Distributed Optimistic Concurrency Control for High Performance Transaction Processing

Erhard Rahm
Univ. Kaiserslautern
FB Informatik, Postfach 3049
6750 Kaiserslautern, West Germany [1]

Alexander Thomasian
IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights,NY 10598

Abstract

The performance of high-volume transaction processing systems is determined by the degree of hardware and data contention. This is especially a problem in the case of distributed systems with global transactions accessing and updating objects from multiple systems. While the conventional two-phase locking method of centralized systems can be adapted for concurrency control in distributed systems, it may restrict system throughput to very low levels. This is due to a significant increase in lock holding times and associated transaction waiting time for locks, as compared to centralized systems. Optimistic concurrency control (OCC) which is similarly extensible to distributed systems has the disadvantage of repeated transaction restarts, which is a weak point of currently proposed methods. We propose a new hybrid method based on OCC followed by locking, which is an integral part of distributed validation and two-phase commit. This new OCC method assures that a transaction failing its validation will not be re-executed more than once, in general. Furthermore deadlocks, which are difficult to handle in a distributed environment, are avoided by serializing lock requests. We outline implementation details and compare the performance of the new scheme with distributed two-phase locking.

1. Introduction

For the past decade, optimistic concurrency control (OCC) has attracted a great deal of attention in the database research community. Since the original proposal in 1979 [11], a large number of refined and extended OCC schemes have been proposed for centralized and distributed database systems (see [15] for an overview). Though virtually all commercial database management systems still use (two-phase) locking for synchronizing database accesses, OCC protocols have been implemented in several prototypes, particularly for distributed environments [19], [5], [8], [13], [14]. Also, the (centralized) high performance database system IMS Fast Path actually uses a combination of OCC and locking for 'hot spot' objects where locks are only held during commit processing to reduce lock contention [7]. In this paper we propose a similar OCC scheme for distributed transaction systems that uses commit duration locking to guarantee global serializability and to reduce lock contention compared to standard locking.

Existing performance studies for OCC are mostly restricted to centralized database systems and the simple validation scheme from [11] which causes an unnecessarily high number of restarts (these unnecessary rollbacks can be avoided, e.g., by using timestamps for conflict detection (see [20] and [15]). One of the most comprehensive studies of this kind is presented in [1]. Their simulation results show that even the simple validation scheme exhibits performance characteristics similar to two-phase locking, except in cases of high CPU utilization (since restarts affect performance more seriously when the available CPU capacity is constrained). In [4], these observations are confirmed for distributed database systems. The main emphasis of this paper is, however, on replicated databases (for which the optimistic approach allows replication control with fewer messages than 'read-any, write-all' schemes). Another performance comparison of OCC and locking in distributed database systems is presented in [10]. Their experiments, however, which indicated superior performance for OCC, were limited to two nodes and were mainly influenced by I/O bottlenecks.

A main advantage of the optimistic approach is that it is deadlock-free, since deadlock detection schemes for distributed database systems tend to be complex and have frequently been shown to be incorrect [9]. Alternative deadlock resolution schemes, on the other hand, like avoidance or timeout techniques often suffer from poor flexibility and may cause a high number of unnecessary transaction aborts. Optimistic protocols also promise a higher degree of concurrency and shorter response times than locking schemes where lock conflicts result in the deactivation of transactions. So high performance requirements may not be satisfiable with standard locking protocols, particularly since higher lock contention levels have to be anticipated in distributed database systems. This is because the total number of concurrent transaction activations (multiprogramming level) increases with the number of systems thus raising the lock conflict probability. Furthermore, inter-system communication delays increase lock holding times and require higher multiprogramming levels per node in order to overlap transaction deactivations due to remote requests.

On the other hand, the applicability of optimistic schemes proposed so far is mainly restricted to environments with moderate conflict probability. With longer transactions or higher frequency of update accesses these schemes generally cause an intolerably high number of restarts and are susceptible to 'starvation' (i.e. transactions may never succeed due to permanent restart). To overcome these problems, some authors proposed a combination of locking and OCC (e.g., [12] and [3]), where transaction may be synchronized either pessimistically or optimistically. Though this is a step in the right direction, the resulting schemes are no longer deadlock-free and may be difficult to control for real applications.

In this paper, we propose a new OCC protocol which we believe offers substantial benefits over existing optimistic schemes and can be used for high performance transaction processing. The protocol to be described exhibits the following main characteristics:

- Transactions are executed optimistically, i.e. they generally do not have to wait until conflicting transactions release their locks.
- Before global validation is performed, the validating transactions request appropriate locks for all items accessed. Locks are only held during commit time (if validation is successful) so that lock conflicts are far less likely than with standard locking.
- If validation should fail, all acquired locks are retained by the transaction while being executed again. This kind of 'preclaiming' guarantees that the second execution will be successful if no new objects are referenced. In this way, frequent restarts as well as starvation can be avoided.
- The lock requests do not cause any additional messages.

[1] This work was done while the first author was at IBM Research.

CH2728-4/90/0000/0490$01.00 © 1990 IEEE

- Deadlocks can be avoided by requesting the locks in an appropriate order.
- The protocol is fully distributed.

One key concept utilized here is *phase-dependent control* [6], i.e., a transaction is allowed to have multiple execution phases, with different concurrency control methods in different phases. The current paper is thus a special case, tailored to distributed systems, with an optimistic policy in the first phase and locking in the second. Even if a transaction is known to be conflicted, its execution is continued in *virtual execution* mode, despite the fact that it cannot complete successfully. While CPU processing is mainly wasted in the virtual execution mode, disk I/O (and CPU processing required for disk I/O) in fact results in fetching data, which will be referenced again after the transaction is restarted. This ***prefetching*** of required data is specially valuable when we have *access invariance* [6] i.e., the property that a transaction will find the set of objects required for its re-execution in the database buffer (the transaction may access the same set of objects or at least related objects which will have been prefetched). Another benefit of virtual execution studied in [6] for the centralized (non-distributed) case, is the possibility of determining what locks a transaction may acquire in a second execution phase (required if validation fails). The present paper describes an algorithm which permits an efficient use of the latter property in a distributed environment.

The next section describes the system and transaction execution model assumed in this paper. General validation strategies for distributed databases are then reviewed and discussed in section 3. Our proposed protocol which is based on a distributed validation scheme is outlined in section 4. In section 5 we compare the performance of our scheme with distributed two-phase locking.

2. *System and transaction processing model*

Though our protocols are in principle applicable to a wide range of distributed database systems, we restrict our considerations here for definiteness to locally distributed systems without replication (***partitioned databases***). The proximity of the processors permits a high-speed interconnect generally required for high performance transaction systems as well as a flexible load distribution, e.g., via special front-end processors. Replicated databases are less desirable in a local environment where read accesses against the partition of another node are satisfied much faster than in a geographically distributed system. Additionally, data availability can be easily improved by mirrored disks and by attaching every disk drive to at least two nodes (so that after a node crash the corresponding database partition can still be accessed).

For transaction processing in partitioned database systems basically two approaches called ***database call shipping*** and ***I/O request shipping*** can be chosen [22]. With the former approach, the database operations are always executed where the data objects reside. The remote operations of a transaction are usually executed within subtransactions or cohort processes. With I/O request shipping, on the other hand, all database operations of a transaction are processed at its *site of origination* (i.e. the system at which the transaction arrived or was routed to) and remote objects are requested from the owner node. In this paper, we will concentrate on the I/O request shipping approach which was reported to allow for better performance than the database call shipping alternative when a high communications bandwidth is available [22]. A main reason for this was that database call shipping gives little flexibility for transaction routing since a node must process all operations against its database partition. Thus the performance (communication frequency, CPU utilization) with this approach is largely determined by the static partitioning of the database.

Similarly to OCC protocols in centralized database systems, in our scheme a transaction is processed in three phases: a read phase, a validation phase and a possible write phase [11]. During the ***read phase*** all database operations of a transaction are executed at its site of origination. Accesses to remote objects result in an I/O request to the owner system and the object is stored in the database buffer of the requesting system. Updates are performed on private object copies which are only accessible to the modifying transaction. The ***validation and write phase*** are started at the end of the transaction (EOT) and are here combined with the distributed two-phase commit protocol in order to avoid extra messages (see below). Validation basically has to ensure global serializability; conflict resolution generally relies on aborting transactions as opposed to blocking in locking protocols. The write phase is only executed by successfully validated update transactions. In this phase, sufficient log data must be forced to non-volatile storage and the modifications are made visible to other transactions by copying the private modifications into the database buffer.

3. *Validation strategies for distributed OCC*

The simplest OCC protocol for distributed databases would be a ***central validation scheme*** where all validations are sequentially performed at a central system. Such an approach is not considered here since it introduces a potential performance bottleneck, as well as a single point of failure. Furthermore, extra messages are required for sending the validation requests to the central node.

In the ***distributed validation approach***, a transaction generally validates at all nodes which were involved in its read phase (i.e. which control the partitions that were accessed by the transaction). As a consequence, a transaction can be processed without any inter-system communication when it has referenced only 'local' data objects being stored at its home site. 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 [17]:

- At EOT when all database operations of the transaction have been executed, the originator site of the transaction sends a PREPARE message to all nodes involved in the transaction's execution. 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 a local validation is successful, the modifications of local database objects as well as a ***pre-commit*** record are logged and an O.K. message is sent to the coordinator site. Otherwise, a FAILED message is returned and the site forgets about the transaction.
- The second phase of the commit protocol starts after the coordinator site 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 bringing the modified objects into the database buffer (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 of one node, but not in the serialization order of another node). An easy way to solve this problem is to enforce that *at all sites the (local) validations of a global transaction are processed in the same order*. 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 sites. Here, a broadcast (or multicast) message is used for sending the validation request and these requests need just to be processed in the order they were received [15]. Other strategies, which are more generally applicable use unique EOT timestamps or a token-ring topology to serialize validations [16].

Another difficulty for distributed database systems is the ***treatment of pre-committed database objects***, i.e. modifications of a pre-committed but not yet committed transaction. Here, basically three strategies can be pursued [16]:

- The conventional approach would be to ignore the fact that a pre-committed object copy exists and to access the unmodified object version. This, however, leads to the abortion of the accessing transaction in the case when the pre-committed transaction is successful (since the modifications of the pre-committed transaction must be seen by all transactions which are validated later).
- A more optimistic approach would be to allow accesses to pre-committed modifications though 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 to pre-committed transactions and to make sure that a transaction cannot commit if some of the accessed database modifications are still uncommitted.
- To avoid the problems associated with the two fore-mentioned strategies, we propose to block accesses to pre-committed 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 the write phase).

4. Description of the advanced OCC scheme

Our scheme is based on the distributed validation approach sketched above and uses exclusive locks to avoid accesses to pre-committed objects. In order to solve the starvation problem associated with other OCC schemes, we make extended use of locking by requesting locks for all objects (not only for modified ones) at EOT before the validation. A similar idea has been proposed for data sharing (shared disk) systems, however assuming a central node performing all validations [18] and [17]. In that proposal, locks are acquired at the central node *after* a validation has failed. Of course, such a scheme is also applicable to centralized database systems [6]. These locks are held only during commit processing if the validating transaction is successful. If the transaction should fail, the locks are retained during the re-processing 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. This is because of the prevalence of short and preplanned transaction types (e.g., debit-credit) in this environment, which usually access the same set of objects in repeated executions (high degree of access invariance).

We assume that a broadcast message is used to simultaneously start the lock request and validation phase of a transaction at all systems concerned and that these requests are processed in the order they are received. This allows not only for a parallel commit processing (supporting short response times), but also for guaranteeing global serializability (see above) as well as avoidance of deadlocks. Deadlocks are prevented since transactions request all their locks at once and the lock request phases of global transactions are subject to system-wide serialization via a broadcast mechanism. For lock acquisition we distinguish between read (shared) and write (exclusive) locks with their usual compatibility. Validation is performed by using timestamps associated with objects and by checking whether the object versions seen

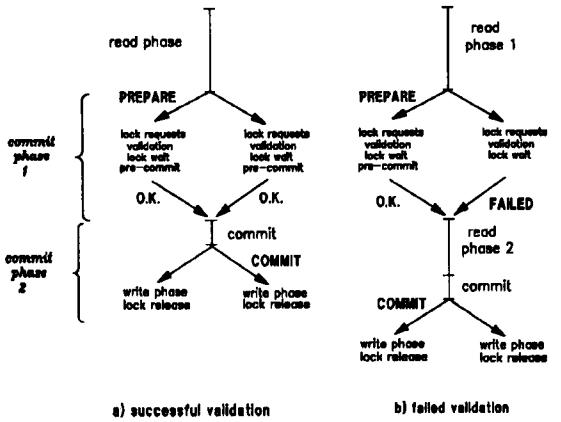

Figure 1: Transaction execution flow for successful and failed validations

by a transaction are still up-to-date. This is not automatically ensured by a successful lock acquisition since locks are requested after the object accesses so that unnoted modifications by committed transactions (for which the locks have already been released at validation time) may have been performed.

Figure 1 shows the various phases during the execution of a global transaction for a successful first execution (Figure 1a) as well as for the case of a validation failure (Figure 1b). As indicated in Figure 1, commit phase 1 consists of a lock request and validation phase, followed by pre-commit 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 not returned before 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 (Figure 1a). If any validation failed, the transaction is re-executed under the protection of the acquired locks. If no additional objects are accessed in the second execution, the transaction can be immediately committed at the end of its second read phase and the write phases and the release of the locks are performed at the respective nodes. Newly referenced database objects are subject to a complete commit protocol including lock acquisition and validation (not shown in Figure 1b).

We now provide a more detailed, procedural description of the proposed protocol. As mentioned above, we assume an I/O request shipping approach for accesses to remote data during the read phase of a transaction. 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 (we assume here that the write set of a transaction is a subset of its read set). 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 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 pre-committed transactions or by already failed transactions during their second execution. An X-lock indicates that the transaction holding the lock attempts to modify the object; in order to avoid unnecessary rollbacks we delay object accesses during the read phases until an X-lock is released. Also, an X-lock results in the abortion of validating transactions that have accessed the unmodified object version (before the lock was set). Read locks are set for accessed objects which have not been modified. Though these locks are basically not required for a correct synchronization, they prevent that the object will be updated (invalidated) by other transactions. Thus they guarantee a failed transaction a successful re-execution if it accesses only its locked objects. Incompatible lock requests are appended to the WL waiting list according to the request order.

We now describe how the lock acquisition and validation phase of a transaction T at system S is processed during commit phase 1. This processing takes 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.

```
< <  VALID := true;
  for all  k in RS (T,S) do;
    if (X-lock set or X-request is waiting for k) then
                                        VALID := false;
    if lock conflict then do;
      if k in WS (T,S)
                then place X-request into waiting list WL;
                else place S-request into WL;
    end;
    else do;  {no lock conflict}
      if k in WS (T,S)
                then XT := T        {acquire X-lock};
                else append T to ST list  {acquire S-lock};
    end;
    if wct (k,T) < WCT (k) then VALID := false;  {validation}
  end; > >
  if VALID then do;
    wait (if necessary) until all lock requests at S are granted;
    write log information;  {pre-commit}
    send O.K.;
  end;
  else do;
    wait (if necessary) until all lock requests at S are granted;
     send FAILED;
   end;
```

It is to be noted that all locks for the read and write set elements are *requested* within the critical section, even if lock conflicts occur for some requests or the transaction is to be aborted. This is required to avoid deadlocks and since the locks have to be acquired to achieve the pre-claiming effect for the re-execution of a failed transaction. Therefore, as a measure of precaution, even the read locks, only required for failed transactions, are always requested before validation. Another reason is that deferring these lock requests until the validation result is known could result in deadlocks and/or additional communication overhead.

Although we request all locks before the validation, it is to be emphasized that lock conflicts do not delay validation but result at first only in appending the lock request to the wait list. 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 rates could seriously be limited since the validations are to be performed in the same order at every node concerned. Therefore, a delay in the critical section of one node would delay all other validations. The use of timestamps allows in fact 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 abortion of the requesting transaction. For instance, when T requests an X-lock and only S-locks are granted (and no other X-requests are waiting) then T is not aborted but waits until the release of the read locks before returning the O.K. message to the coordinator. If T were aborted in this case, then the same waiting time for the X-lock would occur, but the transaction also had to be re-executed.

For a failed transaction, a system returns the FAILED message after the transaction has acquired all of its locks at this node. This message is also used to transmit the most recent copies of the locked objects so that ***separate I/O requests during the second execution are prevented***. The re-execution of a failed transaction is started as soon as it has acquired its locks at all nodes concerned. If no new objects are referenced, the second execution can be performed without any communication interruptions (since the remote objects were already obtained) or I/O delays (if all objects can be held in main memory). As a result, the re-execution of a transaction should usually be much faster and cheaper than its first execution. So even for failed transactions comparatively short lock holding times can be expected. Also, Figure 1 shows that the number of messages for commit processing does not increase for failed transaction, in general, since after the second execution no validation is required anymore if no additional objects have been referenced.

Another important advantage not mentioned so far is that remote objects can be kept in the database buffers to save remote I/O requests for other transactions, too, thus making use of locality of reference. Note that the buffer contents do not have to be completely coherent since accesses to invalidated objects are detected during validation. To eliminate invalidated remote objects from the buffers, information about which objects have been modified can be asynchronously broadcast to all nodes (together with other broadcast messages). Locking schemes cannot take advantage of such a buffering of remote objects for reducing inter-system communication since they always have to acquire their locks at the systems controlling the objects.

5. Comparison with distributed two-phase 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, our OCC scheme is considered as robust as distributed two-phase locking [2], since it mainly depends on the robustness of the commit protocol required in both schemes. The ***deadlock freedom*** of our protocol considerably simplifies the complexity of an actual implementation and avoids special fault tolerance provisions required for deadlock detection schemes.

The ***number of messages*** required for transaction processing is a primary performance indicator in distributed systems. In this respect, the hybrid OCC scheme and two-phase locking require about the same communication overhead for remote I/O requests and the commit protocol. The OCC protocol, however, avoids extra messages which may be required with locking for deadlock detection and it can reduce the number of remote I/O requests by caching remote objects (see above).

The main performance differences between the OCC and locking protocols are expected to result from the different detection and resolution of concurrency control conflicts (lock waits versus transaction restarts). It should be clear from the discussion in the previous section, that the lock contention for the hybrid OCC scheme is generally significantly lower than with distributed two-phase locking (2PL).

While in the OCC protocol locks are mostly held only during commit processing, 2PL acquires the locks before the actual object accesses. As a consequence, locks are held during large portions of the transaction's execution phase including delays for local I/O, remote I/O requests and lock conflicts. On the other hand, the OCC scheme generally aborts more transactions than standard locking where restarts occur only for deadlock resolution. The shorter lock holding times may, however, allow better response times for our hybrid OCC scheme than with distributed 2PL, thus favoring a reduced number of concurrency control conflicts. Furthermore, the number of restarts is limited compared to purely optimistic protocols, due to the acquisition of locks making it unlikely that a transaction is restarted more than once.

To allow for a quantitative performance comparison, our OCC protocol as well as the distributed version of 2PL have been implemented in a ***simulation*** system of a locally distributed transaction processing complex with partitioned databases. The model includes the concurrency control components as well as buffer management at every system and considers delays and overhead for CPU, I/O and communication. Key parameters are the number of systems, the CPU speed, transaction profile (number of I/O requests, write frequency, locality of data access, arrival rates, ...) and the cache sizes. Although space limitations do not permit us to provide a more detailed description of the simulation study, we want to summarize some preliminary simulation results in order to underline the attractiveness of our scheme.

A general observation is that in order to fully utilize fast processors to achieve high transaction rates, high multiprogramming levels (MPL) are required to overlap I/O and communication delays during the execution of transactions. However, data contention (e.g. probability of lock conflict per lock request) has been shown to increase proportionally with the concurrency degree of transactions. Transaction blocking due to lock conflicts reduces the effective MPL, thus lowering transaction throughput. With OCC, restarts waste CPU processing so that the CPU becomes saturated at lower MPL's than it would if there was no data contention. The wasted processing is determined by the fraction of transactions being restarted (100% in the worst case), since the second execution of the transaction is always successful (provided we have access invariance). The effective system throughput is determined by the useful (total minus wasted) CPU utilization. The processor speed and its useful utilization determine the effective MPL, i.e., the number of transactions that can run to completion successfully.

Our simulation results show that the OCC method has a performance similar to distributed 2PL for low data contention levels. There are few restarts with OCC and very few transactions are in the blocked state with 2PL. In experiments with faster CPUs, the MPL had to be increased to attain a higher throughput by keeping the processors busy thus introducing an increased data contention level. For these configurations our *hybrid OCC protocol clearly outperformed 2PL* where high lock contention levels prevented the effective degree of concurrency to increase significantly. As a result, 2PL allowed only for modest transaction rates and CPU utilization. With the hybrid OCC scheme, on the other hand, the effective throughput could be increased as long as the system was not saturated. The increased number of transaction restarts (due to the higher level of data contention) could be more easily tolerated with the faster processors than in the cases with slow CPUs and fewer abortions. With fast processors we observed a throughput improvement for up to relatively high MPL's because of sufficient excess capacity for re-executing failed transactions. A detailed performance analysis of the schemes appears in [21].

6. Summary

We presented a new optimistic concurrency control protocol for distributed high-performance transaction systems. Unlike other proposals for OCC in distributed systems, our scheme 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 schemes is that locks are held only during commit processing, in general, thus considerably reducing the degree of lock contention. As first simulation results have confirmed, this is of particular benefit for high-performance transaction processing complexes with fast processors. For these environments, the maximal throughput is often limited by lock contention in the case of pure locking schemes. The new hybrid OCC protocol, on the other hand, often allows here for significantly higher transaction rates since the extra overhead required for re-executing failed transactions is more affordable than under-utilizing 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 scheme many re-executions of failed transactions can be processed without any interruption for local I/O or remote data requests.

Our current effort is to investigate the performance of the new method in more detail and compare its performance with other algorithms. We are also working on new protocols which depend less on access invariance.

References

1. R. Agrawal, M. Carey, and M. Livny. "Concurrency control performance modeling: alternatives and implications," ***ACM Trans. on Database Systems 12,***4 (December 1987), 607-654.
2. P. A. Bernstein, V. Hadzilacos, and N. Goodman. ***Concurrency Control and Recovery in Database Systems.*** Addison Wesley, 1987.
3. H. Boral and I. Gold. "Towards a self-adapting centralized concurrency control algorithm," ***Proc. ACM SIGMOD Conf. on Management of Data,*** 1984, pp. 18-32.
4. M. J. Carey and M. Livny. "Distributed concurrency control performance: a study of algorithms, distribution, and replication," ***Proc. 14th Int'l Conf. on Very Large Data Bases,*** Los Angeles, CA , August 1988, pp. 13-25.
5. D. H. Fishman, M. Lai, and W. K. Wilkinson. "Overview of the Jasmin database machine," ***Proc. ACM SIGMOD Conf. on Management of Data,*** 1984, pp. 234-239.
6. P. A. Franaszek, J. T. Robinson, and A. Thomasian. "Access invariance and its use in high-contention environments," ***IBM Research Report RC 14704,*** Yorktown Heights, NY, July 1989 (to appear in ***Proc. 7th Int'l Data Eng. Conf.,*** Los Angeles, CA, February 1990).
7. D. Gawlick. "Processing 'hot spots' in high performance systems," ***Proc. IEEE 1985 Spring COMPCON,*** San Francisco, CA, February 1985, 249-251.
8. M. L. Kersten and H. Tebra. "Application of an optimistic concurrency control method," ***Software - Practice and Experience 14,***2 (1984), 153-168.
9. E. Knapp. "Deadlock detection in distributed databases," ***ACM Computing Surveys 1,***4 (December 1987), 303-328.
10. W. J. Kohler and B. P. Jenq. "Performance evaluation of integrated concurrency control and recovery algorithms using a distributed transaction testbed," ***Proc. 6th IEEE Int'l Conf. on Distributed Computing Systems,*** Boston, Mass. Sept. 1986, pp. 130-139.
11. H. T. Kung and J. T. Robinson. "On optimistic methods for concurrency control," ***ACM Trans. on Database Systems 6,***2 (June 1981), 213-226 (also presented at 5th VLDB Int'l Conf., 1979).
12. G. Lausen. "Concurrency control in database systems: a step towards the integration of optimistic methods and locking," ***Proc. ACM Annual Conf.***1982, pp. 64-68.
13. M. D. P. Leland and W. D. Roome. "The Silicon database machine," ***Proc. 4th Int'l Workshop on Database Machines,*** Springer-Verlag, 1985, pp. 169-189.
14. S. J. Mullender and A. S. Tanenbaum. "A distributed file service based on optimistic concurrency control," ***Proc. 10th ACM Symp. on Operating System Principles,*** 1985, pp. 51-62.
15. E. Rahm. "Design of optimistic methods for concurrency control in database sharing systems," ***Proc. 7th IEEE Int'l Conf. on Distributed Computing Systems,*** West Berlin, Sept. 1987, 154-161.
16. E. Rahm. "Concepts for optimistic concurrency control in centralized and distributed database systems" ***IT Informationstechnik 30,***1, (1988), pp. 28-47 (in German).
17. E. Rahm. "Empirical performance evaluation of concurrency and coherency control protocols for data sharing," ***IBM Research Report RC 14325,*** Hawthorne, NY, December 1988.
18. A. Reuter and K. Shoens. "Synchronization in a data sharing environment," Unpublished report, IBM San Jose Research Center, 1984.
19. W. D. Roome. "The intelligent store: a content-addressable page manager," ***Bell Systems Tech. Journal 61,***9 (1982), 2567-2596
20. I. K. Ryu and A. Thomasian. "Performance analysis of centralized databases with optimistic concurrency control," ***Performance Evaluation 7,***3 (1987), 195-211
21. A. Thomasian and E. Rahm. "A new distributed optimistic concurrency control method and a comparison of its performance with two-phase locking," ***IBM Research Report RC 15073,*** Hawthorne, NY, October 1989.
22. P. S. Yu, D. W. Cornell, D. M. Dias, and A. Thomasian. "On coupling partitioned data systems," ***Proc. 6th IEEE Int. Conf. on Distributed Computing Systems,*** Boston, Mass. Sept. 1986, pp. 148-157