Differential Logging: A Commutative and Associative Logging Scheme for Highly Parallel Main Memory Database

Juchang Lee Kihong Kim Sang K. Cha

Graduate School of Electrical Engineering and Computer Science
Seoul National University
{juch, next, chask}@kdb.snu.ac.kr

Abstract

With a gigabyte of memory priced at less than $2,000, the main-memory DBMS (MMDBMS) is emerging as an economically viable alternative to the disk-resident DBMS (DRDBMS) in many problem domains. The MMDBMS can show significantly higher performance than the DRDBMS by reducing disk accesses to the sequential form of log writing and the occasional checkpointing. Upon the system crash, the recovery process begins by accessing the disk-resident log and checkpoint data to restore a consistent state. With the increasing CPU speed, however, such disk access is still the dominant bottleneck in the MMDBMS. To overcome this bottleneck, this paper explores alternatives of parallel logging and recovery.

The major contribution of this paper is the so-called differential logging scheme that permits unrestricted parallelism in logging and recovery. Using the bit-wise XOR operation both to compute the differential log between the before and after images and to recover the consistent database state, this scheme offers the room for significant performance improvement in the MMDBMS. First, with logging done on the difference, the log volume is reduced to almost half compared with the conventional physical logging. Second, the commutativity and associativity of XOR enables processing of log records in an arbitrary order. This means that we can freely distribute log records to multiple disks to improve the logging performance. During the recovery time, we can do parallel restart independently for each log disk. This paper shows the superior performance of the differential logging comparatively with the physical logging in the shared-memory multiprocessor environment.

1. Introduction

Emerging data-intensive applications such as e-commerce and mobile value-added information services require graceful processing of highly concentrated user transaction profiles. The traditional DRDBMS cannot effectively deal with such a requirement because of the overhead associated with processing the disk-resident data indirectly through the main-memory buffer. With a gigabyte of memory priced at less than $2,000 these days, the MMDBMS emerges as an economically viable alternative to the DRDBMS with the data structures and algorithms optimized for the in-memory access.

The MMDBMS can show higher performance than the DRDBMS not only for read transactions but also for update transactions by orders of magnitude. This is because the disk access in the MMDBMS is reduced to the sequential form of log writing and the occasional checkpointing [4]. Upon the system crash, the recovery begins by accessing the disk-resident log and checkpoint data to restore a consistent state. However, in the MMDBMS, such disk access is still the dominant bottleneck, especially, for update-intensive applications. To overcome this bottleneck, this paper explores the alternatives of parallel MMDB logging and recovery. The availability of low-cost but high-speed multiprocessor platforms with many commodity disks justifies this direction of research.

The novel contribution of this paper is the so-called differential logging scheme that permits unrestricted parallelism in logging and recovery. Based on the nice properties of the bit-wise XOR operation, this scheme uses this single operation both to computes the differential log between the before image and the after image, and to recover a consistent database state from the checkpoint data and a collection of log records. Compared with the well-known DRDBMS-oriented recovery schemes such as ARIES [9], the differential logging offers the room for significant performance improvement in MMDBMS. First, since logging is done on the difference, the log volume is reduced to almost half compared with the physical logging, which writes both the before and after images. Second, since the recovery process involves a single XOR operation, there is no need to distinguish the redo and undo phases as in the typical DRDBMS. In addition, since the XOR operation is both commutative and associative, the log records can be processed in an arbitrary order, independent of the serialization order. This means that we can freely distribute log records in parallel to an arbitrary number of disks to improve the logging per-

1063-6382/01 $10.00 © 2001 IEEE

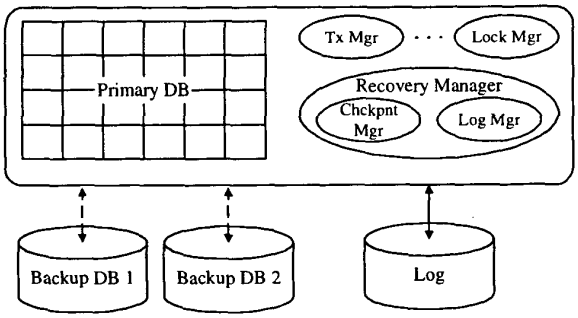

Figure 1. Logical structure of MMDB

formance. During recovery, we can do parallel restart independently for each of log disks. Even intermixing the log replay and the backup DB replay is allowed to pursue the maximum degree of parallelism.

To verify the benefit of using the differential logging, we have implemented the parallel recovery schemes based on the differential logging and compared them with those based on the physical logging. Both types of schemes use the transaction-based log partitioning to distribute log records to multiple log disks for parallel logging and parallel restart. In a series of experiments conducted in a 4-CPU multiprocessor environment with multiple log disks, the differential logging outperforms the best physical logging implementation by about twice in the transaction processing performance and by about six times in the log processing time for the recovery.

This paper is organized as follows. Section 2 presents an overview of MMDB recovery and discusses the problems associated with parallelizing it. Section 3 presents the definition and properties of differential logging and the parallel MMDB logging and recovery algorithms based on it. Section 4 presents a framework for parallel logging and recovery based on the transaction-based log partitioning, and explores the levels of parallelism achievable in the multiprocessor environment with multiple disks. Section 5 presents the result of the experiment conducted to compare the differential logging with the physical logging. Section 6 briefly compares the differential logging with the previous MMDB recovery schemes, and section 7 finally concludes this paper.

2. Parallelism in MMDBMS

2.1. MMDB Structure

Figure 1 shows the usual MMDB structure. The primary DB keeps the up-to-date data in memory. The log manager maintains the in-memory log buffer and the disk-resident log volume. The checkpoint manager creates backup database copies from time to time to avoid replaying the entire log during the recovery time. Typically, the two most recent backup copies are kept. The recovery manager recovers a consistent in-memory DB from the log and the most recent backup DB copy in the case of the system failure.

This paper assumes that the primary DB consists of a set of fixed-length in-memory pages, and each page consists of a fixed number of slots. The slot length is identical in a single page but may differ among pages. This is to say that the slot is the basic unit of logging as well as storage management. In this slotted page scheme, variable-length records are handled by linking multiple slots [13]. We assume the slotted page scheme just for simplicity, but our discussion in this paper is also valid for the heap scheme, which allocates an arbitrary size of memory [7].

2.2. Logging, Checkpointing, and Restart

To guarantee the durability of the committed transactions, all of the redo log information (the "after image" in the physical logging) is flushed to the disk before writing the commit record. The undo log information (the "before image" in the physical logging) is also created before every update to prepare for aborting the transaction. On the transaction abort, the compensation log record (CLR) is generated to facilitate the undo of the aborted transaction during post-crash restart [9].

To allow normal transactions to run simultaneously during the checkpoint, we focus on the fuzzy checkpoint policy [5]. Since an inconsistent backup copy can be made with this policy, the undo log is required to be flushed to the disk. However, the WAL (Write Ahead Log) protocol of DRDB, which requires flushing the undo log information to the disk before writing a data page on disk, is not necessary for the MMDB. Instead, the undo log is flushed in bulk just before finishing the checkpoint. If the system crashes during checkpointing to a backup copy, the database can be recovered from the other backup copy.

With the physical logging scheme, the above discussion translates to forcing both the before and after images to be flushed to the log before the commit, abort, or checkpoint [9]. In addition, during post-crash restart, the log records for the same resource have to be applied by the serialization order. The well-known ARIES-based recovery algorithm proceeds in the following order:

1. Read the most recent backup copy into the primary DB.
2. Redo by scanning the log forward from the beginning of the most recent checkpoint.
3. Undo by scanning the log backward for the loser transactions that were active at the time of crash.

2.3. Parallel Logging and Restart

Since the log write and read is the dominant bottleneck in MMDBMS, it has been suggested to use multiple log disks

[2]. However, using multiple log disks in parallel has several problems to be solved, which will be discussed in this section. To the best of our knowledge, these problems have not been studied.

We first list two desirable restart properties for parallel logging schemes. First, if possible, it is desirable to avoid the cost of merging logs by the serialization order during the post-crash restart. When the physical or the logical logging scheme is used, log records for the same resource have to be replayed by the serialization order [9][3]. Otherwise, the consistent state of the database cannot be recovered. Second, it is desirable to distribute the processing of log records in parallel to multiple CPUs. This is because the log transfer rate may exceed the processing rate of a single CPU when multiple log disks are used.

These two properties, desirable to reduce the restart time, can be achieved by partitioning log records by the resource. For example, consider partitioning the database into several segments and storing the log records for each segment to a different disk [3]. Then, each log can be replayed in parallel. However, this resource-based partitioning scheme has the following three problems.

1. When updates are skewed to certain segments, the disks for other segments are not fully utilized.
2. Suppose that the system crashes in the middle of committing a transaction that updated multiple segments. Since the log records are written to multiple disks simultaneously, the records for a certain segment might not be written although the commit record has been already written to a disk. In this case, it is impossible to recover the consistent database. To avoid this problem, the commit record can be written to all the disks. However, this complicates the task of determining whether a transaction committed or not during restart.
3. Suppose that there are many transactions that update multiple segments. Since log records are flushed to the disk before committing a transaction, such a transaction incurs multiple disk I/Os, one for each segment. This leads to a significant decrease of transaction throughput, compared with the case where each transaction incurs only one disk I/O.

Our proposal in this paper is to use the differential logging scheme. Since the redo and undo operations of the differential logging scheme are commutative and associative, those two desirable restart properties are automatically achieved. Therefore, we can freely distribute log records to multiple disks such that the above problems with the resource-based partitioning scheme do not occur. One such a distribution scheme is the transaction-based partitioning, which will be described in section 4.

In addition to the logging, checkpointing can be also performed in parallel using multiple disks [5]. This paper assumes that the backup database is partitioned into several disks.

3. The Differential Logging

3.1. Definitions and Properties

Definition 1 (Differential Log) Assume that a transaction changes the value p of a slot to q. Then, the corresponding differential log $\Delta(p, q)$ is defined as $p \oplus q$, where $\oplus$ denotes the bit-wise XOR operation.

Definition 2 (Redo and Undo)
Using the differential logging, the redo and undo operations are defined as $p \oplus \Delta(p, q)$ and $q \oplus \Delta(p, q)$, respectively, where p is the before image and q is the after image.

For example, consider that p is 0010 and q is 1100. Then, the corresponding differential log is 1110 (= 0010⊕1100). The redo 0010⊕1110(i.e. $p \oplus \Delta$) recovers q correctly, and the undo 1100⊕1110 (i.e. $q \oplus \Delta$) recovers p correctly. To ease the discussion of the differential logging, we first list the properties of bit-wise XOR operator.

1. Existence of identity: $p \oplus 0 = p$
2. Existence of inverse: $p \oplus p = 0$
3. Commutative: $p \oplus q = q \oplus p$
4. Associative: $(p \oplus q) \oplus r = p \oplus (q \oplus r)$

Theorem 1 (Recoverability) Assume that the value of a slot has changed from b_0 to b_m by m updates and each update u_i ($i = 1, \ldots, m$) has generated the differential log $\Delta(b_{i-1}, b_i)$. Then, the final value b_m can be recovered from the initial value b_0 and the differential log, and the initial value can be recovered from the final value and the differential log.

Proof:
i) Recovery of b_m from b_0:

$b_0 \oplus \Delta(b_0, b_1) \oplus \Delta(b_1, b_2) \oplus \ldots \oplus \Delta(b_{m-1}, b_m)$
$= b_0 \oplus (b_0 \oplus b_1) \oplus (b_1 \oplus b_2) \oplus \ldots \oplus (b_{m-1} \oplus b_m)$
$= (b_0 \oplus b_0) \oplus (b_1 \oplus b_1) \oplus \ldots \oplus b_m = 0 \oplus \ldots \oplus 0 \oplus b_m = b_m$

ii) Recovery of b_0 from b_m:

$b_m \oplus (b_{m-1} \oplus b_m) \oplus \ldots \oplus (b_0 \oplus b_1)$
$= b_m \oplus (b_m \oplus b_{m-1}) \oplus \ldots \oplus (b_1 \oplus b_0) = \ldots = b_0$ ■

Theorem 2 (Order-Independence of Redo and Undo) Given the initial value b_0 of a slot and differential logs Δ_i, $i = 1, \ldots, m$, where some of Δ_i may be undo logs, , the final value b_m can be recovered applying the differential logs in an arbitrary order.

Proof:
Assume that differential logs are applied in the order of $\Delta_{k(1)}$, $\Delta_{k(2)}, \ldots, \Delta_{k(m)}$, where $k(i) \in \{1, 2, \ldots, m\}$ and $k(i) \neq k(j)$ iff $i \neq j$ for all i and j. Then, the final value of the slot is

$b0 \oplus \Delta_{k(1)} \oplus \Delta_{k(2)} \oplus \ldots \oplus \Delta_{k(m)}$

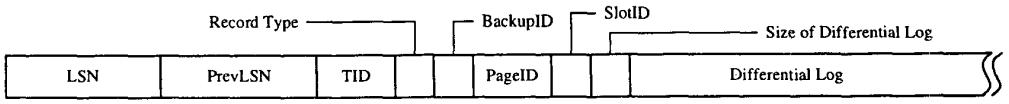

Figure 2. Structure of differential log records

Since $\oplus$ is commutative, we can change the order of applying $\Delta_{k(i)}$ into the following sequence.

$b0 \oplus \Delta_1 \oplus \Delta_2 \oplus ... \oplus \Delta_m$

This completes the proof. ■

Compared with the physical logging scheme, the differential logging has the following nice properties that offer room for significant improvement of the logging and the restart performance in the MMDBMS.

- The log records can be processed in an arbitrary order. This means that we can freely distribute log records to multiple disks to improve the logging and the restart performance. The restart techniques to get unrestricted parallelism will be given in section 4.
- Redo and undo operations can be mixed in an arbitrary order. This means that two separate passes of redo and undo during restart is not necessary. We develop a one-pass restart algorithm in section 3.4.
- Compared with the physical logging, the log volume is reduced to almost half since the redo log and the undo log are same. Thus, both the time to write the log during the normal operation and the time to read the log during the restart decrease.

3.2. Logging

Figure 2 shows the structure of a differential log record. The LSN (log sequence number) represents the order of log record creation. In our implementation, we use the physical location of a log record on disk as its LSN. The PrevLSN is used to chain the log records of a transaction for fast backward traversal. The TID represents the transaction identifier that created the log record. There are six record types: *begin*, *abort*, *commit*, *DL*, *begin_checkpoint*, *end_checkpoint*. The first three are used to record the beginning and the end of a transaction. Only log records of the *DL* type have the remaining five fields, whose meaning is self-evident except BackupID, which indicates the more recent one of the two backup DBs. We will explain its usage in section 3.3 and section 3.4.

When updating a slot, the transaction generates a log record of the DL type by applying the bit-wise XOR operation to the before and the after images. Then, the record is appended to the log buffer, which is flushed to the log disk when the buffer overflows or the transaction commits.

3.3. Checkpointing

The differential logging can be used with any of the fuzzy checkpointing, the action consistent checkpointing, and the transaction consistent checkpointing. When making consistent checkpoints, the existing checkpointing algorithms for the physical logging can be used directly. However, when making a fuzzy checkpoint, we have to deal with two synchronization problems.

- During checkpointing, a dirty page should not be copied into a backup DB while a transaction is updating a slot in the page. Otherwise, a mixture of the before and the after images may be copied, and then the page cannot be recovered from a crash.
- Since the XOR operation used for redo and undo is not idempotent, we have to know whether the update recorded in a certain log record has been reflected to a backup. If so, the record should not be replayed when rolling the corresponding transaction forward.

To handle the first problem, we use the simplest locking primitive, the so-called mutex, so that the checkpointer and update transactions may have to acquire the mutex for a page before copying or updating. And, to deal with the second problem, we use a flag named BackupID. Each page has a BackupID field in its header, and this field is copied into each differential log record. This field of a page is toggled after flushing the page to the appropriate backup DB during checkpointing.

The corresponding update and fuzzy checkpointing algorithms are presented in Algorithms 1 and 2. Algorithm 1 is used by a transaction when updating a slot. After checkpointing, Algorithm 2 records the LSN of *end_checkpoint* record in the log anchor, which keeps the active status of log, so that the *end_checkpoint* record can be located quickly for restart.

Algorithm 1. Update

```
1. Generate the corresponding differential log
   record.
2. If the global checkpoint flag is set, ac-
   quire the mutex for the page.
3. Copy the BackupID flag stored in the page's
   header into the BackupID field of the log
   record.
4. Update the page.
5. Release the mutex if it was acquired previ-
   ously.
6. Append the log record to the log buffer.
```

Algorithm 2. FuzzyCheckpoint

```
1. Begin with the following.
   A. Set the global checkpoint flag.
   B. Create a begin_checkpoint record and ap-
```

pend it to the log buffer.
C. Choose the backup DB that was the least recently checkpointed as the current backup DB.
2. Scanning the database page by page, do the following for each page.
A. Acquire the mutex for the page.
B. Toggle the BackupID flag in the page's header.
C. If the dirty bit in the page header that corresponds to the current backup DB is set, copy the page asynchronously into the current backup DB.
D. Release the mutex
3. Finish by doing the following.
A. Reset the global checkpoint flag.
B. Wait until all the asynchronous I/Os complete.
C. Create an end_checkpoint record containing active transaction IDs, and append it to the log buffer.
D. Flush the log buffer.

3.4. Restart

When the differential logging is used, the forward scan of log is not mandatory since the redo operation is commutative and associative. If we scan the log backward, we encounter a commit or an abort record of a transaction before other records of the transaction. Thus, we can skip the records of aborted transactions and loser transactions. On the other hand, when scanning forward, all the log records are usually replayed presuming the commit because the abort ratio is usually low. This strategy needs the separate undo pass to roll back loser transactions. The undo pass scans the log backward from the end of the log, and it may incur I/Os if some of the log data of loser transactions have been flushed out of memory. Although the differential logging can also be used with the two-pass restart strategy, we present only the backward one-pass restart algorithm due to the space limitation.

When scanning backward, two special cases need to be handled. One is the transaction that was active at the time of crash, and the other is the transaction that aborted after checkpointing. Since there is no commit or abort record for the first type of transactions, we should skip the log records that appear without a corresponding commit or abort record. Since the pages updated by the second type of transactions might have been copied into a backup DB in an inconsistent state, we need to roll back the reflected updates by those transactions. To identify these two types of transactions, two tables CTT(committed transaction table) and ATT (aborted transaction table) are maintained. The detailed algorithm is given in Algorithm 3.

Algorithm 3. One-Pass Restart

1. Read the current backup DB ID from the log anchor.
2. Read the current backup DB, and reconstruct the primary DB as of the checkpointing.
3. Initialize CTT as empty.
4. Scanning the log backward from the end of log until the end_checkpoint record is encountered, do the following depending on the record type.
A. Begin record: remove the corresponding entry from CTT
B. Commit record: create a corresponding entry in CTT
C. Abort record: ignore
D. DL record: if TID is in CTT, redo. Otherwise, ignore it
5. Initialize ATT with TIDs in the end_checkpoint record. Then, let ATT be ATT - CTT
6. Continue scanning the log backward to the begin_checkpoint record doing the following depending on the log record type.
A. Begin record: remove the corresponding entry in either of CTT or ATT
B. Commit record: create a corresponding entry in CTT
C. Abort record: create a corresponding entry in ATT
D. DL record
i. If TID is in CTT and the BackupIDs in the log record and the corresponding page header are same, redo.
ii. If TID is not in CTT and the BackupIDs in the log record and the corresponding page header are different, undo.
iii. Otherwise, ignore the current log record.
7. Undo for all the transactions remained in ATT scanning the log backward.

Since the one-pass restart algorithm just ignores the log records of aborted transactions, not only compensation log records don't have to be made when a transaction aborts but also log records don't have to be flushed before aborting. To facilitate the backward scan by the one-pass restart algorithm, we store the log header after the log body. An alternative is to append the size of log body at the end of each record.

4. Parallel Logging and Restart

4.1. Parallel Logging

For parallel logging, this paper proposes to partition log records based on the transaction ID. Figure 3 shows the logging system architecture with multiple instantiations of the log manager. When a transaction begins, the transaction manager inserts a record in the transaction table. Its LogMgrID field is filled with the identifier of the idlest log

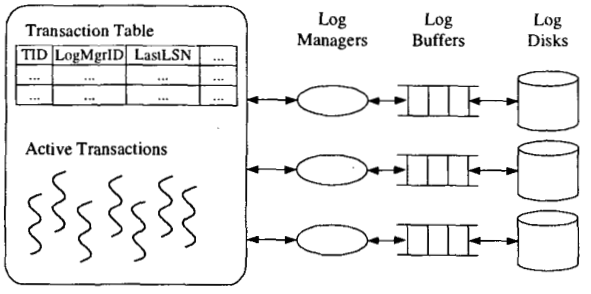

Figure 3. Transaction-partitioned parallel logging

manager, which has the smallest number of log pages to flush. Then, subsequent log records of the transaction are directed to that log manager. The LastLSN is initialized with 0. Each log manager has its own log buffer and log disk.

4.2. Parallel Restart

Figure 4 shows the parallel restart architecture. The restart process is performed by two kinds of agents, BL (backup DB loader) and LL (log loader), which are instantiated and coordinated by the recovery manager. The recovery manager instantiates as many BLs and LLs as the number of backup DB partitions and log partitions, respectively. The BL does two jobs, BR (backup DB read) and BP (backup DB play), and the LL does LR and LP. The BR and the LR are I/O jobs while the BP and the LP are CPU jobs.

Basically, these four jobs are performed sequentially, BR $\rightarrow$ BP $\rightarrow$ LR $\rightarrow$ LP, but they also can be performed in parallel. This section presents four levels of parallel restart schemes.

Parallel Restart Level 1 In this base level of parallelism, multiple BLs run simultaneously, and multiple LLs run simultaneously, but LLs can start only after all the BLs finished their jobs.

Since each BL updates a different part of the primary DB, no synchronization is needed among BLs. However, LLs need to be synchronized so that they should not apply log records for the same resource simultaneously. Such synchronization can be done inexpensively by acquiring the mutex for a page as described in section 3.3. Furthermore, since log records are partitioned by the transaction, the transactions tables used in Algorithms 3 don't have to be shared among LLs.

Parallel Restart Level 2 A simple additional parallelism is to use the pipeline. When the asynchronous I/O is supported, each of BLs and LLs can do their I/O and the CPU jobs simultaneously using a pipeline.

Parallel Restart Level 3 When the backup database and log records are stored in different disks, it is possible to do

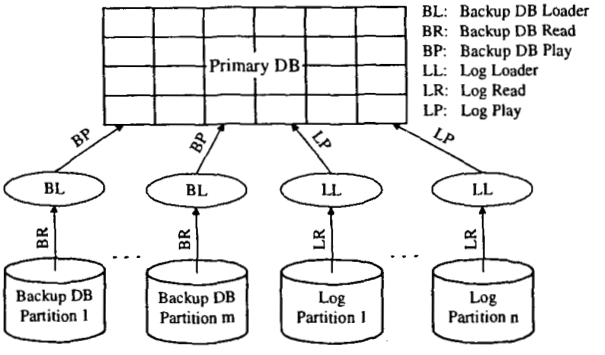

Figure 4. Parallel restart

the BR and the LR simultaneously if the I/O bandwidth of the system allows. However, since the read log records cannot be applied until BLs finish the BR and BP, log records are just piled up in memory. Thus, this level of parallelism is useful when there is enough memory to keep a large number of log records.

Parallel Restart Level 4 The final level of parallelism is to do the BP and the LP simultaneously, i.e., applying log records while copying a backup DB into the primary DB. Although it seems impossible, the differential logging allows it with the following two modifications.

- The BL applies the bit-wise XOR operator to two pages: one in the primary database and the other read from the backup database. Therefore, both the BP and the LP are done using the XOR operator. Of course, the BL has to acquire the mutex before updating a page.
- Before the recovery manager instantiates BLs and LLs, it clears the primary database such that all pages have 0s.

Theorem 3. The restart scheme using the Level 4 parallelism recovers the primary database correctly from the crash.

Proof: Due to the space limitation, we omit the rigorous proof, and give a sketch of it. Since a page in the primary database is initially filled with 0s, applying the XOR to the page and the corresponding page read from the backup DB is equivalent to copying from the backup DB to the primary DB. Since the XOR operator is commutative, BP and LP

Level	Parallelism	Restart Time
1	{BR}$\rightarrow${BP}$\rightarrow${LR}$\rightarrow${LP}	$T_{BR} + T_{BP} + T_{LR} + T_{LP}$
2	{BR | BP}$\rightarrow${LR | LP}	$\max(T_{BR}, T_{BP}) + \max(T_{LR}, T_{LP})$
3	({BR | BP} $\|$ {LR})$\rightarrow${LP}	$\max(T_{BR}, T_{BP}, T_{LR}) + T_{LP}$
4	{BR | BP} $\|$ {LR | LP}	$\max(T_{BR}, T_{BP}, T_{LR}, T_{LP})$

Table 1. Four levels of parallelism in restart
({}: multiple instantiation, |: pipelined parallel execution, "$\|$": parallel execution)

can be done in an arbitrary order. ■

Table 1 compares these four levels of parallelism. We assume that the higher level of parallelism includes all the lower levels of parallelism. For example, all the four levels of parallelism are used in the level 4.

5. Experimental Evaluation

5.1. Experimental Setup

To show the practical impact of the differential logging, we implemented the proposed recovery schemes in our main-memory DBMS kernel called P*TIME [1]. Like its predecessor XMAS [12][13], this system is implemented in C++ to facilitate the object-oriented extension, and supports multithreaded processing for scalable performance on multiprocessor platforms.

Table 2 summarizes three implemented parallel recovery schemes. PL2P combines the physical logging and the two-pass restart algorithm. DL2P combines the differential logging and the two-pass restart algorithm. DL1P combines the differential logging and the one-pass restart algorithm. All of these schemes support pipelining between LR and LP during log processing in restart. When restarting with more than two log disks, PL2P reads log records from log disks in parallel, merge them by the serialization order, and replays them in that order. Therefore, PL2P does the I/O job in parallel but it does the CPU job sequentially. In contrast, DL schemes (DL2P and DL1P) also do the CPU job in parallel. In the experiment, log records are partitioned into one to eight disks using the transaction-based log-partitioning scheme, and the fuzzy checkpointing is used.

The test database is derived from the SMS domain [11], which is one of update-intensive wireless information services. The size of the primary database is roughly 250 MB, which includes 1 million records. For the controlled experiment, we made a simplified SMS message table with three fields: message id, destination address, and message. The message id is a 4-byte integer. The destination address is a 12-byte character array, which consists of a MIN (mobile identification number), the type of MIN, and so on. The message is a 240-byte character array. Thus, the total record or slot size is 256 bytes.

We mixed two types of update transactions: one inserts 2 messages into the table and the other removes 2 messages from the table. These transaction types mock the message receiver and the flusher processes of the SMS system, respectively. The former receives message through the SS#7 network and inserts them, and the latter moves sent messages to the archive in order to keep the MMDB from exploding.

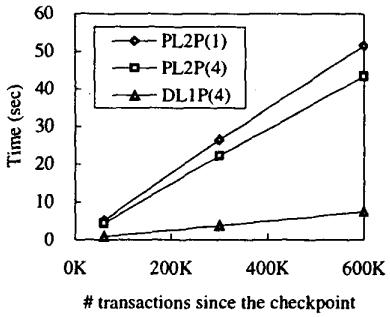

Figure 5. Log processing time with the log volume and log disks

The experiment is conducted on a Compaq ML 570 server with four 700 MHz Xeon CPUs, 6GB RAM, and dozen disks. It has a single I/O channel, which can support up to 160MB/sec. The disks are of the same model. Their average seek time is 6.3 ms, average latency time is 4.17ms, and maximum sustained transfer rate is 29 MB/sec.

5.2. Log Processing Time

As we analyzed in the section 4.2, the restart time is broken down to the backup DB loading time and the log processing time. The first is dominated by the disk access and is independent of the logging scheme. Therefore, in identifying the impact of the differential logging, we first focus on the measurement of the latter varying the volume of log, the number of log disks, and the abort ratio. In the measurement, we expect that:

- DL schemes reduce the total log volume by almost half compared with PL2P, and thus will accelerate the log processing even if a single log disk is used.
- The speedup ratio of DL schemes over PL2P remains constant despite the changes in the number of update transactions since the last checkpoint.
- The benefit of increasing the number of log disks is higher in DL schemes than in PL2P in the multiprocessor environment.

	Logging scheme	Exploited parallelism during restart	Number of passes in restart	Number of log disks	Checkpointing scheme
PL2P	Physical	Only log reading	Two	1 ~ 8	Fuzzy
DL2P	Differential	Log reading and log replaying			
DL1P			One		

Table 2. Tested recovery schemes

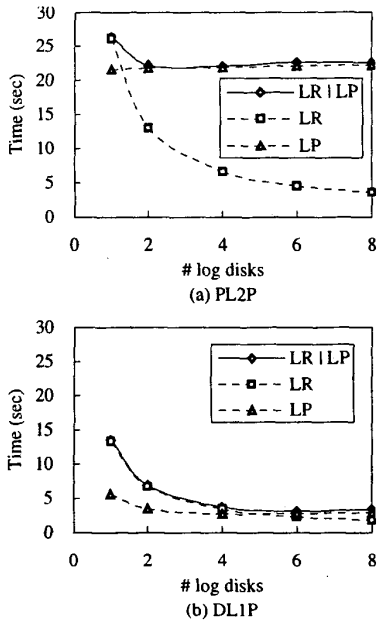

Figure 6. Breakdown of log processing time

- DL1P will outperform DL2P as the abort ratio increases.

Figure 5 shows the log processing time measured by varying the number of update transactions since the last checkpoint. Note that this number determines the total log volume to process during restart. When it is 600K, the log volume is 721MB for PL2P, 377MB for DL2P and 370MB for DL1P. The abort ratio was fixed at 2%. The numbers in the parentheses represent the number of log disks. DL1P(4) is about six times faster than PL2P(4) and about seven times faster than PL2P(1), regardless of the total log volume.

Figure 6 shows the log processing time of PL2P and DL1P (with 600K update transactions since the last checkpoint) by varying the number of log disks. For DL1P, it decreases almost linearly with the number of disks until it reaches a saturation point. On the other hand, for PL2P, the log processing time first decreases linearly and then increases slowly after a certain point. To show where the performance difference comes, two components of the log processing time are shown with broken lines: the I/O time to read log records from disk (LR time) and the CPU time to replay the log (LP time). For both PL2P and DL1P, the I/O time decreases almost linearly with the number of disks. DL1P takes 47% less I/O time than PL2P because of the difference in the total log volume. Note that PL2P is dominated by the LP time when more than two log disks are used, while DL1P is dominated by the LR time regardless of the number of disks. This is because the LP step of PL2P requires merging of logs from different disks by the serialization order.

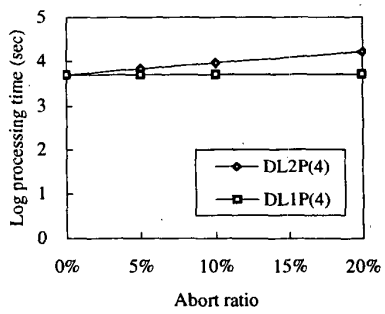

Figure 7. Comparison of DL2P and DL1P with the abort ratio

To compare DL1P with DL2P, we measured the log processing time varying the abort ratio from 0% to 20%. Figure 7 shows the result when four log disks are used and 600K transactions had been performed since the last checkpoint. This result shows that DL1P outperforms DL2P by about 1.15 times when the abort ratio is 20%. This improvement is mainly attributed to the difference in the log volume. Since DL1P does not make compensation log records when a transaction aborts, the log volume of DL1P remains constant regardless of the abort ratio. DL1P is also faster than DL2P in terms of the CPU time because DL1P does not replay the log records of aborted transactions while DL2P does the log records of both the committed and the aborted transactions.

5.3. Total Recovery Time

To see the impact of fully parallel restart based on the differential logging, we measured the restart time for the following recovery schemes:

- PL2P-Seq: PL2P observing the parallelism level 2 rule of section 4. Namely, the backup DB loading and the log processing are done in sequence while the pipelining is allowed in each of these two steps.
- DL1P-Seq: the DL1P version of PL2P-Seq.
- DL1P-Para: DL1P observing the parallelism level 4 rule of section 4. The backup DB processing is intermixed with the log processing.

Figure 8 shows the measured restart time when the backup DB size is 250MB and the processed log volume is 370MB for DL1P and 721MB for PL2P. Figure 8(a) shows the result with two log disks and two backup DB disks and Figure 8(b) with four log disks and four backup DB disks. Here are the observations drawn from this figure.

- As in the log processing time, the backup DB loading time also decreases by half as the number of disks doubles.
- DL1P-Seq outperforms PL2P-Seq significantly, by about two times in Figure 8(a) and by about three times in

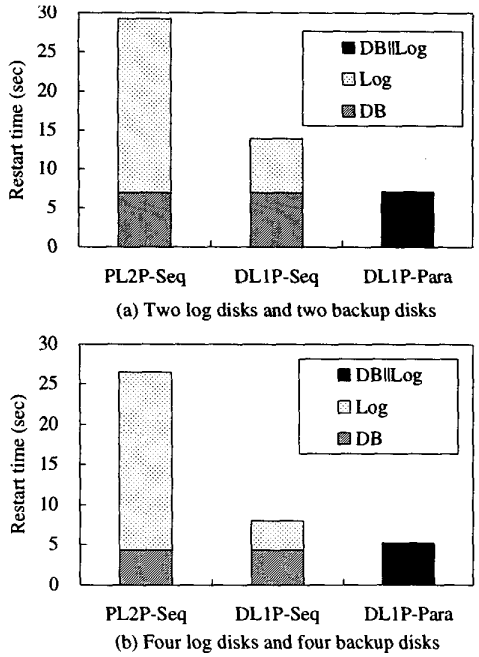

Figure 8. Breakdown of total restart time (the symbol || denotes the parallel processing)

Figure 8(b). The difference comes mainly from the log processing time while the backup DB loading time remains the same.

- The parallelism further decreases the total restart time. DL1P-Para outperforms DL1P-Seq by about two times in both of Figure 8(a) and Figure 8(b).

5.4. Transaction Throughput

In this series of experiment, we measured the logging performance varying the number of log disks and the size of a slot.

Figure 9(a) shows the throughput in TPS (transactions per second) with the number of disks. In this measurement, the abort ratio is 2%. This result shows that the logging performance of DL1P and PL2P increases in proportion to the number of disks. DL1P(8) outperforms PL2P(1) by 9.6 times. Even if the same number of log disks are used, DL1P processes over 1.8 times as many transactions as PL2P during the same time interval.

Figure 9(b) shows the throughput measured varying the slot size. As the slot size increases, the overhead ratio of log header decreases and the ratio of DL1P log volume to PL2P log volume approaches 0.5. As result, the performance gain of DL1P(4) over PL2P(4) approaches two. Note that the performance gain of DL1P(4) over PL2P(1)is about six.

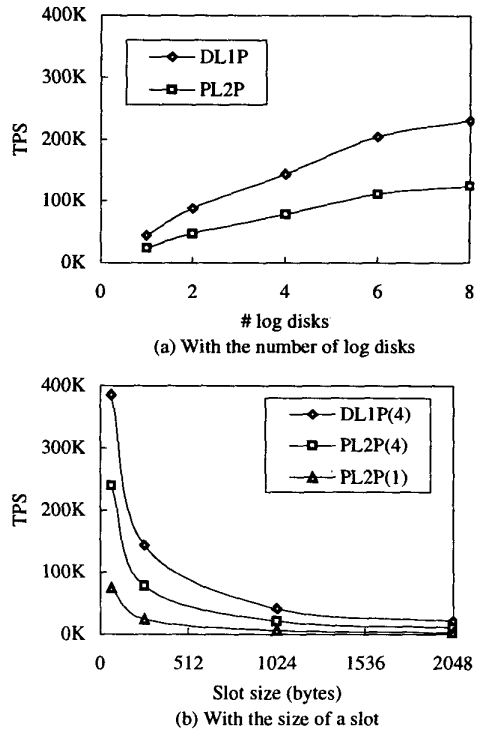

Figure 9. Logging performance with the number of log disks and the size of data objects

6. Related Work

The idea of using the XOR-based logging has been addressed in the DRDBMS context under the name of transition logging [6]. However, if the normal update and recovery operations are applied directly to the disk database with only the XOR difference on the log disk and if the system crashes in the middle of such operations, it is impossible to recover the consistent database state. The shadow page scheme has been considered for each disk update to solve this problem, but it has been abandoned because of its high run-time overhead. Fortunately, the MMDBMS does not have this problem because it applies the update and recovery operations to the main memory copy of the backup database on the disk.

In order to reduce the log write and recovery time, the redo-only logging has been studied in the MMDB context [2][5][8]. By keeping the log of uncommitted, active transactions in memory, it is possible to write only the redo part of the log on disk when a transaction commits. Thus, the disk-resident log volume is reduced to almost half as in the differential logging. However, the differential logging has at least the following three advantages over the redo-only logging. First, it permits the unrestricted parallelism in recovery as mentioned before. Second, it can be used with the

fuzzy checkpointing while the redo-only logging can be used only with the consistent checkpointing schemes [8]. Third, it does not need the memory space to keep the log of active transactions, whose volume can grow substantially when long transactions are involved. Although we compared the differential logging with the redo-only logging, they are basically orthogonal and can be used together. However, the only benefit from such combination is not to write the log data of aborted transactions while the fuzzy checkpointing becomes impossible.

The transaction-partitioned parallel logging and recovery has been studied in the shared disk architecture [10]. This scheme, named ARIES/SD, is similar to the PL2P physical logging scheme introduced in section 5. The difference is that ARIES/SD logs at the page level while the PL2P logs at the slot level.

7. Conclusion

This paper studied the problem of parallel MMDB logging and recovery to increase the transaction processing performance and to reduce the recovery time exploiting the low-cost multiprocessor platforms and commodity disks in the market.

The novel contribution of this paper is to introduce the so-called differential logging for parallel MMDB logging and recovery. Using the bit-wise XOR operation both to compute the log and to recover the consistent state from the log, the differential logging permits unrestricted parallelism in MMDB logging and recovery. In addition, because the log is the XOR difference between the before and after images, the log volume is reduced to almost half compared with the physical logging. We assumed the transaction-based partitioning of log for distribution to multiple log disks and presented two versions of the parallel recovery scheme based on the differential logging. The two-pass version scans the log twice for redo and undo as in the well-known ARIES algorithm, and the one-pass version scans the log only once taking advantage of the nice properties of the differential logging.

An experimental study of comparing the differential logging and the physical logging shows that the former outperforms the latter significantly both in the update transaction throughput and in the recovery time. It is also shown that since the differential logging permits interleaving of the log replaying with the backup DB replaying during the restart, the backup DB loading and the log processing can proceed in parallel to reduce the total recovery time. Such a level of parallelism is unthinkable in the MMDBMS based on the traditional logging schemes or in the context of DRDBMS.

In this paper, we have conducted our experiment in the 4-CPU shared-memory multiprocessor environment. In the future, we plan to study the benefit of the differential logging in different types of parallel computing environment. We also plan to verify the benefit of the differential logging by applying it to the real environment such as e-commerce and mobile value-added information services.

Acknowledgement

This research has been in part supported by the research contract with the SK Telecom of Korea on the feasibility study of the main-memory database for mobile value-added information services and the Brain Korea 21 program sponsored by the Korean ministry of education.

References

[1] S. K. Cha, *et. al.*, "P*TIME: A Highly Parallel Main Memory DBMS Based on Differential Logging," submitted to ACM SIGMOD 2001 Conference for demo.

[2] D. J. DeWitt, *et. al.*, "Implementation Techniques for Main Memory Database Systems," *Proceedings of ACM SIGMOD Conference*, pages 1-8, 1984.

[3] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, 2nd ed., Morgan Kaufmann, 1993.

[4] H. Garcia-Molina and K. Salem, "Main Memory Database Systems: An Overview," *IEEE Transactions on Knowledge and Data Engineering*, 4(6), pages 509-516, 1992.

[5] R. B. Hagmann, "A Crash Recovery Scheme for a Memory-Resident Database Systems," *IEEE Transactions on Computers*, C-35(9), pages 839-843, 1986.

[6] T. Haerder and A. Reuter, "Principles of Transactions-Oriented Database Recovery," *ACM Computing Surveys*, 15(4), pages 287-317, 1983.

[7] H. V. Jagadish, D. Lieuwen, R. Rastogi, and A. Silberschatz, "Dali: A High Performance Main Memory Storage Manager," *Proceedings of VLDB Conference*, pages 48-59, 1994.

[8] H. V. Jagadish, A. Silberschatz and S. Sudarshan, "Recovering from Main-Memory Lapses," *Proceedings of VLDB Conference*, pages 391-404, 1993.

[9] C. Mohan, D. Haderle, B. G. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," ACM *Transactions on Database Systems*, 17(1), pages 94-162, 1992.

[10] C. Mohan and I. Narang, "Data Base Recovery in Shared Disks and Client-Server Architectures," *Proceedings of International Conference on Distributed Computing Systems*, pages 310-317, 1992.

[11] Mobile Lifestreams, "An Introduction to the Short Message Services," White paper, http://www.mobilesms.com/wp/wp2.htm

[12] J. H. Park, Y. Sik Kwon, K. Kim, S. Lee, B. D. Park, and S. K. Cha, "Xmas: An Extensible Main-Memory Storage System for High-Performance Applications," *Proceedings of ACM SIGMOD Conference*, pages 578-580, 1998.

[13] J. H. Park, K. Kim, S. K. Cha, M. S. Song, S. Lee, and J. Lee, "A High-performance Spatial Storage System Based on Main-Memory Database Architecture," *Proceedings of DEXA Conference*, pages 1066-1075, 1999.