Concurrency Control in Real-Time Database Systems: Optimistic Scheme vs. Two-Phase Locking

Jiandong Huang and John A. Stankovic
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 90-66
July 1990

Concurrency Control in Real-Time Database Systems: Optimistic Scheme vs. Two-Phase Locking *

Jiandong Huang[1] and John A. Stankovic[2]
[1]Department of Electrical and Computer Engineering
[2]Department of Computer Information Science
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 90-66
July 1990

Abstract

The two-phase locking approach widely used for concurrency control in database systems have some inherent disadvantages such as deadlock and unpredictable blocking time. These appear to be serious problems with respect to real-time transaction processing, since in a real-time environment transactions need to meet their time constraints as well as their consistency requirements. Integrated with CPU scheduling, we investigate an optimistic concurrency control approach for real-time transaction processing, which possesses the properties of deadlock freedom and predictable blocking time. We also give solutions to the problem of transaction starvation. The proposed optimistic concurrency control scheme is implemented on a real-time database testbed. The performance results show that the optimistic scheme outperforms two-phase locking even when the system is CPU bound.

*This work was supported by the National Science Foundation under Grant IRI-8908693 and Grant DCR-8500332, and by the U.S. Office of Naval Research under Grant N00014-85-K0398.

1 Introduction

The challenging problem in real-time transaction processing is how to meet both transaction timing constraints and data consistency requirements. With respect to data consistency requirements, researchers are approaching this problem from two directions: (1) scheduling real-time transactions with a strict consistency constraint as defined by the notion of serializability [20,1,10] and (2) scheduling real-time transactions with relaxed consistency constraints [19,20,14]. In this work, we study an optimistic scheme [8,9] based on maintaining serializability as an alternative of two-phase locking for real-time concurrency control.

It is interesting to note that with the strict consistency requirement, all the concurrency control schemes for real-time transaction processing considered in the literatures thus far are based on the basic mechanism of two-phase locking (2PL) [20,1,10]. This is not surprising, since 2PL has been well studied in traditional database systems and is being widely used in commercial databases. But 2PL, on the other hand, has some inherent problems such as deadlock and unpredictable blocking time. These appear to be more serious problems with respect to real-time transaction processing, since in a real-time environment transactions need to meet their time constraints as well as consistency requirement.

Recently, Sha et. al. [18] proposed a locking based protocol, called the *priority ceiling protocol*, which has the properties of deadlock freedom and predictable blocking time. The underlying idea of the protocol is to ensure that locking operations are carried out based on a total (priority) ordering of the concurrent transactions. Conflict access is avoided by blocking the transaction outside of its execution. The priority ceiling protocol does provide a method to prevent deadlock occurrence and minimizes the blocking time. But because of its pessimistic nature, it causes unnecessary blocking too, reducing the concurrency of transaction execution. Also, the priority ceiling assignment needs prior knowledge of relations between the data objects and the transactions, i.e., which transactions will access which data objects. This requirement is not feasible for many database applications.

Another class of well-known concurrency control schemes is the optimistic approach [13]. Under the optimistic concurrency control (OCC), no lock is employed. Transactions execution in parallel. Furthermore, OCC provides freedom from deadlock, thus saving the expense that the deadlock detection usually required in locking algorithms. The properties of non-blocking and deadlock freedom are ones desired in real-time transaction processing.

Thus far none of the papers on real-time transaction processing concern using optimistic scheme for concurrency control. This is probably because OCC is not widely used in practice. And it might also lie in the fact that analytical studies [2,15] and simulations [3,5] have revealed that in most cases the locking approach performed better than the pure OCC [13]. The main reason for this is the risk of a high abort rate or even cyclic restarts, especially for long transactions or in the presence of hot spot objects. The optimistic approach will apparently become a loser if there are no enough system resources (CPU and I/O) available.

Different from traditional database systems, CPU scheduling is a key component in real-time database systems [10]. Integrated with CPU scheduling, OCC becomes a good

candidate for concurrency control in real-time databases. This is because OCC's non-blocking nature gives CPU scheduler a greater freedom to schedule real-time transactions.

Based on the concept of optimistic concurrency control, we developed two real-time oriented optimistic schemes using a *pseudo-locking* method. The schemes possesses the properties of deadlock freedom and high degree of parallelism of the original optimistic approach. Integrated with CPU scheduling, the blocking time under the proposed optimistic schemes is limited and predictable as compared with 2PL approach. The preliminary performance studies have shown that the optimistic scheme outperforms 2PL in terms of maximizing the total value that real-time transactions impart to the system as well as maximizing the percentage of real-time transactions in meeting their deadlines.

This paper is organized as follows. We describe our system and transaction model in Section 2. In Section 3, we first give a brief review to the original proposal for optimistic concurrency control, and then we describe our optimistic approach for real-time concurrency control. In Section 4, we present two implementation schemes in detail and discuss some related implications. The real-time conflict resolution strategies are discussed separately in Section 5. The issue of transaction starvation is discussed in Section 6. In Section 7, we present our results. Finally, we give some concluding remarks and point out future research in Section 8.

2 A Real-Time Database System

In this study, we consider a centralized real-time database system. As mentioned at the beginning of the introduction, we require that all the real-time transaction operations maintain data consistency as defined by the notion of serializability.

The two major processing components that we consider in this system are *CPU scheduling* and *concurrency control*. Upon arrival of each transaction, the *scheduling* component first assigns a priority to the transaction according to a certain priority assignment policy (to be described below), and then performs dynamic CPU scheduling for the concurrent transaction processes. The *Concurrency control* component enforces data consistency by using two-phase locking or optimistic concurrency control. Note that the *concurrency control* component shares the transaction priority information with the *CPU scheduling* component in order to carry out the real-time concurrency control decisions.

A real-time transaction is characterized by a value function [10]. In a real-time database, each transaction imparts a value to the system, which is related to its criticalness and to when it completes execution (relative to its deadline). Here criticalness represents the importance of transactions, while deadlines constitute the time constraints of real-time transactions. We use the following formula to express the value of transaction T:

$$V_T(t) = \begin{cases} c_T, & s_T \le t < d_T \\ c_T \times (z_T - t)/(z_T - d_T), & d_T \le t < z_T \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

where t - current time;
s_T - start time of transaction T;
d_T - deadline of transaction T;
c_T - criticalness of transaction T, $1 \leq c_T \leq c_{Tmax}$;
c_{Tmax} - the maximum value of criticalness.

In this model, a transaction has a constant value, i.e., its criticalness value, before its deadline. The value decays when the transaction passes its deadline and decreases to zero at time z_T. We call z_T the *zero-value point*. z_T is calculated by by the following formula.

$$z_T = d_T + (d_T - s_T)/c_T \tag{2}$$

For further details about the value function, the reader is referred to [10].

The transactions considered here are solely soft real-time. Given the value function, real-time transactions should be processed in such a way that the total value of completed transactions is maximized. In particular, a transaction should abort if it does not complete before time z_T, since its execution after z_T does not contribute any value to the system at all. On the other hand, a transaction aborted because of deadlock or data conflict may be restarted if it may still impart some value to the system. The restarted transaction will access the same set of data objects as it did in its first (unfinished) run.

Given the transaction model, now we describe three priority assignment policies considered in this study.

EDF - the transaction with the earliest deadline has the highest priority.

CDF - the transaction with the smallest value d_T/c_T has the highest priority, where d_T is the transaction deadline and c_T is the transaction criticalness.

CDLF - Let l_T be the number of records that transaction T has processed by time t. Then,

$$cdl_T(t) = w \times (d_T/c_T) - (1-w) \times l_T, 0 \leq w \leq 1 \tag{3}$$

where w is a weighting factor. The transaction with the smallest cdl_T value has the highest priority.

EDF uses only timing information to give high priority to transactions which have shorter deadlines. CDF considers both transaction deadline and criticalness. This policy implicitly maximizes the value that each transaction may impart to the system. CDLF takes into account the amount of work each transaction has done as well as its deadline and criticalness. This additional factor added to CDLF attempts to minimize the wasted processing power and to resolve transaction starvation (see Section 6).

Note that the policy CDLF has several implications. First, parameter w is used to weight the cdl_T value between factors d_T/c_T and l_T. When $w = 1$, CDLF becomes CDF. Second, with $0 < w < 1$, if two transactions have a similar d_T/c_T value, then the one which has processed more records will have higher priority over the other. On the other hand, if two transactions have a similar l_T value, the one with smaller d_T/c_T will have higher priority.

3 Optimistic Concurrency Control for Real-Time Transactions

In this section, we first give a brief review of the mechanism of optimistic concurrency control. Then, we describe our optimistic concurrency scheme for real-time database systems.

3.1 Principle of Optimistic Concurrency Control

With optimistic concurrency control (OCC), as originally proposed in [13], the execution of a transaction consists of three phases: read, validation, and write. During the read phase, a transaction first copies the data objects from the database to its own private buffer not accessible by other transactions. Then all the operations, read/write, are performed within the buffer. At the end of the transaction, it gets into the validation phase, where it checks whether or not it was in conflict with any other transactions operating in parallel. The validating transaction should preserve serializability. Since no locks are held, the objects read by the transaction might have been modified meanwhile by concurrent transactions. If so, conflict resolution relies on transaction abort as opposed to blocking in locking algorithms. If no conflict is detected, the transaction is prepared to commit. In the write phase, a reader transaction is automatically committed after successful completing the validation phase. A writer (update) transaction has to force sufficient log data to a safe place for recovery purpose and makes the modifications visible to other transactions.

The key component in OCC is the validation process. In order to validate the serial equivalence criterion, we need to check the execution sequence of concurrent transactions. Here the basic idea is that if transaction T_i comes before transaction T_j, then the writes of T_i should not affect the read phase of T_j and T_i should not overwrite T_j. To satisfy this requirement, one of the following conditions must hold.

1. *write set* of T_i does not intersect the *read set* of T_j and T_i completes its write phase before T_j starts its write phase.

2. *write set* of T_i does not intersect the *read set* or *write set* of T_j and T_j completes its read phase before T_j completes its read phase.

If the checking condition is not satisfied, according to [13], the validating transaction will be aborted and restarted.

As an example, Figure 1 shows the scenario for transaction validation. Let RS(T) and WS(T) denote the *read set* and *write set* of transaction T, respectively. Now suppose T_j is in its validation phase. Then in order to satisfy condition 1, RS(T_j) has to be checked against WS(T_1). To meet condition 2, RS(T_j) must be checked against WS(T_2) as well as RS(T_2). T_3 is not involved in T_j's validation since it is still in its read phase.

A distinguishing nature of OCC approach is its non-blocking property and, in turn, avoids deadlock. These properties are specially attractive to real-time concurrency control. This is because under timing constraints, the uncertainty, such as blocking and potential

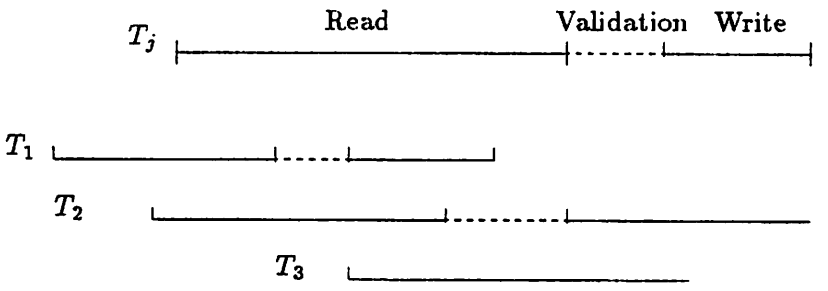


Figure 1: Validation Scenario for T_j

deadlock, should be eliminated as much as possible. In addition, CPU scheduling is a key component for real-time transaction processing [10]. The non-blocking nature gives a greater freedom to the CPU scheduler. Therefore, we propose using OCC as an alternative approach for real-time concurrency control.

In the following sub-section, we discuss how the concept of optimistic concurrency control can be applied to real-time transaction processing.

3.2 Real-Time Optimistic Concurrency Control

With the original OCC [13], as we discussed above, a transaction's destiny is decided at the validation phase. This leads us to consider embedding the transaction's timing constraints into the validation phase. Briefly, during the validation, we not only check the conflict conditions, but also take transaction priority into account. As conflict occurs, the resolution (abort or delay) will be based on the priority of the transactions involved in the conflict.

As we examine the original OCC protocol in detail, however, we would find that it does not support our idea at all. With that protocol, only the validating transaction could be possibly aborted in the case of conflict, regardless of the timing constraints or importance of the conflicting transactions. In other words, even though the validating transaction found transactions which conflicted with itself to have lower priorities, it cannot preempt them. The problem lies in the fact that in the validation phase the validating transaction only checks those transactions which have committed. Of course, the committed transaction can never be aborted, even though its priority is the lowest among the conflicting transactions.

It is clear that for real-time transactions we want to allow the validating transaction check conflict against the concurrent transactions which are still active, so that as conflict is detected, any transaction could be aborted. One way to achieve this is to do forward validation [17,7]. Forward validation works in such a way that the validating transaction T_j checks conflict against those transactions T_i (i = 1,2, ..., n) that are still in their read phase. As in the original OCC, serializability under forward validation is guaranteed by assuring that read sets are always clean and write sets will never be overwritten. To satisfy the requirements, the following two conditions must hold:

1. $\mathrm{WS}(T_j) \cap \mathrm{RS}(T_i) = \{\}$, **and**

2. T_j completes its write phase before T_i starts its write phase.

Note that we may relax condition 2 to allow concurrency in the write phase, but this needs more checking conditions which will lead to higher complexity and overhead in implementation. As a result, the further increase of concurrency might not yield a real performance gain. Thus, in this study we only consider the above two conditions for forward validation.

Using the forward validation method, we now give an algorithm for validation of real-time transactions. Let T_j be the validating transaction and T_i (i = 1,2,...,n) the transactions in read phase.

```
VALID := true
for T_i (i = 1,2, ..., n) do
    if WS(T_j) ∩ RS(T_i) ≠ {}
      then VALID := false
if VALID
    then go to the write phase
    else invoke real-time conflict resolution
end if;
```

Note that in order to satisfy condition 2, the validation phase and the write phase should be carried out within one critical section. With this algorithm, the validating transaction checks with active transactions which are still in their read phase. Thus, when a conflict is detected, there is a great deal of flexibility in resolving the conflict among all of the transactions involved in the validation. Here the real-time conflict resolution becomes an important influential factor to the performance of real-time transactions. We will discuss various conflict resolution strategies in a later section.

Now let us consider an example of the forward validation scheme. In figure 2, there are four transactions T_j, T_{done}, T_1, and T_2 where T_j is the validating transaction. By forward validation, T_j does not check transaction T_{done} since T_{done} has completed before the T_j's validation. T_1 and T_2 are both active, i.e., they are in their read phase. Thus T_j's WS(T_j) is checked against RS(T_1) and RS(T_2). If there is a conflict, say WS(T_j) $\cap$ RS(T_2) $\neq$ {}, then either T_j or T_2 should be aborted, or T_j may just wait until T_2 finishes, depending on the conflict resolution policy applied.

Besides the high degree of freedom in handling and optimizing conflict resolution, forward validation scheme has some other advantages over the original OCC. For example, since only writers are subject to validation and write set is usually a subset of the read set, checking is much less frequent compared to the original OCC scheme. Thus, forward validation is more efficient. Also, under forward validation, conflicts are detected earlier because the validating transaction checks conflict with transactions in read phase. This early detection may reduce the waste of system resources consumed by later-aborted transactions.

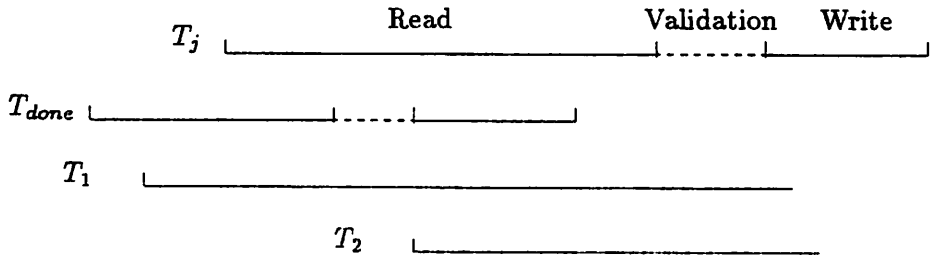


Figure 2: Forward Validation Scenario for T_j

4 Implementation

The forward validation scheme discussed above gives the real-time concurrency control component a flexibility to deal with the detected conflict. But it also brings in some new problems, since the validating transaction must be checked against active transactions that are still in their read phase. In this section, we propose implementation schemes using a *pseudo-locking* method.

With forward validation scheme, each transaction T_i maintains its own *read set*, RS(T_i), and *write set*, WS(T_i). Besides, there is a common pseudo-lock table, PLT, shared by concurrent transactions. We define two lock modes - *read-phase lock* (R-lock) and *validation-phase lock* (V-lock) and apply them to PLT for the dynamic read-set checking. R-lock is set by transactions in read phase, while V-lock is set only by the transaction in validation phase. The two lock modes are incompatible. In the following, we present two implementation schemes, with respect to the parallelism of the I/O operations in write phase.

Throughout this section, we use T_i (i = 1,2,...,n) to denote transactions in read phase and T_j the transaction in validation phase. We bracket the critical section by "<" and ">".

4.1 Serial I/O: RT-OCC-SIO

A simple way to guarantee the validation conditions (see Section 3.2) is to embed the validation phase and write phase in one critical section. We call this scheme as *Serial I/O*, since I/O operations in write phase are carried out in the critical section.

During the read phase, each transaction T_i works on its own buffer space, but it also set R-lock in PLT for every data object in its RS(T_i). During the validation phase, the validating transaction T_j checks its WS(T_j) against RS(T_i) by setting V-lock in PLT. If no conflict occurs, that means that the write set of the validating transaction does not intersect with the read set of any other active transactions. At this point, the validating transaction deletes its R-locks in PLT and goes to its write phase. A failure of V-lock setting, on the other hand, indicates that a conflict has occurred, since the object in WS(T_j) is also shared by other transaction(s). In this case, the real-time resolution policy is invoked to resolve the conflict (see Sec. 5). We give the protocol in the following.

RT_OCC_SIO:

- **Read phase:**

  < **for** every data object in RS(T_i) **do**
  set an R-lock in PLT
  if a V-lock already exists in PLT >
  then wait for V-lock release

- **Validation and Write Phase:**
  VALID := true

  < **for** every data object in WS(T_i) **do**
  set a V-lock in PLT
  if an R-lock of that object exists
  then VALID := false
  release T_j's R-locks in PLT
  if VALID
  then go to the write phase
  else invoke real-time conflict resolution
  release T_j's V-locks in PLT >

RT_OCC_SIO is a simple and easy-to-implement protocol. It can be applied to main memory resident database systems where write phase is done in main memory, or to disk resident databases which are query-oriented.

On the other hand, serial I/O may not be necessary if there are rare conflicts between update transactions. Also, since the write phase is embedded together with the validation phase in a critical section, the critical section may become a system bottleneck. This is especially true to the disk resident databases. To separate I/O operation from the critical section, we give another implementation scheme, called *Parallel I/O*, in the following.

4.2 Parallel I/O: RT-OCC-PIO

In order to separate I/O operations from the critical section and at the same time to guarantee the validation condition of *no over-write* (see Section 3.2), the transactions in read phase need to set R-lock based on their *write set* as well as *read set*. Let T_{RH} be the transaction which holds a R-lock in PLT. We give the protocol for parallel I/O in the following.

RT_OCC_PIO:

Read phase:

< **for** every data object in RS(T_i) **and** WS(T_i) **do**
set an R-lock in PLT
if a V-lock of that object already exists in PLT >
then wait for V-lock release

Validation and Write Phase:
VALID := true

< **for** every data object in WS(T_j) **do**
set a V-lock in PLT
if an R-lock of that object exists **and** $T_{RH} \neq T_j$
then VALID := false
release T_j's R-locks in PLT
if not VALID
then invoke real-time conflict resolution >
else go to the write phase
< release T_j's V-locks in PLT>

In this protocol, the I/O operation is not in the critical section anymore. Yet, validation condition 2 still holds. This is guaranteed by letting transaction in its read phase set R-locks for data objects in its *write set.*

Compared with RT_OCC_SIO, RT_OCC_PIO provides a greater concurrency with respect to I/O operations. On the other hand, RT_OCC_PIO may cause a higher conflict rate, since there exist not only read/write conflicts but also write/write conflicts now. Which protocol is chosen depends on the type of database system (memory or disk-resident) and the kind of workload (read/write ratio).

4.3 Some Implications

There are several common implications to the two protocols presented above.

First, the *pseudo-locking method* used here is different from *two-phase locking* (2PL). With the *pseudo-locking* method, V-lock is issued at the end of a transaction and the locking period is the duration of validation phase plus write phase. With 2PL, however, the write lock is issued whenever the update transaction accesses a data object for update and the locking period may be as long as the transaction lifetime. Furthermore, the R-lock used by the pseudo-locking method will not block any concurrent transactions in read phase, while under 2PL any conflict between read/write locks will block the conflicting transactions.

Second, the two protocols proposed here are deadlock-free, even though R-lock and V-lock are used by the pseudo-locking method. The deadlock freedom is guaranteed by

letting the validation transaction set V-locks in the critical section and hold them until it terminates (i.e., either aborts or commits). Since no more than one transaction could be in validation, it is not possible for the validating transaction to wait for any V-locks held by other transactions. Thus, a wait-for cycle can not be formed.

Third, blocking [1] may occur in transaction's read phase due to access conflict. This is necessary for enforcing the two validation conditions (see Section 3.2). Unlike blocking under 2PL where a high priority transaction may be blocked by low priority transactions for an uncertain period of time, the blocking time on a data object under the optimistic scheme is no longer than the write phase of the conflicting transaction.

Fourth, the critical section might become a bottleneck when operations of validation process are heavy. A solution to this problem is to make use of CPU scheduling to raise the process priority of the validating transaction to the highest among the concurrent transactions, thus reducing the validation processing time. In addition, we may use transaction priority to manage the access to the critical section. In case that more than one transaction is waiting for the critical section, one with the highest priority will get access first. Therefore, the worst case blocking time for the critical section is limited to the period of transaction validation.

5 Conflict Resolution

With the real-time oriented optimistic concurrency control schemes proposed above, conflict could happen either in the validation phase of a transaction when it sets V-lock, or in the read phase of a transaction when it sets R-lock for an object to be worked on. For the latter case, in order to enforce the notion of serializability, the transaction needs to wait for the conflicting transaction to finish its write phase. For the former case, on the other hand, a certain strategy is needed to resolve the conflict. The conflict resolution should aim at improving the performance of real-time transactions - in terms of maximizing *deadline guarantee ratio* and maximizing *the total weighted value*. In this section, we propose a set of strategies for resolving the conflict resulting from the validation phase.

5.1 Never Abort Validating Transaction: NAV

Always let the validating transaction commit and abort all the conflicting transactions.

This strategy guarantees that as long as a transaction reaches its validation phase, it will always finish. The advantage of this strategy is that the resources (CPU, I/O, etc.) consumed by a finishing (validating) transaction are never wasted. Applying CPU scheduling, we expect that transactions with higher priority have higher chance to reach the validation phase and in turn have higher chance to commit.

[1] *Blocking* refers to the situation where a high priority transaction is waiting for a low priority transaction due to access conflict.

5.2 Weak Condition Abort: WCA

Abort the validating transaction only if its priority is less than that of all the conflicting transactions.

This strategy takes transaction priority into account. The condition is still in favor of the validating transaction, aiming at reducing the wasted resources from aborted transactions.

5.3 Strong Condition Abort: SCA

Abort the validating transaction if its priority is less than that of one of the conflicting transactions.

In this strategy, transaction priority is the main concern. The validating transaction will never block a transaction with higher priority.

5.4 Validation Wait: VW

The validating transaction steps out its validation phase and wait for the conflicting transactions to finish before re-entering its validation phase.

In some cases, the strategy of aborting conflicting transactions appears too conservative, causing unnecessary transaction abort. Consider the situation when the validating transaction conflicts with the transactions which only have read operations. If the validating transaction has a lower priority as compared with other conflicting ones, instead of being aborted, it may be deferred. In other words, this transaction is "preempted" from its validation phase and is placed into a waiting queue to wait until all of the conflicting transactions finish their validation.

6 Starvation Problem

With traditional optimistic concurrency control schemes, transaction starvation may occur where some transactions may be restarted again and again due to conflict. In general, the longer the transaction, the greater the risk of starvation.

Some solutions to the starvation problem have been proposed [16,17]. They basically rely on the method of limiting the number of transaction restarts. This sort of solutions is inadequate for real-time transactions, since under real-time environment, transactions need to meet their deadlines.

In this study, we solve the starvation problem by using CPU scheduling together with conflict resolution. Since the two processing components use the information of transaction priority, policies for priority assignment become the key factor in resolving transaction starvation. The priority assignment policy, CDLF, that we proposed in Section 2 has a special concern with transaction starvation. In formula (3), besides the deadline and

criticalness information, the transaction length is also taken into account. A transaction which has accessed more data objects will have higher priority. For long transactions, the more data objects they process, the less chance they will be aborted. Thus, transaction starvation is expected to be prevented.

7 Experimental Results

7.1 The Test Environment

Our experiments were conducted on the RT-CARAT testbed - a centralized, secondary storage real-time database system. The testbed contains all the major functional components of a transaction processing system, such as transaction management, data management, log management, and communication management. The testbed is built on top of the VAX/VMS operating system. By appropriate settings of various VMS system parameters, non-essential overheads of VMS, such as memory paging and process swapping, are eliminated.

RT-CARAT is a closed queueing network. Users submit transaction requests one after another. A transaction performs a certain number of predefined operations, called *steps*, and each operation may access a certain number of records and do a certain amount of computation. A transaction terminates upon completion or a termination abort. The transaction deadline is randomly generated from a uniform distribution within a deadline window, $[d\_base, \alpha \times d\_base]$, where d_base is the window baseline and α is a variable determining the upper bound of the deadline window. For each workload in the experiments, d_base is specified first by the formula:

$$d\_base = avg\_rsp - stnd\_dvi$$

where avg_rsp is the average response time of the same real-time transactions when executed in a non real-time database environment under 2PL, and $stnd\_dvi$ is the standard deviation of the response time. Besides the deadline, each transaction, when initiated, is randomly assigned a criticalness from a uniform distribution. We specify the criticalness of transaction T as a function of its class, i.e.,

$$c_T(class) = class_{MAX} - class + 1, \quad class = 1, 2, ..., class_{MAX}$$

where $class_{MAX}$ is the maximum number of transaction classes. The smaller the class number, the higher the corresponding criticalness, and vice versa. Once the deadline and criticalness are specified, the value function of the transaction is fixed and the transaction value can be computed at any time (see Section 2).

Table 1 summarizes the parameter settings in our experiments. The table is divided into two parts. The first part presents the parameters that are kept constant across all workloads, and the second part are those parameters that change according to the different

performance measurements. In our experiments, two separate disks are used, one for the database and the other for the log. The database consists of 1000 physical blocks (512 bytes each) with each block containing 6 records for a total of 6,000 records. In all the experiments, the multi-programming level in the system is 8. Transactions are divided into 8 classes (levels) according to their criticalness. Transaction access to the database is uniformly distributed. The number of records accessed at each transaction step is fixed at 4. Running on a MicroVAX II, RT-CARAT is a CPU bound system. To eliminate further CPU cost, we set computation time of each transaction step at 0.

Table 1: System Parameters

Parameter	Setting
Disks	disk1: database; disk2: log.
Database size	1000 blocks (6000 records)
Multiprogramming level	8
$class_{MAX}$ (transaction classes)	8
Access distribution	uniform
Records accessed per trans. step	4 records
Computation time per trans. step	0 units
P_w (prob. of write transactions)	0.0 - 1.0
x (steps per transaction)	2 - 10 steps
α (deadline window factor)	2.0 - 5.0

A critical factor in comparing concurrency control protocols is P_w, the probability that a given transaction is a write transaction. Changing P_w will affect transaction conflict rate and hence the performance of different concurrency control protocols. Another parameter that we varied in the experiments is x, the number of steps per transaction. It is used to examine the sensitivity of the concurrency control protocols with respect to transaction length. Deadline window factor, α, is a timing-related parameter which specifies the deadline distribution of real-time transactions. This parameter enables us to examine the system performance in response to the tightness of transaction deadlines.

We use the following metrics to evaluate the proposed algorithms and protocols.

- Deadline guarantee ratio (DGR_{class}) - the percentage of transactions in a class that complete by their deadline.
- Average deadline guarantee ratio (ADGR) - the percentage of transactions in all classes that complete by their deadline, i.e.,

$$ADGR = \frac{1}{8} \sum_{class=1}^{8} DGR_{class}$$

- Weighted value (WV_{class})- the total value of all transactions in a class that complete by their zero-value points (z_T) divided by the total maximum value of the invoked transactions in all classes.

- Total weighted value (TWV) - the sum of weighted values in all classes, i.e.,

$$TWV = \sum_{class=1}^{8} WV_{class}$$

- Abort ratio (AR_{class})- the percentage of transactions aborted in a class, due to deadlock (under 2PL) or validation conflict (under OCC).

- Total abort ratio (TAR) - the percentage of transactions aborted in all classes, i.e.,

$$TAR = \sum_{class=1}^{8} AR_{class}$$

- CPU utilization - the percentage of time that the CPU is busy.

The data collection in the experiments is based in the method of replication. The statistical data has 95% confidence intervals with no greater than ± % of the point estimate for total weighted value. In the following graphs, we only plot the mean values of the performance measures.

7.2 Experiments

In this section, we present some experimental results from our performance studies. As an initial step, the experiments presented in the following focused only on the comparison of the two concurrency control approaches, *optimistic* and *two-phase locking*, in connection with CPU scheduling. The proposed real-time policies for conflict resolution and starvation problem were not exercised at this point.

Table 2 lists the protocols and policies examined in the experiments. MFQ is a CPU scheduling policy commonly used in non real-time database systems. In this study, it is used as a baseline policy for the purpose of performance comparisons. For concurrency control, the protocol used in the optimistic scheme is RT_OCC_PIO (see Section 4.2). In addition, since WAIT and NAV are the only policies for conflict resolution used in OCC and 2PL respectively, we take them as default policies in the following.

Our discussion will emphasize the performance results of the two concurrency control protocols. For detailed performance analysis of CPU scheduling policies, the reader is referred to [11].

Table 2: Protocol Selection

Component	Protocol	Meaning
CPU scheduling	MFQ	Multi-level feedback queue
	EDF	Earliest deadline first
	CDF	Criticalness and deadline
Concurrency control	OCC (RT_OCC_PIO)	Optimistic scheme
	2PL	Two-phase locking
conflict resolution		
for OCC	NAV	Never abort validating transaction.
for 2PL	WAIT	Lock requester always waits.

7.2.1 Experiment 1: Protocol overhead

Before studying the performance of OCC and 2PL in detail, we first compare the implementation overhead of the two concurrency control protocols. To capture the pure implementation overhead, we exercise the read-only workloads, i.e., $P_w = 0$. The deadline window factor α is set at 2.

Figures 3-4 show the deadline guarantee ratio and CPU utilization, with respect to transaction length, respectively. For simplicity, we only plot the performance results of OCC and 2PL using EDF for CPU scheduling. As we can see, the performance of OCC and 2PL are almost identical. This performance similarity also holds for the other two CPU scheduling policies, MFQ and CDF.

It is clear that the implementation overhead of the two concurrency control protocols are basically the same. This is due to the fact that even though the two protocols are different in the high level implementation techniques (two-phase locking vs. forward validation), the underlying low level implementation techniques are basically the same. Both protocols rely on locking technique for data access control, and them both use a hash table to manage locks - *write* and *read* locks for 2PL and *R-lock* and *V-lock* for OCC. Also, the implementation of the two protocols involves the use of a critical section for lock management.

Note that there is one distinguishing implementation difference between OCC and 2PL, i.e., 2PL employs deadlock detection and resolution while OCC does not. However, our previous studies have shown that the deadlock detection and resolution on RT-CARAT does not incur significant overhead.

7.2.2 Experiment 2: Performance by transaction class

To compare OCC with 2PL under different CPU scheduling policies, now we examine the system performance with respect to transaction class of criticalness. In the experiment, all the transactions are equal in length with $x = 6$, $P_w = 0.2$, and $\alpha = 3$.

Figure 5 plots the transaction deadline guarantee ratio versus transaction class. Our first observation is that regardless of the concurrency control protocols used, CPU scheduling policies, EDF and CDF perform better than the baseline policy MFQ (except for the transactions in classes 7 and 8 when executed under CDF). In addition, CDF results in a higher deadline guarantee ratio for the transaction classes with high criticalness. Examining the performance of two concurrency control protocols, the reader can see that under MFQ, 2PL outperforms OCC. This result is consistent with the conclusions made in previous studies in traditional database systems [3,2,5,15]. This is understandable, since in our experiment CPU contention exists (CPU utilization is around 90%) and the abort rate is as high as 21% for OCC under MFQ (see Figure 6). Here the interesting result is that integrated with real-time oriented CPU scheduling, OCC outperforms 2PL. In particular, under EDF, OCC achieves more than a 3% increase in deadline guarantee ratio on the average over the 8 transaction classes. With CDF, OCC achieves as high as 10% increase of the deadline guarantee ratio over 2PL for more critical transactions.

The conflict abort ratio is shown in Figure 6. As one can expect, overall, the abort ratio under OCC is higher than that under 2PL. But note that integrated with the CPU scheduling policy EDF, the abort ratio under OCC is greatly reduced. This is because EDF lowers concurrency of active transactions, hence reducing the conflict rate [11]. CDF achieves lower abort ratio for the transaction classes of high criticalness. However, it leads to higher abort ratio for less critical transactions. This is why the transactions in class 7 and 8 have low deadline guarantee ratio under CDF.

The performance results demonstrated here show only one point in our test space. In the following, we expand our experiments along three dimensions, i.e., *deadline distribution*, *write/read ratio* and *transaction length*.

7.2.3 Experiment 3: Varying deadline distribution α

In this experiment, we vary the tightness of deadline distribution by changing the deadline window factor α from 2 (tight) to 5 (loose). Transactions have the same length - 6 steps. We set P_w at 0.2.

Figure 7 depicts the average deadline guarantee ratio of real-time transactions. With the CPU scheduling policy EDF, OCC outperforms 2PL when transaction deadline is loose, but the result is reversed when deadlines become tight. This is due to the deadline-sensitive nature of EDF and the higher abort rate of OCC. The CPU scheduling policy CDF does not perform so well as EDF when the deadline is loose. Note that under CDF, OCC provides higher deadline guarantee ratio than 2PL regardless of deadline distribution. Examining the performance under the baseline MFQ, 2PL is always better than OCC. As mentioned above, this result can be expected in a kind of traditional database systems where system resources (CPU and I/O) are limited and conflict rate is relatively high.

Another performance measure, the total weighted value, is shown in Figure 8. The performance results are similar to what we have seen in Figure 7, except that CDF performs relatively better, especially in the case when the deadline is tight. This is because unlike EDF, CDF takes transaction criticalness as well as deadline information into account.

7.2.4 Experiment 4: Varying write/read probability P_w

In this experiment, we vary the probability of write/read transactions. As in the previous experiments, transaction length is fixed at 6 steps. The deadline window factor α is set at 4.

Figures 8 shows the total weighted value that committed transactions imparted to the system. As P_w increases, the data access conflict increases. Overall, the total weighted value drops. Comparing the two concurrency control protocols, one can see that with MFQ, 2PL is always better than OCC and the performance difference between the two protocols becomes larger as P_w increases. However, integrated with EDF and CDF, OCC may perform better than 2PL. In particular, OCC has better performance, under EDF when P_w is less than 0.45, and under CDF when P_w is greater than 0.45.

The conflict abort ratio and CPU utilization are illustrated in Figures 10 and 11, respectively. As one can expect, the conflict abort ratio under OCC, and in turn the transaction restart rate, is very sensitive to the change of P_w, while under 2PL, the abort ratio has a limit because of blocking. In addition, due to high restart rate, OCC makes CPU utilization constantly high, while under 2PL the CPU utilization decreases, due to blocking, when P_w increases.

7.2.5 Experiment 5: Varying transaction length

To study the sensitivity of the concurrency control protocols to transaction length, we vary x from 4 to 10 in step of 2, with P_w and α being fixed at 0.2 and 4, respectively.

Figure 12 shows the transaction performance with respect to the total weighted value. Here one can see again that integrated with CPU scheduling policy EDF, OCC outperforms 2PL. As we increase the transaction length, OCC_CDF also outperforms 2PL. We further conducted experiments for long transactions, with $x = 10$ and $Pw = 0.2$, by varying α. The results (not shown here) indicate that combined with CPU scheduling, OCC always outperforms 2PL in terms of total weighted value, and OCC provides higher transaction deadline guarantee ratio than 2PL as long as transaction deadlines are not extremely tight.

The reason that OCC performs particularly well for long transactions with write/read mix is because of its "non-blocking" nature. Even though the OCC protocol (RT_OCC_PIO) uses a "pseudo-locking" scheme in practice (in order to implement *forward validation*), the incurred blocking time is small compared to the transaction blocking time under 2PL. Figure 13 plots the average blocking time per blocked lock request[2]. With 2PL, the blocking time increases almost in proportional to transaction length, while under OCC, the blocking time is basically a constant.

[2]A transaction may also be blocked when it accesses a critical section. The experimental measures show that the blocking time resulted from accessing critical section is one to two magnitude smaller than the blocking time resulted from lock request. Hence, the blocking time due to accessing critical sections is not counted here.

8 Summary

In this work, we have proposed an optimistic scheme, in connection with CPU scheduling, for real-time transaction concurrency control. Using a *pseudo locking* method, we have developed two protocols for implementation of the optimistic concurrency control scheme, which have the properties of deadlock freedom and predictable blocking time. The two protocols can be applied to main memory resident and disk resident database systems, respectively. In addition, we have proposed a set of algorithms for conflict resolution accompanied with the optimistic scheme. We have also developed a CPU scheduling scheme to accommodate starvation problem commonly found in optimistic concurrency control schemes. The proposed protocols and algorithms have been implemented on a real-time database testbed.

The experimental results indicate that integrated with real-time oriented CPU scheduling, OCC outperforms 2PL with respect to total weighted value, and OCC provides higher transaction deadline guarantee ratio than 2PL as long as transaction deadlines are not extremely tight. Due to its limited blocking time, OCC performs better than 2PL particularly for long transactions when data contention exists.

Note that the optimistic concurrency control approach exhibits its best performance when system resources (CPU and I/O) are not under contention [3]. But the testbed that we used for performance studies is a CPU bound system. That means the proposed OCC scheme can perform even better if CPU contention does not exist. Furthermore, OCC's performance can be improved when a global buffer is used in the system [12]. Data buffering will benefit both OCC and 2PL. However, the restart-oriented OCC will make better use of the buffer than the blocking-oriented 2PL.

The work presented in this paper is an initial step in studying optimistic schemes for real-time transaction processing. There are a number of problems for future work. For example, real-time oriented conflict resolution policies were not exercised in our performance studies. The performance of OCC is expected to be improved when the proposed real-time oriented conflict resolution policies are applied, which aim at reducing wasted system resources (resulting from high restart rate). On the other hand, our previous studies on 2PL [10] have shown that the real-time oriented conflict resolution policies are important. Thus, the performance of the two concurrency control protocols with using real-time oriented conflict resolution remains to be investigated. Also, we have not explicitly addressed the problem of transaction starvation, even though a CPU scheduling based solution has been proposed. The two concurrency control protocols will be further studied under the worklords of the mix of long and short transactions.

References

[1] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *Proceedings of the 14th VLDB Conference*, Aug. 1988.

[2] Agrawal, R. and D.J. DeWitt, "Integrated Concurrency Control and Recovery Mechnisms: Design and Performance Evaluation," *ACM TODS 10(4)*, 1985.

[3] Agrawal, R., M.J. Carey and M. Livny, "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Transaction on Database Systems, Vol.12, No.4*, Dec. 1987.

[4] Buchmann, A.P., D.R. McCarthy, M. Hsu, and U. Dayal, "Time-Critical Database Scheduling: A Framework For Integrating Real-Time Scheduling and Concurrency Control," *Data Engineering Conference*, Feb. 1989.

[5] Carey, M.J., M.R. Stonebraker, "The Performance of Concurrency Control Algorithms for Database Management Systems," *Proceedings of the 10th VLDB Conference*, 1984.

[6] Eswaran, K.P., J.N. Gray, R.A. Lorie and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," ACM Communication, 19(11), November 1976.

[7] Harder, T. "Observations on Optimistic Concurrency Control Schemes," Information Systems, Vol. 9, No.2, 1984.

[8] Huang, J., "Optimistic Concurrency Control for Real-Time Transactions," *A Research Report*, Dept. of Electrical and Computer Engin., University of Massachusetts, April 1988.

[9] Huang, J., "Real-Time Transaction Processing," *Ph.D. Dissertation Prospectus*, Dept. of Electrical and Computer Engin., University of Massachusetts, June 1989.

[10] Huang, J., J.A. Stankovic, D. Towsley and K. Ramamritham, "Experimental Evaluation of Real-Time Transaction Processing," *Proceedings of the 10th Real-Time Systems Symposium*, Dec. 1989

[11] Huang, J., J.A. Stankovic, D. Towsley and K. Ramamritham, "Real-Time Transaction Processing: Design, Implementation and Performance Evaluation", *A Technical Report, COINS 90-43*, University of Massachusetts, May 1990.

[12] Huang, J. and J.A. Stankovic, "Buffer Management in Real-Time Databases," *A Technical Report, COINS 90-65*, University of Massachusetts, July 1990.

[13] Kung, H.T. and J.T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems, Vol.6, No.2*, June 1981.

[14] Lin, K.J., "Consistency issues in real-time database systems," *Proceedings of the 22nd Hawaii International Conference on System Sciences*, Jan. 1989.

[15] Menasce, D.A. and T. Nakanishi, "Opmistic versus Pessimistic Concurrency Control Mechnisms in Database Management Systems," *Information Systems 7(1)*, 1982.

[16] Peinl, P. and A. Reuter, "Empirical Comparison of Database Concurrency Control Schemes," *Proceedings of the 9th VLDB Conference*, 1983.

[17] Pradel, U., G. Schlageter and R. Unland, "Redesign of Optimistic Methods: Improving Performance and Applicability," *Proc. IEEE 2nd Int. Conf. on Data Engineering*, 1986.

[18] Sha, L., R. Rajkumar and J.P. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record*, March 1988.

[19] Son, S.H., "Using Replication for High Performance Database Support in Distributed Real-Time Systems," *Proceedings of the 8th Real-Time Systems Symposium*, Dec. 1987.

[20] Stankovic, J.A. and Wei Zhao, "On Real-Time Transactions," *ACM SIGMOD Record*, March 1988.

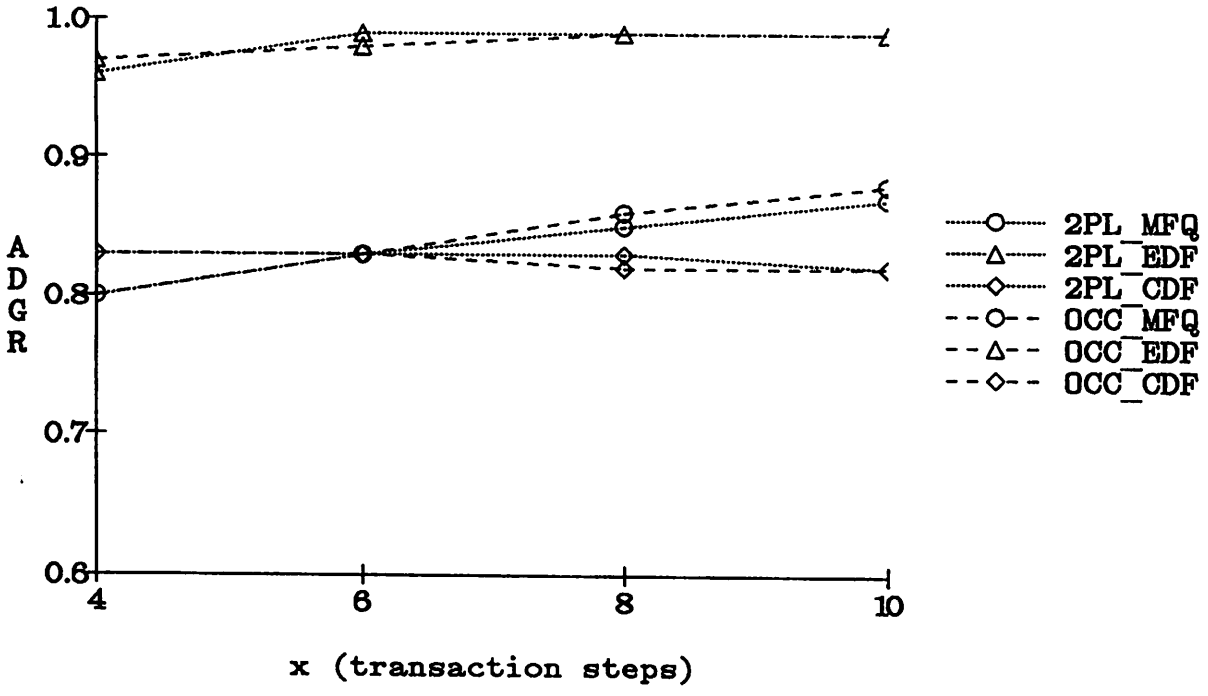


Figure 3. Protocol overhead measures, Pw=0, a=2

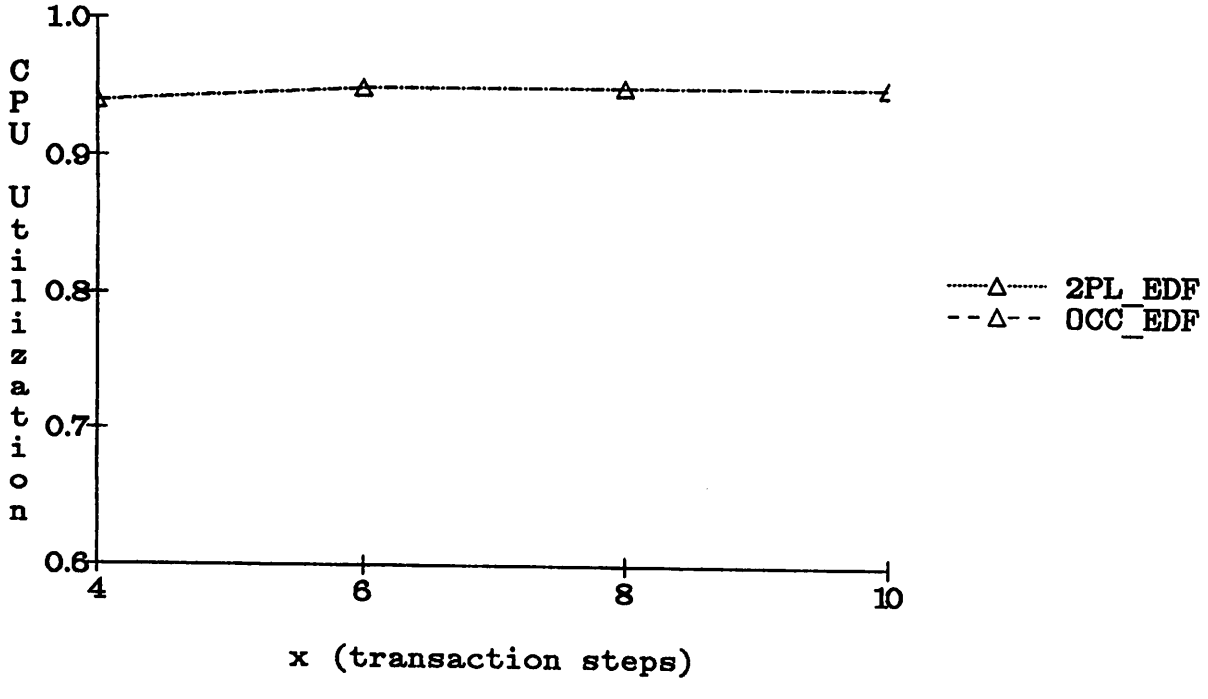


Figure 4. Protocol overhead measures, Pw=0, a=2

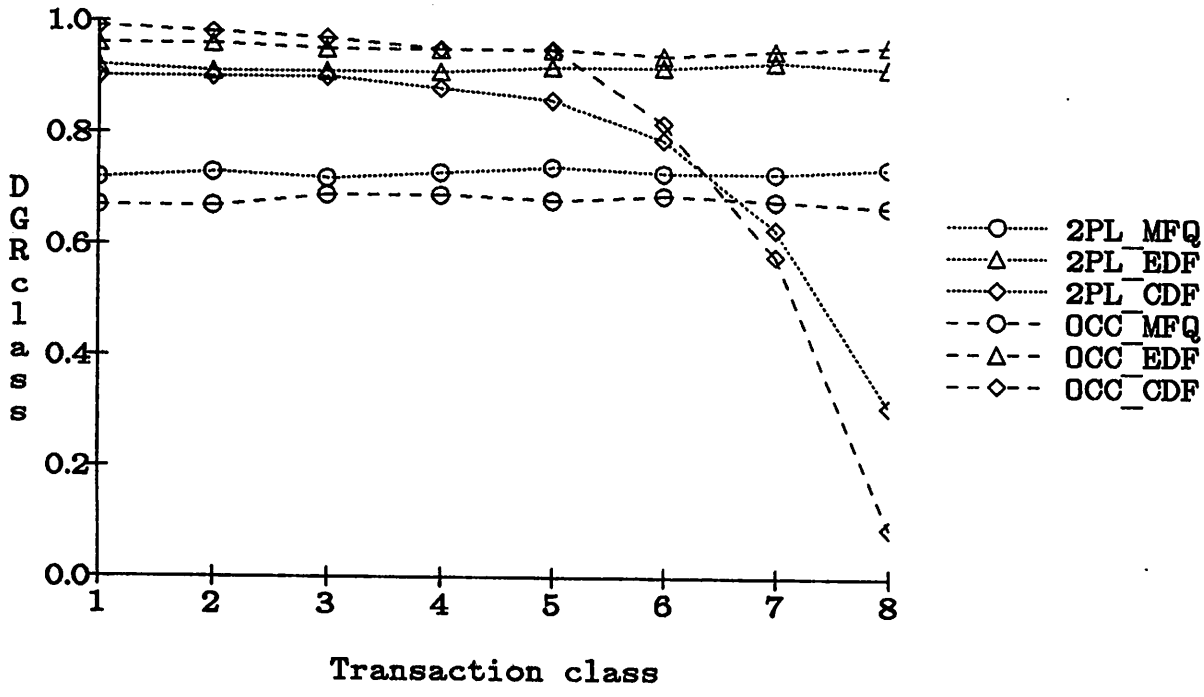


Figure 5. Performance by class, x=6, Pw=0.2, a=4

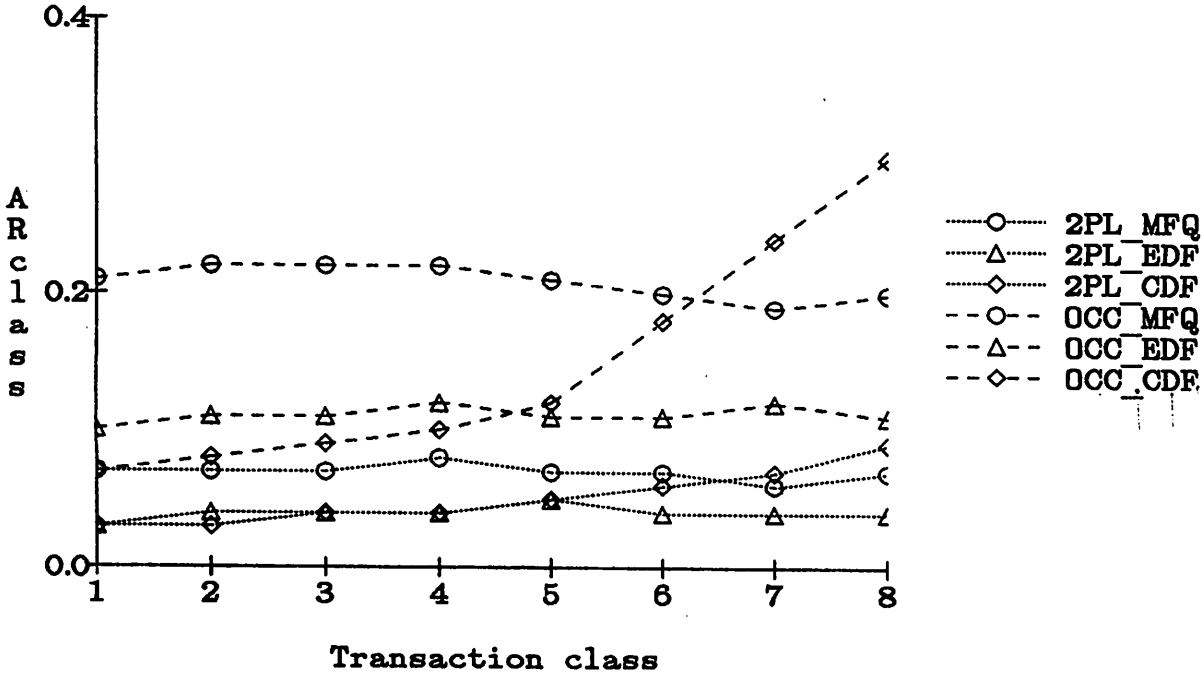
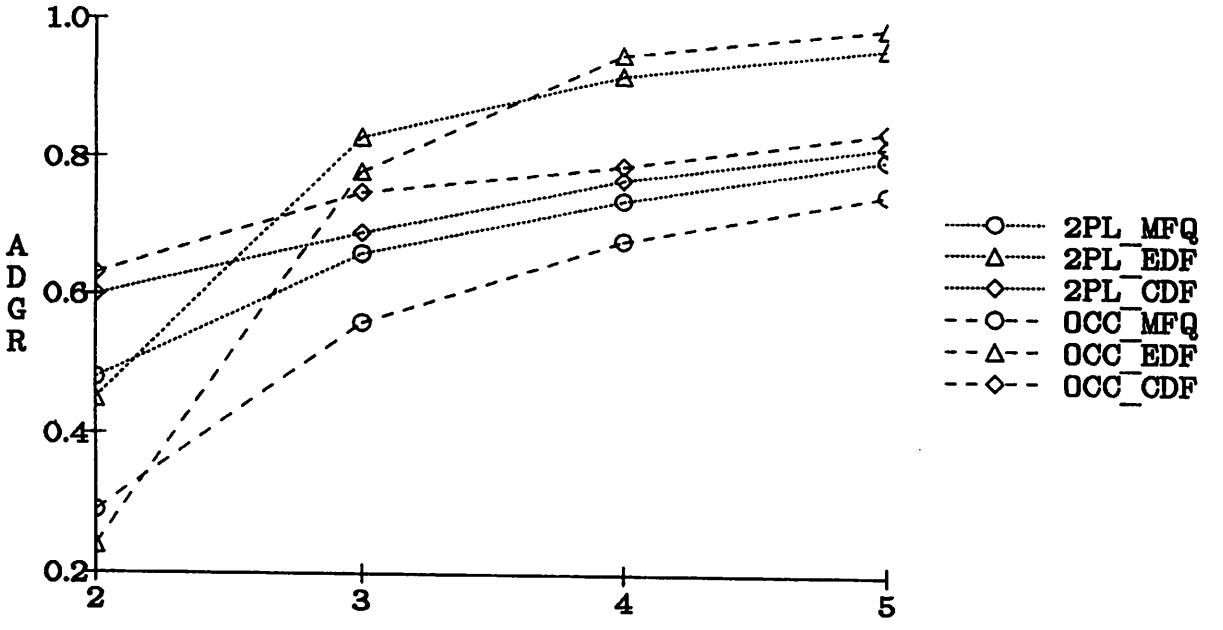
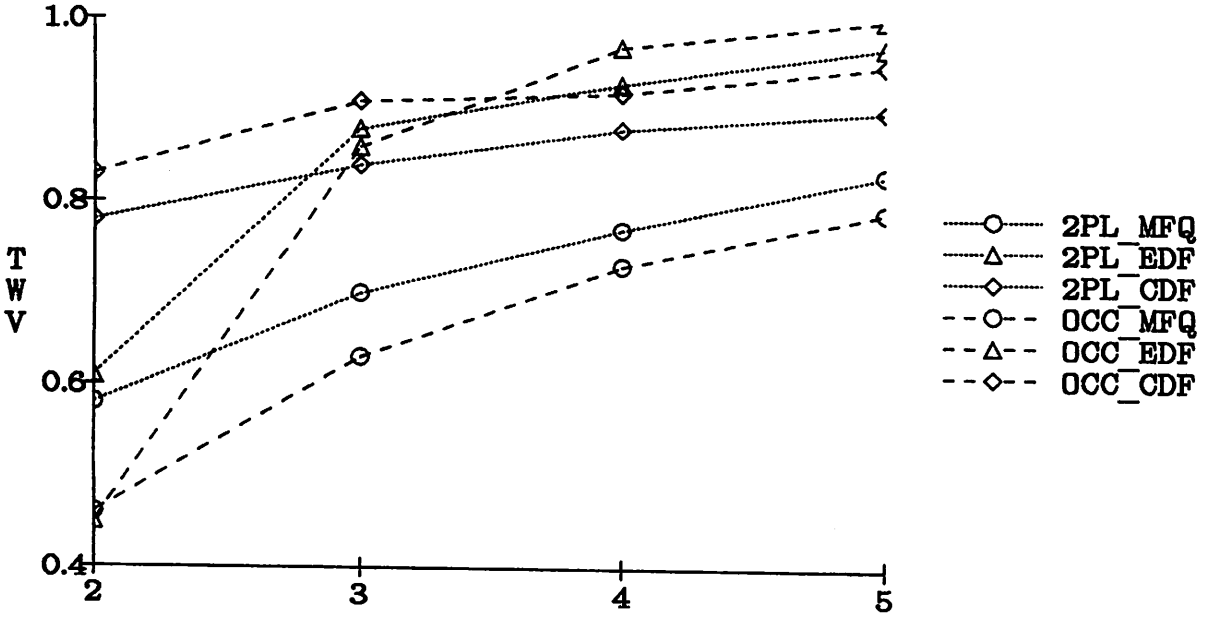


Figure 6. Performance by class, x=6, Pw=0.2, a=4



a (deadline window factor)

Figure 7. Varying a, x=6, Pw=0.2



a (deadline window factor)

Figure 8. Varying a, x=6, Pw=0.2

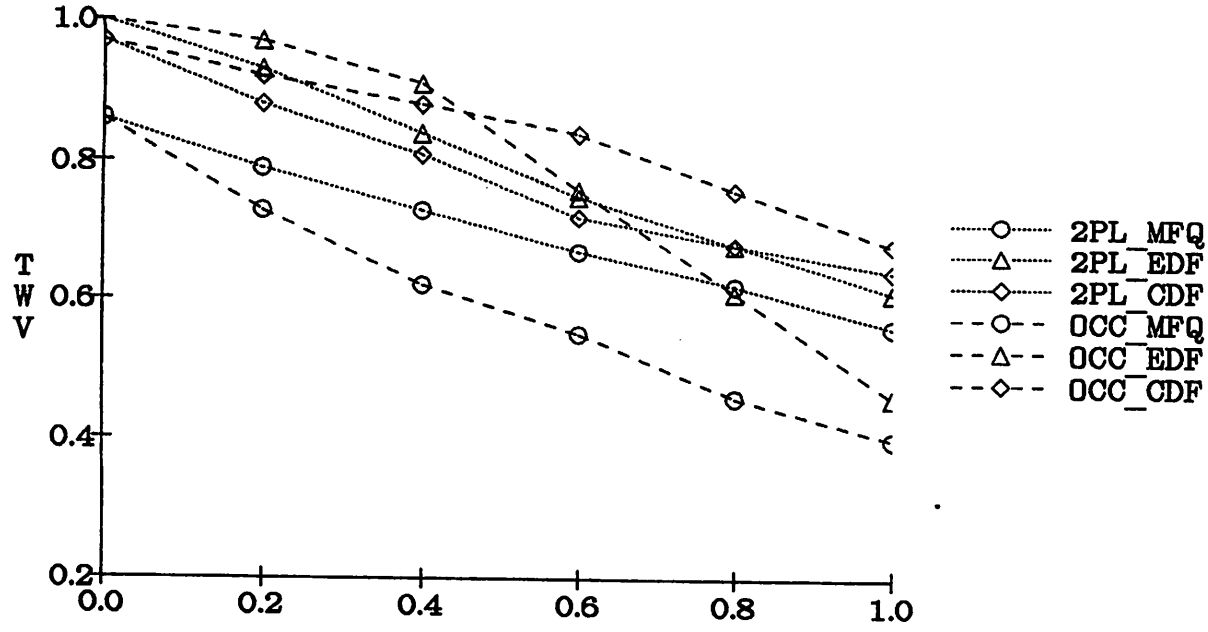


Figure 9. Varying Pw, x=6, a=4

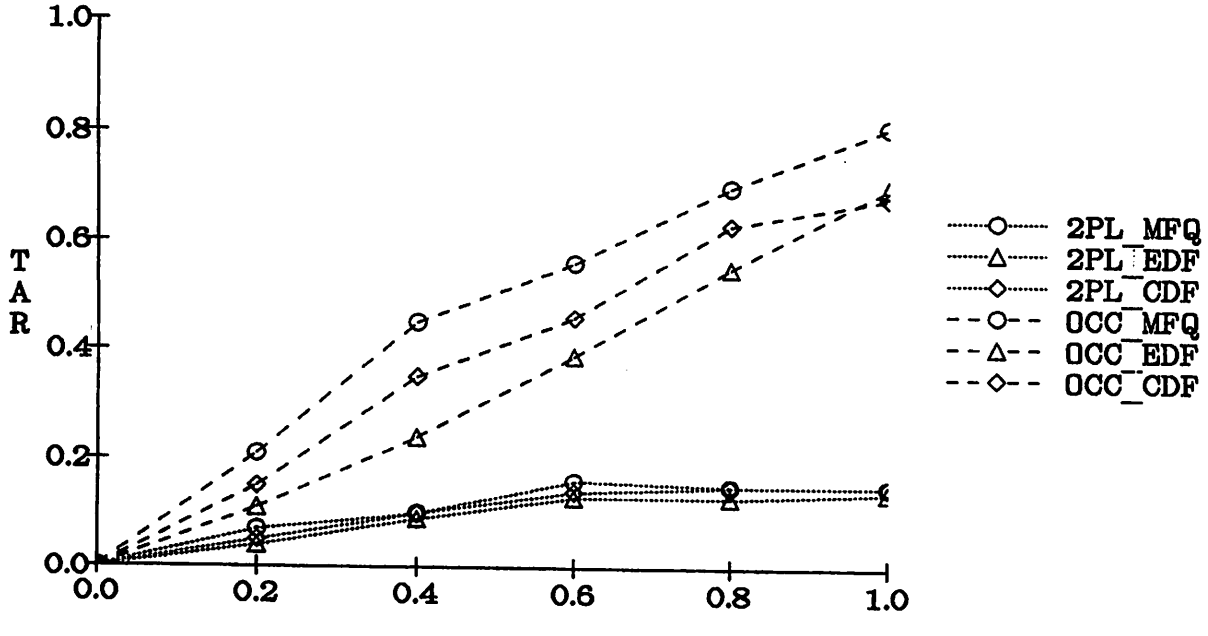


Figure 10. Varying Pw, x=6, a=4

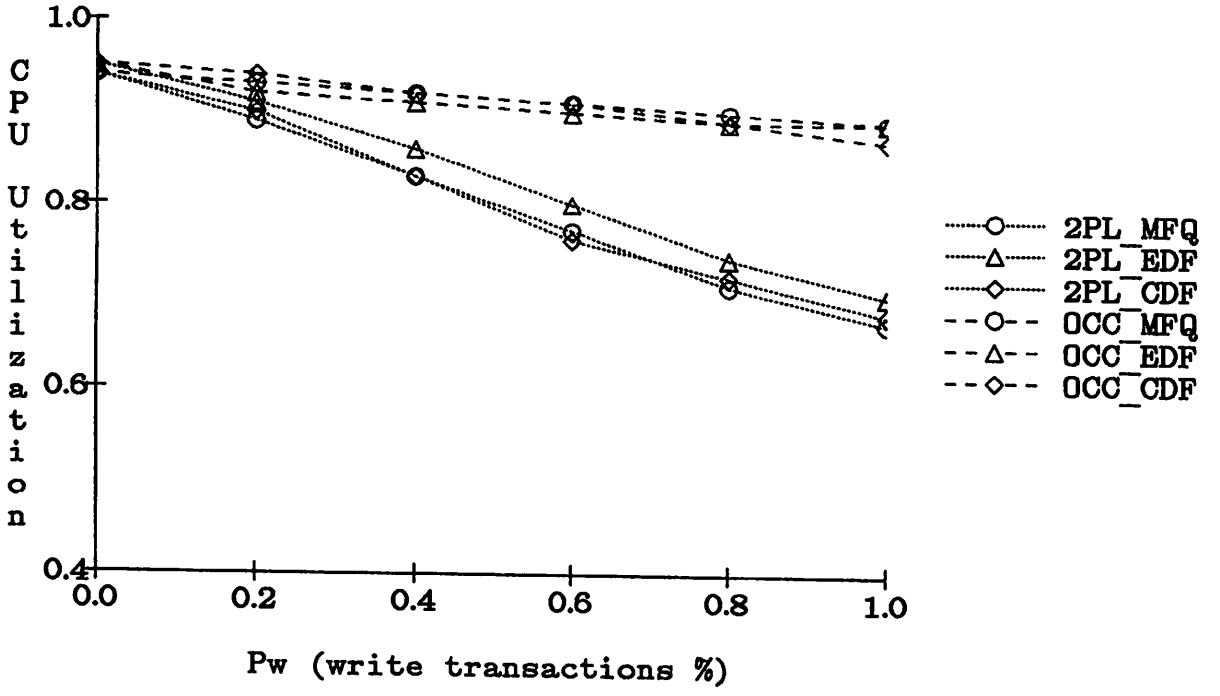


Figure 11. Varying Pw, x=6, a=4

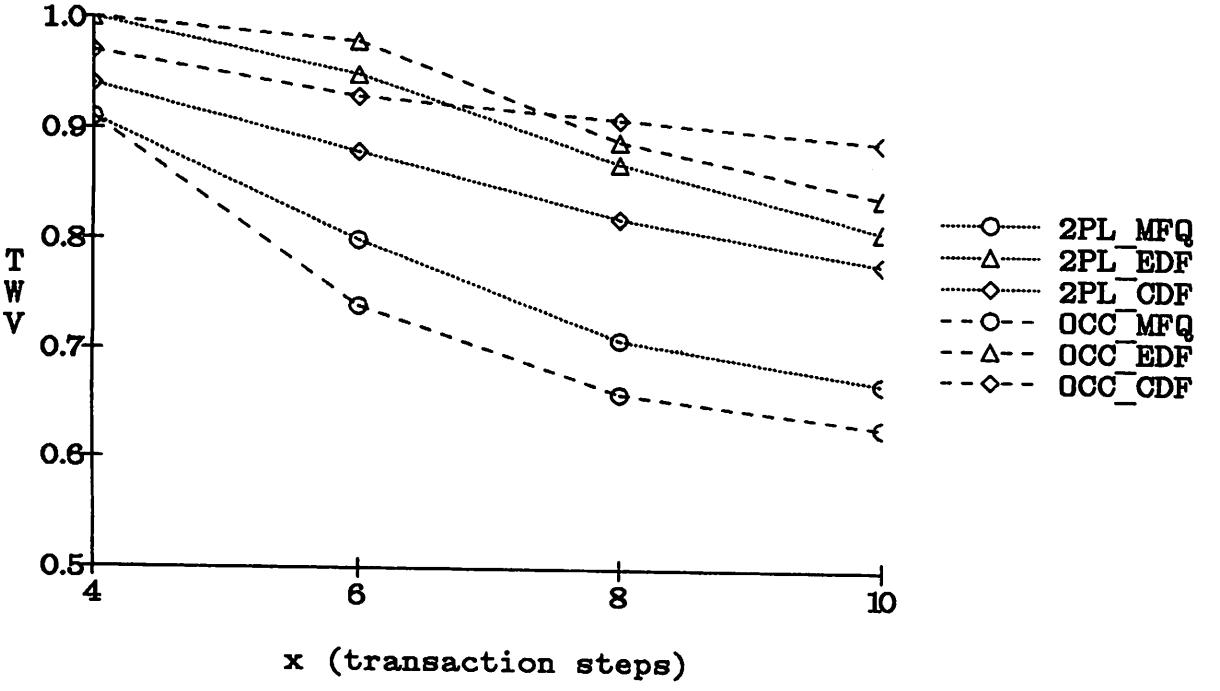


Figure 12. Varying transaction length x, Pw=0.2, a=4

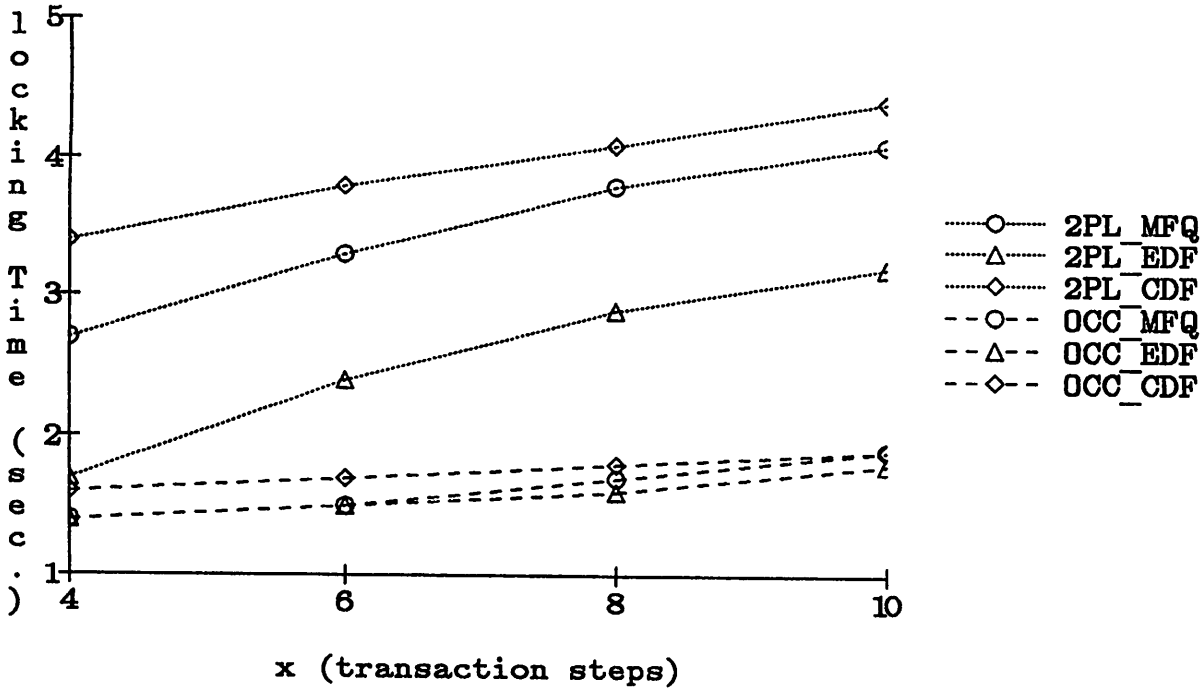


Figure 13. Varying transaction length x, Pw=0.2, a=4