On Real-Time Databases: Concurrency Control and Scheduling

Philip S. Yu and Kun-Lung Wu
IBM Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Kwei-Jay Lin
Dept. of Computer Science
University of Illinois
Urbana, IL 61801

Sang H. Son
Dept. of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract

In addition to maintaining database consistency as in conventional databases, real-time database systems must also handle transactions with timing constraints. While transaction response time and throughput are usually used to measure a conventional database system, the percentage of transactions satisfying the deadlines or a time-critical value function is often used to evaluate a real-time database system. Scheduling real-time transactions is far more complex than traditional real-time scheduling in the sense that (1) worst-case execution times are typically hard to estimate, since not only CPU but also I/O requirement is involved; and (2) certain aspects of concurrency control may not integrate well with real-time scheduling. In this paper, we first develop a taxonomy of the underlying design space of concurrency control including the various techniques for achieving serializability and improving performance. This taxonomy provides us with a foundation for addressing the real-time issues. We then consider the integration of concurrency control with real-time requirements. The implications of using run policies to better utilize real-time scheduling in a database environment are examined. Finally, as timing constraints may be more important than data consistency in certain hard real-time database applications, we also discuss several approaches that explore the non-serializable semantics of real-time transactions to meet the hard deadlines.

Index terms: concurrency control, real-time databases, real-time scheduling, real-time transactions, serializability, schedulability.
### Contents

1 Introduction ................................................. 1

2 Concurrency control ........................................ 3
   2.1 Conflict detection .................................... 3
   2.2 Conflict resolution .................................. 4
      2.2.1 2PL and variations ............................. 4
      2.2.2 OCC and variations ............................. 5
      2.2.3 Locking with deferred blocking ................ 5
      2.2.4 Multiversioning ................................ 6
   2.3 Serialization rule and order ......................... 7
      2.3.1 Without readjusting serialization order ...... 8
      2.3.2 Dynamically readjusting serialization order .. 9
   2.4 Run policy ........................................... 10

3 Integrating concurrency control with real-time requirements 11
   3.1 Scheduling real-time transactions ................... 12
      3.1.1 Priority inversion ............................... 12
      3.1.2 Scheduling priority ............................. 12
      3.1.3 Overload management ............................ 13
      3.1.4 I/O scheduling .................................. 14
   3.2 Adapting CCs to real-time environments ............. 15
      3.2.1 Serialization order ............................. 15
      3.2.2 Conflict detection and resolution .............. 16
   3.3 Real-time CCs in the literature ....................... 16
      3.3.1 2PL and variations ............................. 16
      3.3.2 OCC ............................................ 19
      3.3.3 Multiversion schemes ............................ 19
   3.4 Extending other CCs to real-time environments ....... 19
   3.5 Further remarks on conflict resolution .............. 20
   3.6 Run policy ........................................... 22

4 Special requirements of hard real-time applications and their implications 23
   4.1 Hard real-time requirements ......................... 23
   4.2 Implications ......................................... 24

5 Relaxing serializability in real-time transactions .......... 25
   5.1 External and temporal consistencies ................ 25
   5.2 Guaranteeing consistencies by assigning periods ..... 27
   5.3 Trading serializability for external consistency ... 28

6 Conclusions ................................................. 29
1 Introduction

Real-time databases have received growing attention in recent years [2, 3, 12]. Several real-time database applications can be found in program trading in the stock market, computer integrated manufacturing, military command and battle management, and network management systems. Besides maintaining database consistency, real-time database systems must also handle transactions with timing constraints. For example, in a hard real-time database system, transactions must be scheduled in such a way that they can be completed before their deadlines. On the other hand, in a soft real-time database system, timeliness is important but late results are still valuable, though the value reduces as the tardiness increases.

Various scheduling algorithms have been developed to schedule real-time tasks to meet their timing constraints, e.g., see [41, 42, 45, 53, 75], given the arrival time, deadline, worst-case execution time, and criticality of each task. These algorithms typically consider either one type of resource, such as the CPU, or simultaneous usage of multiple types of resources. However, most of them are not directly applicable to schedule real-time transactions, since transaction executions in general involve the CPU and I/O in a serial fashion and the worst-case execution time cannot be meaningfully estimated. For instance, the I/O rate (i.e., the buffer miss rate) in a database system is hard to predict, and is affected by not only the buffer size and the workload mix but also the concurrency control (CC) scheme [15]. Since the I/O time generally is at least an order of magnitude larger than the CPU time, a worst-case estimate assuming a zero buffer hit probability is not meaningful. It may provide such an overestimate of the total execution time that the scheduler can be misled to abandon many transactions prematurely. In addition, traditional real-time scheduling usually do not address the data consistency issue, whereas consistency has to be maintained by the CC in database systems. Therefore, the difficulties in designing scheduling algorithms to meet transaction deadlines come not only from the scheduling of multiple hardware resources in a serial manner, but also from the scheduling of data resources by combining CC with timing constraints [12].

On the other hand, many variations of CC schemes have also been introduced to improve concurrency and system performance (e.g., see [4, 9, 11, 19, 72, 74]) in conventional database environments. Recently, there have been several studies dealing with CC in real-time databases addressing the transaction scheduling aspect. These real-time CC schemes are mainly extended or adapted from existing CC schemes to the real-time environments, such as two phase locking and optimistic CC schemes, and multiversion schemes [9]. Some of the CC schemes are more naturally adaptable to real-time requirements, while others are less so. For example, in optimistic concurrency control (OCC) schemes, the CC decision tends to be more or less independent of the CPU scheduling [23, 24, 27]. However, in the two-phase locking (2PL) approach, the CC decision
can nullify the CPU scheduling priority, as a low priority transaction may hold a lock requested by a high priority transaction, thus forcing the high priority transaction to wait for the lower priority one [57]. Several techniques have been developed to add on to the conventional CC schemes to cope with the real-time requirements.

In addition to the real-time requirements, other major differences also exist between conventional and real-time database systems. For example, while transaction response time and throughput are usually the performance metrics to measure conventional database systems, the percentage of transactions satisfying the real-time constraints tends to be the measure for real-time databases. In traditional databases, when a data-access conflict occurs, the preference tends to be based either on fairness (e.g., given to the transaction that first grabs the data resource) or on resource consumption (e.g., given to the transaction which has made more progress toward its completion). In real-time databases, however, the preference tends to be based on criticality and given to the transaction that is closer to its deadline. However, this can cause two problems. First, more CPU resource is wasted if nearly completed transactions are aborted in favor of the more time critical transactions. Second, longer transactions may be harder to get through, thus creating a starvation problem.

In contrast to the two major, but separate, research efforts on real-time scheduling and conventional database CC, studies on real-time databases are still in the early stages. The goal of this paper is to bridge the gap between these efforts by developing a taxonomy of the underlying design space of CC, encompassing the various techniques for achieving serializability and improving performance. This taxonomy of the CC design space will provide us a basis to incorporate the different approaches introduced in the real-time CC literature into the same framework and to achieve a better understanding of the issues and opportunities of designing real-time-oriented CC. In addition, we examine the integration of CC with real-time scheduling, and explore the concept of run policy to utilize real-time scheduling in the database environment. Finally, as satisfying timing constraints may be more important than maintaining data consistency in some hard real-time applications, even the real-time-oriented CC may not be sufficient to guarantee that the hard deadlines will be met. Thus, we also discuss several approaches that may relax database consistency in order to meet the hard deadlines.

The paper is organized as follows. In section 2, we develop a taxonomy of the underlying design space of concurrency control. We then examine the integration of conventional CC with real-time scheduling and the implications of run policies in Section 3. In Section 4, we discuss the special requirements of some hard real-time database applications and their implications. Several approaches that may trade off data consistency for meeting the hard deadlines are presented in Section 5.
2 Concurrency control

CC schemes have the responsibility to ensure that although transactions are executed concurrently with interleaving operations, the committed or certified transactions can be ordered or given a certification time stamp ordering so that the net effect on the database is equivalent to the execution of these transactions in a serialized order one at a time. Note that after a transaction is committed, its effect on the database becomes permanent. The CC design space can be classified along several dimensions: conflict detection, conflict resolution, serialization rule and order, and run policy. Such a classification provides us with a nice framework from which various techniques for achieving serializability and improving performance can be better illustrated.

2.1 Conflict detection

There are two ways to detect a conflict: either before the granule access or after the granule access. The former is referred to as the pessimistic approach. The latter is referred to as the optimistic approach, where checking for serializability is done later at the certification time.

Several mechanisms can be used to facilitate the detection process, such as locks, time stamps and serialization graphs (SGs) [9]. For each of these mechanisms, there is one type of pessimistic CC scheme using it exclusively for conflict detection. For example, two-phase locking (2PL) schemes use locks, time stamp ordering (TSO) schemes use time stamps, and serialization graph testing (SGT) schemes use serialization graphs. For OCC, however, any one of the three mechanisms can be used to detect conflicts at the certification time.

For a lock-based scheme, a lock must first be obtained before an access is made to a granule. However, different access modes and compatibility matrices can be used to allow for more concurrency. For example, read (shared) and write (exclusive) modes are often introduced to distinguish between read and write accesses in a 2PL scheme. A read lock is only compatible with other read locks; a write lock is incompatible with any other lock. When locks are used by an OCC scheme, an additional type of lock mode, referred to as a weak lock, can be introduced as well [71, 72]. A weak lock, which is in contrast to the normal (strong) lock, is only used to indicate that a granule is being accessed. It is compatible with other weak locks and the strong locks of shared mode, but is incompatible with the strong locks of exclusive mode. In addition, an extension can also be made in 2PL to allow each transaction to use a different granule size that is most appropriate for its execution. This is referred to as multi-granularity locking (MGL) [21].

For a TSO scheme, each transaction is assigned a time stamp before execution. For each granule access, the transaction’s time stamp is checked against the time stamp of the last transaction that has accessed the same granule to determine whether the predefined time stamp order can be maintained.

For an SGT scheme, the CC manager maintains an SG that represents the execution ordering of
all the transactions in the history, and checks for cycles. In general, information about a transaction cannot be deleted at commit time. It can be deleted only after the committed transaction will not, at any time in the future, be involved in a cycle of the SG.

2.2 Conflict resolution

Once a conflict is detected, a conflict resolution mechanism needs to be put in place. A conflict resolution mechanism needs to decide which candidate transaction(s) (the lock requester or lock holders) to penalize, and choose an appropriate action and a suitable timing for the action. There are two possible actions that are most frequently used: blocking (or wait) and abort (or restart). (Two other alternatives are multiversioning (Section 2.2.4), and dynamically readjusting the serialization order (Section 2.3.2), which will be discussed later.) If a conflict is detected before the granule access, either blocking or abort can be used to resolve the conflict. However, if a conflict is detected after the granule access, only abort is appropriate. An alternative to blocking for conflict resolution is delaying commit. That is to say instead of waiting for a lock, the transaction is waiting for the commit order. This can be implemented in a multiversion scheme and other locking schemes like the ordered-sharing schemes in [6] to be further discussed later. As to a suitable timing for an action, it is immediate for blocking, but it can be either immediate or delayed, e.g., delayed until the certification time, for abort.

2.2.1 2PL and variations

For any 2PL scheme, conflict detection is through obtaining a lock before each granule access. In 2PL, if there is a lock conflict, the requesting transaction is blocked and put into a wait state. One simple variant of 2PL is 2PL with no waiting, which aborts the requesting transaction upon conflict [6-4]. Other variants also exist. For example, in [55], two variants of 2PL were considered: wound-wait and wait-die. Each transaction is given a number or priority based on its arrival time. In essence, an older transaction has a higher priority and receives a more favorable treatment in conflict resolution. In the wait-die scheme, if the requester is older, then the requester waits, otherwise the requester is aborted. In the wound-wait scheme, if the lock requester is older, then the current lock holder is aborted, otherwise the requester waits.

However, a 2PL scheme may suffer from a cascade of blocking, where waiting transactions continue to hold locks and thus block other transactions. There are several variants of 2PL designed to avoid cascading blocking, including the running priority scheme [17] and the wait depth limited (WDL) scheme [19]. Both schemes try to avoid building up a long waiting chain, but they differ in the candidate transactions selected to penalize upon conflict. The running priority scheme gives a higher priority to the running transaction in conflict resolution. When a lock conflict occurs and the holder is in the wait state, the holder is aborted. Furthermore, to maintain a wait-depth
of one, if the holder is running, the requester will be put into the wait state only if there is no other transactions waiting for the lock. The WDL scheme with depth $d$, referred to as WDL($d$), aborts transactions upon conflict to maintain a wait chain of depth no longer than $d$. Note that a penalized transaction in the WDL scheme need not be aborted immediately, and an optimization can be achieved by a delayed abort concept [68]. The penalized transaction can be put into an abort-waiting state, and is aborted only if the conflicting transaction is successfully committed.

### 2.2.2 OCC and variations

For all OCC schemes, a conflict is always detected after the granule access. We discuss two variations, pure OCC and broadcast OCC, which differ in the timing of conflict resolution. Other OCC variants that differ in the serialization order will be discussed later in Section 2.3.2. Here we assume a weak lock approach is used to detect conflicts [72]. For each granule access, a weak lock can be requested asynchronously.

First consider the pure OCC scheme where conflict detection and resolution are both done at the transaction certification time. When a transaction completes its execution, it requests the CC manager to certify all its accessed granules. At the certification time, if the certifying transaction has not yet been marked for abort the CC manager will convert all of its weak locks of exclusive mode into strong locks, and mark any conflicting transactions for abort. (Otherwise, the certification request will be denied, and the certifying transaction is aborted.) The strong locks will be held during the commit interval, and any weak lock request against a strong lock held in the exclusive mode will cause the requesting transaction to be marked for abort. (If the weak lock is obtained synchronously, however, a weak lock request against a strong lock can be put into a wait state to avoid abort at the expense of a short wait for commit processing [70].)

A variation of the pure OCC scheme, referred to as the broadcast OCC scheme, is to abort or restart a transaction immediately after it is marked for abort [34]. Unlike the pure OCC scheme where the abort action is delayed until the certification time, the broadcast OCC scheme aborts immediately. Note that the delayed abort in the pure OCC case is different from that in the locking case discussed in Section 2.2.1 in the sense that the delay can never save the abort in the pure OCC while it may in the case of locking. The advantage of delayed aborts in the pure OCC is on fewer IO's during reruns (see more explanations on Section 2.4, thus lowering the abort probability for a rerun transaction, as studied in [72, 73] where it was shown that the savings in I/O's on rerun transactions in the pure OCC can outweigh the savings of immediate aborts in the broadcast OCC.

### 2.2.3 Locking with deferred blocking

As mentioned in Section 2.2.1, a conventional 2PL scheme tends to suffer from a cascade of blocking. On the other hand, an OCC scheme may suffer from wasting resources due to transaction aborts
and restarts. In OCC, transactions become more vulnerable (i.e., more likely to be involved in conflicts and be marked for abort) as they make more progress toward completion, since more granules are accessed. As a result, longer transactions would incur higher abort probabilities in OCC [73], thus creating a fairness issue. Furthermore, the cost of aborting a nearly completed transaction is certainly higher than that of aborting a newly started transaction.

One approach to addressing this issue is to combine OCC with 2PL using the concept of deferred blocking [70]. This is referred to as the locking with deferred blocking (LDB) scheme. The transaction execution is divided into a non-blocking phase and a blocking phase. At the start of the execution, a transaction is in the non-blocking phase where it simply obtains a weak lock synchronously for each granule access. That is to say in this phase, transactions may wait for strong locks but do not block other transactions. After a transaction has access a predefined number of granules, it tries to enter the blocking phase. It will try to convert all the weak locks on the already accessed granules into strong locks as in the certification process of a conventional OCC. If successful, the transaction will switch to the blocking phase. It will then obtain a strong lock on each subsequent granule access and wait for the lock, if held under an incompatible mode. This will prevent from aborting a transaction at the later stage of its life except in the case of a deadlock, and will also reduce the average holding time of a strong lock, thus lessening the blocking effect.

2.2.4 Multiversioning

Maintaining multiple versions of a granule is another technique that can be used to resolve conflicts. Multiversion CC can be based upon any of the three basic CC schemes: 2PL [8], TSO [54] and SGT [9]. Under multiversion TSO, a new version can be created for each update, if it has not been read by any transaction with a later time stamp. Otherwise, the conflict cannot be resolved by introducing a new version and the updating transaction will be aborted. For each read, the latest version created before the transaction’s time stamp is returned. Delaying commit is needed to make sure that all the transactions that created the versions read by the certifying transaction must commit first.

Multiversion 2PL can have a two-version (two-copy) and multiple-version (multiple-copy) variants. Under conventional 2PL (i.e., the single version 2PL), a write lock prohibits a read lock on the same granule. To resolve this type of write-read or read-write conflict, a two-version 2PL can be devised. When a write lock request is granted, a new version is created while the old version is retained. Subsequent read lock requests will be granted to the old version. However, a write-write conflict still results in blocking as in conventional 2PL. Delaying commit on the certifying transaction is needed to ensure that any transaction that reads the older version of a granule updated by the certifying transaction must commit first. Multiversion 2PL with more than two copies can be introduced to relax the write-write type of conflict. Note, however, when uncertified versions are
allowed to be read by transactions, cascading aborts can occur.

Multiversioning is particularly effective in supporting concurrent processing of short update transactions and long-running queries (read-only transactions). Multiversion schemes based on 2PL that maintain at most two versions for each granule may increase the level of concurrency by reducing, but not eliminating, data contention. The increase in the level of concurrency is limited because only one single old version is maintained, and queries may still delay transactions. One simple way to completely eliminate data contention is by maintaining an unrestricted number of versions for each granule, and serializing a query before all the concurrently active update transactions when the query arrives. However, storage overhead and version management complexity may make the unrestricted multiversion approach unattractive. One possible solution is to maintain a small number of consistent database snapshots and force queries to read from one of the snapshots. Versions created between snapshots can be discarded, thus only a fixed number of versions are maintained for each granule. Various implementations which employ a fixed number of versions have been independently proposed, such as [10, 16, 43, 44, 66]. These fixed-version schemes differ primarily in the version/locking granularity and the management of snapshots. A detailed comparison and contrast among them can be found in [66]. In [66], a general class of such fixed-version schemes, referred to as dynamic finite versioning (DFV) schemes, were described. In DFV, a small number of consistent logical snapshots are dynamically derived, not physically copied, to provide timely access for queries. Intermediate versions created between snapshots are automatically discarded, thus reducing storage overhead. Dirty pages in DFV can be written back to the database before commit (called the STEAL policy in [22]), and at the same time, consistent logical snapshots can be advanced automatically without quiescing any of the ongoing transactions and queries.

2.3 Serialization rule and order

Each CC scheme enforces a specific serialization order. For example, the serialization order may be based on the start execution time, the completion time (i.e., the certification time), or a dynamically derived order, such as the granule access time. Based on this order, each CC scheme uses certain rules to decide whether or not a certain sequence of execution interleavings among transactions will satisfy the serialization order or requirement. However, serialization order can also be dynamically readjusted to resolve conflicts and reduce the number of transactions needed to be aborted or put into a wait state. We first consider in Section 2.3.1 several schemes without readjusting the serialization order, and then examine in Section 2.3.2 different schemes that enlarge the eligible subset of serializable transactions by readjusting the serialization order.
2.3.1 Without readjusting serialization order

Start time or prespecified time-stamp order

A TSO scheme assigns a time stamp to each transaction before it is executed. This time stamp usually is the start time of the transaction. If, during the course of execution, a transaction cannot be certified based on that time stamp ordering, it will be aborted.

Completion time

The pure OCC scheme only checks to see whether a transaction can be certified at the end of its execution. Thus, the certification based on a weak lock implementation effectively tries to certify transactions based on the completion order. Even though the broadcast OCC scheme tries to immediately abort transactions that are conflicting with the newly certified transaction, the serialization order is still based on completion times. However, the certification can also be based on time stamps, and the serialization order would be based on the time stamp order. As transactions may not complete in an order according to their start times, a serialization order based on start times may lead to more aborts than that based on completion times.

Granule access order

A 2PL scheme tries to preserve a serialization order based on the granule access order. (An SGT scheme is another example.) Here we assume no ordering among the compatible lock requests. If transaction X accesses a granule after transaction Y with an incompatible lock request, its serialization order must come after transaction Y. Even when the lock modes are compatible, if there are already intervening incompatible lock requests by other transactions, the serialization order of transaction X must also come after transaction Y. However, if all the locks held are released at the commit time, the serialization order of a 2PL scheme will also be the same as that of the completion times of the transactions.

Granule access order with two-level serialization

An extension to the standard locking approach to increase concurrency is to introduce the concept of two-level serialization. We distinguish between the serializations at the transaction level on commit order and the granule access level. Separate mechanisms can be introduced to manage each of the two levels. Under the two-level serialization concept, the serialization of a certain type of conflicting granule accesses only needs to be constrained at the transaction level. (This is in contrast to 2PL in that the blocking on a conflicting granule access continues through the entire period that the lock is held.) Certainly, serialization at the individual granule access level does not imply serialization at the transaction level, which can involve multiple granule accesses. A
delaying-commit mechanism is needed to ensure that a transaction is committed only after all the transactions that have a conflicting granule access prior to that transaction terminate (commit or abort). Cascading aborts may occur in this situation, however.

In [4, 6], an ordered-sharing locking scheme was presented, which can be classified into this category. In addition to the conventional shared and exclusive modes, an ordered shared mode and a more flexible compatibility matrix were introduced. An update-in-place policy was used to execute granule (read and write) operations [22]. With the ordered-shared mode, transactions are allowed to hold locks in the ordered-shared mode of previously incompatible lock modes on the same granule. However, if transactions obtain locks in the ordered-shared mode on the same granule, the corresponding operations to that granule must be carried out in the same order as that in which lock requests are granted [6]. Furthermore, a transaction may not release any lock as long as it is waiting for other transactions to complete [6]. That is to say that a transaction commit is delayed until all preceding transactions terminate.

2.3.2 Dynamically readjusting serialization order

Dynamic time stamp allocation [7], and time stamp interval allocation [7, 11, 47, 74] schemes have been proposed to have the certification time stamps dynamically derived and re-ordered either at granule access time or at the certification time. These schemes are designed to address the read-write conflict issue for transactions with a mixed read and write behavior.

Consider the following example. Suppose that transaction X reads data items \(D_1, D_2\) and \(D_3\) and updates \(D_3\), and transaction Y reads data items \(D_3\) and \(D_4\) and updates \(D_5\), i.e.,

\[
T_X : R_X(D_1), R_X(D_2), R_X(D_3), W_X(D_3), C_X;
\]

\[
T_Y : R_Y(D_3), R_Y(D_4), W_Y(D_5), C_Y;
\]

where \(R(.)\) and \(W(.)\) represent the read and update operations, respectively, and \(C\) indicates the commit. Also, suppose that transaction X reads \(D_1, D_2\) and \(D_3\), then transaction Y reads \(D_3\) and \(D_4\), then transaction X updates \(D_3\) and commits (successfully), following which transaction Y attempts to update \(D_5\) and commit, i.e.,

\[
R_X(D_1), R_X(D_2), R_X(D_3), R_Y(D_3), R_Y(D_4), W_X(D_3), C_X, W_Y(D_5), C_Y.
\]

An OCC method would abort transaction Y since it is deemed to have read a “wrong” version of \(D_3\), though transaction Y could actually be certified since its effect is equivalent to the serial log of transaction Y committing first and transaction X next. That is to say transaction Y cannot be certified with a time stamp larger than that of transaction X but can be certified with a time stamp smaller than that of X. The serialization order based on the completion times is more restricted than necessary, and a dynamically derived serialization order may avoid this type of abort.
The time stamp interval approach can be based either on TSO [7, 47], or on the OCC certification approach [11, 74]. However, as pointed out in [11], the time stamp intervals derived by the TSO-based approach are limited since the serialization order is determined as soon as the conflicts occur. There are different ways to explore the concept of time stamp interval based on the certification-oriented approach. The basic idea is that each version or value of a data granule is only valid for a certain period of time, i.e., between two consecutive updates. The transaction which has read a particular version of a data granule can only be certified with a time stamp in the valid interval of the data. If multiple granules are read, an intersection of their valid intervals, which may be null, has to be taken. Additionally, the granules updated by a transaction cannot be read by a certified transaction with a later time stamp.

In [74], the time stamp or interval of time stamps was dynamically derived by maintaining a limited time stamp history of accessing transactions for each data granule currently being accessed. At the transaction certification time, for each accessed granule the time stamp of the accessed version is compared with the time stamp history of the granule to determine its valid interval. This provides the information to re-order transactions at the certification time and to derive a back-shifted time stamp for certification in order to eliminate most unnecessary aborts due to read-write conflicts. This is referred to as certification based on Time Stamp History (TSH).

2.4 Run policy

Since conflicts exist when accessing data, a transaction may need to be restarted several times before it can be successfully committed. However, there is a dependency between the successive runs due to the buffer retention effect, which was first recognized by Yu et al [69, 71, 72, 73]. With sufficient amount of buffer, the data brought in during the previous runs of a transaction will continue to be retained in the buffer, and the rerun of the transaction will have fewer disk IOs, a shorter run time and a lower abort probability. The property that any rerun of a transaction accesses the same granules as those accessed during the first run even though the runs may be separated by those of conflicting transactions was called access invariance and was also explored in [18]. In [72, 73], it was shown that pure OCC outperforms broadcast OCC. That is to say during the first run of a transaction it is better to complete the execution, and bring in all the data blocks even after it is marked for abort.

Different CC schemes can be applied to the first-run and rerun transactions. In [73], various hybrid CC schemes and their performance were considered, including using broadcast OCC and 2PL for the rerun transactions and pure OCC for the first-run transactions. Although running transactions to the end makes sense during the first run, it is a waste on CPU resource for the subsequent runs. Thus, a better strategy is to use the pure OCC for the first-run transactions and the broadcast OCC for the rerun transactions. Alternatively, since information about all the locks
requested by the transaction can be obtained at the end of the first run from the CC manager, a variant of 2PL, referred to as static 2PL, can be used to obtain all the required locks before the rerun. This would reduce the blocking effect of the conventional (dynamic) 2PL, and guarantee that each transaction at most runs twice.

3 Integrating concurrency control with real-time requirements

Most existing scheduling algorithms for real-time tasks are not directly applicable to scheduling real-time transactions, because scheduling algorithms for real-time tasks typically require as input the arrival time, deadline, worst-case execution time, and criticality of each task. However, in a database environment the worst-case execution time cannot be meaningfully estimated, except maybe in the case of main memory databases [2]. Since transaction executions typically involve both CPU and I/O and the I/O rate (i.e., the buffer miss rate) is hard to predict and has a dominant effect on the response time, a worst-case estimate assuming a zero buffer hit probability is not meaningful.

On the other hand, a CC scheme may inhibit the proper functioning of a real-time scheduling algorithm. For example, under standard 2PL, a lower priority transaction may hold a lock requested by a higher priority transaction, thus forcing the higher priority transaction to wait for the lower priority one. Furthermore, real-time CC may need a different set of criteria for conflict resolution. While transaction response time and throughput are the metrics to measure conventional database systems, the percentage of transactions satisfying the timing constraints or deadlines tends to be the measure for real-time databases. Upon conflict, there is a trade-off of either favoring the transactions with a higher priority (or earlier deadline) or the transactions that are nearly completed or have made more progress. Overly favoring the higher priority transactions can create two side effects. First of all, more CPU resource can be wasted, if nearly completed transactions are aborted to favor the higher priority transactions. Second, with transactions of variable lengths, longer transactions tend to get discriminated against and thus are much less likely to meet their deadlines.

In this section, we focus on the integration of conventional CC with real-time scheduling. Maintaining data consistency, existing CC algorithms are extended to real-time environments. Approaches that may relax data consistency in order to guarantee that the deadlines be met will be discussed in Section 5.
3.1 Scheduling real-time transactions

3.1.1 Priority inversion

In a real-time database environment, conflict resolution of the CC scheme may interfere with CPU scheduling. When blocking is used to resolve a conflict such as in 2PL, a priority inversion phenomenon can occur if a higher priority transaction gets blocked by a lower priority transaction. In [57], a priority inheritance approach was proposed to address this problem, where the lower priority transaction that is blocking any higher priority transactions inherits the highest priority among the transactions it has blocked.

With priority inheritance, however, the low priority transaction that has inherited a higher priority may still be blocked by other lower priority transactions. Consequently, a higher priority transaction may have to wait for the completions of several lower priority transactions. In [57, 58], a read/write priority ceiling protocol was proposed where a high priority transaction can be blocked by at most the duration of a single low priority transaction. A means is provided so that a total ordering can be maintained among the transactions currently holding locks. It requires that transactions have prespecified fixed priorities and information be maintained for each data granule on the highest priority transactions that may read and write the granule. However, the priority ceiling protocol reduces the effective concurrency to one, resulting in an idle CPU during an IO. Thus, this protocol is more suitable for main memory type of database, unless the data granules required by a transaction can be prefetched before its execution.

In [14], a dynamic priority ceiling protocol was considered. It is an extension of the priority ceiling protocol in [57] and is based on the earliest deadline first, instead of fixed priority. Every resource has a dynamic priority ceiling and the priority ceilings of resources are updated at every task completion time.

3.1.2 Scheduling priority

It is generally observed that scheduling transactions according to the earliest-deadline-first policy, in which priorities are based on transaction deadlines, can minimize the number of late transactions except when the system is highly loaded [41]. Thus, most research studies in real-time databases with hard deadlines have taken the earliest-deadline-first approach for scheduling priorities [2, 25, 27]. (In [2], the least slack was also considered and was shown to be inferior to the earliest deadline. In [58], a fixed priority was used for the priority ceiling approach.) However, for real-time databases with soft deadlines, other criteria can be used to assign scheduling priorities as well [12, 26]. For example, the criticality of a transaction can be defined and used for the scheduling priority [26].

In an environment where transactions are of variable sizes, longer transactions generally encounter a higher number of conflicts [73]. This is true even if the CC manager exercises a fair
policy or a policy that is biased toward the longer transactions in resolving conflicts, since the longer transactions access a larger number of granules. If the transaction length is not available \textit{a priori}, the earliest-deadline-first scheduling may not be able to give the longer transactions a higher priority early enough to make the deadline, thus creating the starvation (or fairness) problem.

However, if transactions can be classified into \( n \) different types based on their lengths, which are known \textit{a priori}, a better scheduling policy can be designed to address the issue of starvation. For example, a \textit{weighted priority} scheduling policy was proposed in [27]. The weighted priority for a transaction is calculated as \( (\text{dl} - t)/w(i) \), where \( \text{dl} \) is the transaction deadline, \( t \) is the scheduling time, and \( w(i) \) is a weighting factor for transactions of type \( i \) based on their lengths. (If \( w(i) = 1, i = 1, \ldots, n \), or \( n = 1 \), i.e., all transactions are of the same length, the scheme becomes the earliest-deadline-first scheme.)

3.1.3 Overload management

As mentioned previously, scheduling real-time transactions based on the earliest-deadline-first policy can minimize the number of late transactions in systems operating in a low to moderate level of load [2]. However, as the system becomes highly loaded, transactions would gain relatively high priorities only when they are close to their deadlines and, as a result, may not have enough time to complete before their deadlines. By executing these transactions that are too close to their deadlines can create a vicious cycle in such a way that other transactions that are further away from their deadlines get delayed and, as a result, will also miss their deadlines.

Therefore, as pointed out in [25], the earliest-deadline-first scheduling should only be applied to the largest subset of the transactions that can all be completed within their deadlines. This is based on the observation in [30] that if a given set of tasks that can be scheduled to meet their deadlines, an earliest-deadline-first scheduling should also meet all or (most of) the deadlines. Thus, some type of flow control needs to be put into place to detect a system overload so that the system does not partially execute transactions and abort them after missing their deadlines. It is much better off to abort a certain percentage of the transactions before they ever get executed so that the system does not get overloaded.

Alternatively, certain transactions can be put into a separate class with lower priorities, and are only executed when the CPU is free from executing other transactions that are scheduled to meet their deadlines as in [25], where an adaptive earliest deadline (AED) scheme was proposed. In AED, the incoming transactions are divided into a HIT group and a MISS (i.e., the overflow) group, depending on the current system condition. A feedback control mechanism is used to detect overload conditions and modify transaction group assignment accordingly. Priorities are assigned based on the earliest-deadline-first policy for transactions in the HIT group, while they are randomly assigned for transactions in the MISS group. Upon arrival, each transaction is randomly assigned
a unique key and the transactions are maintained in a key-ordered list. If the key is below the
threshold of the HIT group, it would be assigned to the MISS group. The threshold is adjusted
dynamically so that the number of transactions missing their deadlines in the HIT group is very
small.

As discussed in Section 3.1.2, starvation (or fairness) can be a problem when transactions are
of different lengths. Longer transactions are less likely to complete before their deadlines. When
the system is overloaded, the fairness problem becomes even more severe. In [49], an adaptive
earliest virtual deadline (AEVD) scheduling scheme, which is an extension of the AED scheme, was
proposed to address the fairness issue in an overloaded system. In AEVD, the virtual deadlines
are computed based on both arrival times and deadlines. Since transactions with longer execution
times will arrive earlier relative to their deadlines, the AEVD scheme can raise their priorities in a
more rapid pace as their durations in the system increase. Consequently, longer transactions can
exceed the priorities of shorter transactions that have earlier deadlines but later arrival times.

3.1.4 I/O scheduling

Two important issues of I/O scheduling in real-time databases need to be carefully examined. One
is the implication of scheduling order on I/O service time, and the other is the distinction between
reads and writes.

In I/O scheduling, the total service time of an I/O request depends strongly upon the scheduling
order. This is due to the fact that the seek time, the time to position the disk arm on the desired
track, is a major component of the total service time of a request and is strongly affected by
the scheduling order. In contrast, in CPU scheduling, the total service time of a CPU request
is largely independent of the scheduling order, except that some additional cost may be added if
preemption scheduling is used. Furthermore, in contrast to CPU services, I/O services are generally
non-preemptive. (In [62], a high-level design of a priority-driven preemptive I/O scheduler was
described, and the performance studies showed that the gain is very sensitive to the preemption

cost.)

To minimize the seek time, an elevator-type scheduling scheme is often used in traditional
disk scheduling. In [1], several real-time-oriented algorithms were proposed and studied. It was
shown that an algorithm combining the earliest deadline first and the elevator scan concept with
feasible deadline considerations can lead to a substantial improvement over conventional algorithms.
Under this algorithm, the disk head seeks toward the request with the earliest deadline, and services
other requests along the way. An estimation of the service time for a request is made before the
scheduling to determine whether the request can be serviced in time. If the deadline cannot be
met, the requesting transaction is aborted. Another scheme considered in [13] (which focused on
read-only workloads) groups disk requests into multiple queues based on their priorities, one queue
for each priority level, and applies the elevator scan algorithm to each queue separately.

Another important issue related to I/O scheduling in a database environment is that, while write requests which are typically executed after a transaction is committed can be deferred, read requests to the disk are usually synchronous in the sense that a transaction cannot proceed until its disk read is completed. One example of database systems that use deferred writes is IBM’s DB2 database system [65]. However, write requests can become synchronous when the number of deferred write pages exceeds a certain threshold in the buffer as in IBM’s DB2 [65]. The concern of a synchronous write after a certain number of deferred writes was studied in [1].

In [32], an alternative non-deferred write scenario was considered where disk write requests must be completed before the write locks can be released. (These different write strategies can have different recovery implications.) The read locks, however, are released when transactions begin to commit. Different priority schemes that distinguish read and write requests were considered. A dynamic priority scheme was proposed and shown to be effective. Under the scheme, a read request has the same priority as the issuing transaction and write requests are given the lowest priority except for the case with waiting transactions where priority inheritance is used.

3.2 Adapting CCs to real-time environments

3.2.1 Serialization order

Serialization order based on the start times like TSO is generally bad for conventional non-real-time databases, and would be even worse for real-time databases. In TSO, aborts occur when conflicting accesses to the same granules are out of the prespecified time stamp order. By favoring transactions that have a higher priority on CPU scheduling but start later, i.e., have a later time stamp, it will greatly increase the abort probability of transactions that have a lower priority but start earlier, i.e., have an earlier time stamp.

On the other hand, serialization order based on the completion times generally performs better under the conventional database environments, as evidenced by the better performance of OCC compared with TSO. The OCC scheme also goes naturally with the real-time environments in the sense that higher priority transactions can get scheduled earlier and committed according to the completion times.

However, serialization order based on granule access times, like the standard 2PL, is somewhat restricted since the lower priority transactions may have already made accesses to some of the granules needed by the higher priority transactions and can cause the higher priority transactions to wait for the lower priority ones. However, this can be solved using either an abort mechanism [2], or the priority inheritance mechanism if conflict resolutions are done through blocking, as discussed in Section 3.1.1.

It should be noted that, in real-time databases, transaction deadlines can be used as a guidance
for the serialization order to help resolve conflicts. Ideally, we simply want each transaction to commit before its deadline. However, it may be difficult in achieving that. For example, a real-time TSO that simply assigns time stamps according to the deadlines can be too restricted. This is due to the fact that, if a lower priority transaction with a larger time stamp already makes the conflicting accesses to granules required by a higher priority transaction, the higher priority transaction may have to be aborted. Nevertheless, this concept can be useful in certain situations and we'll explore it in later discussions about the two-level serialization type of schemes [5, 40, 60].

3.2.2 Conflict detection and resolution

A conflict can occur between a pair of transactions or between a single transaction and a set of other transactions. When a conflict between a pair of transactions is detected, one of the two transactions will be penalized. The penalized transaction can be either put into the wait state or be aborted. In a real-time environment, it is natural to pick the lower priority transaction as the candidate to be penalized.

Next consider the case that a single transaction may be in conflict with a set of other transactions. For example, in the standard 2PL, a transaction with a write request can conflict with a set of read lock holders. In OCC, a conflict is detected at certification time and a set of other transactions may be in conflict with the transaction that is requesting for certification. In either situation, some transactions in the conflict set may have a higher priority and others may have a lower priority than the requesting transaction. One possible strategy is to penalize the requesting transaction if there is any higher priority transaction in the conflict set. That is to say the transactions in the conflict set are penalized only if they all have a lower priority than the requesting transaction; otherwise, the requesting transaction gets penalized.

Other strategies can also be designed to take additional factors into consideration, however. For example, in OCC, the transaction requesting for certification may hold an advantage over the transactions in the conflict set, since it is close to completion. It can be too stringent if the requesting transaction is aborted simply because a single higher priority transaction is present in the conflict set. A more relaxed condition should be sought for. In [23, 24, 27], this dimension was explored for the broadcast OCC, as we shall discuss later in Section 3.5.

3.3 Real-time CCs in the literature

3.3.1 2PL and variants

As mentioned in Section 3.1.1, the conventional 2PL can suffer from priority inversion in real-time databases. Although 2PL can be used with priority inheritance to alleviate this effect, a cascade of blocking can still exist and the blocking duration of a high priority transaction can be substantial. The priority ceiling scheme in [57, 58] can reduce the blocking time to one transaction execution
time. However, it is still considered to be too restricted, since it is primarily for main memory databases and a full knowledge of the granule access pattern of a transaction is required to set the priority ceiling of each granule.

Note that 2PL tends to cause long blockings because it forces a transaction to delay its unlock actions until all locks have been acquired, even though some of the locked resources are only used early in the execution. Long blockings are definitely undesirable in real-time databases. In [46], the priority ceiling scheme has been extended to ensure that there are no long blockings. The new protocol, called the \textit{convex ceiling protocol} (CCP), checks the priority ceilings of those resources to be unlocked when a transaction does not need them any more. If the transaction will not lock any other resource with a higher priority ceiling, these resources are unlocked immediately. Serializability is still guaranteed in CCP, however. In other words, CCP reduces the worst-case blocking length for some high priority transactions and thus achieves a better schedulability condition than 2PL with the priority ceiling protocol.

In [2], another variant of 2PL was proposed, where abort is used for conflict resolution to avoid the priority inversion problem. This was referred to as the 2PL-HP (2PL with High Priority) in [23, 24]. In 2PL-HP, a conflict is resolved in favor of the higher priority transactions. When a conflict occurs, if the requesting transaction has a higher priority than all the lock holders, the lock holders are aborted and the requester gets the lock. Otherwise, it waits for the holder to release the lock. Thus, the 2PL-HP is in essence similar to the wound-wait scheme [55], which is one of the conventional CC schemes considered in Section 2.2.1. However, the two schemes differ in their priority assignment. In the wound-wait scheme, priorities are based on the arrival times instead of the deadlines, since the objective is fairness so every transaction will eventually receive a high enough priority to get through.

While 2PL with priority inheritance can face a cascade of blocking, the 2PL-HP can cause high resource utilizations due to the priority-oriented transaction aborts. In traditional database environments, various schemes, such as the LDB (locking with deferred blocking), have been proposed to strike a balance between waiting and abort, as explained in Section 2.2.3. The similar concept is applicable to the real-time database environments. One such compromised scheme, referred to as the \textit{conditional priority inheritance} (CPI) scheme in [28, 29], is to use the priority inheritance scheme only when the lower priority conflicting transaction is near its completion. Otherwise, the scheme operates similar to the 2PL-HP where the lower priority conflicting transaction is aborted. Assuming that the number of locks or granules required by each transaction is known \textit{a priori}, a transaction is considered to be close to completion if it is within $k$ granule accesses away from its completion, where $k$ is a prespecified threshold. This would protect the nearly completed transactions from being aborted.

Thus, the CPI scheme is close in spirit to the LDB scheme proposed for the traditional databases,
since both schemes avoid aborting transactions which have made more progress. In essence, under CPI once a transaction is only \( k \) more granule accesses away from its completion, it is equivalent to be in the blocking phase of the LDB scheme. The CPI scheme can therefore be viewed as consisting of two phases: non-blocking high-priority transaction phase and blocking high-priority transaction phase. However, differences exist between CPI and LDB. The only difference between them in the blocking phase is in the usage of the priority inheritance concept by CPI. However, the two schemes differ more significantly during the non-blocking phase; under CPI, a conflict is resolved at each granule access based on the priority, while under LDB, the conflict is resolved at the end of the non-blocking phase.

Various optimization techniques to increase concurrency discussed in Section 2.3.1 can be applied or generalized to real-time databases. Consider the two-level serialization approach. In [5], the ordered-sharing locking approach was extended to the real-time environments. The delaying-commit mechanism was extended as follows. When a transaction is eligible for commit, it is put on a wait state until either it reaches its deadline, or the original requirement on the ordered-sharing scheme is satisfied. In the former case, it also needs to abort all preceding transactions with which it has an ordered-shared relationship [5].

Another approach to exploring the two-level serialization concept is the scheme proposed in [40, 60] which dynamically adjusting the serialization order. The scheme provides a more flexible compatibility matrix to serialize the granule accesses among transactions as compared to the read-write compatibility matrix of the conventional 2PL. In [40, 60], it was assumed that writes are not performed in-place and transactions go through three phases: read/execution phase, waiting for commit phase and final commit phase. The compatibility matrix depends on the current phase of the lock holder. The transaction level serialization is provided by a separate delaying-commit mechanism. The unique part of the scheme is the recognition of the fact that the scheduling is in favor of the higher priority transactions and that the CC schemes tries to achieve dynamically a serialization order for conflicting transactions based on the deadline or priority. (Although this scheme is proposed under the real-time database environment, the only part that is specific to real-time is the serialization ordering based on the priority. It should be applicable in a more general context, for example by using time-stamp ordering instead of priority.) It thus differs from the ordered-sharing scheme in the sense that serialization order for transactions is not based on the granule access order. At the granule level, a subsequent read request of a higher priority transaction is considered to be compatible with the write lock from a lower priority transaction not yet reaching the commit phase. At the transaction level, the lower priority transaction will only be committed after the higher priority transaction is committed, if appropriate.
3.3.2 OCC

To adapt OCC into the real-time database environments, the issue is how to incorporate priorities into conflict resolution. In [23, 24], this consideration was specifically applied to broadcast OCC. As the certifying transaction may conflict with a set of transactions, where some may have a higher priority and others may have a lower priority than the certifying transaction, three approaches on conflict resolution were considered. If there is no higher priority transaction in the conflict set, the certifying transaction will always be committed. Otherwise, the first approach immediately aborts the certifying transaction, and the second one takes a delayed abort (see Section 2.2.1) as in [19, 68], instead of immediate abort. The third approach uses a different concept which tries not to penalize the requesting transaction unless the percentage of higher priority transactions in the conflict set exceeds a prespecified threshold.

3.3.3 Multiversion schemes

In [32], several extensions of the multiversion 2PL scheme were considered for real-time databases, including both the two-version 2PL and multiversion 2PL. The extensions are focused mainly on the conflict resolution mechanism. For example, in the two-version 2PL case, the priority inversion problem can occur either directly or indirectly. It can occur directly if a higher priority transaction makes a lock request on a granule which is already locked by a lower priority transaction in a conflicting mode. It can occur indirectly, e.g., if a lower priority transaction makes a read request on a granule which is already write-locked by a higher priority transaction, or if a higher priority transaction makes a write request on a granule which is already read-locked by a lower priority transaction. Although the requested lock can be granted in either case in the traditional two-version 2PL, it can cause the higher priority transaction later on to delay its commit until the lower priority transaction terminates in real-time databases. The direct priority inversion issue can be addressed by aborting the lower priority transaction. However, for the indirect case, the lower priority transaction can either be put into a wait state or be aborted. Furthermore, dynamically adjusting the serialization order can also be made to favor the higher priority transactions [32].

3.4 Extending other CCs to real-time environments

Even though the pure OCC scheme tends to be the better performing OCC in conventional databases [72], there is no study on extending it to real-time databases. However, we expect the pure OCC scheme can be extended to real-time databases in a way similar to the broadcast OCC scheme. The conflict resolution is similar. The only difference is that once the conflict resolution is made and, if the penalized transaction is not the certifying transaction, the abort would be delayed instead of being immediate.
There also have been no previous studies on extending the dynamic interval time stamp approach in [11] and the TSH scheme in [74] to real-time database environments, though the extension is straightforward as any other OCC schemes. The issue is still on the conflict resolution when a certifying transaction is in conflict with one or multiple ongoing transactions (some of them may be with a higher priority). This is the very same problem encountered by other OCC schemes like the broadcast OCC [23, 24]. Since these dynamic time stamp schemes make more transactions eligible for commit by reducing the number of read-write conflicts, they would be expected to perform better in real-time environments than the pure OCC scheme or broadcast OCC scheme.

A real-time-oriented WDL scheme, referred to as RWDL, can also be designed. Consider the case with a wait depth of one and exclusive locks only. In RWDL, the $V(i)$ function can simply be defined to be the priority or deadline of the transactions. The abort rule when the wait length exceeds one will then be based on the priority instead of the progress that the transaction has already made. Priority inheritance can still be applied to the wait case. As the wait depth is limited to one, the wait time on each lock conflict will not exceed the execution time of a single transaction. A more sophisticated approach would be to design $V(i)$ so as to strike a balance between the considerations on the priority (or earlier deadline) and the work in progress. For example, in RWDL, $V(i)$ can be defined as a pair of functions $(V_1(i), V_2(i))$, where $V_2(i)$ is based on the priority and $V_1(i)$ can be based on the number of locks obtained as in the conventional database environments. If one of the conflicting transactions has a $V_1(i)$ that exceeds some prespecified threshold $k$, $V_1(i)$ would be used as the criterion to abort transactions. Otherwise, $V_2(i)$ would be used as the criterion to abort transactions.

3.5 Further remarks on conflict resolution

Performance studies on CC in the traditional database environment, such as [19, 70], have shown that the abort-oriented conflict resolution approaches that favor transactions making more progress tend to improve the transaction throughput and response time. Intuitively, aborting a transaction near completion has to be more expensive than aborting a transaction just started. In a variable-length transaction environment, longer transactions need a more favorable treatment from the CC manager to avoid a large number of restarts.

However, in a real-time transaction environment, the timing constraint is generally the first concern. When abort is used to resolve transaction conflicts, the abort candidates under most real-time CC schemes tend to be the lower priority transactions regardless of other factors. Clearly, this can become a problem when the variation of the transaction lengths becomes large. In an OCC-type scheme, a long transaction suffers in two ways. Since it accesses a larger number of granules and takes longer to execute, it is more likely to be marked for abort before it reaches commit. This is true even in the conventional database environment. However, in the real-time
environment, even if a long transaction reaches a commit stage, since it has accessed more granules, it is more likely that its conflict set (whose average size is proportional to the number of granules accessed) includes some higher priority transactions.

Clearly, not only the deadline requirement but also the need to take care of longer transactions and transactions further along in their deadlines are also important factors to be considered in choosing the abort candidate(s). A cost or priority function which can trade off these factors needs to be developed. The abort cost function need not be the same as the CPU scheduling priority. However, a CPU priority other than the earliest-deadline-first can also be devised to help long transactions. The weighted priority scheduling policy in [27] (discussed in Section 3.1.2) is such an example. The performance study in [27] considered a mix of two transaction types with different fixed lengths known \textit{a priori} to the scheduler. The study showed that, although the average rate of meeting the deadline does not change by giving the long transaction type a higher weight under a variant of the broadcast OCC scheme based on synchronized weak locks discussed in Section 2.2.2, the percentage of long transactions missing the deadlines can be reduced at the expense of short transactions. Also, the 2PL-HP was relatively indifferent to the weighted priority scheduling policy.

In [28, 29], it was shown that the CPI scheme that generalizes the 2PL-HP to avoid aborting lower priority transactions that are near completion can improve performance. In [23, 24], it was shown that under the broadcast OCC, taking the strategy to abort a certifying transaction just because it conflicts with a higher priority transaction, which may be in its early stage, leads to an inferior performance. Therefore, delaying the abort to see whether that higher priority transaction can make it to the commit stage can improve performance. However, the delayed certifying transactions can still be aborted by other committing transactions during the wait.

Experimentally shown in [23, 24], an abort resolution scheme based upon the percentage of higher priority transactions in the conflict set can lead to a better performance when the percentage threshold was set to 50% in order to abort the certifying lower priority transaction. Although such an abort criterion was somewhat ad hoc, it did show that aborting lower priority transactions which are near completion, or further along in their progress, can be sub-optimal. In contrast, in [27], two OCC schemes favoring higher priority transactions in conflict resolution were evaluated. These two schemes put the lower priority certifying (nearly completed) transactions in a delayed aborting state, if the number of higher priority transactions in the conflict set reaches 100% and 50% thresholds, respectively. However, the study showed that neither scheme leads to a better performance (in terms of meeting the deadlines) than the conventional broadcast OCC which is in favor of the certifying transaction. These seemingly contradictory results in [27] and in [23, 24] show that it can be difficult in identifying a set of simple and general criteria that can be applied to make optimal trade-offs between aborting the earlier deadline transactions and the nearly completed transactions.
Another observation from the performance study in [23, 24] is that the conventional CC scheme, specifically the broadcast OCC, is comparable on the percentage of missed transactions to its best performing real-time-oriented variation once the level of resource contention becomes high. That is to say a fair conflict resolution scheme to preserve work completed may not be worse than a priority- or deadline-oriented conflict resolution scheme even in terms of meeting the deadlines.

Similar type of conflict resolution problem also exists for 2PL-type schemes using abort. Consider the case of RWDL with multiple lock modes like read and write. (In [19], only exclusive mode was considered and this issue did not arise.) Let us examine the following example. A transaction, which already has some other transactions waiting for it, makes a write lock request against a granule with multiple read lock holders. The issue is whether to abort the requester or to abort the multiple holders in the conflict set as some holder may have higher priorities and more or fewer locks held and some other may have lower priorities and more or fewer locks held than the requester. More studies are needed to understand the trade-offs of the different conflict resolution schemes.

### 3.6 Run policy

In contrast to the conventional database environments, the dimension of run policy has not yet been well explored in the real-time database environments. (An initial work in this direction is the work in [48] where a transaction is executed in two phases: a *prefetch* phase during which all the data are brought into the main memory without considering data conflicts, and an *execution* phase during which the transaction is executed.) The run policy can in fact be even more critical in the real-time environments. By tracking the number of locks obtained or granules accessed during the first-run of a transaction, one can get a rough estimate of the transaction length. With knowledge on transaction length for the rerun transactions, better scheduling can be made to meet the deadline requirement. This is especially important for the longer transactions if the workload has a mix of variable length transactions.

Consider the following run policy, which is an extension of one of the policies considered in [72] for the conventional database environment. In the first run, all transactions will be executed to the end to bring in all the required data granules, even if the transactions are marked for abort. The transaction length would then be estimated based on the number of locks requested, or some other run-time statistics. The list of locks accessed during the first run can be obtained from the CC manager at the end of the first run. In the second run, a static locking policy can be adopted, where locks on all granules required are obtained before a transaction is executed. If it is needed to reduce the waiting time to obtain all these locks, all the conflicting transactions with lower priorities can be aborted. Since there is no I/O requirement and all locks are acquired before execution, if given the highest priority, the transaction can run to completion in one burst without intervening I/O, CPU and lock waits. No multiprogramming is needed in a single processor.
system, either. Thus, we can schedule all the rerun transactions based on their deadlines in a serial order. Since not only the deadline but also a reasonable estimate of the execution time is known, we can apply conventional earliest-deadline-first scheduling technique to substantially enhance the number of transactions meeting the deadlines. Note that with a good run policy the issue of longer transactions being discriminated may also be partially solved.

For each rerun transaction, at the end of its first run, the CPU scheduler can determine its position in the CPU schedule based on its deadline. If the transaction can make the deadline, it is inserted into the schedule, otherwise it is aborted. The schedule can also maintain information on the laxity affordable by each rerun transaction in the schedule. Note that the execution times of rerun transactions are orders of magnitude less than the first-run transactions. The blocking effect from the currently running rerun transaction, if any, tends to be very small. The major concern is the effect of the first-run transactions. Hence, aborting the currently running rerun transaction to favor the new rerun transaction with a higher priority in case of conflict should not be necessary. (This is in contrast to the main memory database environment considered in [2], where all transactions are of similar lengths and serial executions of transactions are compared less favorable to parallel executions.)

Certainly, there are still first-run transactions to be scheduled. One way is to schedule transactions strictly based on their priorities or deadlines. The side effect is that some of the more deterministic rerun transactions in the schedule may need to be aborted, if a higher priority, first-run transaction holds on to the data resources for too long a period. Another alternative is to give a bias toward the rerun transactions as they have less uncertainty on the execution time estimate. If a first-run transaction has a higher priority, it would be allowed to execute until there is no more laxity for some of the rerun transactions in the schedule. At that time, the first-run transaction is delayed or aborted so that the rerun transactions on the schedule can be executed to meet their deadlines.

4 Special requirements of hard real-time applications and their implications

4.1 Hard real-time requirements

Since many hard real-time systems are used in safety-critical applications, they must provide high and predictable performance. For some applications, missing the deadlines simply is not acceptable. Also, an unpredictable system can do more harm than good under abnormal conditions. Although many database systems are fast enough for traditional applications in which a response time of a few seconds is often acceptable, these systems may not consistently provide an acceptable response time for certain real-time applications. As mentioned before, there are many reasons why traditional
database systems may have unpredictable performance. For example, under 2PL, blocking will cause transactions to be delayed and it is often difficult for a transaction to predict how long the delay may be since the blocking transactions themselves in turn may be blocked by other transactions. Additionally, transaction processing typically requires accesses to disks, and the response time is often unpredictable.

Moreover, databases in hard real-time systems may also have the following unique problems:

1. Many data objects in a database may correspond to active data objects in the real world. Their values may be subject to change by themselves, regardless of the database state and activities.

2. A real-time database may never be completely correct. As soon as a real-world value is recorded in the database, it may already be out of date.

3. Different data objects in a database may be recorded at different times. Their values may not co-exist in the same real-world snapshot.

Thus, real-time databases may need special CC schemes to ensure that transaction results are correct with respect to the above problems.

4.2 Implications

As mentioned before, one of the most interesting challenges in building real-time databases is the integration of CC and real-time scheduling such that concurrency and resource utilization can be maximized subject to three constraints: data consistency, transaction correctness, and timing constraints [63]. To meet timing constraints, many real-time task scheduling algorithms can be extended to transaction scheduling, while CC schemes are still used to maintain data consistency. These approaches have been discussed above in Section 3. However, such an approach has the inherent disadvantage of being limited by the CC, since existing CC schemes resolve conflicts mainly by a combination of two measures: blocking or abort of transactions (Section 2.2). Both are barriers to meeting time-critical schedules.

For certain real-time transactions with hard deadlines, being limited by the CC can have another severe drawback. Note that hard real-time transactions must be guaranteed that their deadlines will be met. To make such a strong guarantee, we cannot simply use the best-effort scheduling algorithms. For example, under 2PL, we must have scheduling algorithms which can control the locking behavior to guarantee the locking delays. To do this, advance knowledge on the resource and data requirements of transactions may be required.

An alternative is to relax the serializability requirement of real-time databases. Traditional CC schemes induce a serialization order among conflicting transactions. Although in some applications
weaker consistency is acceptable [20], a general-purpose consistency criterion that is less stringent than serializability has been difficult to find. However, since real-time databases can have a different notion of transaction correctness (see Section 5), we may trade the serializability for other database performances. Based on the argument that timing constraints may be more important than data consistency in real-time databases, attempts have been made to satisfy timing constraints by sacrificing database consistency temporarily [36]. As long as other important requirements are satisfied, a transaction may want to provide a result using acceptable but non-serializable information.

It should be noted that the above observations may not be applicable to all real-time database systems. Since many real-time transactions have soft deadlines and many systems can use slightly out-of-date information, conventional CC schemes integrated with real-time scheduling, such as the ones discussed in Section 3, may be perfectly acceptable. The motivation here is to point out that opportunities as well as needs exist for designing new, more powerful CC schemes for next generation real-time database systems. Several of such approaches will be presented in the following section.

5 Relaxing serializability in real-time transactions

For certain real-time applications, serializability may not be necessary for concurrent transactions. To facilitate timely executions and meet the deadlines, we may wish to relax the definition of correctness in database transactions. Since hard real-time systems are often used to respond to external stimuli (e.g. in combat systems) or to control physical devices (e.g. in auto-pilot systems), a timely and useful result is much more desirable than a serializable but out-of-date response. As long as the result of a transaction is correct with respect to the situation in the real world, whether or not the transaction is serializable with other transactions may not be a concern to the application. Depending on the semantics and the requirements of transactions, a real-time system may be able to apply different measures to different transactions.

In this section, we first introduce a set of new correctness criteria. We then review the techniques for producing correct but non-serializable real-time database schedules. All techniques utilize some semantic information about the transactions or data objects so that a system may produce schedules that are acceptable to specific applications.

5.1 External and temporal consistencies

Databases may be used by real-time systems to store information about physical devices and operation environments. Since the real world is always changing, it is up to real-time systems to ensure that their databases are always consistent with the real world. Ideally, a data object in a real-time database should always contain the exact value of its real-world counterpart. However, this may be impossible or too expensive to implement for most applications. A less expensive
and more practical solution is to make sure that all data objects read by a real-time transaction have acceptable approximations of their real-world values. We call this requirement the external consistency requirement for transactions.

Another important issue for real-time transactions is that the values of the objects used by a transaction must be from the same real-world snapshot, i.e., the values must have existed at approximately the same time. If a transaction uses some new facts mixed with old facts, the transaction may have an incorrect picture about the real world and thus makes a wrong decision. We call this requirement the temporal consistency requirement for transactions.

Assume that a transaction $T_i$ is a sequence of operations, $O_{ij}(D_{ij})$, $O_{i2}(D_{i2}), \ldots$, where each operation $O_{ij}$ accesses only one data object $D_{ij}$. A data object, however, may be used in more than one operation in a transaction. A transaction thus can be defined by a sequence of $O_{ij}(D_{ij})$'s. A history $H$ of a set of transactions $T$ is thus a sequence of $O_{ij}(D_{ij})$ from many transactions $T_i$'s. To reason about the desirable history for real-time databases, we define a timestamp $S$ for each of the transaction operations. The timestamp of each operation $S(O_{ij})$ in a history is defined to be the real time when the operation is performed. The timestamp of the object $S(D_{ij})$ is the time when $D_{ij}$ was created.

Given a history of transaction executions, the external consistency requirement can be defined by the following equation for all read operations $[38]$:

$$\forall i, j, |S(O_{ij}) - S(D_{ij})| \leq \epsilon_{ij}.$$  

The equation specifies that, for each read operation of a real-time transaction $T_i$, the data value read by the operation must be within the valid lifespan $\epsilon_{ij}$ of the data. $\epsilon_{ij}$ may depend also on the semantics of the operation $O_{ij}$. In most practical applications there should be a single $\epsilon$ value for each data object type.

To check for the temporal consistency of a transaction, we need to compare the timestamps for all data objects read by a transaction. In other words, transaction $T_i$ may require that the difference between the timestamps of all objects it reads to be smaller than $\delta_i$:

$$\forall i, j, k, |S(D_{ij}) - S(D_{ik})| \leq \delta_i.$$  

Sometimes, the data in a data set may have a strong temporal consistency constraints. For example, the three dimensional attributes of an aircraft location must be temporally consistent whenever they are used in a computation. In that case, we can define a temporal consistency requirement for a data set $U$ in terms of $\xi_U$:

$$\forall i, j, k, \text{ where } \{D_{ij}, D_{ik}\} \subseteq U, |S(D_{ij}) - S(D_{ik})| \leq \xi_U.$$  

The idea of temporally consistent data set can be compared to the Atomic Data Set (ADS) proposed by Rajkumar [52]. In the ADS approach, a database is decomposed into disjoint ADS's.
and uses the modular CC scheme for real-time database CC. The consistency of each ADS can be maintained independently of the other ADS's. A setwise two phase locking scheme is then used to make sure that transactions are run serializably with respect to each of the atomic data sets. However, no concept of temporal consistency is defined in ADS. We believe that the two concepts are compatible and schemes can be implemented to satisfy both of them.

An extensive set of simulations have been performed in [61] to study the performance of various CC schemes in maintaining the temporal consistency of data in hard real-time systems. A multiversion database model is assumed in the study and the transactions are assumed to be mostly periodic. The study compares the 2PL and the OCC schemes, and finds that the OCC scheme is less effective in maintaining temporal consistency.

5.2 Guaranteeing consistencies by assigning periods

To guarantee that all real-time consistency requirements are satisfied, we need to have scheduling algorithms which can guarantee a predictable performance. We distinguish those transactions that update a real-time database to reflect the real-world values from those transactions that retrieve data from the database. This is because update transactions are often executed periodically and it is up to the real-time database system to decide how often they should be executed. To make sure that transactions in a real-time database are always externally and temporally consistent, the following issues must be discussed [37]:

1. Given a set of consistency requirements, can we define the period of each update transaction so that the consistency requirements are always satisfied? In other words, how do we convert the timing consistency requirements into the periods of update transactions?

2. Given a set of periodic real-time transactions, what level of temporal and external consistency can we always guarantee?

In [37], the consistency requirements are converted into the upper and lower bounds for update transaction periods. The idea is that whenever a retrieve transaction is executed, it must be able to find an externally consistent value for each of the objects it reads. If the external consistency for \( O_{ij} \) is \( \epsilon_{ij} \), there must be a version of \( D_{ij} \) created within the past \( \epsilon_{ij} - \epsilon_i \) time where \( \epsilon_i \) is the execution time of \( T_i \). Therefore, the update transaction for data \( D_{ij} = D_k \) must be executed with a period \( P_k \) as follows:

\[
\forall i, j, k, \text{ where } D_{ij} = D_k, \ P_k \leq (\epsilon_{ij} - \epsilon_i).
\]

Given two update transaction periods, we can derive the maximum distance between their versions. In general, the maximum distance \( d \) between any version of \( D_i \) and any version of \( D_j \) is

\[
d(D_i, D_j) \leq \frac{\min(P_i, P_j)}{2}.
\]
where \( P_i \) and \( P_j \) are the update periods for \( D_i \) and \( D_j \), respectively.

After the bounds for the period of each update transaction are defined, we can assign a period length to each update transaction so that all retrieve transactions can be guaranteed to find externally and temporally consistent data in their executions.

5.3 Trading serializability for external consistency

For some real-time systems, it may be difficult for the system to satisfy both the external and temporal consistencies, while maintaining the serializability of transactions. On the other hand, external consistency can be satisfied only if update transactions are not blocked (for too long) when recording new values in the database. Several techniques have been proposed in the literature to allow transactions to be executed even when the serializability may not be ensured. In this section, we review several such techniques.

Based on the concept of external consistency, Lin [36] has classified the operations in a real-time transaction into two classes: those in the \( E\)-part recording external events in the database (i.e. to maintain external consistency) and those in the \( I\)-part maintaining internal consistency. If a real-time transaction has a stringent deadline, a database may allow the internal consistency to be ignored temporarily in order to complete the transaction before its deadline. With the division of I-part and E-part in each transaction, a transaction compatibility table (TCT) can be defined in a system. During run-time, when a real-time transaction is scheduled, the table is inspected to see if it needs to wait for the completion of those transactions arrived earlier. A transaction requiring an externally consistent data set must wait until all updates by its predecessor transactions are finished. A transaction is a \textit{predecessor} transaction of \( T \) if it updates some data objects that will be used by \( T \). To decide whether a transaction \( T_2 \) should be executed after transaction \( T_1 \) which arrived earlier, TCT\((T_1, T_2)\) is inspected. If \( T_1 \) is not a predecessor of \( T_2 \), it is acceptable to execute \( T_2 \) before \( T_1 \). Otherwise, we must finish \( T_1 \)'s E-part before starting \( T_2 \). After that, depending on the relationship between \( T_1 \) and \( T_2 \), we may be able to delay or even omit \( T_1 \)'s I-part.

The TCT approach is useful if some real-time transaction must be executed immediately and the serializability measures have prevented the transaction from doing so. On the other hand, to use the technique, we must pre-analyze all transactions to see if they are compatible with each other, which is a tedious process. The other issue is the run-time efficiency in using the TCT approach. Thus the technique may be more suitable for pre-run-time analysis when an optimal static real-time schedule is to be generated.

Another approach to relaxing serializability is the epsilon serializability (ESR) [50, 51, 67]. ESR is a generalization of serializability that explicitly allows a limited amount of inconsistency in transaction processing in order to enhance the degree of concurrency between update transactions and queries. Here, queries are read-only transactions. ESR controls the amount of inconsistency
with each inconsistent state, defined by its derivation (or a distance) from a consistent state. The bounded inconsistency in ESR is automatically maintained by a divergence control (DC) algorithm [67]. DC algorithms are designed by systematically relaxing the corresponding CC algorithms during conflict resolution [67]. When a read-write (or write-read) conflict is detected and the inconsistency accumulated so far is still within the specified limit, then the conflicting access is granted. Otherwise, the access is denied and a typical conflict resolution method, such as blocking or abort, can be used. Depending on the amount of inconsistency tolerable by the read-only queries, the reduction in the amount of data contention between queries and update transactions can be substantial [31].

ESR has several important applications in real-time database systems. A concrete example of ESR application is replication control in distributed real-time databases. It offers the possibility of maintaining mutual consistency of replicated data asynchronously. A distributed real-time database which supports ESR permits temporary and limited differences among data object replicas; these replicas are required to converge to the standard one-copy serializability as soon as all the update messages arrive and are processed. A recent simulation study shows a significant improvement in terms of system responsiveness can be achieved by using ESR in a distributed real-time database system as measured by the number of transactions that meet their deadlines [59].

ESR does not rely on application specific semantics of queries except the fact that they are read-only. However, there have been approaches that take application-specific semantics into consideration to support real-time database applications [33, 35]. One such approach, using the concept of similarity among data [35], was proposed as a correctness criterion for CC in real-time data-intensive applications. In this study, a real-time database is considered as a collection of data objects that are used to model the real world. Due to the dynamic nature of the real world, a real-time database can only capture the values of real-world objects up to a certain precision. Thus, the consistency constraints on real-time databases are inherently concerned with imprecise values [39]. With the concept of similarity, more concurrency can be allowed, and the level of data contention between queries and transactions can be reduced. Scheduling algorithms using this technique are being investigated.

6 Conclusions

Concurrency control in conventional databases and real-time scheduling are two separate research areas that have been actively studied. However, research in real-time databases is still in its early stages. Conventional database systems are not designed for time-critical applications and lack features required for supporting real-time transactions. Unfortunately, existing real-time scheduling algorithms are not directly applicable to real-time databases. Meeting the requirements of real-time databases will require a balanced and coordinated effort between concurrency control and
transaction scheduling. In this paper, we tried to bridge the gap between these two separate research efforts to facilitate future research in real-time databases.

We have examined the approaches to integrating existing CC schemes with real-time scheduling algorithms. To meet more deadlines, CC schemes can be modified to favor more urgent transactions. We also studied the approaches that explore the non-serializable semantics in real-time applications. Since real-time database systems need to maintain external and temporal consistency, non-traditional measures of correctness could be employed to improve the performance and to meet the timing constraints of real-time transactions.

For real-time transactions with soft deadlines, conventional database CC schemes modified to provide preferential treatment to high priority transactions may be sufficient. Many applications require only real-time database systems that can support this type of transactions. A primary goal of such database systems is to minimize the number of transactions that miss the deadlines. However, for hard deadline critical transactions, the system should provide a guarantee that they will be completed by the deadlines. For these transactions, non-serializable but still semantically correct execution in a timely fashion is highly desirable.

Real-time database systems have some unique requirements. Over the near future, applications that need real-time databases would become large and complex, distributed, and need to operate in a highly dynamic environment. For some applications, missing certain timing constraints could have catastrophic consequences. The design and implementation of real-time database systems for such applications introduce many new and interesting problems. Meeting the challenges from all of the characteristics would require more extensive and coordinated research efforts in many of the topics listed below:

- modeling techniques and language constructs for real-time transactions and databases to specify timing properties and temporal consistency in an unambiguous manner. Relationships between consistency constraints and timing constraints need to be easily and clearly specified.
- priority-based scheduling algorithms and concurrency control schemes that can, in an integrated and dynamic fashion, manage transactions with precedence, resources (including communication resources and I/O devices), and timing constraints. In particular, resource allocation policies and distributed transaction management schemes must be integrated.
- new metric for database correctness, performance, and predictability. Methods that enable the trade-offs between serializability and timeliness, between precision and timeliness, and other types of trade-offs that can be used to improve the new real-time performance metric need to be studied.
- database architecture that supports predictable executions of real-time transactions. Since a database system must operate in the context of a given operating system, methods to integrate
operating system functions with that of database systems in a cooperative and predictable manner are necessary.

References


