A New Distributed Optimistic Concurrency Control Method and a Comparison of its Performance with Two-Phase Locking

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

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

Abstract

The performance of high-volume transaction processing systems is determined by the degree of contention for hardware resources as well as data. Data contention is especially a problem in the case of data partitioned (e.g., distributed) systems with global transactions accessing and updating objects from multiple systems. While the conventional two-phase locking (2PL) method of centralized systems can be adapted to data partitioned systems, it may restrict system throughput to levels inconsistent with the available processing capacity. This is due to a significant increase in lock holding times and associated transaction waiting times for locks in distributed data partitioned systems, as compared to centralized systems. Optimistic concurrency control (OCC) is similarly extensible to data partitioned systems, but has the disadvantage of repeated transaction restarts, which is a weak point of currently proposed methods. We present a new distributed OCC method followed by locking, such that locking is an integral part of distributed validation and two-phase commit. This 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 2PL through a detailed simulation, which incorporates queueing effects at the devices of the computer systems, buffer management, concurrency control, and commit processing. It is shown that in the case of higher data contention levels, the hybrid OCC method allows a much higher maximum transaction throughput than distributed 2PL. We also report the performance of the new method with respect to variable size transactions. It is shown that by restricting the number of restarts to one, the performance achieved for variable size transactions is comparable to fixed size transactions with the same mean size.

1. Introduction

Since the original proposal of optimistic concurrency control in 1979 [9], a large number of OCC schemes have been proposed for centralized and distributed database systems (see [14] for an overview). Though virtually all commercial database management systems still use two-phase locking (2PL) for synchronizing database accesses, OCC protocols have been implemented in several prototypes, particularly for distributed environments [18], [4], [6], [11], and [12]. In this paper we propose a hybrid OCC scheme for transaction processing in partitioned database systems (including distributed databases) that uses commit duration locking to guarantee global serializability [2] and to reduce lock contention compared to standard locking.

A main 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 [7]. Alternative deadlock resolution schemes are prevention and time-out techniques. The difficulty with time-outs is the determination of an appropriate time-out interval. Several deadlock-free locking schemes such as wound-wait and wait-die [2] have been proposed, but their performance in distributed database systems requires further investigation. Furthermore high performance requirements may not be satisfiable with 2PL, particularly since higher lock contention levels have to be anticipated in distributed database systems. This is because the total number of concurrent transactions activated (Multi-Programming Level-MPL) is further increased with the number of systems, thus raising the lock conflict probability. The lock contention probability is further increased due to the increase in lock holding times, which is due to the extra delays introduced by inter-system communication.

An important issue in OCC is the efficiency of the validation method. The simple validation scheme first proposed in [9] (for centralized systems) causes an unnecessarily high number of restarts, which can be prevented, e.g., by using timestamps for conflict detection [20], [13]. There have been several extensions of the original validation method to a distributed environment (see e.g., [3] and [1]) ([1] uses versioning to improve performance for read-only transactions). With longer transactions or a 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 repeated restarts). To overcome these problems, some authors proposed a combination of locking and OCC (see e.g. [10]) where transactions 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.

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

- 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 (the second execution tends to be much faster than the first one as is explained in Section 4). This kind of "pre-claiming" 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 prevented.
- The lock requests do not cause any additional messages.

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

One key concept utilized here is ***phase-dependent control*** [5], i.e., a transaction is allowed to have multiple execution phases, with different concurrency control methods in different phases, e.g., optimistic CC in the first phase and locking in the second phase. 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 after the transaction is restarted. This ***prefetching*** of required data is specially valuable when we have ***access invariance*** [5], 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 pre-fetched). Another benefit of virtual execution is the possibility of determining the locks which a transaction should acquire in a second execution phase (if any) [5]. The present paper describes an algorithm 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. An experiment to compare OCC and 2PL using the Cm* experimental system at CMU was reported in [17]. 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 [8], 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. A simulation study was undertaken as part of this effort, although an approximate analysis based on the approach in [20] (for a centralized system) is feasible. The problem with an approximate analytic solution is that it requires simplifying modeling assumptions to make the analysis tractable and that a simulation of the system would be required in any case for validation purposes.

The next section describes the model for the computer systems, the database, and transactions considered in this paper. General validation strategies for OCC in distributed databases are then reviewed in Section 3. Our proposed protocol, which is based on a distributed validation scheme is outlined in Section 4. In Section 5 we describe the simulation model and compare the performance of 2PL and the new OCC method. Conclusions appear in Section 6.

2. System and Transaction Processing Model

Though our protocols are in principle applicable to a wide range of distributed database systems, we restrict our discussion to locally distributed systems with no data replication, i.e., a partitioned database. This is because of the practical significance of shared-nothing systems based on multi-micro systems. 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 a front-end processor. 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).

A transaction arriving at the system is routed by a front-end processor to the system which holds relevant data. This node will constitute the ***primary node*** of transaction execution. In case the transaction references data which is not available locally 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 by sub-transactions or cohort processes where the data objects reside. With I/O request shipping, on the other hand, the required data is sent to the primary node where it is processed (e.g., a capability provided by Tandem's file system). While the proposed CC method is applicable to both approaches, 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 communication 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. The partitioning of the database affects the frequency of inter-system communication and CPU utilization and hence the overall performance.

Similarly to the OCC protocol in a centralized system, in our scheme a transaction is processed in three phases: a ***read phase***, a ***validation phase*** and a ***write phase*** if the validation was successful [9]. The last two phases are initiated at the end of transaction execution (EOT) and are combined here with the distributed ***two-phase commit protocol*** in order to avoid extra messages (see Section 3).

3. Validation Strategies for Distributed Databases

The simplest OCC protocol for distributed databases would be a ***central validation scheme*** where all validations are sequentially performed at a central system. This 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 scheme***, 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 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 ***pre-commit*** 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 as well as a ***pre-commit*** 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 of one node, but not another). An easy way to solve this problem is to enforce that ***at all nodes 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 nodes. Here, a multi-cast message is used to send the validation request (including a message to the primary node in case it holds data accessed by the transaction) and these requests need to be processed in the order they are received. Other strategies which are more generally applicable use unique EOT timestamps or a circulating token to serialize validations (a discussion of such schemes in the context of a data-sharing system appears in [14]).

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 [14]:

1. 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 abort 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).
2. A more optimistic approach would be to allow accesses to pre-committed 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 pre-committed transactions and to make sure that a transaction cannot commit if some of the accessed database modifications are still uncommitted. Note that this method does not lead to deadlocks since transactions blocked in the optimistic mode do not hold any locks.
3. It seems best 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 transaction commit . On the other hand this approach contributes to the elongation of the duration of the optimistic first phase, decreasing the chances of a successful validation.

The results reported in this paper are based on the first approach, since it is the simplest to implement and for the set of parameters considered in this study resulted in a small increase in the fraction of transactions failing their validation compared to the third approach.

4. Description of the Hybrid 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 extensive use of locking by requesting locks for all objects (not only for modified ones) at EOT before the validation.[1] If the validating transaction is successful, these locks are held only during commit processing. 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 in this environment, which usually access the same set of objects in repeated executions (high degree of access invariance).

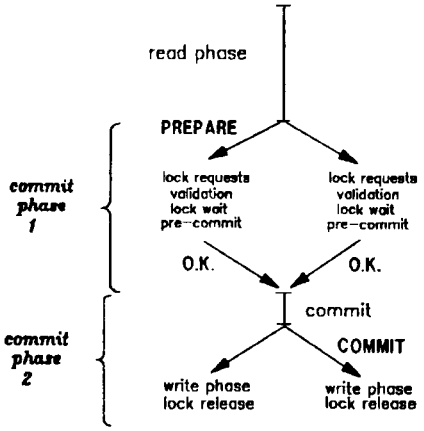
We assume that a multi-cast message (over the broadcast medium) is used to simultaneously make lock requests and start the validation phase of a transaction at all systems concerned and that these requests are processed in the order they are received at each node. This not only allows parallel commit processing (supporting short response times), but also guarantees global serializability, as well as avoidance of deadlocks. Deadlocks are avoided 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 matrix. Validation is performed by using timestamps associated with objects and by checking whether the object versions seen 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.

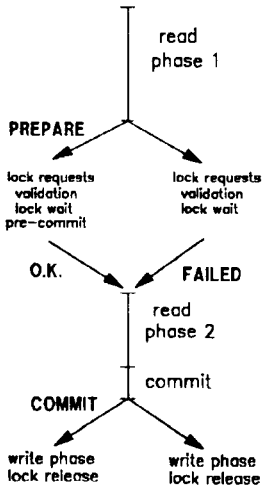
Figure 1 shows the various phases during the execution of a global transaction for a successful first execution (Figure 1a) as well as for the case of a validation failure (Figure 1b). As indicated in Figure 1, commit phase 1 consists of a lock request and validation phase, followed by pre-commit logging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

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 revert to 2PL and obtain locks only for the newly referenced objects. The issue of "access variance" and the performance of the fore-mentioned methods is beyond the scope of this discussion.

[1] A similar idea has been proposed for data sharing (shared disk) systems, however, assuming a central node performing all validations [16], [15]. In these proposals, locks are acquired at the central node ***after*** a validation has failed.



a) successful validation



b) failed validation

Figure 1: Transaction execution flow for successful and failed validations

We now provide a more detailed, procedural description of the proposed protocol. 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 prevent unnecessary rollbacks we may delay object accesses during the read phase until an X-lock is released (see Section 3). 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 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. Incompatible lock requests are appended to the WL waiting list according to the request order.

We now 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.

```
< <  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;
               {validation}
     if wct (k,T) < WCT (k) then VALID := false;
  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 guarantees that deadlocks cannot occur since all locks are requested atomically with respect to other transactions, because (i) all locks on a node are requested in a critical section, (ii) 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 pre-claiming 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 lock conflicts do not delay validation, but result at first only in appending unsuccessful lock requests 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 throughput could be seriously 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 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 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. 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 avoided.*** 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 the database buffer). 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 transactions, in general, since after the second execution no validation is required anymore if no additional objects have been referenced.

Data objects can be buffered remotely (in addition to their home node) to save remote I/O requests. This approach is quite suitable for an OCC based method, but not for 2PL. This is because a remote access to acquire a lock is required for 2PL, even if the data is locally available, but this is not so for OCC. In the case of OCC the buffer contents do not have to be up-to-date, since accesses to invalidated objects are detected during validation. The buffer coherence problem can be addressed using a buffer invalidate or refresh policy. This is beyond the scope of this paper.

5. 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 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.

The relative performance of OCC and 2PL is quantified in this section using a simulation study. In Sections 5.1, 5.2, and 5.3 we describe the simulation model for: (i) the multi-computer system, (ii) the partitioned database, and (iii) transactions. Simulation results are reported in Section 5.4

5.1. The Multi-Computer System Model

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

1-Multi-system configuration. There are $N = 4$ computer systems, consisting of tightly-coupled 4-way multiprocessors. The total processing capacity per system is varied to study this effect on the relative performance of 2PL and OCC methods. We considered 100, 200, and 400 MIPS per system or 25, 50, 100 MIPS per processor, respectively.

2-Inter-system communication. A high bandwidth bus with broadcast capability interconnects the computer systems. The communication delay is assumed to be negligibly small. We take into account, however, the CPU overhead to send and receive messages.

3-I/O subsystem. The I/O configuration, more specifically the number of disks per system, is selected to match the corresponding CPU processing capacity, such that the ratio of CPU and disk utilization, taking into account the database cache hit ratio (see below), is 75/20. 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., non-local 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 a significant improvement in performance in conjunction with read-only transactions (queries) for OCC. High contention items (see Section 5.3) 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.

5-Logging and recovery. Non-volatile (random access) storage is 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 on disk (such a capability is provided in modern cached disk controllers). This results in reducing lock holding time for both CC methods.

5.2. 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. The effective database size for each category of data items at each system is $D_{high} = 1000$ and $D_{low} = 31000$. 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.

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 I/O request shipping architecture postulated in our study requires locking or timestamping data items (in the first phase of OCC) at the level of disk blocks.

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 (e.g., 50% reduction in lock conflicts when 70% of accesses are in shared mode). Note that the same effect can be achieved by setting $D_{high} = 2000$ rather than $D_{high} = 1000$.

5.3. 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. Transaction sizes and their relative frequencies used in our simulation are as follows: 4(0.20), 8(0.20), 16(0.35), 32(0.25). In addition we consider the case 16(1.0), i.e., a single transaction class with a fixed size equaling the mean for variable size transactions. This assures the same throughput at a single system (or when all transactions are local) for fixed and variable size transactions when there is no data contention thus allowing an easy comparison. This is not so in the case of a multisystem configuration with global transactions, since ***variable size transactions tend to access data at a fewer number of distinct remote nodes than fixed size transactions on the average.*** To be specific, for the set of parameters used in our simulations fixed (resp. variable) size transactions access 3.25 (resp. 2.97) nodes on the average, which always includes the primary node of transaction execution. This issue is discussed thoroughly in [21].

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). Each transaction is routed to a system at which it exhibits a high degree of locality. 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.

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 $I_{disk} = 5000$ instructions are required (the processing required to retrieve cached data is considered to be negligible). It takes $I_{send} = 5000$ instructions to send (resp. receive) a message. Therefore 20,000 instructions are executed for inter-system 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 $I_{commit} = 5000$ 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 $I_{pre-commit} = 5000$ instructions are executed at the primary node of transaction execution (mainly to write a pre-commit log record). There is also a per system overhead of I_{send} and $I_{receive}$ to send and receive PRECOMMIT messages. Pre-commit processing at secondary nodes from which data was accessed requires $I_{remote} = 5000$ instructions, which includes writing pre-commit 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.

5.4. Simulation Results

A discrete-event simulation program was written to compare the performance of 2PL and OCC methods. The overall system throughput for all N systems is the performance measure of interest in comparing the distributed 2PL and the new OCC method. 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 of the overall throughput.

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. Given in Figure 2 are the transaction throughputs (in transactions per second) versus the per system degree of concurrency or MPL for the three cases where each one of the four systems has a total processing capacity of 100, 200, and 400 MIPS, respectively. Each graph depicts the throughput characteristic for NDC, 2PL, and OCC for fixed and variable transaction sizes. Each point on the graphs corresponds to the mean obtained from three runs, such that the system throughputs measured in different runs were within 5% of each other (with the exception of the thrashing region for 2PL). With no data contention a slightly higher overall throughput is attained in the case of variable size transactions relative to fixed size transactions. This is due to the tendency of variable size transactions to access objects at a fewer number of distinct remote nodes than fixed size transactions, as noted earlier. This results in reduced CPU processing for inter-system communication and a slightly higher throughput for variable size transactions compared to fixed size transactions.

In the case of NDC as M (the number of activated transactions) 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 "well-behaved" multiprogrammed computer system affected only by hardware resource contention, but before bottlenecks in systems software are encountered.

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 work 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 (see e.g., [19] and [2]). The maximum throughput attained by 2PL (T_{2PL}^{max}) indicates the best performance attainable by 2PL for the given degree of data contention.

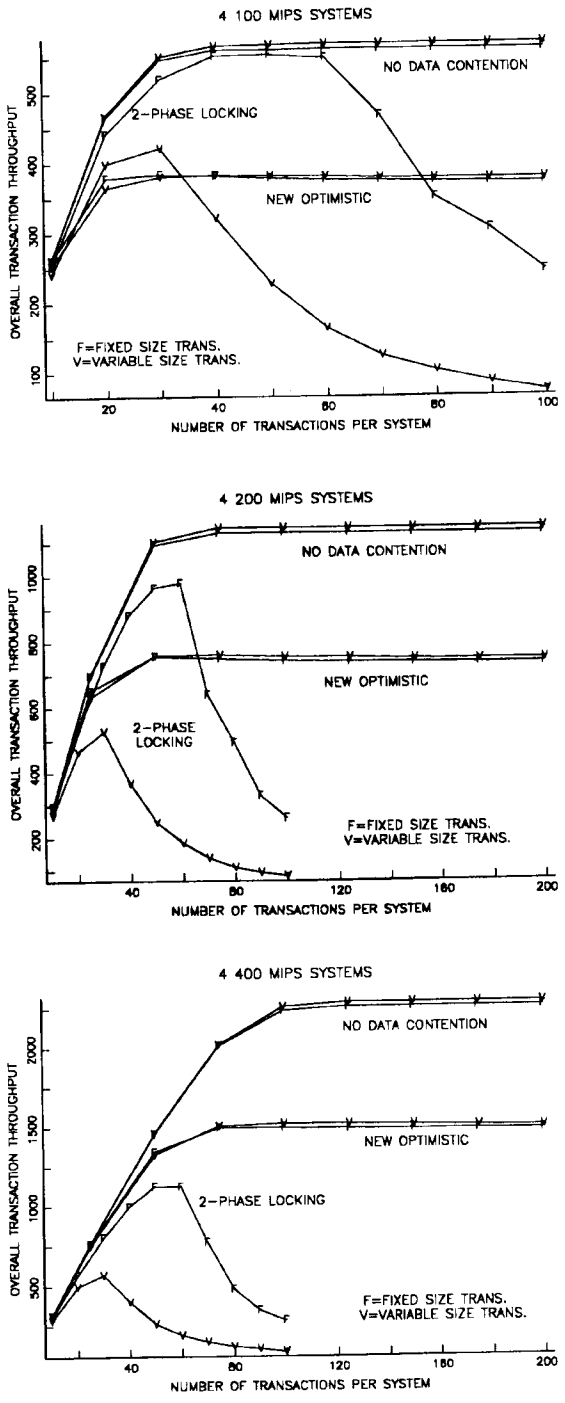


Figure 2: System throughput (in transactions/sec.) versus MPL for three cases with 100, 200, and 400 MIPS processing capacity per node.

It can be observed that the throughput for 2PL in the case of variable size transactions is much lower than the corresponding throughput for fixed size transactions. This can be explained as follows [19]: (i) The mean number of locks held by variable size transactions with a given mean say L tends to be higher than that of transactions with fixed size L, resulting in an increase in the lock conflict probability. (ii) The mean delay per conflict (waiting for a lock) is also higher when transaction sizes are variable rather than fixed.

In the case of OCC $T_{OCC}(M)$ follows $T_{NDC}(M)$ initially, but as M is increased further $T_{OCC}(M) < T_{NDC}(M)$, which is due to the wasted processing caused by failed validations. The performance for OCC is determined by the fraction of transactions validated successfully in the first phase. A higher degradation in performance is thus expected in the case of variable size transactions, since longer transactions with OCC are restarted with a higher probability than shorter transactions on two accounts, which constitutes a quadratic effect (this follows from the analysis in [20]): (i) they access more data items, (ii) they stay longer in the system and thus encounter a larger number of transaction commits. The fact that this effect is negligible in our study is attributable to the fact that long transactions can be restarted only once. Variable size transactions slightly outperform fixed size transactions beyond the point where the maximum throughput is achieved, i.e., the processors become 100% utilized. This is because of the inherently higher throughput attainable by variable size transactions. It should be noted that as the transaction concurrency is increased, the maximum system throughput for the OCC method is obtained at the point where the processors are 100% utilized. 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 comparing 2PL and OCC we have three cases based on the relative speeds of the processors (Figure 2).

1. In the case of 100 MIPS systems, the 2PL method outperforms the OCC method both for fixed and variable size transactions.
2. In the case of 200 MIPS systems, 2PL outperforms OCC in the case of fixed size transactions, but the reverse is true in the case of variable size transactions.
3. In the case of 400 MIPS systems, OCC outperforms 2PL for both fixed and variable size transactions. OCC (resp. 2PL) peak at 1490 (resp. 570) transactions per second in the case of variable size transactions, which is a factor of 2.7 improvement in the maximum throughput achievable by the system.

It follows from the above discussion that this trend continues for even faster processors. In the limit the mean throughput for OCC will be determined by the maximum degree of concurrency for transactions holding locks in the second phase.

Other performance measures of interest are the (per class) mean response times and device utilizations. CPU utilization can be deduced simply in cases when there is no or little wasted processing (NDC and 2PL) as the product of system throughput and the mean transaction processing time at the CPU. The CPU is 100% utilized beyond the point that the throughput achieves asymptotic behavior in the case of NDC. In the case of OCC, the CPU is 100% 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: (i) that they are acceptably low, and (ii) 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 parameters such as: (i) the level of data contention (as noted earlier), (ii) the number of systems, (iii) imbalanced transaction loads at each system and non-uniform remote accesses, yielded results supporting our conclusions.

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 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 simulation results have confirmed, this is of particular benefit for high-performance transaction processing complexes with fast processors. For these environments, the maximum throughput is often limited by lock contention in the case of pure locking schemes. 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 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.

This work can be extended in several directions. 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. Another area of investigation is the performance of the proposed variants of the OCC method to deal with access variance (see Section 4) with respect to 2PL.

References

1. D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. "Distributed optimistic concurrency control with reduced rollback," ***Distributed Computing 2***,1 (1987), 45-59.
2. P. A. Bernstein, V. Hadzilacos, and N. Goodman. ***Concurrency Control and Recovery in Database Systems.*** Addison-Wesley, 1987.
3. S. Ceri and S. Owicki. "On the use of concurrency control methods for concurrency control in distributed databases," ***Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks,*** February 1982, pp. 117-129.
4. 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.
5. P. A. Franaszek, J. T. Robinson, and A. Thomasian. "Access invariance and its use in high-contention environments," ***Proc. 7th Int'l Data Engineering Conf.,*** Los Angeles, CA, February 1990, pp. 47-55.
6. M. L. Kersten and H. Tebra. "Application of an optimistic concurrency control method," ***Software - Practice and Experience 14***,2 (1984), 153-168.
7. E. Knapp. "Deadlock detection in distributed databases," ***ACM Computing Surveys 1***,4 (December 1987), 303-328.
8. 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.
9. 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 ***Proc. 5th Int'l Conf. on Very Large Data Bases,*** 1979).
10. 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.
11. 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.
12. 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.
13. 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, pp. 154-161.
14. E. Rahm. "Concepts for optimistic concurrency control in centralized and distributed database systems" ***IT Informationstechnik 30***,1, (1988), pp. 28-47 (in German).
15. E. Rahm. "Empirical performance evaluation of concurrency and coherency control protocols for data sharing," ***IBM Research Report RC 14325,*** Yorktown Heights, NY, December 1988.
16. A. Reuter and K. Shoens. "Synchronization in a data sharing environment," Unpublished report, IBM San Jose Research Center, 1984.
17. J. T. Robinson. "Experiments with transaction processing on a multi-microprocessor system," ***IBM Research Report RC 9725,*** Yorktown Heights, NY, December 1982.
18. W. D. Roome. "The intelligent store: a content-addressable page manager," ***Bell Systems Tech. Journal 61***,9 (1982), 2567-2596.
19. I. K. Ryu and A. Thomasian. "Analysis of database performance with dynamic locking," ***IBM Research Report RC 11428,*** Yorktown Heights, NY, October 1986 (to appear in the ***Journal of the ACM***).
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 "On the number of remote sites in distributed transaction processing," *IBM Research Report RC 15430,* Yorktown Heights, NY, January 1990.
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.