LU Logging - An Efficient Transaction Recovery Method

Sreenivas Gukal
Edward Omiecinski
Umakishore Ramachandran

GIT–CC–93/21

July 20, 1993

Abstract

In this paper, we present *LU-Logging*, an efficient transaction recovery method. The method is based on *flexible-redo/minimal-undo* algorithm. The paper describes an implementation which avoids the overheads of deferred updating used in previous no-undo implementations. An update by a transaction to a data record does not immediately update the data record. Instead, it generates a redo log record and associates it with the data page. Each page in the data base has an associated log page, which contains the still-uncommitted log records of the updates for the data page. The log page is read from and written to the disk along with the corresponding data page. This gives the flexibility of applying the redo log records any time after the transaction commits, in particular when the data page is read by another transaction. We call this updating as lazy. For aborted transactions the redo log records are just discarded. Simulation studies show that the overhead during normal transaction processing for *LU-Logging* is comparable to that of traditional *Logging*. The crash recovery time is shown to be an order of magnitude faster than that for traditional *Logging*.

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

1 Introduction

Most of the present data base management systems support the notion of transactions. The transaction mechanism guarantees both view and failure atomicities. To support the notion of transactions, the database management system should have a recovery mechanism to keep track of changes being made to the state of the database. Over the years, several recovery mechanisms have been proposed, analyzed and implemented.

A recovery mechanism has two objectives: low overhead during normal processing and fast recovery after system crash. The overhead during normal processing should be low, since even a small amount of extra work here adds up over time. On the other hand, system crashes do not occur frequently (typically, on the order of twice a week). Usually, new transactions are allowed after the system is brought up after failure, even before the recovery is completed. Methods like [GMLB 81, Crus 84, MHLPS 92] partially recover the state of the database first, and then, while continuing with the recovery, allow normal processing on that part of the database which has been recovered. Some others use a *hot standby* [Borr 84] or non-volatile memory [CKKS 89] for increased data availability for new transactions during crash recovery. Hence the performance of any recovery mechanism, during crash recovery, is based on how fast the system can be brought up and how much of the database is made available subsequent to a crash for normal transaction processing.

A recovery algorithm needs to perform "redo" if it permits a transaction to commit before all its updates are reflected on the stable database. On the other hand, if uncommitted updates are allowed in the stable database, the recovery algorithm may have to perform "undo". *Redo/undo* (e.g. traditional Logging) and *redo/no-undo* (e.g. Shadowing) are two widely used recovery algorithms [BeHG 87]. *Redo/undo* has low overheads during normal transaction processing, but takes a long time to process failures during crash recovery. On the other hand, *redo/no-undo* has higher overheads for normal processing, and considerably lower overheads for crash recovery due to no-undo, compared to *redo/undo* algorithm. This work proposes a new recovery method and an implementation, which is based on *flexible-redo/ minimal-undo* algorithm. The method avoids the high transaction processing overheads using flexible-redo, and maintains low overheads for crash recovery due to minimal-undo. The implementation is done in the context of a multiprogrammed uniprocessor data base system, that supports concurrent transactions.

Several implementations have tried the idea of making use of no-undo property in atleast a restricted form in implementing *redo/undo* algorithm. The no-undo helps in limiting the amount of work to be undone during crash recovery. In [Ong 84, Curt 88] two log files are maintained. The first one, called the logical log file, records logical changes in the form of undo and redo logs. The second one, called the physical log file, is used to log the before images of the objects to be updated. Periodically the data base is made physically consistent by a "consistent point" event. Crash recovery consists of first applying the physical log to roll back the state of the data base to the physically consistent state at the last "consistent point". Thus all transactions that started after the "consistent point" and aborted need not be undone. The logical log now is used

to redo committed transactions and to undo aborted ones (which started before the "consistent point"), bringing the database to a logically consistent state. System R [GMLB 81] periodically saves an action-consistent state of the data file. An incremental log is used for the undo and redo log records to save changes to shared files. Crash recovery starts with the action-consistent copy of the data file and applies the redo log records to restore the lost updates. Here again, only the aborted transactions, which started before the checkpoint, need to be undone. System R does action-consistent backups at the **file level**. [Reut 80] uses transaction-consistent backups, called shadow pages, at the **page level**. In this method, the shadows and the modified data pages are stored adjacent to each other on the disk. Locking is done at the page level. Each page-read brings in both the shadow page and the modified page. Only the modified page needs to be written back. A modified page becomes a shadow page after the transaction that modified it commits. Here no undos are ever necessary. There have been no performance studies reported for the above methods.

LU-Logging keeps transaction-consistent backups at the **record level**. An update to a data item by a transaction generates a redo log record without actually updating the data item. The redo log record is associated with the page of the data item. When the data page is forced from main memory to disk, all its uncommitted redo log records are copied into a log page and written to the disk along with the data page. Keeping the redo log records with the data page gives the flexibility of applying the updates to the data items any time after the transaction commits. We call this updating as "lazy" (LU stands for **L**azy **U**pdates). To abort a transaction, only its redo log records in the log pages need to be discarded. Thus undoing updates requires minimal processing. This algorithm can be termed as *flexible-redo/minimal-undo*.

The rest of the paper is organized as follows. Section 2 briefly describes, compares and contrasts *Logging* and *Shadowing* recovery methods as examples of *redo/undo* and *redo/no-undo* algorithms. Section 3 explains in detail the *LU-Logging* recovery method and its implementation. Sections 4 and 5 present the transaction processing and crash recovery performance of *LU-Logging* as compared to ARIES [MHLPS 92], a well-known implementation of *redo/undo* algorithm. Media recovery is discussed in section 6. The paper ends with conclusions and suggestions for future work.

2 Logging & Shadowing

The traditional *Logging* mechanism [Gray 78, HaRe 83, AgDe 85] is an excellent example for the *redo/undo* algorithm. Here, every update operation, besides updating the data record, also creates an "undo" and a "redo" log record. These log records are written to an append-only log on the disk. A write-ahead-log protocol is used to ensure proper recovery. According to this protocol, before a page containing uncommitted updates is forced to the disk, its undo log records must be written to the log on the disk. Also, before a transaction is committed all of its redo log records should be forced to the disk. When a transaction aborts or rolls back, the undo log records are used to restore the previous state of the modified records. Checkpointing is done occasionally to reduce the amount of work during recovery. A checkpoint record consists of information

like the dirty pages list, the point in the log on disk to start the crash recovery from, etc. The actions during checkpointing and the contents of the checkpoint record depend on the particular implementation. The crash recovery algorithm consists of using the log on the disk to un-do the effects of aborted transactions and to re-do the lost updates of committed transactions.

The *Shadowing* method [Lori 77, AgDe 85] is based on the redo/no-undo algorithm. Here, when a transaction wants to modify a page, a copy of the page is made and the modifications are done to the copy. The original unmodified copy is called the "Shadow Page". When a transaction commits, all the copies of the pages it has modified are forced to the disk in place of the original pages. If a transaction aborts, all the copies it has modified are just discarded. Crash recovery is also simple due to the no-undo. All the copies of pages modified by uncommitted aborted transactions are removed.

Logging is done at the record level, whereas Shadowing is usually at the page level. Thus concurrency at a finer granularity is possible using Logging. Shadowing requires additional disk space, twice the size of the data base in the worst case when every data page has a copy. Logging needs space for the log on the disk. This log on the disk needs to be periodically saved to tape.

The two methods mainly differ in the way the updates are done. In Logging, the updates are done immediately. So committed transactions do not have any overhead. Shadowing defers the updates until the commit time. So every committed transaction has to first write its modified copies back to the disk. Since the majority of the transactions commit, Shadowing has a higher overhead during normal transaction processing. On the other hand, Shadowing starts improving performance as the number of aborted transactions increases. The overhead for an abort is minimal for Shadowing. In Logging, to un-do an aborted transaction, the undo log records should be applied to the modified pages. The log records and/or the modified data pages might have already been forced to the disk. So the un-doing might require several disk I/O operations.

Crash recovery is quite costly in Logging. It usually consists of three phases, an analysis phase, a redo phase and an undo phase. The last checkpoint record is first determined and the log is scanned forward from the checkpoint record to analyze the state of the database before the crash. Using this information, the log is scanned forward applying the redo log records and then scanned backwards applying the undo log records for aborted transactions. Here again, the actual process depends on the particular implementation. IMS combines the analysis and redo phases into one. Some methods [GMLB 81, Crus 84] perform redo only for the lost updates of committed transactions. ARIES [MHLPS 92] does a redo for all the missing updates first, and then does an undo for the aborted updates. Whatever the mechanism, the recovery process in Logging is much more expensive than that using Shadowing.

Several earlier studies have compared the transaction processing overheads of Logging and Shadowing, e.g. [AgDe 85]. It has been shown that Logging incurs much lower transaction processing overheads than Shadowing for the reasons mentioned above. Hence in our simulations, we consider only Logging for comparing the overheads for *LU-Logging*.

3 LU-Logging

LU-Logging preserves the advantages of using no-undo, while removing the overheads of deferred updating (as used in other implementations based on *redo/no-undo* algorithm), by using lazy updating. Lazy updating helps maintain the overheads during normal processing at a level comparable to that of Logging. Crash recovery is fast since there is little overhead for aborting transactions. We first discuss the main concepts and data organization. Later, we describe the implementation of the recovery mechanism.

3.1 Overview

In this method, whenever a transaction requires an update for a record, only a redo log record is created, without actually updating the data record. The redo log records of a transaction are maintained in the main memory until the commit time. Before committing, the log records are written to the append-only log on the disk. Once a transaction has committed, its redo log records can be applied to update the corresponding data records. Besides the append-only log on the disk, each data page has an associated log page on the disk. This log page contains all the yet-to-be-applied redo log records for the corresponding data page. When a page is written to the disk from the main memory, all its redo log records for in-progress transactions are copied and written to the corresponding log page. When a data page is read from the disk, the corresponding log page is also read. This gives the flexibility of applying the redo log records to the data pages any time after the transaction commits. The redo log records for aborted transactions can just be discarded. The append-only log on the disk is used for crash recovery.

LU-Logging allows locking at the record level. If strict two-phase locking is used, there can be at most one outstanding redo log record for any data record. Also if the size of the log record is not greater than the size of the data record, the size of all the outstanding redo log records for a data page cannot be greater than the size of the data page. Hence one log page is sufficient to hold all pending redo log records for a data page. If the above assumptions are relaxed, the size of all the outstanding log records for a data page may exceed the size of the data page.

Earlier attempts based on no-undo used **deferred** updates. The deferred update method requires reading all the affected data pages into main memory at commit time and applying the updates, resulting in additional I/O. Since *LU Logging* keeps the redo log records with the data pages, the updates of a transaction can be applied to the affected data pages any time after the transaction commits. We call this **lazy** updating. This flexibility eliminates the main source of inefficiency of performing the updates at the commit time during normal transaction processing.

Since *LU-Logging* does not allow uncommitted updates, data records need not be undone if a transaction fails. However, the log records (of the aborted transaction) associated with the data records have to be discarded. We describe in the next section how to efficiently discard the invalid log records. The minimal-undo part facilitates faster transaction aborts. This property is utilized to significantly reduce the crash recovery times. We present simulation studies for evaluating the various overheads of

LU-Logging as compared to those of Logging in different environments. We also show how the crash recovery time can be reduced by an order of magnitude as compared to that of Logging.

The main drawback of *LU-Logging* is the additional disk space required for the log pages. In the worst case, the amount of disk space required is double the size of the database. Most of the implementations which do not perform immediate updates have this overhead. Since disk space is not expensive, we believe that the additional cost is worth the efficiency.

The section on transaction processing analyzes how the overheads for *LU-Logging* compare to those for normal Logging. We consider several database environments with different operating parameters and discuss how the overheads vary based on the parameters. The simulation results presented give an idea of the overheads that can be expected. Crash recovery is explained in a separate section. There the recovery process is analyzed and is shown to perform better than Logging. Before continuing with the description, we like to clarify the terminology we use. When a log record is *written out* from the main memory to the disk, it means that the log record is deleted from the main memory and written on to the disk. When a log record is *copied* from the main memory to the disk, it means that a copy of the log record is made and written to the disk. The original copy of the log record in the main memory still exists after the copying.

3.2 Implementation

All actions by transactions can be classified as either update (write, insert and delete) or read. Each update operation generates only a redo log record. Each log record is assigned unique, increasing number called the log sequence number (LSN). Each log record in the main memory is a member of two lists, one based on the transaction number, called transaction log list, and the other based on the page number, called page log list. When a transaction (T) updates a record in a page (P), a redo log record is generated and kept in the main memory. This log record is inserted in the transaction log list of (T) and the page log list of page (P). The log records of a transaction remain in memory until the transaction commits or aborts.

When a data page (P1) is written to the disk, (P1) is first checked to see if it has some log records by in-progress transactions (i.e. if the page log list for (P1) is not empty). If there are no such log records, just the data page is written to the disk. If (P1) has some log records in the page log list, then these log records are copied into a separate log page, and both the data page (P1) and the log page are written to the disk. The page table entry for (P1) is marked to indicate that the data page has an associated log page on the disk. This process is reversed when a page is to be read. The page table is first checked to see if the page has a log page written along with it. If not, just the data page is read. If the log page exists, it is also read along with the data page. (Note that the log pages exist only in the disk. In the main memory, the log records exist in the log lists.) For each log record in the log page,

- If the transaction that created it has committed, the log record is used to perform the update.

- If the corresponding update has become invalid (aborted or rolled back), the log record is discarded.

- If the transaction that created it is still in-progress, a copy of the log record is still in the main memory, and hence the retrieved log record can be ignored.

Whenever a data page is brought in or written out of the main memory, an I/O-log record is generated and added to a list, called I/O log list, which is global to all transactions. The I/O-log record contains the page number and the associated transfer operation ("read" or "write"). When a transaction commits, its transaction log list is forced to the append-only log on the disk. All the I/O log records in the I/O log list are also written out along with the transaction log list. These I/O-log records are used during crash recovery to determine the order in which the data pages are read into and written out from the main memory (see section 5).

The log records for a transaction (T) are deleted when (T) aborts or rolls back. If (T) aborts, all its log records are deleted and the transaction aborted. On the other hand, if (T) rolls back, all its log records until the previous savepoint, i.e., the point where the state of the transaction was saved previously, are deleted from the lists of the corresponding pages, and then the transaction is continued from the savepoint. An aborted or a rolled back transaction (T) uses its transaction log list to determine the pages that contain the invalid log records that it created. If a data page (P), that has a log record to be deleted, is not in the main memory, then the transaction (T),

1. marks the page table entry of (P) to indicate that (P) contains an invalid log record.

2. generates a record containing the page number, transaction-id and LSN of the log record to be deleted. This record also contains a bit to indicate whether the log record is for an abort or a roll back. The record is kept in the main memory invalid log list.

When a transaction (T) aborts, it just discards all the log records in its transaction log list. For those log records that have copies on log pages on the disk, (T) updates the invalid log list as described above. No log records are written to the disk. Hence once a transaction aborts it does not require any additional I/O operations. When a transaction (T1) commits, its transaction log list is written to the append-only log on the disk and deleted from the main memory. If (T1) has any rolled-back log records still in the invalid log list, the corresponding log records are marked when writing to the disk. These markings help during crash recovery to determine the invalid log records (due to roll-backs) of the committed transactions.

The main disadvantage of delaying updates of a transaction until commit time is that the transaction will not be able to see its own updates. This problem can be easily alleviated here, since the redo log records of a transaction are maintained in the main memory until the transaction terminates. Suppose a transaction wants to read a data item and it already holds the exclusive lock for that item. Here the redo log list for the transaction is first checked for the log record for that data item. By using the

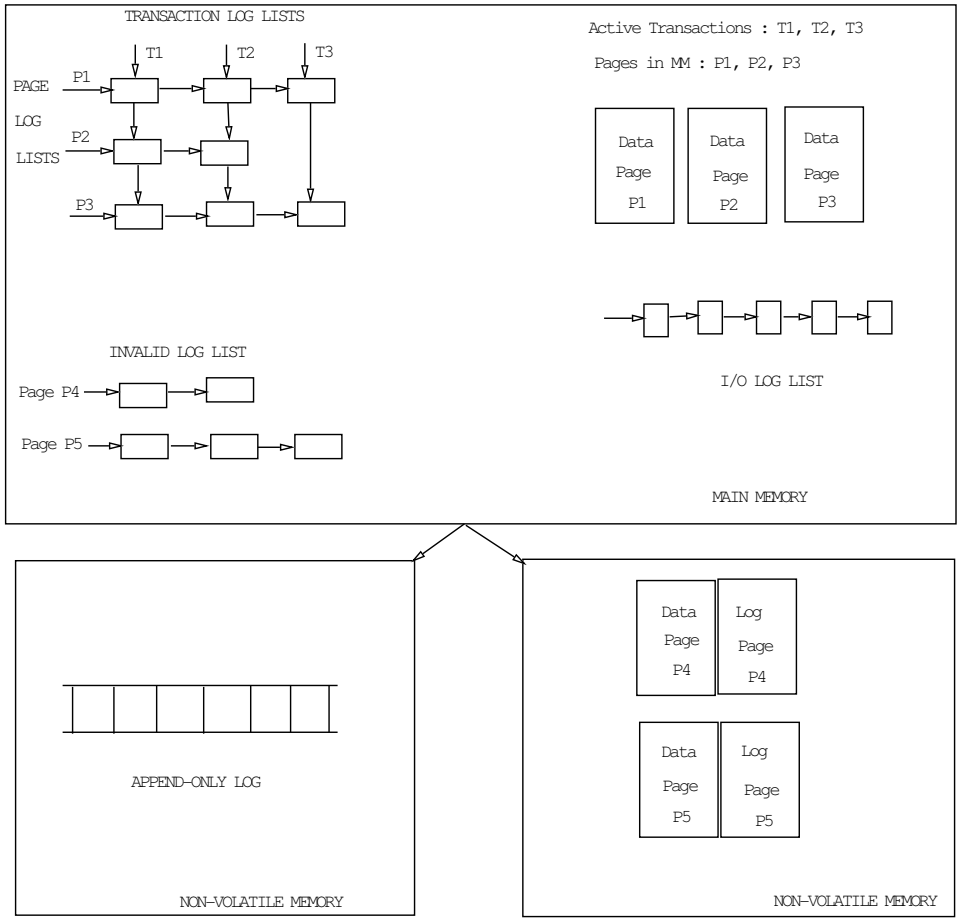

Figure 1: Organization of Log Records

value of the item in the data page and the redo log record, the previous update can be determined (similar to the method described in [BeGH 87]).

All the data structures explained above are illustrated in Figure 1. The figure shows the state of the data base at an instant when three transactions (T1), (T2) and (T3) are executing. Three pages (P1), (P2) and (P3) are in the main memory. The main memory contains the page log lists for (P1), (P2) and (P3) and the transaction log lists for (T1), (T2) and (T3). Two pages (P4) and (P5) are on the disk. The log pages for (P4) and (P5) are also on the disk. The invalid log list in the main memory contains the references of the invalid logs in the log pages of (P4) and (P5). When (P4) or (P5) (along with its associated log page) is next read into the main memory, the invalid log list is used to determine which log records are to be deleted from the associated log page. The log records, in the log pages, which neither are in the invalid log list nor belong to the current transactions (T1), (T2) or (T3) must belong to committed transactions. The I/O log list contains all the I/O log records for page transfers since the last transaction committed. If a data page, say (P1), is written to the disk, its page log list is copied into the associated log page for (P1) on the disk. When a transaction commits or aborts, all the log records in its transaction log list are deleted. These log records are written to the append-only log, if the transaction commits. Else, they are

just discarded.

Each data page contains a field called the *loadLSN*, which is used for recovery. It gives the LSN of the I/O-log generated to record the transfer of this page from the disk to the main memory. The value in the *loadLSN* field is meaningful only when the page is in the main memory.

The state of the database is saved at regular intervals using checkpointing. Checkpointing here can be done simultaneously with normal database operation. The process consists of scanning all the entries in the main memory invalid log list. If an entry belongs to an aborted transaction, the transaction identifier is noted. If the entry corresponds to a roll back, both the page number and the LSN of the entry is noted. The checkpoint record contains the following information:

1. Transaction identifiers of all the aborted transactions whose log records are not yet removed.

2. Page numbers and LSNs of the yet-to-be-removed log records for rolled-back transactions.

3. Minimum of the *loadLSN*s of the dirty pages currently in the main memory.

4. Transaction identifiers of all in-progress transactions.

Checkpointing is done by first writing a "begin checkpoint" record and then collecting all the above information and writing the checkpoint record on the append-only log. Section 5 explains how the data organization presented here is used during crash recovery.

The first step in crash recovery is to identify the last checkpoint record written to the append-only log. This requires scanning back the append-only log from the end searching for the last checkpoint record. This overhead can be eliminated by writing a copy of the checkpoint record, and its location on the append-only log, at a well-known location on the hard disk every time checkpointing is done. The checkpoint record is first written to the append-only log and then to the well-known location. The well-known location always holds the last checkpoint record written. The crash recovery process can now directly access the last checkpoint record without scanning the append-only log.

4 Transaction Processing

Overheads for recovery during normal transaction processing should be low. This is because, as noted before, even a small amount of extra work per transaction adds up over time to large overheads. This section analyzes qualitatively how the overheads in *LU-Logging* differ from those of traditional Logging. Various parameters that affect the recovery overhead are identified. We have conducted a number of simulations varying these parameters. The simulation results are used to validate the qualitative analysis.

The overhead for *LU-Logging* may exceed that of traditional Logging in two respects. Since all the redo log records for in-progress transactions are maintained in the main

memory, the log records occupy more space in the memory, reducing the space available for data pages. This might, in turn, result in more I/O operations for data pages due to potentially higher number of page faults. If the size of the main memory is far greater than the additional space required for *LU-Logging*, the resulting extra overhead is expected to be negligible. On the other hand, if the main memory is small, the log records are expected to occupy a significant portion of the main memory, making the extra overhead for *LU-Logging* high.

The second reason for the extra overhead is that whenever a data page is read from or written to the disk, the corresponding log page should also be transferred. This results in additional I/O operations. There are two ways to reduce this overhead. The log page can be placed adjacent to the data page so that both pages can be read or written in a single I/O operation. The only overhead here is the additional transfer time. The overhead can be further reduced by avoiding writing or reading log pages when they are empty. The other way out is to use a pair of synchronous disks. The data pages can be placed on one disk and the associated log pages on the other. Both pages can be read or written in a single I/O operation. The transfer time overhead is avoided using this method.

LU-Logging has minimal CPU overhead and no I/O overhead for transaction aborts. In traditional Logging, the undo log records of the aborted transaction have to be applied to the modified pages. This might require reading log records and the data pages from the disk. Hence, just as for the "redo/no-undo" algorithm, the performance of the *LU-Logging* mechanism starts improving, as compared to traditional Logging, as the number of aborts and roll-backs increases.

A number of simulation studies were carried out to determine the performance of the *LU-Logging* mechanism during normal transaction processing. ARIES was selected as the benchmark to compare against, since it uses the write-ahead-log protocol, accommodates flexible buffer management policies (steal and no-force) and has an overhead, for normal transaction processing, typical of any traditional Logging mechanism.

4.1 Simulation Parameters

The database is modelled as having a number of data pages, each page containing a fixed number of records. The main memory has a fixed number of buffer pages. Each transaction is modelled as a number of data references. A reference can be either a read or an update reference. When a data page is required by a transaction, it first checks if the page is in the main memory. If not, a buffer page is freed in the main memory and the required data page is read. Strict two phase locking is used for concurrency control. Page replacement is based on Least-Recently-Used policy.

The hardware consists of a single processing unit, which is time-shared between multiple transactions. The number of active transactions in the system is determined by the degree of multi-programming, which is held constant. When a transaction is done, either committed or aborted, it leaves the system and a new transaction is started. Both Logging and *LU Logging* methods require non-volatile storage to keep the data base and the append-only log. Traditional Logging requires a random-access storage for the append-only log, since it may have to read the log during normal processing

for un-doing aborted or rolled-back transactions. A large capacity sequential-access storage is also required to backup the append-only log on the random-access storage as it becomes full. The append-only log and the data base are not both kept on the same disk, since a single disk failure might result in losing all the information. Hence traditional Logging requires at least two disks, one for keeping the data base and one for the append-only log, and a tape drive for backing up the append-only log.

LU-Logging never reads the append-only log except for crash recovery. All the log records it requires during normal transaction processing are either in the main memory or in the log pages associated with the data pages on the disk. Hence, here, just the sequential-access storage is sufficient for the append-only log. The data base and the log pages can be kept in a single disk or in a pair of synchronized disks, as explained in Section 4. We have simulated both combinations. The first combination, called *LU-Logging* (1), has a single disk for both data pages and log pages. Here, a log page is placed just next to the corresponding data page. The overhead for reading or writing a log page is the additional transfer time. The second combination, referred to as *LU-Logging* (2), has two disks, one for the data pages and the other for the log pages. A log page is placed at the same place on one disk as the corresponding data page is placed on the other disk. This combination has no transfer time overhead.

LU-Logging also incurs additional CPU overhead for maintaining the extra data structures. When a data page, which has some associated log records of in-progress transactions, is written to disk, a log page should be constructed and written along with the data page. Constructing the log page has the CPU overhead of copying the associated log records from the page log list to the log page. Managing the I/O log list and the invalid log list also require CPU processing. These overheads are captured in the simulation model by requiring *LU-Logging* to use the CPU for a constant "Log Record Generation Time" every time a log record is created or copied in any one of the log lists.

A database simulation can have a large number of parameters. Most of the parameters that we felt would not make a significant difference were made constant. Some of the constant parameters are:

- The degree of multi-programming is 40.
- Page size is fixed at 4k.
- Both data records and log records are each 100 bytes long.
- Size of the database is 5000 pages.
- CPU time slice is 10ms.
- Seek time is 25ms, latency time, 5ms and page transfer time, 3ms.
- A transaction, after each data access, requires CPU for an average of 5ms.
- Acquiring or releasing a lock takes 100 μs.
- Acquiring or releasing a latch takes 50 μs.

- Generating a log record takes 100 μs.

The variable parameters identified are:

- Ratio of reads to updates - three ratios 20:80, 50:50 and 80:20 are considered.
- Page reference pattern - both uniform page access and eighty-twenty (i.e. eighty percent of the references are for twenty percent of the pages) access are used.
- Size of the main memory versus the total references generated by all the active transactions - this is explained in detail in the following paragraph.
- Percentage of aborts - 0%, 5% and 10% of the transactions are aborted. The expected aborts percentage in an actual data base environment seems to be about 3 to 5% [HaRe 83, Gray 81]. The references are so designed as not to get transactions into deadlocks.

The third variable parameter needs some explanation. Suppose (n) is the number of transactions that can be active at any given instant (i.e. degree of multiprogramming) and each transaction has (r) data references. The total data references, and hence the maximum pages referred, by the (n) transactions is (n * r). Consider the uniform reference pattern case. If the size of the main memory is greater than (n * r), all the pages that an active transaction ever refers can be kept in the main memory until the transaction commits. Hence no log pages need ever be written along with data pages. On the other hand, if the size of the main memory is less than (n * r) some of the data pages with pending redo log records may have to be forced to the disk, and hence the overhead for writing the log pages comes into the picture. For the eighty-twenty reference pattern, the situation is slightly different. The twenty percent most frequently accessed data pages contain eighty percent of the log records. If the size of the main memory is greater than twenty percent of the data base size, all the most frequently accessed pages can be kept in the main memory, thus reducing significantly the overhead for writing log pages.

We also varied the number of I/O servers (by having N buffers and N I/O servers, page P in (P mod N) buffer) in our simulations to check if the number of I/O servers affects the recovery. This brought out a subtle difference between traditional Logging and *LU-Logging* mechanisms. In traditional Logging, all log records are written to a single log in the main memory. If a transaction has to write a log record, but finds that there is no space in the main memory, it has to free a buffer page, by either writing in-memory log records to the append-only log on the disk, or by forcing a data page to the disk. The transaction blocks while the space is being freed. Since the other transactions also use the same log, all other transactions that want to write log records are also blocked. These blocked transactions, once space is available in the main memory, again queue up for CPU. This adds additional waiting times (in the CPU queue) to the total time. As the number of I/O servers is increased, the CPU starts becoming the bottle neck. At about six I/O servers, the CPU utilization rate goes up to 0.64 and waiting times for CPU become longer. This increases the total time for ARIES to much more than that of *LU-Logging*. In the *LU-Logging* method,

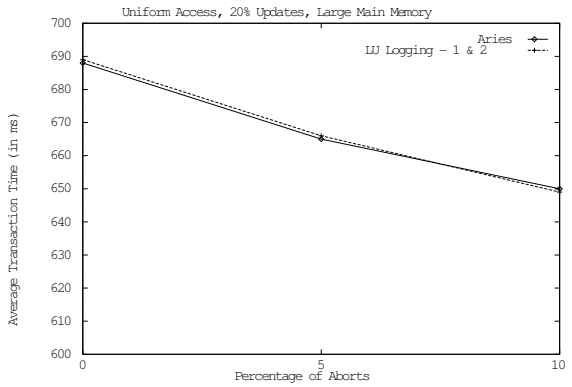

Figure 2: Transaction Time vs Aborts

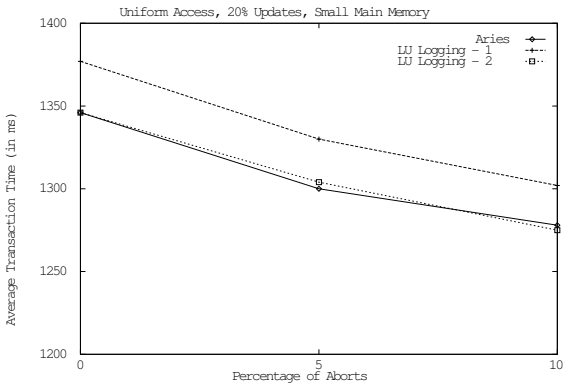

Figure 3: Transaction Time vs Aborts

each transaction maintains its own transaction log in the main memory. Hence the above "domino" effect is not present. This "domino" effect for ARIES can be avoided by policies like reserving certain space in the main memory for log and freeing main memory space in advance. We found that the relative performances of the two methods using multiple (less than six) I/O servers are almost the same as those using a single I/O server. The average transaction times are used to compare the different methods in the results to be presented next.

4.2 Simulation Results

Our aim in this section is to show that the performance of *LU-Logging* is comparable to that of traditional Logging during normal processing. In the next section, we show that the crash recovery time of *LU-Logging* is much better than that of Logging.

Simulations are done for ARIES and for the two variations of *LU Logging*. As mentioned in the previous sub-section, *LU-Logging*-1 uses a single disk to hold both data and associated log pages, whereas *LU Logging*-2 uses two synchronous disks, one for the data pages and the other for the log pages. Only about 10% of the simulation code is different between ARIES and *LU-Logging* methods. For each set of parameters, the same transactions are used for all methods. Two thousand transactions are first run to generate a steady state. Then five hundred transactions are run and the average time per transaction is determined.

Figure 2 shows the uniform reference pattern case, with Read:Update ratio of 80:20. Here, each transaction contains 20 data references. The size of the main memory is 1000 pages. The main memory is more than the total page references (20 * 40) by all the active transactions. Here both the variations of *LU-Logging* yield almost similar results, which shows that very few log pages are written with the data pages. This result is to be expected since all the pages a transaction accesses can be in the main memory until commit time and hence there are never any pending redo log records when a data page is forced to the disk. The differences between *LU-Logging* and ARIES are never more than 1%. The graph also shows that the transaction time for *LU-Logging* starts improving as the percentage of aborts increases.

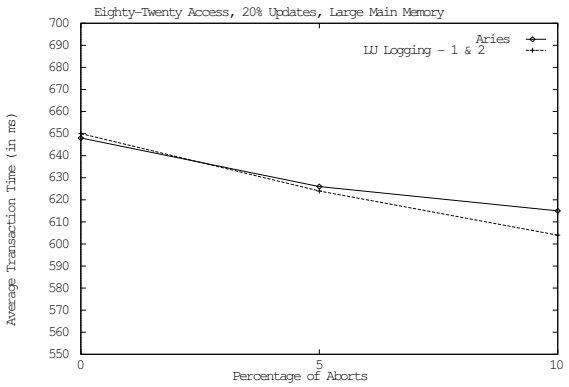

Figure 4: Transaction Time vs Aborts

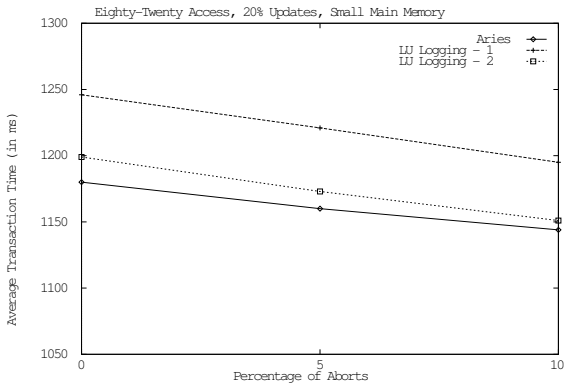

Figure 5: Transaction Time vs Aborts

Figure 3 is for the uniform reference pattern with the main memory smaller than the total page references. The main memory size is still fixed at 1000 pages, but the number of data references per transaction is changed to 40, thus bringing up the total page references to 1600. The graph shows that *LU Logging* (1) performs about 2% worse than ARIES at 0% aborts. The difference slightly reduces as the percentage of aborts increases. *LU-Logging* (2) closely matches ARIES at 0% aborts and gradually becomes better as the percentage of aborts increases.

The next two figures are for the eighty-twenty reference pattern. The number of references per transaction is held constant at 40 for both cases. Figure 4 shows the variations when the main memory size is 1200 pages, which is greater than 20% of the database size. Here again, both versions of the *LU-Logging* scheme perform the same. Both of them perform slightly worse than ARIES at no aborts, slightly better at 5% aborts and nearly 2% better at 10% aborts.

Figure 5 gives the performance with a main memory size of 500 pages, which is less than 20% of the data base size. As in the uniform case, *LU-Logging* (2) has performance comparable to that of ARIES. The differences were never more than 2%. *LU-Logging* (1) performs about 5% worse at no aborts to 4% worse at 10% aborts. The performance of *LU-Logging* (1) deteriorates further as the size of the main memory is reduced.

The next three figures show how the transaction times change as the different parameters are varied. All three figures are for uniform reference pattern, with a main memory size of 1000 pages and degree of multiprogramming of 40. Figure 6 is an extension of Figure 3, as the percentage of aborts is varied up to 30. In Figure 7, the percentage of aborts is fixed at 10%, the number of references at 40 and the percentage of updates is varied from 20 to 80. The two figures show that *LU-Logging* performs better as the amount of work to be undone (for transaction aborts) increases. Figure 8 holds the aborts at 10%, updates at 20% and varies the number of data references per transaction from 20 to 80. As the number of data references increases, the main memory becomes insufficient to hold all the pages of the transactions until they commit. The additional transfer time overhead for *LU-Logging* (1) comes into picture now. At 80 references per transaction, *LU-Logging* (1) performs 5% worse than ARIES. *LU-Logging* (2) does not have this overhead and hence performs the same ARIES.

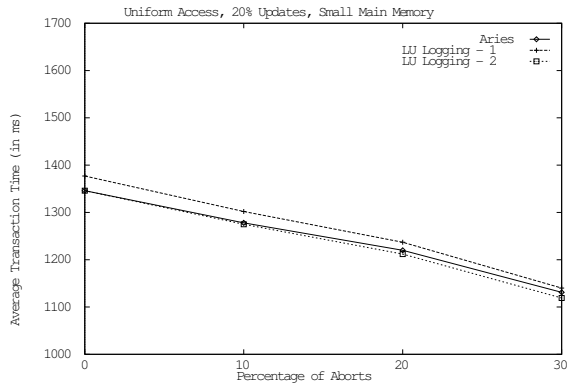

Figure 6: Transaction Time vs Aborts

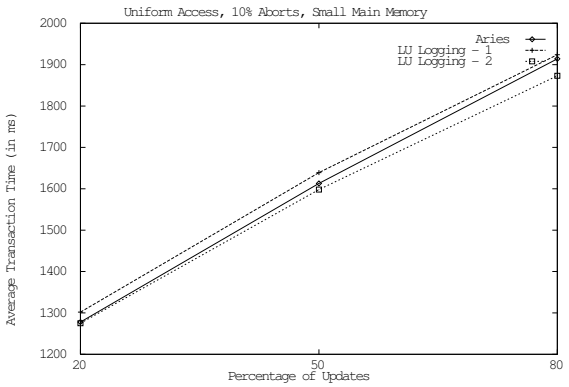

Figure 7: Transaction Time vs Updates

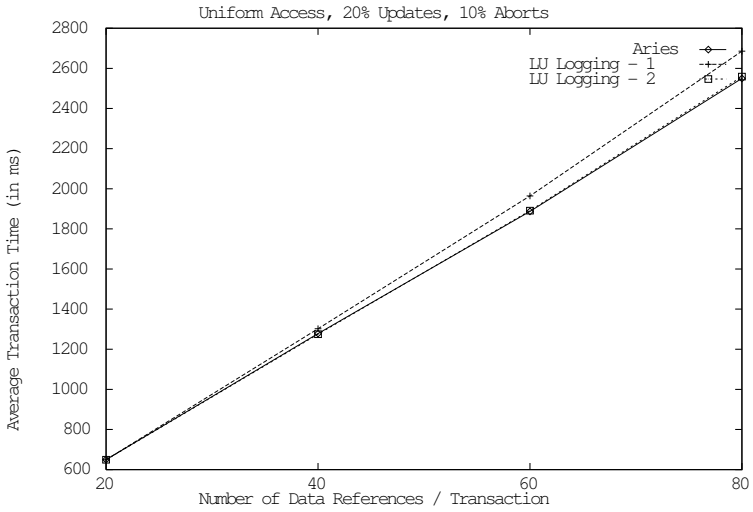

Figure 8: Transaction Time vs References

LU-Logging keeps all the log records of an in-progress transaction in the main memory until the transaction finishes. Thus, this method requires more main memory space for log records than ARIES. A simple calculation gives the amount of buffer space required for log records. Let (n) be the degree of multi-programming, (r) the number of data references per transaction, and (u) the percentage of updates. If (l) log records fit a buffer page, the maximum buffer space *LU-Logging* requires for log records in the main memory is (nru/l) pages and the average number is (nru/2l) pages. Figures 9 and 10 show how the buffer space requirement for log records varies with different parameters for both methods.

Figure 9 shows how the average buffer space for log records required by the two methods varies as the percentage of updates is changed. The parameters for this figure are the same as those for Figure 7 (uniform access, 10memory, degree of multiprogramming is 40 and 40 data references per transaction). The buffer space requirement for *LU-Logging* grows more rapidly than that of ARIES. But the average buffer space required (about 16 pages at 80% updates) is quite small compared to the total main

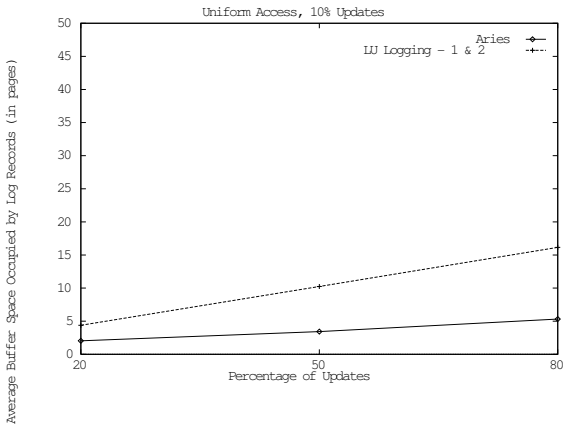

Figure 9: Log Buffer Space vs Updates

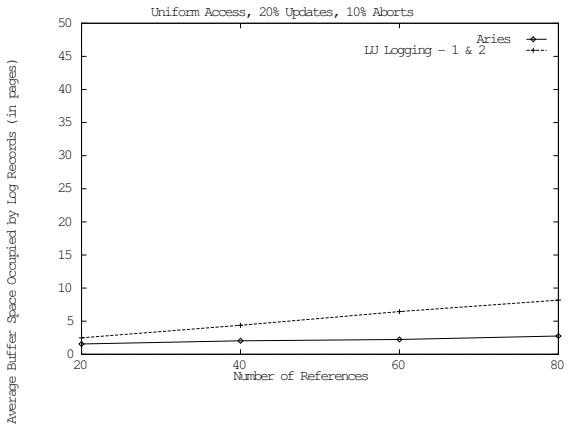

Figure 10: Log Buffer Space vs References

memory space of 1000 pages. Figure 10 shows the variation in buffer space occupied by log records as the number of data references per transaction is varied (with the same parameters as those of Figure 8, uniform access, 20updates, 10multiprogramming is 40). Here again, the buffer space occupied by *LU-Logging*, though greater than that of ARIES, is still small (less than 10 pages). From Figures 7, 8, 9 and 10, it is clear that the additional buffer space required by *LU-Logging* is too small to have a significant effect on the average transaction time. The maximum number of buffer pages required for log records in all the simulations never exceeded 40 pages. Since the values of the simulation parameters considered here are typical of many database systems, it is safe to assume that there is always enough space in the main memory to hold all the log records of in-progress transactions.

The simulation results indicate that the overheads are mainly due to the additional I/O time required to read and write log pages along with the data pages. The conclusions are that a single disk is sufficient for both data and log pages,

1. when the size of the main memory is greater than the total number of page references of all concurrent transactions, or

2. when the main memory can hold all the frequently accessed pages.

In other cases, when the size of the main memory is small, it is recommended to have data pages and log pages on separate synchronous disks. Note that ARIES performs better than both variations of *LU-Logging* when all transactions commit. At about 3 - 5% aborts, which is the typical figure considering roll-backs and deadlocks, *LU-Logging* matches, and sometimes performs better than, ARIES.

5 Crash Recovery

When the system crashes, all data in the main memory is lost. The redo log records at this stage can be divided into two classes; those in the log pages and the ones in the append-only log. The recovery process needs to determine which of the redo log records in the log pages correspond to committed transactions and have to be applied,

and which ones are by aborted transactions and hence have to be discarded. Besides, the updates made by committed transactions to data pages in the main memory just before the crash are also lost. The append-only log contains the redo log records of all committed transactions. This log should be used to determine the redo log records for the lost updates. The recovery process consists of the following:

1. Get the *min-loadLSN* value from the last checkpoint record.
2. Scan the append-only log forward from the *min-loadLSN* log record.
   (a) Group the log records based on their page numbers.
   (b) Determine the aborted transactions.
   (c) Determine the yet-to-be-deleted log records of rolled-back transactions.
3. Read each data page and its associated log page and perform the necessary updates.

Each of these steps is explained in detail below. A few properties are first presented, which are later used to show the correctness of the recovery process.

- Let (M) be the size of the main memory in pages. At any given instant, there are at most (M) data pages in the main memory. Some, or all, of these pages may contain updates of committed transactions that are not yet reflected on the disk. When the system fails, all the (M) data pages in the main memory at that instant are lost. Only these (M) pages may have committed updates, which are lost. Hence there are at most (M) data pages for which updates are lost.

- Only committed updates may need to be re-done. A transaction, and hence its updates, are committed only after the transaction log list is forced to the disk. Any data page that contains at least one of these updates must have been read into the main memory before the update, and therefore before the commit. Hence, the "read" I/O-log record of the page must appear in the append-only log before the log records for that page for any committed transaction.

- The append-only log on the disk contains all the I/O-log records in the same order in which they occurred. If this log is scanned sequentially, all the "read" I/O-log records, which have no matching "write" I/O-log records, are potential candidates requiring redo. As explained above, these candidates cannot be more than the number of pages in the main memory.

- By construction, the log on the disk contains only committed redo log records.

- Redo log records for a particular data record appear in the same order in the append-only log as the corresponding updates that are applied to the data record. This follows from the strict 2-phase locking protocol. Suppose a record (R) is updated by two transactions (T1) and (T2), both of which commit. If (T1) is the

first transaction to update (R), (T2) can update (R) only after (T1) commits. Hence the redo log records for (R) appear in the append-only log on the disk in the same order.

- If there are more than one outstanding redo for a particular data record, only the last redo needs to be applied. Any other earlier redo can be discarded. The size of the log record is assumed not to be larger than the size of the corresponding data record. Hence, the size of the outstanding redo log records for a data page cannot be more than the size of the data page.

The first phase in the recovery process is to determine the potential data pages that lost updates and the corresponding redo log records. The last checkpoint record is accessed from the well-known location on the hard disk. The *min-loadLSN* value in the last checkpoint record gives the log record from which to scan forward. Initially the main memory contains only one empty redo log page, corresponding to the data page number recorded with the *min-loadLSN* in the checkpoint record. While scanning the log forward, if a "read" I/O-log is encountered, a corresponding empty redo log page is initialized in the main memory. If a "write" I/O-log is encountered, the corresponding redo log page in the main memory, if present, is discarded. Since there are never more than (M) data pages in the main memory at any given instant, the main memory is sufficient to hold all the redo log pages, active at any given instant while scanning. For each redo log record encountered, if the corresponding redo log page is in the main memory, the redo log record is added to that log page. If there is an earlier redo log for the same data record, the earlier one is discarded. At the end of the scan, we have in the main memory all the committed log records (grouped by page numbers), whose updates might have been potentially lost because of the crash.

The process described above determines all the potential log records for any lost updates. The next step in the first phase is to determine the invalid log records in the log pages. All the log records for aborted transactions are invalid. So it suffices to find the list of aborted transactions to delete their invalid log records. But for rolled-back transactions, only the rolled-back log records are invalid. To delete these invalid log records, we also need their page numbers and LSNs.

The checkpoint record gives the list of aborted transactions. This record also contains the information about the yet-to-be-removed invalid log records of rolled-back transactions. All the transactions that are in-progress during checkpointing are also recorded in the checkpoint record. A forward scan from the checkpoint record can determine all the transactions aborted after the checkpoint record was written. The list of yet-to-be-removed log records of rolled-back transactions can be constructed based on the log records that are marked as rolled-back. This scan can either be done along with the scan for redo log records or separately.

At the end of the scanning phase, the main memory contains all the potential log records for any lost updates and the list of aborted transactions. A new checkpoint record, containing all the above information, is now written to the append-only log. All the information collected in the first phase is maintained in the main memory.

Normal database processing can be started after the first phase. If a transaction requires a data page, it first reads in both the data page and the associated log page.

The data page is checked with the list of updates and aborted transactions determined in the first phase. Any lost updates by committed transactions are now applied and the invalid log records are discarded. Once a data page is updated, the information, if any, about the data page, collected during the first phase, is deleted from the main memory. The data page is now available to the transaction.

There is no way to determine, from the append-only log, all the log pages on the disk that have invalid redo log records by aborted transactions. Consider an example. Suppose (T1) is an active transaction in the system just before the crash. Also suppose (T2) is the last transaction committed before the crash. If (T1) accessed a page (P) after (T2) had committed, and then page (P) had migrated to the disk, the log page for the data page (P) would contain the redo log record for (T1). This information is not on the append-only log since nothing is written to the append-only log when a data page is written to the disk. Now if the system crashes, (T1) is aborted. The only way to weed out the redo log records by aborted transactions is to check every page. The recovery process determines which transactions are to be aborted. This information is kept in the memory. Now, if a new transaction accesses page (P), it can determine, using the information from the recovery process, which redo log records should be applied and which ones discarded. But, since we do not know which pages contain redo log records by the aborted transactions, we do not know when to remove the information about the aborted transactions from the main memory. Also, page (P) may not be accessed for a long time, during which the system might crash a number of times. So the information about aborted transactions may have to be stored forever. To overcome this problem, the recovery procedure , after the first phase, starts a low priority process (LPP). This process accesses each data page not already recovered by some other transaction. The (LPP) makes the page consistent by applying any outstanding log records by committed transactions and discarding the aborted ones. To free up the main memory space occupied by the information collected in the first phase, the (LPP) should first access the pages with lost updates.

If the system crashes during the first phase, the recovery process can again be restarted from the first phase after the system is brought up. At the end of the first phase a checkpoint record containing all the information collected is written to the append-only log. If the system crashes now, the recovery process starts from the checkpoint record written before. The page table is assumed to be lost after the system crash. Hence the (LPP) has to be started afresh after every restart.

Figure 11 shows how the average main memory space required for the lost redo information changes as the number of references is varied. To determine the amount of buffer space required for the lost redo information collected, the simulation process is stopped once in every 100 ms and the number of "lost" redo log records (if the system were to fail at that instant) is determined. Both the uniform access and the 80-20 access cases have 1000 pages of main memory. The degree of multiprogramming is fixed at 40. The amount of space required decreases as the number of references increases. With a large number of references, all the data pages required by a transaction cannot be kept in the main memory until the end. Data pages are forced to the disk more frequently as the number of references increases. This frequent swapping of the data pages reduces the number of updates that can be lost if the system crashes. In the 80-20 access case,

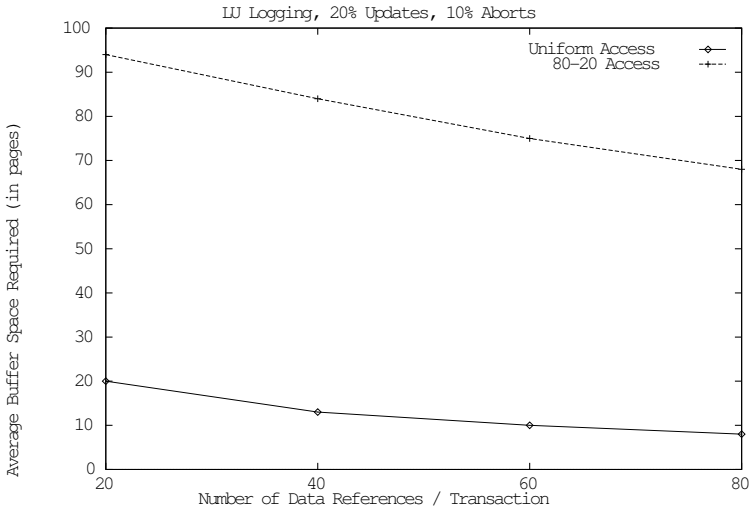

Figure 11: Buffer Space Required to Hold Lost Redo Log Records

the frequently accessed data pages tend to stay in the main memory for a long time (because of the Least-Recently-Used policy) and hence accumulate a large number of updates not reflected on the disk. This results in a large number of lost updates (as shown in Figure 11) if the system crashes. By following a suitable policy to force old data pages from the main memory, this situation can be avoided. It is also interesting to note that the standard deviation for the buffer space required above is quite small.

The overhead of the recovery process as described above is considerably smaller than that of ARIES. ARIES first scans the append-only log forward from the checkpoint record in the analysis phase. Then in the redo phase, it scans the append-only log from the LSN noted in the checkpoint record. All the lost redos are applied immediately, which requires reading and writing data pages. Once the state before the crash is established, the append-only log is scanned backwards un-doing the effects of all the aborted transactions. Here again the necessary data pages may have to be read, the updates un-done and the data pages written back to the disk. The overhead for *LU-Logging* is just the scanning part of the redo phase. Also note that the size of the append-only log for *LU-Logging* is less than half that for Logging. This is because *LU-Logging* generates only redo logs and the log records of only the committed transactions are written to the append-only log. Logging, on the other hand, generates two log records (redo and undo) for each update and writes all log records generated to the append-only log. The overhead for the (LPP) is not considered here since the (LPP) does not interfere with normal transaction processing. The (LPP) accesses only those pages not accessed by any transaction. It requires at most one page of main memory, holds at most one page latch and operates only when the system is lightly loaded.

We refer to [JhKh 92] for an excellent quantitative analysis of the crash recovery process. That paper examines the recovery time in a database system which uses a write-ahead log protocol, specifically ARIES. The analytical equations for log scan time, data I/O, log application and undo processing time are presented there. The overhead during the first phase in *LU-Logging* scheme is the same as the log scan time

in ARIES. A few of the important conclusions of [JhKh 92] are presented below to give an idea of the difference in overheads.

1. For uniform access patterns

   - The recovery time is dominated by the data I/O, followed by the log application and scan time.
   - The data I/O time is an order of magnitude more than the log application and scan time.

2. For non-uniform access patterns, as the probability of access to a *hot page* increases, both the log scan and log apply times increase with the data I/O time remaining fairly constant.

In *LU-Logging*, the entire database is accessible for normal transaction processing after the first phase. In contrast, in ARIES [Moha 91], after the analysis phase, all the data pages that may potentially be affected by redo or undo phases are not made available for normal processing. After the redo phase, all pages that may potentially contain updates to be un-done are not released for transactions. This is particularly restrictive in a case like an eighty-twenty access pattern, where the most accessed pages are most likely to be the ones in the main memory before the crash. If these pages are not released until the end of the recovery process, a majority of the new transactions are effectively blocked.

6 Media Recovery

Media recovery is required if some or all of the data pages on the disk are lost. The entire database is occasionally dumped on to backup tapes. This dumping process can proceed concurrently with the database operation. To take a backup of a data page, it is first brought in to the main memory along with its log page. All the log records in its log page that belong to committed transactions are applied to the data page. The data page now contains all the committed updates until the last committed transaction. The data page is now written to the backup along with the transaction identifier of the last committed transaction.

To recover a lost page, the corresponding data page in the last backup is first loaded into the main memory. The data page from the backup also indicates the last committed transaction until which the page contains updates. The transactions appear on the append-only log in the same order in which they committed. The append-only log should be scanned from the committed transaction indicated in the backup data page. All the log records corresponding to the data page are applied and the lost data page can be reconstructed.

7 Summary and Future Work

In this paper, we introduce a new recovery method, *LU-Logging* based on *flexible-redo/minimal-undo* algorithm. While all the earlier methods based on the no-undo paradigm used deferred updating, our method uses lazy updating. This eliminates the overheads due to deferred updating during normal transaction processing. We presented an efficient implementation of the proposed method. The different data structures and log organizations required for the implementation are also described.

Using simulations and qualitative analysis, we demonstrate that the overhead during normal transaction processing for *LU-Logging* compares favorably with that of Logging. The simulations also identify the best disk configurations in different data base environments. The crash recovery process is explained in detail. The crash recovery times are shown to be an order of magnitude faster than those for Logging.

LU-Logging can easily handle B-Tree indices. When a data record is updated, its entry in an index may shift depending on which fields are updated. The technique here is to create a new entry (based on the updated value) without actually deleting the previous entry. If the transaction commits, the old entry is deleted. If the transaction aborts, the new entry is removed. As pointed out in [MoPL 92], this approach avoids the *next key locking* during key delete. Since index entries of deleted keys are maintained atleast until the end of transaction, *logical* key deletion capability comes automatically.

The *LU-Logging* method seems to be suitable for multi-processor data base environments. We are currently looking at how *LU-Logging* can be extended to the shared-everything and the shared-nothing environments.

References

[AgDe 85] Agarwal, R., Dewitt, D.J. *Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation*, ACM Transactions on Database Systems, December 1985.

[BeHG 87] Bernstein, P.A., Hadzilacos, V., Goodman, N. *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Pub. Co., 1987.

[Borr 84] Borr, A. *Robustness to Crash in a Distributed Database: A Non Shared-Memory Multiprocessor Approach*, Proc. 10th International Conference on Very Large Data Bases, August 1984.

[CKKS 89] Copeland, G., Keller, T., Krishnamurthy, R., Smith, M. *The Case for Safe RAM*, Proc. 15th International Conference on Very Large Data Bases, August 1989.

[Crus 84] Crus, R. *Data Recovery in IBM Database 2*, IBM Systems Journal, Vol. 23, No. 2, 1984.

[Curt 88] Curtis, R. *Informix-Turbo*, Proc. IEEE Compcon, February-March 1988.

[GMLB 81] Gray, J.N., McJones, P., Lindsay, B.G., Blasgen, M.W., Lorie, R.A., Price, T.G., Putzolu, F., Traiger, I.L. *The Recovery Manager of System R Database Manager*, ACM Computing Surveys, June 1981.

[Gray 78] Gray, J.N. *Notes on Database Operating Systems*, In R.Bayer, R.M.Graham and G.Seegmuller (Eds.) Lecture Notes in Computer Science, Springer Verlag, New York, 1978.

[Gray 81] Gray, J. *The Transaction Concept: Virtues and Limitations* , Proc. 7th International Conference on Very Large Data Bases, September 1981.

[HaRe 83] Haerder, T., Reuter, A. *Principles of Transaction- Oriented Database Recovery*, ACM Computing Surveys, December 1983.

[JhKh 92] Jhingran, A., Khedkar, P. *Analysis of Recovery in a Database System Using a Write-Ahead Log Protocol*, ACM-SIGMOD, June 1992.

[Lori 77] Lorie, R.A. *Physical Integrity in a Large Segmented Database*, ACM Transactions on Database Systems, March 1977.

[MHLPS 92] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P. *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*, ACM Transaction on Database Systems, March 1992.

[Moha 91] Mohan, C. *A Cost-Effective Method for Providing Improved Data Availability During DBMS Restart Recovery After a Failure*, Proc. 4th International Workshop on High Performance Transaction Systems, September 1991.

[MoPL 92] Mohan, c., Pirahesh, H., Lorie, R. Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions, ACM SIGMOD, June 1992.

[Ong 84] Ong, K. *SYNAPSE Approach to Database Recovery*, Proc. 3rd ACM SIGACT-SIGMOD Symposium, April 1984.

[Reut 80] Reuter, A. *A Fast Transaction-Oriented Logging Scheme for UNDO Recovery*, IEEE Transactions on Software Engineering, July 1980.