VLDB Journal, **4**, 87-126 (1995), Hector Garcia-Molina, Editor

Ordered Shared Locks for Real-Time Databases

Divyakant Agrawal, Amr El Abbadi, Richard Jeffers, and Lijing Lin

Received August, 1992; revised version received, December, 1993; accepted July, 1994.

Abstract. We propose locking protocols for real-time databases. Our approach has two main motivations: First, locking protocols are widely accepted and used in most database systems. Second, in real-time databases it has been shown that the blocking behavior of transactions in locking protocols results in performance degradation. We use a new relationship between locks called ordered sharing to eliminate blocking that arises in the traditional locking protocols. Ordered sharing eliminates blocking of read and write operations but may result in delayed termination. Since timeliness and not response time is the crucial factor in real-time databases, our protocols exploit this delay to allow transactions to execute within the slacks of delayed transactions. We compare the performance of the proposed protocols with the two-phase locking protocol for real-time databases. Our experiments indicate that the proposed protocols significantly reduce the percentage of missed deadlines in the system for a variety of workloads.

Key Words. Concurrency control, transaction management, time-critical scheduling.

1. Introduction

Databases are being used increasingly for a wide spectrum of applications, and many of these applications impose different and often conflicting demands on the underlying system. One such example involves using databases for real-time applications, referred to as real-time database systems. Some of the applications that require real-time response include military tracking, medical monitoring, and stock arbitrage systems. Such systems must process requests within definite time bounds, and it is the inclusion of timing constraints that characterizes real-time database systems. In general, a constraint is expressed in the form of a *deadline,* which indicates that a transaction must be completed before some specific time in the future. In contrast to traditional databases, where the primary goal is to

Divyakant Agrawal and Amr El Abbadi are Associate Professors of Computer Science, University of California, Santa Barbara, CA 93106, *agrawal@cs.ucsb.edu.* Richard Jeffers, M.S., is Software Designer, Tandem Computers, Inc., and Lijing Lin, M.S., is Software Engineer, Digital Media International, Santa Barbara, CA 93105.

minimize the response time of user transactions and maximize throughput, the main objective of real-time databases is to ensure that transactions meet their deadlines and to minimize the percentage of transactions that miss deadlines in the system.

Real-time systems can be divided into two main types: those with *hard* deadlines and those with *soft* deadlines (Abbott and Garcia-Molina, 1988). Hard real-time systems have deadlines that always must be met by transactions, while soft real-time systems have deadlines that may be missed at some cost to the system. Usually there is an associated value function with each transaction, which decreases after the expiration of the deadline. In addition, the term *firm* real-time systems is used to describe systems that derive no benefit from completing tardy transactions. As a result, tardy transactions are aborted as soon as they are detected. We choose to investigate this latter type of system. If tardy transactions do provide a benefit to the system, the analysis and judgement of a protocol's performance is complicated by the weighting of the values assigned to each transaction (termed a transaction's *criticalness*; Huang et al., 1989). In databases, there are two aspects to scheduling transactions: concurrency control for the execution of transactions that maintain database consistency, and CPU and I/O scheduling for the execution of read and write operations. In this article, we concentrate on the transaction scheduling aspects for concurrency control in real-time databases. The issue of physical resource (CPU and I/O) scheduling has been dealt with extensively elsewhere (Abbott and Garcia-Molina, 1988, 1989, 1990; Buchmann et al., 1989; Huang et al., 1989).

Most commercial database systems use the two-phase locking protocol (Eswaran et al., 1976) for concurrency control. The two-phase locking protocol is preferred over other methods for concurrency control (Reed, 1978; Kung and Robinson, 1981; Bernstein and Goodman, 1981) due to its simplicity and ease of implementation. Unfortunately, the blocking behavior of locking protocols can greatly degrade the performance of real-time database systems due to a phenomenon called *priority inversions* (Sha et al., 1990). Recent performance studies (Haritsa et al., 1990*a*, 1990*b*) have shown that some variants of the optimistic protocol (Kung and Robinson, 1981) outperform two-phase locking in real-time databases where transactions have *firm* deadlines. The authors point out that transaction blocking in the two-phase locking protocol results in unpredictable delays, causing transactions to miss their deadlines.

The research presented in this article is motivated by the following two facts: the popularity of the locking approach in most database systems, and the potential of attaining superior performance with optimistic or non-blocking concurrency control protocols in real-time databases. Optimistic protocols have failed to migrate to commercial database environments. We propose a new variant of the locking approach, referred to as ordered sharing (Agrawal and El Abbadi, 1990), in real-time databases with firm deadlines. Ordered sharing can be used to eliminate blocking of read and write operations. However, transactions may be subject to delay at commitment. In traditional databases, this delay potentially could result in poor response time for transactions. However, in real-time databases, timeliness

in meeting a transaction's deadline, and not response time, is the crucial factor. We can exploit the slack of a delayed transaction to complete the execution of any transactions causing the delay. To terminate, a delayed transaction that reaches its deadline may either have to abort itself or abort a lower priority transaction that has not yet completed. To summarize, our approach is to eliminate blocking of read and write operations, and to exploit any available slack in a transaction to improve the overall performance of the system by decreasing the number of transactions that miss their deadlines.

In this article, we start by reviewing some of the protocols that have been proposed for real-time database systems. We then introduce the locking primitive that extends standard locking, and describe several locking protocols for real-time databases. These protocols address different failure and recovery aspects of database systems. The issue of recovery has not been fully addressed by previous real-time database designs, especially those that use the optimistic approach for concurrency control. The rest of the article presents a simulation model based on Carey (1983) and on simulation results that demonstrate the superiority of our approach over the two-phase locking protocol used in real-time databases. We also analyze different aspects of the proposed protocols under various workloads.

2. Real-time Databases

A *database* is a collection of *objects*. Users interact with the database by invoking *transactions*. A transaction is a sequence of read and write operations that are executed atomically on the objects. The execution of a transaction must be *atomic* (i.e., a transaction either *commits* or *aborts*). Finally, a transaction is guaranteed to be *correct* (i.e., it maps the database from one consistent state to another consistent state). The execution of a set of transactions is modeled by a structure called a *history*. A history is *correct* if it is serializable (Bernstein et al., 1979; Papadimitriou, 1979). All protocols considered in this article ensure serializability. To ensure that aborting a transaction does not influence previously committed transactions, we must require that for every transaction T that commits, its commit operation follows the commit of every other transaction from which T reads. Such executions are called *recoverable* (RC; Hadzilacos, 1988). Recoverability, however, does not guarantee freedom from cascading aborts. Cascading aborts occur when a transaction reads from another uncommitted transaction that later aborts, forcing the former transaction to abort as well. Cascading aborts can be prevented by requiring transactions to read only committed values. Executions that satisfy this requirement are said to *avoid cascading aborts* (ACA; Hadzilacos, 1988). Finally, if the database uses *in-place* updating, it is convenient to implement the abort mechanism by restoring the *before-images* of all aborted writes (Verhofstad, 1978; Härder and Reuter, 1983; Bernstein et al., 1987). To use before-images to eliminate the effects of aborted transactions, read and write operations on a data object x must be executed only on committed values. Executions with this property are called *strict* (ST; Hadzilacos, 1988).

Many widely used concurrency control protocols use *locking* as a basic primitive for synchronization. Traditionally, there are two types of relationships between locks: shared and non-shared. For example, read locks can be shared but a write lock cannot be shared with any other lock. Transactions in real-time databases have *deadlines* associated with them, which can be used to assign priorities to transactions (priorities can be assigned based on other criteria, e.g., criticalness; Buchmann et al., 1989; Huang et al., 1989). In this article, we study concurrency control protocols for real-time databases with firm deadlines and a transaction with an earlier deadline is considered to have a higher priority over a transaction with a later deadline. Litwin and Shan (1991) proposed a concurrency control mechanism for heterogeneous databases based on value dates that are similar to deadlines. Transactions are assigned value dates that are used for concurrency control and transaction termination. The value dates-based protocol has some similarities with real-time concurrency control protocols. We now discuss some real-time concurrency control protocols that are related to our work.

Strict two-phase locking (Eswaran et al., 1976) is the most widely accepted concurrency control protocol. One especially undesirable property of strict two-phase locking in real-time databases is that a low priority transaction may block a high priority transaction; this phenomenon is called *priority inversion* (Sha et al., 1990). Therefore, in real-time databases, this protocol is augmented with a high priority conflict resolution scheme to ensure that high priority transactions are not blocked by low priority transactions, thus avoiding priority inversions. This two-phase locking protocol is referred to as 2PL-HP (Abbott and Garcia-Molina, 1988) and can be summarized as follows:

1. A transaction T must obtain read (write) locks before executing read (write) operations. If T's lock has a non-shared relationship with locks held by any transaction, and if all such transactions have a lower priority than T, then they are aborted and T can acquire its lock. Otherwise, T is blocked until the locks are released by the higher priority transactions.
2. Transactions release all their locks at commitment.

When locks with two types of relationships (shared and non-shared) are used, there are three types of blocking that can occur in the system: *read-write blocking* occurs when a transaction holds a read lock on an object and a lower priority transaction requests a write lock on the same object; *write-read blocking* and *write-write blocking* can be defined similarly. In 2PL-HP, a blocked transaction has to wait until all higher priority transactions holding conflicting locks commit. Furthermore, this protocol may suffer from *wasted restarts* (i.e., when a high priority transaction aborts a lower priority one and is itself later aborted). Note that deadlocks may also contribute towards wasted restarts. In this article, however, we only consider wasted restarts that are not due to deadlocks. Sha et al. (1991) proposed a locking-based protocol that avoids the blocking of high priority transactions (and thus priority inversions) for at most the duration of a single embedded transaction.

Haritsa et al. (1990*a*) proposed a variant of the optimistic protocol (Kung and Robinson, 1981) for real-time database. The proposed protocol is referred to as the *optimistic protocol with broadcast commit* (OPT-BC; Menasce and Nakanishi, 1982), and is shown to have better performance in real-time databases than the 2PL-HP protocol. The protocol can be summarized as follows. Transactions are allowed to execute without any synchronization until they reach their commit point, at which time they enter a validation phase. At this point, the validating transaction broadcasts a request that forces the abort of other uncommitted running transactions with which it conflicts. This implies that a validating transaction always commits. Furthermore, this protocol does not suffer from any wasted restarts, since a transaction that forces the restart of another transaction is guaranteed to commit.

In OPT-BC, the relative priorities of transactions are not taken into account when a transaction forces the abort of another transaction. Hence, it is possible for a lower priority transaction to abort a higher priority transaction if the former reaches its validation phase first. The *optimistic wait protocol* (OPT-WAIT; Haritsa et al., 1990*b*) was designed to overcome this problem by delaying a validating transaction from committing if a higher priority transaction is in the current set of uncommitted transactions with which it conflicts. Once there are no such transactions executing, the transaction aborts any lower priority uncommitted transactions with which it conflicts. An extension of OPT-WAIT is the OPT-WAIT-50 protocol, where a validating transaction waits only if $\geq 50\%$ of the currently uncommitted transactions have higher priority. The main purpose of this waiting is to detect when waiting is beneficial, in terms of giving preference to high priority transactions, versus no waiting, and its advantages in terms of avoiding late restarts for low priority transactions and a possible increase in the number of conflicts. Haritsa et al. (1990*a*) analyzed these variants of the optimistic approach and presented simulation results indicating that, under conditions of low data contention, delaying the validation of low priority transactions results in improved performance. On the other hand, under conditions of high data contention, OPT-WAIT-50 provides the best overall improvement in performance.

Huang et al. (1991) developed a locking variant of the optimistic concurrency control protocol and compared its performance with the class of two-phase locking protocols for real-time databases. Some of their results do not completely agree with the simulation results of Haritsa et al. (1990*a*, 1990*b*), which may be due to the differences in the simulation models and in the physical implementation schemes. However, both studies indicate that transaction blocking is the main disadvantage in adapting two-phase locking to real-time databases.

Lin and Son (1990) described a concurrency control protocol that uses a mixture of the locking and optimistic protocols. A transaction executes in three phases: a read phase, a wait phase, and a write phase. During the read phase, a transaction obtains read and write locks, and executes read operations, but performs write operations in its private space. Low priority transactions are blocked if a higher priority transaction holds a conflicting lock (except in the case of write operations

where write locks do not conflict, since no write operations are performed during this phase). A high priority transaction requesting a write lock aborts a lower priority transaction with a read lock on the object. However, if the higher priority transaction requests a read lock, on which a lower priority transaction has a write lock, both transactions are allowed to hold locks. During the wait phase, a transaction must wait for all higher priority transactions to commit. After committing, a transaction performs all its write operations, and write operations are executed in accordance with the serialization order between transactions (a timestamp for this purpose is assigned during the commit phase). In general, the protocol dynamically adjusts the serialization order between transactions in favor of higher priority transactions. This flexibility is achieved due to the deferred update approach, which allows high priority read operations to be serialized before lower priority write operations that may have already been executed.

Kim and Srivastava (1991) proposed an alternative approach, which uses multiversion concurrency control to reduce the number of rejected transactions and, thus, improve the overall performance of real-time database systems. Two approaches were proposed, one based on the two-version two-phase locking protocol (2V2PL) (Bayer et al., 1980; Stearns and Rosenkrantz, 1981), and another based on the multi-version two-phase locking protocol (MV2PL; Chan et al., 1982). The main advantage of these protocols is that priority inversion due to blocking can be eliminated. In particular, the read operation of a high priority transaction can always read the committed version of the object. A write operation creates a new version of the object and, thus, write operations in the MV2PL protocol never block each other. As in the optimistic approach, before committing, transactions must validate that no concurrent uncommitted transactions have executed write operations. A simulation study was performed and it demonstrated that the use of multiple versions can improve the performance of the concurrency control protocols in real-time databases.

Sha et al. (1991) proposed a real-time locking protocol to deal with the priority inversions problem that arises in real-time databases. The protocol employs the notion of *priority inheritance*, in which a lower priority transaction blocking a higher priority transaction inherits the priority of the latter to avoid priority inversions (Sha et al., 1990). In addition, they proposed a priority ceiling protocol by associating three attributes with each object in the database: *write_priority*, *absolute priority*, and *r/w priority* ceilings. Based on these parameters, they imposed certain restrictions on transactions when they attempted to access these objects. Although the proposed protocol is free from deadlocks, it has significant overheads for making scheduling decisions. In particular, a transaction cannot read/write lock a data object unless its priority is higher than the r/w priority of all data objects locked by other transactions. This rule is referred to as the *ceiling* rule. In summary, the proposed protocol has desirable features (i.e., no priority inversions and absence of deadlocks), but its applicability is limited to databases with a small number of objects. For large real-time databases, the above protocol is not very practical since it tags three extra

attributes with each object, and it requires a global search to enforce the ceiling rule.

3. Locking Protocols with Ordered Sharing

Agrawal and El Abbadi (1990) introduced a new locking primitive that allows a new relationship, referred to as *ordered sharing*, between locks. The new relationship provides mutual exclusion during operation execution but, unlike the traditional locking approaches, it does not require mutual exclusion after an operation has been executed. Instead of exclusion, the new relationship between locks captures the relative order of operations executed by concurrent transactions. Ordered sharing can be used with two-phase locking to eliminate the three types of blocking: read-write, write-write, and write-read blocking. For example, to eliminate read-write blocking, a transaction T_j can be granted a write lock on an object even if another transaction T_i holds a read lock on the same object. We say that there is an *ordered shared relationship* from T_i's read lock to T_j's write lock. Similarly, write-read blocking can be avoided by granting read locks with an ordered shared relationship with respect to write locks. Finally, write-write blocking can be eliminated by granting write locks with an ordered shared relationship from the previous write locks on the same object. To ensure serializability, protocols with ordered sharing must observe the following rule:

> *Ordered Sharing Rule:* If T_j acquires a lock with an ordered shared relationship with respect to a lock held by another transaction T_i, the corresponding operation of T_j must be executed after that of T_i. Furthermore, T_j cannot commit until T_i terminates (i.e., commits or aborts).

The ordered shared relationship can be interpreted as allowing the constrained sharing of locks in the following manner. If two operations acquire locks with an ordered shared relationship between them, the first lock excludes the second operation from executing until the first operation has been executed. Once the operation has been executed, the second operation is executed and the order of operation execution is the same as the order of lock acquisition. In this sense, mutual exclusion between operations is for a short duration only, instead of for longer periods as is necessary when non-shared relationships are used by locking protocols for executing conflicting operations. If ordered sharing is used to eliminate blocking, transactions may be delayed at commit. However, this delay does not block other transactions from executing read and write operations.

We now describe the two-phase locking protocol with ordered sharing (2PL-OS) adapted for real-time databases. In particular, we assume that we have shared relationships between read locks and ordered shared relationships between the remaining three types of conflicts between locks. Also, the *update-in-place* policy (Gray et al., 1981; Härder and Reuter, 1983) is used to execute operations. The

protocol can be summarized as follows:

1. Transactions acquire locks before executing operations, and release all their locks at commit (abort) as in strict two-phase locking.
2. When a transaction T is ready to commit, it waits until either the ordered sharing rule is satisfied or it reaches its deadline. In the former case it commits whereas, in the latter case, it tries to commit by aborting all preceding transactions with which it has an ordered shared relationship.[1]
3. When a transaction T aborts, it releases all its locks and causes the abort of all transactions which read values written by T. (Therefore, when a transaction decides to commit at its deadline, it may have to abort if it had read uncommitted data.)

This protocol is useful for real-time databases since it does not block any read and write operations from executing. There is a possibility of delay when transactions commit, but this does not result in performance degradation since, in real-time databases, timeliness of transactions is of greater value than the response time of transactions. Furthermore, this delay can be exploited to execute other transactions within a delayed transaction's slack. Unfortunately, 2PL-OS suffers from the undesirable phenomenon of cascading aborts which, in turn, may result in wasted restarts. We have argued elsewhere that locking protocols that suffer from cascading aborts have poor performance (Agrawal et al., 1992; 1994). In the following, we develop several variants of 2PL-OS that do not suffer from the problem of cascading aborts.

In the first variant, cascading aborts are avoided by retaining write-read blocking (i.e., ordered sharing is not allowed from write locks to read locks). Hence, a non-shared relationship from write locks to read locks is used in this protocol, which is referred to as ACA 2PL-OS (avoid cascading aborts 2PL-OS). ACA 2PL-OS is a hybrid of 2PL-HP, when locks with non-shared relationships are used, and 2PL-OS, when locks with ordered shared relationships are used. In particular, when a transaction tries to acquire a read lock on an object, and all writers have lower priorities, they are aborted and the transaction can acquire a read lock; otherwise, the transaction is blocked. For the other two types of conflicts (i.e., read-write and write-write) locks with ordered sharing are used and transactions must adhere to the ordered sharing rule. Since write locks are held until commit, transactions cannot read uncommitted data. Hence, all executions resulting from this protocol avoid cascading aborts and, therefore, can be used for real-time databases.

The second variant avoids cascading aborts in 2PL-OS by exploiting the before-images of objects and is referred to as the two-phase locking protocol with ordered sharing and before-images, 2PL-OS/BI (Agrawal et al., 1992). Our approach is similar to the one used by the multi-version two-phase locking protocols (Bayer et

1. Later in the article, we investigate the impact of aborting the delayed transaction itself.

al., 1980; Stearns and Rosenkrantz, 1981). When a transaction T executes a write operation on an object x, it creates a new (uncommitted) value for x. The original (committed) value of x is referred to as the *before-image* of x. When T commits, the before-image is discarded and the new value becomes the committed value of x. If T aborts, the before-image of x is used to restore x to its prior state. Since the protocol allows multiple writers to execute concurrently, the state of the object is represented by a single committed version and several uncommitted versions corresponding to the different values written by each uncommitted transaction. When a transaction reads an object, instead of reading the value written by an uncommitted transaction, it always reads the current committed version of the object, thus avoiding the possibility of cascading aborts. Hence, there is a reversal of the ordered shared relationship between the two transactions. In particular, write-read blocking is eliminated by allowing the reader to read committed data and requiring an ordered shared relationship from the read lock to the write lock. Hence, the reader must commit before the writer as mandated by the ordered sharing rule. This non-restrictive 2PL-OS protocol uses ordered sharing to eliminate write-read blocking and uses before-images to avoid cascading aborts. 2PL-OS/BI has all the desirable properties of 2PL-OS, especially the property of allowing other transactions to execute within the slack of a committing transaction. Deadlines of transactions are used only to force termination if transactions have reached their deadlines and are delayed due to the ordered sharing rule.

2PL-OS/BI does not suffer from the problems of either *wasted* or *mutual* restarts (Haritsa et al., 1990*b*). A wasted restart occurs when an executing transaction is aborted by another transaction that later misses its deadline. Since only committing transactions can cause restarts of other transactions in 2PL-OS/BI, all restarts are useful. ACA 2PL-OS, on the other hand, may suffer from wasted restarts since a reader may abort a lower priority writer, and later the reader itself aborts. Since priorities are assigned when transactions are created and do not change during their lifetimes, our protocols do not suffer from the problem of mutual restarts.

Both ACA 2PL-OS and 2PL-OS/BI result in executions that avoid cascading aborts but are not strict. For this class of executions, the recovery scheme based on value-logging and restoring before-images of aborted transactions can give rise to inconsistencies. We discussed a recovery implementation for such protocols (Agrawal and El Abbadi, 1991; Alonso et al., 1994), which is a simple variation of the standard log-based recovery (Härder and Reuter, 1983; Mohan et al., 1992). To continue using the standard recovery scheme based on before-images and value-logging, 2PL-OS/BI can be restricted so that it accepts strict histories (Hadzilacos, 1988). In strict histories, at most one uncommitted version of a data object exists at any time. This variant, referred to as ST 2PL-OS/BI, does not permit ordered sharing between concurrent writers. Instead, a non-shared relationship is used between conflicting write locks of different transactions. In particular, when a transaction tries to obtain a write lock on an object, and another transaction with a lower priority has a write lock on the same object, then the latter is aborted and the former is granted the

lock. Otherwise, the requesting transaction is blocked. For other types of lock conflicts the rules of 2PL-OS/BI are used. The advantage of ST 2PL-OS/BI is that it does not require any modifications to the underlying recovery architecture.

4. Illustrative Examples

In this section, we illustrate the advantages and tradeoffs between the two-phase locking protocols with ordered sharing, and various other protocols that have been proposed for real-time database systems. The purpose of this section is to highlight the main features of our protocols, in comparison with the other protocols.

Consider two transactions T_5 and T_7, where both transactions update objects x and y. For simplicity, we assume that each transaction needs 4 time units to execute, and that T_5 has a firm deadline at time 5, while T_7 has a firm deadline at time 7 (i.e., T_5 has a higher priority than T_7). We start by showing the execution of these two transactions, assuming a standard strict two-phase locking protocol. If transaction T_7 starts at time 0 by locking x and y, and T_5 starts at time 1, then the higher priority transaction T_5 is blocked because T_7 holds conflicting locks on objects x and y. When transaction T_7 terminates at time 4, it is already too late for T_5 to execute (it needs 4 time units, but its deadline is at time 7). Thus, the higher priority transaction does not make its deadline. The 2PL-HP protocol (Abbott and Garcia-Molina, 1988) overcomes this problem by aborting the lower priority transaction T_7, when T_5 requests its locks on x. However, when T_5 commits at time 5, it is too late for T_7 to execute; thus, the lower priority transaction can not make its deadline.

The OPT-BC protocol (Menasce and Nakanishi, 1982; Haritsa et al., 1990*a*) executes operations without requiring them to obtain locks. Hence, in the above scenario, transaction T_7 starts executing at time 0, and transaction T_5, which arrives at time 1, is allowed to execute concurrently with T_7. When T_7 reaches its validation phase, it forces the abort of T_5, since T_5 conflicts with T_7 on x and y. Hence, T_7 commits at time 4, and it is too late for T_5 to execute. As with the 2PL-HP, the OPT-WAIT protocol was designed to avoid the aborts of high priority transactions. In this case, when T_7 reaches its validation phase, it waits for the higher priority transaction to terminate. Unfortunately, when T_5 reaches its validation phase (at time 5) it forces the abort of transaction T_7, and now it is too late for T_7 to execute.

If ordered sharing is used in the above scenario, the two transactions may be allowed to commit if the operations on the two objects are executed in the same order. In particular, assume that when T_7 starts at time 0, it obtains locks on objects x and y, and then, when T_5 starts at time 1, it obtains locks on objects x and y with ordered shared relationships with respect to the locks of T_7. In this case, when transaction T_7 tries to commit at time 4, it satisfies the ordered sharing rule (it is waiting for no transaction), and hence is allowed to commit. When at time 5, T_5 tries to commit, the ordered sharing rule also holds (all transactions that it has been waiting for have already committed). Hence, both the high priority

transaction T_5 and the low priority transaction T_7 commit and meet their deadlines. Of course, if the order in which locks are obtained on the two objects is not the same, a deadlock will occur, and one of the two transactions will be aborted (in the simulation studies described in the next section, the lower priority transaction is aborted).

Our protocols exploit ordered sharing to avoid the blocking of operations, with the potential possibility of blocking commit operations. Furthermore, by delaying the commitment of some high priority transactions until it is absolutely necessary to commit (if the transactions reach their deadlines), lower priority transactions may be allowed to commit in the slack of the higher priority transactions. Consider another example where T_7 needs 4 time units and has deadline at time 7 and T_{10} needs 6 time units and has deadline at time 10. If T_{10} starts at time 0 and T_7 starts at time 1, then T_7 will try to commit at time 5. In this case, the ordered sharing rule may not hold if T_{10} had acquired a lock on some object before T_7. Instead of forcing the abort of T_{10}, transaction T_7 waits. At time 6, T_{10} commits and the ordered sharing rule is satisfied and, hence, T_7 can also commit since all transactions for which it is waiting have committed. Hence, T_{10} has exploited the available slack between the time when T_7 terminates and its deadline. Thus, both transactions commit and meet their deadlines. Note that if T_{10} needed 8 time units (instead of 6), then T_7 reaches its deadline at time 7 and, since it has higher priority than T_{10}, it forces T_{10} to abort.

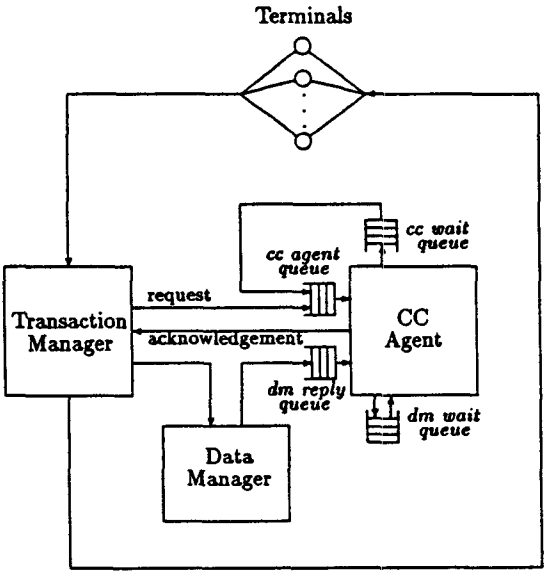
5. Simulation Model

To evaluate the performance of the proposed locking protocols for real-time databases, a database simulation model based on Carey (1983), Agrawal et al. (1987*a*), and Haritsa et al. (1990*a*) was developed. This simulation model uses the SIMSCRIPT II.5 language (Law and Larmey, 1984; Russell, 1983, 1987) and implements a centralized database. It is divided into three main components: a Transaction Manager (TM), a Concurrency Control Agent (CCA), and a Data Manager (DM). The TM is responsible for issuing lock requests, the CCA schedules these requests according to the specifications of the protocol, and the DM is responsible for granting access to the physical data objects.

5.1 Logical Queuing Model

The logical simulation model, shown in Figure 1, represents a closed queuing model of a single-site database system. There are a variable number of terminals, *num_terms*, which effectively control the maximum multiprogramming level of the system. A terminal initiates a transaction, and then is delayed from submitting its next transaction for a Poisson-distributed interval, the *inter_arrival_delay* parameter. Each transaction has an associated deadline that is calculated as (Haritsa et al., 1990*a*):

Figure 1. Logical Queuing Model

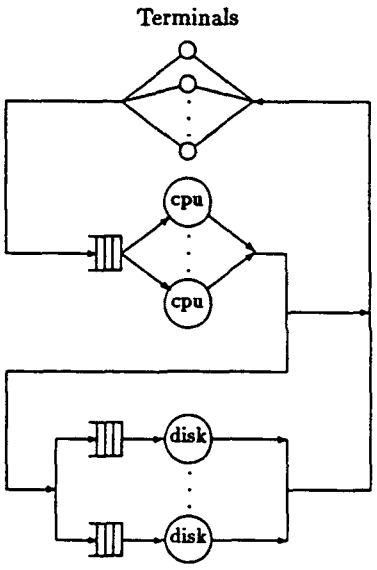


$$deadline = txn\_start\_time + (slack\_factor * estimated\_total\_txn\_time)$$

Slack_factor is an input parameter that controls the tightness or looseness of the deadlines, and *estimated_total_txn_time* is the estimated total service time for the transaction (this is a function of I/O and CPU time needed to process all operations of a transaction). For example, with slack factor two, a transaction has double its estimated time to complete execution. The TM issues both the lock requests and the actual database operations for each transaction. In addition, the TM determines if a transaction misses its deadline. The input parameter *knows_txn_reqs* indicates whether the system is aware of the amount of time a transaction will need to finish processing. If *knows_txn_reqs* is false, then the TM can only determine missed deadlines when they expire (referred to as *not tardy;* Abbott and Garcia-Molina, 1988). Otherwise, the TM can determine at each database operation whether a transaction should be aborted early because it can no longer make its deadline (referred to as *feasible deadlines;* Abbott and Garcia-Molina, 1988). In the majority of our experiments, we use the not-tardy policy, because it makes fewer assumptions about the system's capabilities.

The CCA processes lock requests received through the *cc_agent_queue*. When a lock request is received, it is possible that the requested item is already locked by another transaction and that the requester is given a non-shared relationship with the lock holder (e.g., in the cases of 2PL-HP, ACA 2PL-OS, and ST 2PL-OS/BI). In this scenario, the deadlines of the conflicting transactions are compared. If the requesting transaction has an earlier deadline, the lock holder is aborted

Figure 2. Physical Queuing Model



and the lock is granted to the requester. Otherwise, the requester is placed in the priority queue, *cc_wait_queue*, sorted by deadlines. This scheduling policy is referred to as *earliest deadline,* and has the advantages of simplicity and relatively good performance (Abbott and Garcia-Molina, 1988). Delayed lock requests will be rescheduled once the conflicting operations have released their locks. When a lock request can be granted, an acknowledgement is sent back to the TM, which then forwards the database operation to the Data Manager. When the DM executes database operations it must adhere to the order in which lock requests were granted by the CCA. This order is preserved through the use of the *dm_wait_queue* and the *dm_reply_queue* through a handshake mechanism (Agrawal et al., 1994). Locks are released when transactions commit or abort.

The use of deadlines to schedule operations prevents the formation of deadlocks in 2PL-HP. Unfortunately, deadlocks may form in all protocols with ordered sharing. For this reason, we use a deadlock detection strategy based on wait-for graphs. Whenever a deadlock is detected, the transaction with the latest deadline is chosen as the victim and aborted. The rationale for this choice is based on the fact that the transaction with the latest deadline has the lowest priority among the deadlocked transactions.

5.2 Physical Queuing Model

Underlying the logical model of Figure 1 are two physical resources, the CPU and the I/O (i.e., the disk) resources. A certain amount of resource overhead is associated with each lock request and each database access. Lock requests require

only CPU service while database accesses require both CPU and I/O services. The physical queuing model, shown in Figure 2, is very similar to the one used by Agrawal et al. (1987*a*, 1987*b*) and Carey et al. (1990) in which the parameters *num_cpus* and *num_disks* specify the number of CPU servers and the number of I/O servers. The CPU servers are modeled as a pool of servers, all identical and serving a common CPU queue sorted by transaction priorities (deadlines). Thus, the CPU scheduling policy is based on transaction deadlines. Unlike the CPU servers, a separate queue is associated with each I/O server. When a transaction needs service, it randomly selects a disk (with all disks being equally likely) and waits in the I/O queue associated with the selected disk. I/O requests are also sorted by transaction priorities. That is, I/O scheduling is also based on transaction deadlines. The parameters *cpu_time* and *io_time* represent the amount of CPU and I/O time associated with reading or writing a data object. Both of these parameters are modeled as uniform distributions. The parameter *cc_req_delay* represents the amount of CPU time associated with servicing a concurrency control request (lock request), which is assumed to be a constant. A special flag *inf_res* is used to to override the use of *num_cpus* and *num_disks* to model the ideal environment with unlimited resources. When this flag is set, the simulation ignores the server queues, and transactions are delayed only for the amount of time associated with *cpu_time* and *io_time* (i.e., there is no waiting in the queues for access to the physical devices).

5.3 Transaction Generation

For each transaction, the sequence of operations and the data objects to be accessed are determined by the TM in a probabilistic manner. The size of the database is assumed to be *db_size*. It is also assumed that each transaction performs at most one read and/or write operation per data object. The transaction characteristics are determined by the transaction size, *txn_size*, the update transaction percentage, *update_txn_pct*, and the write operation percentage, *write_op_pct*, parameters. The transaction size represents the average number of operations performed by a transaction, the mean of a uniform distribution between *txn_size* $\pm$ 5. The update transaction percentage represents the percentage of transactions that will be *update transactions*. The write operation percentage determines what percentage of an update transaction's operations will be writes and has a uniform distribution of *write_op_pct* $\pm$ 20. Table 1 summarizes the parameters used in the simulation model. Note that *cpu_time* and *io_time* have a uniform distribution (i.e., *cpu_time* $\pm$ 3 and *io_time* $\pm$ 5.

5.4 Simulation Settings

In real-time databases, the most important metric is the percentage of transactions that miss their deadlines. In addition, the throughput rate, which is defined as the number of transactions successfully completed per second, is useful for analyzing the performance of the system. A form of the batch means method was used for

Table 1. Simulation Model Parameter Definitions

Parameter	Description
db_size	Number of objects in database
num_terms	Number of terminals
num_cpus	Number of cpus
num_disks	Number of disks
inf_res	Infinite resource flag
txn_size	Mean transaction size
update_txn_pct	Update transaction percentage
write_op_pct	Mean Write operation percentage
inter_arrival_delay	Transaction inter-arrival delay
cpu_time	Mean CPU time for accessing an object
io_time	Mean I/O time for accessing an object
cc_req_delay	CPU time for servicing a lock request
slack_factor	Slack Factor
knows_txn_reqs	System knows service requirements

the statistical analysis. Each simulation consisted of a minimum of four repetitions, each consisting of 2,000 seconds of simulation time. The first 200 seconds of each repetition were discarded to let the system stabilize after initial transient conditions. In general, we achieved 90% confidence intervals for our results. If a 90% confidence interval was not attained in four repetitions of the simulation, then additional runs were made. (We did not generate the excessive number of repetitions that would have been necessary to get 90% confidence intervals for very small values, e.g., miss percentages below 2%.) Table 2 provides a summary of the values chosen for the input parameters in all experiments.

6. Experiment Results

In this section, we present and analyze the results of the simulation experiments for protocols 2PL-HP, ACA 2PL-OS, 2PL-OS/BI, and ST 2PL-OS/BI. In each of the following experiments, the number of terminals, *num_terms*, is varied to include multiprogramming levels that are considered reasonable for actual database systems. This provides a wide range of operating conditions with respect to data contention (lock conflict) and resource contention (waiting for CPUs and disks). To evaluate the effect that resources have on the system, one CPU resource and two disk resources were chosen to represent one *resource unit* (Agrawal et al., 1987*a*, 1987*b*). Resource related experiments were performed by varying the number of resource units rather

Table 2. Simulation Model Parameter Values

Parameter	Value
db_size	1000 data objects
txn_size	20 operations
inter_arrival_delay	10 seconds
update_txn_pct	60 percent
write_op_pct	50 percent
cpu_time	12 milliseconds
io_time	35 milliseconds
cc_req_delay	3 milliseconds

than by individually varying the number of CPUs or the number of disks. By using this combination of CPUs and disks, resource utilization in the system turns out to be slightly I/O bound (disk utilization is slightly higher than CPU utilization; Agrawal et al., 1987*b*).

6.1 Effect of Multiprogramming Level

We first evaluate the effect of varying the multiprogramming level on the performance characteristics of the four locking protocols for real-time databases. The experiment is based on a system with four resource units where transactions have a slack factor of three. We assume that transactions are only aware of their deadlines but do not know of their exact requirements in terms of CPU and I/O time (i.e., the *not tardy* approach is used for executing transactions). We refer to these settings as the *baseline* experiment.

Figure 3 illustrates the throughput, miss percentage, number of total restarts, and number of useful restarts in the four protocols. We also plot the average arrival rate of transactions in the throughput graph of Figure 3. The throughput graph illustrates that 2PL-OS/BI, which does not incur any blocking, has the best performance among all the protocols. In particular, the maximum throughput for 2PL-OS/BI is 6.75 in comparison to 4.6 for 2PL-HP. This represents a 47% improvement over 2PL-HP. The maximum throughput for ACA 2PL-OS is approximately 5.75 (a 25% improvement) and for ST 2PL-OS/BI is approximately 6.35 (a 38% improvement). The thrash points for 2PL-HP, ACA 2PL-OS, 2PL-OS/BI, and ST 2PL-OS/BI are 75, 83, 95, and 95, respectively. After the thrash points, as the multiprogramming level is increased, the four protocols converge indicating that, in our protocols, aborts due to deadlocks dominate the system and the effect of "blocking" versus "non-blocking" is marginalized.

Figure 3. Baseline case: slack=3, resource units=4, not tardy policy

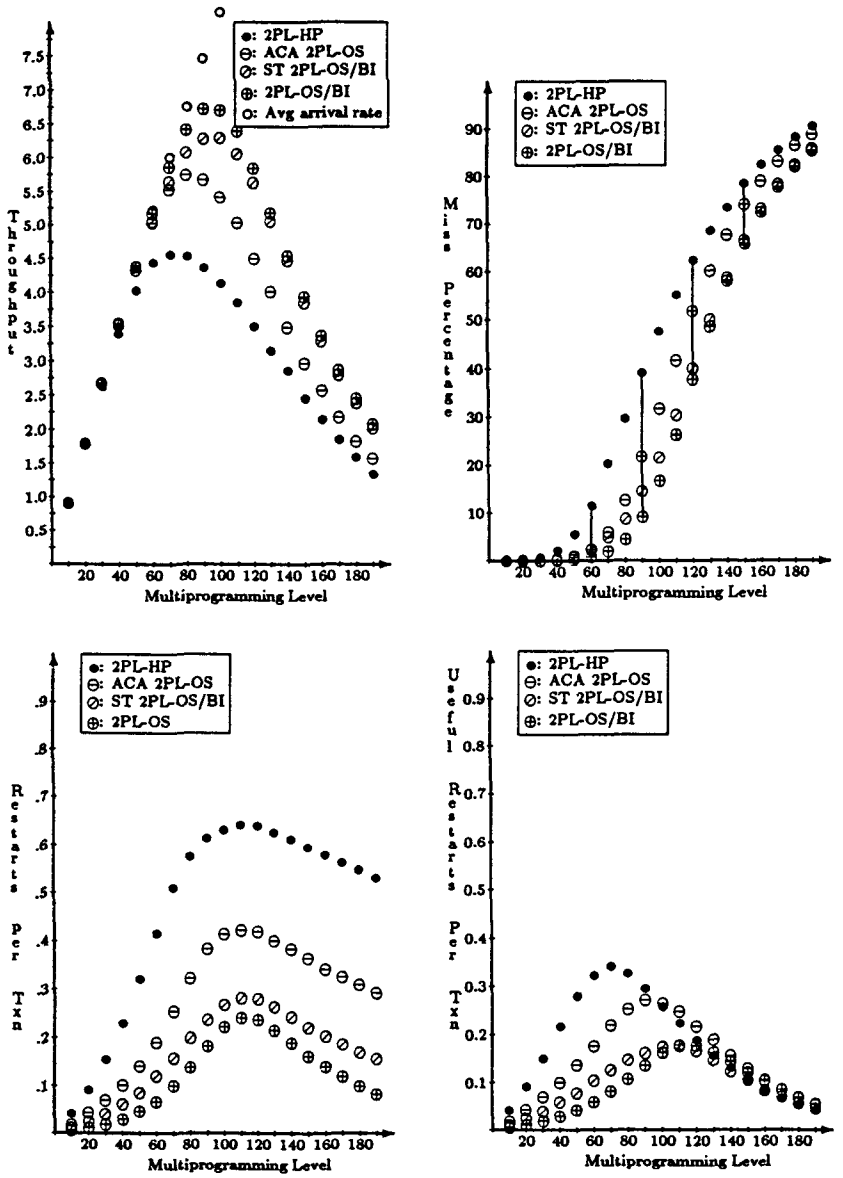


Table 3. Sample Illustration of Confidence Intervals for Multiprogramming Level of 80 Terminals

Protocol	Throughput			Miss Percentage		
	Mean	Interval†		Mean	Interval†	
2PL-HP	4.53	4.49	4.57	29.74%	29.15%	30.33%
ACA 2PL-OS	5.74	5.67	5.80	12.65%	11.5%	13.80%
ST 2PL-OS/BI	6.07	6.04	6.09	8.74%	8.20%	9.29%
2PL-OS/BI	6.41	6.39	6.44	4.49%	4.13%	4.86%

† Intervals with 90% confidence.

The differences among the protocols become more obvious when we examine the miss percentage graph in Figure 3. In particular, the rate of increase of transactions missing their deadlines is initially much sharper for 2PL-HP than it is for 2PL-OS/BI. At the point when 2PL-HP exhibits its maximum throughput (75 terminals), 25% of the transactions are already missing their deadlines. In contrast, only 3% of the transactions miss their deadlines in 2PL-OS/BI, around 7% do so in ST 2PL-OS/BI, and 9% miss in ACA 2PL-OS. At the thrash point of 2PL-OS/BI (95 terminals), the percentage of transactions that miss their deadlines is 12.5% for 2PL-OS/BI, 17% for ST 2PL-OS/BI, 27% for ACA 2PL-OS, and 44% for 2PL-HP. The miss percentages are indistinguishable in the four protocols at low multiprogramming level or equivalently at low data contention (10 to 30 terminals). However, at medium to high data contention (more that 40 terminals), 2PL-OS/BI misses significantly fewer transaction deadlines than 2PL-HP. At very high multiprogramming levels, the vast majority of transactions miss their deadlines under any protocol.

Note that, as in OPT-BC (optimistic broadcast; Haritsa et al., 1990*a*), only a committing transaction can generate restarts in 2PL-OS/BI. In 2PL-HP, ACA 2PL-OS, and ST 2PL-OS/BI, a transaction causing a restart may later abort. Thus, there is an increased likelihood of wasted restarts in protocols with blocking and this explains the superior performance of 2PL-OS/BI. In particular, Figure 3 illustrates the number of restarts per generated transaction in the protocols, and demonstrates that, in general, 2PL-OS/BI has fewer restarts than the other protocols. This can be explained by noting that 2PL-OS/BI, and to a lesser extent ST 2PL-OS/BI and ACA 2PL-OS, restart transactions only when it is absolutely necessary (i.e., transactions allow others to run in their slack). Figure 3 shows the number of *useful restarts,* which was defined by Haritsa et al. (1990*b*) as the restarts caused only by transactions that eventually commit. Recall that some aborts may occur due to deadlocks; this justifies the slight discrepancy between the total restarts and useful restarts in 2PL-OS/BI. Table 3 is a sample of interval range for the mean values plotted in the graphs. Note that we used the batch-means method to achieve 90% confidence in the indicated intervals. In general, the intervals were quite tight for the mean values resulting from all experiments.

Figure 4. slack=3, resource units=3(top)=5(bottom), not tardy policy

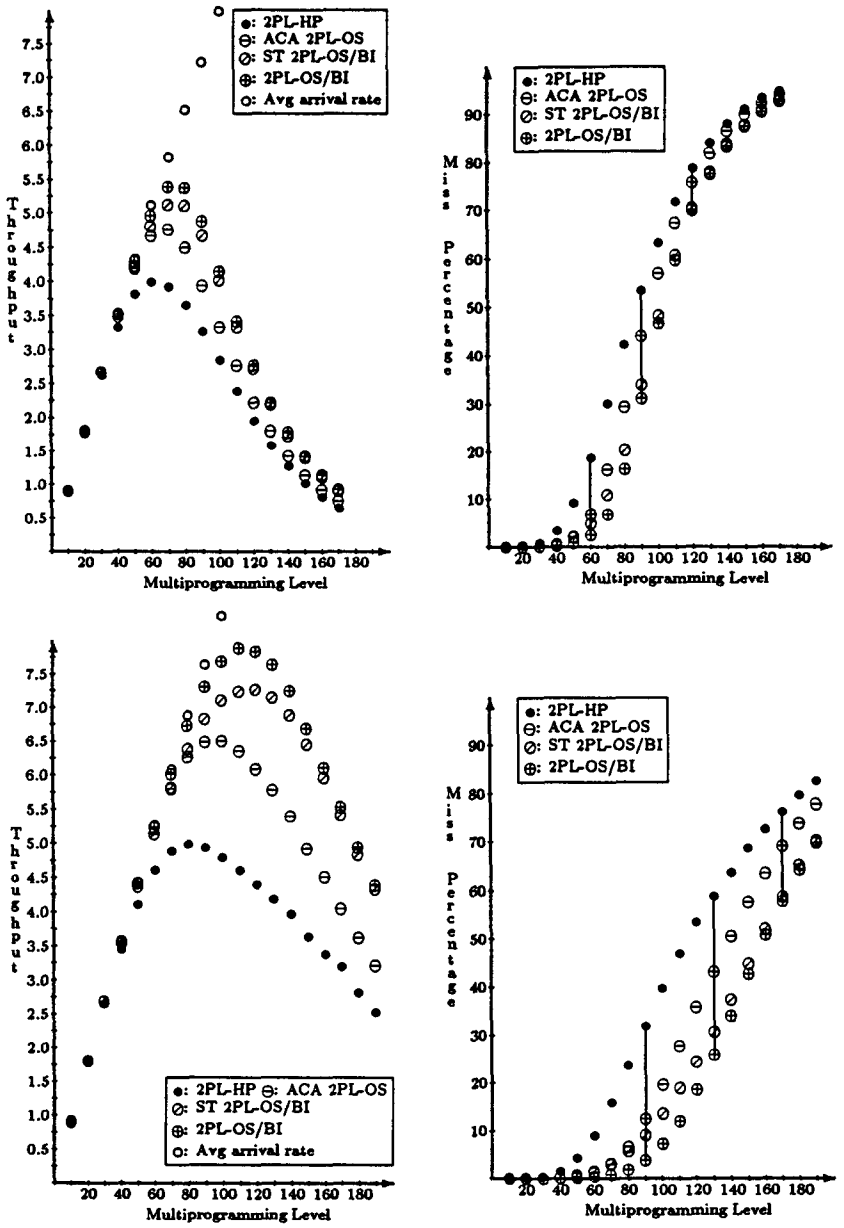
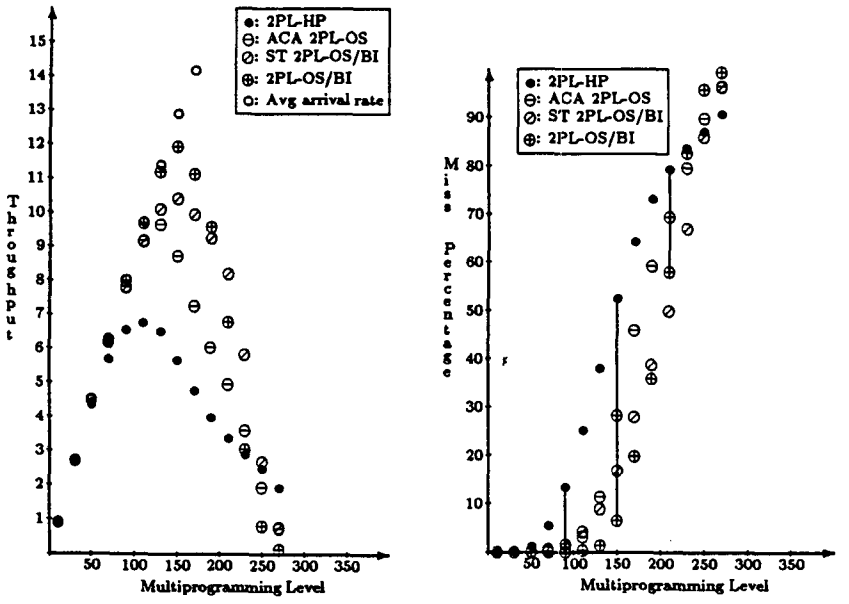


Figure 5. slack=3, unlimited resource units, not tardy policy



6.2 Effect of Resources

Figures 4 and 5 report the throughputs and miss percentages in the four protocols for a variable number of resources, with the remaining parameters the same as in the baseline experiment. The experiments were conducted with three, five, and unlimited resource units. When the number of resources is reduced from four to three, the relative improvements due to ordered sharing is less pronounced (see Figure 4). This is due to increased resource contention in the system. When the number of resources is increased from four to five, the throughputs and miss percentages in protocols 2PL-OS/BI, ST 2PL-OS/BI, and ACA 2PL-OS are significantly better than in 2PL-HP. In the case of five resources, all protocols with ordered sharing are able to meet many more deadlines than 2PL-HP. For example, 2PL-OS/BI does not miss any deadlines up to the multiprogramming level of 60 terminals, at which point 2PL-HP misses 10% of the deadlines. Similarly, ST 2PL-OS/BI and ACA 2PL-OS do not miss any deadlines up to 50 terminals. Once again, this is due to the desirable property of protocols with ordered sharing, which eliminates blocking at the expense of possible delays in transaction commitment. Hence, protocols with ordered sharing are particularly useful when there is less contention for resources in the system.

To further validate our hypothesis that removing resource contention results in better performance of 2PL-OS/BI, ST 2PL-OS/BI, and ACA 2PL-OS in real-time

databases, we conducted the baseline experiment with unlimited resources (see Figure 5). In this case, the percentage of transactions missing their deadlines in 2PL-HP is significantly larger than in all the other protocols at all but the highest multiprogramming levels (the rapid rise in missed deadlines after the thrash points of protocols with ordered sharing is due to the increase in deadlock formation). For example, at 110 terminals 2PL-OS/BI meets virtually all its deadlines, ST 2PL-OS/BI and ACA 2PL-OS miss 3% to 4% of their deadlines, and 2PL-HP misses more than 25% of its deadlines. This indicates that eliminating blocking for both data and resources can be significantly beneficial for protocols with ordered sharing in real-time databases. This is of special interest, because real-time systems frequently provide extra resources to handle peak load conditions or to provide for fault-tolerance (Haritsa et al., 1990*a*). Ordered shared locking protocols are better able to use these additional resources. Two-phase locking would fail to make use of them if data contention was already degrading performance.

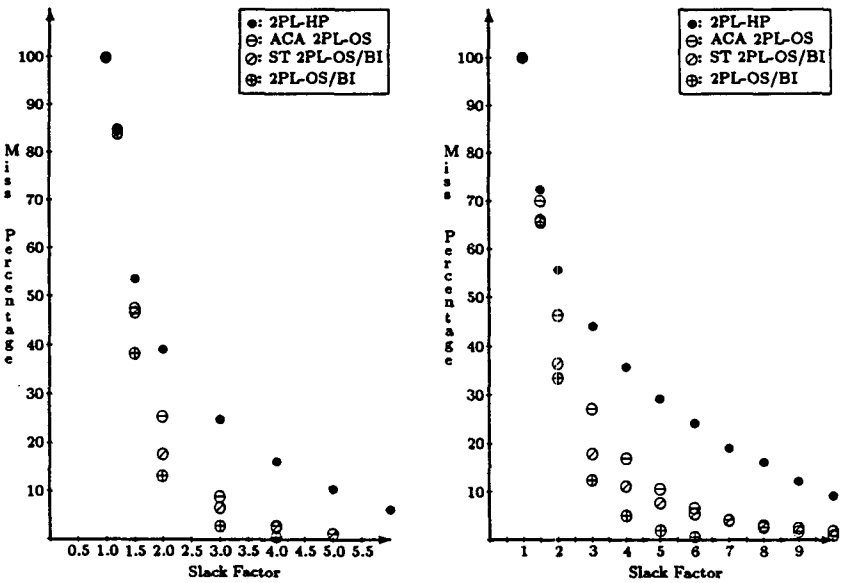
6.3 Effect of Slack

In this experiment, we evaluate the effects of varying the slack factor on the performance of the four protocols. In particular, the slack factor was varied from 1 to 9, and the experiment was conducted with four resource units and at two multiprogramming levels. We chose 75 terminals, which is the thrash point for 2PL-HP, and 95 terminals, which is the thrash point for 2PL-OS/BI. The results of this experiment are illustrated in Figure 6.

Figure 6 shows that when deadlines are very tight (i.e., when the slack factor is close to 1), the performance of all the protocols is indistinguishable. As the slack in deadlines is increased, protocols with ordered sharing start demonstrating superior performance. In general, they all miss significantly fewer deadlines than 2PL-HP beyond the slack factor of 1.5. For example, with 75 terminals and slack factor 4, 2PL-OS/BI has almost no transactions missing their deadlines, while 2PL-HP has about a 16% miss rate, and the other two have miss rates below 3%. As the slack is increased, all protocols asymptotically reach a miss percentage of 0. At the thrash point of 2PL-OS/BI, which is a heavy load for 2PL-HP, the miss percentage in 2PL-OS/BI becomes insignificant at about a slack factor of 6. 2PL-HP is still missing 24% of its transactions under the same conditions.

The improvement in performance of protocols with ordered sharing is due to the execution of transactions during the potential delay of other transactions' commitment. In particular, when the slack factor is greater than 1.5, we notice that 2PL-OS/BI, and to a lesser extent ST 2PL-OS/BI and ACA 2PL-OS, miss significantly fewer deadlines than 2PL-HP. This can be explained by considering the possible interactions between two transactions that execute conflicting operations on the same object. Assume that the deadlines of the two transactions, T_1 and T_2, are such that T_2 has a later deadline than T_1. First, consider the case when T_2 holds a lock on the common object, and later, T_1 requests a lock on the object. In 2PL-HP, T_2 is aborted and restarted since it has lower priority. In 2PL-OS/BI, on

Figure 6. Varying slack factor, 75 terminals(left), 95 terminals(right)



the other hand, T_2 is given a chance to terminate within the slack available for T_1. Furthermore, note that since T_1 is not blocked, it can acquire all its other locks and execute all its operations. When T_1 reaches its deadline, if T_2 has not terminated, T_2 is forced to abort and restart. If, however, the slack of T_1 is sufficiently large, both transactions may be able to commit within their respective deadlines without the need for restarts and without wasting resources. The significant decrease in missed deadlines for protocols with ordered sharing as the slack rises beyond 1.5 is caused by the increasing number of transactions that are able to commit within the slack of other transactions.

Second, consider the case where T_1 holds the lock, and T_2 later requests a lock. In 2PL-HP, T_2 is blocked and must wait until T_1 terminates, before proceeding to acquire locks and execute any operation. Furthermore, note that T_1 may eventually abort and, hence, this blocking may be referred to as *wasted blocking*. Since transactions that use ordered sharing do not block read and write operations, T_2 is allowed to progress, acquire its locks, and execute operations. Note, however, that T_2 may not commit before T_1 terminates. But this cannot have any undesirable side effects since T_1 has an earlier deadline than T_2 and, hence, T_1 must be out of the system by the time T_2 arrives at its deadline. In general, by not blocking read and write operations, ordered sharing allows transactions to advance and execute

all their operations and, hence, fully utilize the system resources. On the other hand, and as we have demonstrated, the delayed commitment does not affect the timeliness of transactions, and is actually beneficial to lower priority transactions that may commit during this delay.

6.4 Effect of Delayed Commitment

In this experiment, we examine the importance of delaying the commitment of a transaction to allow other transactions to finish in its slack. Ordered sharing benefits system performance both by reducing blocking and by using a transaction's slack to finish transactions that would otherwise be aborted. Specifically, when a transaction is ready to commit but has to wait due to ordered shared relationships with respect to other transactions, and delayed commitment is not employed, then it first aborts all transactions it is waiting for and then commits. For this experiment all parameters are kept the same as in the baseline experiment; the only change is in the immediate commitment of a completed transaction.

Figure 7 shows the throughputs of ST 2PL-OS/BI and 2PL-OS/BI with and without delaying a transaction's commitment. For comparison purposes, the throughput of 2PL-HP is also shown. Recall from Section 6.1 that 2PL-OS/BI showed a 47% improvement in maximum throughput over 2PL-HP. When delayed commitment is not used, 2PL-OS/BI results in a 33% increase over 2PL-OS/BI. The remaining 14% improvement is due to the delay of a transaction's commitment. ST 2PL-OS/BI without delayed commitment shows a gain of 30% over 2PL-HP, and with delayed commitment it shows a gain of 38%. The corresponding miss percentages are also illustrated in Figure 7. At 2PL-HP's thrash point of 75 terminals, the miss percentages are 25% for 2PL-HP, 8.5% for ST 2PL-OS/BI without delayed commitment, 7.5% for 2PL-OS/BI without delayed commitment, 6.7% for ST 2PL-OS/BI with delayed commitment, and 3% for 2PL-OS/BI with delayed commitment. As reducing missed deadlines is of primary importance in real-time database systems, the use of delayed commitment in 2PL-OS/BI is especially important. At this multiprogramming level, the number of missed deadlines has been reduced by more than half. At 2PL-OS/BI's thrash point of 95 terminals, the reduction is approximately 33% (from 20.7% without delayed commitment to 12.4% with the delay).

For ST 2PL-OS/BI, there is only a slight benefit gained by delaying transaction commitment. Clearly, the most significant gain is due to reducing read-write and write-read blocking. For 2PL-OS/BI, however, the use of delayed commitment has a more significant effect. We can explain this difference in terms of the frequency and type of operation conflicts that cause delays at commitment. In both of these protocols, read-only transactions will never be delayed at termination, because read operations can never be involved in an ordered shared relationship from a writer to a reader. Hence, delayed transactions must be update transactions. Furthermore, a delayed transaction will not have to wait for much time due to read-only transactions. Any read-only transaction will complete without being blocked or delayed. Therefore, an update transaction can be delayed substantially only by other update transactions.

Figure 7. Effect of Delayed Commitment (Baseline Settings: slack=3, resource units=4)

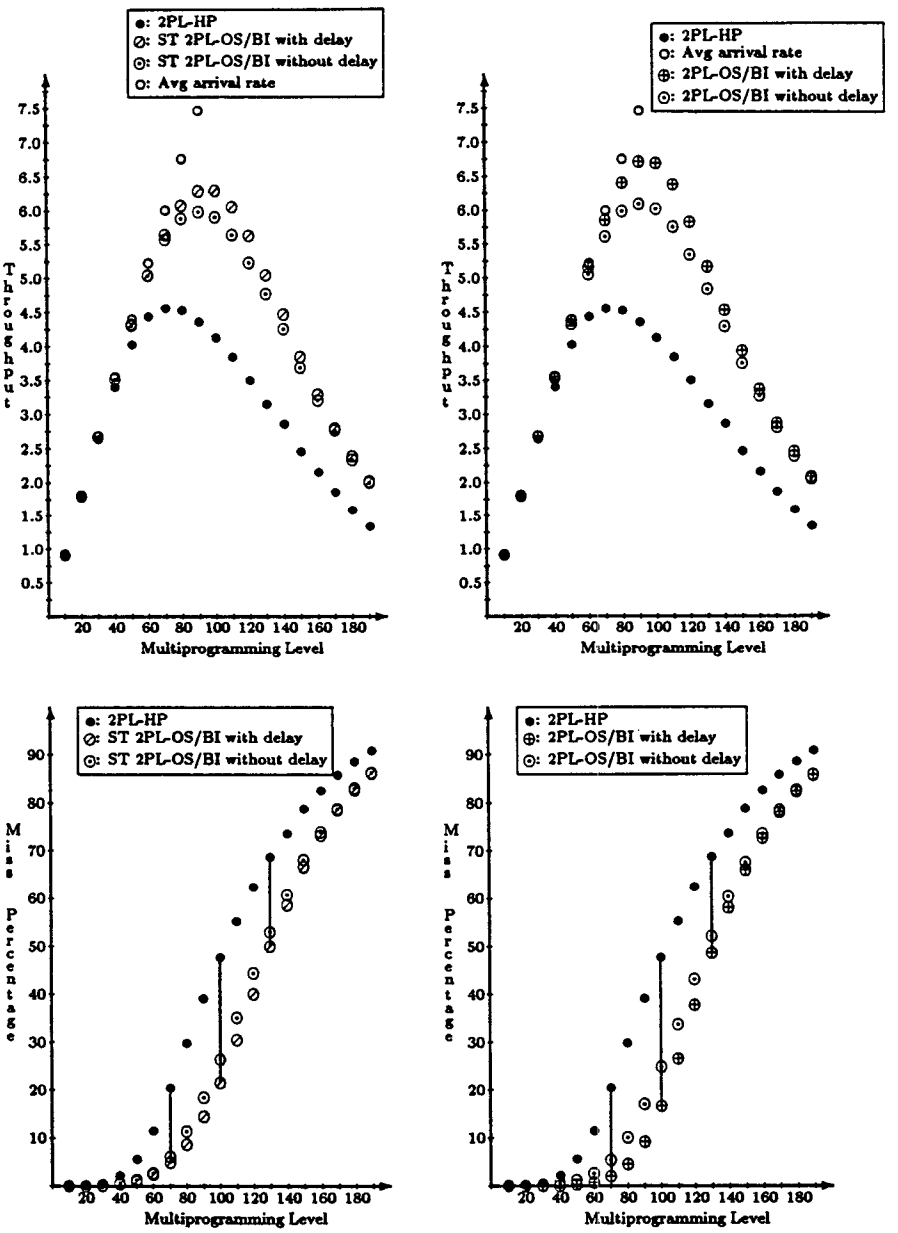
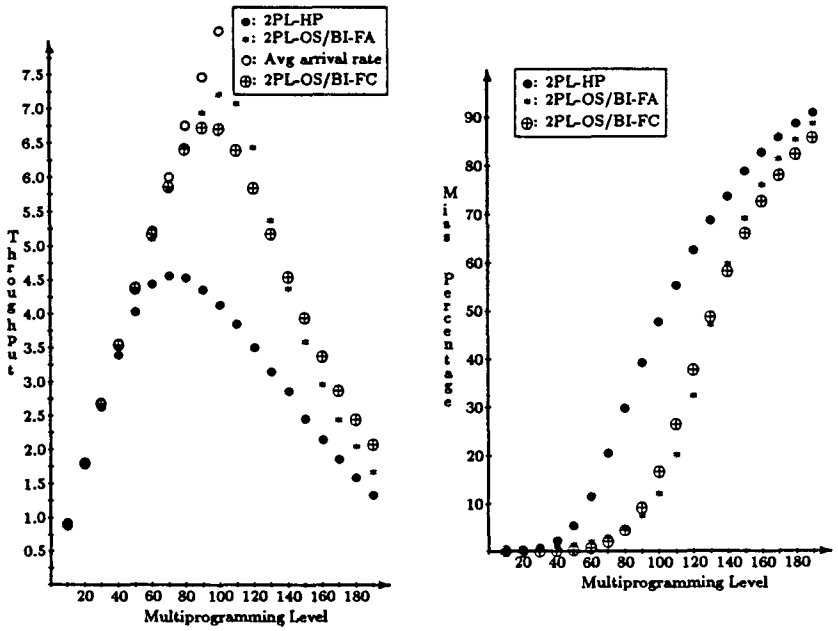


Figure 8. slack=3, resource units =3(top) =5(bottom), not tardy policy



Recall that ST 2PL-OS/BI uses blocking for write-write conflicts, while 2PL-OS/BI does not. This has two effects with respect to ST 2PL-OS/BI. First, there will be correspondingly fewer situations in which a transaction can reach its commit point while being delayed for termination by another transaction. Some transactions which would have used delayed commitment in 2PL-OS/BI will instead be blocked in ST 2PL-OS/BI. Furthermore, transactions that are blocked for some time and then are delayed upon termination will have wasted some of their slack during the periods when they were blocked. Hence, the available slack time to do useful work (i.e., the advantage of delayed commitment) may be much reduced. Second, transactions blocked by write-write conflicts will not be able to benefit from the slack in another transaction's delayed commitment. As a result ST 2PL-OS/BI does not benefit from delayed commitment as strongly as does 2PL-OS/BI.

6.5 Alternative Policies for Delayed Termination

The ordered sharing rule restricts the termination of transactions. In particular, if transaction T_i acquires a lock with an ordered shared relationship with respect to a lock held by another transaction T_j, then T_i cannot commit until T_j terminates. We call T_j a *predecessor* of T_i. When T_i is ready to commit, and if there is at least one predecessor, T_i must wait. When T_i reaches its deadline, if there are any predecessors, two different policies may be considered: *forced commit* policy (i.e., T_i aborts all predecessors and commits itself), and *forced abort* policy (i.e., T_i

aborts itself). Note that, with forced aborts and unlike with forced commits, the different ordered sharing protocols may suffer from wasted restarts. In particular, when a transaction reaches its deadline, it may be aborted due to a lower priority transaction, which itself is later aborted. We compare the performance of 2PL-OS/BI, ST 2PL-OS/BI, and ACA 2PL-OS with these two different policies.

Figure 8 illustrates the performance of 2PL-OS/BI with forced commit (2PL-OS/BI-FC) and forced abort (2PL-OS/BI-FA) policies in the baseline case. This figure shows that if the multiprogramming level is < 70 terminals, the performance of 2PL-OS/BI-FA is slightly worse than that of 2PL-OS/BI-FC. With the increase in multiprogramming level, the performance of 2PL-OS/BI-FA is slightly better than that of 2PL-OS/BI-FC. At the multiprogramming level corresponding to the maximum throughput of 2PL-OS/BI-FA, only 12.09% of the transactions miss their deadlines compared to 16.67% in 2PL-OS/BI-FC and 47.68% in 2PL-HP. Figure 9 shows the simulation results for a variable number of resources. When the number of resources is reduced, the margins between 2PL-OS/BI-FA and 2PL-OS/BI-FC become smaller due to increased resource contention. When the number of resources increases from four to five, the margins are larger. At the point of 120 terminals, 2PL-OS/BI-FA gives a throughput of 8.52 transactions per second, a 8.7% increase over 2PL-OS/BI-FC and a 93.6% increase over 2PL-HP. The miss percentage at this point is 13.4% for 2PL-OS/BI-FA, 18.6% for 2PL-OS/BI-FC and 53.4% for 2PL-HP. At the point of 130, the throughput of 2PL-OS/BI-FA is 8.5, a gain of 11.2% and 102.8% over 2PL-OS/BI-FC and 2PL-OS, respectively.

Figure 10 shows the performance of ST 2PL-OS/BI with forced commit (ST 2PL-OS/BI-FC) and forced abort (ST 2PL-OS/BI-FA) policies in the baseline case. Similar to 2PL-OS/BI, if the number of terminals is < 70, ST 2PL-OS/BI-FA performs slightly worse than ST 2PL-OS/BI-FC. When the number of terminals is ≥ 70 but ≤ 140, ST 2PL-OS/BI-FA gives larger throughput and fewer miss percentage than ST 2PL-OS/BI-FC. The maximum throughput of ST 2PL-OS/BI-FA is 6.5, a 2.8% increase over ST 2PL-OS/BI-FC. At this peak point, the miss percentage is 20.3% for 2PL-OS/BI-FA, 21.6% for 2PL-OS/BI-FC and 47.7% for 2PL-HP. However, when the number of terminals is > 140, ST 2PL-OS/BI-FA performs worse. Also similar to 2PL-OS, increasing the number of resource units leads to larger margins, and vice versa (Figure 11). In the case of five resource units, ST 2PL-OS/BI-FA gives a maximum throughput of 7.4, a 2.1% improvement on ST 2PL-OS/BI-FC and a 68.8% improvement on 2PL-HP.

We observe that, for a given multiprogramming level, the difference between 2PL-OS/BI-FA and 2PL-OS/BI-FC is larger than that between ST 2PL-OS/BI-FA and ST 2PL-OS/BI-FC. To explain this discrepancy, let us consider the average number of predecessors of a transaction that are ready to commit and reach their own deadlines. Because of the existence of write-write blocking in ST 2PL-OS/BI, this number in 2PL-OS/BI is greater than in ST 2PL-OS/BI. So, if a transaction always aborts itself at commitment, then 2PL-OS/BI-FA avoids aborting more predecessors than ST 2PL-OS/BI-FA. Hence, 2PL-OS/BI-FA performs better than ST 2PL-OS/BI-FA.

Figure 9. slack=3, resource units=3(top)=5(bottom), not tardy policy

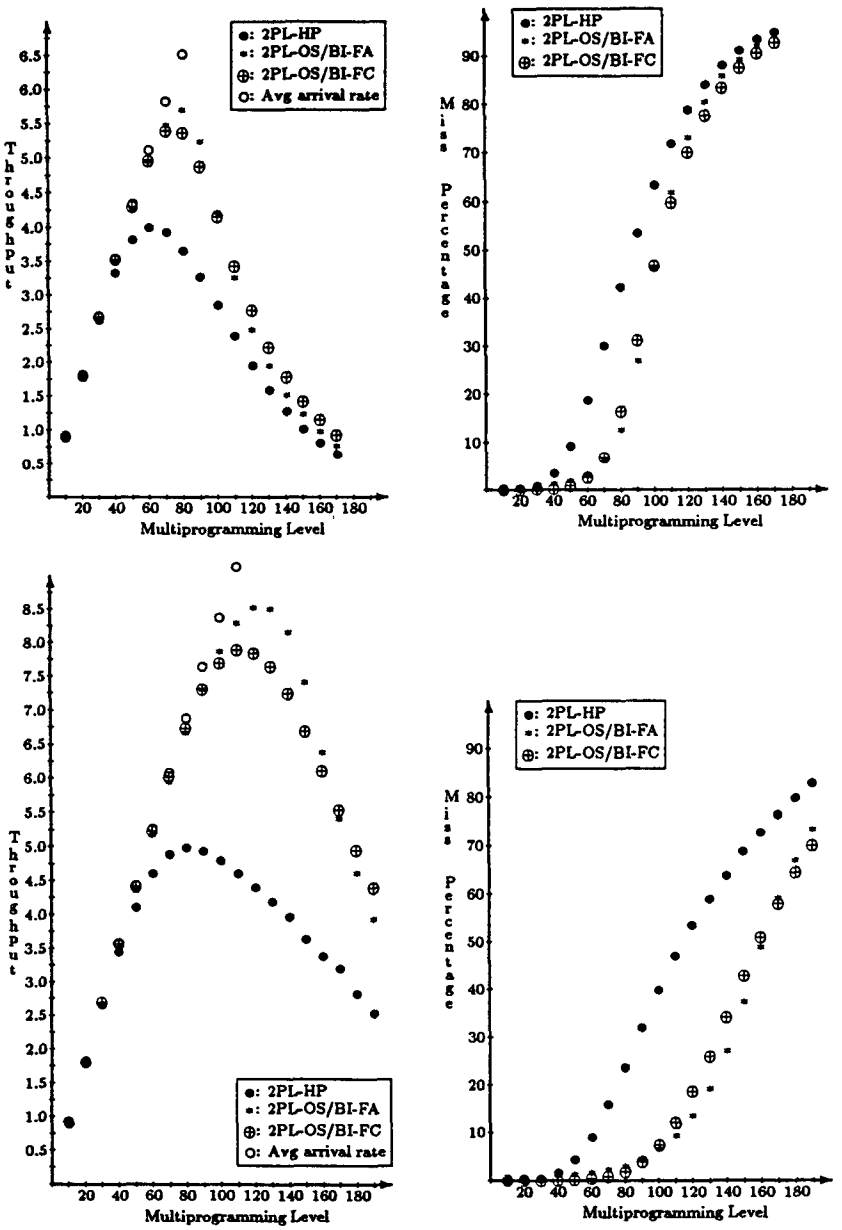
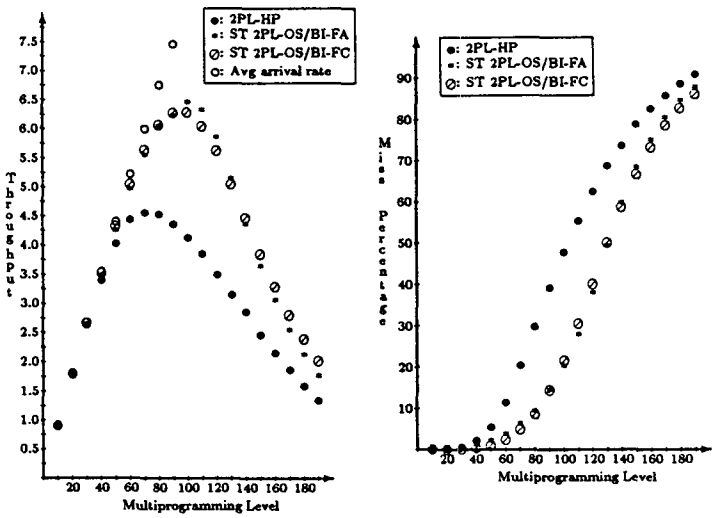


Figure 10. Baseline Case: slack=3, resource units=4, not tardy policy



In ACA 2PL-OS/BI, there is write-read blocking. Recall that in our simulation model, update-txn-pct is 60% and write-op-pct is 50%. So only 30% of operations are write operations and 70% are read operations. This implies that write-read blocking happens more frequently than write-write blocking. So, for transactions reaching their deadlines, the average number of predecessors in ACA 2PL-OS/BI is greater than that in ST 2PL-OS/BI. Hence, in ACA 2PL-OS/BI-FA, the commitment of those transactions that are waiting until their deadlines will abort more predecessors than in ST 2PL-OS/BI-FA. This degrades the performance of ACA 2PL-OS/BI-FA. Our experiments show the same results as this predication. Figure 12 reports the performance of ACA 2PL-OS/BI-FA in comparison of ACA 2PL-OS/BI-FC in baseline case. The maximum throughput of ACA 2PL-OS/BI-FA is 5.5, a 3.3% decrease compared with ACA 2PL-OS/BI-FC. At this peak point, the miss percentage of ACA 2PL-OS/BI-FA is 24.8%, 3.0% more than of ACA 2PL-OS/BI-FC. From Figure 13 we observe that the discrepancy between ACA 2PL-OS/BI-FA and ACA 2PL-OS/BI-FC seems to be smaller if the number of resources decreases from four to three, and this discrepancy is larger if the number of resources increases from four to five. This is similar to the results of 2PL-OS/BI and ST 2PL-OS/BI.

6.6 Effect of Known Requirements

In this experiment, we evaluate the effect of feasible deadlines on the performance of the four protocols. Feasible deadlines or known requirements allow the transaction

Figure 11. slack=3, resource units =3(top) =5(bottom), not tardy policy

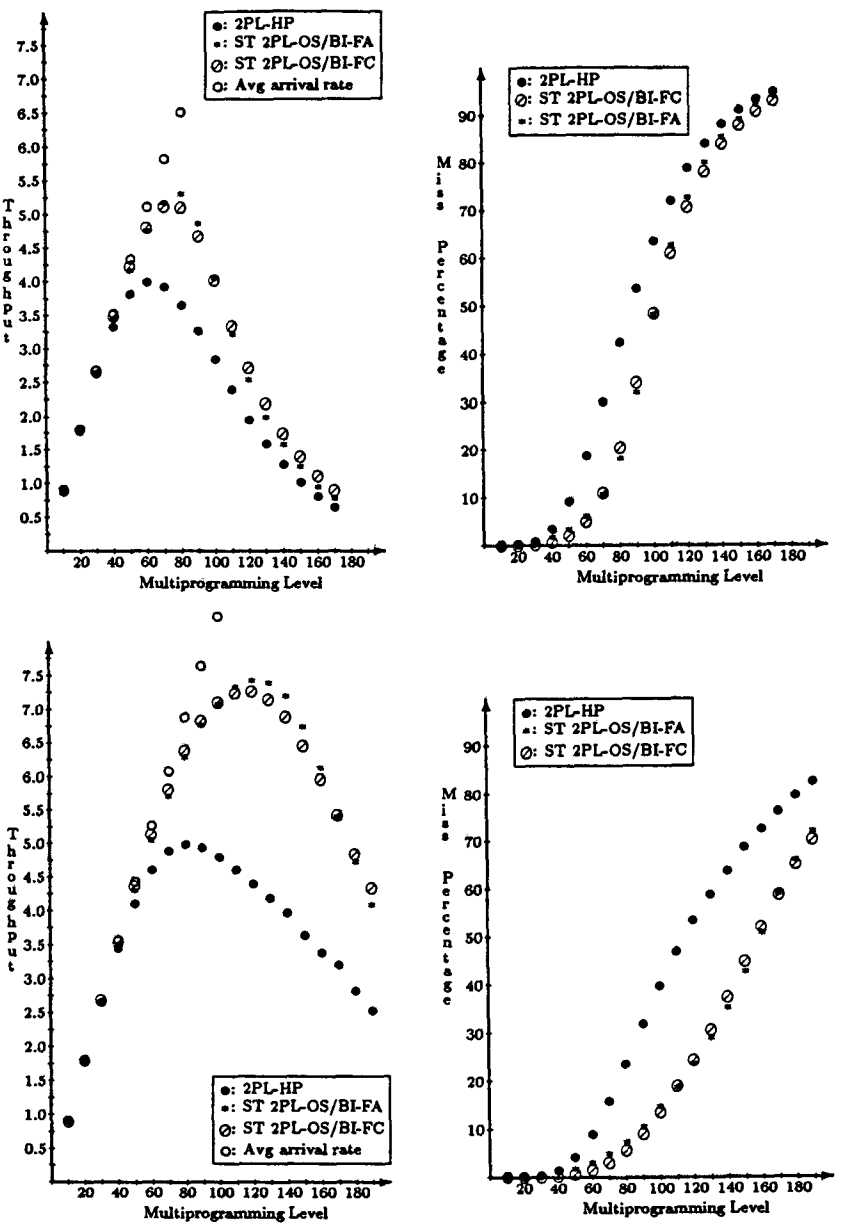
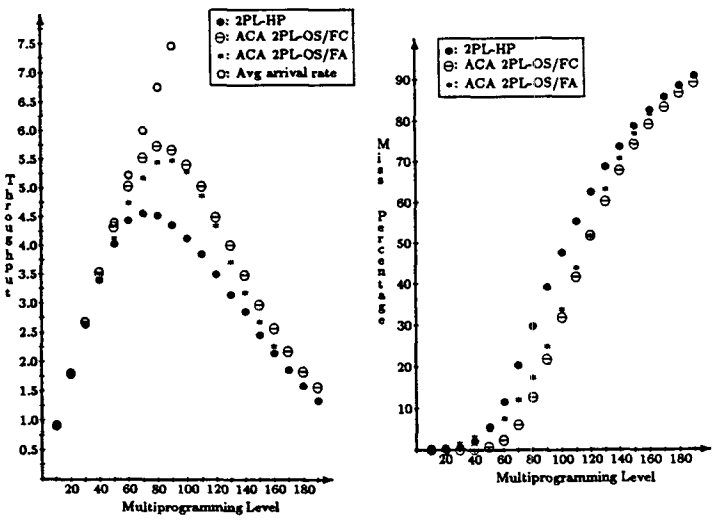


Figure 12. Baseline Case: slack=3,resource units=4, not tardy policy



manager to abort a transaction as soon as they can determine that the time needed to execute the transaction is more than the slack available. Note that this approach is not very practical in general, since the exact requirements for each transaction are difficult to ascertain *a priori.* However, the experiment does provide a better understanding of the locking protocols for real-time databases.

Figure 14 illustrates the throughputs and miss percentages for the four protocols. The parameters chosen for this experiment are identical to those in the baseline experiment, except for the feasible deadlines policy. When compared to the baseline experiment with unknown requirements, the performance of all protocols improves. Since transactions that cannot make their deadlines are aborted early, fewer resources are wasted resulting in improved performance. However, the impact of known requirements is more significant for 2PL-HP, since it benefits from the feasible deadlines policy to reduce the number of wasted restarts. Although 2PL-HP benefits more from known requirements (e.g.,the thrash points of all four protocols are nearly identical), there is still a significant difference between 2PL-HP and the other three protocols. For example, at the thrash point of 110 terminals, the miss percentages are 36% for 2PL-HP, 29% for ACA 2PL-OS, 22% for ST 2PL-OS/BI, and 19% for 2PL-OS/BI.

In a direct comparison of the baseline graphs (see Figure 3) with those for feasible deadlines, we note a change in the shape of the curves. The policy of feasible deadlines seems to flatten out and attenuate the curves. This is due in part to the *half-and-half effect* reported by Carey et al. (1990). Beyond the thrash

Figure 13. slack=3, resource units=4, feasible deadlines policy

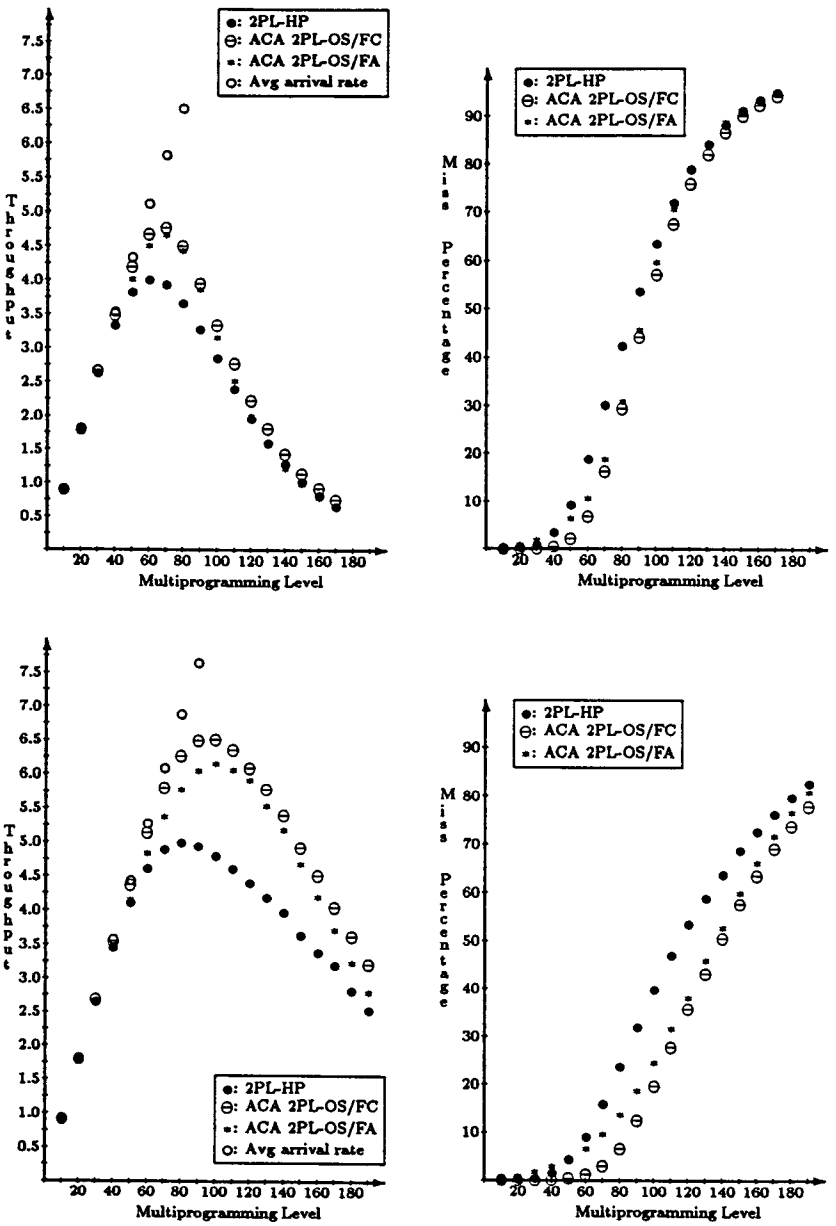
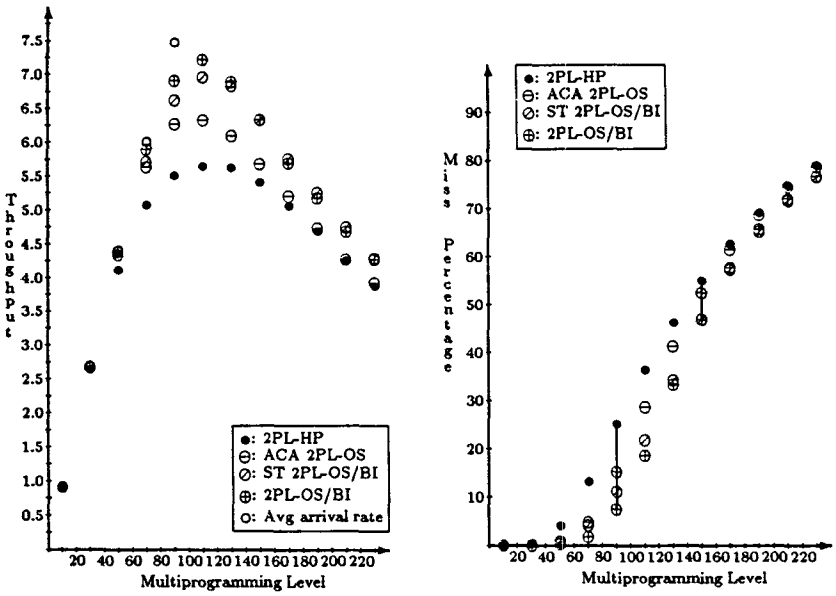


Figure 14. slack=3,resource units=4, feasible deadlines policy



point, aborting transactions actually helps to stabilize throughput by reducing data contention. The feasible deadlines policy accomplishes some of this effect, because it aborts transactions as soon as their deadlines cannot be met. Furthermore, the inter-arrival delay parameter prevents a terminal from introducing a new transaction, immediately resulting in a stablizing effect on the throughput.

7. Comparison of Two-Phase Locking Protocol with Transient Verisons

In this section, we compare two-phase locking protocols with ordered sharing with an instance of the two-phase locking protocol that uses transient or uncommitted versions of data. In particular, we consider the two-version two-phase locking protocol (Stearns and Rosenkrantz, 1981). The reason for this choice is the obvious resemblance of 2PL-OS/BI with 2V2PL and other studies have shown the superiority of 2V2PL over 2PL for real-time databases.

The 2V2PL scheduler uses three types of locks: read locks, write locks, and certify locks. The scheduler sets a read(write) lock before it processes a read(write) operation. When a transaction commits, the 2V2PL must convert all write locks of the transaction into certify locks. Figure 15 illustrates the lock compatibility table used in 2V2PL. By allowing two versions of a data item, the conflict between

Figure 15. Lock Compatibility Matrix for Two-version 2PL

Lock Requester T_j	Lock Holder T_i		
	ReadLock$_i$	WriteLock$_i$	CertifyLock$_i$
ReadLock$_j$	Y	Y	N
WriteLock$-j$	Y	N	N
CertifyLock$_j$	N	N	N

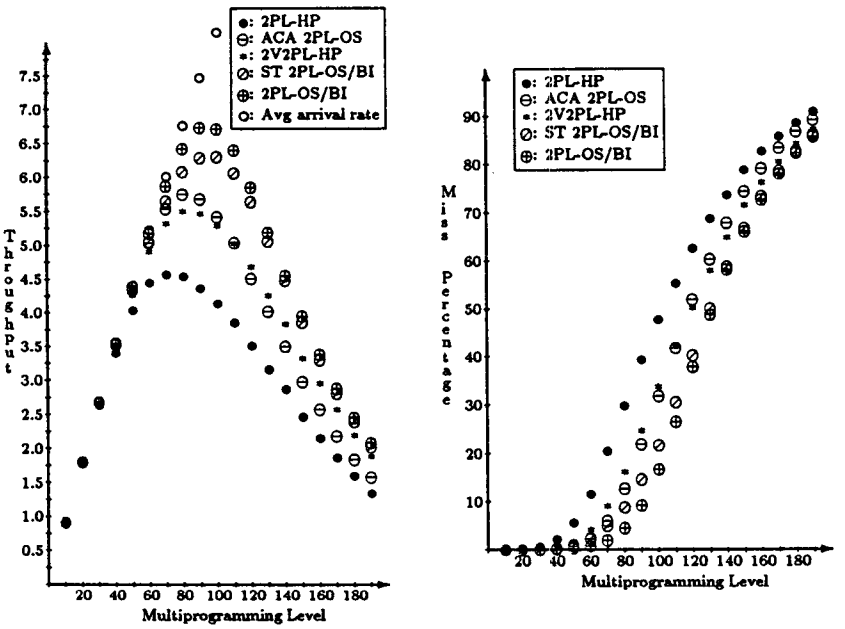
read and write locks is eliminated in 2V2PL. In particular, read locks are set on committed version of the data-object whereas a write lock results in an uncommitted copy of data-object being created and a write lock is being set on that copy. When a transaction commits, it converts all its write locks to certify locks that conflict with other read and write locks. After obtaining the certify locks, the transaction installs the uncommitted version of the data-object as the new committed value.

In real-time databases, 2V2PL needs to be adapted to use the deadlines of transactions. We call this modified protocol 2V2PL-HP and summarize it below:

1. When the scheduler receives a request for a read lock on an object, the request is delayed if the certify lock on this object is currently held by a high priority transaction. Otherwise, the lock is granted by aborting lower priority transactions with the certify locks.
2. When the scheduler receives a request for a write lock on the object, the request is delayed if a write lock or a certify lock on this object is currently held by a high priority transaction. Otherwise the lock is granted by aborting the lower priority transactions with write or certify locks.
3. When the scheduler receives a commit request, it has to convert all write locks held by this transaction into certify locks. A certify lock could be delayed due to read locks held by higher priority transactions. Otherwise, the conversion is permitted by aborting lower priority transactions with read locks. Since transactions in this protocol always read committed data, the 2V2PL-HP histories/execution do not suffer from the problems of cascading aborts.

The comparison of 2V2PL-HP with the proposed protocols in this article are illustrated in Figure 16 (Baseline Case) and Figure 17 (Variable Resources). When the number of terminals is < 100, 2V2PL-HP has worse performance than ACA 2PL-OS and better performance than 2PL-HP and when the number of terminals becomes > 100, 2V2PL-HP performs better than ACA 2PL-OS, but worse than ST 2PL-OS/BI and 2PL-OS/BI. This case also happens consistently in Figure 17, and the switching points are also located at about 100 terminals. This means that the differences between these protocols don't change when the resource units vary. Figure 18 illustrates the missed percentages for five protocols when the slack factor varies.

Figure 16. Baseline case: slack=3, resource units=4 not tardy policy



In the case of 75 terminals, 2V2PL-HP performs worse than ACA 2PL-OS. This is the same as in Figures 16 and 17. In the case of 95 terminals, the margin of differences between ACA 2PL-OS and 2V2PL-HP is small since 95 terminals is near the switching point, which is about 100 terminals.

Comparing the two protocols 2PL-HP and 2V2PL-HP, we note that although certify locks in 2V2PL-HP behave much like write locks in ordinary 2PL-HP, the time to certify a transaction is usually much less than the total time to execute it. Hence 2V2PL-HP's certify locks delay read operations for less time than 2PL-HP's write locks delay read operations. Therefore, the 2V2PL-HP performs better than 2PL-HP. However, since existing read locks delay a transaction's certification in 2V2PL-HP, the improved concurrency of read operations comes at the expense of delaying the certification.

The conflict relationship between the read and write operations in 2V2PL-HP is the same as that in ST 2PL-OS/BI. But, in 2V2PL-HP, a transaction's certification may abort other lower priority transactions due to the read-certify conflict while, in ST 2PL-OS/BI, a transaction delayed for others does not abort any transaction until it reaches its deadline. Thus, in ST 2PL-OS/BI some lower priority transactions for which some higher priority transactions are waiting may have enough time to commit. So, the main reason for the better performance of ST 2PL-OS/BI is that on the average, a commit operation in ST 2PL-OS/BI aborts fewer transactions

Figure 17. slack=3, resource units=3(top),=5(bottom), not tardy policy

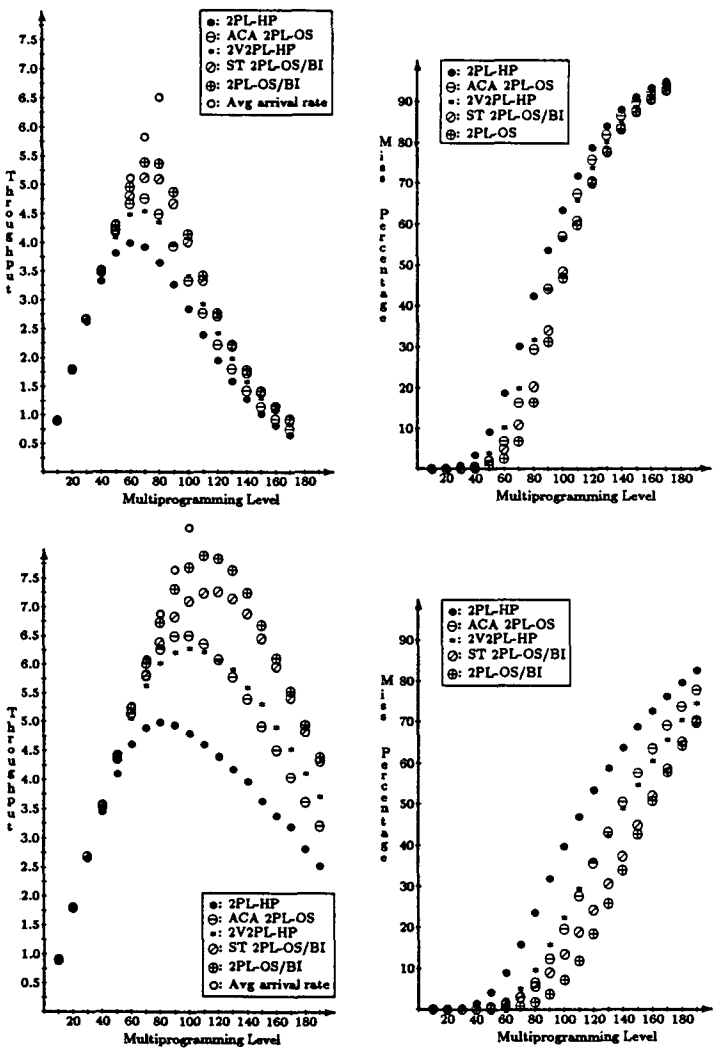
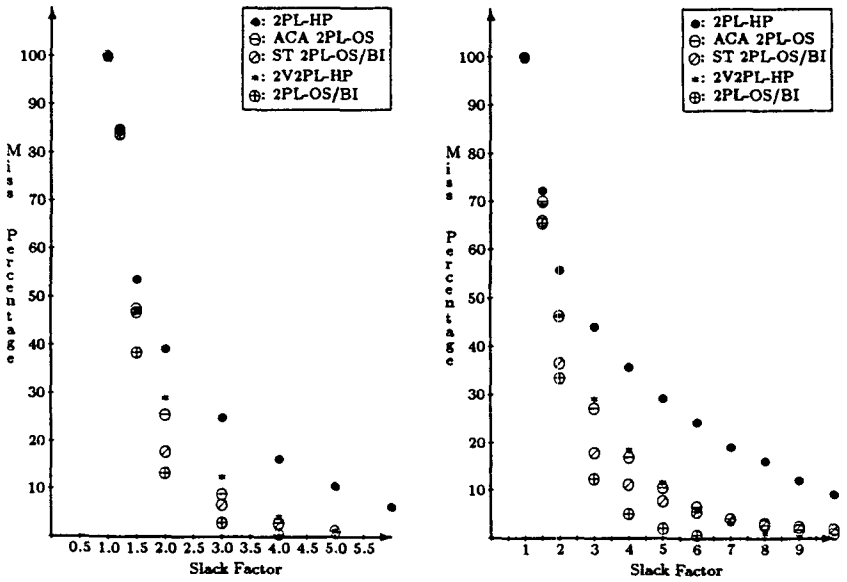


Figure 18. Varying slack factor, 75 terminals(left), 95 terminals(right)



than in 2V2PL-HP.

We note that in 2V2PL-HP a transaction's certification may abort immediately other lower priority transactions with a read lock and, in ACA 2PL-OS, a transaction delayed for other transaction does not abort any transaction until its deadline. This leads to the better performance of ACA 2PL-OS than 2V2PL-HP when the multiprogramming level is low. When the multiprogramming level increases, the performance of these two protocols switches mainly because the write-read delay in ACA 2PL-OS dominates the certify-read delay in 2V2PL-HP.

8. Conclusion

In this article, we describe locking-based protocols for real-time database systems. Our approach is motivated by two main concerns. First, locking protocols are widely accepted and used in most database systems. Second, in real-time databases it has been shown that the blocking behavior of transactions in locking protocols results in performance degradation. We proposed using a new relationship between locks called *ordered sharing* to eliminate blocking. Ordered sharing has the desirable property of eliminating blocking of read and write operations at the expense of a possible delay at transaction commitment. Unlike conventional databases where this delay may degrade response time, in real-time databases we exploit this delay by allowing other transactions to run within the slacks of delayed transactions.

Eliminating all types of blocking through ordered sharing may, however, result in cascading aborts. We overcome this problem by using before-images, which are generally maintained for recovery purposes. We compared the performance of the proposed protocols with 2PL-HP, the two-phase locking protocol for real-time databases. Our performance results clearly establish the superiority of the proposed protocols over 2PL-HP. In general, 2PL-OS/BI has the best performance followed by ST 2PL-OS/BI and ACA 2PL-OS and, finally, 2PL-HP. In the region of low data contention or low workload, all four protocols exhibit comparable performance. However, at higher load or higher data contention, 2PL-OS/BI misses significantly fewer deadlines than 2PL-HP. Once again, at very high data contention all protocols miss almost all deadlines. Our resource related experiments indicate that protocols with ordered sharing benefit significantly from an abundance of resources. Finally, even with known requirements (feasible deadlines), the proposed protocols still provide better performance than 2PL-HP. Note that, in the usual circumstance of unknown requirements, the difference in performance is more striking. We also compared the multiversion locking protocol for real time databases and showed that, in general, the proposed protocols result in improved performance.

Acknowledgments

This research is supported by the NSF under grant number IRI-917904.

References

Abbott, R. and Garcia-Molina, H. Scheduling real-time transactions: A performance evaluation. *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, Los Angeles, CA, 1988.

Abbott, R. and Garcia-Molina, H. Scheduling real-time transactions with disk resident data. *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam, The Netherlands, 1989.

Abbott, R. and Garcia-Molina, H. Scheduling I/O requests with deadlines: A performance evaluation. *Proceedings of the Eleventh IEEE Real-Time Systems Symposium*, Lake Buena Vista, FL, 1990.

Agrawal, D. and El Abbadi, A. Locks with constrained sharing. *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, Nashville, TN, 1990. To appear in the *Journal of Computer and System Sciences*.

Agrawal, D. and El Abbadi, A. Ordered sharing: A new lock primitive for database systems. Technical Report TRCS 91-18, Department of Computer Science, University of California, Santa Barbara, CA 93106, 1991.

Agrawal, D., El Abbadi, A., and Jeffers, R. An approach to eliminate transaction blocking in locking protocols. *Proceedings of the ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, San Diego, CA, 1992.

Agrawal, D., El Abbadi, A., and Lang, A.E. The performance of protocols based on locks with ordered sharing. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):805-818, 1994.

Agrawal, R., Carey, M.J., and Livny, M. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609–654, 1987*a*.

Agrawal, R., Carey, M.J., and McVoy, L. The performance of alternative strategies for dealing with deadlocks in database management systems. *IEEE Transactions on Software Engineering*, 13(2):1348–1363, 1987*b*.

Alonso, G., Agrawal, D., and El Abbadi, A. Reducing recovery constraints on locking protocols. *Proceedings of the 1994 ACM Symposium on Principles of Database Systems*, Minneapolis, MN, 1994.

Bayer, R., Heller, H., and Reiser, A. Parallelism and recovery in database systems. *ACM Transactions on Database Systems*, 5(2):139–156, 1980.

Bernstein, P.A. and Goodman, N. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

Bernstein, P.A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems*. Reading, MA: Addison Wesley, 1987.

Bernstein, P.A., Shipman, D.W., and Wong, W.S. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, 5(5):203–216, 1979.

Buchmann, A.P., McCarthy, D.R., Hsu, M., and Dayal, U. Time-critical database scheduling: A framework for integrating real-time scheduling and concurrency control. *Proceedings of the Fifth IEEE International Conference on Data Engineering*, Los Angeles, CA, 1989.

Carey, M.J. Modeling and evaluation of database concurrency control algorithms. Ph.D. thesis, Electronics Research Library, College of Engineering, University of California, Berkeley, September 1983.

Carey, M.J., Krishnamurthi, S., and Livny, M. Load control for locking: The half-and-half approach. *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, Nashville, TN, 1990.

Chan, A., Fox, S., Lin, W.K., Nori, A., and Ries, D.R. The implementation of an integrated concurrency control and recovery scheme. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Orlando, FL, 1982.

Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

Gray, J., McJones, P., Blasgen, M., Lindsay, N., Lorie, R., Price, T., Putzolu, F., and Traiger, I. The recovery manager of the system R database manager. *ACM Computing Surveys*, 13(2), June 1981.

Hadzilacos, V. A theory of reliability in database systems. *Journal of the ACM*, 35(1):121–145, 1988.

Härder, T. and Reuter, A. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.

Haritsa, J., Carey, M.J., and Livny, M. Dynamic real-time optimistic concurrency control. *Proceedings of the Eleventh IEEE Real-Time Systems Symposium*, Lake Buena Vista, FL, 1990*a*.

Haritsa, J., Carey, M.J., and Livny, M. On being optimistic about real-time constraints. *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, Nashville, TN, 1990*b*.

Huang, J., Stankovic, J.A., Ramamritham, K., and Towsley, D. Experimental evaluation of real-time optimistic concurrency control schemes. *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, Barcelona, Spain, 1991.

Huang, J., Stankovic, J.A., Towsley, D., and Ramamritham, K. Experimental evaluation of real-time transaction processing. *Proceedings of the Tenth IEEE Real-Time Systems Symposium*, Santa Monica, CA, 1989.

Kim, W. and Srivastava, J. Enhancing real-time DBMS performance with multiversion data and priority based disk scheduling. *Proceedings of the Twelfth IEEE Real-Time Systems Symposium*, Location?, 1991.

Kung, H.T. and Robinson, J.T. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

Law, A. and Larmey, C. *An Introduction to Simulation Using SIMSCRIPT II.5*. Los Angeles: CACI, Inc., 1984.

Lin, Y. and Son, S.H. Concurrency control in real-time databases by dynamic adjustment of serialization order. *Proceedings of the Eleventh IEEE Real-Time Systems Symposium*, Lake Buena Vista, FL, 1990.

Litwin, W. and Shan, M.C. Value dates for concurrency control and transaction management in interoperable systems. *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, Kyoto, Japan, 1991.

Menasce, D. and Nakanishi, T. Optimistic versus pessimistic concurrency control mechanisms in database management systems. *Information Systems*, 7(1), 1982.

Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

Papadimitriou, C.H. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

Reed, D.P. Naming and synchronization in a decentralized computer system. Technical Report MIT-LCS-TR-205, Massachusetts Institute of Technology, Cambridge, Massachusetts, September, 1978.

Russell, E.C. *Building Simulation Models with SIMSCRIPT II.5*. Los Angeles: CACI Inc., 1983.

Russell, E.C. *SIMSCRIPT II.5 Programming Language*. Los Angeles: CACI Inc., 1987.

Sha, L., Rajkumar, R., and Lehoczky, P. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

Sha, L., Rajkumar, R., Son, S.H., and Chang, C. A real-time locking protocol. *IEEE Transactions on Computers*, 40(7):793–799, 1991.

Stearns, R.E. and Rosenkrantz, D.J. Distributed database concurrency control using before-values. *Proceedings of the ACM SIGMOD Conference on the Management of Data*, New York, NY, 1981.

Verhofstad, J.S.M. Recovery techniques for database systems. *ACM Computing Surveys*, 10(2):167–196, 1978.