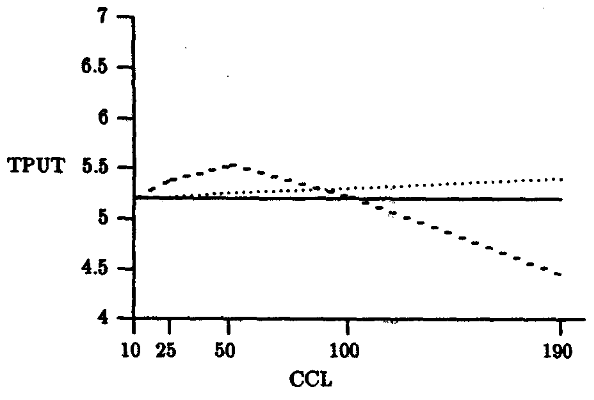

Fig 19: Thruput for Cache Size 60, Hit Prob 0.5, Int Think 0, MPL 25

overall throughput. They also show notify locks to substantially improve throughput as the number of caching clients increases. This is consistent with the results of Figure 8 in which the notify locks algorithm gets good usage out of its cache when the cache size is small and the level of multiprogramming is low.

The results of the second figure display more clearly the instability of the notify locks algorithm. As was seen in Figures 5, 7, 9 where the performance of the notify locks algorithm deteriorated as the CPU utilization went up, Figure 19 shows this phenomenon more clearly. The throughput first improves as the number of notify locks caching clients increases. But when the number of caching clients exceeds 50, the CPU saturation due to notify messages starts showing up and results in a severe deterioration in throughput by the time the number of caching clients reaches 190.

7 Conclusion

We have presented in this paper two techniques for combining a cache consistency algorithm with a concurrency control algorithm. The first algorithm is based on a new kind of lock, *cache locks*. The second algorithm is an adaptation of notify locks. Both algorithms add very little complexity to existing lock managers that implement two-phase locking and both algorithms may coexist with non-caching transactions that use two-phase locking.

The coupling of cache consistency and concurrency control simplifies the resulting system in two ways. First, in a database system cache consistency and concurrency control must be coordinated since caches are invalidated on transaction boundaries, i.e. updates are made public when write locks are released. This coordination is easier when the modules are combined. Second, the client code is simplified because there are fewer cases to deal with, e.g. cache invalidation can be treated as denial of a lock request or a concurrency control abort.

Our performance data show the relative merits of both algorithms and their effect on non-caching transactions. They show that the notify locks algorithm has better performance than cache locks under some circumstances. However, the notify locks algorithm is very sensitive to CPU utilization and multiprogramming level. And since database management systems tend to be CPU bound, use of this scheme may be risky. Furthermore, the cache locks algorithm exhibits good performance when it can take advantage of its cache. It is stable under all conditions and never does worse than non-caching two-phase locking. Finally, the performance results show that by carefully choosing the appropriate caching algorithm for the prevailing conditions, caching can significantly improve the overall throughput of the server.

Acknowledgements

The authors would like to thank Curtis Kolovson and Peter Schauble for their helpful comments on an earlier draft of this paper.

References

[1] R. Agrawal, M. J. Carey, and M. Livny, "Models for Studying Concurrency Control Performance Alternatives and Implications", *Proceedings of the 1985 SIGMOD Conference*, Austin, TX, May 1985.

[2] J. A. Blakeley, P.-A. Larson, and F. W. Tompa, "Efficiently Updating Materialized Views", *Proceedings of the 1986 SIGMOD Conference*, Washington, D.C., June 1986.

[3] H. Boral, and I. Gold, "Towards a Self-Adapting Centralized Concurrency Control Algorithm", *Proceedings of the 1984 SIGMOD Conference*, Boston, MA, May 1984.

[4] O. P. Buneman, and E. K. Clemons, "Efficiently Monitoring Relational Databases", *ACM Transactions on Database Systems*, Vol. 4, No. 3, September 1979.

[5] M. F. Fernandez, and S. B. Zdonik, "Transaction Groups: A Model for Controlling Cooperative Transactions", *Proceedings of the Third International Workshop on Persistent Object Systems: Their Design, Implementation and Use*, Newcastle, Australia, January 1989.

[6] H. M. Gladney, "Data Replicas in Distributed Information Services", *ACM Transactions on Database Systems*, Vol. 14, No. 1, March 1989.

[7] M. F. Hornick, and S. B. Zdonik, "A Shared, Segmented Memory System for an Object-Oriented Database", *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, January 1987.

[8] H. Kung, and J. Robinson, "On Optimistic Methods for Concurrency Control", *ACM Transactions on Database Systems*, Vol. 6, No. 2, June 1981.

[9] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms, "A Snapshot Differential Refresh Algorithm", *Proceedings of the 1986 SIGMOD Conference*, Washington, D.C., June 1986.

[10] J. E. B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, Ph.D. thesis, Massachusetts Institute of Technology, April 1981.

[11] M. A. Neimat, T. Connors, W. Hasan, K. Wilkinson, "The Papyrus Integrated Data Server", submitted to *The Fourth International Workshop on Persistent Object Systems: Their Design, Implementation and Use*, Marthas Vineyard, MA, September, 1990.

[12] T. Risch, "Monitoring Database Objects", *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam, The Netherlands, 1989.

[13] A. Rosenthal, S. Chakravarthy, B. Blaustein, J. Blakeley, "Situation Monitoring for Active Databases", *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam, The Netherlands, 1989.

[14] H. Schwetman, *CSIM Reference Manual* MCC Technical Report, ACA-ST-252-87, November, 1987.