Appears in the *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Jose, CA, May 1995

Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks

Atul Adya **Robert Gruber** **Barbara Liskov** **Umesh Maheshwari**

Laboratory for Computer Science,
Massachusetts Institute of Technology,
545 Technology Square, Cambridge, MA 02139
{adya,gruber,liskov,umesh}@lcs.mit.edu

Abstract

This paper describes an efficient optimistic concurrency control scheme for use in distributed database systems in which objects are cached and manipulated at client machines while persistent storage and transactional support are provided by servers. The scheme provides both serializability and external consistency for committed transactions; it uses loosely synchronized clocks to achieve global serialization. It stores only a single version of each object, and avoids maintaining any concurrency control information on a per-object basis; instead, it tracks recent invalidations on a per-client basis, an approach that has low in-memory space overhead and no per-object disk overhead. In addition to its low space overheads, the scheme also performs well. The paper presents a simulation study that compares the scheme to adaptive callback locking, the best concurrency control scheme for client-server object-oriented database systems studied to date. The study shows that our scheme outperforms adaptive callback locking for low to moderate contention workloads, and scales better with the number of clients. For high contention workloads, optimism can result in a high abort rate; the scheme presented here is a first step toward a hybrid scheme that we expect to perform well across the full range of workloads.

1 Introduction

In a distributed object-oriented database system in which persistent storage for objects is provided at server machines and applications run at clients, client caching is needed to provide good performance for applications. This paper presents an efficient concurrency control scheme for use in such a system. The scheme provides serializability for transactions, and also external consistency [12] so that transaction commit order as observed by clients is the same as the real time order. The scheme is an optimistic algorithm [18] that uses backward validation [17]. It performs better than other concurrency control algorithms for an important class of workloads — when there is low to moderate contention between user transactions.

The scheme uses timestamps generated from local clocks to define the serial order of transactions; we assume clocks are loosely synchronized, which is the case in networks today [24]. This approach is simpler and cheaper than previous techniques [2, 6, 14, 25]; other techniques are discussed in Section 3.6. More importantly, timestamps allow us to truncate transaction history and still avoid spurious aborts. By reading its local clock, a server can estimate the range of timestamps of transactions that are likely to request a commit there; only information needed for those transactions is retained. Our technique has the desirable property that we depend on the assumption about synchronization only for performance; if clocks get out of synch we may abort some transactions that could have committed but we will never commit transactions that should have aborted. Our use of synchronized clocks here is similar to their use in other distributed algorithms [20], but to our knowledge this is the first time that they have been used in this way in a concurrency control scheme.

The scheme has good performance in both space and time. It maintains only a single persistent version per object and requires no disk storage for concurrency control information. Its primary memory requirements are low: it only needs to retain information about objects that have been modified recently, and about transactions that are in the process of committing or that have committed recently. Our storage requirements are roughly equivalent to those of distributed pessimistic schemes; we compare them to the storage requirements of other optimistic schemes in Section 3.6.

The best concurrency control method proposed for client-server object-oriented systems prior to our work is adaptive callback locking [4]. Our technique outperforms adaptive callback locking in workloads in which there is low to moderate contention; we present the results of simulation studies that show this in Section 4. The reason for our good performance is that we send fewer concurrency

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136, and in part by the National Science Foundation under Grant CCR-8822158.

control messages than other schemes. Other than the standard 2-phase commit messages required of all distributed schemes, no messages are sent for concurrency control while transactions are running or committing. After a transaction commits, servers send invalidation messages to clients that have obsolete copies of objects in their caches. However, these messages are piggy-backed on other messages already being sent.

Contention causes transactions to abort in our scheme (as it does in all optimistic methods) whereas in a pessimistic approach contention only causes delays (and occasional deadlocks). Therefore, our approach is not appropriate when there is high contention. We plan to address this failing by using a hybrid scheme in which the system switches dynamically to a pessimistic approach for high-contention objects [15]. The design of the hybrid approach and its performance analysis is ongoing [16].

The remainder of the paper is organized as follows. Section 2 describes the environment for our work. Section 3 describes our algorithm and compares it to other work on optimistic concurrency control. Section 4 compares our performance to adaptive callback locking. We conclude with a discussion of what we have accomplished.

2 The Environment

Our work has been done in the context of the Thor object-oriented database [21, 22]. Thor allows user applications to share a universe of persistent objects. Objects are encapsulated for safe sharing, and applications access them by invoking methods that observe or modify their object's state. Application computations occur within transactions so that persistent objects can be maintained consistently despite concurrent accesses from multiple applications and possible failures. Each application runs a single transaction at a time. It specifies when to commit the current transaction: the transaction includes all methods invoked by the application since the last commit point.

Applications run on client machines. Persistent objects, however, are stored at servers (on the server disks). There may be many servers; the server where an object resides is referred to as its *owner*. New persistent objects may be created as a result of the methods invoked by an application, and objects can migrate between servers. Both object creation and object migration require transaction support but we ignore them in this paper; see [1] for a discussion of these issues.

To improve performance, methods are executed on the client machine using locally cached copies of objects. Objects are *fetched* from their servers when needed, and when an object is fetched, a number of related objects are prefetched [7]. The server tracks the objects in the client cache; for each client, it maintains a table called the *cached set* that records this information. The cached sets are used as part of transaction processing as discussed below; they are also used for other purposes, such as garbage collection [23].

The code that manages the cache on the client machine is part of Thor and is called a *front end*. Each server has a cache of objects in main memory, which it uses to satisfy fetch requests from clients. The organization is shown in Figure 1.

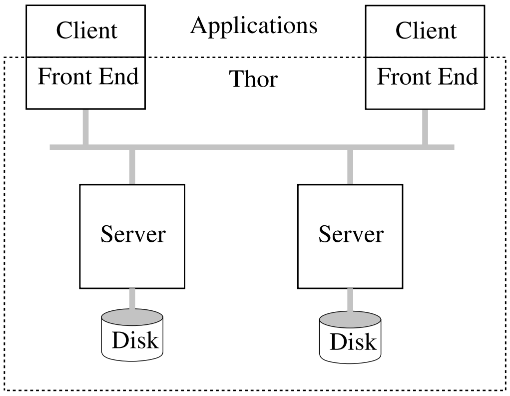


Figure 1: The client-server model of Thor.

The front end keeps track of objects read and written by the current transaction T at its client. When the client requests a commit, the front end gathers the data relevant to T:

- *validation information* — identity of each object used by T along with the type of access (read or write)
- *installation information* — modified copies of objects

This information is collected only for mutable objects (objects whose state can change) since concurrency control is not necessary for immutable objects. Thor makes this optimization possible by distinguishing between the two kinds of objects.

To commit a transaction, the front end sends its installation and validation information to a server that is the owner of some of the objects used by the transaction. This server commits the transaction unilaterally if it owns all objects used by the transaction. Otherwise, it acts as the *coordinator* of a commit protocol with the other owners, called the *participants*. We use a standard 2-phase protocol [13]. We describe the protocol briefly here to provide a context for our scheme.

In phase 1, the coordinator sends *prepare* messages containing the validation and installation information to the participants. Each participant tries to *validate* the transaction; we will describe how validation works in Section 3. If validation succeeds, the participant logs the installation information on stable storage and sends a positive response to the coordinator; otherwise it rejects the transaction. If all participants respond positively, the coordinator commits the transaction by logging a commit record on stable storage. Then it notifies the client of its decision. Note that the delay observed by the client before it can start the next transaction is due to phase 1 only; phase 2 happens in the background. Phase 1 includes two stable log updates, but the optimizations suggested by Stamos [26] can reduce this to a single log update.

In phase 2, the coordinator sends *commit* messages to the participants. On receiving a commit message, a participant *installs* new versions of its objects that were modified by that transaction (so that future fetches see the updates), logs a commit record on stable storage, and sends an acknowledgement to the coordinator.

If a transaction has not modified any object at some participant, we say it is *read-only at the participant*. Such a participant has no installation information to log, and does not need a commit message from the coordinator during phase 2.

If a transaction has not modified an object at any participant, we say it is *read-only*. For such transactions, no stable information needs to be logged and phase 2 is not required. Some other concurrency control schemes (e.g., callback locking and some multi-version schemes [2]) guarantee that all read-only transactions are serializable, so committing such transactions does not require any communication with the servers. Our scheme, however, does require validation for read-only transactions, and therefore phase 1 messages must be sent to the participants. Fortunately, the coordinator of a read-only transaction does not need to be reliable, and does not need to use stable storage. Therefore, the front end can act as the coordinator: it sends the prepare messages to the participants and collects their responses. This saves a message round trip, reducing the latency for committing read-only transactions to a single message round trip (and no stable updates).

After a read-write transaction for some client C has committed at a server, the server sends *invalidation messages* to clients other than C that are caching objects installed by that transaction; it determines what invalidation messages to send by looking at the cached sets. These messages are not required for correctness but they have two desirable effects:

1. If a front end receives an invalidation message for an object x that has been read by the current transaction, it simply evicts x, thus avoiding a potential abort that would otherwise result from reading x later.

2. If the current transaction has already read x, the front end aborts the transaction immediately, thus limiting wasted work (since the transaction would abort later anyway).

An additional benefit of this approach is that it offloads some of the validation work from the server to the front ends.

Front ends send acknowledgements after processing invalidation messages; when the server receives the ack, it removes the information about the invalidated objects from that client's cached set. Both invalidation messages and acks are piggy-backed on other messages being exchanged between the client and server. There is always a certain amount of such traffic: in addition to fetch and commit requests and replies, servers and clients exchange "I'm alive" messages for failure detection purposes. Therefore, our scheme does not cause extra message traffic, although messages may be bigger.

Although the cached sets are maintained for each client, their space overhead is not large because we store it at a coarse granularity. Objects are clustered in groups; the cached set contains the identifiers of the groups containing objects cached at a client rather than individual object identifiers. The client cache may not contain all objects in a group and therefore some unnecessary invalidation messages may be sent (piggy-backed on other messages). These messages do not cause any spurious aborts at the client, however, since an invalidation message causes an abort only if the corresponding object is being used at the client.

3 An Efficient Validation Scheme

The purpose of validation is to prevent the commit of any transaction that would violate the consistency requirements. Two desirable consistency properties traditionally provided by transactions are:

1. *Serializability*: The committed transactions can be placed in a total order, called the serialization order, such that the actual effect of running the transactions is the same as running them one at a time in that order.

2. *External consistency*: The serialization order is such that, if transaction S committed before T began (in real time), S is ordered before T.

Our scheme uses *backward validation* [17] to preserve consistency: a validating transaction T is checked against all transactions that have already validated successfully.

This section describes how we perform validation. It assumes the validation algorithm is embedded in the commit protocol described in Section 2. It describes our algorithm under the assumption that servers do not fail; we discuss how to tolerate server crashes in Section 3.5. We describe a sequential algorithm. As we discuss below, our validation scheme uses only in-memory data structures; validation can be performed quickly (e.g., without disk delays), and therefore a parallel algorithm is not required.

3.1 Global Serialization

In a distributed system, transactions that have read and modified objects at multiple servers must be serialized in the same order at all servers. We order transactions in a new way: we use timestamps taken from real clocks. (The ordering techniques used by some other optimistic schemes are discussed in Section 3.6.) Our scheme takes advantage of the presence of *loosely synchronized clocks* in distributed systems. Loose synchronization implies that the clocks at different nodes in the network may differ by at most a small skew (say, a few tens of milliseconds). The presence of such clocks is a reasonable assumption for current systems; protocols such as the Network Time Protocol [24] provide such a facility. These clocks simplify our algorithm and, since their values are close to real time, they allow us to make time-dependent design decisions and to reason about the performance of our scheme.

Each transaction T is assigned a timestamp T.ts by its coordinator when the coordinator receives the commit request from the client. The timestamp consists of the local time at the coordinator augmented with the coordinator's server ID to make it globally unique: T.ts = $\langle$time, server-ID$\rangle$. Timestamps are totally ordered by the time field, with ties resolved using the server ID. Transactions are serialized in timestamp order.

After assigning a timestamp to transaction T, the coordinator sends a prepare message to each participant P; the message includes the following validation information:

- T.ts: the timestamp of T.
- T.ReadSet: IDs of objects at P that were read by T.
- T.WriteSet: IDs of objects at P that were modified by T.
- The identity of the client where T ran.

Transactions S and T *conflict* if one has modified an object that the other read or modified. Upon receiving the validation information, each participant performs checks to ensure that conflicting transactions can be serialized in timestamp order. To simplify our algorithm, we arrange the read set to always contain the write set, i.e., if a transaction modifies an object but does not read it, we enter the object in the read set anyway. This implies that some transactions might exhibit spurious read-write conflicts, but aborts due to such spurious conflicts are rare because it is unlikely that a transaction writes an object without reading it.

Each participant uses the validation information sent by the coordinator to validate the transaction locally. It records the validation information of all successfully validated transactions in a *validation queue*, or VQ. The validation information of the validating transaction is compared with the records in the VQ to detect conflicts and ensure serializability and external consistency. For now, assume that validation records are never removed from the VQ.

If an incoming transaction T fails a validation check against a prepared transaction S, the participant aborts T by sending a negative ack to the coordinator. It cannot abort S instead of T because, in the 2-phase commit protocol, a participant cannot unilaterally abort a prepared transaction. Thus, the participant has to abort T even if it has an earlier timestamp than S.

The rules for validation differ according to the timestamp order between S and T. The next two sections describe these checks.

3.2 Checks Against Later Transactions

In this section, we consider the checks performed by each participant to validate a transaction T against a validated transaction S that has a later timestamp.

Since different transactions may be timestamped at different coordinators, and clocks are only loosely synchronized, a transaction T that begins after some other transaction S committed may actually receive a timestamp that is earlier than that of S (although this situation is very unlikely). In this case, T must not be committed if it read any object that S modified, since that would violate external consistency.

We preserve external consistency by ensuring that T fails validation if it conflicts with a validated transaction S that has a later timestamp:

> For each validated transaction S with timestamp later than T, we check that T did not read any object that S modified, and that T did not modify any object that S read. We call this the *later-conflict* check.

Here is an informal argument why this check provides external consistency while preserving serializability: If S committed before T began, S must have validated successfully at all its participants before T arrived for validation at any of them. If S has a later timestamp than T, the later-conflict check confirms that T does not conflict with S at any common participant. In this case, it does not matter whether T is serialized before or after S: both orders are valid serial orders, but while one is consistent with the timestamp order (T before S), it is the other that is externally consistent (S before T). Thus, the existence of a serialization order that is externally consistent is guaranteed, although it may be different from the timestamp order.

The later-conflict check is actually stronger than would be required for serializability alone: it might cause a transaction to abort even if it is serializable. We expect such aborts to be rare. They can happen only when the two conflicting transactions start to commit within the same "time window." The size of this window depends on message delays and clock skews. We conjecture that the window is usually very small (in the absence of failures such as clocks out of synch or network partitions), and therefore relatively few transactions start committing within the same window. The probability that some transactions within such a small group conflict in their use of some object is very small, and spurious aborts are possible only for those transactions and only if the one that reads is given a lower timestamp than the one that writes and their prepare messages arrive at some participant in the wrong order.

3.3 Checks Against Earlier Transactions

Now we consider the checks performed by each participant to validate a transaction against validated transactions that have earlier timestamps. For each validated transaction S with timestamp earlier than T:

1. S read object x and T modified x: No check is necessary because S could not have read the version of x written by T, since T is not yet committed.

2. S modified object x and T read x: We ensure that T read the version of x written by S or a later transaction. There are two cases:

   (a) If S is not committed, the check fails because T could not have read the result of an uncommitted transaction.

(b) If S is committed, the outcome depends on the version T read. We call this the *version check*.

The version check is the only check that depends on the specific versions of objects used by the validating transaction. Actually, we employ a simpler check, the *current-version check*:

Check that T has read the latest installed version of x.

The current-version check is safe because it is at least as strong as the version check. It might seem at first that the current-version check is stronger than the version check and would cause some transactions to fail validation that might have succeeded with the version check, but this is not so. Consider a transaction T that read x and has passed the later-conflict check discussed in the previous section, i.e., T has been validated successfully against all validated transactions with later timestamps. This implies that no validated transaction with a later timestamp could have written x. Now, if T passes the version check, it must have read a version at least as recent as that installed by any committed transaction with an earlier timestamp. The above two claims together imply that T must have read the latest version, and therefore will pass the current-version check.

The current-version check could be performed by associating a version number (such as the timestamp of the installing transaction) with each object. The version number of the object read by T can then be checked against that installed by S. However, storing a version number per object consumes disk storage as well as space in the server cache.

Instead, we perform the current-version check without using version numbers. Recall that the server maintains a cached set for each client to keep track of the objects that a client is caching; it uses the set to recognize when to send invalidation messages. The server now also maintains an *invalid set* for each client; the invalid set identifies those objects in the cached set that have been invalidated. As part of committing a transaction, the server adds the appropriate object identifiers to the invalid sets of other clients. To check whether a validating transaction read the latest version of an object, the participant checks the invalid set of the client where the transaction ran and rejects the transaction if it used an object in that invalid set.

As mentioned earlier, when a front end receives an invalidation message, it drops the listed objects from its cache, aborts the current transaction if it used one of those objects, and then sends an acknowledgement to the server. On receiving the ack, the server removes the listed objects from the invalid set of the client. Invalid sets will be small (just a few entries) because of the acks (see Section 4 for some data that confirm this expectation). The only time a set might be large is if the client has failed or the server and client cannot communicate. Failures are expected to be rare, and furthermore if a server cannot communicate with a client for some period of time (e.g., a few minutes), it shuts the client down; our shut down protocol is discussed in [23].

Thus, the memory requirements for validation are quite modest, modest enough that we can keep the needed information in primary memory. We require cached sets, invalid sets, and the VQ. We have already argued that the cached sets and invalid sets are small. The next section discusses how to keep the VQ small as well.

3.4 Truncation

In a practical implementation, the validation records for previously validated transactions cannot be retained indefinitely. Old validation records need to be truncated from the VQ to make space for new ones. Truncation is also necessary because otherwise transactions will need to be validated against a growing number of validation records.

We truncate information as follows. We never remove information about read-write transactions that have not yet committed. We do remove information about committed transactions, however, and also about read-only transactions. The invalid sets provide a compact summary of the effect of modifications of committed transactions on running transactions. To capture the information about what was read by removed transactions, we maintain a *threshold* timestamp. This timestamp is guaranteed to be greater than or equal to the timestamps of all transactions whose information has been removed from the VQ. Thus, our scheme maintains the following invariant:

> The validation record is retained for all uncommitted read-write transactions and for all transactions with timestamps above the threshold.

Note that the VQ may contain records for transactions with timestamps below the threshold.

A transaction T timestamped below the threshold fails validation. This additional validation check is called the *threshold check*. The check is required because information necessary for the later-conflict check has been discarded from the VQ. On the other hand, for a transaction that passes the threshold check, the earlier checks are sufficient.

The choice of where to maintain the threshold is important: setting it too low would result in undue retention of validation records (a long VQ), and setting it too high would cause transactions to fail the threshold check (spurious aborts). The threshold should be kept as high as possible, yet low enough that most transactions arriving for validation are still timestamped above it. We take advantage of the fact that our timestamps are based on loosely synchronized clocks to establish such a level.

Recall that a transaction is timestamped by the coordinator based on its local clock, and then the coordinator sends prepare messages to the participants. If *msg_delay* is the estimated bound on message delays (including retransmissions) and *skew* is the bound on clock skews, the timestamp of the arriving transaction will almost always be later than the local time at the participant minus (msg_delay+skew). Thus, it is desirable for threshold to lag behind the local time by (msg_delay+skew). We refer to this interval as the *Threshold Interval*. We truncate the VQ periodically; at that point we compute a new threshold, and then remove validation records for transactions whose timestamp is lower than the new threshold, provided they are committed or are read-only at this participant,

A simple alternative to a threshold that lags behind the local time is to remove transactions from the VQ at their commit point and set the threshold to the timestamp of the latest committed transaction [1]. Given that the commit of a distributed transaction is preceded by phase one messages and log forces, chances are that this threshold level would be sufficiently behind the local time to allow transactions to pass the threshold check. However, single-site transactions commit much faster than distributed transactions; removing their validation records at commit time would lead to a high threshold, causing distributed transactions to fail validation.

In summary, our scheme allows the threshold to be set at a level lower than the timestamp of the latest committed transaction. This improves the chances of a transaction passing the threshold check. In particular, the threshold can lag behind the local time by an interval that can be tuned to tradeoff speedy truncation of validation records against the likelihood that a transaction will pass the threshold check.

We have now described all of the validation checks performed at each particapant to validate a given transaction T; Figure 2 summarizes these checks.

Threshold Check

If T.ts $<$ Threshold then
    Send abort reply to coordinator

Checks Against Earlier Transactions

For each uncommitted transaction S in VQ
such that S.ts $<$ T.ts
    If (S.WriteSet $\cap$ T.ReadSet $\neq \phi$) then
        Send abort reply to coordinator

Current-Version Check

% T ran at client C
For each object x in T.ReadSet
    If x $\in$ C's invalid set then
        Send abort reply to coordinator

Checks Against Later Transactions

Later-Conflict Check

For each transaction S in VQ
such that T.ts $<$ S.ts
    If (T.ReadSet $\cap$ S.WriteSet $\neq \phi$)
    or (T.WriteSet $\cap$ S.ReadSet $\neq \phi$) then
        Send abort reply to coordinator

Figure 2: Validation Checks for Transaction T

3.5 Crash Recovery

When a server recovers from a crash, it must ensure that transactions it validates after the crash are serializable with transactions it validated before crashing. The straightforward way to ensure this would be to log the validation information (VQ and threshold) on stable storage, so that it can be recovered after a crash. For read-write transactions, the validation record can be logged along with the installation information without causing any significant increase in the latency. Read-only transactions, however, do not have any installation information, so logging validation information for them would increase their latency. Therefore, we do not log validation information for such transactions.

When the VQ is recovered from the log after a crash, the records of read-only transactions are missing. We can accommodate the missing information if we can preserve the truncation invariant given in Section 3.4. We do this as follows. We maintain a *stable threshold*, which is always later than the timestamp of any transaction that has ever validated at the server. On recovery, the threshold is set to the logged value of the stable threshold. This technique is similar to that used for at-most-once messages in [20].

The stable threshold must be increased whenever the timestamp of a validating transaction is later than its current value. We avoid frequent updates to the stable threshold by increasing it in jumps, e.g., setting it to one second ahead of the current clock time, so that it does not need to be increased for a number of subsequent transactions. Thus, most read-only transactions do not need to log any stable updates to the stable threshold. In fact, the stable threshold can be increased whenever there is an opportunity to do so without delaying any client, e.g., when the installation information of a read-write transaction is logged, or in the background.

It is desirable to keep the stable threshold close to local time plus some suitably chosen interval δ. If δ is too small, the stable threshold will need to be updated frequently; if it is too large it may cause unnecessary aborts of transactions after recovery. However, no matter what δ is chosen, failures may cause some transaction aborts after recovery because information is missing, and we must make worst-case assumptions: we must assume that the incoming transaction with a timestamp below the threshold conflicts with some transaction that committed before the crash, even though it might not. Note that these aborts happen only when recovery is very fast; otherwise, transactions with timestamps earlier than the stable threshold will have aborted long before recovery occurs.

Use of the stable threshold implies that there is no need to log any information about reads. In fact our log contains only installation information, which is sufficient to recover the write information in the VQ.

The cached sets are also not maintained on stable storage. Instead, the server maintains the addresses of clients that have cached its objects. After a crash, the server communicates with the clients and rebuilds their cached sets. (If it is unable to communicate with one of the clients for some period, it shuts the client down as discussed earlier.)

Invalid sets are recovered from the combination of the installation information in the log and the recovered cached sets. The sets may contain unnecessary entries, due to lost acks. However, a simple protocol allows us to recover the information from these lost acks. When a transaction commits that causes invalidations, the server generates an *invalidation number* that is stored in the transaction's commit record; later invalidation numbers are always larger than earlier ones. The invalidation number is included in the invalidation message, and the front ends remember the number of the last invalidation they have acknowledged for that server. This information is sent to the server along with the cached set; this allows the server to discard all entries from the invalid set that have been acknowledged by the front end.

3.6 Comparison with Other Optimistic Schemes

Eswaran *et al.* [8], and later Kung and Robinson [18], suggested the idea of using optimism for concurrency control. Since then a number of optimistic schemes have been discussed in the literature.

Other distributed schemes achieve a global serialization order using atomic multicast [25] or logical clocks [2, 14]. Atomic multicast adds additional per-message overhead, while logical clocks must be explicitly managed as part of the two-phase commit, complicating the algorithm. The scheme described in [6] used the following approach: To validate a transaction T, a participant computes the set of other validating transactions that T conflicts with, waits for these to commit or abort, and then makes its commit decision. This process can lead to deadlocks, which are resolved with timeouts and retries during phase 1. Our use of loosely synchronized clocks avoids all these problems; more importantly, it also allows us to make time-dependent decisions (e.g., about what transactions to remove from the transaction history).

Optimistic schemes can be classified [17] into *forward* and *backward* validation schemes. Backward validation algorithms such as our scheme and those proposed in [2, 3, 6, 18, 19, 25] validate a transaction against already-committed transactions, while forward validation algorithms such as O2PL [10] validate a transaction against currently executing transactions. (Both approaches also validate against other validating transactions.) In a client-server system, forward validation requires a validating transaction to contact all clients that are caching updated objects, to obtain latches on the cached copies. If a latch cannot be obtained, the transaction aborts, otherwise it commits and releases the latches (updating or invalidating the client caches). This approach adds an additional validation phase to all commits, even when only a single server is involved; moreover, the delay incurred is observed by the committing client. In contrast, backward validation just uses the standard 1 or 2 phases required for a single-site or distributed commit, as we described for our scheme, and is therefore a better choice for a client-server system.

Previous optimistic schemes have largely ignored implementation issues regarding time and space overheads. Some schemes [2, 6, 18, 19] validate a transaction against all transactions serialized between its start and end times; a large amount of validation information must be stored to validate a long transaction. In our scheme, the invalid set summarizes most information required for validation in a compact way, while our VQ is truncated according to a bound on expected message delay; the information we maintain is not proportional to transaction length, but still allows us to correctly validate long transactions.

Multi-version schemes [2, 3, 19] keep multiple versions of objects to provide a consistent view of the database to all active transactions, making validation unnecessary for read-only transactions. However, this approach has very high space overheads. It also still requires validation of read-write transactions (where our efficient scheme could be usefully applied).

Most optimistic schemes store some concurrency control information per object (e.g., the scheme in [2] maintains two timestamp values with each version); we call this the version number approach. Space overhead for version numbers can be significant for small objects; e.g., most objects in the OO7 benchmark [5] are smaller than 100 bytes; an 8 byte version number would add 8% overhead. Another potential problem with version numbers is that they are usually cached and uncached with the rest of their object state; missing version numbers would then require disk reads during validation. In contrast, all of our validation information can be kept in main memory.

4 Simulation Study

This section presents a simulation study to demonstrate that our optimistic concurrency control (OCC) outperforms an adaptive-granularity callback locking scheme that caches read locks with data at the clients; we call this scheme ACBL[1] (adaptive callback locking). Carey, Franklin, and Zaharioudakis [4] demonstrated through simulation that ACBL outperforms non-adaptive callback schemes. Earlier, Franklin and Carey [9, 10] used simulation to explore a number of different schemes, and concluded that a non-adaptive callback scheme was the best overall choice; since ACBL is even better, we decided to compare OCC directly with ACBL.

ACBL is a page-based scheme that normally does locking and callbacks at the page level, but switches to object level locking and callbacks for pages that exhibit read-write sharing. This approach is better than a pure page-based scheme because it avoids "false conflicts," which lead to unnecessary waiting or aborts. It is better than a pure object-based scheme because it avoids sending multiple write lock

[1] ACBL is the "PS-AA" scheme in [4].

requests or multiple callbacks for objects in the same page for those cases where a single page-level lock or callback can be used.

To compare OCC to ACBL, we designed a page-based variant of OCC that we call AOCC (adaptive OCC). Like ACBL, page-based caching is used at the server and at clients and pages are sent in fetch replies. Also like ACBL, clients can mark objects within cached pages as being "unavailable," and the cached set for the client tracks which pages are cached at the client and which objects have been marked unavailable. Invalid sets record the identifiers of invalidated objects and these identifiers are sent in the invalidation messages. In response to an invalidation message, a client's front end drops the object's page if the current transaction isn't using any objects in that page; otherwise it just marks the invalidated object as unavailable. The front end informs the server of new page drops and newly unavailable objects in its acknowledgement message, allowing the server to update the cached set appropriately (and to remove entries the invalid set).

AOCC's adaptive mark/drop strategy is similar to the way ACBL handles a callback. (Note that AOCC uses object-level conflict detection; it only makes an object/page tradeoff with respect to invalidations.) While it is not clear that this approach is an improvement over the simple object marking scheme described in Section 3, it is the right choice for this study: with respect to measures such as client and server hit ratios, fetches per commit, and disk reads per commit, AOCC and ACBL have similar behavior, allowing us to focus on the different concurrency control costs for the two schemes.

The rest of this section briefly describes our simulator and then presents two sets of experiments. We first examine a low to moderate contention workload, and show that AOCC outperforms ACBL. We also show that AOCC scales better than ACBL with the number of clients. We then examine the question of read-only commits: AOCC requires a round-trip at commit time to validate the transaction's read set, while ACBL does not. We show that the percentage of read-only transactions must be very high for ACBL to outperform AOCC.

4.1 System Configuration

For simplicity, we model a single server system that services multiple clients. With only one server, distributed 2-phase commit is not required, and the validation checks for our scheme reduce to just the invalid set check. We believe the relative performance of AOCC vs. ACBL would not change if we were to model multiple servers and a 2-phase commit. The main differences between the two schemes involve the number of round-trip message delays incurred during transaction execution (ACBL incurs more message delays) and the number of aborts that can occur due to contention (AOCC is more susceptible to aborts). Adding a two-phase commit adds an additional round-trip delay for both schemes; this should not have a significant impact on the relative performance.

Parameter	**Setting**
Number of clients	1 – 24
Object size	100 bytes
Page size	4 Kbytes
Objects per page	40
Database size	1300 pages
Server disks	2
Server cache size	50% of DB
Server commit log size	50% of DB
Client cache size	25% of DB
Server CPU speed	50 MIPS
Client CPU speed	25 MIPS
Disk setup cost	10000 cycles
Fetch disk access time	1600 μsecs / Kbyte
Installation disk access time	1000 μsecs / Kbyte
Fixed network cost	10000 cycles
Variable network cost	2500 cycles / Kbyte
Network bandwidth	80 Mbits / sec
Cache lookup	300 cycles
Register / Unregister	300 cycles
Validation time per object	0–300 cycles
Deadlock detector frequency	0.01 seconds
Cost of deadlock detection	0 cycles

Figure 3: System and Resource Parameters and Settings

Figure 3 shows the relevant system parameters for our simulator. Initiating a disk or network access (send or receive) has a fixed CPU overhead; the network has an additional charge per Kbyte. For fetches that miss in the server cache, a disk read is required; the throughput chosen assumes clustered client access patterns, which results in a good amortized read cost. Installation of committed updates occurs out of a large in-memory commit log in our system, and can be scheduled very efficiently. (For the details of the in-memory log and the server's use of the disk, see [11].) The register/unregister cost is used to model the work done at the server to update a client's cached set; lock maintenance is included in this registration cost for ACBL. To be conservative in favor of ACBL, it is not charged any cycles for deadlock detection, while we do charge AOCC for commit-time validation of each identifier in the read set (the cost is proportional to the actual size of the client's invalid set).

As in the original ACBL study, we chose a relatively small database size to allow us to simulate client and server caches that are a large fraction of the total database size. It is the relative sizes of the caches and the database, along with the characteristics of the workload, that are important for comparing the different schemes, and not the actual sizes involved.

Figure 4 shows the relevant parameters for our workload generator. The database is split into regions: there is a special shared region containing objects used uniformly by all clients, per-client private regions mostly used by the "owning" client, and a final unclaimed area (the remainder of the database). There are three kinds of accesses: private accesses go to a client's own private region, shared accesses

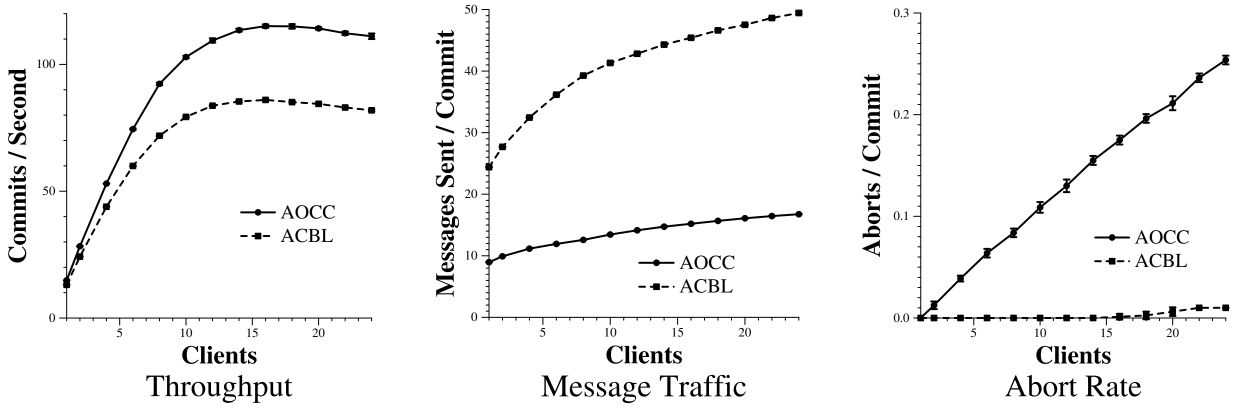


Figure 5: SH/HOTCOLD: Low To Moderate Contention Workload

Parameter	**Setting**
Pages per private region	50
Pages in shared region	50
Private access prob.	70%
Shared access prob.	10%
Rest of db access prob.	20%
Private write prob.	5%
Shared write prob.	5%
Rest of db write prob.	5%
Average trans. size	200 objects
Cluster size	5–15 objects
Read access think time	50 cycles / byte
Write access think time	100 cycles / byte
Think time between trans.	0
Percent forced read-only	0% – 100%

Figure 4: Workload Parameters and Settings

go to the shared region, and rest-of-database accesses go anywhere but the shared region or the client's own private region. Access and write probabilities are given for each type of access. In addition, accesses can be clustered within a page: once an appropriate page is chosen, a clustered set of accesses within this page occurs. (For the settings given, transactions average 200 object accesses total and 10 objects per page; thus they access 20 pages on average.)

Clients execute transactions continuously, applying a sequence of read and write accesses as determined by the workload generator. If a transaction aborts it is restarted immediately with the same sequence of accesses. In the case of an abort in AOCC, for those objects that the client read that are invalid, if the server is caching the current object state or it is available in the in-memory log, then the new state is sent with the abort reply and the object is removed from the client's invalid set; since the invalid set is also sent, all invalid cached objects are either updated or removed from the cache when the abort reply is received.

For each read or write access, some "think time" is charged at the client; this models local computation performed as part of the read or write. (The think times are per byte; multiply by 100 for the object size used here.)

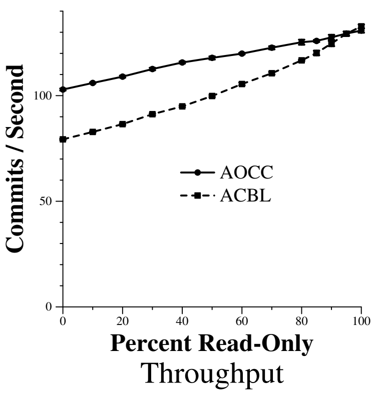
The workload settings shown are similar to the HOTCOLD workload in [4], except for the addition of a 50 page shared region; thus we call this the SH/HOTCOLD workload. We added the shared region because we believe that both uniform and biased page sharing is likely to occur in real workloads. While conflicts can occur over private pages due to rest-of-database accesses, these pages are almost always in use only at their own client; in contrast, shared pages are accessed uniformly, and are more likely to be used concurrently by multiple clients. It is possible to specify a percentage of the transactions to be forced read-only; this feature is used for the second set of experiments.

A simulator run involves 50000 commits, with results reported at 5000-commit intervals; all graphs have error bars for the 95% confidence interval.

4.2 Low-To-Moderate Contention

Figure 5 present the results for the SH/HOTCOLD with a 70/10/20 access split (private/shared/rest-of-db) and a 5% write probability for all accesses. First examine the left two graphs, which show throughput in commits per second and message traffic expressed as the number of messages per commit. The number of messages sent is a good predictor of the relative performance of the two schemes. AOCC and ACBL both send fetch messages when an access misses in the client cache; the two schemes have similar client cache behavior, and perform roughly the same number of fetches. These are the only messages AOCC uses prior to commit. In contrast, for a write access ACBL requires at least one message round-trip to acquire a write lock, plus callbacks to other clients caching the object. If a page-level write lock is acquired, further writes to other objects in the page do not require messages, otherwise each new write access within the page requires a new lock request (and new callbacks).

We find that AOCC outperforms ACBL across a range of parameter settings, as long as contention remains in the low to moderate region, with the amount of improvement



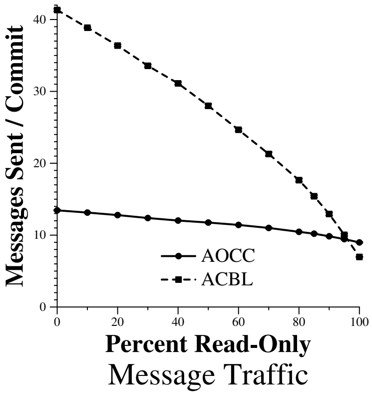


Figure 6: Effect of Read-Only Transactions

depending critically on the number of messages required. For the 5% write probability shown here, AOCC betters ACBL by between 14% and 36%. For a 10% write probability, where more lock requests and callbacks are required, AOCC shows up to a 50% improvement.

The performance of both schemes levels off as clients are added, but for different reasons. For ACBL, as more messages are sent, the server becomes heavily utilized due to message processing. At 24 clients, the server is 94% utilized, of which 87% is due to message processing. For AOCC, the disk bandwidth becomes a problem: AOCC uses over 80% of the full disk bandwidth at 12 clients, and reaches 97% utilization at 24 clients. The callback scheme is also affected by increased disk usage, but its disk utilization does not increase as quickly as AOCC, due to the extra message delays it incurs; it hits 70% utilization at 24 clients. AOCC is able to fully saturate the disks because it can sustain a higher commit rate.

Consider the number of messages sent per commit where there are from 1 to 10 clients. In this region no system resource is more than 75 percent utilized; thus it is useful for examining issues of scalability. For AOCC, an average of 0.5 additional messages is required per client added to the system, while for ACBL, an average of 1.9 additional messages is required. (This difference can be seen in the initial slopes of the two curves.) This finding supports our claim that AOCC scales better with the number of clients caching shared data.

Finally, consider the right-most graph in Figure 5, which shows the abort rate in aborts per commit. While ACBL's abort rate stays near zero, AOCC's abort rate grows linearly with the number of clients, and eventually becomes quite large: 1 in 5 transaction executions aborts at 24 clients. Note that AOCC outperforms ACBL by 36% at this point; the extra messages due to the re-execution of aborted transactions are still much lower than the messages required for locking. A high abort rate is acceptable in some cases but not in others; this issue is discussed in Section 5.

4.3 Read-Only Transactions

Our conclusion from the SH/HOTCOLD workload is that AOCC outperforms ACBL, even as contention becomes reasonably high. However, this workload contains very few read-only transactions. As stated earlier, locking performs better than an optimistic scheme for such transactions, since transactions can commit locally at the client. However, if a transaction modifies even one object, more messages are used by ACBL than AOCC.

We performed an experiment to compare ACBL with AOCC as the percentage of read-only transactions is varied. This experiment used the same SH/HOTCOLD workload, with the following modification: to obtain a workload with roughly X percent read-only transactions, each transaction produced by the workload generator is forced to be read-only X percent of the time, in uniform fashion. (The remainder of the transactions are generated as before; almost all of these are read-write.) The number of clients is fixed at 10.

Figure 6 shows the results. The graph of messages sent per commit once again tells the story. AOCC uses roughly the same number of messages regardless of how many read-only transactions there are; it sends a few less messages as the abort rate drops from 10% to 0%. In contrast, for the read-only transactions ACBL does not use lock requests, callbacks, or a commit request. Ultimately, at 95% read-only, the message counts are the same. For the 100% read-only case, ACBL is only sending fetch requests while AOCC is also sending a commit request. Not surprisingly, the crossover on the throughput graph also occurs at 95% read-only. Note that at 100% read-only, ACBL outperforms AOCC by only 1.5%, whereas even at a mix as high as 70% read-only, AOCC outperforms ACBL by 11% (it outperforms ACBL by 30% for the 0% read-only case). Our conclusion is that for almost all real-world workloads, AOCC is the better choice.

4.4 Discussion

It is important to note that the adaptive nature of ACBL is a big help for the SH/HOTCOLD workload; had we compared AOCC to a non-adaptive callback scheme, the differences shown would have been even larger. At 10 clients, 94% of all lock requests result in page-level write locks: ACBL uses many fewer messages than an object-based locking scheme would. On the other hand, about 4% of all pages in the system are using object-level locking. Note that the shared pages constitute 3.8% of the total pages: these are the pages most likely to exhibit false sharing, and using object-level locking here makes ACBL perform better than a page-based locking scheme.

We also used the simulator to measure the distribution of invalid set sizes at validation time. For a 24 client SH/HOTCOLD workload with 10% write probability for all accesses, 70% of the time the invalid set size was zero (no validation required), and over 99% of the time the size was less than 10; the maximum size was 24. This shows three things: the invalid sets maintained by the servers are small, the invalid set check can be performed quickly, and piggy-backed invalidations do not add significant overhead to message sizes.

5 Conclusions

This paper describes an efficient implementation of optimistic concurrency control for client-server object-oriented database systems. To provide reasonable performance to user transactions, these systems cache copies of persistent objects at client machines, run method calls at the clients, and provide a concurrency control mechanism to ensure that transactions are serializable.

Our scheme is simple to implement and provides external consistency as well as serializability. It performs better than other optimistic schemes with respect to both space and time. It does not store any concurrency information in objects or on the disk; instead it maintains just a small amount of validation state in the primary memories of servers. It adds no new messages over those required of any scheme for fetching and commit processing (but message sizes can be larger). It requires no disk I/O at commit time over what is ordinarily needed for two-phase commit; in particular it does not need to read concurrency control information from disk to validate transactions.

The scheme uses loosely synchronized clocks to truncate validation information. Sufficient information is maintained to allow validation for all "recent" transactions, i.e., all transactions for which prepare messages are likely. To our knowledge, this is the first time loosely synchronized clocks have been used in this way in a distributed concurrency control scheme.

The paper presents results of simulation experiments that compare the performance of an adaptive optimistic scheme (AOCC) with adaptive callback locking (ACBL). For low to moderate contention workloads, AOCC performs substantially better than ACBL, and exhibits a lower growth rate in the number of messages required per commit as clients are added to the system.

As is the case for all optimistic methods, our scheme can suffer from a large number of aborts when there is high contention. This is not a problem for some kinds of transactions, including transactions strictly under program control, and also user-initiated requests that can be redone on abort without further input (e.g., increment the salary of each employee in set S by 10%). For transactions in this class, aborts do not matter as long as the abort rate is not so high that performance problems result. Indeed, we show in Section 4 that even with an abort rate of 1 in 5 aborts, AOCC outperforms ACBL by a significant margin.

However, there are also transactions where aborts do matter, either because user intervention would be required or because the abort rate is so high that locking is the preferred mechanism. We intend to support a hybrid concurrency control scheme that uses optimism as the default but in addition supports locking. Applications can use explicit lock requests; this handles the user intervention case. In addition, the system will automatically detect which objects are under high contention and use locking for those objects, retaining the low message overhead of our optimistic scheme for all other accesses. The resulting system should have better performance than a locking-only system, but with a much lower abort rate than a pure optimistic scheme. Our research into the design and performance evaluation of such a hybrid scheme is ongoing [15, 16].

Acknowledgements

The authors are grateful to Dawson Engler, Wilson Hsieh, James O'Toole and the referees for their helpful comments.

References

[1] A. Adya. Transaction Management for Mobile Objects using Optimistic Concurrency Control. Tech. Report MIT/LCS/TR-626, MIT Lab. for Computer Science, January 1994.

[2] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. Distributed Multi-version Optimistic Concurrency Control with Reduced Rollback. *Distributed Computing*, 2(1), 1987.

[3] P. Butterworth, A. Otis, and J. Stein. The Gemstone Database Management System. *CACM*, 34(10), October 1991.

[4] M. Carey, M. Franklin, and M. Zaharioudakis. Fine-Graned Sharing in a Page Server OODBMS. In *Proceedings of the 1994 ACM SIGMOD*, Minneapolis, MN, May 1994.

[5] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 Benchmark. In *Proceedings of ACM SIGMOD*, May 1993.

[6] S. Ceri and S. Owicki. On the Use of Optimistic Methods for Concurrency Control in Distributed Databases. In *Proceedings of the 6th Berkeley Workshop*, 1982.

[7] M. S. Day. *Client Cache Management in a Distributed Object Database*. PhD thesis, Massachusetts Institute of Technology, February 1995.

[8] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The Notion of Consistency and Predicate Locks in a Database System. *CACM*, 19(11):624–633, November 1976.

[9] M. Franklin. Caching and Memory Management in Client-Server Database Systems. Tech. Report (Ph.D.) 1168, Computer Sciences Dept., Univ. of Wisconsin – Madison, July 1993.

[10] M. Franklin and M. Carey. Client-Server Caching Revisited. In *Proc. Int'l Workshop on Distributed Object Management*, Edmonton, Canada, August 1992.

[11] S. Ghemawat. *Main-Memory Log: A Simple Solution to the Disk Bottleneck*. PhD thesis, Massachusetts Institute of Technology, Forthcoming.

[12] D. K. Gifford. Information Storage in a Decentralized Computer System. Tech. Report CSL-81-8, Xerox Parc, 1983.

[13] J. N. Gray. Notes on Database Operating Systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, pages 394–481. Springer-Verlag, 1979.

[14] R. E. Gruber. Optimistic Concurrency Control for Nested Distributed Transactions. Tech. Report MIT/LCS/TR-453, MIT Laboratory for Computer Science, June 1989.

[15] R. E. Gruber. Temperature-Based Concurrency Control. In *Third IWOOOS*, pages 230–232, Asheville, December 1993.

[16] R. E. Gruber. *Temperature-Based Concurrency Control*. PhD thesis, Massachusetts Institute of Technology, Forthcoming.

[17] T. Haerder. Observations on Optimistic Concurrency Control Schemes. *Information Systems*, 9(2):111–120, June 1984.

[18] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM TODS*, 6(2):213–226, June 1981.

[19] M. Y. Lai and W. K. Wilkinson. Distributed Transaction Management in Jasmin. In *Tenth VLDB Conf.*, August 1984.

[20] B. Liskov. Practical Uses of Synchronized Clocks in Distributed Systems. In *Tenth PODC Conf.*, August 1991.

[21] B. Liskov. Preliminary Design of the Thor Object-Oriented Database System. Programming Methodology Memo 74, MIT Lab. for Computer Science, March 1992.

[22] B. Liskov, R. Gruber, P. Johnson, and L. Shrira. A Highly-Available Object Repository for use in a Heterogeneous Distributed System. In *Proc. of the 4th Int'l Workshop on Persistent Object Systems*, pages 255–266, September 1990.

[23] U. Maheshwari and B. Liskov. Fault-Tolerant Distributed Garbage Collection in a Client-Server, Object-Oriented Database. In *Third PDIS Conference*, pages 239–248, Austin, September 1994.

[24] D. L. Mills. Network Time Protocol: Specification and Implementation. DARPA-Internet Report RFC 1059, DARPA, July 1988.

[25] E. Rahm and A. Thomasian. A New Distributed Optimistic Concurrency Control Method and a Comparison of its Performance with Two-Phase Locking. In *Proceedings of Tenth ICDCS*, 1990.

[26] J. W. Stamos. A Low-Cost Atomic Commit Protocol. Tech. Report RJ7185, IBM Almaden, CA, December 1989.