Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment

C. Mohan
Inderpal Narang

Data Base Technology Institute, IBM Almaden Research Center, San Jose, CA 95120, USA
{mohan, narang}@ibm.com

Abstract This paper proposes schemes for fast page transfer between transaction system instances in a shared disks (SD) environment where all the sharing instances can read and modify the same data. Fast page transfer improves transaction response time and concurrency because one or more disk I/Os are avoided while transferring a page from a system which modified it to another system which needs it. The proposed methods work with the *steal* and *no-force* buffer management policies, and fine-granularity (e.g., record) locking. For each of the page-transfer schemes, we present both recovery and coherency-control protocols. Updates can be made to a page by several systems before the page is written to disk. Many subtleties involved in correctly recovering such a page in the face of single system or complex-wide failures are also discussed. Assuming that each system maintains its own log, some methods require a merged log for restart recovery while others don't. Our proposals should also apply to distributed, recoverable file systems and distributed virtual memory in the SD environment, and to the currently popular client-server object-oriented DBMS environments where the clients cache data.

1. Introduction

One approach to improving the capacity and availability characteristics of a single-system transaction system (e.g., a data base management system (DBMS)) is to use multiple systems. There are two major architectures in use in the multisystem environment: ***shared disks (SD)*** or also called ***data sharing*** [DIRY89, MoNa91b, MoNP90, MoNS90, Rahm86, Rahm89, Shoe86], and ***shared nothing (SN)*** or also called ***partitioned*** [Ston86]. With SD, all the disks containing the data base are shared among the different systems. Every system that has an instance of the transaction system executing on it may access and modify any portion of the data base on the shared disks. Since each transaction system instance has its own buffer pool and because conflicting accesses to the same data may be made simultaneously from different systems, the interactions among the systems must be controlled via various synchronization protocols. This necessitates the use of global locking facilities and protocols for the maintenance of the coherency of the data buffered (*cached*) in the different systems. SD is the approach used in IBM's IMS/VS Data Sharing product [StUW82], TPF product [Scru87] and the Amoeba research project [MoNa91b, MoNP90, MoNS90, SNOP85], and in DEC's VAX DBMS[1] and VAX Rdb/VMS[1] [KrLS86, ReSW89]. More recently, for the VAXcluster[1] environment, third-party DBMSs like ORACLE[1] and INGRES have been modified to support SD. Hitachi and Fujitsu also have products which support the SD environment. SD has also, of late, become popular in the area of distributed virtual memory [Li88, WuFu89].

With SN, each transaction system instance *owns* a portion of the data base and only that portion may be directly read or modified by that instance. That is, the data base is *partitioned* amongst the multiple systems. The kind of synchronization protocols mentioned before for SD are not needed for SN. But, a transaction accessing data in multiple systems would need a form of two-phase commit protocol (e.g., the industry-standard Presumed Abort protocol of [MoLO86]) to coordinate its activities. SN is the approach taken in Tandem's NonStop SQL[1] [Tand87], Teradata's DBC/1012[1] [Nech88], MCC's Bubba [BACCD90], and University of Wisconsin's Gamma [DGSBH90]. There are many advantages and disadvantages with both SD and SN [Bhid88, PMCLS90, Shoe86, Ston86]. Our intention in this paper is not to argue the relative merits of the two approaches. Even though major products have come out which support either SD or SN, the debate still goes on. We concentrate on solving some problems relating to SD.

The rest of the paper is organized as follows. In the remainder of this section, we first introduce the buffer-coherency problem and IMS's solution for handling it. Then, we state the assumptions that we make in proposing our solutions. In section 2, we provide a brief overview of the different page-transfer schemes. The details of the Medium, Fast and Super-Fast schemes with record locking are covered in section 3. Due to space constraints in this paper, we do not discuss the optimized versions of these protocols that are possible when the granularity of *logical* locking is a page, rather than a record [MoNa91b]. In section 4, we compare our proposals with existing proposals and implementations by others. Lastly, in section 5, we summarize our contributions.

1.1. Buffer Coherency and Page Transfer

In a single-system transaction system, if a transaction were to update a record in a page, then that update is made *visible* to other transactions once the updating transaction, after committing, releases the (exclusive) lock on the record. This visibility is the result of performing the update in the same buffer pool which is shared among all the transactions. We call this ***instant propagation of updates***. However, in SD, when a transaction updates a record in one system, the update is not reflected instantly in the other systems' buffer pools. This is the ***buffer-coherency*** problem. Special protocols must be used to ensure that transactions do not see data that is not current. An updated page can be propagated by the updating system to the other sharing systems in many ways such as via disk only, or disk and intersystem communication links, or links only. A factor which plays a role in how propagation could take place is whether the single-system transaction system writes *pages* updated by a transaction to disk at commit time. The latter policy is followed by IMS, VAX DBMS, VAX Rdb/VMS, etc.

In IMS Data Sharing, propagation of updates takes place as follows: (1) the updater writes the page to disk, (2) *after* the disk I/O completes, the updater sends a message to the other systems to invalidate their cached, if any, copies of the page, (3) the other systems acknowledge the invalidation messages *after* their cached copies have been marked invalid, (4) the updater releases its (exclusive) locks on modified data *after* receiving all the acknowledgements, and (5) the other systems, when they need to access the cached copies which have been marked invalid, read the page from disk to get a more recent copy of the page. Actions (1) and (2) typically happen at commit time, thereby increasing the lock hold time and the transaction response time. For IMS, propagation via disk is a natural approach to take because (1) IMS in the single-system environment, before releasing the exclusive locks of a transaction, writes the updated pages to disk (follows the ***force*** policy) anyway and (2) IMS in the SD environment supports only *page-level* concurrency between systems for updates.[2]

If a single-system transaction system like DB2[1], which does not write updated pages to disk at commit time (follows the ***no-force*** policy), were enhanced to operate in the SD environment, then update propagation via disk and sending of invalidation messages would be very expensive in terms of concurrency and transaction response time, when compared to a single-system environment. Then, in this context, the following question arises: how are updates to be propagated when the updater neither writes the page to disk nor sends invalidation messages? One possible answer is: the updater (1) leaves a *trail* with the global lock manager using which the other systems can detect that their cached versions of the page are not current, and (2) sends the updated page quickly to the other systems *when they need it*. The former is called ***detection*** and the latter ***reaction***. Since detection takes place when the transactions in the other systems actually need a more current version of the page, it is imperative that we make the reaction part execute as fast as possible. Therefore, in this paper, we describe schemes for fast page transfer between systems and their recovery implications in case of a variety of failures. The detection technique that we employ is similar to the techniques used in [Rahm86], which calls it *on-request invalidation* (*Check-on-Access*), and [DIRY89], and in DEC's VAXcluster file system [KrLS86], VAX DBMS and VAX Rdb/VMS [ReSW89].

Fast page transfer improves transaction response time and concurrency because one or more disk I/Os are avoided while transferring a page from a system which modified it to another system which needs it. This permits updates to be performed on a page by several systems before the page is written to disk, thereby further increasing concurrency and amortizing the cost of disk writes. With such a flexible scheme, care must be taken to ensure that recovery is performed correctly should failures occur. For the schemes that we propose, we ensure that recovery is performed correctly in the face of (1) loss of messages and (2) single system or complex-wide failures. The proposed schemes work with the *steal* and *no-force* buffer management policies [HaRe83] and fine-granularity (e.g., record) locking. Assuming that each system maintains its own log, some of our schemes require a merged log for *restart* recovery while others don't. Of course, a merged log will always be needed for *media* recovery. We also present some techniques for enhancing data availability when one or more systems fail while holding some important locks. Our proposals should also apply to distributed, recoverable file systems and distributed virtual memory in the SD environment, and to the client-server architecture environments where the clients cache data obtained from the server. The latter has become popular in the object-oriented data base area [CFLS91, DMFV90, WiNe90].

1.2. Assumptions

In proposing our solutions, we make the following assumptions about the transaction system.

Log Management For performance reasons, each of the systems maintains its own log to which log records are first written. For the purpose of handling data recovery, one system in the SD complex which has connectivity to all the local logs' disks produces a merged version of those logs. A standby log merge process is available to take over in case the current merge process fails. The local log manager associates with each log record a ***log sequence number*** (***LSN***) which is a monotonically increas-

1 DB2 and IBM are trademarks of the International Business Machines Corp. NonStop SQL and Tandem are trademarks of Tandem Computers, Inc. DEC, VAX DBMS, VAX, VAXcluster and Rdb/VMS are trademarks of the Digital Equipment Corp. Oracle is a registered trademark of the Oracle Corp. DBC/1012 is a trademark of the Teradata Corp.

2 In IMS, updating transactions, in addition to acquiring commit-duration global exclusive locks on *records* that they modify, also obtain *commit-duration* global exclusive locks on the *pages* which contain those modified records. These global exclusive page locks do not prevent multiple updating transactions within the same system from modifying the same page concurrently. IMS supports *record-level* concurrency between an updating system and reading systems of a page since a transaction *reading* records in a page acquires global (share) locks only on records and not on the page.

ing value. Typically, in single-system (nonSD) transaction systems, LSNs are the logical addresses of the corresponding log records [MHLPS89]. At times, version numbers or timestamps are also used as LSNs [MoNP90]. Here, we are assuming that the LSN is a timestamp and that the clocks across the SD complex are perfectly synchronized.

Recovery Recovery is based on ***write-ahead logging* (*WAL*)**. In WAL systems, an updated page is written back to the same disk location from where it was read. That is, **in-place updating** is performed on disk. Even in the buffer pool, in-place updating is performed. The ***WAL protocol*** asserts that the log records representing changes to a page must already be on stable storage **before** the changed page is allowed to replace the previous version of that page on disk. Every page in the data base has a ***page_LSN*** field which contains the LSN of the log record that describes the *latest* update to that page. This allows the page state to be related precisely with respect to the log records that have been written for that page in order for recovery to be performed correctly. The buffer manager also uses the page_LSN information to ensure that the log has been written to stable storage (***forced***) up to that LSN before it writes the modified page to disk. We are assuming that an ARIES-style [MHLPS89, MoPi91, RoMo89] recovery method is in use. This means that the so-called ***compensation log records (CLRs)*** will be written to describe the updates that are performed as a result of rolling back some actions of a transaction. Writing CLRs allows us to think of the page state, as reflected by page_LSN, as always going forward in time, even though at times some earlier updates of a transaction might be getting undone [MHLPS89].

Lock Management Locking is managed by a global lock manager (***GLM***) in conjunction with one local lock manager (***LLM***) in each system. When it is not necessary to distinguish between LLM and GLM, we use the generic term ***LM*** (lock manager). The transaction system makes its lock request to its LLM which may then forward it to GLM. This is similar to the way lock management is done in the DEC VAXcluster [KrLS86] for the SD environment. Such a lock manager provides global locking functions for its clients but it does *not* perform disk I/Os for them. LM assists a transaction system instance in determining which transaction system instance, if any, has a dirty version of a particular page. A page version is considered to be ***dirty*** if the buffer pool (***cached***) version of the page is more recent than the disk version of the page. Given a message and a lock name, LM can send the message to the current holder(s) of that lock. This is called the ***notify*** mechanism. In order to deal with a failure of GLM, a backup GLM is defined and it monitors the state of the primary GLM to determine when to take over. When a backup GLM takes over, it communicates with LLMs to reconstruct GLM's global lock table information. When GLM notices that an LLM has failed, it will release all the locks, except those that were specifically asked to be ***retained***, that were held by the failed LLM. To recover from the failure of multiple systems in the SD complex, GLM's lock tables are periodically checkpointed. Such a failure is treated as an ***SD-complex failure***. It should be emphasized that the focus of this paper is not on how to build a highly available LM. The design assumed above can be easily changed to produce a more distributed LM.

Buffer Management The buffer manager (***BM***) is free to adopt the very flexible policies of steal and no-force [HaRe83]. If a page modified by a transaction is allowed to be written to disk before that transaction commits, then the ***steal*** policy is said to be followed by BM. Otherwise, the ***no-steal*** policy is said to be in effect. Steal implies that during normal or restart rollback, some undo work might have to be performed on the disk version of the data base. If a transaction is *not* allowed to commit until all *pages* modified by it are written to disk, then the ***force*** policy is said to be in effect. Otherwise, the ***no-force*** policy is said to be in effect. With the force policy, during restart recovery, no redo work will be necessary for committed transactions. No-force decreases lock hold times. It also allows write I/Os to disk to be performed more efficiently by writing multiple pages in one I/O and by amortizating the cost of a disk write of a page over updates made by several transactions. Many more arguments in favor of adopting the no-force and steal policies are given in [MHLPS89].

Locking and Coherency-Control Protocols We do not wish to permit the same page to be updated concurrently in different systems since that would require that a mechanism exist to merge those updates into a single version of the page. Furthermore, we could have difficulties with storage management when fine-granularity (e.g., record) locking is supported in the most general way (e.g., as in ARIES/IM [MoLe89] and ARIES/KVL [Moha90]). To avoid these problems, we use a ***physical* (*P*)** lock on a page to *serialize* the updating of that page by multiple systems. Note that, unlike ***logical* (*L*)** locks which are held for the duration of a transaction, physical locks are not held for the duration of a transaction. P locks will never be involved in deadlocks, unlike L locks. In this paper, P locks are acquired only on pages, while L locks are acquired only on records. Hence, later in the paper, we do not always identify the type (L or P) of a lock explicitly. P locks need to be held only as long as a system is caching a page in its buffer pool. P locks are acquired by BM on behalf of the transaction system while L locks on records are acquired by the data manager on behalf of individual transactions. To update (read, respectively) a record, the transaction gets an X (S) lock on the record. The compatibility relationships amongst the different modes of locking, for both L and P locks, are shown in Figure 1. A check mark ('√') indicates that the corresponding modes are compatible. That is, in the case of such an entry, if transaction T1 were to hold the lock in the mode indicated by the row and T2 were to request the lock in the mode indicated by the column, then T2's request will be granted immediately.

	S	U	X
S	√	√	
U	√		
X			

Figure 1: Lock Mode Compatibility Matrix

P locks are also used to detect that a cached page in a particular buffer pool is not the latest version of the page. BM gets an S lock on a page before caching it in the local buffer pool. This lock is held as long as the page remains cached in the local buffer pool. Before allowing a transaction to dirty a clean page, BM gets a U lock on the page. This lock must be held by BM as long as the page remains dirty and it is cached by this BM. As a result of these locking protocols, the intersystem concurrency is multiple readers and an updater per page as far as the buffer managers are concerned. Of course, because of the record locking performed by transactions, a given page can contain at any time the uncommitted updates of transactions running in any number of the sharing systems. LM will be asked to retain all the U and X locks.

Communications The different systems in the complex are directly connected to one another via high-speed communication links (***Comm-links***) whose performance is orders of magnitude better than that achievable by communicating via the shared disks. When pages are shipped directly between the systems, a datagram protocol is used with no guarantees about delivery. Using datagrams is important to assure that the cost of shipping pages directly is not high.

Distributed Transactions For simplicity, we assume here that each transaction executes entirely within a transaction system instance. It is easy to extend the proposed schemes to work in a complex in which a single transaction might span multiple transaction system instances in order to exploit parallelism even more than what is possible within a single system [PMCLS90]. With this assumption, we are not precluding the possibility of the transactions executing in this complex being distributed transactions which also access data outside of this complex. For distributed data base management purposes, this complex is thought of as a single *node* of the distributed system.

2. Overview

In an SD environment, if a *dirty* page is cached in one system (referred to as the **owner**), then a different system requiring access to that page must get the current version of the page from the owner. P locks are used to detect that a cached page in a particular buffer pool is not the latest version of the page. When a transaction T1 updates a record in page P1 in system S1, it updates P1's page_LSN. When T1 commits, P1's current page_LSN is sent to LM along with the unlock requests for the L locks. LM registers the page_LSN in the P lock entry for P1, *before* unlocking the L locks. When T2 in S2 locks a record in P1, it requests P1's page_LSN. When LM returns the latter as part of granting the L lock, S2 can detect if its cached version, if any, of P1 is not current. If P1 in S2 is not current, then LM assists S2 in getting the current version of P1 from P1's owner S1.

Before updating a page, a system which is not already the owner of the page must first become the owner of the page by acquiring a U mode P lock on the page. The ownership of a page can be given up by a system only after the dirty page is written to disk or as part of transfer of ownership to another system. The current owner of a page could transfer the page to a requesting system

1. by writing the page to disk and then making the requestor read it from disk, or
2. by a memory-to-memory transfer.

With memory-to-memory transfer, the response time and concurrency advantages are the same as with caching a dirty page in a single-system environment. This is because, with memory-to-memory transfer, it is possible to save 2 disk I/Os - a write I/O by the owner and a read I/O by the requestor.

Next, we give a brief description of the mechanics of a page transfer and how LM assists a requestor in getting the latest verison of a page.

When BM acquires a U mode P lock to **update** a page, it declares to LM that BM's ***page-transfer procedure*** should be invoked if (1) there is an intersystem lock conflict involving that page (i.e., another system wants to update the page), or (2) the requestor, in another system, makes a nonconflicting lock request for that page (i.e., another system wants to read the page). BM will then make the page available to the requestor.

BM can transfer the page to the requesting system using one of the following schemes: ***Simple, Medium, Fast***, and ***Super-Fast***. For each scheme, we compare the number of I/Os and messages to obtain for *updating* purposes a dirty page cached in another system. This comparison is done to motivate the development of faster page-transfer schemes.

A transaction in the requesting system initiates access to a page by invoking its BM via the ***fix_page*** call (also called ***pin***). The fix_page request will also indicate whether the transaction intends to update the page. If the page is not already cached, then BM will request a P lock for the page in the appropriate mode (S for read access and U for update). If the page is cached but the request is for update and the current system is not already the owner of the page, then BM will make a request to upgrade (from S to U mode) the P lock for the page. This request triggers LM to invoke the page-transfer procedure in the owning system, if there is one. BM in the owning system then transfers the page. The owning BM, if necessary, downgrades its P lock so that the requesting BM's lock is granted. The downgrading from U to S would be required if the request is for a U lock. The subsequent fix_page processing depends on the scheme used for the page transfer.

Next, we briefly describe four page-transfer schemes. In this section, we consider the case where an owner for a page exists and another system wants to become the *owner*. The schemes are also applicable when the second system only wants to read the page. The latter case is covered in the section "3. Details of the Page-Transfer Schemes".

2.1. Simple Scheme

In this scheme, the following actions occur on the owner and the requestor sides. We are not showing the lock request message from the requestor to GLM and the lock grant message from GLM to the requestor since these costs are common to all the schemes discussed here.

- A lock conflict message is sent from GLM to the owner.
- A disk I/O is performed by the owner to write the page to disk. Due to the WAL protocol, there will be an implicit log force.
- *After* the disk I/O completes, a message is sent from the owner to GLM for downgrading the P lock to S mode.
- A disk I/O is performed by the requesting system to read the page from disk.

Therefore, using the Simple scheme, the costs of a page transfer are 2 messages, 2 I/Os and, possibly, a log force. As it should be obvious, the simple scheme is very costly compared to the single-system case where BM locates the updated page in memory at CPU speed. These additional costs increase transaction response time and decrease concurrency. Hence, the need for more efficient page-transfer schemes. In IMS, VAX DBMS, VAX Rdb/VMS and Oracle 6.2 in a VAXcluster, a page is transferred from one system to another via disk. This simple scheme is not discussed further in the rest of this paper.

2.2. Medium Scheme

Continuing with the above example, the Medium scheme differs as follows: The owning BM writes the page to disk and *simultaneously* ships the page directly to the requestor using the comm-link. After the disk write is complete, the owning BM downgrades the P lock to the S mode. Then, LM grants the lock to the requestor.

The costs involved in accessing a page using the Medium scheme are as follows:

- A lock conflict message from GLM to the owner.
- A disk I/O by the owner to write the page to disk. Due to the WAL protocol, there will be an implicit log force.
- A message to send the page directly to the requesting BM.
- A message from the owner to GLM for downgrading the P lock.

Therefore, using the Medium scheme, the costs of a page transfer are 3 messages, 1 I/O and, possibly, a log force. This scheme is more efficient than the Simple scheme, because (1) the page transfer is a memory-to-memory transfer which should be much faster than a disk I/O and which should reduce the contention on the disk arm, and (2) the page transfer is overlapped with the disk write. Of course, there is no guarantee that the requestor would receive the shipped page in a timely manner or receive it at all. Therefore, care has to be taken so that the requestor does not (1) wait forever for the page to arrive, or (2) use a stale version of the page. The details about the avoidance of such problems are described in the section "? on page ?".

2.3. Fast Scheme

The Fast scheme differs from the Medium scheme as follows: the owner BM does *not* write the page to disk. In the page-transfer procedure, the owning BM issues, if necessary, a log force for ensuring the WAL protocol, ships the page to the requestor and then downgrades the P lock. Then, LM grants the lock to the requestor. With the Fast scheme, the costs of a page-transfer are 3 messages, no disk I/O and, possibly, a log force. The disk write, which in the Simple and Medium schemes causes most of the delay, is entirely eliminated in this scheme.

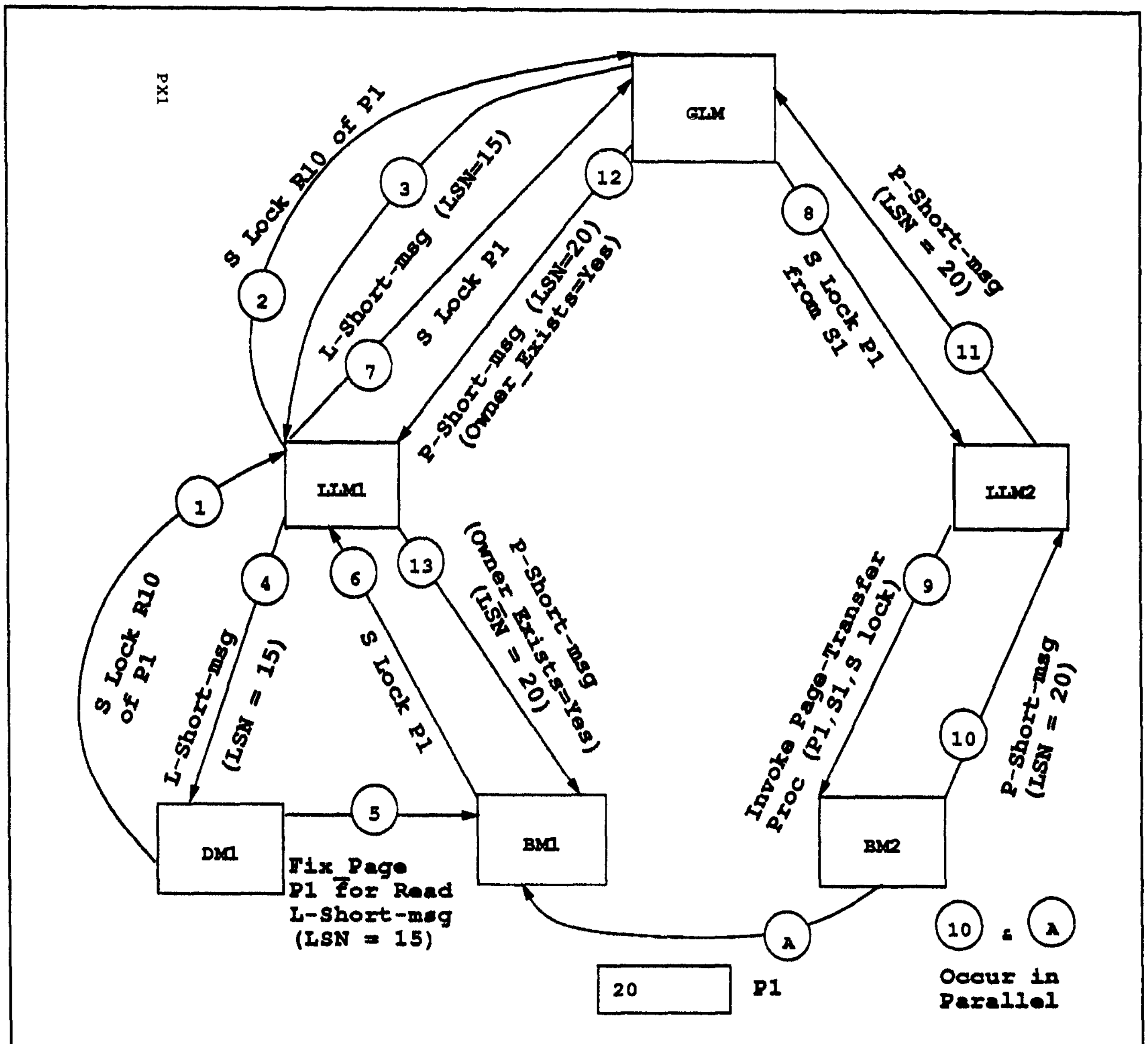
In the normal case, the Fast scheme provides better response time and concurrency than the Simple and Medium schemes. However, it complicates page recovery since a *dirty* page may be transferred from one system to another and since it may contain updates from more than one system. In the Simple and Medium schemes, a dirty page contains updates of only the owning system. Hence, with the Fast scheme, during recovery from a system failure or on noticing the nonarrival of a shipped page, a merged log of all the systems may be required to recover a dirty page. Since a dirty page may contain updates from multiple systems which have not been reflected in the disk version of the page, for each such page, the transaction system records a value called the ***Recover LSN*** (***RLSN***) at GLM. An RLSN is the earliest log point in the merged log from where the log must be scanned to redo the changes logged for the associated page in case the system owning the page were to fail before writing the page to disk. Since the clocks across the SD complex are synchronized, we use a timestamp as the value of RLSN. In the section "3.3.1. Assigning and Tracking Recover LSN", we discuss how the RLSN value at GLM is manipulated.

2.4. Super-Fast Scheme

With the Super-Fast scheme, the owner is *not* required to ensure that the log is forced up to the LSN of the page before shipping the page. With this scheme, the costs of a page transfer are 3 messages, no disk I/O, and no log force. However, in order to ensure that the WAL protocol is followed before the dirty page is written to disk by some owning system ultimately, this scheme requires the tracking of the LSN values associated with a dirty page on a per system basis for all the systems whose updates to the page have not yet been reflected in the disk version of the page. For each updating system, the LSN to be remembered is the LSN of the page when the page was shipped by that system to some other system. The page can be written to disk only after all those updating systems have forced their respective logs up to the LSNs being tracked. Note that this is required since we have assumed that each system has its own log which makes the log force of each system independent of those of the other systems.

3. Details of the Page-Transfer Schemes

In this section, we describe the Medium, Fast and Super-Fast schemes in detail when the granularity of *logical* locking is a record within a page. Due to space constraints, we do not discuss here the optimized versions of these protocols that can be used when the granularity


PX1
GLM
LLM1
LLM2
DM1
BM1
BM2
1 S Lock R10 of P1
2 S Lock R10 of P1
3 L-Short-msg (LSN=15)
4 L-Short-msg (LSN = 15)
5 Fix Page P1 for Read L-Short-msg (LSN = 15)
6 S Lock P1
7 S Lock P1
8 S Lock P1 from S1
9 Invoke Page-Transfer Proc (P1,S1,S lock)
10 P-Short-msg (LSN = 20)
11 P-Short-msg (LSN = 20)
12 P-Short-msg (LSN=20) (Owner_Exists=Yes)
13 P-Short-msg (Owner_Exists=Yes) (LSN = 20)
A
20 P1
10 & A Occur in Parallel

In this scenario, a transaction T, in system S1, finds out, via the L-Short-message returned on L locking record R10 in page P1, that it needs to access P1's version with an LSN that is at least as high as 15. Buffer manager BM1 in S1 has no cached version of P1, but BM2 in S2 has a dirty version of P1. BM1, by P locking P1 in S mode and with the assistance of LM, acquires P1 with LSN 20 from the owner (BM2). The P-Short-message sent by BM2 informs BM1 that the shipped page has 20 as its LSN. Since only an S mode P lock is requested, BM2's Page-Transfer procedure does not write P1 to disk, irrespective of the page-transfer scheme.

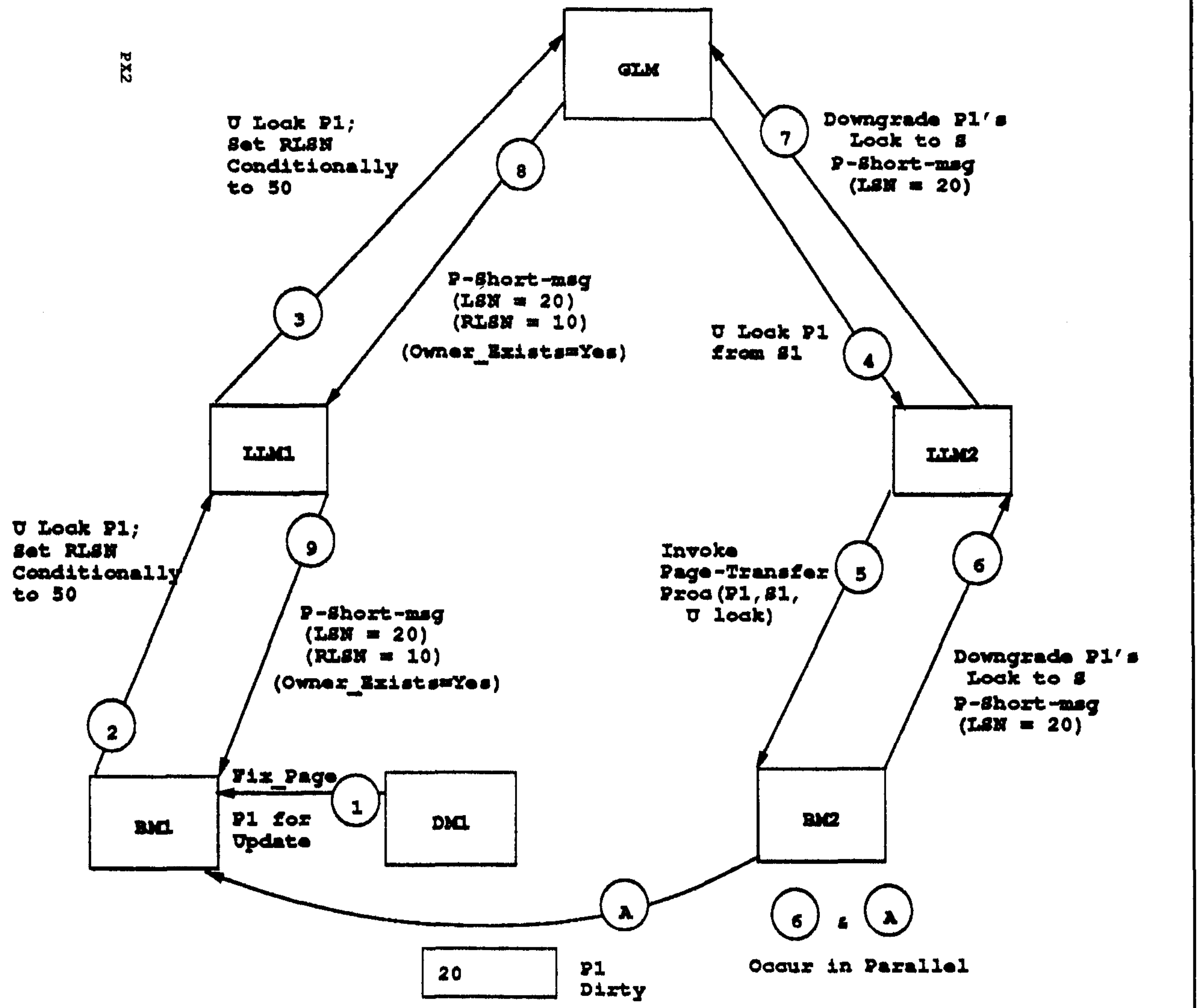
Figure 2: Scenario Showing Logical and Physical Locking, and Page Transfer

of *logical* locking is a page, rather than a record [MoNa91b].

We illustrate message flows relating to locking and page transfer for 4 different scenarios in Figure 2, Figure 3, Figure 4, and Figure 5. The figures include descriptions of the illustrated scenarios. We will refer to these figures in the following descriptions. Before we delve into the details of the different page-transfer schemes, we discuss, in the next subsection, how the lock manager assists in dealing with the buffer-coherency problem.

3.1. Lock Manager's Coherency Assists

For record locking, LM assists in maintaining a page coherent in the following ways:


FX2
GLM
U Lock P1; Set RLSN Conditionally to 50
8
3
P-Short-msg (LSN = 20) (RLSN = 10) (Owner_Exists=Yes)
7
Downgrade P1's Lock to S P-Short-msg (LSN = 20)
U Lock P1 from S1
4
LLM1
LLM2
U Lock P1; Set RLSN Conditionally to 50
9
P-Short-msg (LSN = 20) (RLSN = 10) (Owner_Exists=Yes)
2
Invoke Page-Transfer Proc(P1,S1, U lock)
5
6
Downgrade P1's Lock to S P-Short-msg (LSN = 20)
Fix_Page P1 for Update
1
BM1
DM1
BM2
A
6 & A
Occur in Parallel
20
P1 Dirty

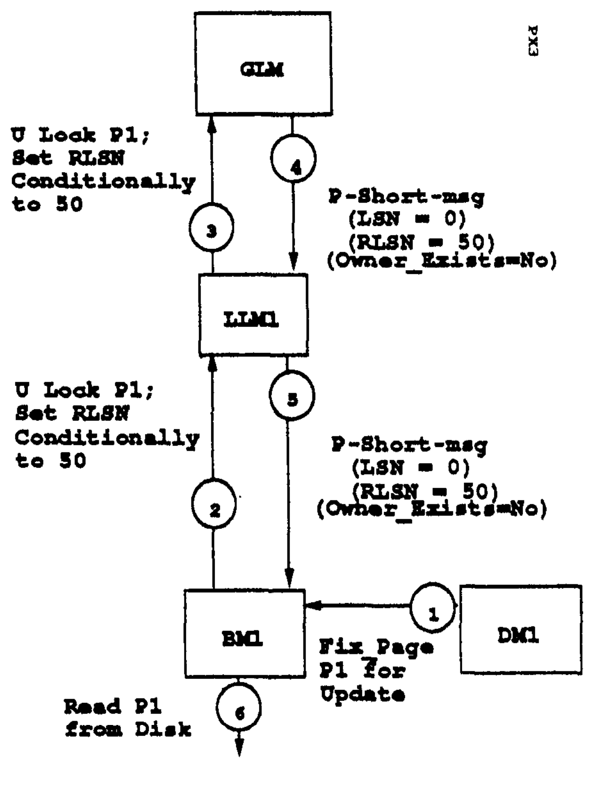
In this scenario, a transaction T, in system S1, wants to update page P1. Buffer manager BM1 in S1 has no cached version of P1, but BM2 in S2 has a dirty version of P1. BM1, by P locking P1 in U mode and with the assistance of LM, acquires the current version of P1 from BM2. BM2's Page-Transfer procedure downgrades its U mode P lock on P1 to the S mode. The P-Short-message sent by BM2 informs BM1 (the new owner) that the shipped page has 20 as its LSN. When BM2 ships the page to BM1, it indicates that the page is dirty. This means that either the Fast or the Super-Fast scheme is being used for page transfer. LM ignores the RLSN value of 50 provided by BM1 since the page is already dirty and its current RLSN is 20.

Figure 3: Scenario Showing Physical Locking, and Page and Ownership Transfer

In each P lock's lock table entry, LM assigns a field for keeping track of the LSN of the associated page. This field is in addition to the RLSN, the log point for page recovery, mentioned before. LM initializes the LSN field to zeroes. LM replaces this field's value in its lock table entry only when an LSN provided by a system is greater than the currently stored value. Each lock table entry for a held P lock would have at least the following pieces of information:

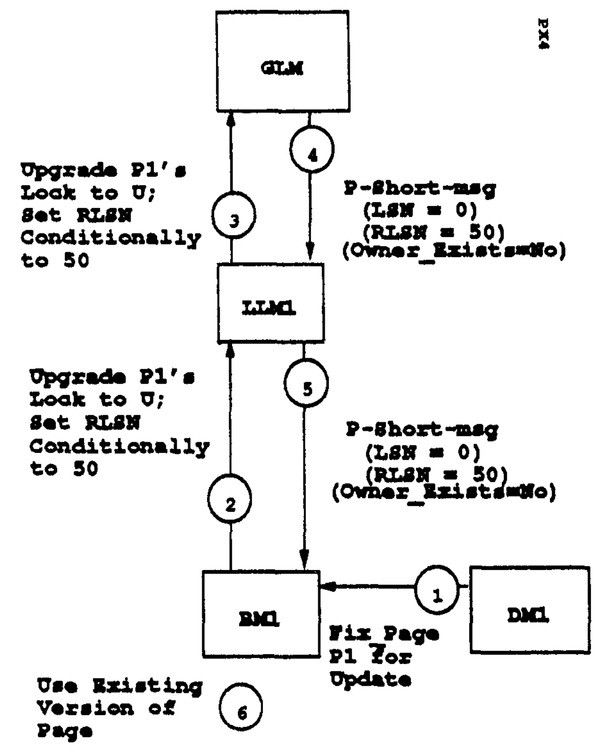
Page Lock Name	Recover LSN (RLSN)	Current LSN (LSN)	Lock Holder, Waiter Info, ...

With an unlock request, LM accepts a list of lock names and their associated LSNs. When a transaction terminates and the transaction system issues an unlock call to release all the (L) locks held by the transaction, it also sends, along with the unlock request, a list of *page*


GLM
U Lock P1; Set RLSN Conditionally to 50
P-Short-msg (LSN = 0) (RLSN = 50) (Owner_Exists=No)
LLM1
U Lock P1; Set RLSN Conditionally to 50
P-Short-msg (LSN = 0) (RLSN = 50) (Owner_Exists=No)
BM1
Fix Page P1 for Update
DM1
Read P1 from Disk
FX3

In this scenario, a transaction T, in system S1, wants to update page P1. BM1 has no cached version of P1 and currently there is no owner for P1. BM1 P locks P1 in U mode and LM sets the RLSN of P1 to the value 50 provided by BM1 since there is no previous owner for the page. On being told, via the P-Short-message generated by GLM, that there is no previous owner for P1, BM1 reads the page from disk.

Figure 4: Scenario Showing Ownership Acquisition with No Previous Owner and No Cached Version of Page


GLM
Upgrade P1's Lock to U; Set RLSN Conditionally to 50
P-Short-msg (LSN = 0) (RLSN = 50) (Owner_Exists=No)
LLM1
Upgrade P1's Lock to U; Set RLSN Conditionally to 50
P-Short-msg (LSN = 0) (RLSN = 50) (Owner_Exists=No)
BM1
Fix Page P1 for Update
DM1
Use Existing Version of Page
FX4

In this scenario, a transaction T, in system S1, wants to update page P1. BM1 already has a cached version of P1 and currently there is no owner for P1. BM1 upgrades its P lock on P1 to the U mode and LM sets the RLSN of P1 to the value 50 provided by BM1 since there no previous owner for the page. On being told, via the P-Short-message generated by GLM, that there is no previous owner for P1 and that the LSN of the page is 0, BM1 realizes that its cached version is current and continues to let it be used by its transactions.

Figure 5: Scenario Showing Ownership Acquisition with No Previous Owner and a Cached Version of Page

(P) lock names and those pages' current LSNs to LM. The purpose of passing the list with the unlock call is to register the LSNs of the updated pages so that other systems which have cached those pages may verify the currency of their pages. It is using this information that the page incoherency *detection* problem discussed before is solved. LM updates the supplied LSNs before processing the accompanying unlock request. Also, when BM steals a dirty page's buffer slot by writing the page to disk, it passes to LM the LSN of the page with the unlock request for the P lock.

LM supports a ***verify*** option with a lock request. The verify option is used by the data manager to ensure that it will read only the correct version of a needed page. That is, it helps the systems in dealing with the *detection* problem. The verify option returns the LSN associated with a second lock name provided in the lock request. With this option, when the data manager issues the *record* lock request, it gets the LSN for the corresponding *page* lock (see steps 1, 2, 3 and 4 in Figure 2). If the latter lock is not currently held by any system, then LM returns an LSN value of zero which implies that the latest version of the page is on disk (see steps 4 and 5 in Figure 4). A value of zero will be returned even if the page lock is currently held by one or more systems, including possibly an owner, in case the owner has not so far committed its updates (see steps 4 and 5 in Figure 5). In all cases, a returned value of zero means that the *latest committed* version of the page is definitely on disk.

The LSN value is looked up by LM after the record lock is granted to the requestor and, obviously, before LM returns to the requestor with the lock-granted response. This is important since, at the time the record lock request is made, another system might be still modifying that record and we need the LSN of the corresponding page *after* the modification of the other system is committed. By delaying looking up the LSN until the record lock becomes grantable to the requesting system, LM can guarantee that it would have come to know about the LSN of the latest committed version of the page. The latter is made possible since a transaction's logical locks are not released until it is ensured that the current LSNs of the pages modified by the transaction have been stored at LM. It was to accomplish this that the LSNs are updated *before* the unlock requests accompanying the LSNs are processed during transaction termination, as mentioned before.

The LSN requested as part of a verify request is returned as a Short-message that accompanies the lock grant response. We refer to the Short-message returned with an L lock request as an ***L-Short-message*** (see steps 3 and 4 in Figure 2). The L-short-message is used in fix_page processing (see step 5 in Figure 2). If the cached page's LSN is *less than* the LSN in the L-short-message, then BM needs to obtain a new version of the page. BM would use LM's Notify option to get a new version of the page from the owner, if there is one. BM would note in the buffer control block (***BCB***) which is associated with the buffer pool slot that is allocated for the page that a new version has been requested so that subsequent requestors who need a more recent version than the cached version are made to wait until the new version arrives. The *readers* whose requests can be satisfied with the older version can continue to use the cached version. If LM indicates that there is no owner for the page, then that would imply that the disk version of the page is the latest version. In that case, BM would read the page from disk.

In the page-transfer procedure, the owner's BM sends the page directly to the requesting system's BM (see step A in Figure 2). In addition, it sends, via LM (LLM on owner to GLM to LLM on requestor), a ***P-Short-message*** (see steps 10, 11, 12 and 13 in Figure 2). The P-Short-message contains the LSN of the shipped page. The owner attaches the message to the P lock related downgrading operation if it does such an operation as part of the page-transfer processing. Otherwise, it passes to LM just the P-Short-message as a response to the Notify message presented to it earlier by LM. A flag (***Owner_Exists***) is included in the Short-message which indicates whether or not an owner exists for the page (Yes or No, respectively). This flag is set by GLM. If there is no owner for the requested page, then GLM creates the P-Short-message and includes the LSN that it has for the page. When there is no owner, GLM will already have an entry for this page if at least one system is already holding an S lock on the page. In this case, the LSN value could be nonzero as a result of the page having been updated and then the U lock having been given up by the updater after the latter updated the LSN at GLM. Otherwise, the LSN will be zero due to the fact that the lock table entry would have been created as a result of the current lock request (see steps 3 and 4 in Figure 4). In the latter case, as mentioned before, the LSN field would have been initialized to the value of zero. If no owner exists and the requestor finds that its cached version, if any, is out of date, then it can obtain the latest version from disk (see step 6 in Figure 4).

Due to space constraints, we are unable to include some interesting details about our protocols here. The reader is referred to [MoNa91a] for those details.

3.2. *Medium Scheme Details*

3.2.1. Avoiding Using Stale Pages

In the Medium scheme, the page is transferred by the owner initiating a disk write (write is done only in the case of ownership transfer for a dirty page) and simultaneously shipping the page via comm-link, but without requiring a guarantee that the requestor will receive the shipped page. The page may arrive (1) before the P lock is granted to the requestor (the normal case), (2) after the P lock is granted and the page is already cached (because it was read from disk), (3) after the requesting BM read the page from disk, allowed it to be modified, wrote it to disk and purged it from the buffer pool, or (4) all actions of (3) followed by a fix_page request which causes a P lock to be requested. In cases (3) and (4), the received old version of the page is referred to as a ***stale page***. Below, we describe how we handle the abnormal cases (2), (3) and (4).

If the P lock is granted and the page is not cached, then the BCB for the page is marked to indicate that the page will be read from disk. Note that there is no timeout mechanism to request the page again from the owner. In the case of a U mode lock request, since the lock would have been granted to the requestor only after the disk write was complete, the requestor can read the page from disk. If the S mode lock had been requested, since the page would not have been written to disk by the owner, the requestor would have to first ensure that the page gets written to disk before it does the read from disk. This can be accomplished by the requestor becoming the owner by upgrading the lock to the U mode. Subsequently, if the originally shipped page were to arrive when there is already a cached version, then the shipped version will be discarded.

We avoid using a stale version of the page as follows: In case (3), when the page arrives a BCB for it would not exist and hence the received page will be discarded. In case (4), when the P lock is granted, the LSN included in the P-Short-message will be used to ensure that the cached page will not be used if it is not the current version.

3.2.2. Recovery from Failures

Since the Medium scheme writes the updated page to disk before another system is allowed to update the page, only one system's log records are needed to recover the page in case of a system failure. For a single system failure, the failing system would have retained U locks on the dirty pages that were in its buffer pool at the time of its failure. These pages are recovered using the failed system's log. Even if the retained locks are lost because of a catastrophic failure of LM, the log records of only one system will possibly be reapplied for a given page. In the ARIES-style recovery methods [MHLPS89, MoPi91, RoMo89], the LSN of a log record relating to a page is compared with the page_LSN and only if the latter is *less than* the former is that log record's update redone.

With reference to restart recovery after a system failure, the following points should also be noted:

- During the undo pass, the U lock must be reacquired on an affected page if it is not already held. This U lock acquisition will not cause deadlocks since, even during forward processing, U locks are not involved in deadlocks.

- In the case of recovery from a single system failure, a page involved in redo recovery (i.e., a page for which the U lock was held at the time of system failure) is transferable to any other system which needs it *after* the redo pass is completed. If the failed system is in its restart recovery, then LM would queue the incoming remote lock request until the failed system indicates that its page-transfer procedure is enabled. The transaction system would enable the page-transfer procedure at the end of the redo pass (i.e., after *repeating history* for all the missing updates [MHLPS89]).

- The current way of determining the restart recovery point (e.g., by the analysis pass of a single system recovery method like ARIES) would ensure that all the log records which might have to be reapplied will be encountered during the redo pass of restart recovery. This will be the case even if there is an SD-complex failure.

- In the case of an SD-complex failure (GLM and at least one LLM failed), which is expected to be very rare, no surviving system will be granted any *L* locks by the backup GLM which has taken over until all the failed systems recover completely (i.e., redo and undo passes are completed). *P* locks will not be granted until all the failed systems (1) complete their *redo* pass of recovery, and (2) on completion of the redo pass, they reacquire the needed *P* locks (U mode for dirty pages and S mode for nondirty pages) for pages currently in their buffer pools and register LSNs for the dirty pages in their buffer pools. At this point, GLM can reconstruct its lock table entries for all the *P* locks by gathering the information from all the LLMs. The *undo* pass of recovery for none of the recovering systems can be started until this happens. During this undo pass processing, P locks may need to be acquired as in normal processing. Once the undo pass is completed for all the recovering systems, GLM will be able to populate its lock table with all the *L* locks needed to protect all the uncommitted updates for the in-doubt (prepared state of two-phase commit [MoLo86]) transactions in the recovered systems and all the locks held by active transactions in the systems which did not fail. Note that the redo (undo) passes for the different recovering systems can be performed in parallel.

3.3. Fast Scheme Details

The key considerations in the Fast scheme are:

- A dirty page is transferred from one system to another without writing it to disk. If, as a result of this action, the page's ownership is also transferred, then if BM were to maintain a queue of dirty pages, call it ***Dirty_Q***, to support deferred and batched writes to disk, then the shipped page will be removed from it. The page can contain committed and/or uncommitted updates from multiple systems. When a dirty page is transferred to another system for updating by the latter (ownership transfer), it is the latter system's responsibility to write the page to disk. That is, the system which has the U lock on the page (the owner) is the one responsible for writing the page to disk. The owner is also responsible for recovering the page in case the owner fails before writing the page to disk. Of course, the ownership may be further transferred without the page being written to disk.

- Since, during the transfer of ownership of a page, BM removes the page from Dirty_Q of the transferring system, the LSN of the page's earliest unapplied (to the disk version of the page) log record is not factored in the computation of the restart recovery point which is checkpointed by the previous owner. For example, if there is only one dirty page in system S1 and its ownership is transferred to system S2, then the next checkpoint in S1 would result in the recording of the restart recovery point to be the start of this checkpoint as opposed to the RLSN of the dirty page (see [MHLPS89]). But, the RLSNs of all the pages which are owned by a recovering system must be factored in the calculation that determines the restart recovery redo point starting from which redo might have to be performed using the encountered log records.

- Since, during the transfer of ownership of a page, BM removes the page from Dirty_Q of the transferring system, there is a time period during which if the RLSN is lost (e.g., as a result of an SD complex failure), then the recovery of the page would be jeopardized. The following scenario shows that: A page was held in U mode in system S1 and system S2 requests it in U mode. S1 ships the page to S2, removes it from Dirty_Q, and releases the lock. Then, S1's next checkpoint starts which records the restart point which is later than the earliest unapplied log record for the dirty page which was shipped. Now the complex fails. The RLSN is lost and S1's and S2's checkpoint information will not position us to include the relevant log records of the dirty page. Hence, to correctly deal with this problem, we need to checkpoint GLM's lock table, including the RLSNs, on a periodic basis. This is a complex-wide checkpoint of the dirty page list. The lowest recorded RLSN is used to determine the restart redo recovery point during an SD-complex restart.

- A copy of the page is shipped via comm-link, as in the Medium scheme. A P-Short-message is sent to the requestor with the lock grant. The usage of a stale version of the page is avoided in the same way as in the Medium scheme. However, with the Fast scheme, if a stale version is cached or the page is not received by the time the lock is granted, the requestor **cannot** read the page from disk and use it as it is since, even during ownership transfer, the previous owner does not write the page to disk. The requestor first becomes the owner of the page, if it hasn't already become the owner as a result of getting the P lock. In doing so, it asks, via the lock request message, that the page be written to disk by the previous owner. If the previous owner has not failed, then it writes the page to disk and lets the requestor read the page from there. If the previous owner has failed, then the requestor would recover the page. Such a recovery involves reading the older version of the page from disk and applying the log records by scanning the *merged* log from the RLSN to the LSN when the owning system failed. An upper bound for the latter can be obtained by the new owner by noting, at the time it becomes the owner of the page, what the LSN would be if a new log record were to be written right then.

3.3.1. Assigning and Tracking Recover LSN

With the Fast scheme, since a dirty page's ownership is transferred without first writing it to disk, the page's Recover LSN (RLSN) has to be tracked at GLM to recover the page correctly in case the new owner fails before the page is written to disk. To accomplish this, BM assigns and tracks RLSN in the BCB when a U lock is requested for a page or whenever the page's state changes from nondirty to dirty. BM chooses as RLSN the LSN that would be associated with a log record if it were to be written now (essentially the end-of-log LSN). LLM and GLM initialize the RLSN field of a lock table entry to the maximum number that can be stored in that field (referred to as ***Hi-Value***). An RLSN value of Hi-Value for a page implies that no recovery is needed for that page. When a U lock is requested for a page, the P lock request would include the RLSN value assigned by BM. BM would request that the value be set *conditionally* by LM. LM would set its lock table entry's RLSN field to the supplied value if the current RLSN value at LM is Hi-value. This means that when a dirty page's ownership is being transferred from one system to another, without the page being written to disk, the RLSN value at LM is **not** modified. In any case, LM would return to BM the RLSN value that it has *after* it processes the lock request. When a U lock is released or downgraded to an S lock without the ownership of the page being transferred to another system (which can happen only after the current owner writes the page to disk), LM can set RLSN to Hi-Value.

To reduce the log range that would have to be processed for page recovery, BM in the owning system pushes the RLSN forward after writing the page to disk, but before it is dirtied again, by asking LM to set RLSN to Hi-value *unconditionally*. In this case, when the page becomes dirty again, BM would have to first update RLSN at LM *before* allowing the update to take place. Alternatively, the RLSN can be pushed after the page becomes dirty again to the higher value tracked in the BCB without the value being set at LM to Hi-value in between. Pushing the RLSN forward is not required by the algorithms presented here. This is an optimization to reduce the range of the log that would have to be scanned in case a failure happens and the page needs to be recovered.

3.3.2. Recovery from a Single System Failure

We first discuss the case when the page locks and their RLSNs are available from GLM at the time of the restart of a failed system. A page which needs redo recovery would have a U lock held and its RLSN will not be equal to Hi-value. The minimum of the RLSNs of all the pages for which U locks were retained by the recovering system is taken into account in computing the start point for the log scan of the redo pass [MHLPS89]. The merged log is scanned during the redo pass for redoing any updates which might be missing from the pages. A log record's update would be redone only if the U lock is held and the page's LSN is less than the LSN of the log record. The log is scanned up to ***End-LSN*** (the last log record written by the recovering system before it failed).

If a system requests a page lock which is retained in the U mode and the failed system has not begun its recovery processing, then GLM can grant the lock to the other system along with the message *you recover the page*. This option improves availability to the data. GLM can indicate, via the P-Short-message, to the requestor the need for recovering the page before it is used. With this enhancement, under the above conditions, even if only the S lock was requested, GLM will grant the U lock to make that page recovery possible.

The P-Short-message would have the following additional information:

- Indicator - *you recover the page.*
- System-ID of the system which retained the lock.

The requestor can query the system merging the local logs to determine the End-LSN of the failed system that held the U lock. As before, the RLSN kept at GLM would be returned when the lock is granted. The requestor can then read the page from disk, scan the merged log from RLSN to End-LSN of the failed system and recover the page. When such a recovery is done, the recovered page is marked dirty and placed in the Dirty_Q.

3.3.3. Recovery from an SD-Complex Failure

An SD-complex failure is characterized by the loss of all the locks at GLM and the inability to recreate at least some of them since one or more LLMs would have also failed. This means that for the U mode page locks, the LSNs and RLSNs would have also been lost. In such an event, the start point for the redo processing scan of the log cannot be determined as in the case of a single system failure. For this reason, periodically, a system takes a GLM checkpoint by first writing a *Begin_GLM_Checkpoint* log record and then requesting the IDs of all pages and associated RLSNs for pages with RLSNs not equal to Hi-value from GLM and writes them into an *End_GLM_Checkpoint* log record. The following is required to determine the restart redo point after an SD-complex failure:

- The end_GLM_checkpoint log record must be accessed and, based on its contents, the minimum of the RLSNs must be determined. If no page had an RLSN value smaller than Hi-value when the GLM lock table checkpoint was taken, then the above minimum is set to be the LSN of the begin_GLM_checkpoint log record.

- The merged log must then be processed starting from the LSN which is the minimum of the LSN of the begin_GLM_checkpoint log record and the LSN determined in the previous step. This redo processing is similar to the way it is done in ARIES [MHLPS89]. Until the redo scan reaches the *begin*_GLM_checkpoint log record, only log records relating to pages in the GLM checkpoint log record need to be processed. After that point, all log records would have to be processed until the end of the log is reached.

- The rest of the processing here is mostly the same as that for the Medium scheme. But, unlike in the case of the Medium scheme, here the redo pass for *all* the systems must be performed by *one* system by doing a single scan of the merged log. The easiest thing to do at the end of this pass is to write to disk all the dirty pages. If this is not desirable, then the RLSNs can be determined as redo is performed by associating with each page the LSN of that log record whose redo causes the page state to go from nondirty to dirty. The lock table entries for the dirty pages that are in the buffer pool at the end of the pass can be initialized from those RLSNs. Once the redo pass is completed by a single system on behalf of all the failed systems, the undo passes can be performed in parallel by the individual systems, as in the Medium scheme.

3.4. Super-Fast Scheme Details

In addition to the key points described for the Fast scheme, the following points also apply to the Super-Fast scheme:

- To enforce the WAL protocol, a page cannot be written to disk until all the log records written for that page by the different updating systems have been forced to stable storage. The tracking of these log records is done as follows: Associated with each dirty page, there are a certain number of slots. Each slot is used to track the LSN of the *latest* log record written by one of the systems which updated the page and whose updates have not yet been reflected in the disk version of the page. If a slot is available, then an updating system notes (or modifies its already existing entry) the LSN of the log record it just wrote for this page. Otherwise, the system would follow the Fast scheme when it is asked to transfer ownership of the page. That is, it would force the log before transferring the page. If a dirty page's ownership is transferred without some updating systems' logs having been forced to the requisite points, then the information in the slots is also passed on to the new owner along with the page.

- Before writing a dirty page to disk, its owner ensures that all the systems which updated the page have forced their respective logs up to the LSNs noted in the corresponding slots (see below for a method to do this check efficiently). If the log is not already known to have been forced up to the desired LSN in another system, then the owner sends a message to that system and requests it to do so. The interesting question that now arises is what happens if such a system had failed and hence it wouldn't respond. We may have, in the buffer pool, a dirty page which has some updates for which there are no log records on stable storage. Therefore, the page must be recovered by the owner by reading its old version from disk and redoing its updates using the merged log. Before doing such a page recovery, all surviving systems which had previously updated the page must be made to force their log records up to the requisite LSNs. This is required because it would be incorrect to miss a log record which is not yet forced and which relates to an update made by another system which may be committed later on.

- On a periodic basis, each system would register with GLM the highest LSN up to which that system's log has been forced to stable storage. This highest LSN is referred to as ***Hi-LSN***. GLM locates the entry corresponding to the system-ID and replaces its Hi-LSN value. Periodically, when it sends a message to a particular system, GLM would forward Hi-LSNs of all the other systems. Each system has a vector of the other systems' IDs and their respective Hi-LSNs. This vector's information is updated based on the messages from GLM.

4. Comparisons with Existing Work

We know of no existing work where

- fine-granularity (e.g., record) locking is supported with as much flexibility (e.g., semantically-rich modes of locking) as our schemes do,
- the combination of *no-force* and *steal* buffer management policies are supported,
- the extent of data availability under failure conditions is as high as with our schemes,
- recovery issues are addressed for the different schemes in as much depth as we have done,
- partial rollbacks are supported,
- the Super-Fast scheme is described, and
- pages are shipped using datagrams.

4.1. Rahm's Scheme

The only paper in the SD area that discusses recovery to a reasonable extent in the context of concurrency and coherency control is [Rahm89]. In the following, we compare our work with Rahm's protocols in that paper.

Rahm's coherency control and recovery protocols are designed for supporting a very particular form of concurrency control protocol called primary copy locking (***PCL***). With PCL, the data base is divided into *logical* partitions and each system is assigned the synchronization responsibility (or primary copy authority (***PCA***)) for one partition. This PCA/PCL method increases the burden on the data base administrator who now has to decide how to partition the data base into logical partitions. This is very similar to the data base design problem in the partitioned (shared nothing) architecture [PMCLS90, Shoe86].

When a system fails, access to all the data for which that system was the PCA is denied. In our case, only the data for which the failed system had retained locks would be unavailable until recovery is completed for the failed system. Almost always, the latter data would be a much smaller portion of the data base than would be the case with the former data.

The detection technique that we employ is similar to the technique used by Rahm. The LSN for a locked page is maintained only at the PCA system for that page. Rahm supports essentially only page locking, although he hints at how the protocols might be extended for some *very restricted forms of record locking*. This means that index concurrency control methods like ARIES/KVL [Moha90] and ARIES/IM [MoLe89] cannot be supported in the SD environment by his protocols.

Rahm's protocols support only physical, not logical or operation, logging [MHLPS89]. Further, they do not support partial rollbacks. No logging is done of updates performed during rollbacks of transactions. That is, compensation log records (CLRs) are not written. As a result, media recovery requires a two-pass algorithm to make sure that no log records written by uncommitted transactions are redone. Also, it requires that image (archive) copies be taken with locking being done on the copied data to ensure that no uncommitted data is copied. With this approach, image copy will take a longer time to finish, will be more expensive in terms of CPU overhead and there will be more interferences between the image copy operation and (regular) transactions. We support the cheaper fuzzy image copy method of [MHLPS89].

Rahm supports only the *no-steal* buffer management policy (i.e., pages with uncommitted data cannot be written to disk). As argued in [MHLPS89], this is an inflexible and expensive policy, especially when fine-granularity locking is being done. This will be the case even if large amounts of real memory are available. Also, too much bookkeeping is needed to enforce the policy.

Like us, Rahm also allows a modified page to be shipped over a comm-link. Before a page is written to disk, it may be modified by many systems. However, in Rahm's scheme, only the PCA node for the page has the authority to write the page to disk. Consequently, at commit time, an updating system has to send pages updated by the transaction to the pages' respective PCA systems. This leads to wasted buffer storage since the updated version of the page is present in at least two buffer pools when the updating system is different from the PCA system for the page. Further, when a page is modified in any system other than its own PCA system, double logging is required: logging is done in the modifying system as well as in the PCA system for that page. For this reason, the updating system sends the log records written by a transaction to the PCA system for the affected data at commit time. These commit time actions can significantly increase the communication traffic, in terms of volume of data, on the intersystem communication network. They also increase the complexity of the software since log records have to be separated by the PCA nodes of the updated data. In our schemes, any system which is currently the owner (updater) of the page has the authority to write the page to disk.

Rahm does not describe shipping the page with the Super-Fast scheme. That is, not forcing the log before shipping the page. We addressed the page-recovery issues when the Super-Fast scheme is used.

Rahm does not mention his assumptions about the characteristic of the communication protocol (guaranteed delivery or datagram) used for shipping the page. Rahm's protocols ship the page with the lock grant message if the page is present in the buffer pool of the PCA system and that version of the page is not already present in the buffer pool of the locking system.

4.2. Dias et al.'s Scheme

An approach to concurrency and coherency control is presented by Dias et al. in [DIRY89]. There are some major differences between our approach and theirs. They do not deal with most of the failure and recovery implications of their design. They support only page-level granularity of locking by transactions between systems for reads and updates. Their GLM treats transactions, rather than LLMs, as the owners of locks. This means that the number of locks acquired, and the message and processing overhead will be higher if multiple transactions within a system access the same page. Further, separate page-level (global) locks are used for coherency control and these locks are acquired by BM. The message overhead is reduced for these extra locks by piggybacking them on transaction lock requests. Due to space constraints, we did not describe here our optimized protocols for page locking by transactions. Those protocols [MoNa91b] are even more efficient than the protocols of Dias et al.

Dias et al. require that the system always write pages modified by a transaction to disk or an intermediate shared storage before commit. Hence, a requestor always reads the page from disk or the intermediate storage. For their Check-on-Access scheme, they do not track the LSN of the page at GLM. Instead, when a page is updated and the updating transaction releases its locks at GLM, their scheme invalidates the page at GLM for the other systems by releasing the other systems' BM locks, if any, for that page.

4.3. DEC's VAXcluster Scheme

As mentioned before, DEC's VAX DBMS and VAX Rdb/VMS [KrLS86, ReSW89] support the SD environment in a VAXcluster with a detection scheme very similar to ours. They also use version numbers. Like IMS, those systems also force updated pages to disk at commit time, use physical logging and use the simple scheme for page transfer between systems.

Unlike our assumption that each system has its local log, in the VAXcluster, all the sharing systems use a single global log. Having a single log for direct use by all the systems becomes expensive since *every write* to the global log requires acquiring a *global lock* to serialize the space allocation in the log file. In the VAXcluster, even single system failures are very disruptive. This is because, locking activities across all the sharing systems are suspended until the failed system's recovery is completed by one of the surviving systems. Because of the force policy being used, recovery involves only rolling back uncommitted transactions. This could take a very long time if some long update transactions which were executing on the failed system have to be rolled back.

5. Summary

A transaction system, such as DB2, which does not write an updated page to disk at transaction commit has the current version of a page in its buffer pool. In an SD environment, each sharing transaction system instance has its own buffer pool. Therefore, when a system requests a page whose current version is cached in another system (referred to as the owner), the owner must provide the page to the requestor. We proposed *efficient* schemes by which the owner provides a copy of the current version of the page to the requestor without disk I/Os. These schemes improve transaction response time and concurrency. Techniques which enhance availability of data in the presence of failures were also described.

We described how the owner ships the page, the requestor ensures that it always uses the current version of the page, and the system recovers the page in case of failures. The methods presented here do not rely on any timeout mechanisms. We did not discuss media recovery specifically since there is nothing special that needs to be done for it, except for the use of the merged log. The latter is very similar to the way recovery from an SD-complex failure is handled in the Fast and Super-Fast schemes. The algorithm for fuzzy image (dump) copy proposed in [MHLPS89] can be easily adapted for use in the SD environment.

In the following, we summarize what we consider to be the novel features of our schemes.

1. Support for the *no-force* and *steal* buffer management policies, and fine-granularity locking and partial rollbacks in a flexible fashion, thereby accommodating even the high-concurrency index locking protocols like ARIES/IM [MoLe89] and ARIES/KVL [Moha90]. Also, support for nested transactions by using the ARIES/NT logging and recovery method [RoMo89].

2. The concept of keeping the recovery starting point, referred to as the Recover LSN (RLSN) of the page, at the lock manager and the use of a merged log to allow transfer of committed or uncommitted data from one system to another without having to write the modified page to disk first.

3. The idea of communicating the LSN of the latest update that each system made to a page so that a page can be sent through and modified by a series of systems without the modifying systems having to write the page or the log to disk before they pass on the page to a succeeding system in the series.

4. The use of Recover LSN to enhance availability of data by allowing the recovery of a page by one system when another system has failed while owning the update privilege on the page.

5. The use of a Short-message as grant data with a lock to ensure that the requestor never uses a stale version of the page which was transferred via a datagram.

6. The idea of checkpointing the GLM lock table to recover in the case of an SD-complex failure (i.e., when locks are lost).

7. The idea of transferring the page via datagram concurrently with the disk write which reduces the programming system complexity by not requiring a merged log for restart recovery.

The choice of a particular scheme for page transfer can be based on the following criteria:

- *Intersystem Contention for the Page* For low contention data, Medium scheme should suffice; for high contention data, Fast and Super-Fast schemes should be considered.

- *Requirement of a Merged Log* For Fast and Super-Fast schemes, restart recovery requires a merged log. For the Medium scheme, restart recovery does not require a merged log. Of course, with all schemes, for media recovery, a merged log is required.

- *Record Locking Versus Page Locking* With record locking, the Fast and the Super-Fast schemes should be considered since the intent is to improve concurrency within a page. With page locking, the Super-Fast scheme does not apply [MoNa91b].

- *Complexity of Programming* The complexity of programming for the different schemes in increasing order is: Simple, Medium, Fast and Super-Fast.

Our schemes can be easily incorporated in SD systems which are currently using schemes which are less efficient and which also have poor availability characteristics. Our schemes should also apply to distributed virtual memory and distributed, recoverable file systems in the SD environment and to the currently popular client-server object-oriented DBMS environments where the clients cache data.

Acknowledgements We would like to convey our thanks to Pat Selinger for her editorial comments.

6. References

BACCD90 Boral, H., Alexander, W., Clay, L., Copeland, G., Danforth, S., Franklin, M., Hart, B., Smith, M., Valduriez, P. *Prototyping Bubba, a Highly Parallel*

Database System, **IEEE Transactions on Knowledge and Data Engineering**, Vol. 2, No. 1, March 1990.

Bhid88 Bhide, A. *An Analysis of Three Transaction Processing Architectures*, **Proc. 14th International Conference on Very Large Data Bases**, Los Angeles, August 1988.

CFLS91 Carey, M., Franklin, M., Livny, M., Shekita, E. *Data Caching Tradeoffs in Client-Server DBMS Architectures*, **Proc. ACM SIGMOD International Conference on Management of Data**, Denver, May 1991.

DGSBH90 DeWitt, D., Ghandeharizadeh, S., Schneider, D., Bricker, A., Hsiao, H.-I, Rasmussen, R. *The Gamma Database Machine Project*, **IEEE Transactions on Knowledge and Data Engineering**, Vol. 2, No. 1, March 1990.

DIRY89 Dias, D., Iyer, B., Robinson, J., Yu, P. *Integrated Concurrency-Coherency Controls for Multisystem Data Sharing*, **IEEE Transactions on Software Engineering**, Vol. 15, No. 4, April 1989.

DMFV90 DeWitt, D., Maier, D., Futtersack, P., Velez, F. *A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems*, **Proc. 16th International Conference on Very Large Data Bases**, Brisbane, August 1990.

HaRe83 Haerder, T., Reuter, A. *Principles of Transaction Oriented Database Recovery - A Taxonomy*, **Computing Surveys**, Vol. 15, No. 4, December 1983.

KrLS86 Kronenberg, N., Levy, H., Strecker, W. *VAXclusters: A Closely-Coupled Distributed System*, **ACM Transactions on Computer Systems**, Vol. 4, No. 2, May 1986.

Li88 Li, K. *IVY: A Shared Virtual Memory System for Parallel Computing*, **Proc. 1988 International Conference on Parallel Processing**, August 1988.

MHLPS89 Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P. *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*, To Appear in **ACM Transactions on Database Systems**. Also available as **IBM Research Report RJ6649**, IBM Almaden Research Center, January 1989; Revised November 1990.

Moha90 Mohan, C. *ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes*, **Proc. 16th International Conference on Very Large Data Bases**, Brisbane, August 1990.

MoLe89 Mohan, C., Levine, F. *ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging*, **IBM Research Report RJ6846**, IBM Almaden Research Center, August 1989.

MoLO86 Mohan, C., Lindsay, B., Obermarck, R. *Transaction Management in the R* Distributed Data Base Management System*, **ACM Transactions on Database Systems**, Vol. 11, No. 4, December 1986. Also available as **IBM Research Report RJ5037**, IBM Almaden Research Center, February 1986.

MoNa91a Mohan, C., Narang, I. *Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment*, **IBM Research Report RJ8017**, IBM Almaden Research Center, March 1991.

MoNa91b Mohan, C., Narang, I. *ARIES/SD: A Transaction Recovery and Concurrency Control Method for the Shared Disks Environment*, **IBM Research Report**, IBM Almaden Research Center, Forthcoming, 1991.

MoNP90 Mohan, C., Narang, I., Palmer, J. *A Case Study of Problems in Migrating to Distributed Computing: Page Recovery Using Multiple Logs in the Shared Disks Environment*, **IBM Research Report RJ7343**, IBM Almaden Research Center, March 1990.

MoNS90 Mohan, C., Narang, I., Silen, S. *Solutions to Hot Spot Problems in a Shared Disks Transaction Environment*, **IBM Research Report**, IBM Almaden Research Center, December 1990.

MoPi91 Mohan, C., Pirahesh, H. *ARIES-RRH: Restricted Repeating of History in the ARIES Transaction Recovery Method*, **Proc. 7th International Conference on Data Engineering**, Kobe, April 1991. Also available as **IBM Research Report RJ7342**, IBM Almaden Research Center, February 1990.

Nech88 Neches, P. *The Ynet: An Interconnect Structure for a Highly Concurrent Data Base Computer System*, **Proc. 2nd Symposium on the Frontiers of Massively Parallel Computation**, Fairfax, October 1988.

PMCLS90 Pirahesh, H., Mohan, C., Cheng, J., Liu, T.S., Selinger, P. *Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches*, **Proc. 2nd International Symposium on Databases in Parallel and Distributed Systems**, Dublin, July 1990. An expanded version of this paper is available as **IBM Research Report RJ 7724**, IBM Almaden Research Center, October 1990.

Rahm86 Rahm, E. *Primary Copy Synchronization for DB-Sharing*, **Information Systems**, Vol. 11, No. 4, 1986.

Rahm89 Rahm, E. *Recovery Concepts for Data Sharing Systems*, **Technical Report 14/89**, University of Kaiserslautern, October 1989.

ReSW89 Rengarajan, T.K., Spiro, P., Wright, W. *High Availability Mechanisms of VAX DBMS Software*, **Digital Technical Journal**, No. 8, February 1989.

RoMo89 Rothermel, K., Mohan, C. *ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions*, **Proc. 15th International Conference on Very Large Data Bases**, Amsterdam, August 1989. A longer version of this paper is available as **IBM Research Report RJ6650**, IBM Almaden Research Center, January 1989.

Scru87 Scrutchin, T. *TPF: Performance, Capacity, Availability*, **Proc. IEEE Compcon Spring '87**, San Francisco, February 1987.

Shoe86 Shoens, K. *Data Sharing vs. Partitioning for Capacity and Availability*, **Database Engineering**, Vol. 9, No. 1, March 1986.

SNOP85 Shoens, K., Narang, I., Obermarck, R., Palmer, J., Silen, S., Traiger, I., Treiber, K. *Amoeba Project*, **Proc. IEEE Compcon Spring '85**, San Francisco, February 1985.

Ston86 Stonebraker, M. *The Case for Shared Nothing*, **IEEE Database Engineering**, Vol. 9, No. 1, 1986.

StUW82 Strickland, J., Uhrowczik, P., Watts, V. *IMS/VS: An Evolving System*, **IBM Systems Journal**, Vol. 21, No. 4, 1982.

Tand87 The Tandem Database Group *NonStop SQL: A Distributed, High-Performance, High-Availability Implementation of SQL*, **Proc. 2nd International Workshop on High Performance Transaction Systems**, Asilomar, September 1987.

WiNe90 Wilkinson, K., Neimat, M.-A. *Maintaining Consistency of Client-Cached Data*, **Proc. 16th International Conference on Very Large Data Bases**, Brisbane, August 1990.

WuFu89 Wu, K.-L., Fuchs, W.K. *Recoverable Distributed Shared Virtual Memory: Memory Coherence and Storage Structures*, **Proc. 19th International Symposium on Fault-Tolerant Computing**, Chicago, June 1989.