TECHNISCHE
UNIVERSITÄT
MÜNCHEN

## INSTITUT FÜR INFORMATIK

Sonderforschungsbereich 342:
Methoden und Werkzeuge für die Nutzung
paralleler Rechnerarchitekturen

# Logging and Crash Recovery in Shared-Disk Parallel Database Systems

Giannis Bozas, Susanne Kober

TUM-I9812
SFB-Bericht Nr. 342/06/98 A
Juni 98

TUM–INFO–06-I9812-70/1.–FI

# Logging and Crash Recovery in Shared-Disk Parallel Database Systems[1]

**Giannis Bozas[*], Susanne Kober**

[*]Institut für Informatik, Technische Universität München
Arcisstr. 21, D-80333 München, Germany
e-mail: bozas@informatik.tu-muenchen.de
Susanne.Kober@sdm.de

## Abstract

In this work, we develop concepts for implementing a logging and recovery component to deal with node crashes in a shared-disk system environment. This is done, using several previously published strategies and adapting those algorithms to fit our special system needs. Our environment is characterized by the following issues: the global lock manager is statically distributed among the system's nodes and employs a hierarchical synchronization protocol for efficient transaction processing. A local lock manager on each node administers local lock requests. Committed modifications to a data page are administrated by the respective *page owner*, which may - in order to allow maximum adaptability - dynamically migrate across the system. These distribution aspects evoke special problems for the logging component of the system, which are discussed in this paper in detail. We present two logging strategies and discuss their tradeoffs with respect to system performance, recovery costs and reliability. At last, we show that our recovery component is able to reconstruct any runtime system information as well as corrupted pages after a single or multiple node failure, simply by collecting intact information residing on surviving nodes and reading the local log data from permanent storage.

## 1 Introduction

In this paper, we focus on aspects of failure transparency in parallel database systems and especially in shared-disk systems, the architecture of choice in our work. Each database system, be it a centralized or a distributed architecture, has to fulfil the ACID-paradigm [3] when executing a transaction. Atomicity and durability are guaranteed by the logging and recovery subsystem, which means that either all or none of the performed data modifications of a transaction are reflected in the system, and that all committed updates survive arbitrary failures (node crashes in this context). Because there are strong interdependencies between the different components of a parallel shared-disk database system, designing a simple and efficient logging and recovery com-

1. This research is directed by Prof. R. Bayer Ph.D. and Prof. Dr. B. Mitschang and is part of the more extensive project "Tools and Methods for Using Parallel Computer Architectures" running at the department of computer science of the Technische Universität München since 1990 (sponsored by DFG contract SFB 342).

ponent is a very difficult job.

This paper describes and evaluates concepts for implementing a logging and recovery component for the **MIDAS**-system (**M**un**I**ch parallel **DA**tabase **S**ystem), which is developed at the Technische Universität München. Though we strongly rely on existing concepts for the development of our solutions, we outline necessary adaptations and additional features to optimize the coordination between our logging and recovery component and existing synchronization and coherency control protocols.

### 1.1 Overview

The remainder of this paper is structured as follows. Section 2 provides information of the system in use and presents a brief introduction to architectural issues and the synchronization and coherency control algorithms which underlie our solutions.

In Section 3 we outline some basic ideas about logging and recovery before describing two different protocols for the execution of transaction COMMIT. Special attention is paid to system performance issues and communication costs, as both concepts are compared to each other and their tradeoffs are discussed. The rest of the section deals with adaptations that have to be made when using a static instead of dynamic distribution of page ownership in the system.

Section 4 presents methods for the execution of crash recovery after a single or multiple node failure, based on the previously introduced logging strategies. We discuss how system information lost during the crash can be consistently reconstructed using inter-node communication and how committed data modifications are redone, in case they are not already present in the physical database. We focus on aspects of communication costs and dependencies between the different phases of recovery execution. Once more, versions of the algorithms to be used with static page ownership are provided.

In Section 5 we discuss the differences between our solutions and work previously published by several authors. Moreover, we outline why existing work had to be modified to fit our special environment and which are the deficiencies of some of the previous solutions. The paper concludes with a summary in Section 6.

## 2 Description of the System Environment

In this section we present an overview of the general hardware and software architecture of the anticipated system environment. The intention is, on the one hand to explain the background necessary to understand the logging and recovery algorithms, and on the other hand some design decisions made during the development of our concepts. A more detailed description of those aspects can be found in [5] and in previous work [6][20].

### 2.1 Shared-Disk Architecture

In this class of systems an arbitrary number of identical nodes (with respect to their HW- and SW-configuration) is loosely coupled with each other via a communication link, which is not further specified in this context. While the nodes share access to a common database which may be spread across several disks, each of the nodes disposes exclusively of its main memory (database cache). A transaction executes on a single node in the system which avoids the need for distributed transaction execution as known from shared nothing systems. For this purpose, a node must read a referenced data object from disk into its private cache, which is referred to as *vertical data replication*. Since a node is not forced to remove a cached object from its main memory at EOT, this means that (possibly) different versions of the object may exist at the same time on different nodes. This is known as *horizontal data replication*, and additionally introduces the need for consistent administration of all cached copies (cache coherence).

We summarize the features which differentiate shared-disk systems from traditional centralized database systems:

- Many nodes share access to a common database, which may reside on more than one disk.
- The nodes collaborate to enable consistent transaction processing in the sense of consistency and isolation.
- Horizontal data replication introduces additional effort, in order to keep the system in a logically consistent state.

The following paragraphs explain the way, the anticipated system deals with those aspects of collaborative data processing.

## 2.2 Lock Management

To prevent the bottleneck of a centralized global lock manager each node is responsible for the synchronization of a fixed set of objects and processes the corresponding global lock requests. Thereby, a static distribution of the global lock manager and a logical fragmentation of the shared database is introduced, in order to guarantee maximum performance and parallelism during the synchronization of concurrent transactions. A local lock manager (LLM) supports the work of the global lock manager (GLM) on each node by administrating all local transactions' lock requests. Thus any lock information is duplicated (in a local and a global lock table) in the system, which has a big impact on the recovery mechanism introduced in Section 4.

Our lock manager uses a hierarchical synchronization protocol based on an RX-strategy [20]. The protocol operates on an object hierarchy consisting of subsegments, pages and subpages. Each subpage is a fixed size part of a page unit. Fine granularity locking is supported to a sufficient extent when the conflict ratio is high, while the synchronization effort is reduced by employing coarse granularity locking when the conflict ratio is low. The elaborate protocol adapts to the current conflict situation by employing the mechanism of de-escalation and escalation of locks during transaction execution. Thus, the method is an efficient trade-off between maximum parallelism and minimum administration overhead.

## 2.3 Buffer Management

A node's main memory buffer is usually by factors of hundreds or thousands smaller than the physical database. Therefore, an efficient strategy is necessary for the administration of the restricted number of frames in the system's cache, where data pages have to be loaded from disk before performing any data manipulation on them. The method employed by the buffer management component when choosing frames to be replaced or used in the cache, influences not only the system's runtime performance, but also the necessary overhead for the logging and recovery component. An elaborate buffer strategy, which minimizes the need for expensive disk I/O, has to be evaluated w.r.t its logging activity, which is necessary to ensure atomicity and durability of data modifications in the system. We refer to [18] for a suitable classification of strategies in use.

In our system, a transaction does not write through its modifications to the physical database at EOT. This is called a *NOFORCE* strategy and is very efficient with respect to minimization of I/O overhead. In this case, many sequential modifications by different transactions may be accumulated in a cached data page before it has to be updated physically on permanent storage. This is especially interesting for frequently modified pages (hot pages) which otherwise would cause a physical write operation any time they are modified. Nevertheless, NOFORCE introduces the necessity of logging committed updates on permanent storage (*REDO-logging*), in order to make them survive system crashes (loss of main memory contents), as is defined by the durability property.

To simplify the recovery algorithm in the sense of guaranteeing transactional atomicity we use a *NOSTEAL* strategy for selecting disposable frames in main memory. This means that frames containing dirty (uncommitted) data modifications cannot be reused until the active transaction commits or aborts. Thus, no dirty data are written to disk and we avoid the need for performing *UNDO-operations* during crash recovery.

## 2.4 Cache Coherency

A problem to be dealt with especially in shared-disk systems is horizontal data replication. Different versions of the same data object have to be efficiently administrated. To prevent interferences between several nodes, which may update different subpages of the same page simultaneously (which is supported by our synchronization protocol), we define one node at a time to become owner of the page, whenever at least one page copy is cached in the system. A page which is not cached at any node is not assigned an owner. It is the page owner's responsibility to administrate all existing page copies and to keep the disk version of the page in a consistent state.

With respect to page ownership, we introduce another logical fragmentation of the database. In order to guarantee maximum node autonomy and to minimize communication overhead, we allow dynamic assignment of a page to an owner across the system's nodes according to their data access patterns. Whenever an owner must remove its local page copy from its cache, it has to check if any other page copies exist in the system. For this purpose, each page owner administrates a copyset in its page table [6], which contains an identifier for any node that possibly still caches a page copy. When the copyset is empty, there are no page copies left in the system. Otherwise, the page owner requests one of the remote copy holders to become the new page owner. In any case, the actual page version is written to the physical database before the page owner is allowed to remove its page copy from its buffer. [5] and [6] provide a more detailed description of the protocols for the page ownership migration procedure.

An updating node, which is not the owner of a modified (sub-)page, contacts the owner at transaction COMMIT time and submits the new version. This means that at any time the page owner administrates the latest page version. Any other node which references the (sub-)page in the future has to be supplied with the actual version, which results in an invalidation of old (sub-)page copies residing in other caches. To keep the necessary inter-node communication overhead at a low level, we use an on-request scheme [17] for the detection of invalidated page copies at remote nodes.

For this purpose, we introduce vectors of version numbers for each page, consisting of entries for each subpage. At system start the number is initialized at 0 for every subpage and is incremented during transaction processing for every committed modification of the respective subpage. By keeping these vectors physically persistent in the page header, a strictly monotonous global order can be defined on the version vectors of each page. Every vector value identifies exactly one state of the page, even while the page exists only on disk and there are no copies cached anywhere in the system. Additionally, a copy of the vector is administrated by the GLM of the page and each node that caches a page copy stores the version vector in its page table.

Any time a node references an unlocked data object, it sends the version number of the cached copy (-1, if no local copy exists) together with the lock request to the GLM. The GLM decides on the validity of the requested object and, in case of an invalidation of the remote copy, instructs the current page owner to supply the requestor with the valid object version. Since at least some of the necessary communication may be embedded in synchronization messages, the advantages of this method over broadcast- or multicast invalidation schemes become apparent.

## 3 Two Different Logging Concepts

Having basically described the system environment, especially the underlying protocols for synchronization and cache coherency, we now outline our suggestions for the realization of a logging subsystem. After introducing fundamental ideas, we present two alternative protocols for the execution of transaction COMMIT. Finally, we discuss the advantages and disadvantages of each solution.

### 3.1 Preliminaries for the Logging Subsystem

The logging subsystem must assure that data modifications are consistently and persistently represented in the database even when failures (crashes) occur during transaction processing. We have already pointed out that the organization of the buffer management has a great influence on the algorithms of the logging subsystem.

**Implications of the Buffer Management**
Transactions, which have been active on a node at the time this node crashed, are aborted and none of their modifications is allowed to reside in the physical database after recovery. Since we employ a NOSTEAL strategy this holds, and no UNDO steps have to be taken during crash recovery, as the physical database cannot be dirtied by uncommitted data. On the other hand, NOFORCE requires the REDO of committed data modifications, which are not yet represented in the physical database. Therefore, any update operation performed by a successfully committed transaction has to be made durable before the modification is made visible to other transactions by releasing the locks on the data objects. This is an important concept in the sense of transactional consistency and durability and is referred to as WAL-Principle (Write-Ahead-Log).

**Log Organization**
During transaction processing, log data are written to a dedicated area on stable storage, usually to an additional disk. The log is organized as a sequential array of formatted data, so-called log

records. All transactions compete for access to the end of the log file. To prevent the global log file to become a system bottleneck, we distribute the conceptual global log into local log files, one on each processing node. Each of these private logs is exclusively written by the local node, but may be read by any other node during recovery processing.

**Log Record Types**
We already mentioned that log data is organized in log records, which is a suitable means for efficient administration. Because log data must cover various different needs, we introduce several kinds of formatted log records: the log manager writes UNDO- and REDO-records containing the physical before-image (BFIM) respectively after-image (AFIM) of a modified subpage. Our decision for physical state logging is motivated by the simplicity of the recovery actions to be taken and the low cost of log space when employing subpage-images. Note that we need UNDO information only for the execution of transactional ROLLBACKS, but not for recovery purposes (because of NOSTEAL). The records additionally contain the number of the executing transaction (transaction_id) and an identifier of the corresponding subpage (subpage_id).

To trace the progress of transactions for which log records exist, the log manager writes COMMIT- and ABORT-records containing the transaction_id at the successful respectively unsuccessful end of a transaction. Another record type, the CHCKPT-record, is explained below in our discussion of checkpointing.

**Log Sequence Numbers**
We need a mechanism which enables us to relate each log record to exactly one version of the respective subpage. This is achieved by the use of monotonically increasing and unique log sequence numbers (LSNs), which are explained in detail in the next subsection. Each LSN should be understood, both, as a unique record identifier and as a conceptual address on the log. Thus, we define a Start_LSN for each page, which is the LSN of the REDO-record representing the first update operation on the page not yet represented in the physical database and which is kept in the page's page table entry. Additionally, the header of each page contains the LSN of the REDO-record representing the last modification applied to one of the page's subpages. This value is updated on the owner's cached page, whenever a modification is logged at the page owner. Whenever the owner writes the page to disk this value is then reflected into the physical page. This strategy allows us to determine during recovery processing, whether an update operation found on the log has to be redone or whether it is already represented in the physical database. Note that it is not necessary to administrate the LSN for each subpage, since only pages are exchanged between external and main memory.

**Checkpointing**
Additionally, the system takes occasionally fuzzy checkpoints, writing a system snapshot to stable storage in order to simplify future recovery actions. Each CHCKPT-record contains information on the set of active transactions, pages containing data not yet propagated to the physical database (REDO candidates) and the log address of the first record to be processed at REDO recovery (Start_REDO_LSN, which is the minimum of all REDO candidates' Start_LSNs).

## 3.2 Distributed LSN-Scheme

In our architecture, each transaction processing node writes log information for the pages it owns to its private log. Since page ownership migrates dynamically across the system the physically distributed global log is not divided into disjunct partitions. Instead, many log files may contain records referring to the same subpage, according to the time relation of their page ownership. Because each log record has to uniquely identify a page version in the system, our scheme must carefully assign LSNs to the respective log records. We rely on ideas of [14] for the presentation of a suitable solution to this problem.

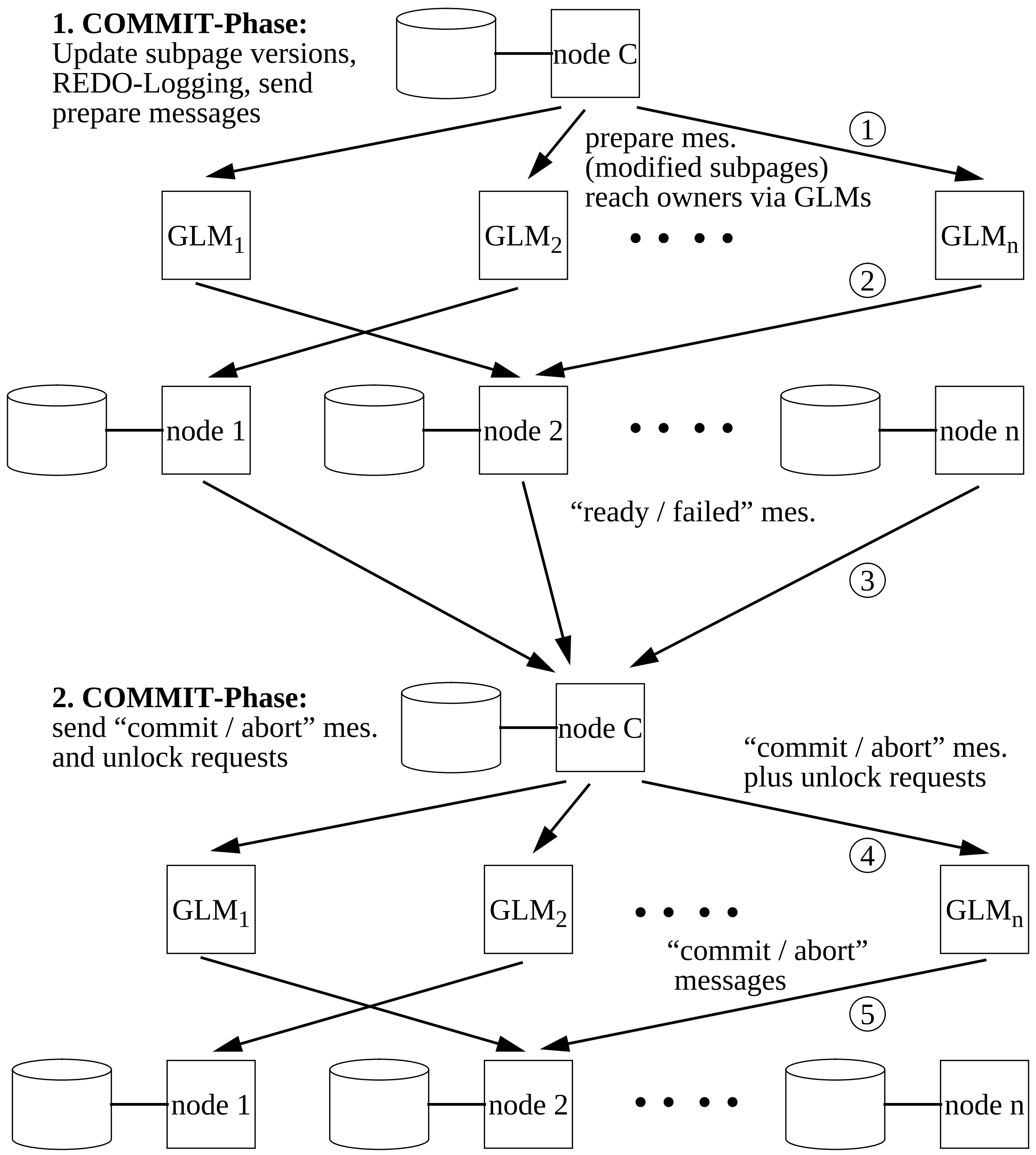
During logging, a node finds the maximum value between the header LSN of the corresponding page and the highest LSN on the local log file, increments it and assigns the resulting value to the new log record. Thus we guarantee, that both, LSNs inside each log file as well as the LSNs for the distributed log records of a single page increase monotonically. Unfortunately, we have to accept that the LSNs provided by this scheme can no longer be used directly as log addresses, but must be explicitly converted into physical addresses for this purpose.

## 3.3 Two-Phase COMMIT Protocol: Single Logging Solution

In the following paragraphs, we will concentrate on the realization of REDO logging. Nevertheless, we have to provide a strategy for UNDO logging. Because of the use of an RX-synchronization protocol (instead of RAX [1]) we have to perform ROLLBACK on the owner's page copy in case of ABORT, in order to reconstruct the previous consistent page version, which is not necessarily in the physical database. To ensure the WAL principle, the page owner (PO) must log the BFIM of the local page copy before admitting uncommitted data modifications during the execution of the following COMMIT-protocol. The algorithms for the execution of transactional ABORT are identical to those of centralized database systems.

We are now ready to present a first concept for a two-phase COMMIT protocol, which is illustrated in Figure 1. Node C, on which the committing transaction executes, transfers its modified subpages to their assigned page owners via the GLM at transaction COMMIT (steps ① and ②). Subpages which are assigned to C itself are treated identically, only without the described communication steps. The page owners overwrite their local page copies with the new version (after having performed the necessary UNDO-logging), adapt the version numbers of the respective pages and log REDO information to the local log file. Any modified page is to denote DIRTY, since it contains still uncommitted updates at this time. A page owner answers "ready" or "failed" to node C, depending on whether it performed the previous steps correctly or not (step ③). This marks the end of the first COMMIT phase.

C decides at the beginning of the second phase whether a successful COMMIT is possible or an ABORT of the transaction is necessary. In the latter case, at least one of the remote nodes has answered "failed" or has been timed out. The result is logged locally at C which means a COMMIT- respectively ABORT-record is written to C's private log file. Any other node may ask C about the transaction's state from now on. C informs the remote page owners ("commit"- or "abort"-message) and thus makes them complete their logs by a COMMIT- or ABORT-record as well. Note that we need to perform UNDO on the previously updated page copies in case of a failure of the first phase. In case the result at node C is COMMIT, the lock managers may release the locks after receiving the messages at step ④, before forwarding the "commit" to the page owners (step ⑤). Otherwise, the GLMs must forward the "abort" result to the page owners, which must first undo the modifications on the pages, thereby reading UNDO-records from the local log, before contacting the GLMs again (in another message passing round not shown in Figure 1), enabling the latter to release the page locks. In this case, it is impossible to unlock earlier, because



1. COMMIT-Phase:
Update subpage versions, REDO-Logging, send prepare messages
node C
prepare mes. (modified subpages) reach owners via GLMs
1
GLM1
GLM2
GLMn
2
node 1
node 2
node n
"ready / failed" mes.
3
2. COMMIT-Phase:
send "commit / abort" mes. and unlock requests
node C
"commit / abort" mes. plus unlock requests
4
GLM1
GLM2
GLMn
"commit / abort" messages
5
node 1
node 2
node n

**FIGURE 1. Two-Phase COMMIT: Single Logging Protocol**

otherwise dirty data would become visible to other transactions. The scheme requires close coordination between the lock managers and the logging subsystems of the participant nodes, which is a substantial disadvantage for system performance.

At the end of the second phase, node C does not get any acknowledgment from the remote page owners, so that any node which fails after successful completion of only the first phase has to contact C to learn about the final result.

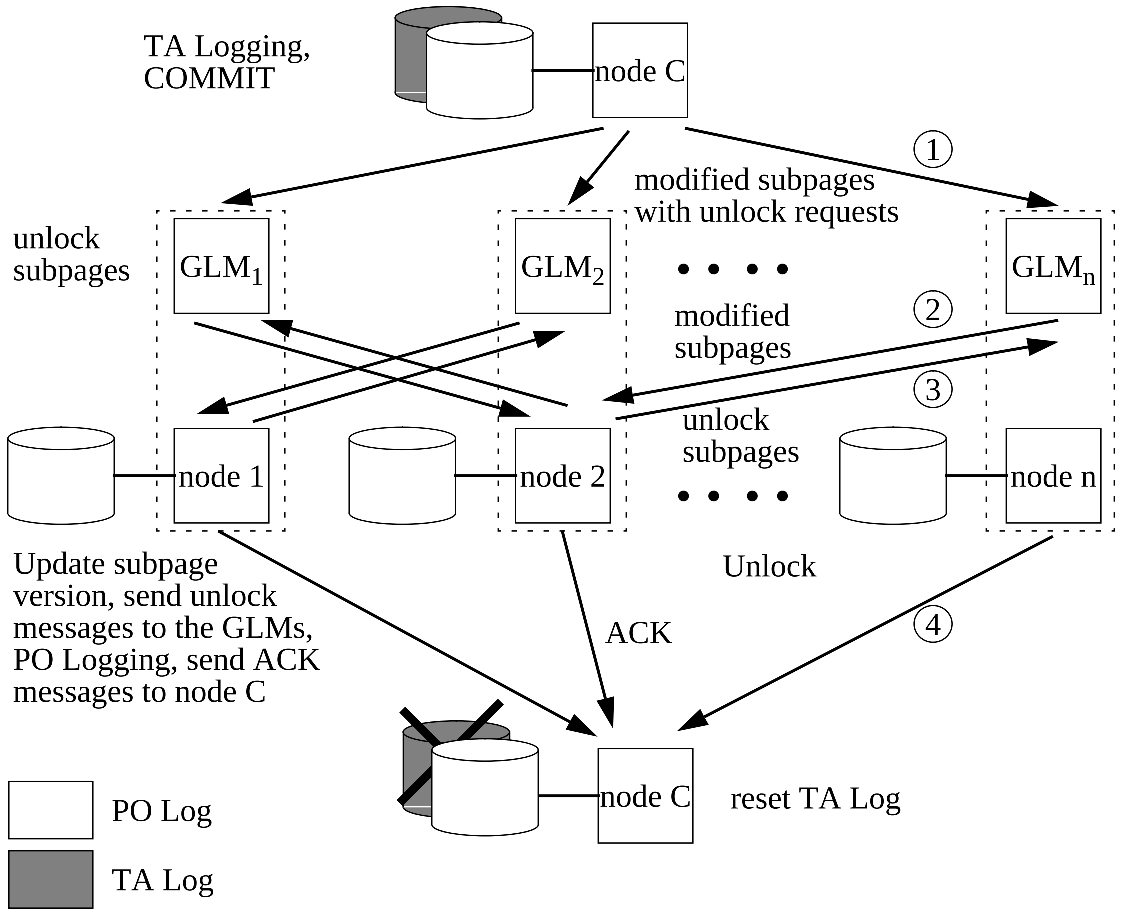


**FIGURE 2. One-Phase COMMIT: Double Logging Protocol**

## 3.4 One-Phase COMMIT (optimized) Protocol: Double Logging Solution

Because of the relatively high communication cost in the execution of the presented 2PC algorithm, the unavoidable delay in releasing the global locks and the resulting performance loss, we developed an alternative solution, based on the ideas of [16]. This time we perform the COMMIT locally on node C without requiring any coordination with the page owners. This reduces dependencies and contributes to the idea of maximum node autonomy.

However, the new solution requires additional log space in order to guarantee the durability of any modification at node C. For this purpose, we define separate log files, called TA-logs, which are either stored on separate disks or on extra partitions of the existing log disks. The TA-log of C is a temporary log space for REDO-records of update operations performed by C on pages not owned by C, as well as for UNDO-records for pages owned by C itself (for the execution of ROLLBACK). The permanent log files on each node, defined as private logs in the first protocol, are now called PO-logs for better understanding. Figure 2 illustrates the protocol in detail.

When committing a local transaction, node C writes any log information for modified subpages, including the respective UNDO-records for pages that it owns, to its private TA-log. As soon as the final COMMIT-record is on permanent storage, the transaction is definitely committed and its data modifications may be made visible to other transactions.

Note that this takes place in our protocol without any communication steps. C may now release the local locks and send the remaining unlock requests as well as not owned modified subpages to the corresponding page owners via their GLMs (steps ① and ②). After the remote POs perform the necessary processing on their local page copies and lock tables, they acknowledge this to the GLMs (step ③), which in their turn release the global locks without any further coordination with other nodes in the system.

After the execution of the described steps, the page owners transfer the necessary REDO data onto their local PO-log (UNDO-information need not be written, as the transaction has already committed at this time). This I/O is done asynchronously to the further transaction processing, which limits the overhead of the double logging protocol to a great extent. After the log records are completely written to the page owners' PO-log, the POs acknowledge the transfer to node C (step ④), which, after having received an acknowledgment from any involved page owner, may reset its own TA-log. This is the case, because the REDO records for the transaction are now made durable by the page owners and need no longer be kept redundant at node C.

## 3.5 Discussion

We presented two alternative solutions for the COMMIT-problem in a logically partitioned distributed environment. The first concept is based on a regular two-phase COMMIT algorithm. This introduces well known problems such as high communication cost, low node autonomy and thus late lock releases. The latter problem has severe effects on inter-transaction parallelism in the system. Since locks on modified subpages have to be kept by the GLMs till the executing transaction definitely commits in the second phase, lock holding time increases and a previously modified object cannot be reused by any other transaction until the second communication phase comes to an end. Therefore, this technique may lead to higher ABORT rates by timing out a large number of waiting transactions.

Therefore, we developed the alternative one-phase solution, where locks may be released sooner[2] and autonomously by the GLMs. COMMIT takes place on C locally, which optimizes transaction COMMIT time but also introduces a new problem. According to the WAL-principle all log data has to be written out to stable storage before data modifications are made visible to concurrent transactions by releasing the corresponding locks. This implies the necessity of additionally logging all modifications temporarily at node C on a separate log file. Thus log space is doubled. However, C's TA-log may be reset as soon as all log data are correctly transferred onto the page-owners' PO-logs. Furthermore, double logging implies doubling the number of I/O operations. Considering the fact, that writing the PO-log records takes place asynchronously, we can tolerate this overhead, which is compensated by the higher inter-transaction parallelism enabled by this strategy.

## 3.6 Adaptations for Static Page Ownership Distribution

Both concepts (single/double logging) can be adapted to support a simpler concurrency/coherency

2. It takes 3 message passing rounds and 1 logging session before being able to release locks in the one-phase commit protocol, compared to 4 message passing rounds (5 in case of abort) and 2 (3 in case of abort) logging sessions in the two-phase commit protocol.

scheme, where page ownership does not migrate across the nodes but is statically distributed at system start: the GLM of a page is simultaneously assigned its ownership. Thus, there is no need for the extra subpage transfer in the COMMIT protocols between GLM and PO, as in the case of a dynamic page ownership environment.

The static distribution of the page ownership also implies that all log records for a page are written to the same log-file, namely the (PO-)log of the GLM/PO. This has the effect of dividing the global conceptual log into disjunct partitions. We do not longer need to coordinate the LSN numbers between the nodes in the system, but may assign LSNs individually on each node, monotonically increasing for each log file. When a node fails, we know the set of pages (owner partition) affected by the crash as well as the log file on which the necessary log data can be found.

## 4 Recovery Processing

Based on the previously described logging schemes performed during COMMIT processing of any update transaction, we are now ready to present the actions to be taken during the recovery after a single or multiple node failure. Before a crashed node may restart, its main memory data structures must be consistently reconstructed, as well as the last consistent state of the database. In the following we describe the steps to be performed.

### 4.1 Reconstruction of the Global Lock Information

The global lock table of a failed node can be restored independently of any log information and thus in parallel to the reconstruction of data pages described in Section 4.2. The only assumption that has to be made in this context is that the reconstruction of lock information has to be finished, before the PO-Recovery, described in Section 4.3, can begin.

In Section 2.2 we have already outlined that lock information is duplicated during regular processing in the GLM's global lock table as well as the referencing node's local lock table. It is due to this redundancy that the recovery of the global lock manager is quite simple as is shown in the following example, presented in Figure 3. It presents the lock information before the crash (at node 1) and the reconstruction procedure after the crash. We use the following notation. First, for the LLM table: **lock-kind** (X-, R-, IR-, IX-lock), **page-id.subpage-id** (in case of a page lock the subpage-id is omitted and the LLM keeps track of the actually accessed subpages in a separate space, see **READ/WRITE** notation in Figure 3), **transaction-id** (e.g. T3). In the GLM table, we additionally store the transaction's home node along with the transaction-id (**transaction-id, transaction-node**).

The surviving nodes 3 and 5 send their LLM-table entries to the failed GLM node 1, which recovery coordinator R merges the received lock messages to reconstruct the partition's GLM-table. Each surviving node learns about another node's failure by broadcast (sent by the first node realizing the failure or by a predetermined recovery coordinator node R). It then searches its own local lock table for locks held on objects of the failed node's partition and sends this information to node R. This node collects and merges the received messages and thus reconstructs the relevant entries for the lost global lock table. Note that entries from the lost local lock table are not represented in the reconstructed version. This situation is nevertheless correct, since any local transaction's locks are released when the transaction aborts. Therefore, the local lock table of a restarting

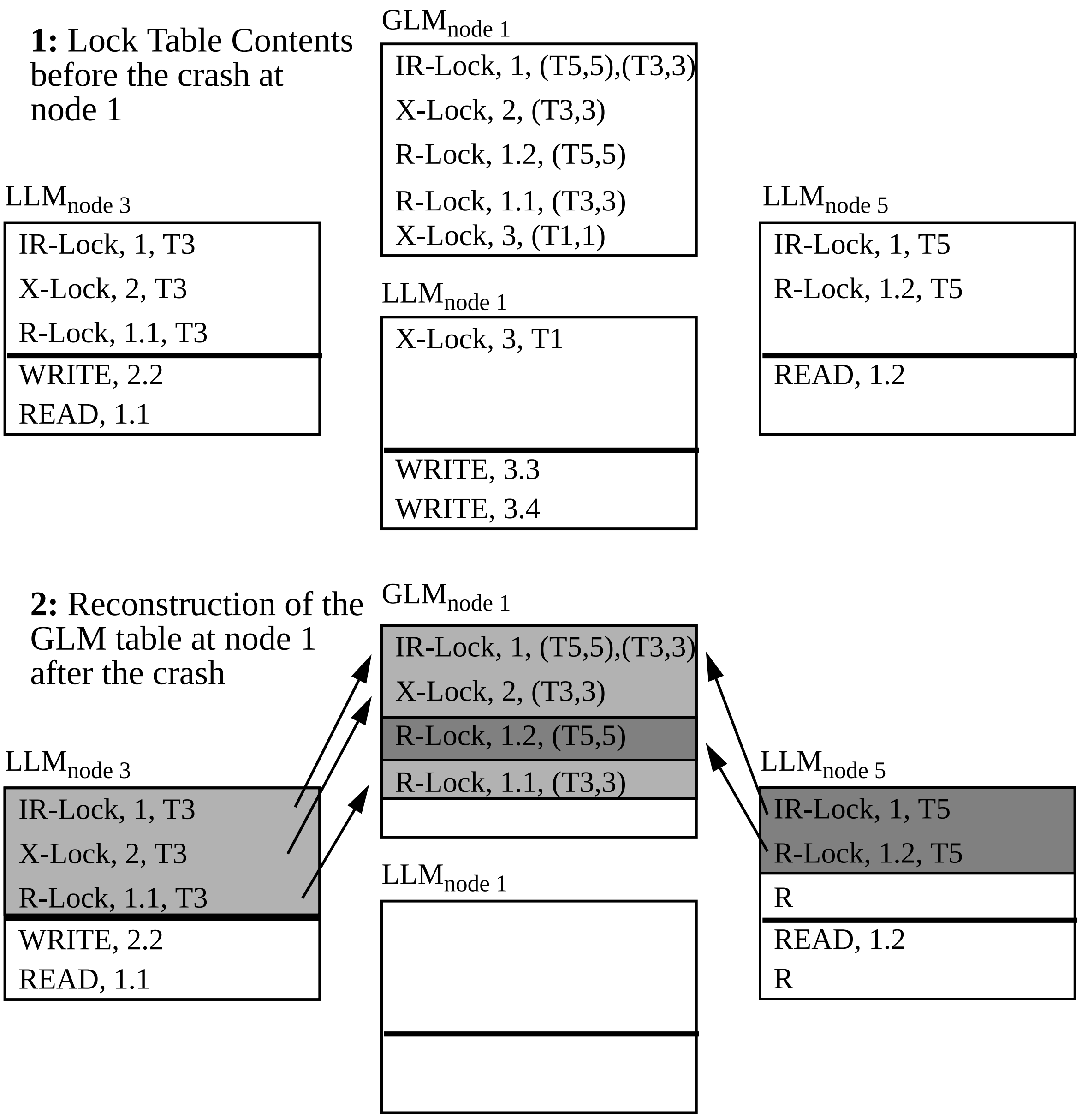


**FIGURE 3. Reconstruction of the GLM Table after a Crash**

node is empty as well.

## 4.2 REDO-Recovery

Independently from the steps for the reconstruction of lock information described above, the following actions are necessary to bring the database in a consistent state, using the physical database on disk and the physical log information. The necessary REDO-recovery (NOFORCE/NOSTEAL) is performed in three phases, which are sequentially executed as described below:

**Analysis Pass**
The first step to be taken is to read and evaluate the failed node's (PO-)log, in order to collect

information required for the second pass, in which REDO recovery is performed.

First of all, information on the set of winner and loser transactions is processed. Winners are those local transactions, which committed correctly before the failure occurred and may therefore have caused modifications which have to be redone in order to guarantee durability. Those transactions have written a COMMIT record to the local log in case of single logging. In case of double logging any REDO record found on the local log belongs to a winner transaction and therefore we do not need this distinction.

Based on this information we determine the set of pages potentially requiring REDO recovery (REDO candidates). Those are pages for which a REDO record of a winner transaction has been found on the local log. As we do not know when the page has been written to disk the last time, we have to consider it in the following REDO pass. All REDO candidates have to be loaded into the cache. The LSNs of the last REDO record reflected in the page can be read from the page header. By determining the minimum of those LSNs we get the Start_REDO_LSN, which is the starting point on the log for the execution of REDO recovery.

The effort of the analysis pass can be restricted by the use of checkpoints as described in Section 3.1: we do not have to read the log from the beginning then, but we can start from the last CHCKPT-record and read through the most recent log data.

Using the double logging strategy, the work to be performed during analysis can be restricted to determining the log starting point for the following REDO pass. We do not need more information, since we have to prepare PO recovery in the course of REDO actions, and thus, must consider every page as a potential REDO candidate. Thus, Start_REDO_LSN may be administrated as a persistent pointer on safe storage in this case.

**REDO Pass**
During the REDO pass the log has to be read a second time, starting at the previously determined Start_REDO_LSN. As usual in centralized database systems we employ a REDO-record for the recovery of a cached database page, if and only if the page's LSN stored in its header is less than the REDO-record's LSN. This algorithm is also supported by our LSN scheme (see Section 3.2) where LSNs increase for each page across the nodes. Applying the stored AFIM for a subpage, the page LSN has to be set accordingly and the coherency information (version number) has to be incremented. This guarantees consistent administration of recovered pages and permits quick restart after writing recovered pages through to disk at the end of recovery processing.

In our distributed environment, the REDO pass is also used in order to collect information on the owner partition of a crashed node before the failure. We have already pointed out that according to the protocols for the transfer of page ownership, a node may only find usable log records on its private log, when it had been owner of the respective page. Therefore, we insert each page_id and its version vector into a list (PO-list), whenever a log record is applied for the page during REDO recovery. At the end of the REDO pass the list contains information on any page that has been owned by the failed node and its newest version did not reside on disk. These pages are missed by the algorithm but a consistent state of distribution for page ownership may nevertheless be reconstructed, as will be shown below in Section 4.3.

**Transfer of TA-Log-Information**
Yet, there are some difficulties to be regarded when the double logging strategy is employed. In

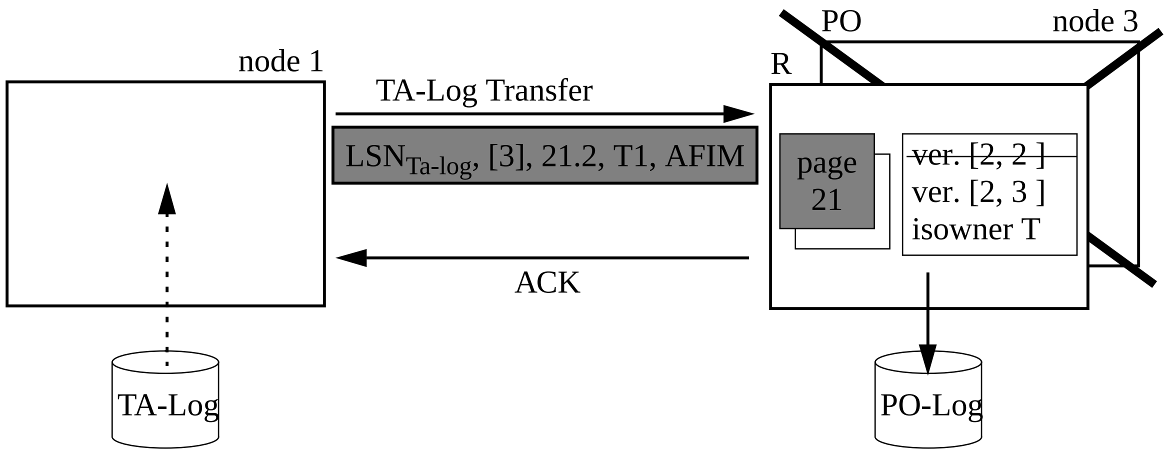


**FIGURE 4. Example: Transfer of a Missing Ta-Log Record after a Crash**

this case, we can think of situations where one or more TA-logs had not been completely transferred from a committing transaction at node C to the page owners at other nodes, before a node crash occurred. We can think of C itself, having failed, or one of the POs before it had accomplished its PO-log or before it sent the acknowledgment back to node C. Any of these situations has to be distinguished. This can be achieved by inserting the version number of the described subpage into each REDO-record on the TA-log.

Whenever node C times out waiting for the POs' acknowledgments, or whenever a failed node finds unacknowledged records on its own TA-log, it sends the relevant entries to the assigned POs via their GLMs once again. Comparing the existing reconstructed page version in the page table to the number denoted in the TA-log-record, a PO can determine, whether the record must be transferred to the node's PO-log or whether an acknowledgment has to be sent to C without any additional action. An example for this situation is shown in Figure 4.

This situation shows node 3[3] after its crash. R reconstructed subpage 21.2 from the local log information up to version number 2. Yet, there is still a REDO-record with version 3 existing on node 1's TA-log for this subpage. The received record is transferred to node 3's PO-log by R and the page version is overwritten with the subpage's AFIM. After the record is written to permanent storage, R sends an acknowledgment to node1, which in turn may reset its TA-log.

After having accomplished these additional steps, every page belonging to the failed owner's partition is reconstructed in its latest state and should now be written to the physical database to avoid the need for a new recovery pass in case of a second failure of the same node.

## 4.3 Reconstructing (Dynamically Distributed) Page Ownerships

Up to this point we did not require any temporary coordination between the different phases of recovery among the different nodes. In order to proceed now to the reconstruction of the page ownership, our algorithm requires that the reconstruction of the global lock table information has

3. Node 3 owned page 21 (isowner T), which comprises of two subpages.

terminated.

During a crash, the affected nodes loose their memory contents and thus the owner table information for pages of their partition. Surviving nodes have to lock owner table entries referring to a failed PO, as soon as they hear about its failure. Therefore, the reconstruction must proceed in two steps.

**First step: unblocking the locked owner table entries at surviving nodes by claiming the page ownership**
The failed node divides the PO list created during REDO (see Section 4.2) according to the GLM distribution of the relevant pages and sends each fragment as an “i_am_PO”-message to each GLM. These messages serve to claim again an existing page ownership of the failed node to the surviving GLM, which still locks the owner table entry. In the same way, the recovering node receives information from all surviving nodes regarding ownerships for pages of its GLM partition, which will be used in reconstructing the owner table of the failed node (see second step). The surviving GLMs compare the received “i_am_PO”-messages with their locked owner table entries and give the locks free. Our logging and recovery protocols guarantee that only one node per page can claim the page ownership, as only one node can possess REDO-records with changes that are not yet reflected on the page (see Lomet’s ONE-LOG redo requirement [7], Section 5).

Pages for which the old PO does not claim its ownership (because it did not find a REDO record for the page on its log), and for which other copies possibly exist in the system, must be assigned a new owner. This is achieved by making one of the lock holders become PO. Unlocked page copies in the system must be invalidated to guarantee a consistent copyset after a failure of the PO. The new copyset is the set of nodes holding a lock on the page or subpage. Note that locks on a subsegment do not guarantee that a page belonging to the subsegment is cached at the lock holding node.

**Second step: reconstructing the owner table of the failed node**
The lost owner table of each failed node itself has also to be reconstructed. This is achieved similar to the reconstruction algorithm for the global lock information, by receiving “i_am_PO”-messages from each other node in the system and merging the information into a new owner table. This is simple, because our algorithm guarantees that at most one node announces its ownership for the same page. Pages of the local partition which have no PO assigned, have to be treated similar to the above case: since the global lock information is already reconstructed, a new PO must be assigned from the set of lock holders. The copyset for every page is the set of page- or subpage-lock holders, after unlocked copies have been invalidated. The necessary communication steps and the reconstruction of the GLM’s owner table are illustrated in Figure 5.

After a failure of nodes 2 and 3, both GLMs’ owner tables have to be reconstructed. Surviving node 1 sends an “i_am_PO” announcement for page 17 to the corresponding GLM (node 3). Node 2, which already performed REDO for page 21 announces its ownership for page 21 respectively. This means, it has to send an “i_am_PO” message to the GLM of page 21, which in this example is node 3. Node 3, has already performed REDO for page 15 and knows that this page belongs to its own partition, as it itself is the GLM. After merging the received “i_am_PO”-information the (crashed) node’s owner table is partially reconstructed.

In case of double logging, the transfer of existing TA-log-records (see Section 4.2) has to be accomplished after every page is assigned a PO. Node C, which has unacknowledged records on

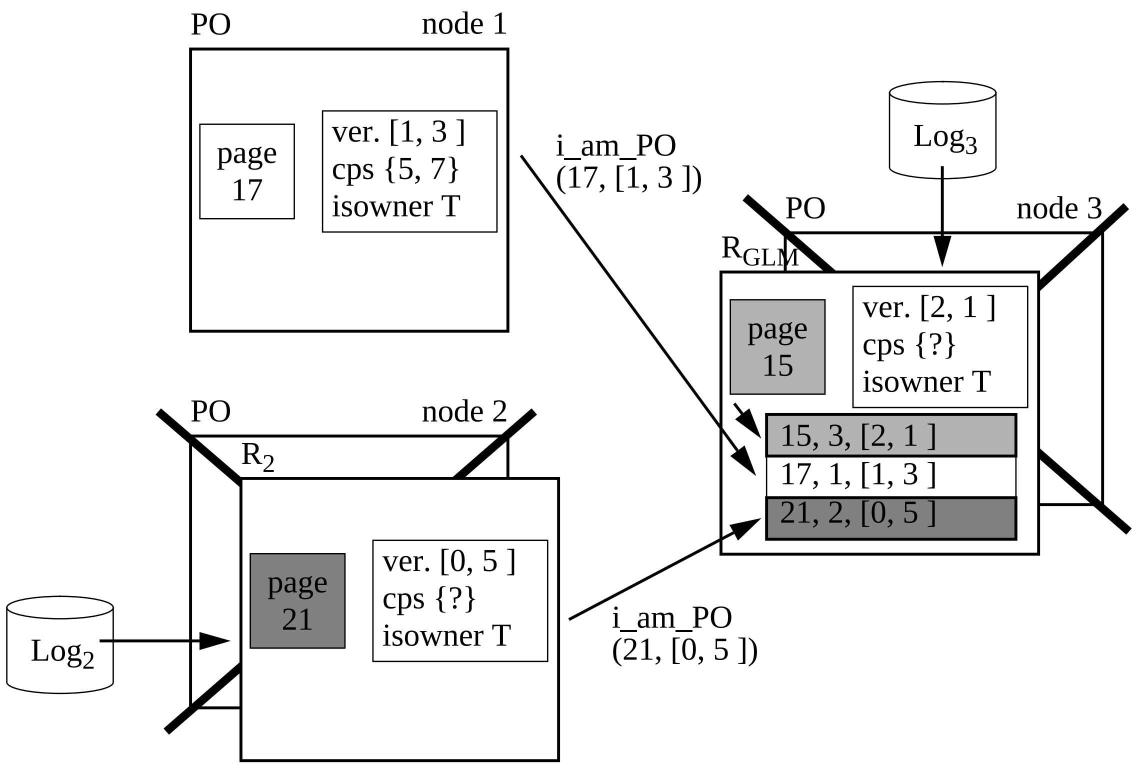


**FIGURE 5. Example: Reconstruction of the Owner Table with i_am_PO Messages**

its TA-log, sends the records to the GLM which can now identify the PO of the page. If the page does not have a PO after REDO-recovery (no lock holders) the GLM itself takes over the ownership for the page and logs the TA-records in its private PO-log.

## 4.4 Adaptations for Static Page Ownership Distribution

In case of a static distribution of page ownerships among the system's nodes, recovery does not have to reconstruct any distribution information (owner table, copyset) as described in the previous sections. Instead, every failed node knows exactly about the set of pages belonging to its own partition and requiring REDO from its local log. It does not have to communicate with other nodes to announce page ownership or to determine a new page owner for pages, which actual versions are currently on disk.

Not only communication overhead, but also temporal dependencies between the recovery processes taking place on different nodes simultaneously after a multiple node failure are avoided by this strategy. Moreover, there is no temporal dependency between the reconstruction of lock information and REDO-recovery on each node, as it was necessary in the first case, when PO-recovery required the global lock information to be in a consistent state before it could start execution.

# 5 Related Work

**ARIES**
In [11], Mohan and al. present some of the most important ideas on logging and recovery in their presentation of the ARIES system. They introduce the idea of log sequence numbers as identifiers for log records in a multiple node system, as well as the execution of fuzzy checkpoints. The algorithm is described for a NOFORCE/STEAL environment and performs record level operation (logical) logging. The most basic idea is the introduction of compensation log records, which are logged during the execution of ROLLBACKs. Since we employ a NOSTEAL strategy many of these concepts are not reflected in our work.

**Rahm's Double Logging**
The idea of double logging is introduced by Rahm in [16], where the concept of Primary Copy Locking (PCL) and it's implications for logging and recovery are discussed. The presented protocols rely on a single logfile per node, where modifications of the node's own pages as well as updates of remote pages are logged. In Rahm, only those remote pages are transferred to their owners and logged a second time there. This results in less I/O overhead than in our protocols, where every modification has to be logged twice.

Nevertheless, our protocol has several advantages compared to Rahm's ideas. Although we have to deal with more I/O-operations, our solution results in a simpler LSN-scheme, where in case of static distribution of page ownership no coordination of LSNs between different nodes is necessary. Moreover, only records which are relevant for the recovery of a failed node are written to a node's private log (PO-log) in correct sequential order. The additional space needed for the second logfile is restricted to a minimum since the TA-log may be reset early. Furthermore, our concept enables us to complete a transfer of log information to the page owner after single or multiple node failures. This problem is totally ignored in [16].

**Lomet's ONE-LOG Redo Requirement**
The general concept of multiple REDO-logs in a shared-disk database management system (DBMS) is based on the work by Lomet [7]. Private logs are used there to avoid sending remote logging messages and the need for synchronization on a common (single) log. The goal is to support the ONE-LOG redo recovery paradigm, which demands that for the recovery of each failed node only the node's private log must be read. The paradigm is valid in our environment, too, but had to be expanded to fit the requirements of an architecture with dynamic distribution of page ownership. In our system, REDO is performed on each node using the PO-log's records. However, in some cases it is necessary to take into account the contents of existing TA-logs, in addition to the (private) PO-logfile.

**Mohan's LSN-Scheme**
Our LSN-scheme used, when dynamically distributed page ownerships are employed, is closely related to Mohan's ideas [14], where a similar problem of logging and recovery in shared-disk DBMS's with dynamic coherency strategies is discussed. This is where the concept of Update Sequence Numbers (USN's) is developed, which is reused in our environment. The technique enables us to identify the actual owner of a page from the REDO-information on its local log.

However, Mohan's solution basically differs in the conception of dynamic page ownership. The authors do not assume the existence of a logical fragmentation of the physical database into distinct owner-partitions, but always declare the holder of an exclusive lock the owner of an object, which then is responsible for writing lock information and the actual page copy to permanent storage. The solution is based on a coherency protocol using P- and L-locks, which can be compared to some kind of authorization scheme. Unfortunately, these concepts support only a very restricted form of record locking, where it is not possible to concurrently update different records of the same page at different nodes in the system. The right to modify a record of a page on an arbitrary node is bound to the (exclusive) possession of a P-lock for that page. In contrast, our protocols support the hierarchical lock-protocol to a full extent and allow subpages of the same page to be modified concurrently at different nodes.

## 6 Summary and Conclusions

In this work we presented concepts for a logging and recovery component in a NOFORCE/NOSTEAL shared-disk DBMS employing a hierarchical RX-lock protocol. Our suggestions are based on two different protocols: a two-phase commit, single logging protocol and a one-phase commit, double logging protocol. The protocols were compared to each other with respect to runtime performance and performance of recovery execution after a single or multiple node failure.

The double logging protocol doubles the absolute number of I/O-operations and the log space compared to the single logging solution. Yet, part of the necessary I/O may be executed asynchronously and every node’s temporary TA-log may be reset after the transfer of the records to the corresponding page owners is completed. Better runtime performance makes the protocol especially attractive for our purposes, since it can be assumed that recovery is performed very seldom compared to regular transaction processing. Thus, the additional communication overhead needed during recovery processing can be tolerated.

Both protocols may be used with both statically as well as dynamically distributed page ownerships. The impact of dynamic distribution of page ownership on the recovery algorithm has been also examined. The static concept is very promising with respect to recovery aspects, since recovery work is executed autonomously on every failed node and does not require additional internode communication during recovery. In contrast, the dynamic approach allows more runtime flexibility, since page ownership is adapted to the system’s actual referencing behavior. In order to achieve this maximum degree of adaptability, we must accept higher efforts during recovery processing, when the actual state of distribution has to be reconstructed after a system failure.

Our theoretical examination shows that the combination of double logging and static page ownership is the most promising solution for our system environment. Yet, this hypothesis has still to be proved by performance tests.

## 7 References

[1] Bayer R., Heller H., Reiser A.
Parallelism and Recovery in Database Systems. In *ACM Transactions on Database Systems*, Vol. 5, No. 2, June 1980, pp. 139 - 156.

[2] Gray, J., Reuter, A. :
Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[3] Härder, T., Reuter, A. :
Principles of Transaction-Oriented Database Recovery. In *ACM Computing Surveys*, Vol. 15, No. 4, 1983.

[4] Härder, T. :
Realisierung von operationalen Schnittstellen. In *Datenbank-Handbuch*, Springer, Berlin, 1987 (in german).

[5] Hübner, S. :
Logging und Recovery in parallelen Datenbanksystemen. *Diploma Thesis at the Technische Universität München*, Dept. of Computer Science, 1997 (in german).

[6] Listl, A. :
Effiziente Pufferverwaltung in parallelen relationalen Datenbanksystemen. *Ph. D. Thesis at the Technische Universität München*, Dept. of Computer Science. Infix, Sankt Augustin, 1996 (in german).

[7] Lomet, D. B. :
Recovery for Shared Disk Systems Using Multiple Redo Logs. *Technical Report CRL 90/4*, DEC Cambridge Research Laboratory, 1990.

[8] Mitschang, B. :
Datenbanksysteme I. *Lecture Notes*, *Technische Universität München*, Dept. of Computer Science, SS 1995 (in german).

[9] Mitschang, B. :
Architektur und Implementierung von Datenbanksystemen. *Lecture Notes, Technische Universität München*, Dept. of Computer Science, WS 95/96 (in german).

[10] Mitschang, B. :
Transaktionssysteme, parallele und verteilte Datenbanksysteme. *Lecture Notes, Technische Universität München*, Dept. of Computer Science, SS 1996 (in german).

[11] 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. In *ACM Transactions on Database Systems*, Vol. 17, No. 1, 1992.

[12] Mohan, C., Levine, F. :
ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, San Diego, 1992.

[13] Mohan, C., Narang, I. :
Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment. In *Proc. of the 17th International Conference on Very Large Data Bases,* Barcelona, 1991.

[14] Mohan, C., Narang, I. :
Data Base Recovery in Shared Disks and Client-Server Architectures. In *Proc. of the 12th International Conference on Distributed Computing Systems,* Yokohama, 1992.

[15] Mohan, C., Pirahesh, H. :
ARIES/RRH: Restricted Repeating of History in the ARIES Transaction Recovery Method. In *Proc. of the 7th International Conference on Data Engineering*, Kobe, 1991.

[16] Rahm, E. :
Recovery Concepts for Data Sharing Systems. *Technical Report 14/89, University of Kaiserslautern*, Dept. of Computer Science, 1989.

[17] Rahm, E. :
Mehrrechner Datenbanksysteme - Grundlagen der parallelen und verteilten Datenbankverarbeitung. Addison-Wesley, Bonn, Paris, Reading Mass, 1994 (in german).

[18] Reuter, A. :
Maßnahmen zur Wahrung von Sicherheits- und Integritätsbedingungen. In *Datenbank-Handbuch*, Springer, Berlin, 1987 (in german).

[19] Rothermel, K., Mohan, C. :
ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions. In *Proc. of the 15th International Conference on Very Large Data Bases*, Amsterdam, 1989.

[20] Schmalz, I. :
Implementierung eines hierarchischen Synchronisationsverfahrens für ein paralleles Datenbanksystem. *Diploma Thesis at the Technische Universität München*, Dept. of Computer Science, 1996 (in german).

SFB 342: Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

bisher erschienen:

Reihe A

**Liste aller erschienenen Berichte von 1990-1994 auf besondere Anforderung**

| | |
|---|---|
| 342/01/95 A | Hans-Joachim Bungartz: Higher Order Finite Elements on Sparse Grids |
| 342/02/95 A | Tao Zhang, Seonglim Kang, Lester R. Lipsky: The Performance of Parallel Computers: Order Statistics and Amdahl's Law |
| 342/03/95 A | Lester R. Lipsky, Appie van de Liefvoort: Transformation of the Kronecker Product of Identical Servers to a Reduced Product Space |
| 342/04/95 A | Pierre Fiorini, Lester R. Lipsky, Wen-Jung Hsin, Appie van de Liefvoort: Auto-Correlation of Lag-k For Customers Departing From Semi-Markov Processes |
| 342/05/95 A | Sascha Hilgenfeldt, Robert Balder, Christoph Zenger: Sparse Grids: Applications to Multi-dimensional Schrödinger Problems |
| 342/06/95 A | Maximilian Fuchs: Formal Design of a Model-N Counter |
| 342/07/95 A | Hans-Joachim Bungartz, Stefan Schulte: Coupled Problems in Microsystem Technology |
| 342/08/95 A | Alexander Pfaffinger: Parallel Communication on Workstation Networks with Complex Topologies |
| 342/09/95 A | Ketil Stolen: Assumption/Commitment Rules for Data-flow Networks - with an Emphasis on Completeness |
| 342/10/95 A | Ketil Stolen, Max Fuchs: A Formal Method for Hardware/ Software Co-Design |
| 342/11/95 A | Thomas Schnekenburger: The ALDY Load Distribution System |

SFB 342: Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

| | |
|---|---|
| 342/12/95 A | Javier Esparza, Stefan Römer, Walter Vogler: An Improvement of McMillan's Unfolding Algorithm |
| 342/13/95 A | Stephan Melzer, Javier Esparza: Checking System Properties via Integer Programming |
| 342/14/95 A | Radu Grosu, Ketil Stolen: A Denotational Model for Mobile Point-to-Point Dataflow Networks |
| 342/15/95 A | Andrei Kovalyov, Javier Esparza: A Polynomial Algorithm to Compute the Concurrency Relation of Free-Choice Signal Transition Graphs |
| 342/16/95 A | Bernhard Schätz, Katharina Spies: Formale Syntax zur logischen Kernsprache der Focus-Entwicklungsmethodik |
| 342/17/95 A | Georg Stellner: Using CoCheck on a Network of Workstations |
| 342/18/95 A | Arndt Bode, Thomas Ludwig, Vaidy Sunderam, Roland Wismüller: Workshop on PVM, MPI, Tools and Applications |
| 342/19/95 A | Thomas Schnekenburger: Integration of Load Distribution into ParMod-C |
| 342/20/95 A | Ketil Stolen: Refinement Principles Supporting the Transition from Asynchronous to Synchronous Communication |
| 342/21/95 A | Andreas Listl, Giannis Bozas: Performance Gains Using Subpages for Cache Coherency Control |
| 342/22/95 A | Volker Heun, Ernst W. Mayr: Embedding Graphs with Bounded Treewidth into Optimal Hypercubes |
| 342/23/95 A | Petr Jancar, Javier Esparza: Deciding Finiteness of Petri Nets up to Bisimulation |
| 342/24/95 A | M. Jung, U. Rüde: Implicit Extrapolation Methods for Variable Coefficient Problems |
| 342/01/96 A | Michael Griebel, Tilman Neunhoeffer, Hans Regler: Algebraic Multigrid Methods for the Solution of the Navier-Stokes Equations in Complicated Geometries |

| | |
|---|---|
| 342/02/96 A | Thomas Grauschopf, Michael Griebel, Hans Regler: Additive Multilevel-Preconditioners based on Bilinear Interpolation, Matrix Dependent Geometric Coarsening and Algebraic-Multigrid Coarsening for Second Order Elliptic PDEs |
| 342/03/96 A | Volker Heun, Ernst W. Mayr: Optimal Dynamic Edge-Disjoint Embeddings of Complete Binary Trees into Hypercubes |
| 342/04/96 A | Thomas Huckle: Efficient Computation of Sparse Approximate Inverses |
| 342/05/96 A | Thomas Ludwig, Roland Wismüller, Vaidy Sunderam, Arndt Bode: OMIS --- On-line Monitoring Interface Specification |
| 342/06/96 A | Ekkart Kindler: A Compositional Partial Order Semantics for Petri Net Components |
| 342/07/96 A | Richard Mayr: Some Results on Basic Parallel Processes |
| 342/08/96 A | Ralph Radermacher, Frank Weimer: INSEL Syntax-Bericht |
| 342/09/96 A | P.P. Spies, C. Eckert, M. Lange, D. Marek, R. Radermacher, F. Weimer, H.-M. Windisch: Sprachkonzepte zur Konstruktion verteilter Systeme |
| 342/10/96 A | Stefan Lamberts, Thomas Ludwig, Christian Röder, Arndt Bode: PFSLib -- A File System for Parallel Programming Environments |
| 342/11/96 A | Manfred Broy, Gheorghe Stefanescu: The Algebra of Stream Processing Functions |
| 342/12/96 A | Javier Esparza: Reachability in Live and Safe Free-Choice Petri Nets is NP-complete |
| 342/13/96 A | Radu Grosu, Ketil Stolen: A Denotational Model for Mobile Many-to-Many Data-flow Networks |
| 342/14/96 A | Giannis Bozas, Michael Jaedicke, Andreas Listl, Bernhard Mitschang, Angelika Reiser, Stephan Zimmermann: On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS Project |
| 342/15/96 A | Richard Mayr: A Tableau System for Model Checking Petri Nets with a Fragment of the Linear Time $\mu$-Calculus |

|  |  |
|---|---|
| SFB 342: | Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen |
| 342/16/96 A | Ursula Hinkel, Katharina Spies: Anleitung zur Spezifikation von mobilen, dynamischen Focus-Netzen |
| 342/17/96 A | Richard Mayr: Model Checking PA-Processes |
| 342/18/96 A | Michaela Huhn, Peter Niebert, Frank Wallner: Put your Model Checker on Diet: Verification on Local States |
| 342/01/97 A | Tobias Müller, Stefan Lamberts, Ursula Maier, Georg Stellner: Evaluierung der Leistungsfähigkeit eines ATM-Netzes mit parallelen Programmierbibliotheken |
| 342/02/97 A | Hans-Joachim Bungartz and Thomas Dornseifer: Sparse Grids: Recent Developments for Elliptic Partial Differential Equations |
| 342/03/97 A | Bernhard Mitschang: Technologie für Parallele Datenbanken - Bericht zum Workshop |
| 342/04/97 A | nicht erschienen |
| 342/05/97 A | Hans-Joachim Bungartz, Ralf Ebner, Stefan Schulte: Hierarchische Basen zur effizienten Kopplung substrukturierter Probleme der Strukturmechanik |
| 342/06/97 A | Hans-Joachim Bungartz, Anton Frank, Florian Meier, Tilman Neunhoeffer, Stefan Schulte: Fluid Structure Interaction: 3D Numerical Simulation and Visualization of a Micropump |
| 342/07/97 A | Javier Esparza, Stephan Melzer: Model Checking LTL using Constraint Programming |
| 342/08/97 A | Niels Reimer: Untersuchung von Strategien für verteiltes Last- und Ressourcenmanagement |
| 342/09/97 A | Markus Pizka: Design and Implementation of the GNU INSEL-Compiler gic |
| 342/10/97 A | Manfred Broy, Franz Regensburger, Bernhard Schätz, Katharina Spies: The Steamboiler Specification - A Case Study in Focus |
| 342/11/97 A | Christine Röckl: How to Make Substitution Preserve Strong Bisimilarity |
| 342/12/97 A | Christian B. Czech: Architektur und Konzept des Dycos-Kerns |

| SFB 342: | Methoden und Werkzeuge für die Nutzung paraller Rechnerarchitekturen |
|---|---|
| 342/13/97 A | Jan Philipps, Alexander Schmidt: Traffic Flow by Data Flow |
| 342/14/97 A | Norbert Fröhlich, Rolf Schlagenhaft, Josef Fleischmann: Partitioning VLSI-Circuits for Parallel Simulation on Transistor Level |
| 342/15/97 A | Frank Weimer: DaViT: Ein System zur interaktiven Ausführung und zur Visualisierung von INSEL-Programmen |
| 342/16/97 A | Niels Reimer, Jürgen Rudolph, Katharina Spies: Von FOCUS nach INSEL - Eine Aufzugssteuerung |
| 342/17/97 A | Radu Grosu, Ketil Stolen, Manfred Broy: A Denotational Model for Mobile Point-to-Point Data-flow Networks with Channel Sharing |
| 342/18/97 A | Christian Röder, Georg Stellner: Design of Load Management for Parallel Applications in Networks of Heterogenous Workstations |
| 342/19/97 A | Frank Wallner: Model Checking LTL Using Net Unfoldings |
| 342/20/97 A | Andreas Wolf, Andreas Kmoch: Einsatz eines automatischen Theorembeweisers in einer taktikgesteuerten Beweisumgebung zur Lösung eines Beispiels aus der Hardware-Verifikation -- Fallstudie -- |
| 342/21/97 A | Andreas Wolf, Marc Fuchs: Cooperative Parallel Automated Theorem Proving |
| 342/22/97 A | T. Ludwig, R. Wismüller, V. Sunderam, A. Bode: OMIS - On-line Monitoring Interface Specification (Version 2.0) |
| 342/23/97 A | Stephan Merkel: Verification of Fault Tolerant Algorithms Using PEP |
| 342/24/97 A | Manfred Broy, Max Breitling, Bernhard Schätz, Katharina Spies: Summary of Case Studies in Focus - Part II |
| 342/25/97 A | Michael Jaedicke, Bernhard Mitschang: A Framework for Parallel Processing of Aggregat and Scalar Functions in Object-Relational DBMS |
| 342/26/97 A | Marc Fuchs: Similarity-Based Lemma Generation with Lemma-Delaying Tableau Enumeration |

SFB 342: Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

| | |
|---|---|
| 342/27/97 A | Max Breitling: Formalizing and Verifying TimeWarp with FOCUS |
| 342/28/97 A | Peter Jakobi, Andreas Wolf: DBFW: A Simple DataBase FrameWork for the Evaluation and Maintenance of Automated Theorem Prover Data (incl. Documentation) |
| 342/29/97 A | Radu Grosu, Ketil Stolen: Compositional Specification of Mobile Systems |
| 342/01/98 A | A. Bode, A. Ganz, C. Gold, S. Petri, N. Reimer, B. Schiemann, T. Schnekenburger (Herausgeber): “Anwendungsbezogene Lastverteilung”, ALV’98 |
| 342/02/98 A | Ursula Hinkel: Home Shopping - Die Spezifikation einer Kommunikationsanwendung in FOCUS |
| 342/03/98 A | Katharina Spies: Eine Methode zur formalen Modellierung von Betriebssystemkonzepten |
| 342/04/98 A | Stefan Bischof, Ernst-W. Mayr: On-Line Scheduling of Parallel Jobs with Runtime Restrictions |
| 342/05/98 A | St. Bischof, R. Ebner, Th. Erlebach: Load Balancing for Problems with Good Bisectors and Applications in Finite Element Simulations: Worst-case Analysis and Practical Results |
| 342/06/98 A | Giannis Bozas, Susanne Kober: Logging and Crash Recovery in Shared-Disk Database Systems |

SFB 342: Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

Reihe B

| | |
|---|---|
| 342/1/90 B | Wolfgang Reisig: Petri Nets and Algebraic Specifications |
| 342/2/90 B | Jörg Desel: On Abstraction of Nets |
| 342/3/90 B | Jörg Desel: Reduction and Design of Well-behaved Free-choice Systems |
| 342/4/90 B | Franz Abstreiter, Michael Friedrich, Hans-Jürgen Plewan: Das Werkzeug runtime zur Beobachtung verteilter und paralleler Programme |
| 342/1/91 B | Barbara Paech: Concurrency as a Modality |
| 342/2/91 B | Birgit Kandler, Markus Pawlowski: SAM: Eine Sortier-Toolbox -Anwenderbeschreibung |
| 342/3/91 B | Erwin Loibl, Hans Obermaier, Markus Pawlowski: 2. Workshop über Parallelisierung von Datenbanksystemen |
| 342/4/91 B | Werner Pohlmann: A Limitation of Distributed Simulation Methods |
| 342/5/91 B | Dominik Gomm, Ekkart Kindler: A Weakly Coherent Virtually Shared Memory Scheme: Formal Specification and Analysis |
| 342/6/91 B | Dominik Gomm, Ekkart Kindler: Causality Based Specification and Correctness Proof of a Virtually Shared Memory Scheme |
| 342/7/91 B | W. Reisig: Concurrent Temporal Logic |
| 342/1/92 B | Malte Grosse, Christian B. Suttner: A Parallel Algorithm for Set-of-Support<br>Christian B. Suttner: Parallel Computation of Multiple Sets-of-Support |
| 342/2/92 B | Arndt Bode, Hartmut Wedekind: Parallelrechner: Theorie, Hardware, Software, Anwendungen |
| 342/1/93 B | Max Fuchs: Funktionale Spezifikation einer Geschwindigkeitsregelung |

## SFB 342: Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

| | |
|---|---|
| 342/2/93 B | Ekkart Kindler: Sicherheits- und Lebendigkeitseigenschaften: Ein Literaturüberblick |
| 342/1/94 B | Andreas Listl; Thomas Schnekenburger; Michael Friedrich: Zum Entwurf eines Prototypen für MIDAS |