Enhancing Real-Time DBMS Performance with Multiversion Data and Priority Based Disk Scheduling

Woosaeng Kim, Jaideep Srivastava
Computer Science Department
University of Minnesota
Minneapolis, MN 55455

ABSTRACT

Databases are increasingly being used in real-time applications where timeliness of result is part of the correctness criterion. Time constrained CPU scheduling is one of the critical issues for real-time systems. For a real-time database system (RTDBS) this idea must be extended to concurrency control and disk scheduling. This paper proposes real-time multiversion concurrency control algorithms to fulfill following goals: i) increase concurrency, ii) adjust the serialization order dynamically, iii) work without a priori estimate of a transaction’s runtime. We also propose new disk scheduling algorithms which consider not only the transactions which request I/O but also those affected by I/O. Real-time multiversion concurrency control and disk scheduling algorithms are shown to decrease miss ratio significantly.

1. Introduction

Real-time systems are defined as those systems in which the correctness of computation depends not only on the logical result of the computation, but also on its timeliness. With ever increasing volume of information being handled by real-time systems, these systems are now being equipped with databases for efficient manipulation of information. Such database systems are called Real-Time Database Systems (RTDBSs). Some example applications are program trading, computer aided manufacturing, process control, military command and control, and radar tracking systems. In each case not meeting the timing constraints can lead to substantial penalty.

An important characteristic of databases is their data oriented nature of computation. Thus, data integrity and consistency is of prime concern. Traditional database resource and transaction scheduling has focused on optimizing system performance in an average sense, i.e. minimize average response time and maximize average throughput, while maintaining data consistency and integrity. Traditional work in real-time scheduling, on the other hand, has focused on the timing needs of individual computations, but with little attention on data consistency [5]. In a RTDBS both needs are present.

There has been increasing research interest in RTDBSs, especially in the critical area of resource scheduling. Several real-time concurrency control algorithm have been proposed which give priority to urgent transactions. These are based on either locking [19,11,15], timestamp [11], or optimistic method [8]. These algorithms use blocking or rollback to control the interleaving of transactions. Therefore, the serialization orders produced by these algorithms are bound to the past execution history with no flexibility. For example, when a higher priority transaction requests a conflicting lock which is being held by another lower priority transaction, the only choices are either aborting the lower priority one or letting the higher priority one wait for the lower priority one. However, neither choice may be desirable. Recently, Lin and Son [12] proposed a new concurrency control algorithm which is based on both locking and optimistic methods to adjust the serialization order dynamically as much as possible in favor of more urgent transactions.

In this paper, we propose new multiversion concurrency control algorithms to fulfill the following goals: i) increase concurrency, ii) adjust the serialization order dynamically, iii) work without a priori estimate of a transaction’s runtime. The benefit of multiple versions is to reduce the transaction rejection and thus to increase the degree of concurrency. Several concurrency control algorithms based on multiple versions of data have been reported in traditional database literature [3,4]. However, until now no study has been reported evaluating this approach in the RTDBS environment. Maintaining
multiple versions may not add much to the cost of
concurrency control, because the versions may be needed
anyway by the recovery algorithm [3]. The performance
of the real-time multiversion algorithms proposed here
are compared with conventional multiversion as well as
single version real-time concurrency control algorithms.

Disk scheduling is an important factor for databases
because I/O time usually dominates the response time of
a transaction. Several real-time disk scheduling algo-
rithms are proposed in the database environment. Carey,
et al proposed a priority based disk scheduling algorithm
in which disk requests are grouped on the basis of their
priority, and the elevator algorithm is used within each
group [6]. Abbott and Garcia-Molina proposed algo-
rithms which take advantage of the fact that reading
from the disk occurs before a transaction commits while
writing to the disk usually occurs after the transaction
commits [1,2]. In their model, the priorities of I/O
requests are based on both the priority and status of the
transaction which request them. In this paper, when
priorities of I/O requests are assigned, we consider tran-
sactions which are directly affected by these requests in
addition to transactions which generates them.

The remainder of this paper is organized as follows.
In section 2, we describe our real-time disk resident data-
base system model. The multiversion concurrency con-
trol algorithms are described in section 3. The real-time
disk scheduling algorithms and their properties are
described in section 4. The experimental results are
presented in section 5 and conclusions in section 6.

2. Real-time database model and assumptions

In this section we describe our basic system model
and its assumptions. Our model is an event driven real-
time database that is secondary storage resident. The sys-
tem triggers the corresponding actions when an event
occurs, and schedules transactions to meet the timing
requirements. The model has five components: a transac-
tion generator, a CPU scheduler, a concurrency control
scheduler, a buffer manager, and a disk scheduler.

The transaction generator generates new transactions
and provides all necessary information about the transac-
tion such as its deadline and a list of data access seg-
ments. The function of the CPU scheduler is to assign
priorities to ready transactions and allocate the CPU to
the highest priority transaction. The scheduler maintains
two queues, a ready queue and a blocked queue. If a new
transaction arrives, it is inserted into the ready queue. If
the data request cannot be serviced immediately because
of data conflict or buffer miss, the running transaction
is put in the blocked queue until the data request is granted.
When a transaction is committed or becomes tardy, the
CPU Scheduler removes it from the system. The concur-
rency control scheduler controls the relative order of
data access operations of different transactions. It exer-
cises this control by restricting the order of reads, writes,
commits, and aborts of different transactions. Its goal is
to order these operations properly so that the resulting
execution is serializable, timely, and recoverable. The
role of the buffer manager is to move data between the
buffer and disk in response to requests from higher
layers of the system. Finally, the disk scheduler assigns
priority to I/O requests and services them. The interac-
tion of various components of the model is shown in Fig. 1.

![Fig. 1. RTDBS Model](image)

The unit of database granularity we consider is a data
item, i.e. page. A transaction consists of a sequence of
read and write accesses followed by a period of CPU
usage for processing the page. If a page is not found in
the buffer pool, a disk read is initiated. Write requests
are handled similarly except for their disk activity. We
assume that the buffer pool is large enough so that a
transaction does not have to write modified data items to
disk until after committing. We also assume that the tail
of the log is resides in stable main memory.\footnote{Considering the time critical needs of DBMS and the
advancement in main memory DB technology, we believe this assumption to be reasonable.}

It is very difficult to obtain an accurate estimation of
a transaction's execution time, especially for transactions
that have to perform disk access. The only way to arrive
at a deterministic estimation is by assuming that each
disk access will cause a page fault. In reality, this will
not occur and worst-case execution times based on this
assumption may deteriorate the performance by refusing
to schedule many transactions which could have com-
pleted execution [5]. Therefore, we assume that the exe-
cution time of each transaction is not known when
scheduling decisions are made.

Although the main focus of this paper is concurrency
control and disk scheduling, we still need to discuss the
CPU scheduling algorithm used, since it has a very signifi-
cant effect on overall performance [9,11]. In this paper,
we use earliest-deadline-first (EDF) as the underlying
CPU scheduling algorithm. Since it is known to be optimal in the static case[13] and also have better performance than other algorithms in the dynamic case[10]. Table 1 summarizes the notation used in subsequent discussion.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Transaction</td>
</tr>
<tr>
<td>P(T)</td>
<td>Priority of T</td>
</tr>
<tr>
<td>HPT</td>
<td>High priority transaction</td>
</tr>
<tr>
<td>LPT</td>
<td>Low priority transaction</td>
</tr>
<tr>
<td>HPT(R_{(i)})</td>
<td>HPT which requests (holds) a lock</td>
</tr>
<tr>
<td>LPT(R_{(i)})</td>
<td>LPT which requests (holds) a lock</td>
</tr>
</tbody>
</table>

Table 1

3. Multiversion concurrency control

In single version locking algorithms, a write lock (resp. read lock) on a data item \(x\) prevents transactions from obtaining read locks (resp. write locks) on it. We can avoid this locking conflict by using multiple version of \(x\). Since each write produces a new copy of data, writes do not overwrite each other and reads can read old versions. Thus, we can increase the degree of concurrency and reduce rejecting operation. However, conventional locking-based multiversion algorithms cannot be used directly in a real-time environment. Locking-based algorithms suffer from the priority inversion problem which is undesirable in a real-time environment [14]. A priority inversion occurs when a HPT requests and then blocks on a lock which is held by a LPT. It is essential to eliminate or minimize this problem in a RTDBS environment.

3.1. Two version concurrency control

1) Two version 2PL (2V2PL)

Two version 2PL was proposed to avoid read/write lock conflict [3]. This scheme uses three types of locks: read locks, write locks, and certify locks. These locks are governed by the compatibility matrix\(^2\) in Fig. 2.

<table>
<thead>
<tr>
<th>Lock already set by (T_j)</th>
<th>Read</th>
<th>Write</th>
<th>Certify</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Write</td>
<td>Y</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>Certify</td>
<td>N</td>
<td>N</td>
<td>-</td>
</tr>
</tbody>
</table>

Fig. 2

\(^2\) The symbols ‘Y’ and ‘N’ indicate a lock compatibility and a lock conflict, respectively. The symbol ‘-’ means that this type of lock conflict cannot occur.

The scheduler sets read and write locks as usual, i.e. when it processes read and write operations. A transaction only reads the previous, i.e. most recently committed version of \(x\) to avoid cascading aborts. When the scheduler knows that a transaction is about to commit, it converts all of the transaction’s write locks to certify locks. Since certify locks conflict with read locks, the scheduler can do lock conversion only on those data items that have no read locks owned by other transactions. Thus, the effect of certify locks is to delay a transaction’s committing until there are no active readers of data items it is about to overwrite. When the commitment of a transaction is delayed, we say this transaction is in the certify phase.

2) Unconditional two version 2PL (U2V2PL)

In 2V2PL, two different types of priority inversion problems occur, the so called direct and indirect priority inversion. A direct priority inversion occurs when a HPT requests a lock which is held by a LPT in a conflicting mode. An indirect priority inversion, however, can occur even though a transaction requests a lock held in a non-conflicting mode. For example, when a LPT requests a read operation on the data item that has a write lock owned by a HPT, the concurrency control scheduler can successfully set a read lock in 2V2PL. Thereafter, when the HPT tries to commit, it must wait until the LPT is committed or aborted. In the worst case, this can lead to chained waiting. Thus, in this case, an indirect priority inversion occurs when the concurrency control scheduler sets a read lock for a LPT. This indirect priority inversion problem is shown in the diagram below.

A similar phenomenon happens when a HPT requests a write operation on the data item which has a read lock owned by a LPT. Thereafter, when the HPT tries to commit, it must wait until the LPT is committed or aborted. This indirect priority inversion problem is shown in the diagram below.
We propose the U2V2PL protocol to eliminate both direct and indirect priority inversion problems. U2V2PL avoids the direct priority inversion problem by aborting the LPT when a HPT requests a conflicting lock. However, in some situations, the concurrency control scheduler can adjust the serialization order dynamically instead of aborting the LPT in order to avoid the direct priority inversion problem. Suppose a HPT requests a read operation on the data item that has a certify lock owned by a LPT. The HPT can set a read lock without aborting the LPT if the concurrency control scheduler changes the LPT's certify lock to the previous write lock. This solves the priority inversion problem without sacrificing the work done by the LPT. In a conventional locking algorithm such as 2Pl, there is no way of changing the past history except aborting. U2V2PL also avoids the indirect priority inversion problem either by aborting a LPT or refusing a LPT's lock request even though there is no current lock conflict. The interaction of these locks are governed by two separate compatibility matrices, as shown in Fig. 3.

![Fig. 3](image)

When a HPT attempts to set a read lock on data item x, it is always successful. However, if the LPT has already set a certify lock on x, the concurrency control scheduler must convert this certify lock to a write lock in order to adjust the serialization order in favor of the HPT. When a HPT attempts to set a write lock on x, the concurrency control scheduler aborts the LPT in order to prevent direct and indirect priority inversion problems. A HPT can convert its write lock to certify lock without any data conflict at commit time, because a conflicting read of a LPT would either be refused or have been aborted.

When a LPT attempts to set a read lock on x, the concurrency control scheduler delays the request if a HPT already owns a write or certify lock on it in order to prevent the indirect priority inversion problem. If a LPT attempts to set a write lock, the concurrency control scheduler delays the request unless a HPT owns a read lock. The grant of write request to a LPT does not cause problems since it will wait for a HPT at commit time. Since a certify lock conflicts with a read lock, a LPT cannot convert its write locks to certify locks if a HPT has already set a read lock. Notice that since a transaction T_i waits for another transaction T_j only if P(T_i) < P(T_j), this waiting cannot lead to a deadlock.

3) Conditional two version 2PL (C2V2PL)

The priority inversion problem of 2V2PL can be solved by the unconditional abort method of U2V2PL. However, this solution has the problem that even in its final stages a LPT must be aborted and restarted from the beginning. If a LPT has already finished its work, i.e. is in the certify phase, it might be better not to abort the LPT in order to save the work done by it. This does not hamper the execution of HPT much because the LPT which is in the certify phase is expected to finish soon. Thus, in C2V2PL, a HPT waits for the LPT which is in the certify phase. This algorithm is described by the compatibility matrices in Fig. 4.

![Fig. 4](image)

When a HPT attempts to set a read or write lock on data item x, the concurrency control scheduler does not allow it if this causes the LPT in the certify phase to wait for the completion of HPT. In U2V2PL, when a HPT requests a write operation on the data item that has a read lock owned by a LPT, the latter is aborted to prevent indirect priority inversion. However, in C2V2PL this aborting is delayed until the HPT enters the certify phase in order to minimize the aborting overhead as much as possible. It is possible that the LPT can finish before the HPT enters its certify phase. If the HPT attempts to convert its write lock to certify lock, the concurrency control scheduler aborts the conflicting LPT regardless of its phase. This is because the HPT is also in the certify phase. The processing of lock requests when a LPT attempts to set a lock is the same as in U2V2PL.

### 3.2. Multi version concurrency control

1) Multi version 2PL (MV2PL)

The conflict between write locks in 2V2PL is to ensure that only two versions of a data item exist at any time. If we relax the conflict rules so that write locks do not

---

3 The symbol 'Y*' means that the concurrency control scheduler should convert the existing lock type before it sets a new requesting lock of a HPT to adjust the serialization order.

4 If the LPT has waited for the completion of the HPT itself, the latter aborts the former even if the former is in the certify phase to prevent deadlock.

5 The numerator and denominator of symbol 'ab' stands for the result of the lock request when a lock holder is not in the certify phase or in the certify phase, respectively.
not conflict, then a data item may have many uncertified versions (i.e., versions written by active transactions). However, like 2V2PL, we allow a transaction to read only the most recently certified version in order to avoid cascading aborts. The compatibility matrix of MV2PL is shown in Fig. 5.

<table>
<thead>
<tr>
<th>Lock requested by Tj</th>
<th>Read</th>
<th>Write</th>
<th>Certify</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Write</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Certify</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>

Fig. 5.

Since each transaction can have its own version in MV2PL, there is no data conflict between write locks. However, certify locks conflict with not only read locks but also certify locks, because several transactions can set write locks on the same data item. Thus, lock conversion is delayed until read locks or certify locks of the other transactions are released.

2) Unconditional multi version 2PL (UMV2PL)

To avoid the priority inversion problem of MV2PL, the UMV2PL scheduler handles lock requests like the U2V2PL scheduler. This algorithm is described by the compatibility matrices in Fig. 6.

<table>
<thead>
<tr>
<th>Lock already set by Ti</th>
<th>Read</th>
<th>Write</th>
<th>Certify</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Write</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Certify</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>

Fig. 5.

When a HPT attempts to set a read lock on data item x, it is always successful as in U2V2PL. When a HPT attempts to set a write lock on data item x, the concurrency control scheduler can always set this lock unless the lock type of a LPT is read. If so, the LPT is aborted to avoid the indirect priority inversion problem. A HPT can always convert its read locks to certify locks. However, if a LPT has already set a certify lock, the concurrency control scheduler must convert the LPT’s certify lock to a write lock to adjust the serialization order.

When a LPT attempts to set a read lock on x, the concurrency control scheduler delays the request, unless the lock owned by the HPT is a read lock. If a LPT attempts to set a write lock, it is always successful. However, when a LPT attempts to convert its write lock to certify lock, the concurrency control scheduler delays this lock conversion even though the HPT has set a non conflict write lock, in order to prevent the indirect priority inversion problem.

3) Conditional multi version 2PL (CMV2PL)

To save the work done by a transaction in the final stage, we can prevent the LPT which is in the certify phase from being aborted by a HPT. This algorithm is described by the compatibility matrices in Fig. 7.

<table>
<thead>
<tr>
<th>Lock requested by Tj</th>
<th>Read</th>
<th>Write</th>
<th>Certify</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Write</td>
<td>W</td>
<td>YN</td>
<td>Y</td>
</tr>
<tr>
<td>Certify</td>
<td>C</td>
<td>Y</td>
<td>Y*</td>
</tr>
</tbody>
</table>

Fig. 7.

When a HPT attempts to set a read or write lock on data item x, the concurrency control scheduler does not allow it if this will cause the LPT in the certify phase to wait for the completion of HPT. A HPT which requests a write operation on a data item which has a read lock owned by a LPT delays aborting the LPT until the HPT enters the certify phase in order to minimize aborting overhead. The processing of lock requests when a LPT attempts to set a lock is as the same as in U2V2PL, except when a LPT attempts to convert its write lock to certify lock. If the HPT is not in the certify phase, the concurrency control scheduler allows this lock conversion in favor of the LPT which is in its final stage.

4. Disk scheduling

Disk scheduling is an important factor for DBMS performance because I/O time dominates the total computation time of a transaction. Each transaction requests several different I/O types. In our model there are two different types of I/O requests: reads that are issued by unfinished transactions, and writes (flushing their updates back to disk) that are generated by committing transactions. In all of the following schemes, read locks are released when the transaction actually begins to commit or abort, but each write lock are held until after the modified data item is copied into the database.

1) Fixed priority (FP) and Read preference priority (RP)

Among many real-time disk scheduling algorithms proposed, we adopt FP and RP [1] as our base line algorithms because these can be easily adopted to our environment. In FP, the priority of an I/O request is the same as that of the transaction which issued it. In RP, the priority of a read request is the same as that of its issuing transaction, while a write request is assigned the lowest priority. The rationale behind this scheme is that giving

---

6 We assume that the system uses an undo/redo recovery method which can be integrated nicely with the multiversion concurrency control algorithm [1]. In this environment, the committing transaction which finishes its operations can release its write locks after the system flushes the modified data item back to the disk.
high priority to a write may not enhance performance directly because the transaction which issued the write request has already passed the deadline. On the other hand, read requests should have a high priority because all read requests are generated by active transactions which must meet their timing constraints.

2) Dynamic priority (DP)

In the above two schemes, all I/O requests generated by a transaction have the same priority. In DP, I/O requests may have different priorities depending on the situation even though they are generated by the same transaction. As in RP, the priority of a read request is the same as that of the transaction which issued it. However, the priority of a write request depends on whether another transaction is waiting for the release of this write lock. If a transaction is waiting for the release of this write lock, the write request inherits the priority of the waiting transaction. If no transaction is waiting, the write request gets the lowest priority among all the I/O requests in the queue. The rationale behind this policy is that those write requests for which active transactions are waiting should be treated with utmost urgency to unlock active transactions as soon as possible by releasing the corresponding locks. Thus, in this scheme, the disk scheduler considers not only transactions which are waiting for disk service but also transactions which are blocked by data conflict.

4.1. Properties of the disk scheduling algorithms

In this section we show that the disk scheduling policy DP is no worse than any other policy in our environment. We first list our assumptions and state the notation used.

Let the CPU blocked queue, CPU ready queue, and disk queue be BQ, RQ, and DQ, respectively. Let UNBLOCKx represents the fact that transactions become unblocked, i.e. move from BQ to RQ, in order X (A committing transaction should be unblocked later than an active transaction. Thus, we regard former's priority as the lowest priority). Thus, UNBLOCKEDF means Earliest-Deadline-First (EDF) order. Let T^b, T^u, and T^r be the highest priority transactions in BQ+RQ, BQ, and RQ, respectively. Transactions in BQ are unblocked by various events. We use the notation EDFG to represent the CPU scheduling policy which guarantees the dispatch of T^b whenever such an event occurs.

**Lemma 1:** DP guarantees UNBLOCKEDF if BQ consists only of transactions which are either waiting for I/O requests or have data conflict with committing transaction.

**Proof:** Suppose T^b is blocked by an I/O request. T^b is an active transaction which issued a read request. Since all read requests inherit the transaction's priority, the I/O request of T^u will be serviced first. Thus, T^u is the next to become unblocked and UNBLOCKEDF is achieved. Suppose T^b is blocked by data conflict with a committing transaction T^c. Since all read locks held by T^c are released immediately upon commit, T^u must conflict with some write lock of T^c, which is released only on the servicing of the write request of T^c in DQ. In DP, this write request of T^c inherits the priority of T^u, and is the next to be serviced. Thus, T^u is the next to become unblocked and UNBLOCKEDF is achieved.

**Lemma 2:** If UNBLOCKEDF holds then EDFG is the CPU scheduling policy.

**Proof:** T^b is either T^u or T^r. Suppose that T^b is T^u. After UNBLOCKEDF, T^u will be dispatched. Thus, it satisfies the EDFG policy. Alternatively, suppose that T^b is T^r. Regardless of the order of moving transactions from BQ to RQ, T^u will be dispatched. This also satisfies the EDFG policy.

**Claim 3:** CPU scheduling policy EDFG has smaller miss ratio than other CPU scheduling policies.

We have not been successful in proving Claim 3 completely. However, extensive simulation indicates that it is likely to be true. Proving it would ensure that DP is the optimal disk scheduling policy in our environment.

**Lemma 4:** No disk scheduling policy guarantees UNBLOCKEDF if T^u has data conflict with an active transaction.

**Proof:** If T^u conflicts with an active transaction T^c, there does not exist an I/O request in DQ whose servicing can unblock T^u. The status of T^c is one of the following cases:

Case 1 : T^c is in RQ: Since T^c is not waiting for a disk service, no disk scheduling policy can change T^c's status.

Case 2: T^c is in BQ with an outstanding I/O request: A disk scheduling policy could service T^c's request first. However, since T^c is an active transaction, it will not release its locks, and thus T^u remains blocked.

Case 3: T^c is in BQ because of data conflict with T^c: T^c itself, like T^c, could be in RQ or BQ. A disk scheduling policy can affect the status of transactions only if T^c is in BQ. In any case, at most T^c could become unblocked, and T^u remains blocked. Thus, in all cases UNBLOCKEDF is unachievable.

**Corollary (to Lemma 1):** FP and RP guarantee UNBLOCKEDF in fewer cases than DP.
Proof: FP does not guarantee UNBLOCKED at all, because I/O requests of a committing transaction Ti inherit the original priority Tj. RP does not guarantee UNBLOCKED if BQ consists of a transaction Tj which conflicts with a committing transaction Ti, because the conflicting write request of Tj does not inherit the priority of Ti. □

The above results show that, subject to the truth of Claim 3, DP is guaranteed to have better performance than FP and RP. Our simulation results show that this is indeed true.

5. Performance evaluation by simulation

5.1. The simulation model

Our algorithms have been implemented using the process driven simulation language CSIM. Each test consists of 20 runs where each run continued until at least 500 transactions were executed. The data was collected and averaged over the total number of runs. The deadlock detector runs every five seconds.

The database buffer pool is modeled as a set of frames each of which can contain a single data item, i.e. a page. We do not model buffer manager explicitly, instead we model the buffer pool as a collective set. When a transaction attempts to read a data item, the system generates a random boolean variable which has the value true with probability Pi. If the value is true then the page is already in buffer and the transaction can continue processing. If the value is false then an I/O request is created to read a new data item and placed in the I/O queue.

A transaction is characterized by its timing constraints and its length, i.e. the number of data items to be accessed. Let's the estimated runtime of a transaction T be ET. The initial transaction slack for T is uniformly distributed over (0.5 ET, 6 ET). Transactions arrive with exponentially distributed inter-arrival times and are ready for service upon arriving to the system. To define the system load, let's the mean of estimated runtime of transaction T and mean transaction arrival rate to the system be μ(ET) and λ, respectively. Transaction arrival rate is related to the system load as follows:

System load (SL) = \( \mu(ET) \times \lambda \)

A transaction consists of a sequence of read and write accesses. The CPU time for processing a data page is 10 ms. When the data item is not in the buffer, additional 25 ms is required to access the disk. Data accesses by each transaction are uniformly distributed over the entire database. Our database consists of 300 data items and the total number of data items accessed per transaction is uniformly distributed over (5,25). This range is chosen to make our model a scaled down version of Effelsberg and Haerder [7]. The unit of locking is an individual data item.

For testing the proposed real-time multiversion concurrency control algorithms, several baseline concurrency control algorithms were chosen: strict 2PL, conventional multiversion concurrency control, and conditional restart (CR)[1]. We chose CR scheme as our single version real-time locking-based scheme because it shows good performance with the aid of an accurate runtime estimate. However, this scheme is not realistic because it is difficult to get an accurate estimation of a transaction's execution time for disk resident databases. We introduce this scheme to compare it with real-time multiversion concurrency control algorithms which do not require a runtime estimate at all. To obtain an accurate runtime estimate for CR, the transaction generator provides a predefined page hit information to the algorithm.

For the results presented in this paper we use two metrics. First, we measured the percentage of transactions which missed their deadlines, i.e. the miss ratio(MR), defined as:

\[ MR = \frac{\text{total number of transactions missing the deadlines}}{\text{total number of transactions entering the system}} \]

Second, we measured the percentage of restarts. The restart ratio, RR, is defined as:

\[ RR = \frac{\text{total number of restarts}}{\text{total number of transactions entering the system}} \]

Our simulation parameter values which are close to the past work are summarized in Table 2.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Base Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database size</td>
<td>300</td>
</tr>
<tr>
<td>Number of data accesses per transaction</td>
<td>5 - 25</td>
</tr>
<tr>
<td>Disk access time (ms)</td>
<td>25</td>
</tr>
<tr>
<td>CPU computation time (ms)</td>
<td>10</td>
</tr>
<tr>
<td>Restart overhead (ms)</td>
<td>10</td>
</tr>
<tr>
<td>Page Hit Ratio (Pi)</td>
<td>0.5</td>
</tr>
<tr>
<td>Slack of transaction T</td>
<td>0.5 ET - 6 ET</td>
</tr>
<tr>
<td>Number of runs</td>
<td>20</td>
</tr>
<tr>
<td>Number of transactions per run</td>
<td>500</td>
</tr>
</tbody>
</table>

Table 2.

5.2. Results and discussion

5.2.1. Concurrency control

Fig. 8 compares different concurrency control algorithms with respect to miss ratio. In this experiment we varied the system load when the read/write ratio is 1:1. The system load was varied from 0.1 to 1.1. This corresponds to transaction mean arrival rate from 0.4
transactions/sec to 4.5 transactions/sec. FP was used as the disk scheduling algorithm (We also have similar results when other disk scheduling algorithms are used).

![Graph](image)

**Fig. 8. Miss Ratio (MR) vs. System Load (SL)**

From Fig. 8, we can make the following observations:

1) Real-time multiversion concurrency control algorithms are better than the non real-time single version algorithm (2PL) and the real-time single version algorithm (CR), even when the latter is provided an accurate runtime estimate. This is because real-time multiversion concurrency control algorithms can avoid read/write lock conflicts and write/write lock conflicts. It can be observed that real-time multiversion concurrency control algorithm (CMV2PL) can yield up to 6.3 percent decrease in missed ratio over non real-time concurrency control (2PL) and 4.3 percent decrease over real-time single version concurrency control (CR).

2) Between real-time two version 2PLs (U2V2PL, C2V2PL) and multi version 2PLs (UMV2PL, CMV2PL), the performance of the latter is better than that of the former. There are no data conflict between write locks in multi version 2PLs but there is an increase in certify lock conflicts. However, since there are more write lock conflicts than certify lock conflicts, it is advantageous to eliminate the former even though the latter are increased.

3) Between unconditional and conditional multiversion concurrency control, the latter is slightly better than the former. This indicates that saving a LPT which is in the certify phase increases the overall system performance even though the service for some HPT is delayed.

4) The performance of conventional multiversion concurrency control algorithms (2V2PL, MV2PL) is not better than those of single version concurrency control algorithms. The performance of 2V2PL is worse than that of 2PL. Also, the performance of MV2PL is worse than that of CR. We believe that main reason is that chained waiting is possible in conventional multiversion concurrency control algorithms.

We used database size as a parameter to test algorithms in environments with different conflict ratios. The smaller the database size, the more data conflicts occur. We chose the database size from 100 up to 500 which corresponds to 15 percent and 3 percent fraction of the database being accessed by transactions. Fig. 9 shows the miss ratio when the system load is 0.8. We chose the system load high enough to exercise the algorithms. From Fig. 9, we can observe that the merits of real-time concurrency control algorithms become clear as the data conflict increase, i.e., as the database size decreases. Even the performance difference between unconditional and conditional multiversion 2PL is more evident when the data conflict ratio is high. On the other hand, the performance difference among various algorithms becomes close to each other as data conflict decreases. The results also show that real-time multiversion concurrency control algorithms are quite robust even under high data conflict ratio. From this figure, we can conclude that real-time multiversion concurrency control mechanism can play a significant role in meeting transaction deadlines, especially when the data conflict ratio is high.

![Graph](image)

**Fig. 9. Miss Ratio (MR) vs. Database Size (DS) When SL = 0.8**

Fig. 10 shows the restart ratios of different concurrency control algorithms when read/write ratio is 1:1. As expected, the number of restarts of C2V2PL (resp. CMV2PL) is smaller than that of U2V2PL (resp. UMV2PL). This is because many transactions in their final stages are prevented from being aborted by more urgent transactions. The restart overhead of multi version 2PLs are much smaller than those of two version 2PLs. We can easily admit this result if we compare the number of abort operations in the compatibility matrices in Fig. 3 - Fig. 7. The restart ratio of CR is slightly better than UMV2PL under low load and becomes to lie between those of C2V2PL and UMV2PL under high
load. 2PL has the least restart overhead, because a transaction cannot abort another transaction. A transaction abort can happen only for deadlock resolution.

Fig. 10. Restart Ratio (RR) vs. System Load (SL)

5.2.2. Disk Scheduling

In this experiment we compared three different disk scheduling algorithms when the system load is varied. Two different concurrency control algorithms, 2PL and U2V2PL, are used for each disk scheduling algorithm. The read/write ratio tested is 1:1.

Fig. 11. Miss Ratio (MR) vs. System Load (SL)

From Fig. 11, we can make the following observations:

1) Among the three real-time disk scheduling algorithms, DP has the best performance. In addition, DP’s performance does not degrade much under high load, in contrast to other schemes. This is because DP can service the I/O request dynamically not only for a running transaction which issues an I/O request, but also for a blocked transaction which is waiting for the release of a conflicting lock.

2) The performance of RP is worse than that of DP, but better than that of FP. The reason why FP has the worst performance is that it gives the same priority to I/O requests regardless of the status of the requesting transaction.

3) In this figure, we can observe that disk scheduling has more impact on system performance than concurrency control. DP can yield up to 13 percent decrease in miss ratio over FP. Because of this, the combination of 2PL with DP has better performance than that of combining U2V2PL with FP or even RP, under a wide variation of system load.

In this experiment we studied the different scheduling schemes under varying read/write ratios. The write ratio (WR) was chosen to be 0, 25, 50, 75, and 100 percent. The write ratio of 0 and 100 correspond to read only and write only transactions, respectively. Fig. 12 shows the behavior of three different concurrency control algorithms when the disk scheduling schemes used are FP, RP, and DP, respectively. From Fig. 12, we can observe that the relative performance of the three concurrency control algorithms are same regardless of the disk scheduling schemes used. As the write ratio increases, the performance degrades. However, DP shows that it is quite robust even under high data conflict ratio. Considering the lesser degree of benefits obtained, in some cases it may not be worthwhile to make the concurrency control real-time. In such a scenario, DP alone can play a significant role in improving the system performance.

Fig. 12. Miss Ratio (MR) vs. Write Ratio (WR) when SL = 0.8
6. Conclusion

This paper proposed new concurrency control and disk scheduling algorithms for real-time disk resident databases. Their individual and combined performance, with each other and previous algorithms, was evaluated using a detailed simulation model. In general, we conclude the following:

1) Real-time multiversion is better than both conventional single version and conventional multiversion concurrency control algorithms. It is also better than a real-time single version algorithm which is provided an accurate runtime estimate. Thus, a big advantage of real-time multiversion concurrency control algorithms is to achieve good performance without requiring exact runtime estimates.

2) Multi version 2PLs are better than two version 2PLs at the expense of increased memory cost. It was shown that the merits of conditional multiversion algorithms becomes clearer as the data conflict ratio increases.

3) DP has the best performance among three disk scheduling schemes we have tested. It was observed that DP is very robust even under high system load. Because of this, the performance of the combination of 2PL with DP is even better than the combination of U2V2PL with FP. In some cases it may not be worthwhile to make the concurrency control real-time. In such a scenario, DP alone can play a significant role to improve the overall performance.

4) Making concurrency control real-time has some impact on performance. However, real-time disk scheduling algorithms have more impact in improving the performance of real-time transactions. The main reason is that the concurrency control affects only the pair of conflicting transactions, while the disk scheduling affects all active transactions. Therefore, the decision of the concurrency control scheduler can be viewed as local, while the decision of the disk scheduler can be viewed as global.

References


