Distributed Optimistic Concurrency Control Methods for High-Performance Transaction Processing

Alexander Thomasian, *Senior Member*, *IEEE*

Abstract—There is an ever-increasing demand for more complex transactions and higher throughputs in transaction processing systems leading to higher degrees of transaction concurrency and, hence, higher data contention. The conventional two-phase locking (2PL) Concurrency Control (CC) method may, therefore, restrict system throughput to levels inconsistent with the available processing capacity. This is especially a concern in shared-nothing or data-partitioned systems due to the extra latencies for internode communication and a reliable commit protocol. The optimistic CC (OCC) is a possible solution, but currently proposed methods have the disadvantage of repeated transaction restarts. We present a distributed OCC method followed by locking, such that locking is an integral part of distributed validation and two-phase commit. This method ensures at most one re-execution, if the validation for the optimistic phase fails. Deadlocks, which are possible with 2PL, are prevented by preclaiming locks for the second execution phase. This is done in the same order at all nodes. We outline implementation details and compare the performance of the new OCC method with distributed 2PL through a detailed simulation that incorporates queueing effects at the devices of the computer systems, buffer management, concurrency control, and commit processing. It is shown that for higher data contention levels, the hybrid OCC method allows a much higher maximum transaction throughput than distributed 2PL in systems with high processing capacities. In addition to the comparison of CC methods, the simulation study is used to study the effect of varying the number of computer systems with a fixed total processing capacity and the effect of locality of access in each case. We also describe several interesting variants of the proposed OCC method, including methods for handling access variance, i.e., when rerunning a transaction results in accesses to a different set of objects.

Index Terms—Distributed database systems, transaction processing, optimistic concurrency control, access invariance, commit protocols, system performance modeling.

1 INTRODUCTION

CONCURRENCY control (CC) is an important aspect of distributed transaction processing (the reader is referred to [47] for an overview of CC methods). Virtually all commercial database management systems still use two-phase locking (2PL) for synchronizing database accesses. However, since optimistic methods were first described in [25], a large number of optimistic concurrency control (OCC) methods have been proposed for centralized and distributed database systems [32] and have been implemented in several prototypes, particularly for distributed environments [36], [10], [22], [28], [30]. In this paper, we propose a hybrid OCC method for transaction processing, which combines OCC with locking in partitioned database systems. Locks are only requested at the time of transaction validation and commit processing to guarantee global serializability [5].

The ever-increasing demand for higher throughput for more complex transactions in online transaction processing leads to an increase in the degree of transaction concurrency (denoted by M) and a higher lock contention level, which manifests itself by an increased frequency in transaction blocking due to lock conflicts and restarts (to resolve deadlocks) [46]. In fact, as M increases there may be a sudden reduction in the number of *active* transactions due to transaction blocking, which eventually leads to a severe degradation in performance which is referred to as thrashing [41], [43]. Thus, high-performance transaction processing requirements may not be satisfiable by 2PL. This is particularly so in shared-nothing or data-partitioned architectures for transaction processing, because for a given transaction arrival rate M is increased due to extra delays introduced by intersystem communication.

The OCC protocol allows a high degree of transaction concurrency and has been shown to outperform 2PL in systems with “infinite” [11], [2] or at least “adequate” hardware resources [12], [14]. More specifically, much higher transaction throughputs can be attained by OCC versus 2PL, at the cost of the additional processing capacity required for transaction restarts. Another advantage of OCC methods compared to 2PL is that they are deadlock-free, since deadlock detection schemes for distributed database systems tend to be complex and have frequently been shown to be incorrect [23]. While time-outs can be used for deadlock resolution, their implementation is complicated, since it requires the determination of an appropriate time-out interval [3], [21]. Deadlock-free locking schemes such as wound-wait and wait-die are alternatives [37]. Simulation results in [11], [2] show that these methods are outperformed by optimistic methods when the data-contention

• *The author is with the IBM Thomas J. Watson Research Center, 30 Saw Mill River Rd., Hawthorne, NY 10532. E-mail: athomas@watson.ibm.com.*

Manuscript received 19 Feb. 1992.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 104468.

1041-4347/98/$10.00 © 1998 IEEE

level is high and adequate hardware resources are available. Another approach to prevent deadlocks is to limit the wait depth of blocked transactions to one. The wait depth limited (WDL) concurrency control method achieves this goal while attempting to minimize wasted processing due to transaction restarts [13]. A performance study of this method in a distributed system is reported in [15], [16], which shows that this method also outperforms 2PL in high-lock contention environments.

An important issue in OCC is the efficiency of the validation method. The simple validation method first proposed in [25] (for centralized systems) causes an unnecessarily high number of restarts, which can be prevented, e.g., by using timestamps for conflict detection [38], [31]. There have been several extensions of the original validation method to a distributed environment (see e.g., [8], [1]) (Agrawal et al. [1] use versioning to improve performance for read-only transactions). In the case of longer transactions, these methods generally cause an intolerably high number of restarts and are susceptible to "starvation" (i.e., transactions may never succeed due to repeated restarts). To overcome these problems, some authors proposed a combination of locking and OCC (see e.g., [26]), where transactions may be synchronized either pessimistically or optimistically. Though this is a step in the right direction, the resulting methods are no longer deadlock-free.

The proposed OCC protocol offers substantial benefits over existing OCC methods and can be used for high-performance transaction processing in data-partitioned or shared-nothing systems. The protocol to be described exhibits the following characteristics:

1) Transactions are executed optimistically, ignoring locks held by other transactions on the object. We also investigate an alternative method where an object access is delayed when updates are pending (see Section 2).
2) Before global validation is performed, the validating transactions request appropriate locks for all items accessed. Locks are only held during the commit phase (if validation is successful) so that lock conflicts are far less likely than with standard locking.
3) If validation should fail, all acquired locks are retained by the transaction while being executed again. This guarantees that the second execution phase is successful if no new objects are referenced. In this way, frequent restarts, as well as starvation, can be prevented. We also investigate an alternative method where lock requests may be deferred to a point where the execution of a transaction is considered to be successful (see Section 5.1).
4) Deadlocks in requesting locks are prevented by using a static locking paradigm, i.e., preclaiming locks for objects accessed by a transaction in its first phase and processing these requests in *the same order* at all nodes.
5) Lock requests do not cause any additional messages.
6) The protocol is fully distributed.

A key concept utilized here is **phase-dependent control** [12], [14], such that a transaction is allowed to have multiple execution phases with different CC methods in different phases. The need for multiphase processing arises in a high-data-contention environment, where the processing of transactions in one phase may lead to severe degradation in performance, such as thrashing in 2PL [41], [43] and excessive additional processing due to restarts in OCC [38]. **Two-phase processing** is preferable to these methods from the viewpoint of reduced transaction response time and also reduced additional processing. Even if a transaction is known to be conflicted in its first execution phase, its execution is allowed to continue in **virtual execution** mode [12], [14]. In spite of the additional processing, the virtual execution will result in prefetching of data into the database buffer, which is useful when the transaction is rerun, provided that we have **access invariance** [12], [14], i.e., the property that *the set of objects accessed by a transaction does not vary from execution to execution* (according to physical access invariance the data required for transaction re-execution is prefetched as a result of the first execution phase, such that the buffer is primed when the transaction is rerun [12], [14]). A high degree of access invariance is to be expected in a system with short and preplanned transactions postulated in this study, which usually access the same set of objects in repeated executions. The execution of a transaction in the second phase is much shorter than the first phase, since disk accesses are obviated when access invariance prevails, such that there are very few transactions in the second phase and, hence, little data contention among such transactions. The second phase will thus lead to transaction commit with very high probability.

We are concerned here with a special case of two-phase processing, referred to as the **hybrid optimistic** method, with OCC in the first phase and locking in the second phase. The first phase is based on the **optimistic die** [12], [14] or silent commit option [38], such that a transaction is allowed to proceed to its validation (although it may be already known that its validation will fail), as opposed to the **optimistic kill** [12], [14] or broadcast commit [38], which results in an immediate abort (and restart) of a conflicted transaction when the data read by the transaction is invalidated. A restarted transaction may be run again optimistically (with the optimistic kill policy), but this may result in additional restarts. Rerunning a transaction with a 2PL policy will limit the number of transaction re-executions to one, because of the very low frequency of deadlocks associated with 2PL. In fact, deadlocks can be avoided altogether by preclaiming the locks required in the second execution phase, since the identities of objects accessed by the transaction have been determined in the first phase [12], [14]. Simulation results in [12], [14] show that the CC method used in the second phase has little effect on overall performance. The present paper describes a method which permits an efficient use of the latter property in a distributed environment.

A key measure of the success of the proposed method is its relative performance compared to 2PL, which is demonstrated in Section 4. An experiment to compare OCC and 2PL using the Cm* experimental system was reported in [35]. This experiment was inconclusive in that both CC methods achieved a similar performance. This was due to a system-specific bottleneck, which resulted in the maximum transaction throughput being attained at a rather low

degree of concurrency. Another experiment indicated the superior performance of distributed OCC with respect to distributed 2PL, but was limited to two nodes and the results were influenced by the fact that I/O constituted a bottleneck [24]. A simulation study comparing the performance of 2PL with OCC with centralized validation in a distributed memory system was discussed in [4]. It is interesting to note that most simulation studies of CC methods have dealt with the fully replicated databases (see e.g., [7]).

A simulation study was undertaken as part of this effort, although an approximate analysis based on the approach in [38] (for a centralized system) is feasible. The analysis in [38] allows multiple transaction classes (depending on the number of locks requested) and optimistic die and kill policies. The problem with an approximate analytic solution is that:

1) It requires simplifying modeling assumptions to make the analysis tractable. This is because analytic solutions for the queueing network model for the underlying computer system are only available under rather stringent modeling assumption, e.g., exponential service times for FCFS queues [27].
2) Modeling multiple transaction classes introduces additional complications.
3) We are interested in the peak throughput attainable by the various CC methods under consideration, but most analytic solutions for OCC tend to be less accurate at very high-data-contention levels, which are of interest here. For example, the blocking effect due to static locking, which results from lock preclaiming can only be estimated by a rather elaborate analytic solution [45].
4) A simulation of the system would be required in any case for validating the approximate analytic solution.

The system under consideration and the hybrid OCC method is described in the next section, followed by a more formal description of the OCC method in Section 3. In Section 4, we describe the simulation model and compare the performance of 2PL and the new OCC method. In Section 5, we outline several variants of the proposed OCC method and address the issue of access variance. Conclusions appear in Section 6.

2 System and Transaction Processing Model

Though our protocols are, in principle, applicable to a wide range of data-partitioned systems (including distributed databases), we restrict our discussion to locally distributed systems, referred to as shared-nothing architectures, which are equipped with a high-speed interconnect for internode communication. Such systems are good candidates for high-volume transaction processing, see e.g., [19].

Data allocation is an important consideration in a data-partitioned system. Data allocation may be done to achieve a high degree of locality of data access in systems with prespecified transaction classes, where the database access characteristics of different transaction classes are known a priori. The database can then be partitioned to increase locality as well as to achieve a relatively balanced load among the nodes of the system (see e.g., [48]). Transactions are routed according to their class to the node which holds most of the data required for their processing. This results in reduced internode communication, shorter transaction response times, and reduced CPU overhead, since message costs can be significant [17]. Data allocation can be done solely to balance the load, as is done using hashing mechanisms in some shared-nothing database machines [20]. This reduction in the locality of access is of less consequence in a system with low cost messages and a broadcast capability. In the simulation studies reported in Section 4, we vary the number of nodes and the degree of locality of reference to determine the effect they have on performance.

We postulate a transaction execution model in which a transaction is associated with a **primary node**. Basically two approaches can be used to access data referenced by a transaction, which is not available locally. According to the **function request** approach, the request is processed at the node where the object resides. If necessary a subtransaction or cohort process is invoked to handle the function request at the remote node to which the object belongs. According to the **data-request** approach, the primary node requests an object from the node which owns it and the object is sent to the requesting node for processing. It is postulated in this study that objects correspond to database pages. The data-request approach was reported to provide better performance than the function request alternative when a high-communication bandwidth is available in [48]. This is because function requests allow little flexibility for transaction routing, since a node must process all operations against its database partition, while in the case of data requests most of the processing associated with a transaction is carried out at its primary node. Although the proposed CC method is applicable to both approaches, in this paper we will concentrate on the data-request approach, because of its potential for load balancing and the additional advantage pointed out below of caching data when a transaction is restarted. A similar discussion is applicable to a client-sever architecture, whether the processing of the data is done at the server or the server sends appropriate pages to the client [9].

A transaction is processed in three phases, according to the OCC method:

- a **read phase**,
- a **validation phase**, followed immediately by
- a **write phase**,

if the validation is successful. During the read phase the transaction reads all objects required for its execution. References to remote objects may be handled according to the function request or data-request approach. Updates are done only to a private copy of the object assigned to the transaction at the primary node of transaction execution (resp. the node which owns the object) in the data request (resp. function request) approach. In the validation phase (elaborated on below for the distributed case), a check is made to verify that none of the objects accessed by a transaction has been modified since it was accessed. If the validation is successful, in the following write phase objects updated by the transaction are externalized, i.e., written

onto the global buffer. A transaction failing its validation is restarted. The re-execution of a transaction, with respect to access to remote objects is more efficient than its execution in the first phase. This is because, according to the data-request approach, *remote objects are already available at the primary node of transaction execution and a fresh copy is required only for modified objects*. This is similar to the prefetching effect (from disk) associated with the two-phase processing method described in [12], [14]. When transaction restarts occur, this constitutes the main advantage of the data request compared to the function request approach. Note that a locking method cannot take advantage of the prefetching feature since lock requests need to be sent in any case to the node which owns the object.

In fact, a data-partitioned system may allow for remote as well as local caching of database objects. Thus, in the case of OCC a data request to the owner system is only required in case the object is not cached locally. Note that in the case of 2PL, a message requesting an appropriate lock on the object should be sent to the primary node, although a copy of a remote object may be cached locally. Remote caching is particularly attractive for data which is used mainly in read-only mode and may be enhanced by providing cache coherence mechanisms (as in data-sharing systems [33]), for example, the owner site may request the invalidation of data as part of the commit of a transaction. This approach will increase the chances for a successful validation, but is not required for the correct execution in the case of OCC methods. In fact, we will utilize special messages in conjunction with the optimistic kill policy (see Section 5.1) to notify a transaction that it has been conflicted.

In the **distributed validation method**, a transaction generally validates at all nodes which were involved in its read phase, i.e., nodes which control the partitions that were accessed by the transaction. As a consequence a transaction can be processed without any intersystem communication when it has referenced only local data objects being stored at its primary node. For **global transactions** (i.e., transactions that have referenced multiple partitions) validation and write phases can be integrated into the two-phase commit protocol (required to ensure the atomicity of the transaction) in order to avoid additional messages:

- At EOT when all database operations of the transaction have been executed, the (transaction manager at the) primary node of transaction execution acts as a **coordinator** for commit processing and sends a PREPARE message to all nodes involved in the execution of the transaction (after logging a **prepare** or **precommit** record). This message is now also used as a **validation request** and to return the modified database objects of external partitions to the owner systems. Upon receiving this message, a node performs local validation on behalf of the requesting transaction, where it is checked whether or not local serializability is affected. If local validation is successful, the modifications of local database objects and a **precommit** or **ready** record are logged and an O.K. message is sent to the coordinator node. Otherwise, a FAILED message is returned and the node forgets about the transaction.
- The second phase of the commit protocol starts after the coordinator node has received all response messages. If all local validations were successful, a **commit** record is logged and COMMIT messages are sent to the nodes participating in the commit protocol. The COMMIT message processing at a remote system consists of writing a commit log record and updating the database buffer with modified objects (write phase). If any of the local validations failed, an ABORT message is sent to the nodes which voted O.K. and the transaction is aborted by simply discarding its modifications.

This basic strategy alone does not ensure correctness, since local serializability of a transaction at all nodes does not automatically result in global serializability (e.g., a transaction may precede a second transaction in the serialization order at one node, but not another) [5]. An easy way to solve this problem is to enforce that **the (local) validations of a global transaction are processed in the same order at all nodes**. In this case, the local serialization orders can be extended to a unique global serialization order without introducing any cycles. The global serialization order is thus given by the validation order.

In a local environment with a (reliable) broadcast medium, it is comparatively simple to ensure that validation requests are processed in the same order at all nodes. Here, a multicast message is used to send the validation request (including a message to the primary node itself in case it holds data accessed by the transaction) and these requests should be processed in the order they are received. There is no distinction between local and remote validations. Other strategies which are more generally applicable use unique EOT timestamps or a circulating token to serialize validations [33].

Another difficulty for distributed database systems is the **treatment of precommitted database objects**, i.e., modifications of a precommitted, but not yet committed transaction. Here, basically three approaches can be pursued:

1) The conventional approach would be to ignore the fact that a precommitted object copy exists or is in preparation (a transaction which is being rerun is computing a new value for the object) and to access the unmodified object version. This, however, leads to the abort of the accessing transaction in the case when the precommitted transaction is successful, which is highly likely when the level of data contention is low. Note that the modifications of the precommitted transaction must be seen by all transactions which are validated later.
2) A more optimistic approach would be to allow accesses to precommitted modifications, although it is uncertain whether or not the locally successfully validated transaction will succeed at the other systems too. The problem with this approach is that a domino effect (cascading aborts) may be introduced since uncommitted data is accessed. In any case, one has to keep track of the dependencies with respect to

precommitted transactions and to make sure that a transaction cannot commit if some of the accessed database modifications are still uncommitted. In addition, locks may be held by a transaction which is being rerun because it failed its validation, such that precommitted data is not available. Option 2 will not be considered further in this paper.

3) It seems best to block accesses to precommitted objects until the final outcome of the modifying transaction is known. In general, these exclusive locks are only held during commit processing and are released in phase 2 after transaction commit. This approach is attractive since it is expected to result in a reduction in the number of transactions failing their validation. On the negative side, similarly to 2PL, it incurs transaction blocking.

The effect of options 1 and 3 (referred to hereafter as **OCC1** and **OCC2**, for short) on performance is investigated further in this paper.

In order to solve the starvation problem associated with other OCC methods, we make extensive use of locking by requesting locks for all objects (not only for modified ones) at EOT before the validation. If the validating transaction is successful, these locks are held only during commit processing. If the transaction validation should fail, the locks are retained during the re-execution of the transaction and guarantee a successful second execution, at least if no new objects are accessed. With this technique, starvation can be avoided for typical transaction processing applications.

For lock acquisition we distinguish between read (shared) and write (exclusive) locks. Validation for the optimistic die policy for the first phase is performed by using timestamps associated with objects, by checking whether the object versions seen by a transaction are still up-to-date (an alternate method based on access locks is described in Section 3.2). *This is not automatically ensured by a successful lock acquisition since locks are requested after the object accesses in our method, so that unnoted modifications by committed transactions (for which the locks have already been released at validation time) may have been performed.*

Fig. 1 shows the various phases during the execution of a global transaction for a successful first execution (Fig. 1a) as well as for the case of a validation failure (Fig. 1b). As indicated in Fig. 1, commit phase 1 consists of a lock request and validation phase, followed by precommit logging in the case of a successful local validation.

Irrespective of whether or not the local validation was successful, locks are requested for all data items accessed and the O.K. or FAILED message is only returned after all locks are acquired. If all local validations were successful, commit phase 2 is started consisting of the write phase and the release of all locks (Fig. 1a). If validation at any node failed, the transaction is re-executed under the protection of the acquired locks. If no additional objects are accessed in the second execution phase, the transaction can be immediately committed at the end of its second read phase before the write phases and the release of the locks are performed at the respective nodes. It follows from Fig. 1 that the number of messages for commit processing of failed transactions that reference the same objects during re-execution, is the same as successfully validated transactions. The issue of access variance is discussed further in Section 5.2.

Given the transaction execution model adopted in this paper all information for the recovery of a transaction can be logged at the transaction execution node, i.e., the precommit logging at remote nodes is not required since the data is already logged at the coordinator node. This latter protocol is similar to the **coordinator log** protocol described in [39], which is, however, proposed for a function request paradigm. In other words, a full-blown two-phase commit protocol is not required. The precommit at remote nodes after a successful validation is beneficial since it alleviates

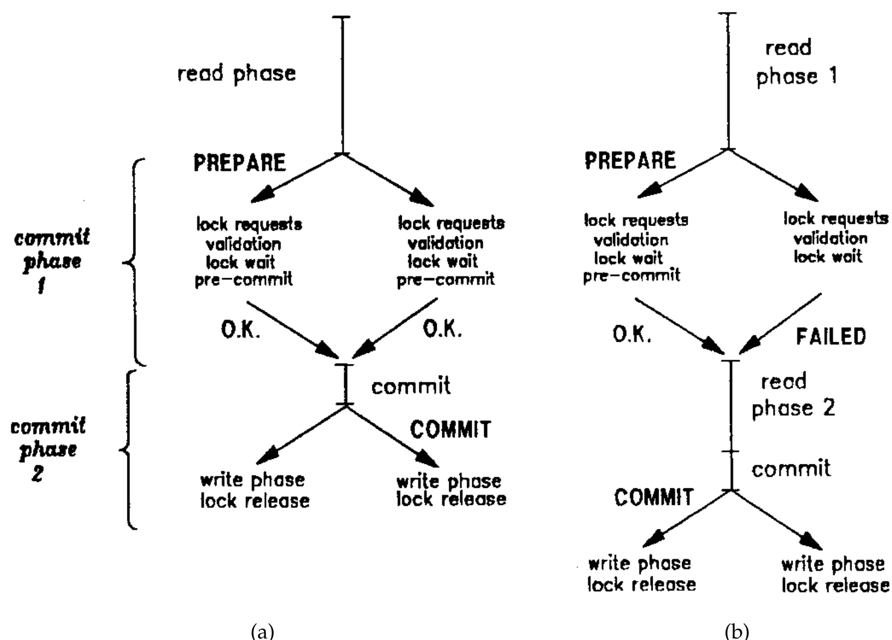


Fig. 1. Transaction execution flow for: (a) successful validation and (b) failed validation.

the need to transmit modified pages. When a transaction failing its validation is re-executed, the modified pages must be transmitted again, since their value has possibly changed. Since given access-invariance, the execution in this phase is guaranteed to be successful, a simplified commit protocol which involves logging at the execution node and sending the modified pages to remote nodes is sufficient. This is because in the case of the failure of a node, the transaction coordinator log can be used for the purpose of recovery [39].

3 Implementations for the Hybrid OCC Scheme

In Section 3.1, we provide a rather detailed procedural description of the proposed protocol, which unless otherwise specified is based on object timestamps. An implementation using access locks is then described in Section 3.2. The first implementation can only be used in conjunction with the optimistic die policy (in the first phase), while the access lock method is more flexible and can be used to implement both optimistic die and kill policies, as well as option 3 (or OCC2) described in Section 2.

3.1 Implementation Using Object Timestamps

The identifiers of all objects accessed and modified by a transaction T are denoted as its **read set** RS(T) and **write set** WS(T), respectively. Every system maintains a so-called **object table** to process lock and validation requests for objects of its partition. For this purpose, the object table entries keep the following information:

OID: ...	**{object identifier};**
WCT:	**integer {write counter};**
XT:	**exclusive lock holder transaction;**
ST:	**shared (read) lock holder transactions;**
WL:	**waiting list for incompatible lock requests;**

WCT is a simple counter which is incremented for every successful object modification and is stored with the object itself (e.g., database pages) as well as in the object table. The WCT value in the object table always refers to the most recent object copy, while the counter value within a given object copy indicates the version number (or timestamp) of this copy. The WCT field is used during validation to determine whether or not the object copies accessed by a transaction are still valid.

Locks are held either by precommitted transactions or by already failed transactions during their second execution. An X-lock indicates that the transaction holding the lock is going to modify the object (at commit time). In order to prevent unnecessary rollbacks, according to the OCC2 method, we delay object accesses during the read phase until an X-lock is released. Also, an X-lock results in the abort of first phase transactions that have accessed the unmodified object version (before the lock was set). Read (R) locks are set for accessed objects which have not been modified. Though these locks are not required for a correct synchronization, they prevent the object from being updated (invalidated) by other transactions. Thus, they guarantee a successful re-execution for a failed transaction, provided it accesses only its locked objects. The need for an additional validation is thus obviated. Incompatible lock requests are appended to the WL waiting list according to the request order.

The execution of a transaction T at its primary node is described below. It is assumed that a transaction performs a sequence of read or write accesses on database objects. Furthermore, it is assumed that every object is read before it is modified (no "blind writes"), i.e., the write set of a transaction is a subset of its read set.

```
{read phase of first execution, OCC1 is the default policy}
for every object O to be accessed do;
   if O belongs to local partition then
      if (OCC2 and (X-lock set or X-lock request is waiting
            for O))
         then
            queue A-lock request into WL;
            wait until conflicting X-locks are released;
         else
            perform object access;
            if OCC2 then append A-lock;
               else
               record O with current version number in RS(T);
               if write access then also add O to WS(T);
            {perform modification on private copy of O}
      end if;
   else
      perform remote object access;
         {wait at remote system when conflicts with
         X-locks arise}
   end if;

   {commit phase 1}
   broadcast validation request to remote systems
         involved during T's read phase;
   local lock acquisition and validation of T; {for details,
         see below}
   receive validation results from remote systems;

   {commit phase 2}
   if (all validations successful) then write commit record;
      broadcast COMMIT to all participating systems;
   else
      (* re-execution *)
      restart and re-execute transaction;
      if (no new objects referenced) then write commit record;
         broadcast COMMIT to all participating systems;
      else
         perform two-phase commit with validation;
      end if;
   end if;
```

Next, we describe the **first commit phase** including lock acquisition and validation of a transaction T at system S. Part of the processing has to take place within a critical section (indicated by << ... >>) against other transactions which are ready to validate. RS(T, S) and WS(T, S) denote the objects of RS(T) and WS(T), respectively, belonging to the database partition of S. With wct(x, t) we denote the version number of the copy of object x as seen by transaction T.

In fact, the information pertaining to the read-set and write set of a transaction, including wct(x, t) can be maintained as part of access locks.

```
<< VALID := true;
   for all O in RS(T, S) do;
      if (X-lock set or X-request is waiting for O)
         then VALID := false;
      if (lock conflict for O)          then
      if O in WS(T, S) then queue X-request into WL;
                        else queue R-request into WL;
   else              {no lock conflict}
      if O in WS(T, S) then XT := T            {acquire X-lock};
                        else append T to ST list {acquire
                           R-lock};
   end if;
      {validation using timestamps, rather than access
         entries}
      if wct(O, T) < WCT(O) then VALID := false;
   end for; >>

   if VALID
   then
      wait (if necessary) until all lock requests at S are granted;
      write log information; {precommit}
      send O.K.;
   else
      wait (if necessary) until all lock requests at S are granted;
      send FAILED;
   end if;
```

It is to be noted that all locks for the read and write set elements are requested within a critical section, even if lock conflicts occur for some requests or the transaction is to be aborted. This guarantees that deadlocks cannot occur since all locks are requested atomically with respect to other transactions,

1) *all locks required by a transaction are requested in a serialized manner*,
2) *the lock request/validation phases are processed in the same order at every node*.

As a measure of precaution, we even request read locks before validation although they are only needed to achieve the preclaiming effect for failed transactions. If read lock requests were deferred until after the global validation result (abort) is known, deadlocks would be possible. Also, requesting these lock requests separately could result in additional communication overhead.

Although we request all locks before the validation, it is to be emphasized that the waiting time for conflicting lock requests as well as the logging delays occur after the validation and are not part of the critical section. This is important because otherwise transaction throughput could be seriously limited, since the validations are to be performed in the same order at every pertinent node. Therefore, a delay in the critical section at one node would delay all other validations. The use of timestamps in fact allows a very efficient validation with just one comparison per write set element.

The procedure shows that a transaction T is aborted either if validation fails, i.e., if some of the accessed object copies have been modified (invalidated) in the meantime, or if such a modification is planned by a previously validated update transaction. The latter is indicated by the fact that another transaction has already requested an X-lock for one of T's read set elements. However, not every lock conflict results in the abort of the transaction making the lock request. For instance, when T requests an X-lock and only R-locks have been granted to other transactions and no other X-requests are pending, then T is not aborted but waits until the release of the read locks before returning the O.K. message to the coordinator. X- and R-requests result in a transaction abort in lock state X and the failed transaction waits for the respective lock before being reexecuted. In lock state R (only read locks granted, no waiting X-requests), an X-request does not result in the abort of transactions requesting the lock, but only in a lock wait.

If all local validations are successful, the coordinator commits the transaction by writing a commit record to its log file. In the **second commit phase**, the coordinator sends a COMMIT message to all pertinent systems. Upon receipt of the COMMIT message for transaction T, the following steps are performed at system S:

```
write commit record for T to log file;
copy objects modified by T into database buffer;
<< for all (O in WS(T, S)) update WCT(O) in object
   table;
release T's locks;
{if possible, activate waiting transactions in WL} >>
send ACK to coordinator;
```

Naturally, the write phase and lock release are also performed at the coordinator system for T. The coordinator must keep the commit information for T until all systems have acknowledged processing of commit phase 2.

If the local validation of a transaction fails at a system, the FAILED message is returned after the transaction has acquired all of its locks at this node. The re-execution of a failed transaction is started as soon as it has acquired its locks at all pertinent nodes. Modified data pages are sent along with lock acquisition notifications. If no new objects are referenced, the second execution may be performed without any I/O delays if all objects can be held in the main memory of the computer system at which the transaction is in execution. As a result, the re-execution of a transaction should usually be much faster and cheaper than its first execution supporting short lock holding times. If no new objects are accessed during re-execution, the transaction's success is guaranteed without further validation. Therefore, the coordinator can write the commit record right after the second read phase and start the second commit phase as described above.

Caching of remote data is particularly beneficial to the OCC method since it permits a *re-execution of a transaction without any remote data requests*, if the same data is accessed as in the first execution. Thus, not only all disk I/Os but also all remote requests may be avoided for failed transactions thereby supporting very fast re-execution. To utilize this idea, however, we have to make sure that a transaction

always sees the most recent object versions during re-execution in order to guarantee a successful execution. This can be achieved without extra messages by using the FAILED message after an unsuccessful validation at a system to transmit the current version of all read set elements to the transaction's primary system before re-execution. Also, when a transaction modifies remote objects, the new object versions have to be returned to the owner system. To avoid extra messages for this, we can send these modifications back together with the validation request after the first read phase or together with the COMMIT message after the second (successful) read phase. In the write phase, these modifications can then be logged and made visible at the owner system before the write locks are released.

3.2 Implementation Using Access Locks

Access locks may be used to defer object access in case exclusive locks are being held on the object (option 3 or OCC2 affecting the read phase of first execution described in Section 2) and in implementing the optimistic kill policy [12], [14] (see Section 5.1). It should be emphasized that access locks obviate the need for maintaining object timestamps and can also be used with the optimistic die policy.

During the optimistic read phase a transaction does not request regular R- and X-locks, but it has to wait for X-locks, according to OCC2. To detect these situations, special Access- or A-locks are requested during the read phase and inserted in the waiting list in the case there is an X-lock granted or already waiting (note that A-locks are always posted for the optimistic kill policy). These A-locks are only used in this case to activate transactions after the release of the conflicting X-locks, but they are not granted or released like regular locks (in particular, X-lock or R-lock requests are never denied because of A-lock requests). The compatibility of the various lock modes is summarized in Table 1. A-lock requests only have to wait in lock state X indicating that an X-lock is already granted or waiting. R- and X-requests can be granted in lock state NL (no locks granted) or A (the object is being accessed by transactions in their read phase).[1]

It is appropriate to point out an implementation detail regarding the validation of transactions in the presence of access locks. Upon the commit of a transaction, these locks

TABLE 1
LOCK COMPATIBILITY MATRIX

	current lock state			
	NL	A	R	X
A	+	+	+	wait
R	+	+	+	abort and wait
X	+	+	wait	abort and wait

1. As a matter of fact, we do not distinguish between the lock state NL and A in the object table since we do not record granted A-locks. This is necessary, however, in the case of the optimistic kill policy or as a replacement to using timestamps to detect deadlocks as discussed below.

may be used directly to determine transactions which have accessed an object and should fail their validation since the object is being updated. This approach can be used in conjunction with both the optimistic die and kill policies, where in the latter case the information that the data read by the transaction has been invalidated is used immediately (see Section 5.1), while in the latter case this is deferred to the validation point of the transaction.

4 PERFORMANCE COMPARISON WITH STANDARD LOCKING

Our comparison will concentrate here mainly on performance aspects, since we are primarily interested in the relative suitability of the protocols for high-performance transaction processing. In terms of fault tolerance, the new OCC method is as robust as distributed 2PL [5], since it mainly depends on the robustness of the commit protocol required in both methods. The deadlock freedom of our protocol considerably simplifies the complexity of an actual implementation.

The relative performance of OCC and 2PL is quantified in this section using a simulation study. In Sections 4.1, 4.2, and 4.3, we describe the simulation model for:

1) the multicomputer system,
2) the partitioned database, and
3) transactions.

Simulation results are reported in Section 4.4

4.1 The Multicomputer System Model

The system model and the settings for the simulation parameters are as follows:

1) **Multicomputer system configuration**. We consider two cases with $N = 4$ and $N = 8$ computer systems, where each computer system consists of a four-way tightly coupled multiprocessor. The total processing capacity in the system is varied to study this effect on the relative performance of 2PL and OCC methods. When $N = 4$, we consider three cases with 100, 200, and 400 MIPS four-way multiprocessors (or 25, 50, 100 MIPS per processor, respectively). When $N = 8$, we consider 50, 100, and 200 MIPS four-way multiprocessors (or 12.5, 25, 50 MIPS per processor, respectively). Thus, we have three sets of systems with an equal total MIPS for comparison purposes.
2) **Intersystem communication**. The computer systems are interconnected by a high-speed network. A broadcast capability in the network would be beneficial in sending appropriate messages as discussed earlier (otherwise appropriate software methods should be adopted to achieve the FCFS processing of broadcast messages). The communication delay is assumed to be negligibly small. We take into account, however, the CPU overhead to send and receive individual messages, i.e., messages to different nodes are considered as separate messages at this level.
3) **I/O subsystem**. The I/O configuration, more specifically the number of disks per system which hold the

database is selected to match the corresponding CPU processing capacity, such that the ratio of CPU and disk utilizations, taking into account the database cache hit ratio (see below) and one transaction processing phase (no transaction reruns), is 75/20. This ratio which was also used in [14] is typical of transaction processing systems since much longer queueing delays can be tolerated at the CPU than the disks. Disk accesses are uniformly distributed (no skew).

4) **Database cache**. A database cache with a global LRU policy for caching local data is considered. This implies that objects are not cached remotely, i.e., nonlocal objects are purged upon transaction commit, but are retained in case a transaction is to be re-executed. Comparative results obtained in this study are, therefore, favorable to 2PL, since remote caching would result in improving the performance of read-only queries with the OCC protocol. High-contention items (see Section 4.2) for local data are always in the cache, while the hit ratio for low contention items is $F_{DB_low} = 0.50$. The cache is large enough such that data referenced by an in progress or restarted transaction is not replaced before the transaction is committed. As far as objects accessed from remote nodes are concerned, these objects also are retained when a transaction is restarted, but are purged when a transaction is completed.

5) **Logging and recovery**. Nonvolatile (random access) storage is assumed to be available for logging, such that synchronous disk I/O for logging is not required. Logging time is, therefore, an order-of magnitude smaller than what would be required to write onto disk. This results in reducing lock holding time for both CC methods.

4.2 Database Access Model

The database model considered in this study is described below:

1) **Database objects**. We distinguish high- and low-contention data items based on their access frequency by transactions. In a system with $N = 4$ nodes, the effective database size for each category of data items at each system is $D_{high} = 1{,}000$ and $D_{low} = 31{,}000$, respectively. A fraction $F_{high} = 0.25$ (resp. $F_{low} = 0.75$) of all transaction accesses are to high- (resp. low)-contention items. High-contention data items, which are thus accessed roughly ten times more frequently than low contention items, determine the level of data contention. In order to maintain the same level of lock contention (for the same total number of transactions) in a system with $N = 8$ and $N = 4$ nodes, we need to maintain the same *effective database size* $D_{eff} = D/(b^2/c + (1-b)^2/(1-c))$ [40], where $b = F_{high}$ and $c = D_{high}/D$. When $N = 4$, we have $D_{eff} = 12{,}400$. Setting $D_{eff} = 6{,}200$ in the case of $N = 8$ yields $D_{high} = 436$ (and $D_{low} = 31{,}564$). The frequency of access to high- or low-contention objects is the same as for $N = 4$. Note that we have assumed that the effective database size does not increase with the degree of transaction concurrency (the increase in the degree of transaction concurrency may be considered to represent an increased arrival rate).

The overall cache hit ratio for a transaction executing for the first time is: $P_{hit} = F_{DB_low} \times F_{low} + F_{high} = 0.625$. This hit ratio also applies to nonlocal database accesses.

2) **Granularity of locking**. The data-request approach postulated in our study requires locking or time stamping data items (in the first phase of OCC) at the level of disk blocks or at the level of appropriately specified objects.

3) **Access mode**. Data items are accessed in exclusive mode, since we are interested in the relative performance of the two methods. Shared accesses would have resulted in a reduction in the data-contention level, i.e., given that the fraction of shared accesses is s would have resulted in an effective database size $D/(1-s^2)$ [40].

Read-only queries spanning a large number of pages in a database can introduce very high levels of lock contention if shared locks are held up to the completion point of a query. This is referred to as degree 3 isolation [18]. Lock contention is significantly reduced if locks are only held while the query is reading the corresponding page (degree 2 isolation). Queries requiring an approximation to the contents of the database are run with degree 1 or in browse mode without acquiring any locks. Another approach to reduce the lock contention due to locking is to use versioning. A simulation study characterizing this effect is [6]. More recently finite-versioning methods have been proposed to limit the space overhead associated with versioning methods [29]. In other words, special methods are used for handling read-only queries, which is an issue beyond the scope of this paper.

4.3 Transaction Processing Model

In this section we describe the characteristics of the transactions.

1) **Transaction "arrivals."** We consider a closed system with M transactions in each system (and $N \times M$ transactions in the complex), i.e., a completed transaction is immediately replaced by a new transaction at the same system.

2) **Transaction classes**. There are multiple transaction classes based on transaction size, i.e., the number of data items (n_c) accessed by a transaction in class c. Transactions are introduced into the system with frequencies f_c, $c = 1, ..., C$ according to what would be expected in a stream of arriving transactions. There are 17 transaction classes with sizes uniformly distributed in the range (8, 24) with a mean transaction size of 16.

The variability of transaction size has a significant effect on performance. This issue has been investigated in the case of 2PL (resp. OCC) methods in [41], [43] (resp. [38]). Another complication is that *variable size transactions tend to access data at a fewer number of*

distinct remote nodes than fixed size transactions with the same mean size on the average [42]. The implication is that fewer nodes need to be involved in the two-phase commit protocol in the case of fixed size transactions as compared to variable size transactions (more details regarding this issue appear in [42]).

3) **Transaction processing stages.**

a) **Transaction initialization**. This requires CPU processing only and the path-length for this stage is I_{init1} = 100,000 instructions. If the transaction is restarted due to failed validation or having been selected the victim for deadlock resolution then I_{init2} = 50,000.

b) **Database processing**. There are n steps in this stage, corresponding to the number of data items accessed from the database (from local or remote partitions). We consider two cases. In the first case, each transaction is routed to a system at which it exhibits a high degree of locality, i.e., the fraction of local accesses at each system is $F_{local} = 0.75$, while the remaining $1 - F_{local}$ accesses are uniformly distributed over the remaining systems. We also consider the case when accesses to database objects are uniform over all N nodes in the system, i.e., $F_{local} = 1/N$.

A data item may be available in the database cache in which case the path-length per data item is I_{cache} =20,000. This includes the overhead for concurrency control. Otherwise when data has to be accessed from disk, an additional I_{disk} = 5,000 instructions are required (the processing required to retrieve cached data is considered to be negligible). It takes I_{send} = 5,000 instructions to send and receive a message. Therefore, 20,000 instructions are executed for intersystem communication to access remote data.

c) **Transaction completion**. The CPU processing in this stage requires $I_{complete}$ = 50,000 instructions. In case a transaction has accessed local data only, it commits without requiring a two-phase commit (after local validation in the case of OCC). Commit processing requires I_{commit} = 5,000 instructions to force a log record onto stable storage.

In case multiple systems are involved in processing a transaction with 2PL, as part of two-phase commit $I_{precommit}$ = 5,000 instructions are executed at the primary node of transaction execution (mainly to write a PRECOMMIT log record). There is also a per system overhead of I_{send} and $I_{receive}$ to send and receive PRECOMMIT messages. Precommit processing at secondary nodes from which data was accessed requires I_{remote} = 5,000 instructions, which includes writing PRECOMMIT records. Each remote system after forcing modified data onto stable storage sends an ACK message in the case of 2PL to the primary system, which in turn sends a COMMIT message to all of the nodes involved after forcing a commit record onto the log. All systems release their locks at this point.

The processing in the case of OCC is more complicated as explained before. If transaction validation is unsuccessful at any node, it is re-executed at the primary node after the required data has been locked and an up-to-date copy of all modified or invalidated data has been made available to the transaction. We have used the same figures for 2PL and OCC, although it is expected that the private workspace paradigm for OCC would require additional processing. The conclusions of this study hold with longer path-lengths for OCC, except that faster processors would be required for OCC to outperform 2PL.

4.4 Simulation Results

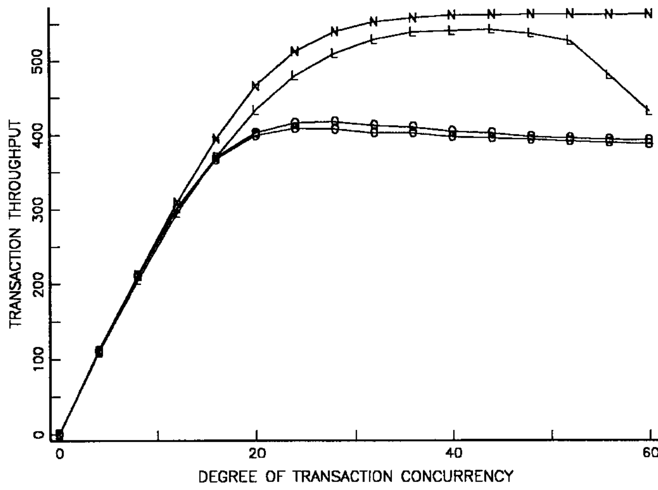
A discrete-event simulation program was written to compare the performance of 2PL and the OCC method with optimistic access (referred at OCC1) and the OCC method with deferred access to locked objects (referred as OCC2). The overall system throughput for all N systems is the performance measure of interest in comparing the distributed 2PL and the new OCC methods. Due to symmetry the throughput at each system is $1/N$ of the overall throughput. Furthermore, due to conservation of flow, the throughput for class c transactions is a fraction $f_c(= 1/17)$ of the overall throughput. Due to space limitations we do not report mean response times for individual transaction classes. In fact, in the case of 2PL, the expansion in the mean response time is proportional to transaction size [41], [43]. This is not so for long transactions at high-lock contention levels, since the susceptibility to encounter deadlocks increases with the fourth power of transaction size [46]. In the case of the hybrid optimistic policy, the probability that a transaction fails its validation at the end of its first phase increases with the square of transaction size, which is referred to as the *quadratic effect* [13], [14],

To quantify the effect of data contention on system performance, we consider a situation when there is No Data Contention (NDC), e.g., we have 2PL or OCC with all accesses in shared mode and system performance is solely determined by hardware resource contention.[2] Given in Figs. 2-5 are transaction throughputs (in transactions per second) versus the per system degree of transaction concurrency (M), for the four cases:

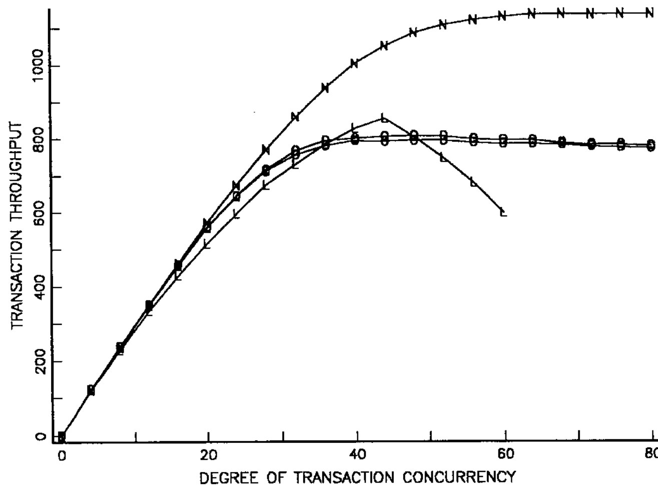
System 1. Four node system with locality of access.
System 2. Eight node system with locality of access.
System 3. Four node system with uniform access.
System 4. Eight node system with uniform access.

Figs. 2a-2c in each case correspond to a system with a total processing capacity of 400, 800, and 1,600 MIPS, respectively. In each case, we plot the overall system throughput for the NDC, 2PL, OCC1, and OCC2, methods (represented by the letters N, L, O, and Q, respectively) versus the degree of transaction concurrency at each node (the total number of transactions in the system is $N \times M$). Each point

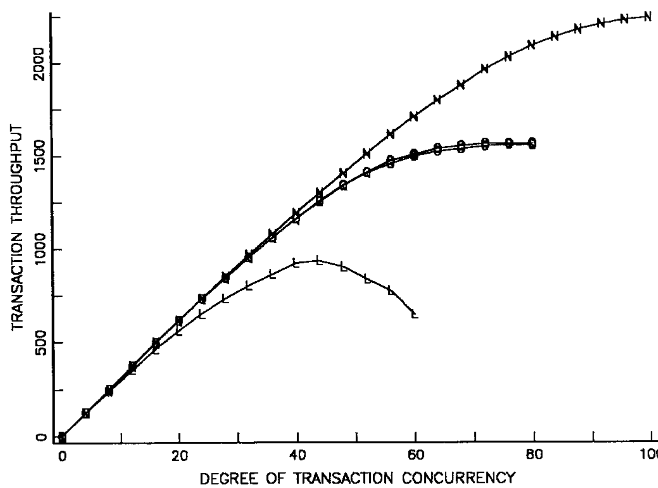
2. For the sake of a fair comparison, we do not consider the usual optimization possible with read-only accesses and carry out a two-phase commit protocol at all nodes at which objects were referenced.



(a)

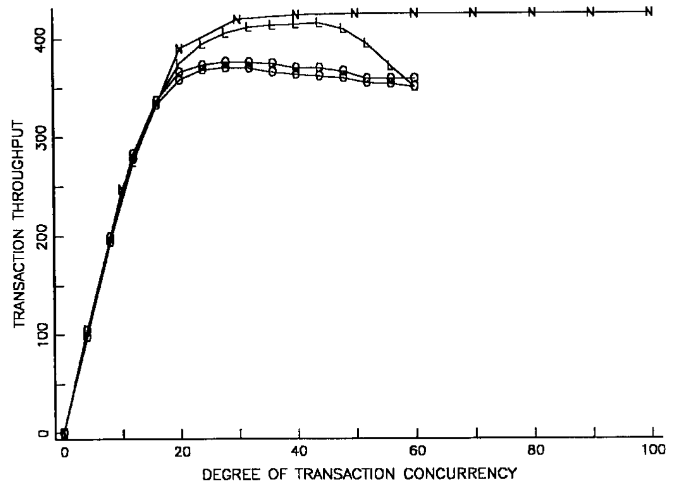


(b)

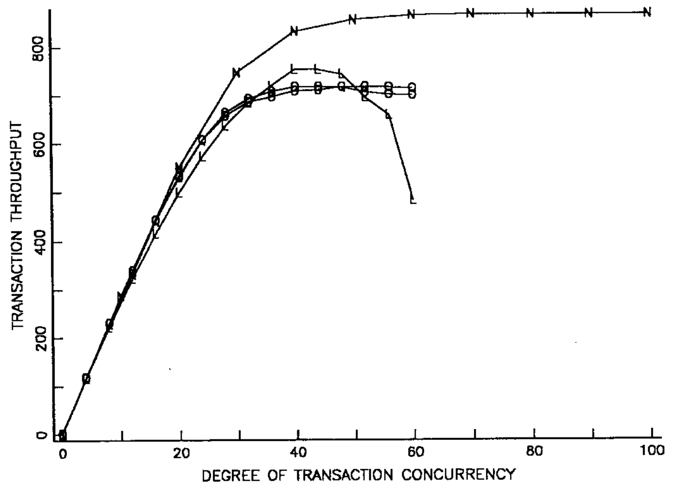


(c)

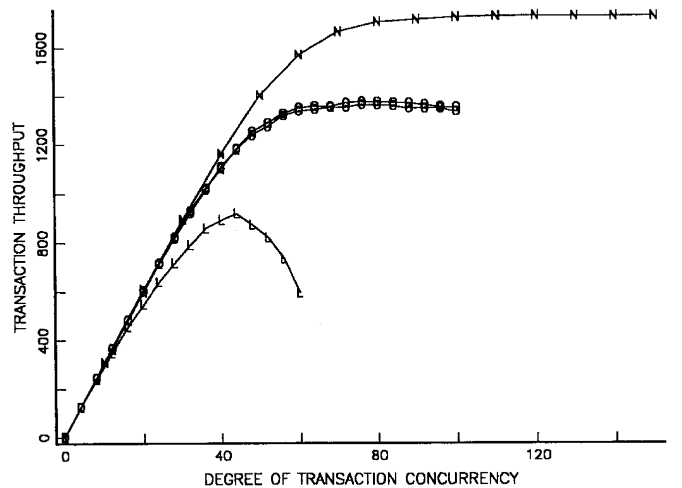
Fig. 2. Throughput versus degree of transaction concurrency characteristics with four nodes and locality of access: (a) 400 MIPS system, (b) 800 MIPS system, (c) 1,600 MIPS system.



(a)

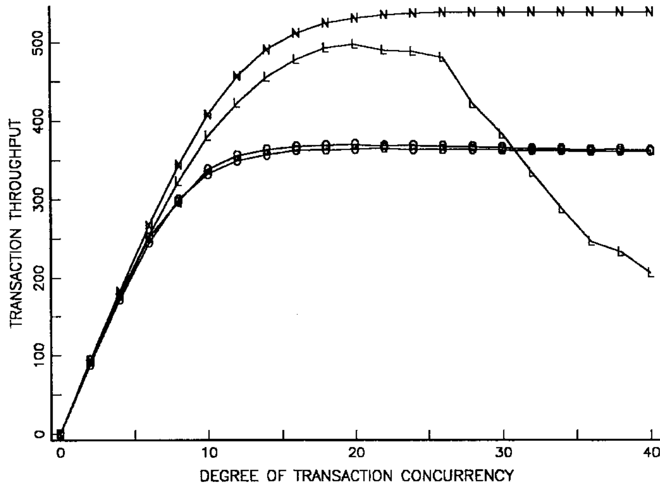


(b)

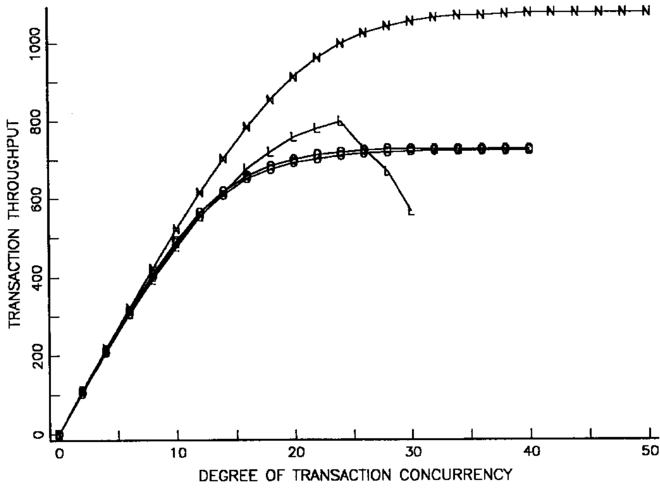


(c)

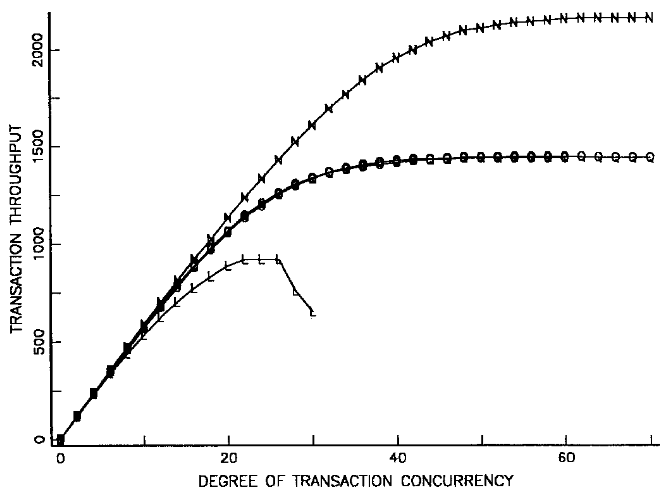
Fig. 3. Throughput versus degree of transaction concurrency characteristics with eight nodes and locality of access: (a) 400 MIPS system, (b) 800 MIPS system, (c) 1,600 MIPS system.



(a)

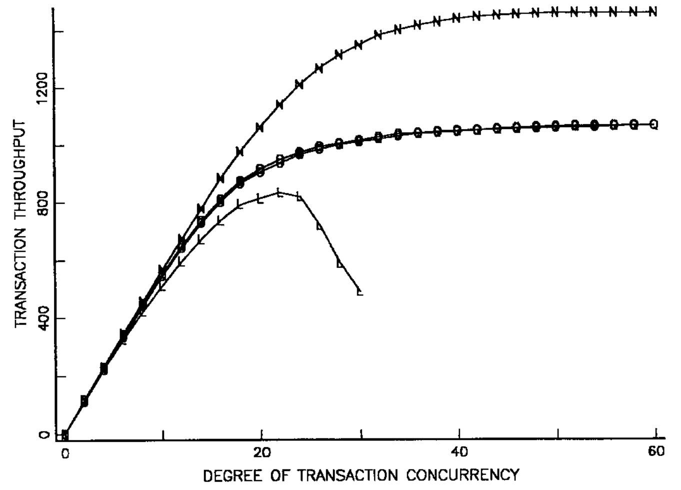


(b)

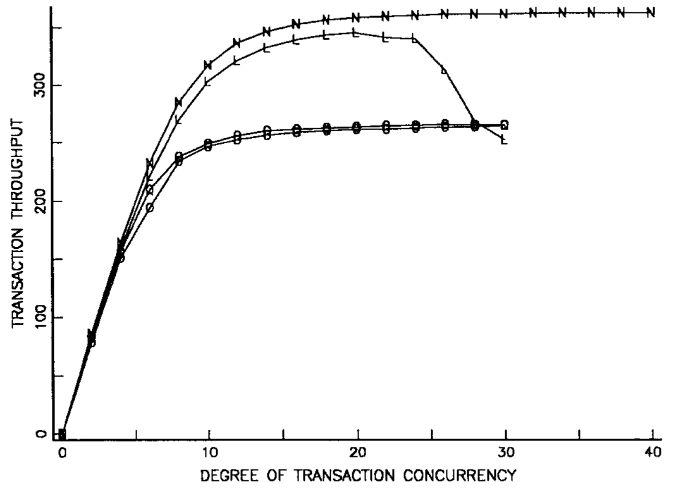


(c)

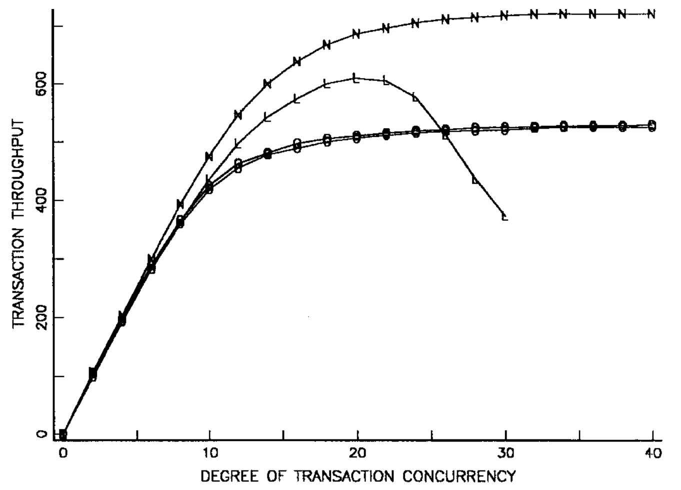
Fig. 4. Throughput versus degree of transaction concurrency characteristic with four nodes and uniform accesses: (a) 400 MIPS system, (b) 800 MIPS system, (c) 1,600 MIPS system.



(a)



(b)



(c)

Fig. 5 Throughput versus degree of transaction concurrency characteristic with eight nodes and uniform accesses: (a) 400 MIPS system, (b) 800 MIPS system, (c) 1,600 MIPS system.

on the graphs corresponds to the mean obtained from three runs, such that the system throughputs measured in different runs were within 5 percent of each other (with the exception of the thrashing region for 2PL).

In the case of NDC, as M is increased the system throughput ($T_{NDC}(M)$) increases initially and saturates beyond the point where the CPU is fully utilized ($T_{NDC}^{\max}$). Such a behavior is typical of a multiprogrammed computer system with an adequately large main memory. In this case, the throughput is determined by the utilization of the bottleneck resource (in our system, the processors, which follows from the assumption made in Section 4.1 regarding the relative utilization of processors and disks). Systems with fewer nodes and locality of access achieve the best performance, which can be ascribed to the fact that intersystem communication for remote data access and commit processing is minimized in this case. There is considerable degradation in performance as the locality of reference is reduced from 75 percent to uniform (for $N = 4$, $F_{local} = 0.25$ and for $N = 8$, $F_{local} = 0.125$). This effect is more severe as the number of nodes is increased, which is due to the increase in the number of accesses to other nodes as well as the number of distinct remote nodes accessed [42] (the latter factor affects the number of nodes to be involved in commit processing).

In the case of 2PL, the system throughput $T_{2PL}(M)$ initially follows $T_{NDC}(M)$ rather closely, since very few transactions are blocked and there is little wasted processing due to restarts to resolve deadlocks. As M is increased further, the number of blocked transactions increases gradually, but the wasted processing due to deadlocks remains small, such that $T_{2PL}(M) < T_{NDC}(M)$. A peak in transaction throughput is achieved, followed by a decrease in system throughput, which constitutes the **thrashing region** for 2PL [41], [43]. The maximum throughput attained by 2PL ($T_{2PL}^{\max}$) indicates the best performance attainable by 2PL for the given degree of data contention. There is the aforementioned degradation in performance due to reduced locality, especially when the number of nodes is higher.

The performance of OCC methods is determined by the fraction of transactions validated successfully in the first phase. This is because the additional processing incurred by transactions failing their validation, which are rerun in the second phase, results in a rather significant degradation in performance, especially in the case of lower MIPS systems. As the degree of transaction concurrency is increased, the CPU utilization due to transactions executing in the first phase is increased. There is also an increase in the degree of data contention and the processing incurred by transactions in the second phase. The maximum system throughput for the OCC methods is thus obtained at the point where the processors are 100 percent utilized, i.e., the blocking effect due to lock contention (including access locks) tends to have a second-order effect. Increasing the concurrency beyond this point results in a slight reduction in throughput, which is due to a decrease in the fraction of successfully validated transactions in the first phase.

Three options in accessing precommitted database objects were discussed in Section 3, at which point qualitative arguments regarding their relative performance were presented. Simulation results have shown that the performance attained by OCC2 is better than OCC1 for the considered cases. The improvement in system throughput, however, remains negligibly small, such that the throughputs attained by the two methods is not distinguishable (see Figs. 2-5). This can be attributed to the fact that although there is a reduction in the fraction of transactions which need to be rerun, the reduction in additional processing is in fact quite small, because the restart frequency for longer transactions remains high (quadratic effect in [13], [14]) and is not reduced. There is the additional effect of transaction blocking time due to access locks associated with this approach. Given the extra complexity associated with A-locks, a pure optimistic policy in phase 1 (OCC1 in Section 2) may be preferable.

The maximum throughput attainable by NDC, 2PL, OCC1, and OCC2 are summarized in Table 2. The following conclusions can be drawn from this table.

1) In a low MIPS system (400 in our case), there is a slight degradation in performance due to data contention when 2PL is in effect. The maximum throughput achieved by the optimistic methods is inferior to 2PL in this case.
2) In an intermediate MIPS system (800 in our case), the performance degradation due to data contention is significant. The 2PL method still outperforms optimistic methods in this case.
3) In a high MIPS system (1,600 in our case) there is a severe degradation in performance due to data contention. However, the performance achieved by OCC methods is significantly higher than 2PL, i.e., this is the region where the OCC method is preferable to 2PL.

TABLE 2
MAXIMUM THROUGHPUTS ATTAINED FOR DIFFERENT SYSTEM CONFIGURATIONS

	No Data Contention	Two-Phase Locking	Optimistic I	Optimistic II
400 MIPS				
4 nodes/L	563	542	408	418
8 nodes/L	539	497	365	370
4 nodes/U	426	416	373	376
8 nodes/U	362	345	264	266
800 MIPS				
4 nodes/L	1,137	853	794	807
8 nodes/L	1,080	797	722	728
4 nodes/U	867	753	717	724
8 nodes/U	723	609	529	532
1,600 MIPS				
4 nodes/L	2,266	1,009	1,555	1,556
8 nodes/L	2,168	919	1,441	1,442
4 nodes/U	1,732	916	1,363	1,375
8 nodes/U	1,467	832	1,066	1,071

L = 75% locality of access with remainder of accesses uniform.
U = uniform accesses.

Other performance measures of interest are the (per class) mean response times and device utilizations. In cases when there is no or little wasted processing (NDC and 2PL), the CPU utilization can be deduced simply as the product of system throughput and the mean processing time at the CPU. The CPU is 100 percent utilized beyond the point that the throughput achieves asymptotic behavior in the case of NDC. In the case of OCC, the CPU is 100 percent utilized at the peak system throughput and also beyond that point, but otherwise CPU utilization can be estimated directly from the simulation or indirectly from the fraction of transactions that fail their validation.

Transaction response times are of interest from two viewpoints:

1) that they are acceptably low, and
2) that they only increase proportionately to transaction size (and not the square of transaction size, for example).

A straightforward implementation of optimistic CC methods may result in an excessive number of restarts and long response times, but this is not a concern for the proposed hybrid OCC method since transactions may be restarted only once.

Simulation studies while varying other system parameters (besides the number of systems and locality of reference) such as:

1) the level of data contention, and
2) transaction load imbalance among the systems,

yielded results supporting our conclusions.

5 Alternatives and Extensions to the Basic OCC Method

In this section, we first describe the implementation of a scheme with deferred lock requests for transaction execution. We then discuss several options to cope with the case when access invariance does not prevail.

5.1 Deferred Lock Requests

As discussed in the Section 1, the adoption of the optimistic die (rather than kill) policy is motivated by the simulation results in [12], [14] for high MIPS systems, which have shown that in a system with access invariance the prefetching of data results in an improvement in performance compared to the optimistic kill policy. There are several cases when this is not so:

1) a system with a main storage database where no prefetching is required,
2) a system with slow processors and/or transaction requiring few disk accesses where the additional processing incurred by the die option cannot be tolerated, and
3) a system with no or little access invariance, in which case no benefit is to be gained by prefetching the data.

Approaches for handling of transactions for which access invariance does not prevail are discussed in the next section.

We propose an alternate method where the initial optimistic/die and the following optimistic/kill phases are followed by lock requests for validation *only if the transaction has not been notified at the completion of its execution that it has been conflicted.* The notification that a conflict occurred is preferably accompanied by the modified object. The restarted transaction need only execute with the optimistic/kill policy, because prefetching data is no longer an issue. Locking will be attempted only when a transaction has executed to completion with *no apparent conflicts*.

This approach has the disadvantage that a transaction may be restarted more than once, but it has the advantage of reducing lock holding times. An invalidate message might still be in transit, when locks are requested, such that absolute successful execution is never guaranteed. This is because deferring the requesting of locks in this manner increases the probability of a successful validation obviating transaction rerun, such that locks will be held for a shorter time period. In effect, there is the potential of an increased higher maximum throughput due to the reduction in lock holding times at the cost of additional processing which may be tolerable only in a system with "adequate" resources. This method is expected to outperform the hybrid OCC method for a narrow range of parameters, if not at all.

To avoid starvation and repeated restarts a transaction which is restarted a given number of times may switch to locking and preclaim its locks. A transaction which has requested locks will wait until it has acquired all locks. At this point it may still require re-execution because of the invalidation of one or more of the objects it requires for execution.

Access locks (A-locks) which were introduced in Section 3.2 are required for implementing the modified CC method, while they are optional for the optimistic/die policy (unless, according to option 2 in Section 2, accesses to objects locked in exclusive mode need to be deferred). An A-lock, identifying the transaction and its primary node, is posted for each accessed object at the node where the object resides. Note that this is done regardless of whether the object is locked or not. The CC manager at each system uses the access locks to keep track of transactions which have *interest* in the objects belonging to the system and these transactions are notified when a lock conflict occurs. In fact, in the case of remote caching of data, which was alluded to in Section 2, similar locks identifying the node (rather than transaction) can be used to invalidate the object in the node's buffer.

When a data item is updated by a committing transaction, then all transactions which had posted access locks for the data item are notified that they have been conflicted. Associated with the invalidation message is the newest copy of the data item (when a data item resides remotely).

The performance of this method can be extrapolated from previous performance studies of optimistic methods, especially [12], [14], where it was shown that the CC method used in the second phase (and further phases) of transaction execution have a negligible effect on overall performance and we expect this to be true also in the current environment. If the maximum system throughput is constrained by lock utilizations and, therefore, at least in theory, the second approach should be able to achieve a higher maximum throughput than the first one. This is expected to be at the cost of increased additional processing,

however, such that the potential increase in the maximum system throughput that can be sustained by the system is expected to occur for a rather narrow range of parameters (if any), e.g., very high-data-contention and "infinite resources."

5.2 Processing Options for Deviations from Access Invariance

Various deviations from *access invariance* are possible. Several examples of how this happens are given in [12], [14]. We first consider the case when this property does not hold for certain classes of transactions. In this case, different policies (optimistic die and kill) may be used according to transaction class. Locking is used for validation in both cases.

When a transaction accesses additional objects during its re-execution, validation must be performed for the newly accessed objects. The request of locks for a subset of objects accessed by a transaction may lead to deadlock, since the locks required by a transaction are no longer requested atomically. An alternative is to release all locks requested as part of the first validation and then request locks for all objects accessed in the second execution phase, in performing the second validation. To elaborate, in the case of an access to a new object, a transaction may revert to its first phase, i.e., release all of its locks and continue its execution in optimistic mode. At the other extreme it may use 2PL and obtain locks only for the newly referenced objects.

In a system with little or no access invariance, the optimistic kill policy is preferable to the optimistic/die policy, because prefetched objects are of little use if the transaction fails its validation and has to be rerun. Validation is only attempted if a transaction executes to completion, but if the validation fails, locks should be released and the whole cycle is repeated. In fact, locking based methods may be more suitable for this environment.

The performance of the aforementioned methods is beyond the scope of this discussion.

6 Conclusions

We presented a new optimistic concurrency control protocol for distributed high-performance transaction systems. Unlike other proposals for OCC in distributed systems, our method limits the number of restarts by acquiring locks to guarantee a failed transaction a successful second execution. Lock acquisition as well as validation are imbedded in the commit protocol in order to avoid any extra messages. Deadlocks are avoided by requesting all locks at once before performing validation. The protocol is fully distributed and employs parallel validation and lock acquisition.

A main advantage compared to distributed locking methods is that locks are held only during commit processing, in general, thus considerably reducing the degree of lock contention. As simulation results have confirmed, this is of particular benefit for high-performance transaction processing complexes with fast processors. For these environments, the maximum throughput may be limited by lock contention in the case of pure locking methods. The new hybrid OCC protocol, on the other hand, allows significantly higher transaction throughputs, since the overhead required for re-executing failed transactions is more affordable than underutilizing fast processors. This is also favored by utilizing large main memory buffers for caching data objects from local and remote partitions. As a result, in the new method many re-executions of failed transactions can be processed without any interruption for local I/O or remote data requests.

An open question is: How does the performance of the hybrid OCC method compare with the distributed WDL method? If we just consider the baseline models for both schemes the following observations can be made:

1) In a system with no data contention both schemes requires the same number of messages.
2) In a high-data-contention system the number of messages required by distributed WDL is not bounded, since each transaction can be restarted repeatedly. Each re-execution requires repeated remote accesses for nonlocal data and extra messages when lock conflicts occur.) On the other hand the hybrid OCC method requires just one restart and precommit messages are resent only once.

There are several complications associated with the comparison of the two methods. Firstly, as described in Section 2, the hybrid OCC method uses a **data-request** approach as opposed the **function request** approach used by the distributed WDL [15], [16]. The advantage of the former method is that a transaction can be re-executed locally after obtaining up-to-date copies of invalidated data, which is not the case with the locking scheme. Secondly, there are many variants of distributed WDL, as well as hybrid OCC, such that a performance comparison of the best WDL and OCC scheme is an area for further investigation.

Other possible directions for extending this work are as follows. A more realistic simulation study would allow shared (in addition to exclusive) locks and the caching of remote data. It is expected that such a configuration would yield more favorable results for OCC than 2PL, especially in an environment where read-only queries are quite common and caching of remote objects is allowed. Another area of investigation, as mentioned in Section 5, is to determine the performance of the variants of the OCC method, including those which deal with access variance.

Acknowledgments

The pseudocodes specifying the concurrency control paradigms in Section 3.1 are extended versions of those appearing in [44] and were contributed by Dr. Erhard Rahm (currently with the Computer Science Department of the University of Leipzig), who also coauthored the preliminary version of this paper [44]. The author also thanks Professor B. Bhargava of Purdue University for handling this paper well beyond his term as editor.

REFERENCES

[1] D. Agrawal, A.J. Bernstein, P. Gupta, and S. Sengupta, "Distributed Optimistic Concurrency Control with Reduced Rollback," *Distributed Computing*, vol. 2, no. 1, pp. 45-59, 1987.

[2] R. Agrawal, M.J. Carey, and M. Livny, "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Trans. Database Systems*, vol. 12, no. 4, pp. 609-654, Dec. 1987.

[3] R. Agrawal, M.J. Carey, and L.W. McVoy, "The Performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems," *IEEE Trans. Software Eng.*, vol. 13, no. 12, pp. 1,348-1,363, Dec. 1987.

[4] M. Bellow, M. Hsu, and V.O. Tam, "Update Propagation in Distributed Memory Hierarchy," *Proc. Sixth IEEE Int'l Conf. Data Eng.*, Los Angeles, pp. 521-528, Feb. 1990.

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

[6] M.J. Carey and W.A. Mouhanna, "The Performance of Multiversion Concurrency Control Algorithms," *ACM Trans. Computer Systems*, vol. 4, no. 4, pp. 338-378, Nov. 1986.

[7] M.J. Carey and M. Livny, "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication," *Proc. 14th Int'l Conf. Very Large Data Bases*, Los Angeles, pp. 13-25, Aug. 1988.

[8] S. Ceri and S. Owicki, "On the Use of Optimistic Methods for Concurrency Control in Distributed Databases," *Proc. Sixth Berkeley Workshop Distributed Data Management and Computer Networks*, pp. 117-129, Feb. 1982.

[9] A. Delis and N. Rousopoulos, "Performance Comparison of Three Modern DBMS Architectures," *IEEE Trans. Software Eng.* vol. 19, no. 2, pp. 120-138, Feb. 1993.

[10] D.H. Fishman, M. Lai, and W.K. Wilkinson, "Overview of the Jasmin Database Machine," *Proc. ACM SIGMOD Conf. Management Data*, pp. 234-239, 1984.

[11] P.A. Franaszek and J.T. Robinson, "Limitations on Concurrency in Transaction Processing," *ACM Trans. Database Systems,* vol. 10, no. 1, pp. 1-28, Mar. 1985.

[12] P.A. Franaszek, J.T. Robinson, and A. Thomasian, "Access Invariance and Its Use in High-Contention Environments," *Proc. Sixth Int'l Data Eng. Conf.*, Los Angeles, pp. 47-55, Feb. 1990.

[13] P.A. Franaszek, J.T. Robinson, and A. Thomasian, "Wait Depth Limited Concurrency Control," *Proc. Seventh Int'l Data Eng. Conf.*, Kobe, Japan, pp. 92-101, Apr. 1991.

[14] P.A. Franaszek, J.T. Robinson, and A. Thomasian, "Concurrency Control for High Contention Environments," *ACM Trans. Database Systems,* vol. 17, no. 2, pp. 304-345, June 1992.

[15] P.A. Franaszek, J.R. Haritsa, J.T. Robinson, and A. Thomasian, "Distributed Concurrency Control with Limited Wait Depth," *Proc. 12th Int'l Conf. Distributed Computing Systems*, Yokohama, Japan, pp. 160-167, June 1992.

[16] P.A. Franaszek, J.R. Haritsa, J.T. Robinson, and A. Thomasian, "Distributed Concurrency Control Based on Limited Wait Depth," *IEEE Trans. Parallel and Distributed Systems,* vol. 4, no. 6, pp. 246-264, Nov. 1993.

[17] J.N. Gray, "The Cost of Messages," *Proc. Seventh Ann. Symp. Principles of Distributed Computing,* Toronto, Ont., Canada, pp. 1-7, Aug. 1988.

[18] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan-Kaufmann, San Mateo, Calif., 1992.

[19] P. Heidelberger and M.S. Lakshmi, "A Performance Comparison of Multimicro and Mainframe Database Architectures," *IEEE Trans. Software Eng.*, vol. 14, no. 4, pp. 522-531, Apr. 1988.

[20] H.I. Hsiao and D.J. DeWitt, "Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines," *Proc. Sixth Int'l Conf. Data Eng.,* Los Angeles, pp. 456-465, Feb. 1990.

[21] B.C. Jenq, B.C. Twichell, and T.W. Keller, "Locking Performance in a Shared Nothing Parallel Database Machine," *IEEE Trans. Knowledge and Data Eng.*, vol. 1, no. 4, pp. 530-543, Dec. 1989.

[22] M.L. Kersten and H. Tebra, "Application of an Optimistic Concurrency Control Method," *Software—Practice and Experience,* vol. 14, no. 2, pp. 153-168, 1984.

[23] E. Knapp, "Deadlock Detection in Distributed Databases," *ACM Computing Surveys,* vol. 1, no. 4, pp. 303-328, Dec. 1987.

[24] W.J. Kohler and B.P. Jenq, "Performance Evaluation of Integrated Concurrency Control and Recovery Algorithms Using a Distributed Transaction Testbed," *Proc. Sixth IEEE Int'l Conf. Distributed Computing Systems*, Boston, pp. 130-139, Sept. 1986.

[25] H.T. Kung and J.T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. Database Systems,* vol. 6, no. 2, pp. 213-226, June 1981.

[26] G. Lausen, "Concurrency Control in Database Systems: A Step Towards the Integration of Optimistic Methods and Locking," *Proc. ACM Ann. Conf.*, pp. 64-68, 1982.

[27] *Computer Performance Modeling Handbook*, S.S. Lavenberg, ed., Academic Press, Orlando, Fla., 1983.

[28] M.D.P. Leland and W.D. Roome, "The Silicon Database Machine," *Proc. Fourth Int'l Workshop Database Machines*, pp. 169-189, Springer-Verlag, 1985.

[29] C. Mohan, H. Pirahesh, and R. Lorie, "Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions," *Proc. ACM SIGMOD Int'l Conf. Management Data*, San Diego, pp. 124-133, June 1992.

[30] S.J. Mullender and A.S. Tanenbaum, "A Distributed File Service Based on Optimistic Concurrency Control," *Proc. 10th ACM Symp. Operating System Principles*, pp. 51-62, 1985.

[31] E. Rahm, "Design of Optimistic Methods for Concurrency Control in Database Sharing Systems," *Proc. Seventh IEEE Int'l Conf. Distributed Computing Systems*, West Berlin, pp. 154-161, Sept. 1987.

[32] E. Rahm, "Concepts for Optimistic Concurrency Control in Centralized and Distributed Database Systems," *IT Informationstechnik,* (in German), vol. 30, no. 1, pp. 28-47, 1988.

[33] E. Rahm, "Empirical Performance Evaluation of Concurrency and Coherency Control Protocols for Database Sharing Systems," *ACM Trans. Database Systems,* vol. 18, no. 2, pp. 333-377, June 1993.

[34] A. Reuter and K. Shoens, "Synchronization in a Data Sharing Environment," unpublished report, IBM San Jose Research Center, 1984.

[35] J.T. Robinson, "Experiments with Transaction Processing on a Multi-Microprocessor System," IBM Research Report RC 9725, Yorktown Heights, N.Y., Dec. 1982.

[36] W.D. Roome, "The Intelligent Store: A Content-Addressable Page Manager," *Bell Systems Technical J.*, vol. 61, no. 9, pp. 2,567-2,596, 1982.

[37] D.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis II, "System Level Concurrency Control for Distributed Database Systems," *ACM Trans. Database Systems,* vol. 3, no. 2, pp. 178-198, June 1978.

[38] I.K. Ryu and A. Thomasian, "Performance Analysis of Centralized Databases with Optimistic Concurrency Control," *Performance Evaluation,* vol. 7, no. 3, pp. 195-211, 1987.

[39] J.W. Stamos and F. Cristian, "A Low-Cost Atomic Commit Protocol," *Proc. Ninth Symp. Reliable Distributed Systems,* Huntsville, Ala., pp. 66-75, Oct. 1990.

[40] Y.C. Tay, N. Goodman, and R. Suri, "Locking Performance in Centralized Databases," *ACM Trans. Database Systems,* vol. 10, no. 4, pp. 415-462, Dec. 1985.

[41] A. Thomasian, "Performance Limits of Two-Phase Locking," *Proc. Seventh IEEE Int'l Conf. Data Eng.,* Kobe, Japan, pp. 426-435, Apr. 1991.

[42] A. Thomasian, "On the Number of Remote Sites Accessed in Distributed Transaction Processing," *IEEE Trans. Parallel and Distributed Processing,* vol. 4, no. 1, pp. 99-103, Jan. 1993.

[43] A. Thomasian, "Two-Phase Locking Performance and Its Thrashing Behavior," *ACM Trans. Database Systems*, vol. 18, no. 3, Sept. 1993.

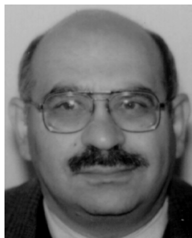
[44] A. Thomasian and E. Rahm, "A New Distributed Optimistic Concurrency Control Method and a Comparison of Its Performance with Two-Phase Locking," *Proc. 10th Int'l Distributed Computing Conf.*, Paris, pp. 294-301, May 1990.

[45] A. Thomasian and I.K. Ryu, "A Decomposition Solution to the Queueing Network Model of the Centralized DBMS with Static Locking," *Proc. 1983 ACM SIGMETRICS Conf. Measurement and Modeling Computer Systems*, Minneapolis, pp. 82-92, Aug. 1983.

[46] A. Thomasian and I.K. Ryu, "Performance Analysis of Two-Phase Locking," *IEEE Trans. Software Eng.*, vol. 17, no. 5, pp. 386-402, Sept. 1991.

[47] A. Thomasian, *Database Concurrency Control: Methods, Performance, and Analysis*, Kluwer Academic, 1996.

[48] P.S. Yu, D.W. Cornell, D.M. Dias, and A. Thomasian, "On Coupling Partitioned Data Systems," *Proc. Sixth IEEE Int'l Conf. Distributed Computing Systems*, Boston, pp. 148-157, Sept. 1986.



Alexander Thomasian received the PhD degree in computer science from the University of California at Los Angeles. He has been a faculty member at Case Western University, Cleveland, and the University of Southern California, Los Angeles, and has served as an adjunct faculty member at the University of California at Irvine and Columbia University in New York. He was a senior staff scientist at the Burroughs Corporation before joining IBM's Thomas J. Watson Research Center in Hawthorne, New York. He is now affiliated with the Networked Data Systems Department, where he is researching the area of digital libraries. He has also done research in the areas of performance modeling and analysis of transaction processing systems, concurrency control methods, parallel and distributed computer systems, and disk arrays. He has published more than 90 papers in these and other areas, and he has received IBM's Invention Achievement and Outstanding Innovation Awards. He is the author of *Database Concurrency Control: Methods, Performance, and Analysis* (Kluwer, 1996). Dr. Thomasian is a senior member of the IEEE, a member of the ACM, and an area editor for *IEEE Transactions on Parallel and Distributed Systems*. He has been on the program committees of the IEEE International Conference on Distributed Computing Systems, the IEEE International Symposium on High-Performance Distributed Computing, the IEEE International Conference on Data Engineering, and the IEEE International Conference on Information and Knowledge Management.