To Appear: ACM SIGMOD International Conference on the Management of Data, San Diego, June 1992.

Crash Recovery in Client-Server EXODUS

Michael J. Franklin
Michael J. Zwilling
C. K. Tan
Michael J. Carey
David J. DeWitt

Computer Sciences Technical Report #1081
March 1992

Computer Sciences Department
University of Wisconsin-Madison

This work was partially supported by the Defense Advanced Research Projects Agency under contracts N00014-88-K-0303 and NAG-2-618, by the National Science Foundation under grant IRI-8657323, and by a grant from IBM Corporation.

Crash Recovery in Client-Server EXODUS

Michael J. Franklin, Michael J. Zwilling,
C. K. Tan, Michael J. Carey, David J. DeWitt

Computer Sciences Department
University of Wisconsin - Madison

ABSTRACT

In this paper, we address the correctness and performance issues that arise when implementing logging and crash recovery in a page-server environment. The issues result from two characteristics of page-server systems: 1) the fact that data is modified and cached in client database buffers that are not accessible by the server, and 2) the performance and cost tradeoffs that are inherent in a client-server environment. We describe a recovery system that we have implemented for a particular page-server system: the client-server version of the EXODUS storage manager. The implementation supports efficient buffer management policies, allows flexibility in the interaction between clients and the server, and reduces the load on the server by performing much of the work involved in generating log records at clients. We also present a preliminary performance analysis of the implementation, examining the relative costs of logging and recovery operations and identifying areas for future improvement.

1. INTRODUCTION

Networks of powerful workstations and servers have become the computing environment of choice in many application domains. As a result, most recent commercial and experimental DBMSs have been constructed to run in such environments. These systems are referred to as *client-server DBMSs*. Recovery has long been studied in centralized and distributed database systems [Gray78, Lind79, Gray81, Moha90, BHG87, GR92] and more recently in architectures such as *shared-disk* systems [MN91, Lome90, Rahm91] and distributed transaction facilities [DST87, HMSC88]. However, little has been published about recovery issues for client-server database systems. This paper describes the implementation challenges and performance tradeoffs involved in implementing recovery in such a system, based on our experience in building the client-server implementation of the EXODUS storage manager [CDRS89, Exod91a, Exod91b].

Client-server DBMS architectures can be categorized according to whether they send requests to a server as queries or as requests for specific data items. We refer to systems of the former type as *query-shipping* systems and to those of the latter type as *data-shipping* systems. Data-shipping systems can be further categorized as *page-servers*, which interact using physical units of data (e.g. individual pages or groups of pages such as segments) and *object-servers*, which interact using logical units of data (e.g. tuples or objects) [DFMV90]. There is still much debate about the relative advantages of the different architectures with respect to current technology trends [Ston90, Comm90, DFMV90]. Most commercial relational database systems have adopted query-shipping architectures. Query-shipping architectures have the advantage that they are similar in process structure to a single-site database system, and hence,

This work was partially supported by the Defense Advanced Research Projects Agency under contracts N00014-88-K-0303 and NAG-2-618, by the National Science Foundation under grant IRI-8657323, and by a grant from IBM Corporation.

provide a relatively easy migration path from an existing single-site system to the client-server environment. They also have the advantage of minimizing communication, since only data which satisfies the query is sent from the server to the requesting clients.

In contrast to the relational DBMS systems, virtually all commercial object-oriented database systems (OODBMS) and many recent research prototypes have adopted some variant of the data-shipping approach (e.g., O2 [Deux91], ObjectStore [LLOW91], ORION [KGBW90]). Data-shipping architectures have the potential advantage of avoiding bottlenecks at the server by exploiting the processing power and memory of the client machines. This is important for performance, since the majority of the processing power and memory in a client-server environment is likely to be at the client workstations. Moreover, as network bandwidth improves, the cost of the additional communication as compared to query-shipping architectures will become less significant. Also, the data-shipping approach is a good match for object-oriented database systems, as such systems are often accessed using a persistent programming language rather than via a separate query language. For performance, database objects are "swizzled" into a main memory representation and cached in the client's memory so that they can be accessed directly by application programs. A workload which demonstrates the performance advantages of data-shipping architectures is described in [Catt91].

Implementing recovery in query-shipping architectures raises few new issues over traditional recovery approaches since the architecture of the database engine remains largely unchanged. In contrast, data-shipping architectures present a new set of problems and issues for the design of the recovery and logging subsystems of a DBMS. These arise from four main architectural features of data-shipping architectures which differentiate them from traditional centralized and distributed systems as well as from other related architectures such as shared-disk systems. These features are:

(1) Database modifications are made primarily at clients, while the stable copy of the database and the log are kept at the server.

(2) Each client manages its own local buffer pool.

(3) Communication between clients and the server is relatively expensive (e.g., compared to local interprocess communication or shared memory access).

(4) Client machines tend to have different performance and reliability characteristics and different hardware capabilities than server machines.

The client-server EXODUS Storage Manager (ESM-CS) is a data-shipping system which employs a page-server architecture. The implementation of recovery in ESM-CS involves two main components. The *logging subsystem* manages and provides access to an append-only log on stable storage. The *recovery subsystem* uses the information in the log to provide transaction rollback (e.g., abort) and system restart (i.e., crash recovery). The implementation of recovery also involves close cooperation with the buffer manager and the lock manager. The recovery algorithm is based on ARIES [Moha90]. ARIES was chosen because of its simplicity and flexibility, its ability to support the efficient STEAL/NO FORCE buffer management policy [HR83], its support for savepoints and nested-top-level actions, and its ability to support fine-grained concurrency control and logical Undo. However, the algorithm as specified in [Moha90] cannot be directly implemented in a page-server architecture because the architecture violates some of the explicit and implicit assumptions upon which the original algorithm is based. In this paper we describe our recovery manager, paying particular attention to the modifications to the ARIES method that were required due to both the correctness and efficiency concerns of recovery in a page-server system. We also discuss several engineering decisions that were made in the design of the ESM-CS logging and recovery subsystems.

It should be noted that the ARIES algorithm has recently been extended in ways that are similar to some of the extensions we describe in this paper. [MP91] describes an extension of the algorithm which can reduce the work performed during system restart. The algorithm used in ESM-CS required a similar extension, not for efficiency, but in order to operate correctly in the page-server environment. [MNP90] and [MN91] describe extensions to ARIES for the shared-disk environment. As would be expected, some of the solutions in that environment are applicable to the page-server environment, while others are not (for both correctness and efficiency reasons). We discuss these extensions and other related work in Section 6.

The remainder of the paper is structured as follows: Section 2 describes the ESM-CS architecture with particular attention to the issues that affect logging and recovery. Section 3 provides a brief overview of ARIES. Section 4 motivates and describes the modifications made to ARIES for the page-server environment. Section 5 presents a study of the performance of the ESM-CS logging and recovery implementation. Section 6 describes related work. Section 7 presents our conclusions.

2. THE CLIENT-SERVER ENVIRONMENT

In this section we first present an overview of the ESM-CS architecture in general, and then concentrate on the design of the ESM-CS logging subsystem. The intent of this description is twofold: 1) it is intended to provide the background necessary to understand the recovery algorithm and its implementation and 2) it highlights some of the engineering decisions that were made in the design of the logging and recovery subsystems. A more detailed description of the architecture can be found in [Exod91a].

2.1. Architecture Overview

ESM-CS is a multi-user system with full support for indexing, concurrency control, and recovery, which is designed for use in a client-server environment. In addition to supporting these new features, ESM-CS provides support for all of the features provided previously by the EXODUS storage manager, such as large and versioned objects [CDRS89]. The system runs under UNIX(tm) on DECstation, SPARCstation, and SUN-4 workstations. The storage manager can be accessed through a C procedure call interface or through the E programming language [RC89, RCS91], a persistent programming language based on C++.

Figure 1 shows the architecture of ESM-CS. The design was driven by the anticipated capabilities, performance, and reliability characteristics of the clients, server, and the network. A server is expected to have more CPU power, more disk capacity, and more memory than a single client, but the sum of the processing power and memory of the clients is expected to be greater than that of the server. Clients are expected to be less reliable than the server and may not have all of their resources available for use by the database system. The main cost of communication is expected to be the CPU overhead of sending and receiving messages. The system consists of two main parts: the client library, which is linked into the user's application, and the server, which runs as a separate process.

The architecture has a clear division of labor between the server and clients. The server is the main repository for the database and the log and provides support for lock management, page allocation and deallocation, and recovery/rollback. Clients perform all data and index manipulation during normal (i.e., non-recovery or rollback) operation. Each client process (i.e., each application that is linked with the

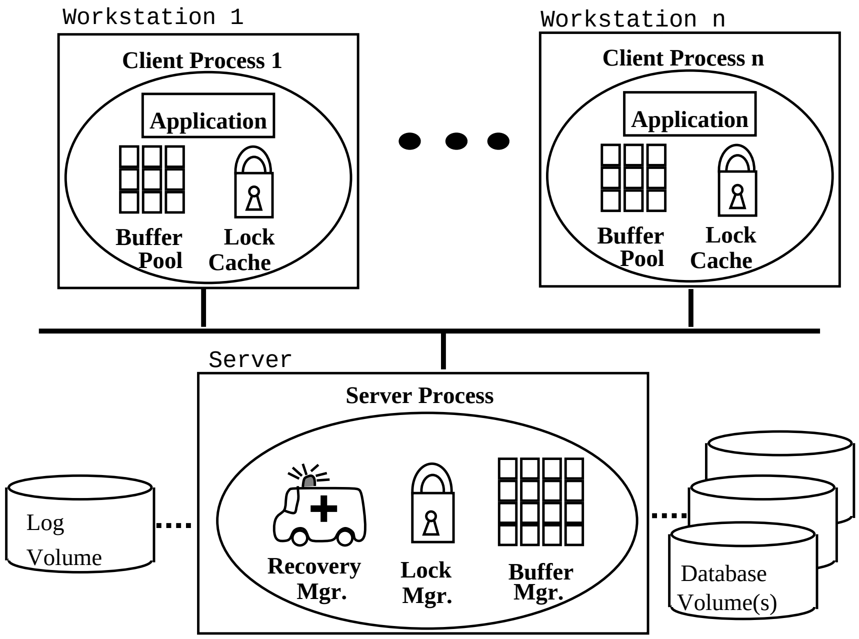

Figure 1: The Architecture of Client-Server EXODUS

client library) has its own buffer pool and lock cache and runs a single transaction at a time. The server is multi-threaded so that it can handle requests from multiple clients, and it uses separate disk processes so that it can perform asynchronous I/O. The current system does not support access to multiple servers from a single client process. Note that while Figure 1 shows clients and the server executing on separate machines, it is possible to run the server and/or any number of clients as separate processes on the same machine. Communication between clients and the server is implemented using reliable TCP connections and UNIX sockets. All communication is initiated by the client and is responded to by the server. That is, the server responds to requests from the client for specific pages, locks, and transaction services. There is no mechanism for the server to initiate contact with a client, since this would have added significant complexity to the clients, requiring them to be multi-threaded in order to respond to unexpected messages from the server.

As stated above, ESM-CS employs a page-server architecture in which the client sends requests for specific data and index pages to the server. ESM-CS uses strict two-phase locking for data and non-two-phase locking for indexes. Data is locked at a page or coarser granularity. Index page splits are logged as nested top level actions [Moha90] so they are committed regardless of whether or not their enclosing transaction commits. During a transaction, clients cache data and index pages in their local buffer pool. Before committing a transaction, the client sends all the pages modified by the transaction to the server. In the current implementation, a client's cache is purged upon the completion (commit or abort) of a transaction. Future enhancements will allow inter-transaction caching of pages [CFLS91].

Clients initiate transactions by sending a *start transaction* message to the server and can request the commit or abort of a transaction by sending a message to the server. The server can decide to abort a transaction due to a system error or deadlock. After aborting the transaction the server notifies the client that the transaction has been aborted in response to the next message it receives from the client, since there is no mechanism for the server to initiate contact with a client. During the execution of a transaction, the client generates log records for all updates to data and index pages. The server manages the log as a circular buffer, and will abort executing transactions if it is in danger of running out of log space.

2.2. Logging Subsystem

One of the main challenges in designing a recovery system is to minimize the negative performance impact of logging during normal operation. As stated in the previous section, the log in ESM-CS is kept at the server. This decision was made for two reasons: 1) we do not want to lose access to data as the result of a client failure, and 2) it is not economical to require that clients have log disks. Given that the log is kept at the server while the operations on data and indexes on behalf of application programs are performed at the clients, an efficient interface for shipping log records from the clients to the server is required. The logging subsystem is an extension of a centralized logging subsystem that is intended to work efficiently in the client-server environment.

2.2.1. Log Records and Data Pages

Our ARIES-based recovery algorithm depends upon the use of the Write-Ahead-Logging (WAL) protocol [Gray 78] at the server. The WAL protocol insures that:

1) All log records pertaining to an updated page are written to stable storage before the page itself is allowed to be over-written on stable storage.

2) A transaction cannot commit until all of its log records have been written to stable storage.

In order to implement the WAL protocol at the server, we enforce a similar protocol between clients and the server. That is, a client must send log records to the server prior to sending the pages on which the updates were performed. This policy is enforced for two reasons. First, it simplifies transaction rollback by insuring that the server has all log records necessary to rollback updates to pages at the server that contain uncommitted updates. If the policy were not enforced, transaction rollback caused by a client crash could require performing restart recovery on the affected pages at the server since necessary log records could be lost due to the client crash. Second, if the policy were not enforced between the client and the server then the server's buffer manager would be complicated by having to manage dependencies between the arrival of log records from clients and the flushing of dirty pages to stable storage.

For efficiency, clients group log records into pages and send them to the server a page at a time. Log pages are buffered at the client, and can be sent to the server at any time during a transaction, provided that they are sent in order and the WAL protocol is satisfied. When logging an update, we place a *log dependency* between the updated page and the last log page in which a log record is written for that update. This dependency is used by the buffer manager to insure that all of the log pages up to and including the log page on which the updated page is dependent are sent to the server before the updated page itself is returned to the server. A similar mechanism is used by the server to enforce the WAL protocol with respect to its buffers and stable storage.

2.2.2. Log Record Structure

The client generates one or more log records for each operation that updates a data or index page. These log records contain redo and/or undo information specific to the operation performed. That is, rather than log entire before and after images of pages, clients log only the bytes that are affected by the operation. This decision was motivated by the desire to reduce two overheads: 1) the expense of sending data from clients to the server, especially because some of the pages are quite large (e.g., pages that make up large objects can be 8K bytes or longer), and 2) the expense of writing to the log.

Another decision that was made in this regard was to not allow log records generated at the client to span log page boundaries. That is, all log records generated by clients are smaller than a log page and are wholly contained in a single log page when sent to the server. This restriction simplifies both the sending of log records at the client and the handling of log record pages at the server. For example, clients can simply send pages of log records to the server as they fill up. The server can receive log pages without needing to manage partial log records and can receive pages of log records from multiple clients without needing to merge multiple pages from a particular client. The restriction is easily implemented at the client but sometimes requires operations to be logged slightly differently from the way they were actually performed. For example, the creation of a data object that is larger than the size of a log page is logged as the *create* of the first portion of the object followed by the *append* of any remaining data.

2.2.3. Server Handling of Log Records

As a result of the log dependency mechanism and the boundary spanning restriction, a client may at times be forced to only partially fill log pages. This could result in the inefficient use of log space and unnecessary writes to the log. The server, however, is not subject to the restrictions imposed on clients and can therefore combine log records received from different clients onto the same log page and can write log records received from a given client across multiple pages in order to conserve log writes. However, the server must preserve the ordering of the log records received from a particular client and must maintain the WAL protocol between the updated pages in its buffer and the corresponding log pages. In addition to log records received from clients, the server also generates certain log records of its own. These log records include records for the commit and abort of transactions and the allocation and deallocation of pages and files. The server also writes log records during system restart and transaction rollback as described in the following section. Server log records are not subject to the size constraint that is imposed on client log records and can span multiple log pages.

3. OVERVIEW OF ARIES

In this section, we present a brief overview of the ARIES recovery method, which is used as the basis for the ESM-CS recovery system. This discussion concentrates on the features of the algorithm that are pertinent to the ESM-CS environment. For a complete discussion of the algorithm, the reader is referred to [Moha90]. ARIES is a fairly recent refinement of the Write-Ahead-Logging (WAL) protocol (see Section 2.2.1 above). The WAL protocol enables the use of a STEAL/NO FORCE buffer management policy, which means that pages on stable storage can be overwritten at any time and that data pages do not need to be forced to disk in order to commit a transaction. As with other WAL implementations, each page in the database contains a Log Sequence Number (LSN) which uniquely identifies the log record for the latest update which was applied to the page. This LSN (referred to as the *pageLSN*) is used during recovery to determine whether or not an update for a page must be redone. LSN information is also used to determine the point in the log from which the Redo pass must commence during restart from a system crash. LSNs are often implemented using the physical address of the log record in the log to enable the efficient location of a log record given its LSN.

Much of the power of the ARIES algorithm is due to its Redo paradigm of *repeating history*, in which it redoes updates for *all* transactions — including those that will eventually be undone. Repeating history greatly simplifies the implementation of fine grained locking and the use of logical undo operations as described in [Moha90, MP91]. Also, as will be described in later sections, the resulting simplicity allows ARIES to be adapted for use in many computing environments.

ARIES uses a three pass algorithm for restart recovery. The first pass is the *Analysis* pass, which processes the log forward from the most recent checkpoint. This pass determines information about dirty pages and active transactions that is used in the subsequent passes. The second pass is the *Redo* pass, in which history is repeated by processing the log forward from the earliest log record that could require redo, thus insuring that all logged operations have been applied. The third pass is the *Undo* pass. This pass proceeds backwards from the end of the log, removing from the database the effects of all transactions that had not committed at the time of the crash. These passes are shown in Figure 2. (Note that the relative ordering of the starting point for the Redo pass, the endpoint for the Undo pass, and the checkpoint can be different than that shown in the figure.) The three passes are described in more detail below.

3.1. Normal Operation

ARIES maintains two important data structures during normal operation. The first is the *Transaction Table*, which contains status information for each transaction that is currently running. This information includes a field called the *lastLSN*, which is the LSN of the most recent log record written by the transaction. The second data structure, called the *Dirty Page Table*, contains an entry for each "dirty" page. A page is considered to be dirty if it contains updates that are not reflected on stable storage. Each entry in the Dirty Page Table includes a field called the *recoveryLSN*, which is the LSN of the log record that caused the associated page to become dirty. Therefore, the *recoveryLSN* is the LSN of the earliest log record that might need to be redone for the page during restart. Log records belonging to the same transaction are linked backwards in time using a field in each log record called the *prevLSN* field. When a new log record is written for a transaction, the value of the *lastLSN* field in the Transaction Table entry is placed in the *prevLSN* field of the new record and the new record's LSN is entered as the *lastLSN* in the Transaction Table entry.

During normal operation, checkpoints are taken periodically. ARIES uses a form of fuzzy checkpoints [BHG87] which are extremely inexpensive. When a checkpoint is taken, a checkpoint record (or possibly several, if there are constraints on log record size) is constructed which includes the contents of the Transaction Table and the Dirty Page Table. Checkpoints are efficient since no operations need be quiesced and no database pages are flushed to perform a checkpoint. However, the effectiveness of

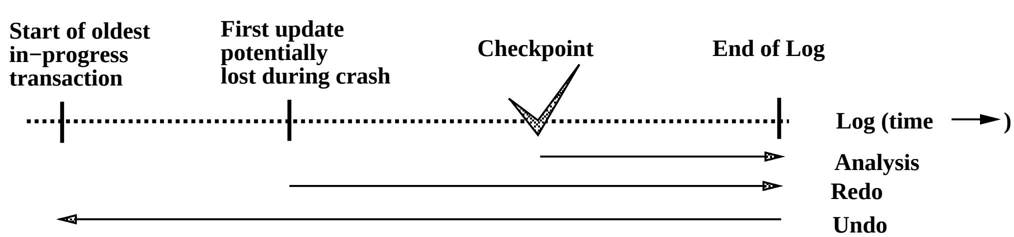

Figure 2: The Three Passes of ARIES Restart

checkpoints in reducing the amount of the log that must be maintained is limited in part by the earliest *recoveryLSN* of the dirty pages at checkpoint time. Therefore, it is helpful to have a background process that periodically writes dirty pages to stable storage.

3.2. Analysis

The job of the Analysis pass of restart recovery is threefold: 1) It determines the point in the log at which to start the Redo pass, 2) It determines which pages could have been dirty at the time of the crash in order to avoid unnecessary I/O during the Redo pass, and 3) It determines which transactions had not committed at the time of the crash and will therefore need to be undone.

The Analysis pass begins at the most recent checkpoint and scans forward to the end of the log. It reconstructs the Transaction Table and Dirty Page Table to determine the state of the system as of the time of the crash. It begins with the copies of those structures that were logged in the checkpoint record. Then, the contents of the tables are modified according to the log records that are encountered during the forward scan. When a log record for a transaction that does not appear in the Transaction Table is encountered, that transaction is added to the table. When a log record for the commit or the abort of a transaction is encountered, the corresponding transaction is removed from the Transaction Table. When a log record for an update to a page that is not in the Dirty Page Table is encountered, that page is added to the Dirty Page Table, and the LSN of the record which caused the page to be entered into the table is recorded as the *recoveryLSN* for that page. At the end of the Analysis pass, the Dirty Page Table is a conservative (since some pages may have been flushed to stable storage) list of all database pages that could have been dirty at the time of the crash, and the Transaction Table contains entries for those transactions that will actually require undo processing during the Undo phase. The earliest *recoveryLSN* of all the entries in the Dirty Page Table, called the *firstLSN,* is used as the spot in the log from which to begin the Redo phase.

3.3. Redo

As stated earlier, ARIES employs a redo paradigm called *repeating history*. That is, it redoes updates for *all* transactions, committed or otherwise. The effect of repeating history is that at the end of the Redo pass, the database is in the same state with respect to the logged updates that it was in at the time that the crash occurred. The Redo pass begins at the log record whose LSN is the *firstLSN* determined by Analysis and scans forward from there. To redo an update, the logged action is re-applied and the *pageLSN* on the page is set to the LSN of the redone log record. No logging is performed as the result of a redo. For each log record the following algorithm is used to determine if the logged update must be redone:

(1) If the affected page is not in the Dirty Page Table then the update does NOT require redo.

(2) If the affected page is in the Dirty Page Table, then if the *recoveryLSN* in the page's table entry is *greater than* the LSN of the record being checked, the update does NOT require redo.

(3) Otherwise, the LSN stored on the page (the *pageLSN*) must be checked. This may require that the page be read in from disk. If the *pageLSN* is *greater than or equal to* the LSN of the record being checked, then the update does NOT require redo. Otherwise, the update MUST be redone.

3.4. Undo

The Undo pass scans backwards from the end of the log. During the Undo pass, all transactions that had not committed by the time of the crash must be undone. In ARIES, undo is an *unconditional* operation. That is, the *pageLSN* of an affected page is not checked because it is always the case that the undo must be performed. This is due to the fact that the *repeating of history* in the Redo pass insures that all logged updates have been applied to the page.

When an update is undone, the undo operation is applied to the page and is logged using a special type of log record called a *Compensation Log Record* (CLR). In addition to the undo information, a CLR contains a field called the *UndoNxtLSN*. The *UndoNxtLSN* is the LSN of the next log record that must be undone for the transaction. It is set to the value of the *prevLSN* field of the log record being undone. The logging of CLRs in this fashion enables ARIES to avoid ever having to undo the effects of an undo (e.g., as the result of a system crash during an abort) thereby limiting the amount of work that must be undone and bounding the amount of logging done in the event of multiple crashes. When a CLR is encountered during the backwards scan, no operation is performed on the page, and the backwards scan continues at the log record referenced by the *UndoNxtLSN* field of the CLR, thereby jumping over the undone update and all other updates for the transaction that have already been undone (the case of multiple transactions will be discussed shortly). An example execution is shown in Figure 3.

In Figure 3, a transaction logged three updates (LSNs 10, 20, and 30) before the system crashed for the first time. During Redo, the database was brought up to date with respect to the log (i.e., 10, 20, and/or 30 were redone if they weren't on stable storage), but since the transaction was in progress at the time of the crash, they must be undone. During the Undo pass, update 30 was undone, resulting in the writing of a CLR with LSN 40, which contains an *UndoNxtLSN* value that points to 20. Then, 20 was undone, resulting in the writing of a CLR (LSN 50) with an *UndoNxtLSN* value that points to 10. However, the system then crashed for a second time before 10 was undone. Once again, history is repeated during Redo which brings the database back to the state it was in after the application of LSN 50 (the CLR for 20). When Undo begins during this second restart, it will first examine the log record 50. Since the record is a CLR, no modification will be performed on the page, and Undo will skip to the record whose LSN is stored in the *UndoNxtLSN* field of the CLR (i.e., LSN 10). Therefore, it will continue by undoing the update whose log record has LSN 10. This is where the Undo pass was interrupted at the time of the second crash. Note that no extra logging was performed as a result of the second crash.

In order to undo multiple transactions, restart Undo keeps a list containing the next LSN to be undone for each transaction being undone. When a log record is processed during Undo, the *prevLSN* (or

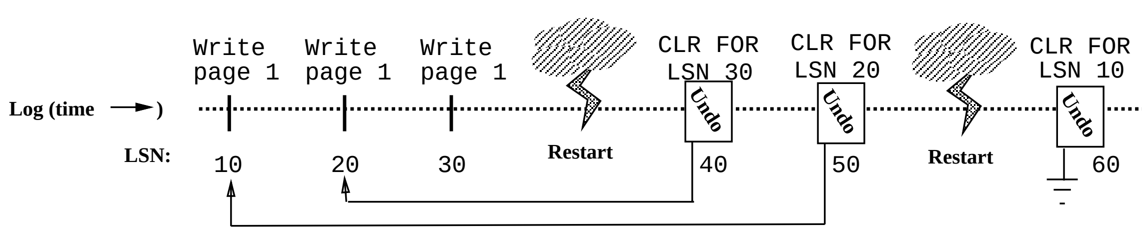

Figure 3: The Use of CLRs for Undo

UndoNxtLSN, in the case of a CLR) is entered as the next LSN to be undone for that transaction. Then the Undo pass moves on to the log record whose LSN is the most recent of the next LSNs to be redone. Undo continues backward in the log until all of the transactions in the list have been undone up to and including their first log record. Undo for *transaction rollback* (for transaction aborts or savepoints) works similarly to the Undo pass of the restart algorithm as described above. The only difference is that during transaction rollback, only a single transaction (or part of a transaction) must be undone. Therefore, rather than keeping a list of LSNs to be undone for multiple transactions, rollback can simply follow the backward chain of log records for the transaction to be rolled back. For transaction abort, rollback continues until the first log record of the transaction has been undone. For savepoints, rollback continues until a specified LSN (the savepoint) is reached.

4. RECOVERY IN ESM-CS

4.1. ARIES and the Page-Server Environment

In this section, we describe the problems that must be addressed when adapting ARIES to a page-server environment and outline the solutions that we implemented. These issues arise from two main architectural features of the page-server environment: 1) the modification of data in client database buffers, while the log and recovery manager are at the server, and 2) the expense of communicating between the clients and the server. The first issue violates several important assumptions of the ARIES algorithm, and thus had to be addressed for correctness of the implementation. The second issue results in performance tradeoffs that are particular to a page-server environment. These tradeoffs had a significant impact on the algorithm design.

The presence of separate buffers on the clients is a fundamental departure from the environment for which the original ARIES algorithm was specified. This difference creates problems with both transaction rollback and system restart. In ARIES, rollback undo is an unconditional operation since it is known that at rollback, the effects of all logged updates appear in the copies of pages either on stable storage or in the server's buffer pool. However, in the page-server environment *the server can have log records for updates for which it does not have the affected database pages*. During rollback, unconditional undo could result in corruption of the database and system crashes due to attempts to undo operations that are not reflected in the server's copy of a page.

This difference in buffering also causes a related problem for system restart. The correctness of the restart algorithm depends on the ability to determine all pages that could have possibly been "dirty" (i.e., different from their copy on stable storage) at the time of a crash. As described in Section 3.2, this information is gathered by starting with the Dirty Page Table that was logged at the most recent checkpoint, and augmenting it based on log records that are encountered during the Analysis pass. In a page-server system, this process is not sufficient, since *there may be pages that are dirty at a client but not at the server*, and hence, do not appear in any checkpoint's Dirty Page Table. This problem, if not addressed, would result in incorrect recovery due to the violation of the repeating history property of Redo.

A problem that arises due to the expense of communication between clients and the server is the inability of clients to efficiently assign LSNs. ARIES expects that LSNs are unique within a log, and that log records are added to the log in monotonically increasing LSN order. In a centralized or shared memory system, this is easily achieved, since a single source for generating LSNs can be cheaply accessed each time a log record is generated. However, in a page-server environment, clients generate log records in parallel, making it difficult for them to efficiently assign unique LSNs that will arrive at the server in monotonically increasing order. Furthermore, if the LSNs are to be physical (e.g., based on log record addresses), then the server would be required to be involved in the generation of LSNs.

To summarize, the issues that must be addressed in a page-server environment are the following:

- The assignment of state identifiers (e.g., LSNs) to place on pages.
- The need to make undo a *conditional* operation.
- Changes to the Analysis pass of system restart to ensure correctness.

These issues and their effects on the algorithm are described in the following subsections. The algorithm is then summarized in Section 4.5.

4.2. Log Record Counters (LRCs) vs. Log Sequence Numbers (LSNs)

As described in Section 3, ARIES requires that each log record be identified by a Log Sequence Number (LSN) and that each page contain a *pageLSN* field which indicates the LSN of the most recent log record applied to that page. These LSNs must be unique and monotonically increasing. It is useful (but not required) that a log record's LSN be the physical address of the record in the log. However, as discussed above, it is not possible to efficiently generate such LSNs in a page-server system. A possible solution to this problem is to allow clients to generate LSNs that are locally (i.e., within each client) unique and monotonically increasing and to *temporarily* place these LSNs in the *pageLSN* field of the affected page before sending the page to the server. Then, either the server would be responsible for converting these local LSNs to global LSNs when the affected pages arrive from the client, or the server could inform the clients of the mapping of its local LSNs to the global ones, making the client responsible for installing the correct LSNs in the data pages before allowing them to migrate back to the server. Such solutions were rejected because of the complication and inefficiency resulting from the tight coupling between the processing (as opposed to the receipt) of log records at the server and the shipping of pages from clients.

In general, the problem with LSNs in a page-server system is that their use is overloaded. They are used in three different ways:

(1) They identify the state of a page with respect to a particular log record.

(2) They identify the state of a page with respect to a position in the log (e.g., an LSN is used to determine the point from which to begin Redo for a page).

(3) They identify where in the log to find a relevant record.

Since clients do not have inexpensive access to the log, they can only be responsible for point 1 above. Therefore, our solution was to separate the functionality of point 1 from the others by introducing the notion of a Log Record Count (LRC). An LRC is a counter that is associated with each page. The LRC *for a particular page* is monotonically increasing and uniquely identifies an operation that has been applied to the page. Instead of storing an LSN on the page, we store the LRC. This field in the page is referred to as the *pageLRC*. In order to map between LRCs and entries in the log, the log record structure is augmented to include an LRC field which indicates the LRC that was placed on the page as a result of the logged operation. Note that for reasons to be explained in the following sections, LRCs have the same size and structure as LSNs (in the current system, this is an eight-byte integer).

LRCs are used in the following way: When a page is modified, the LRC on the page (*pageLRC*) is updated and then copied into the corresponding log record. When the server examines a page to see if a particular update has been applied to the page, the current *pageLRC* is compared to the LRC contained in the log record corresponding to the modification. If the *pageLRC* is *greater than or equal to* the LRC in the log record, then the update is known to be reflected in the page. LRCs have the advantage that, since they are private to a particular page, they can be manipulated at the client without intervention by the server. This removes many of the efficiency and flexibility problems associated with LSNs. There are two main disadvantages of using LRCs however. First, since they are not physical log pointers, they cannot be directly used to serve as an access point into the log. Second, care must be taken to insure that each combination of page id and LRC refers to a unique log record. Our approaches to handling these two problems are addressed in the following sections.

4.3. Conditional Undo

In ESM-CS, log records for operations performed on clients arrive at the server before the dirty pages containing the effects of those operations, thus, when aborting a transaction it is possible to encounter log records for operations whose effects are not reflected in the pages at the server. Obviously, attempting to undo an operation that has not been performed could result in corrupted data. Therefore, we implement undo as a *conditional* operation. When scanning the log backwards during rollback (or restart Undo) the page associated with each log record is examined and undo is performed only for logged operations that had actually been applied.

As described in Section 3.4, undo in ARIES is an unconditional operation — operations are undone without first checking to see if they were ever applied to the page. This is possible in ARIES for two reasons. First, in ARIES all dirty pages are located in the system's buffer pool. Therefore, at the time of rollback, all logged operations are reflected in the pages at the server. Second, history is always repeated during restart Redo. Therefore, it is assured that all of the operations up to the time of the crash are reflected in either the pages on stable storage or in the buffer pool when the Undo pass of restart begins. Note that a later version of ARIES (ARIES/RRH) also uses conditional undo. However, that algorithm uses conditional undo during restart Undo, while in ESM-CS, conditional undo is required for correctness during *rollback* Undo. ARIES/RRH is further discussed in Section 6.

With conditional undo, CLRs must still be written for all undo operations, including those that are not actually performed. However, the *pageLRCs* of the affected pages must not be updated unless the undo operation is actually performed. The reasons for these requirements can be seen in the following example (shown in Figure 4). In the figure, a transaction logged three updates (LRCs 10,11 and 12) for a page, and the page was sent to the server after the first update had been applied but before the others had been applied. When the transaction rolls back, conditional undo results in only LRC 10 being undone. If only the CLR pertaining to that update is written, a problem can arise if the server crashes after logging the CLR but before the page reflecting the undo is written to stable storage (as shown in the figure). Restart Redo repeats history, thereby redoing LRCs 10, 11, 12 and the CLR. The Undo pass encounters the CLR (whose UndoNxtLSN is NIL), and considers the transaction completely undone. This incorrectly leaves the effects of LRCs 11 and 12 on the page. Therefore, rollback must log CLRs for the second and third updates as well, even though the updates were never applied to the page. However, if the *pageLRC* is updated when the fake undo is performed for LRC 12, then rollback would not work properly since when it encounters the log record for LRC 11, it would erroneously infer that the update had been applied to the page (since the *pageLRC* is greater than 11) and would attempt to undo the update, resulting in a corrupted page.

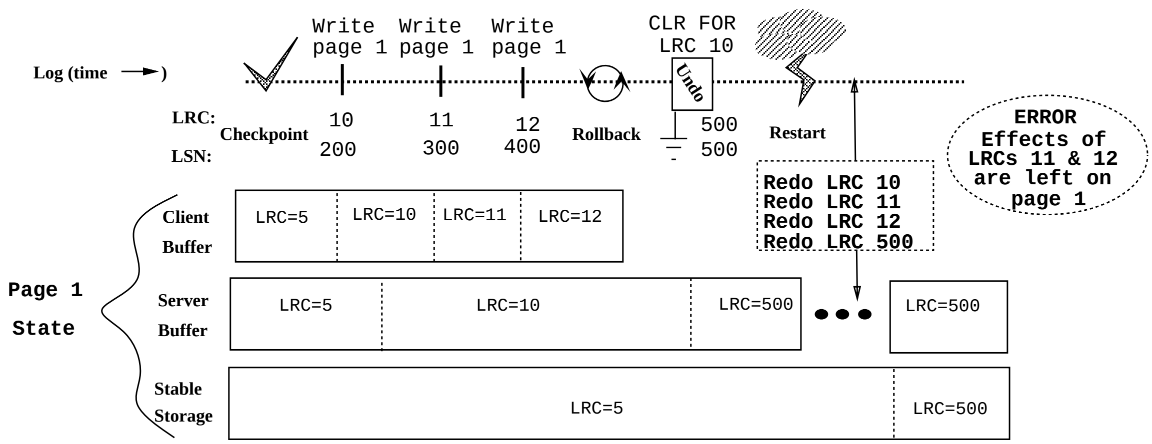

Figure 4: Error Due to Missing CLRs in Conditional Undo

Up to this point, the solution described is to log undo operations, even if they are not performed, but not to update the LRC on the page unless an undo is actually performed on the page. Unfortunately, there is one additional complication that is due to the use of LRCs rather than LSNs. The problem is that in the case where no logged updates to a page are truly undone, the value of the *pageLRC* will still be less than some of the LRCs in the log records of the rolled-back transaction. If this *pageLRC* is simply incremented by updates in subsequent transactions, there will then be values of the *pageLRC* that map to multiple log records. This is a violation of an important invariant and can result in problems in both Redo and Undo.

The above problem could not occur if LSNs were being used, since they are guaranteed to be unique and monotonically increasing, making it impossible to generate a duplicate LSN. This problem is solved by taking advantage of the fact that, while LRCs must be unique and monotonically increasing for a page, they need not be consecutive. The solution requires that the server send the *LSN of the current end-of-log* (i.e., the LSN of the next log record to be written) every time it sends a page to a client. It does this by piggybacking the end-of-log LSN in the message header. When the client receives a data or index page from the server, it initializes the *pageLRC* field of the received page to be the end-of-log LSN that is sent along with the page. Note that the page is not marked dirty by the client when its *pageLRC* is initialized. Therefore, the page will not be written to the server unless an actual update is applied to it.

When a client updates a page, it increments the *pageLRC* on the page. When the server updates a page (e.g., for page formatting, compensation for undo, etc.) it places the LSN of the corresponding log record in the page's *pageLRC* field. The resulting *pageLRC*s are guaranteed to be unique and monotonically increasing (but not necessarily consecutive) with respect to each page for the following reasons:

(1) Page locking insures that a page is modified at only one client at a time during normal operation. The server modifies pages only during restart or rollback, and has exclusive access to the affected pages at that time. Thus, the *pageLRC* cannot be updated in multiple places simultaneously.

(2) Any updates to the page that change the *pageLRC* also generate log records which are flushed to the server before the page is flushed.

(3) Each log record (including CLRs) written increases the server's end-of-log LSN by at least 1.

(4) When the server sends a page to a client, it includes the LSN of the current end-of-log in the message. This LSN is used by the client to initialize the *pageLRC* of the page.

(5) The client increments the *pageLRC* for each log record that it writes.

4.4. Performing Correct Analysis during System Restart

Given the solutions explained in the previous sections, the remaining issue to be addressed is how to insure that the Analysis pass of the system restart algorithm produces the correct information about the state of pages at the time of a crash. There are three related problems to be solved in this regard:

(1) Maintaining *recoveryLSN*s for dirty pages.

(2) Determining which pages may require redo.

(3) Determining the point in the log at which to start the Redo pass.

4.4.1. Maintaining the RecoveryLSN for a Page

During the Analysis pass of the restart algorithm, ARIES computes the LSN of the earliest log record that could require redo. As explained in Section 3.2, this LSN, called the *firstLSN*, is computed by taking the minimum of the *recoveryLSN*s of all of the pages considered dirty at the end of the Analysis pass. In a centralized system, the *recoveryLSN* for each page can be kept by storing the LSN of the update that causes a page to become dirty in the buffer pool control information for that page. Unfortunately, in the page-server environment, clients do not have access to the LSN of an update's corresponding log record when the update is performed (for the reasons described previously).

This problem is solved by having clients attach an *approximate recoveryLSN* to a page when they initially dirty the page. To implement this, we extend the mechanism described in Section 4.3 so that the server piggybacks the LSN of the current end-of-log on *every* reply that it sends to a client. When a client initially dirties a page, it attaches the most recent end-of-log LSN that it received from the server, as the *recoveryLSN* for the page. This LSN is guaranteed to be less than or equal to the LSN of the log record that will eventually be generated for the operation that actually dirties the page. Since the client must communicate with the server in order to initiate a transaction; and since clients must send dirty pages to the server on commit; the approximate *recoveryLSN* will be no earlier than the end-of-log LSN at the time when the transaction which dirties the page was initiated. Typically, it will be more recent than this. When the client returns a dirty page to the server, it sends the approximate *recoveryLSN* for the page in the message along with the page. If the page is not already considered dirty at the server, then it is marked dirty and the approximate *recoveryLSN* is entered in the buffer pool control information for the page at the server.

4.4.2. Determining Which Pages May Require Redo

As described above, a fundamental problem with implementing the ARIES algorithm in the page-server environment is the presence of buffer pools on the clients. One manifestation of this difference is the problem of determining which pages were dirty at the time of a crash. These are the pages for which there may be updates that appear in the log, but not in the copy of the page on stable storage, after the system has crashed. A page is *not* considered dirty by the basic ARIES Analysis algorithm if it satisfies *both* the following criteria:

(1) It does not appear in the Dirty Page Table logged in the most recent complete checkpoint prior to the crash.

(2) No log records for updates to the page appear in the log after that checkpoint.

There are two reasons that a page updated at a client might not appear in the checkpoint's Dirty Page Table. The first is simply that the page was sent back to the server and written to stable storage before the checkpoint was taken. This causes no problems since the page is no longer dirty at this point. The second reason is that the page may have been updated at the client but not sent back to the server prior to the taking of the checkpoint. (Note that even if the page is sent to the server after the checkpoint has been taken, it will be lost during the crash.) In this case, there may have been log records for updates to the page that appeared before the checkpoint. These updates will be skipped by the Redo pass because it will not consider the page to be dirty.

Figure 5 shows an example of this problem. In the figure, a transaction updated a page (page 1) and sent the corresponding log record (LRC 10) to the server without sending the page to the server. After a checkpoint had occurred the client sent the dirtied page (with LRC = 10) to the server followed by a commit request. The server wrote a commit record and forced it to disk, thereby committing the transaction. The server then crashed before page 1 was flushed to disk. In this case, Restart will not redo LRC 10 because according to the ARIES Analysis algorithm, page 1 is not considered dirty (since it neither appears in the most recent checkpoint's Dirty Page Table, nor is referenced by any log records that appear after the checkpoint), and therefore, does not require redo. This would violate the durability of committed updates since the update of LRC 10 would be lost.

Fortunately, the problem of missed dirty pages only has correctness implications for updates of transactions that commit before the system crashes. The reason for this is that the updates of any transactions which had not committed prior to the crash will be undone during the Undo pass of restart. The conditional undo of our algorithm (Section 4.3) can tolerate the absence of the effects of logged updates on a page, providing that all of the missing updates occur later in the log than any updates that were applied to

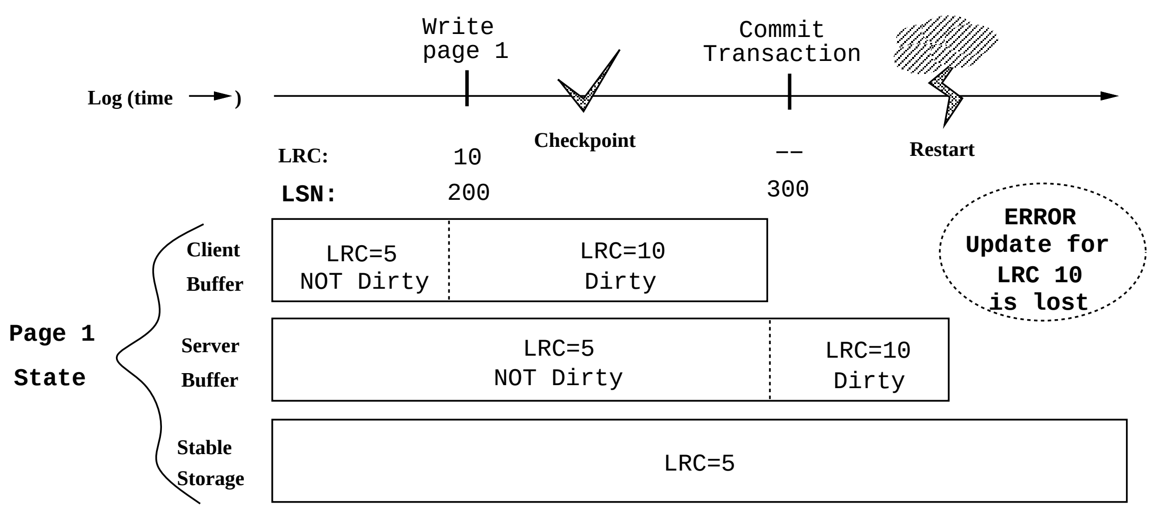

Figure 5: Lost Update Due to Missed Dirty Pages

the page. That condition holds in this case, since the problem arises only when the most recent image of the dirty page was lost during the crash.

Given that the problem of missing dirty pages arises only for committed transactions, we solve the problem by logging dirty page information at transaction commit time. This is done as follows: When a client sends a dirty page to the server, this page and its *recoveryLSN* are added to a list of dirty pages for the transaction. When a page is flushed to stable storage, it is removed from the list. We refer to this list as a *Commit Dirty Page List*. Before logging a commit record for a transaction, the server first logs the contents of the list for the committing transaction. During restart Analysis, when a Commit Dirty Page List is encountered, each page that appears in the list is added (along with its *recoveryLSN*s) to the Dirty Page Table if it does not already have an entry in the table.

An alternative solution we considered was to log the receipt of dirty pages at the server (similar to the logging of buffer operations in [Lind79]), and then during restart Analysis, to add pages encountered in such log records to the dirty page table. While this solution is also a correct one, we felt that the additional log overhead during normal operation could prove to be unacceptable. We also investigated solutions that involved the clients in the checkpointing process. These solutions were rejected because they violate a system design constraint which prohibits the server from depending on clients for any crucial functions.

4.4.3. Determining Where to Begin the Redo Pass

The final problem to be addressed in this section is that of determining the proper point in the log at which to begin Redo. Recall that in ARIES, the LSN at which to begin Redo (called the *firstLSN*) is determined to be the minimum of the *recoveryLSN*s of all of the pages in the Dirty Page Table at the end of the Analysis phase. If a page is not dirty at the time of a checkpoint, then it is known that all updates logged prior to the checkpoint are reflected in the copy of the page that is on stable storage. Therefore, it is safe to begin Redo for the page at the first log record for the page that is encountered during analysis, or anywhere earlier. (Note that it is safe to begin earlier because redo is conditional, so earlier operations will not actually be redone). In the page-server environment, however, this is not the case.

Take for example, the scenario shown in Figure 6. A transaction logged two updates to a particular page (page 1). One log record arrived at the server before a checkpoint, and one arrived after the checkpoint, and the dirty page containing the effects of the updates was not shipped from the client to the server until after the checkpoint. Therefore, page 1 does not appear in the Dirty Page Table recorded in the checkpoint. If the server crashes at the point shown in the figure, then during restart Analysis, when the log record for LRC 11 is encountered, page 1 will be added to the Dirty Page Table with the LSN of that record as its *recoveryLSN* (LSN = 300). Starting Redo for page 1 at this point would result in LRC 11 being redone without LRC 10 having been applied. This could leave page 1 in an inconsistent state.

For pages that are dirtied by a transaction that eventually commits, the Commit Dirty Page List (as described in Section 4.4.2) contains conservative *recoveryLSN*s, which insure that redo will begin at a proper point in the log for such pages. Also, for pages dirtied by a transaction which does not commit, but appear in the Dirty Page Table that is recorded in the most recent checkpoint, the *recoveryLSN* in the Dirty Page Table entry is valid. Therefore, the problem that must be addressed is that of pages pages dirtied by a transaction which does not commit that are added to the Dirty Page Table during the Analysis pass (as shown in Figure 6). To solve this problem, we use a conservative approximation. The solution requires that the Transaction Table structure (described in Section 3.2) that is recorded in each checkpoint

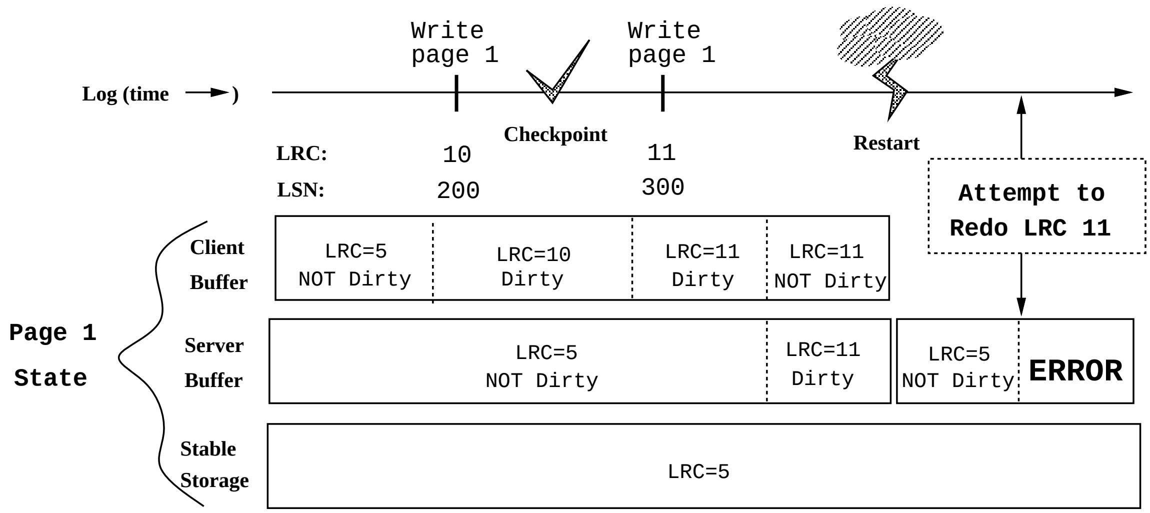

Figure 6: Inconsistent Redo Due to Missed Log Record

to be augmented to include a field for the first LSN generated by each transaction (called the *startLSN*). Then, during Analysis, when a page is added to the Dirty Page Table, it is marked as a newly added page and tagged with the transaction Id of the transaction which caused it to be considered dirty. At the end of Analysis, entries for pages that were added to the dirty page table due to an update by an uncommitted transaction have their *recoveryLSN* replaced by that transaction's *startLSN*. This conservative approximation results in correct behavior, but it may cause extra I/O during Redo because pages may have to be read from stable storage to determine whether or not a logged update must be redone. However, the number of pages for which this conservative approximation is required is kept small by taking checkpoints. Recall that in ARIES, checkpoints are fairly inexpensive since they require no data pages to be forced and no quiescing of operations.

4.5. Summary of the Algorithm

While the preceding discussion was fairly detailed, the resulting algorithm requires only the following changes to the ARIES algorithm:

During Normal Operation:

- When a transaction is added to the Transaction Table, the LSN of the first log record generated by the transaction is entered as the transaction's *startLSN*.
- Each client keeps an estimate of the current end-of-log LSN. This estimate is updated upon receipt of every message from the server.
- When a data or index page arrives at the client, the *pageLRC* of the page is initialized to be the estimated end-of-log LSN. The page is *not* marked dirty as a result of this initialization.
- When a client performs an update on a page, it increments the *pageLRC* on the page and places the new *pageLRC* value in the log record. If this update causes the page to be marked "dirty", then the current estimated end-of-log LSN is entered as the *recoveryLSN* in the page's buffer control information at the client.
- When the server performs an update on a page, it places the LSN of the log record it generates as the *pageLRC* on the page and in the log record. If this update causes the page to be marked

"dirty", then the LSN is also entered as the *recoveryLSN* in the page's buffer control information at the server.

- When a client sends a dirty page to the server it includes the page's *recoveryLSN* in the message.
- When the server receives a dirty page from a client, the page is added to a list of dirty pages for the transaction which dirtied it. If the transaction commits, this list is logged as the Commit Dirty Page List for the transaction.

During Restart Analysis:

- When a transaction is added to the Transaction Table as the result of encountering a log record, the LSN of the log record is entered as the transaction's *startLSN*.
- When a commit log record is encountered, pages that appear in the Commit Dirty Page List are added to the Dirty Page Table. The *recoveryLSN* in the Dirty Page Table entry for each page is set to the minimum of the *recoveryLSN* for the page in the Dirty Page Table (if any) and that in the Commit Dirty Page List.
- At the end of Analysis, all pages that were added to the Dirty Page Table by the Analysis pass due to log records generated by non-committing transactions are given a conservative *recoveryLSN*. This *recoveryLSN* is the *startLSN* of the transaction that caused the page to be considered dirty.

During Restart Redo:

- This pass of the restart algorithm is unchanged except for the use of LRCs for comparisons between log records and pages rather than LSNs.

During Undo (for restart or rollback):

- To undo a log record, the LRC stored in the record is compared to the *pageLRC* of the affected page. If the log record LRC is *greater than or equal to* the *pageLRC* then an actual undo is performed, otherwise a "fake" undo is performed.
- Actual undo is performed by logging a CLR for the undone operation, performing the undo on the page, and placing the LSN of the CLR in the *pageLRC* of the affected page.
- Fake undo is performed simply by logging a CLR for the undone operation. The page itself is not modified and is not marked as dirty. The pageLRC of the page is unaffected by a fake undo.

5. PERFORMANCE

In this section we describe an initial study of the performance of the logging and recovery subsystems of ESM-CS. We first investigate the overhead of logging for several different transaction types and then examine the relative costs of the various components of logging. We also discuss how these costs can be reduced. Finally, we present performance measurements for transaction rollback and restart recovery.

The performance experiments described in this section were run on two SPARCstation ELCs, each with 24MB of memory, running version 4.1.1 of the SunOS operating system. The client and server processes were run on separate machines. The two machines were located on their own isolated Ethernet network. The client and server software was compiled with optimization. The log and database were stored on separate disks using raw disk partitions to remove any operating system buffering. The log page size was 8KB and database page size was 4KB. All times were obtained using the *gettimeofday()* and *getrusage()* system calls and are reported in seconds. The page fault, swapping, and other operating system events recorded by *getrusage()* were checked to ensure that such activity was minimal and did not distort the performance results.

5.1. Logging Experiments

In the first set of experiments we investigated the overhead imposed on transactions by the logging subsystem during normal operation. Three different databases were used for the experiments and are described in Table 1. All three databases initially contain 2MB of data on pages that are approximately 50% full so, each database consists of 4MB of physical space. We describe the results for two different types of transactions applied to the three databases. The first transaction type, *Write*, sequentially scans the database and writes (updates) half of the bytes in each object, updating a total of 1MB of data. The second transaction type, *Insert*, sequentially scans the database and inserts new data at the beginning of each object to increase its size by 50%, resulting in the insertion of 1MB of new data. *Insert* does not increase the number of pages in the database since each page has enough free space to accommodate the inserted data. [1] *Write* transactions log both the old and new data, while *Insert* transactions log only the inserted data.

Table 2 shows the results from running the five experiments with and without logging. In these experiments the server buffer pool was 5 Mbytes and the entire database was cached in the server's buffer pool before each experiment was run. The client buffer pool is also 5 Mbytes so that the entire database fits in the client buffer pool during a transaction. However, as described in Section 2.1, inter-transaction caching is not currently supported, so the client buffer pool is empty at the beginning of each transaction. The large buffer pools were used to in order to help us isolate the effects of logging by removing sources of variability (e.g., other disk I/O) and by making logging a more significant part of the total work performed in the tests. The write-intensiveness of the transactions also accentuates the impact of logging on normal operation. For these reasons, the overhead of logging reflected in Table 2 is much higher than

Database Name	Objects in database	Object size (bytes)	Objects per page	Pages in database
FewLarge	1,000	2,000	1	1,000
SomeMedium	10,000	200	10	1,000
ManySmall	100,000	20	100	1,000

Table 1: Description of Experimental Databases

Experiment name	Number of operations	Operation size (bytes)	Execution Time (sec)		Logging overhead
			Logging on	Logging off	
Write_FewLarge	1,000	1,000	17.37	13.55	3.82 (28%)
Write_SomeMedium	10,000	100	18.43	14.46	3.97 (27%)
Write_ManySmall	100,000	10	32.32	21.36	10.96 (51%)
Insert_FewLarge	1,000	1,000	14.29	12.49	1.80 (14%)
Insert_SomeMedium	10,000	100	15.74	13.05	2.69 (21%)

Table 2: Results for Logging Experiments with Data Cached at the Server

[1] We do not show the results from running *Insert* on the ManySmall database because the overhead for object headers does not allow inserts to be applied to all of the objects on a page without having to forward some objects to other pages for that database.

would be expected in an actual application. The numbers presented in Table 2 were obtained by running each transaction five times and taking the average of the last four runs. For each experiment, we report the total execution time of the transaction both with logging turned on and with logging turned off. This total comprises the time to initiate, execute, and terminate (commit) a transaction, including the time to send all dirty pages to the server (approximately 4 seconds in all cases). The final column shows the execution time overhead incurred due to logging.

As shown in Table 2, the overhead of logging varied substantially for the different transactions. For both transaction types, the overhead of logging increased with the number of operations for which log records were generated even though the amount of actual data that was updated remained constant. This increase was due to the size overhead added for each log record. In ESM-CS, this overhead is 64 bytes — 56 bytes for the record header and 8 bytes for the operation information. As a result of this overhead, the number of log pages generated and written increased considerably when a larger number of smaller operations were performed per transaction. For example, the 1,000 operations of the Write_FewLarge experiment generated 2.7MB of log records in 337 log pages, while the 100,000 operations of the Write_ManySmall experiment generated 8.9MB of log records in 1,090 log pages. Comparing the two transaction types, the logging time overhead of the *Insert* tests was less than that of the *Write* tests; the time overhead of Insert_FewLarge was only 14% compared to 28% for Write_FewLarge. This difference is because *Insert* logs only the inserted data, while *Write* logs both the before *and* after images, resulting in a larger volume of logged data for *Write*.

In order to better understand the cost of logging in ESM-CS, we analyzed the cost of the different components of logging as shown in Table 3. The logging time is broken down into three parts: 1) generating the log records at the client, 2) shipping pages of log records from the client to the server, and 3) writing log pages from the server's buffer to the log disk. To obtain this breakdown we altered ESM-CS to allow these three logging components to be selectively turned on and off. The log record generation time was calculated by computing the difference between running the transactions with only log generation turned on and running them with all of logging turned off. Because the shipping and writing of log pages can occur in parallel with other client and server activity, these costs were measured in two ways. The first way was to separately measure the actual time it took to ship or write a certain number of pages. The columns labeled *actual* contain the results of these measurements. The second way was to selectively turn off the shipping and writing of log pages and compute the differences in time observed by the client. These times are shown in the columns labeled *observed* and reflect the previously mentioned

Experiment name	Generate records	Ship log pages		Write log pages		Total overhead	
		observed	actual	observed	actual	observed	actual
Write_FewLarge	0.48	2.45	2.57	0.78	6.18	3.71	9.23
Write_SomeMedium	0.92	2.03	2.57	0.87	6.18	3.82	9.67
Write_ManySmall	5.19	4.24	8.37	1.35	20.36	10.78	33.92
Insert_FewLarge	0.31	1.05	1.10	0.26	3.03	1.62	4.44
Insert_SomeMedium	0.89	1.23	1.57	0.44	3.75	2.56	6.21

Table 3: Logging Cost Breakdown (seconds)

parallelism. The results in Table 3 are shown graphically in Figure 7. In the figure, the area in white represents the time to perform the transaction without any logging. As would be expected, when considering actual costs, the most significant overhead factor was the writing of log records to disk. Shipping the log pages to the server took about 41% of the time it took to write the pages out to disk. The cost of generating the log records was small in the FewLarge cases but became more significant in the transactions that generated more log records. This occurred because the number of log records generated increased at a faster rate than the number of log pages.

From the client's point of view, the observed cost of shipping was more significant than the writing cost since most of the writing was performed in parallel with other client and server activity. In principle, the shipping of log pages can also be performed in parallel with other activity, but since the compute time of transactions was small in these tests, the network was kept busy by client data page and lock requests. Therefore, the the shipping of log pages to the server had to compete for the network with the shipping of data pages to the client. One exception to this was the Write_ManySmall case. In this case, there was significant compute time to due to the generation of log records, and thus some of the log page shipping was performed in parallel with the generation of log records.

Although comparable published performance results for logging systems are difficult to find, the results from these write-intensive experiments lead us to conclude that the performance of our initial logging implementation is reasonable. The numbers also indicate two possible areas for improvement. First, it is clear from the results that reducing the amount of logged information can result in significant performance improvements, especially for small updates. The current log record overhead size of 64 bytes is slightly larger than the typical log record header size of approximately 50 bytes [Gray91, GR92]. With sufficient coding effort, the ESM-CS log record overhead could be reduced to 56 bytes (but not much smaller). A different approach would be to reduce the number of log records generated in special cases like Write_ManySmall (where most or all of the objects on a page are updated) by logging entire pages. Secondly, it is clear that by performing shipping and writing of log pages in parallel with other activity, the observed cost for logging can be reduced considerably, even for write-intensive transactions. Algorithms to further exploit such parallelism must be investigated.

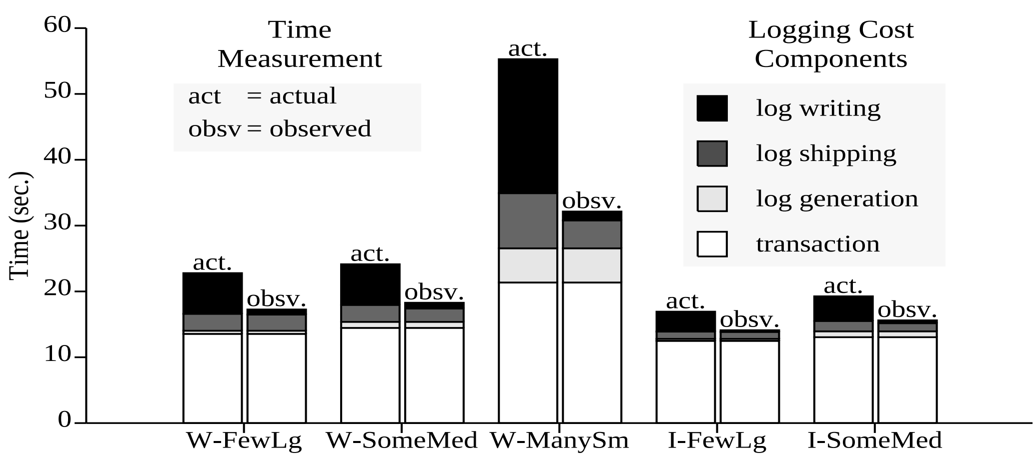

Figure 7: Actual and Observed Logging Costs

5.2. Transaction Rollback and Recovery Performance

In addition to the logging experiments described previously, we also ran some simple experiments to gain insight into the performance of rollback and recovery. These experiments used the databases and *Write* transactions described in the previous section. Table 4 shows the results of these experiments and for comparison purposes also shows the execution times of the transactions with logging turned on (from Table 2). To measure the cost of transaction rollback, we aborted each transaction after all the dirty pages and log records had been shipped back to the server. In this experiment, rollback did not perform any I/O for data pages since the database was cached in the server buffer. For restart recovery we show the time required by the Analysis and Redo phases when the server was forced to crash immediately after the transaction committed. Since the server buffer was large enough to store the entire database, no data pages had been written to stable storage prior to the crash, and thus, all data pages had to be reread from disk during recovery.

The transaction rollback results were primarily determined by the time to read the log, to generate compensation log records, and to write those log records to disk. The cost of actually performing the undo operations was only several seconds in the longest case. Compensation log records for write operations only require that redo information be logged since CLRs are never undone. Therefore, CLRs for writes contain only half as much operation information as normal write log records. However, the fine granularity of the updates in the Write_ManySmall case results in much of the log space being used for log record headers. Therefore, while undoing the Write_FewLarge case generated about half as many log pages as the original transaction, undoing the Write_ManySmall case required almost as much log space as the original transaction. The generation of compensation log records also results in significant log disk arm movement, as these new records must be appended to the log while rollback is trying to scan the log backwards. This disk arm movement is especially expensive in the Write_ManySmall case since the amount of compensation log space generated is much greater. Excessive disk arm movement is a problem particularly for long-running write-intensive transactions. Since ESM-CS is designed to support object-oriented database systems where this type of transaction is likely to be encountered, we plan to try to reduce this disk arm movement by batching the newly written log pages and writing them out in groups.

The restart tests showed a significant increase in the cost of the Analysis and Redo phases as the volume of log data increased. Note that no checkpoints were taken during these tests, so the Analysis phase scanned the entire log but did not access any data pages. The Analysis times can be improved by taking more frequent checkpoints. The Redo phase scanned the log and also read the data pages from stable storage. The cost of actually performing the redo operations was small. A possible way to reduce

Experiment name	Execution time	Rollback time	Analysis time	Redo time
Write_FewLarge	17.37	13.24	2.06	5.65
Write_SomeMedium	18.43	15.86	2.07	6.26
Write_ManySmall	32.32	62.86	8.17	15.32

Table 4: Rollback and Recovery Times (seconds)

the cost of restart would be to use Most Recently Used (MRU) buffering (instead of LRU) for the log pages during the Analysis pass, as Redo scans the log in the same direction as Analysis. Also, we believe that all phases of restart could be improved by prefetching log pages and the pages in the Dirty Page Table during restart. Still, while there are improvements that can be made, the transaction rollback and system restart performance of the current implementation seem to be acceptable.

6. RELATED WORK

In this section, we describe related work including: extensions to the ARIES algorithm, recovery algorithms that have been proposed for shared-disk architectures, recovery in distributed transaction facilities, and recovery in other page-server and object-server systems.

6.1. ARIES/RRH Extensions

The recent ARIES/RRH (Restricted Repeating of History) algorithm [MP91] relaxes the ARIES requirement that history be repeated for all transactions during restart Redo. Under certain conditions explained in [MP91], updates for transactions that were in progress at the time of a crash need not be redone during restart. Restricted repeating of history requires the notion of conditional undo. When ARIES/RRH writes a CLR for an undo operation, it includes the LSN of the record being undone, called the *undoneLSN*, in the CLR. During a subsequent Redo pass (e.g., in the event of a later crash), CLRs are redone only if the the *undoneLSN* stored in the CLR is less than or equal to the *pageLSN* of the affected page. In ARIES/RRH (as in ESM-CS), CLRs are written for all undo operations regardless of whether or not they are actually applied. However, unlike in ESM-CS, where fake CLRs are needed for correctness, fake CLRs are used in ARIES/RRH only to simplify media recovery.

The differences between ARIES/RRH and ESM-CS conditional undo result from the fact that ARIES/RRH was designed to enhance the performance of ARIES during restart, while ESM-CS conditional undo was developed in order to correctly implement transaction rollback in a page-server system. Thus, while conditional undo is an option in ARIES, it is a requirement in ESM-CS. However, since ARIES/RRH can avoid redoing some work during Redo, ESM-CS could benefit by including full RRH support; this extension would require only slight modifications to the existing system.

6.2. Recovery in Shared-Disk Systems

In a shared-disk system [Bhid88] there are multiple processing nodes, each with its own memory, that share a common pool of disks and communicate using messages. This environment has some similarities with the page-server environment. These include: 1) multiple processing nodes with no shared memory, 2) data operations for a transaction being performed on a single processing node, and 3) the database not being partitioned with respect to the various processing nodes. However, there are also many important differences between the environments. These include:

(1) The page-server is an *asymmetric* environment. The capabilities of the clients can be very different than the capabilities of the server. For example, clients are often diskless workstations.

(2) Clients are inherently unreliable. They tend to be in individual offices (or homes) rather than in a machine room. Also, they are often used concurrently for activities other than database access.

(3) There is no synchronized clock that is cheaply available to all of the processors in the system.

(4) Clients and servers are connected over slower links than the nodes in a shared disk system.

(5) The server is a potential bottleneck since it is responsible for providing database access, locking, and logging services to many clients.

6.2.1. ARIES Shared Disk Extensions

ARIES has been extended to the shared disk environment as described in two recent papers. [MNP90] addresses the problems of migrating a single-site database system to the shared disk environment. The problem relevant to our work is the lack of monotonically increasing LSNs due to the use of a separate log for each node in the system. The given solution is to store Update Sequence Numbers (USNs) on pages, rather than LSNs. USNs are initialized based on a clock value at the time the page is formatted, requiring that the clocks be synchronized to within an acceptable (implementation-dependent) limit. Other aspects of the solution depend on the fact that each system has its own log, and can thus maintain information about physical LSNs in the buffer pool for use during recovery. We used LRCs to solve a similar problem in ESM-CS, but due to the lack of synchronized clocks and local logs, our solution uses the estimated end-of-log LSN and approximate *recoveryLSNs* as described in Section 4.

In [MP91] protocols are discussed for transferring pages between nodes in a shared-disk system where each node has its own log. Two of the protocols transfer a page between nodes without writing the page to disk; these protocols are subject to recovery issues similar to those that arise in ESM-CS, as a node can have log records for a page that is not dirty at that node. In these protocols, a page cannot be dirtied unless a lock exists for it in a Global Lock Manager (GLM). To implement recovery, the GLM entries are extended with LSN information, such as the *recoveryLSN* for the page. In fact, the *recoveryLSN* used is an approximation based on the end-of-log LSN for the node requesting the lock.

There are two serious disadvantages to implementing a similar solution in a page-server system. First, the use of the GLM to store dirty page information would negate many of the performance benefits of using coarse-grained (e.g., file-level) locking. Coarse-grained locking is provided as an option in ESM-CS and is used for performance enhancement in other systems such as the VAXcluster version of Rdb/VMS [Josh91]. Second, this solution would preclude the use of non-centralized locking algorithms in the page-server environment such as those described in [CFLS91, WR91]. It was shown in these papers that the overhead of centralized locking in the page-server environment can have a major performance impact under many workloads. Also described in [MP91] are locking protocols that allow fine-grained (e.g. record-level) locking in a data-sharing system; similar ideas could be used to allow fine-grained locking in ESM-CS.

6.2.2. Other Shared Disk Algorithms

[Lome90] describes an algorithm for recovery in systems with multiple logs. This algorithm stores the previous "State Identifier" of an updated page (similar to a *pageLSN* or *pageLRC*) in the log record for the update, thereby allowing logs to be easily merged during redo. Since the algorithm does not require synchronized clocks, it may prove useful in a client-server environment in which clients perform their own logging. As described in Section 2.1, we chose not to implement client logging for several reasons, including the unreliability of clients compared to the server and the expense of extra client disks. [Lome90] acknowledges the reliability problem and suggests possible approaches towards addressing it. Another algorithm for recovery in shared-disk systems with multiple logs is presented in [Rahm91]. This algorithm is defined for a NO STEAL buffer management policy and thus uses logging only for Redo.

The algorithm differs from the ones described previously in that it assigns responsibility for recovery of certain partitions of the database to particular systems. The algorithm depends on the individual logs being available from other systems and may require substantial communication (which would be quite expensive in a client-server system) to perform Redo for a failed node.

6.3. Recovery in Distributed Transaction Facilities

Recovery is also a concern in systems which provide general transaction support in a network computing environment. These systems include the distributed logging facility of CMU's Camelot [DST87] and IBM Almaden's QuickSilver system [HMSC88]. Both are intended to run in a hardware environment similar to the one for which ESM-CS was designed. However, both of these systems have fundamentally different software architectures than ESM-CS. These systems attempt to shield clients from the details of recovery by making servers primarily responsible for logging and recovery. ESM-CS on the other hand, takes the approach distributing much of the logging work to the clients that perform the data operations. This approach is consistent with the page-server architecture used by ESM-CS, which attempts to exploit the resources of clients in order to offload work from the server. Also, the use of the STEAL/NO FORCE buffer management policy in ESM-CS allows for efficient use of the client and server memory resources and the server's disk resources. The support of STEAL/NO FORCE is enabled by the techniques that were described in Section 4.

6.4. Recovery in Page-Server and Object-Server Architectures

As stated earlier, much has been written about systems which implement page-server and object-server architectures, but few details about recovery in those systems have been provided. This is due in part to the fact that many of the systems are commercial systems with proprietary implementations. The O2 system [Deux91] employs an ARIES-based approach that uses shadowing in order to avoid undo. The ORION-1SX system (an object-server version of ORION) [KGBW90] uses a FORCE policy and therefore keeps only an undo log. We are unaware of any systems which have implemented the STEAL/NO FORCE policy for a page-server (or object-server) system.

7. CONCLUSIONS

In this paper, we have described the problems that arise when implementing recovery in a page-server environment, and have presented a recovery method that addresses these problems. The recovery method was designed with the goal of minimizing the impact of recovery-related overhead during normal processing, while still providing reasonable rollback and system restart times. In particular, the method supports efficient buffer management policies, allows flexibility in the interaction between clients and the server, and allows clients to off-load the server by performing much of the work involved in generating log records. We described the implementation of the method in ESM-CS, and presented measurements of the impact of recovery on normal operation and the performance of rollback and system restart. We also compared our work to other related systems and algorithm proposals.

The measurements obtained so far appear promising. Overhead for many cases was reasonable and the study raised issues to be addressed in order to improve the performance of the system. These issues include: reducing log record size, batching writes to the log disk, prefetching from the log during recovery, and exploiting additional parallelism between logging operations on the server and other operations on the client during normal processing. Additional studies of realistic workloads will be required in order to obtain a better understanding of the performance impact of the logging and recovery subsystems. In addition, we plan to extend the recovery system to include media recovery, restricted repeating of history during Redo, and support for inter-transaction caching. Finally, this work has raised a number of interesting possibilities for alternative recovery system designs, and we plan to investigate the performance tradeoffs among these alternatives.

ACKNOWLEDGEMENTS

We would like to thank C. Mohan for a number of informative discussions regarding ARIES and our algorithm, and for suggesting improvements that made our implementation much simpler. Dave Haight did much of the initial work of converting the original EXODUS storage manager from a single-site to a client-server system. Nancy Hall and Zack Xu helped build the new version of the system. Praveen Seshadri provided helpful comments on an earlier draft of this paper.

REFERENCES

[BHG87] Bernstein, P., Hadzilacos, V., and Goodman, N., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[Bhid88] Bhide, A., "An Analysis of Three Transaction Processing Architectures", *Proc. 14th VLDB Conf.*, Los Angeles, Aug. 1988.

[Catt91] Cattell, R., "An Engineering Database Benchmark", *The Benchmark Handbook: For Database and Transaction Processing Systems*, Gray, J., ed., Morgan Kaufmann, 1991.

[CDRS89] Carey, M., DeWitt, D., Richardson, J., Shekita, E., "Storage Management for Objects in EXODUS," in *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. Lochovsky, eds., Addison-Wesley, 1989.

[CFLS91] Carey, M., Franklin, M., Livny, M., and Shekita, E., "Data Caching Tradeoffs in Client-Server DBMS Architectures", *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Denver, June 1991.

[Comm90] The Committee for Advanced DBMS Function, "Third Generation Data Base System Manifesto", *SIGMOD Record*, Vol. 19, No. 3, Sept. 1990.

[DFMV90] DeWitt, D., Futtersack, P., Maier, D., Velez, F., "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems," *Proc. 16th VLDB Conf.*, Brisbane, Australia, Aug. 1990.

[DST87] Daniels, D., Spector, A., Thompson, D., "Distributed Logging for Transaction Processing", *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, San Francisco, May, 1987.

[Deux91] Deux, O., *et al.*, "The O2 System", *CACM*, Vol. 34, No. 10, Oct. 1991.

[Exod91a] EXODUS Project Group, *EXODUS Storage Manager Architectural Overview*, EXODUS Project Document, University of Wisconsin - Madison, Nov. 1991.

[Exod91b] EXODUS Project Group, *Using the EXODUS Storage Manager V2.0*, EXODUS Project Document, University of Wisconsin - Madison, Nov. 1991.

[Gray78] Gray, J., "Notes on Data Base Operating Systems", *Operating Systems - An advanced Course*, R. Bayer, R.M. Graham, G. Seegmuller, eds. Springer-Verlag, N.Y., 1978.

[Gray81] Gray, J., *et al.*, "The Recovery Manager of the System R Database Manager", *ACM Computing Surveys*, Vol. 13, No. 2, June, 1981.

[Gray91] Gray, J., *personal communication*, December 1991.

[GR92] Gray, J., Reuter, A., *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Mateo, CA, to appear 1992.

[HMSC88] Haskin, R., Malachi, Y., Sawdon, W., Chan, G., "Recovery Management in QuickSilver", *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February, 1988.

[HR83] Haerder, T., Reuter, A., "Principles of Transaction Oriented Database Recovery - A Taxonomy", *Computing Surveys*, Vol. 15, No. 4, December, 1983.

[Josh91] Joshi, A., "Adaptive Locking Strategies in a Multi-Node Data Sharing Environment", *Proc. 17th Int'l Conference on Very Large Data Bases*, Barcelona, September, 1991.

[KGBW90] Kim, W., Garza, J., Ballou, N., Woelk, D., "Architecture of the ORION Next-Generation Database System", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, March, 1990.

[Lind79] Lindsay, B. *et al*, "Notes on Distributed Databases, *IBM Research Report RJ2571*, San Jose, July 1979.

[LLOW91] Lamb, C., Landis, G., Orenstein, J. Weinreb, D., "The ObjectStore Database System", *CACM*, Vol. 34, No. 10, Oct. 1991.

[Lome90] Lomet, D., "Recovery for Shared Disk Systems Using Multiple Redo Logs", *Technical Report CRL 90/4*, DEC Cambridge Research Lab, Cambridge, MA, Oct. 1990.

[Moha90] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P., "ARIES: A Transaction Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", *IBM Research Report RJ6649*, *IBM Almaden Research Center*, November, 1990, to appear in ACM Transactions on Database Systems.

[MN91] Mohan, C., Narang, I., "Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment", *Proc. 17th Int'l Conference on Very Large Data Bases*, Barcelona, September, 1991.

[MNP90] 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.

[MP91] Mohan, C., Pirahesh, H., "ARIES-RRH: Restricted Repeating of History in the ARIES Recovery Method", *Proc. 7th Int'l Conference on Data Engineering*, Kobe, April 1991.

[Rahm91] Rahm, E. , "Recovery Concepts for Data Sharing Systems", *Proc. 21st Int'l Symposium of Fault-Tolerant Computing*, Montreal, June, 1991.

[RC89] Richardson, J., Carey, M., "Persistence in the E Language: Issues and Implementation", *Software Practice and Experience*, Vol. 19, Dec. 1989.

[RCS91] Richardson, J., Carey, M., and Schuh, D., *The Design of the E Programming Language*, submitted for publication.

[Ston90] Stonebraker, M., "Architecture of Future Data Base Systems", *Data Engineering*, Vol. 13, No. 4., Dec. 1990.

[WR91] Wang, Y., Rowe, L., "Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture", *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Denver, June 1991.