INTEGRATED SOLUTIONS TO CONCURRENCY CONTROL AND BUFFER INVALIDATION IN DATABASE SHARING SYSTEMS

Erhard Rahm

University of Kaiserslautern, FB Informatik
Postfach 3049, D-6750 Kaiserslautern, West Germany

Abstract
In the near future large transaction processing systems will have to meet high requirements concerning throughput, response times, availability and modular growth. A possible architecture for such high performance systems is Database Sharing (DB-sharing) where multiple loosely or closely coupled processors share access to a single set of databases. This paper addresses the problems of concurrency control and buffer invalidation that are both introduced by the DB-sharing architecture. Concurrency control is required to control the processors' accesses to the shared database. The buffer invalidation problem, on the other hand, results from the existence of a database buffer at each processor. Modification of a page within one buffer therefore invalidates all copies of the same page stored in other processors' buffers. We show how effective solutions to both problems are feasible by extending the synchronization component to control buffer invalidations, too. This is demonstrated by two decentralized approaches from which one is based on locking while the other relies on optimistic synchronization.

1. Introduction

Future transaction processing systems for large applications as in banking or reservation processing will have to meet high performance and availability requirements. Such DB-based systems must be capable of high transaction rates (e.g. 1000 short transactions per second) with equivalent response times compared to present systems [4]. Another key requirement is extensibility of the system (modular growth).
It has been clearly recognized that these demands cannot be fulfilled by database management systems (DBMS) on uniprocessors or tightly coupled multiprocessors. The increase of processing power of these monolithic systems has failed (or will fail) to match the required transaction rates. Furthermore, those centralized systems provide poor availability and extensibility [7].
In order to satisfy the various transaction processing requirements, two basic approaches called DB-distribution and DB-sharing are proposed [7]. These multiprocessor DBMS consist of a set of autonomous processors that are loosely or closely coupled. Each processor owns a local main memory and a separate copy of operating system (OS) and DBMS. With loose coupling inter-processor-communication is exclusively based on messages, whereas in closely coupled systems certain functions may be implemented using a common memory partition [13]. The difference between DB-distribution and DB-sharing results from the assignment of the disk drives to processors:

- In DB-distribution systems each processor owns some fraction of the disk devices and the databases stored on them. Accesses to 'non-local' data require communication with the processor owning the corresponding database partition. This approach is used among others by the TANDEM NonStop system [1] and many distributed database systems such as R* [19].
- In DB-sharing systems each processor has direct access to the entire database. This requires that all processors are physically close (e.g. in one room) and permits a high-speed communication system (e.g. 100 MB/sec). Examples for DB-sharing systems are the Data Sharing facility of IMS/VS [9], Computer Console's Power System [20], the DCS project [17] and the AMOEBA project [18].

A comparison between DB-distribution and DB-sharing can be found in [7]. Here, we concentrate ourselves on loosely coupled DB-sharing systems as depicted in Fig. 1. A global load control located at one or more front-ends distributes each incoming transaction to one of the processors (transaction routing). A transaction can be completely executed at one processor because each CPU has direct access to all parts of the shared database(s). This avoids the necessity of a distributed 2-phase-commit protocol as required in DB-distribution systems.

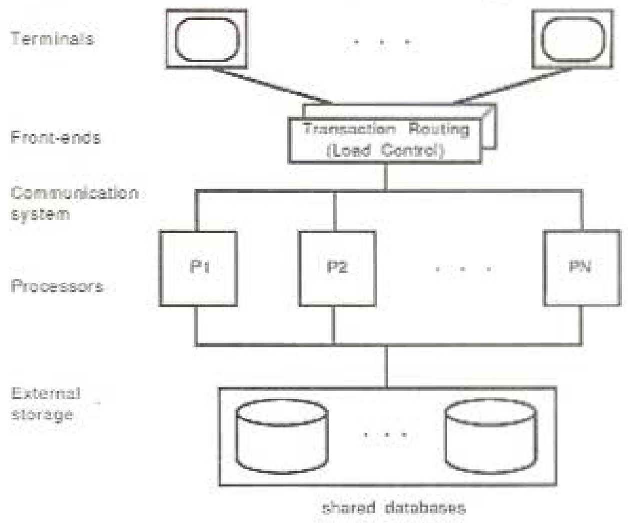


Figure 1: Structure of a loosely coupled DB-sharing system

One of the main advantages of DB-sharing is flexibility. Since each processor can access the entire database, transaction work may be dynamically distributed among the processors according to current needs and system availability. Additional processors can be added without altering the transaction programs or the database schema. Likewise, a processor failure does not prevent the surviving processors from accessing the disks or the terminals. Transactions in progress on a failed processor can be redistributed automatically among the available processors.

Naturally, the design of a DB-sharing system requires new or extended functions to be added, compared to centralized DBMS:

- The synchronization component has to coordinate all accesses to the shared database in order to guarantee serializability of the executed transactions. Since there is no common memory, synchronization requires message exchange among the processors which is much more time consuming than lock request handling in a centralized DBMS (e.g. a few hundred instructions for each lock grant or release). Therefore, the concurrency control algorithm must minimize the average number of synchronization messages per transaction. A survey of conceivable synchronization techniques can be found in [12,16].
- The recovery component is responsible for system-wide logging and recovery. Each processor has to maintain a local log used for transaction undo as well as for crash recovery. In addition, a global log (e.g. for media recovery) is constructed by merging the local log data. Crash recovery is performed by the surviving processors in order to continue transaction processing. Uncommitted transactions of the failed processor are rolled back and restarted on another processor.
- Load control has to find an effective distribution of the current workload against the set of available processors (transaction routing). No processor must be overloaded and the routing should support synchronization with minimal communication cost. To fulfill these tasks, the load control must react dynamically to changes in the workload or within the system configuration (e.g. crash or reintegration of a processor). In [15] load control is discussed in more detail.

In this paper we consider a further problem of DB-sharing systems, namely the buffer invalidation problem which is introduced in the next section. Then, we give solutions to this problem by extending two different synchronization strategies. Finally, we summarize our proposals and discuss some related problems.

2. Buffer Invalidation Problem

Since data can only be manipulated in main memory, part of the database has to be loaded into a main storage area before processing and has to be written back to disk after modification. For this purpose, the DBMS maintains a so-called database buffer consisting of page frames of uniform size; the number of frames can be selected as a DBMS parameter. Since physical access to a page on disk is much more expensive than access to a page in the buffer, a major goal of buffer management is the minimization of physical I/O for a given buffer size [2].

In a DB-sharing environment minimizing disk accesses is even more important in order to avoid waiting situations in front of a disk. Unfortunately, the existence of a buffer in each processor leads to a buffer invalidation problem, because multiple copies of a database page may reside in multiple buffers at any given time. Modification of any copy will thus invalidate all other copies. A solution to this buffer invalidation problem must mainly cope with two points:

1.) Invalidated objects in the buffers must be detected in order to avoid access to obsolete data.
2.) The new contents of modified objects must be propagated to other processors when these objects are requested there.

The solutions to these problems strongly depend on the method used for update propagation to the database on disk (FORCE or NOFORCE [8]). With a FORCE-strategy all modifications of a transaction must be written to the physical database before the transaction commits. In a DB-sharing system this has the advantage that the valid copy of a page can always be read from disk. Therefore, FORCE just requires a solution to the above point 1 (avoidance of access to invalidated copies). However, forcing modified pages to disk at EOT is overly expensive and results in serious performance impacts. Response times of update transactions are considerably increased and page modifications of different transactions cannot be accumulated [8]. To make FORCE acceptable in a DB-sharing environment, one could extend the storage hierarchy by introducing a non-volatile global database buffer to which the modified pages are forced out. Such an optimization leads to a closely coupled DB-sharing system [13].

With NOFORCE, on the other hand, a transaction usually updates only the objects in the (local) database buffer; the pages on disk are obsolete, in general. In a DB-sharing environment this complicates the treatment of buffer invalidation since it must be determined where the current copy of a page can be found. Whereas FORCE implies an exchange of modified pages across the shared disks, with NOFORCE modified objects can also be transmitted via the inter-processor connections. With a high-speed communication system this should be considerably faster provided that overload situations can be avoided.

In a DB-sharing system where a NOFORCE-scheme should be applied (together with update-in-place [8]), it must be ensured that a block (page) B on disk may only be overwritten by a processor owning the most recent copy of B (e.g. the processor that has performed the last modification of B). Otherwise, it could happen that the valid copy on disk is overwritten by an obsolete version; if the valid copy has only been on disk, an inconsistency of the database would have been produced.

A general solution to buffer invalidations is the so-called broadcast approach that usually works in combination with a FORCE-strategy. To avoid access to invalidated objects in the database buffers, the identifiers of modified objects are broadcast before committing the modifying transaction. So, obsolete copies can be detected and are discarded

from the buffers. With FORCE, the valid copy of a database page can then be read from disk.
The broadcast solution is a simple and general approach but it suffers from two severe weaknesses: First, the FORCE-strategy is in general not acceptable for high performance DBMS (see above). Second, the broadcast solution requires a large number of notify messages that increase as a square function of the number of processors.
In the next two sections we show that much more effective solutions are feasible in cooperation with concurrency control. This is demonstrated by two promising sychronization strategies that both rely on decentralized control. First, the primary copy approach based on locking is investigated in section 3, an algorithm that permits a close cooperation with load control. In section 4 an optimistic strategy is described that is attractive for requiring only one synchronization message (for validation) per transaction. Throughout these sections we assume, if not otherwise stated, a NOFORCE-strategy for update propagation; of course, the solutions can also be adapted to a FORCE-policy where the exchange of modified pages is simplified. It is also assumed that synchronization takes place on page (block) level. A coordinated solution to concurrency control and buffer invalidation was also proposed in [3], however, for a centralized locking scheme.

3. Primary Copy Locking (PCL)

In this approach the synchronization responsibility is distributed among all processors. Therefore, the database is logically partitioned into N disjoint parts and each of the N processors performs the global synchronization for one partition. A processor is said to have the primary copy authority (PCA) for its partition [15]. As Fig. 2 shows, each lock manager maintains a global lock table (GLT) to control the objects of its partition and a local lock table (LLT) to keep information about granted or requested locks for local transactions.

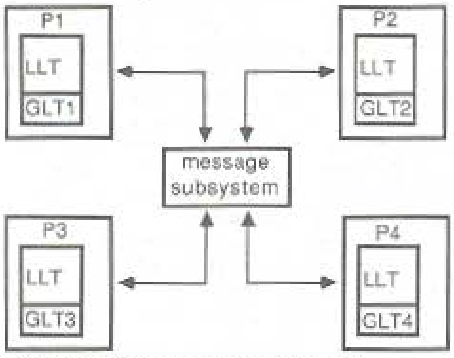


Figure 2: Primary Copy Locking (N = 4)

PCL has the obvious advantage that lock requests from processor P within the partition controlled by P can be managed locally, regardless of external contention. Lock requests for a partition of another processor are sent to the authorized processor. To minimize the number of such 'long' lock requests, load control can use the current PCA distribution to route a transaction to that processor where most of the required data can be synchronized locally. Furthermore, the routing strategy and the PCA distribution can be adapted after the crash or reintegration of a processor, or if the load profile has changed significantly.
The structure of block entries in the LLT and in the GLT as well as a detailed description of the synchronization protocol itself can be found in [14]. Here, we give two alternatives for controlling buffer invalidations within a NOFORCE-environment. Both solutions use additional information in the GLTs and avoid extra messages as far as possible.

Timestamp solution

In this solution, a timestamp is assigned to each block pertaining to the most recent modification of any object in the block. To control buffer invalidations, the primary copy scheme is extended in the following points:

1. For each modified block the responsible lock manager keeps a block entry in its GLT where the timestamp of the latest modification is stored in an additional field.
2. Modified blocks are transmitted to the PCA-processor together with the release message for the X-lock. This has the effect that the valid version of a block can always be found at the PCA-processor or on disk if the block was written out by the PCA-processor.
3. Before a lock request is issued it is checked whether or not the corresponding block resides in the local buffer. If so, then the timestamp is also sent with the lock request to the PCA-lock manager. Using this timestamp information, the PCA-processor can determine whether the local copy is up-to-date. If not, the valid copy is transmitted together with the lock response message or it is told that it can be read from disk.

In the example in Fig. 3, the PCA for block B is assigned to processor P2. It is assumed that block B was most recently changed at time t2 by a transaction T that was executed at a processor P3 not shown in Fig. 3. Together with the release of the X-lock on B, T has also transmitted the modified block containing the new timestamp t2 to processor P2. The modification time t2 is stored in the block entry of B in GLT2, and the block is written into P2's buffer. Assume now, that a transaction at P1 wants to access block B and that a copy of B with timestamp t1 (t1 < t2) resides in the buffer of P1. Since the lock request message also contains this timestamp t1, P2 can detect the buffer invalidation using the block entry of B in its GLT. The valid copy of B can be transmitted to P1 within the lock response message when the lock request is satisfiable.

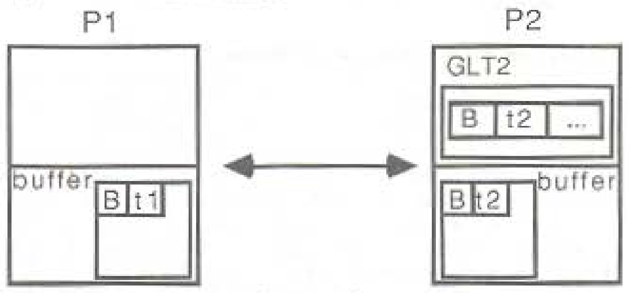


Figure 3: Timestamp solution (example)

Although the timestamp solution does not require extra communication, the release messages for X-locks and the lock response messages (in case of buffer invalidation) are considerably longer than in the basic PCL-scheme. Therefore, the communication overhead may only be tolerable if the PCA distribution together with transaction routing can assure that most block accesses are directed to the local partition.
In order to get a feeling for the required bandwidths, we make a coarse estimation for a 'typical' environment (1000 transactions per second, 50% update transaction, 10 locks per transaction). It is assumed that each update transaction modifies 4 blocks in average and that the block size is 4 KB; lock request and lock response messages should comprise 100 bytes each. Table 1 gives the resulting bandwidth requirements for the three dominating message types and for three different shares of local access (20, 50 and 80%). Note, that the lock response messages can be larger than 100 bytes if they are used to transmit modified blocks. The frequency of these cases depends on the amount of buffer invalidations which is also determined by the transaction routing. In the figures of Table 1 it is assumed that such 'large' lock response messages are necessary for 0.5% of the locks in the case where 80% of the accesses are local and 1% (2.5%) if this share is 50% (20%).
Table 1 shows that the largest fraction of the required bandwidths is due to the release messages used for transmission of modified blocks to the PCA-processor. Although the total bandwidth requirements are still within the technically feasible range, overload situations in the message system are rather likely, especially with a low share of local accesses or if some database partitions are much more frequently modified than others.

	share of local accesses		
	80%	50%	20%
Release messages	1.6	4	6.4
Lock request messages	0.2	0.5	0.8
Lock response messages	0.24	0.7	1.6
Total	2.04	5.2	8.8

Table 1: Bandwidth requirements (in MB/s) if all modified blocks are transmitted to the PCA-processors

In order to avoid such possible bottleneck situations, one can use a revised scheme where the modified blocks need not be propagated to the PCA-processor. In such a scheme, the name of the processor that has performed the last block modification is also stored in the GLT (besides of the timestamp). Thus, the valid page can be requested from this processor when a buffer invalidation is detected. This may result in additional messages and increased waiting times until the newest copy can be accessed; however, the total bandwidth requirements are significantly reduced. The reason for this is that modified pages are only exchanged between processors on demand, i.e. when the latest copy is actually needed in another processor. Since these cases are supposed to be rare (an assumption that could be confirmed by first empirical investigations), the additional delays should be acceptable.
In the next solution to buffer invalidation in the context of PCL, this method for update propagation is also applied and described in more detail.

<u>Invalidation vector solution</u>
This scheme has the advantage that it is no longer necessary to store a timestamp in every block. The GLT does not keep a timestamp for modified blocks, but instead a so-called <u>invalidation vector</u> is maintained. Hence, the PCA-lock manager mainly use two fields in the block entries of the GLT to control buffer invalidations:

```
I : array [1..N] of bit; (* invalidation vector *)
MODIFYING-PROCESSOR: processor that has performed
                     the latest modification.
  (* if no processor is specified, the valid page
     can be found on disk *)
```

The invalidation vector I indicates for each of the N processors whether an obsolete copy of the block may reside in the processor's buffer. This information can be maintained because after the modification of a block B at a processor P, only P has a valid copy of B in its buffer; for all remaining processors a buffer invalidation is possible. Therefore, when the X-lock on B is released the invalidation vector for such a block B is set to '1' for all processors except for the modifying one.
Before a lock request is issued to the PCA-lock manager, it is firstly checked whether the corresponding block already resides in the local buffer. If so, then the invalidation vector allows the PCA-processor to decide whether or not this copy is invalidated. If the requesting processor has no copy of the block or only an obsolete one, then MODIFYING-PROCESSOR specifies the processor where the valid copy may be obtained. After the propagation of the new copy of a block to a requesting processor P, the PCA-lock manager resets I(P) to '0'.
For illustration assume that processor P2 has the PCA for block B, which was most recently changed by processor P3, and that processor P1 has an invalidated copy of B in its buffer (Fig. 4). In the block entry for B in the GLT of P2, P3 is kept as the processor having performed the last modification of B. The invalidation vector I = 110 indicates that P1 and P2 (but not P3) may have an obsolete copy of B in their buffers.
When now P1 issues a lock request message for block B to P2, it is told that there is a copy of B in P1's buffer. Using the invalidation vector, P2 recognizes that the copy in P1 is obsolete and that the correct version of B has to be propagated to P1 if the lock is grantable (a propagation also would be necessary if P1 has no copy of B in its buffer). Since P2 does not know whether the valid block is on disk or only in the buffer of P3, a message is sent to P3 (propagation demand) in order to arrange a correct propagation. P3 sends the lock response to P1 along with the valid block or with the demand to read the block from disk. At P2, the invalidation vector for B is changed to I = 010.
It should be clear that only with three different processors P1, P2 and P3 this procedure has to go through. A good partitioning and load balancing

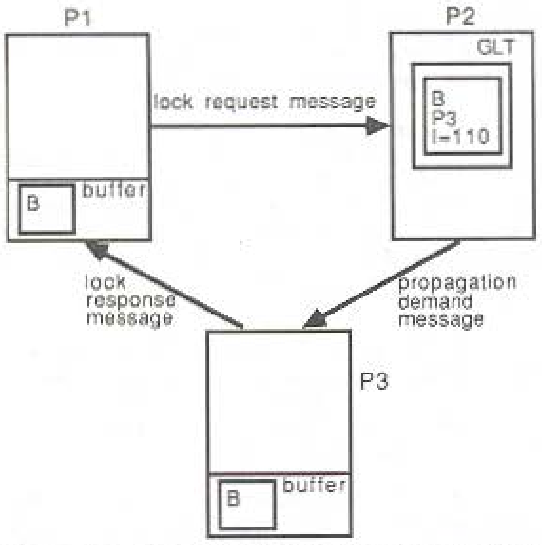


Figure 4: Invalidation vector solution (example)

mechanism, however, should achieve that most (read or update) accesses of blocks are done at the PCA-processors, resulting in simpler treatment and less communication delay for most lock requests. For example, the simplest case would be P1=P2=P3 for avoiding all messages; if P2=P3, the propagation demand would not be necessary and if P1=P2, the lock request message could be saved.

4. Optimistic synchronization

With optimistic concurrency control (OCC), any transaction consists of a read phase, a validation phase, and a possible write phase [10]. During the read phase a transaction performs all updates within a private buffer not accessible by other transactions. The validation has to guarantee serializability of the transactions; conflict resolution relies on transaction abort. The write phase is only required for update transactions which have successfully validated. In that phase, sufficient log data has to be forced to a safe place and the modifications are made visible to other transactions (update propagation).

In [5] two kinds of OCC schemes are distinguished: First, the backward-oriented approach (BOCC), originally introduced in [10], and second, the forward oriented method (FOCC). Here, we only consider the FOCC alternative providing some clear advantages over the BOCC scheme [5].

With FOCC, only update transactions have to validate. Serializability is guaranteed by checking whether there is an overlap between the write set of a validating transaction and the current read set of any transaction not yet finished. For conflict resolution, FOCC offers several alternatives [5], for instance a kill or an abort policy. With the latter, the validating transaction is aborted in case of conflict, whereas with the former all conflicting transactions are aborted.

In DB-sharing systems a FOCC-like synchronization is applicable using a (logical) <u>token ring topology</u> of the processors. Each processor can validate transactions in its master phase only (i.e. when the processor holds the token) guaranteeing that at any point in time no more than one validation is performed in the system. Validation against local transactions can be done with the same techniques as in centralized systems; for validation against non-local transactions, the write sets of locally validated transactions (remember, only update transactions must validate) are sent along with the token (in a so-called buck) to the other processors. A processor must then check all local transactions against these write sets in the buck after receipt of the token.

In such a scheme, the simplest treatment of buffer invalidation and concurrency control is possible if the kill policy is used for global conflict resolution. Therefore, we first discuss this case before the impacts of more complex resolution strategies are investigated.

<u>Kill policy</u>

If the kill policy is used for conflict resolution against external transactions, an update transaction can be committed as soon as it has successfully validated at the local processor, because conflicts at external nodes are resolved by aborting the conflicting transactions. Therefore, the transaction does not need to wait until its write set has completed the ring circulation. Using the kill policy the master phase of a processor P mainly consists of the following steps that should run within a critical section:

1. Global validation.
   The write sets in the buck originating from external transactions are checked against the read and write sets of transactions currently in progress at P. If a conflict occurs the local transaction is aborted (kill policy).
2. Local validation.
   Local update transactions having finished their read phases (and having survived global validation in step 1) are validated against local transactions. For this the kill policy or any other resolution scheme can be applied.
3. Update propagation.
   The updates of external transactions for which a global validation has been performed in step 1 must now be made visible at processor P. The mechanism for that which also removes invalidated pages from P's buffer is explained below. More details about the synchronization protocol itself and why this scheme guarantees serializability can be found in [6].

In order to keep the master phase short, the write sets of locally validated transactions are forwarded within the buck to the next processor before these transactions have completed their write phases (short master phases are extremely important in this scheme, see [6]). This is necessary since the write phases require at least one physical I/O to the log. Nevertheless, for proper synchronization update propagation must assure that no modification is visible to other transactions before the COMMIT-record is written to the log.

For local transactions, this can be achieved as in centralized systems. The modified pages being kept in the private buffer of the updating transaction are made visible as soon as sufficient log data is written to a safe place. For this purpose, the modified blocks are copied from the private buffer

into the local database buffer within a critical section. External transactions cannot access uncommitted modifications, too, since only the write sets but not the data itself are transmitted within the bucks. If an external transaction wants to access such a modified page, an explicit request is required ('propagate-on-demand'). Since the page is not propagated to the requesting processor until the modification is committed, external transactions cannot see 'dirty data'. In order to allow a fast exchange of modified pages, direct communication between any two processors should be possible (e.g. by point-to-point connections).
The write sets in the buck are used to implement such a propagate-on-demand scheme for update propagation as well as to detect buffer invalidations. In the above step 3 of a processor's master phase, the following actions are performed for each block B that belongs to any write set in the buck:

1. If a copy of B resides in the local buffer it is discarded for being obsolete.
2. An entry for B is inserted or adapted in a data structure called MODIFIED-BLOCKS-TABLE (MBT). The MBT which may be organized as a hash table contains an entry for each modified block where the processor is specified that had performed the last update of the block.

Action 1 ensures that no invalidated objects can be accessed in the local buffer. The MBT, updated by action 2, indicates from which processor a modified page must be demanded.
In order to access block B, a transaction T must now apply the following procedure:

```
IF  (T's private buffer contains a copy of B) THEN
                        access this copy;
ELSE  DO;
   IF (the local database buffer holds a copy of B)
     THEN  access this copy;
   ELSE IF (the local MBT keeps an entry for B)
        THEN request a copy of B from the processor
             where B was modified most recently;
        ELSE read B from disk;
END;
```

If neither T's private buffer nor the local database buffer holds a copy of B, the MBT is checked whether B was modified by an external transaction. If so, the block is demanded from the processor where B was modified; otherwise, B can be read from disk.
In order to keep the number of entries within the MBT acceptably small, the processors keep track of the pages, which have been locally modified most recently, that are written to the database on disk due to buffer replacement decisions. The identifier of these pages are sent to all processors within the next buck and allow to remove the corresponding entries from the MBTs.
This simple and effective solution to buffer invalidation and update propagation is only applicable if a pure kill strategy is used for conflict resolution. Unfortunately, the kill policy is not appropriate in many cases because it leads to a high abort rate especially for long transactions [11]. Therefore, we discuss now which extensions of the scheme are required when more flexible resolution strategies are used.

Other strategies for conflict resolution

If not a kill policy is applied, a locally validated transaction can still be aborted by other processors. Therefore, the fate of an update transaction is not determined until its write set has completed one ring circulation. Since response time is increased by the time required for this circulation, it is even more imporant to keep the master phases short. Furthermore, it is difficult to reach modular growth since each additional processor is likely to increase response times.
The above described method for update propagation must also be adapted because the write sets in the buck indicate only possible modifications since the transaction may still be aborted by another processor. Accordingly, it is still undetermined whether or not a local copy of a block that belongs to a write set within the buck is getting obsolete. In order to deal with these possible modifications the block entries in the MBTs are extended to the following structure:

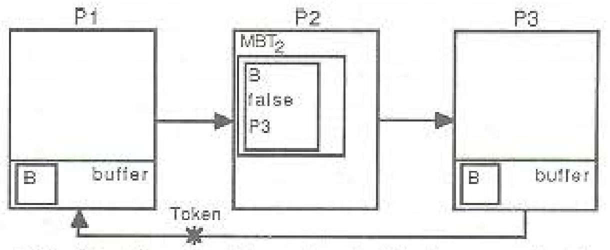
BLOCK-ID:	block identifier;	
MODIFYING-PROCESSOR:		name of the processor where the last successful modification of the block was performed;
IN-DOUBT:	boolean;	(* indicates whether or not the block is subject to a 'possible modification' *);
POSSIBLE-UPDATER:		name of the transaction not yet committed that wants to modify the block;
WAITING-LIST:		list of local transactions that wait until IN-DOUBT = 'false'.

As with the kill policy, the MBT is updated during the master phase after the global and local validation. For each block B belonging to the write set of a (local or external) transaction T that has successfully validated at a processor P, an entry is created or adapted in the MBT of P. Within the entry of B, IN-DOUBT is set to 'true' and T is stored as a POSSIBLE-UPDATER. Since transactions that were aborted at the local processor (or at a preceding processor) are not considered anymore, there is only one POSSIBLE-UPDATER for a block at most. Local transactions that want to access a block for which IN-DOUBT holds, are delayed (within the WAITING-LIST) until the fate of the transaction kept as POSSIBLE-UPDATER is known. With this strategy, it is ensured that always the most recent block can be provided, however, for the price that transactions are sometimes blocked. A more optimistic strategy would be to assume that the modifying transactions will survive and to access their uncommitted modifications. With such a strategy, however, a transaction T that has accessed 'uncertain' data cannot commit until the fate of all POSSIBLE-UPDATERs from which T has received uncommitted data is known. T must be aborted if any of these POSSIBLE-UPDATERs has failed. In the rest of this section, we assume the first ('pessimistic') strategy where only committed modifications are made accessible.
With this scheme, all processors are informed about the fate of an update transaction T as soon as T's write set has completed one ring circulation. If T

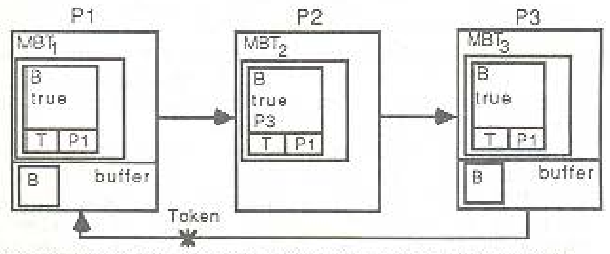
was successful, all processors must be informed that T's modifications are committed. For all blocks belonging to T's write set, the entries in the MBTs are adapted (IN-DOUBT := 'false'; MODIFYING-PROCESSOR := processor where T was executed). If T was aborted only the processors must be notified that are still uninformed about T's fate (in the processor were T was aborted and in the successor processors according to the ring topology, T's write set was not inspected any more). The notifications about T's fate need not be sent within the buck; it seems advisable to inform the other processors directly (e.g. by using a broadcast facility or a point-to-point network) to keep the time of 'uncertainty' as short as possible.
In order to access a block B a transaction T has now to go through the following procedure:

```
IF (T's private buffer contains a copy of B) THEN
    access this copy;
ELSE IF (the local MBT keeps an entry for B)
        THEN DO;
          IF IN-DOUBT (B) THEN delay T until
                    IN-DOUBT (B) = false;
          ELSE send propagation demand to MODIFYING-
                PROCESSOR (B);
        END;
    ELSE read B from the local database buffer or
            from disk;
```

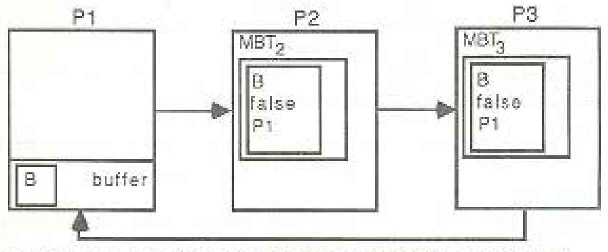
In contrast to the procedure with a kill policy, the local database buffer is here not always consulted in case T's private buffer does not contain a copy of B. The reason for this is that the local buffer may hold blocks that are subject to a possible modification.
The described method for update propagation is now illustrated by the example in Fig. 5. Fig. 5a shows the situation where the most recent copy of block B resides in the buffers of processors P1 and P3. In the MBT at processor P2, P3 is kept as the processor where the last successful modification was performed and that no transaction has notified a possible modification (IN-DOUBT = 'false'). Assume now, that a transaction T at P1 wants to validate and that B belongs to T's write set. After the arrival of the token at P1, T is validated against the write sets in the buck and against local transactions.
Suppose that T has survived these validations. Since B belongs to T's write set, an entry for B is created in P1's MBT with IN-DOUBT = 'true' and POSSIBLE-UPDATER = T. Fig. 5b shows the situation when T has also been successfully validated at P2 and P3 and the buck is on its way from P3 to P1. In that situation no transaction in any processor would be allowed to access block B since it is kept as IN-DOUBT in all MBTs. In Fig. 5c, the situation is depicted when all processors have been informed that T was successful. At P1 the new copy of B is written from the private buffer of T into the local database buffer thereby overwriting the old copy of B; the entry for B in the MBT is not needed any more. At P2 and P3 the entries of B in the MBTs are adapted (IN-DOUBT := false, MODIFYING-PROCESSOR := P1); the copy of B at P3 is discarded for being obsolete. If T had failed to validate at any of the three processors, then the situation of Fig. 5a would have been reestablished.



a) Situation before the token arrives at P1 where transaction T with B $\in$ WS (T) is waiting for validation



b) Situation after successful circulation of T's write set just before the token arrives again at P1



c) Situation when all processors are informed that T has been successfully validated

Figure 5: Use of the MBTs during the validation of a transaction

5. Concluding Remarks

In this paper we have investigated the problems of concurrency control and buffer invalidation in DB-sharing systems. A solution to the latter problem, which is strongly related to the strategy for propagating modified pages to disk (FORCE or NOFORCE), must detect obsolete copies in the local database buffers and provide a method for exchanging modified data between processors.
Since the general broadcast solution (presented in section 2) seems not to be tolerable in high performance environments, we have developed a number of techniques that work in cooperation with one of the two synchronization algorithms discussed and with a NOFORCE-strategy. If the primary copy scheme is used for synchronization, obsolete copies in the buffers can be detected by two basic mechanisms: the use of timestamps, or, even better, by maintaining invalidation vectors. FOCC-like synchronization allows an elegant solution to update propagation and buffer invalidation if a pure kill policy can be used for conflict resolution. Otherwise, access to invalidated objects can also be avoided, however, transactions

may be delayed if they want to access a block being subject to a possible modification.
For update propagation between processors (with NOFORCE) we have always advised a 'propagate-on-demand'-scheme instead of transmitting all modifications to the PCA-processor (in case the primary copy approach is used) or around the token ring (in the FOCC-algorithm). This is because the latter strategy would imply much more message overhead (as was shown for PCL in section 3), although only a small fraction of the modifications is likely to be used in other processors than the modifying one. With the propagate-on-demand scheme, on the other hand, only when a modified page is actually needed, a request is sent to the processor keeping the desired page. The time required for such a page exchange via the communication system should generally be much smaller than a physical read from disk. Furthermore, the number of these request messages can be kept small if an effective transaction routing is possible.
To quantify the usefulness of our proposals to concurrency control and to the treatment of buffer invalidations, we have implemented the primary copy algorithm (with invalidation vectors) as well as the FOCC-scheme within a simulation system. Although our simulations, driven by real-life object reference strings, are not yet completed, it is already safe to say that the communication overhead is kept to a minimum and that the exchange of pages across the communication system results in no performance problems at all. A detailed description of our simulations and an analysis of the results are given in a sequel to this paper.
The description of the integrated solutions has shown that concurrency control and buffer management must cooperate much more than in centralized systems. For instance, the buffer manager must be able to process propagation requests in order to exchange modified pages between processors. Furthermore, the synchronization component can cause that pages that have been recognized as obsolete are discarded from the buffer. In the FOCC-scheme, data is accessed via an MBT that tell where the valid copy of a block is available.
Another important point not discussed thus far are the implications of a processor crash. For instance, during recovery it may be necessary to reconstruct some essential data structures (e.g. lock tables) in order to properly continue synchronization and buffer control. Furthermore, with NOFORCE the redo of transactions successfully executed at the failed processor is more complicated than in centralized systems, since one cannot simply write all after-images of these transactions from the log to the database. This is because one could then overwrite the valid copy of a page by an obsolete after-image. Although a more detailed discussion of the recovery impacts of our algorithms is beyond the scope of this paper, one can say that the integrated solutions tend to complicate recovery protocols. This seems to be the price for efficiency during normal processing.

References

1. Borr, A.J.: Transaction Monitoring in ENCOMPASS. Proc. VLDB, 155-165 (1981).
2. Effelsberg, W., Härder, T.: Principles of Database Buffer Management. ACM TODS, 9 (4), 560-595 (1984).
3. Dias, D.M., Iyer, B.R., Robinson, J.T., Yu, P.S.: Design and Analysis of Integrated Concurrency-Coherency Controls, IBM Research Report RC 11557 (1985).
4. Gray, J. et al.: One Thousand Transactions per Second. Proc. IEEE Spring CompCon, San Francisco, 96-101 (1985).
5. Härder, T.: Observations on Optimistic Concurrency Control Schemes. Information Systems 9 (2), 111-120 (1984).
6. Härder, T., Peinl, P., Reuter, A.: Optimistic Concurrency Control in a Shared Database Environment. Dept. of Computer Science, Univ. of Kaiserslautern/Stuttgart, preliminary version, (1985).
7. Härder, T., Rahm, E.: Multiprocessor Database Systems for High Performance Transaction Systems. Informationstechnik, 28(4), 214-225 (1986), in German.
8. Härder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery. ACM Computing Surveys, 15(4), 287-317 (1983).
9. Keene, W.N.: Data Sharing Overview. in: IMS/VS V1, DBRC and Data Sharing User's Guide, Release 2, G30-5911-0 (1982).
10. Kung, H.T., Robinson, J.T.: On Optimistic Methods for Concurrency Control. ACM TODS, 6 (2), 213-226 (1981).
11. Peinl, P., Reuter, A.: Empirical Comparison of Database Concurrency Control Schemes. Proc. 9th Int. Conf. on VLDB, 97-108 (1983).
12. Rahm, E.: Concurrency Control in DB-sharing Systems. Proc. 16th Annual GI Conf., Springer, Informatik Fachberichte 126, 617-632 (1986).
13. Rahm, E.: Closely Coupled Architectures for DB-sharing. Proc. 9th NTG/GI Conf. on Computer Architecture and Operating Systems, VDE-Verlag, 166-180, (1986), in German.
14. Rahm, E.: Primary Copy Synchronization for DB-sharing. Information Systems, 11(4), 275-286 (1986).
15. Reuter, A.: Load Control and Load Balancing in a Shared Database Management System. Proc. 2nd Data Engineering Conf., 188-197 (1986).
16. Reuter, A., Shoens, K.: Synchronization in a Data Sharing Environment. IBM San Jose Research Lab., preliminary version (1984).
17. Sekino, A. et al.: The DCS - A New Approach to Multisystem Data Sharing, Proc. National Comp. Conf., 59-68 (1984).
18. Shoens, K. et al.: The AMOEBA Project. Proc. IEEE Spring CompCon, 102-105, (1985).
19. William, R. et al.: R* - An Overview of the Architecture, in: Improving Database Usability and Responsiveness, P. Scheuermann (ed.), Academic Press, 1-27 (1982).
20. West, J.C. et al.: PERPOS Fault-Tolerant Transaction Processing. Proc. 3rd Symp. on Reliability in Distributed Software and Database Systems, 189-194 (1983).

This work was financially supported by SIEMENS AG, Munich.