Real Time Database Systems

Jan Lindström
Solid, an IBM Company
Itälahdenkatu 22 B
00210 Helsinki, Finland
jan.lindstrom@fi.ibm.com

March 25, 2008

1 Introduction

A real-time database system (RTDBS) is a database system providing all features on traditional database system such as data independence and concurrency control, while at the same time enforces real-time constraints that applications may have [13]. Like a traditional database system, a RTDBS functions as a repository of data, provides efficient storage, and performs retrieval and manipulation of information. However, as a part of a real-time system, tasks have time constraints, a RTDBS has the added requirement to ensure some degree of confidence in meeting the system's timing requirements [51]. A real-time database is a database in which transactions have deadlines or timing constraints [82]. Real-time databases are commonly used in real-time computing applications that require timely access to data. And, usually, the definition of timeliness is not quantified; for some applications it is milliseconds, and for others it is minutes [96].

Traditional database systems differ from a RTDBS in many aspects. Most important RTDBSs have different performance goal, correctness criteria, and assumptions about the applications. Unlike a traditional database system a RTDBS may be evaluated based on how often transactions miss they deadlines, the average lateness or tardiness of late transactions, the cost incurred in transactions missing their deadlines, data external consistency and data temporal consistency.

For example, a stock market changes very rapidly and is dynamic. The graphs of the different markets appear to be very unstable and yet a database has to keep track of current values for all of the markets of the Stock Exchange.

Numerous real-word applications contain time-constrained access to data as well as access to data that has temporal validity [77]. Consider for example a telephone switching system, network management, navigation systems, stock trading, and command and control systems. Moreover consider the following tasks within these environments: looking up the ”800 directory”, obstacle detection and avoidance, radar tracking and recognition of objects. All of these entail gathering data from the environment, processing information in the context of information obtained in the past, and contributing timely response. Another characteristic of these examples is that they entail processing both temporal data, which loses its validity after a certain time intervals, as well as historical data.

Traditional databases, hereafter referred to as databases, deal with persistent data. Transactions access this data while preserving its consistency. The goal of transaction and query processing approaches chosen in databases is to get a good throughput or response time. In contrast, real-time systems can also deal with temporal data, i.e., data that becomes outdated after a certain time. Due to the temporal character of the data and the response-time requirements forced by the environment, tasks in real-time systems have time constraints, e.g. periods or deadlines. The important difference is that the goal of real-time systems is to meet the time constraints of the tasks.

One of the most important points to remember here is that real-time does not just mean fast [96]. Furthermore, real-time does not mean timing constraints that are in nanoseconds or microseconds. Real-time means the need to manage *explicit* time constraints in a predictable fashion, that is, to use time-cognizant methods to deal with deadlines or periodicity constraints associated with tasks. Databases are useful in real-time applications because they combine several features that facilitate (1) the description of data, (2) the maintenance of correctness and integrity of the data, (3) efficient access to the data, and (4) the correct executions of query and transaction execution in spite of concurrency and failures [81].

Previous work on real-time databases in general has been based on simulation. However, several prototypes of general-purpose real-time databases has been introduced. One of the first real-time database implementations was the disk-based transaction processing testbed, RT-CARAT [43]. Some of the early prototype projects are the REACH (Real-time Active and Heterogeneous mediator system project [15], the STRIP (Stanford Real-time Information Processor) project [6].

Kim and Son [53] have presented a StarBase real-time database architec-

ture. This architecture has been developed over a real-time microkernel operating system and it is based on relational model. Wolfe & al. [106] have implemented a prototype of object-oriented real-time database architecture RTSORAC. Their architecture is based on open OODB architecture with real-time extensions. Database is implemented over a thread-based POSIX-compliant operating system. Additionally, DeeDS project at the University of Skövde [10] and the BeeHive project at the University of Virginia [98] are examples of more recent RTDBS prototype projects.

Another object oriented architecture is presented by Cha & al. [17]. Their M2RTSS-architecture is a main-memory database system. It provides classes, that implement the core functionality of storage manager, real-time transaction scheduling, and recovery. Real-Time Object-Oriented Database Architecture for Intelligent Networks (RODAIN) [55], is an architecture for a real-time, object-oriented, and fault-tolerant database management system. The RODAIN prototype system is a main-memory database, which uses priority and criticality based scheduling and optimistic concurrency control.

At the same time, commercial "real-time" database system products have started to appear in the marked such as Eaglespeed-RTDB [20], Clustra [44], Timesten [19], Empress [], eXtremeDB [76], and SolidDB [78]. Although these products may not be considered true RTDBS from the view-point of many researchers in the RTDB community since most of them only have very limited real-time features, they represent a significant step forward the success of RTDB. Most of these products use main-memory database techniques to achieve a better real-time performance. Additionally, some of them include features for real-time transaction management.

Several research angles emerge from real-time databases: real-time concurrency control (e.g. [57, 23, 66]), buffer management (e.g. [21]), disk scheduling (e.g. [49, 18]), system failure and recovery (e.g. [90, 89]), overload management (e.g. [22, 31, 28]), sensor data [48], security (e.g. [46, 30, 93]), and distributed real-time database systems (e.g. [37, 64, 63, 62, 71, 29, 72, 104]).

While developing RTDB systems that provide the required timeliness of the data and transactions, there are several issues that must be considered. Below is a list of some of the issues that have been the subject of research in this field [84].

- *Data, transactions and system characteristics*: A RTDB must maintain not only the logical consistency of the data and transactions, it must also maintain transaction timing properties as well as temporal consistency of the data. These issues will be presented in more detail in Section 2.

- *Scheduling and transaction processing*: Scheduling and transaction processing issues that consider data and transaction features have been a major part of the research that have been performed in the field of RTDB. These issues will be discussed in more detail in Section 3.

- *Managing I/O and Buffers*: While the scheduling of CPU and data resources have been studied extensively, other resources like disk I/O and buffers has begun only recently [21, 84]. Work presented on [94, 16, 4, 18, 52, 49] has shown that transaction priorities must be considered in the buffer management and Disk I/O.

- *Concurrency control*: Concurrency control has been one of the main research areas in RTDBSs. This issue will be discussed in Section 4.

- *Distribution*: Many applications that require RTDB are not located on a single computer. Instead, they are distributed and may require that real-time data be distributed as well. Issues involved with distributing data include deadline assignment [71, 104, 48, 108], distributed database architectures [10], distributed resource management [38] data replication [72], replication consistency [109] distributed transaction processing [102, 50, 62, 64], and distributed concurrency control [86, 63].

- *Quality of service and quality of data*: Maintaining logical consistency of the database and the temporal consistency of the data is hard [47]. Therefore, there must be a trade-off made to decide with is more important [9, 103].

2 Real-Time Database Model

A real-time system consists of a *controlling system* and a *controlled system* [82]. The controlled system is the environment with which the computer and its software interacts. The controlling system interacts with its environment based on the data read from various sensors, e.g., distance and speed sensors. It is essential that the state of the environment is consistent with the actual state of the environment to a high degree of accuracy. Otherwise, the actions of the controlling systems may be disastrous. Hence, timely monitoring of the environment as well as timely processing of the information from the environment is necessary. In many cases the read data is processed to derive new data [25].

This section discusses the characteristics of data and characteristics of transactions in real-time database systems.

2.1 Data and Consistency

In addition to the timing constraints that originate from the need to continuously track the environment, timing correctness requirements in a real-time database system also surface because of the need to make data available to the controlling system for its decision-making activities [26]. The need to maintain consistency between the actual state of the environment and the state as reflected by the contents of the database leads to the notion of *temporal consistency*. Temporal consistency has two components [91]:

- *Absolute consistency*: Data is only valid between absolute points in time. This is due to the need to keep the database consistent with the environment.
- *Relative consistency*: Different data items that are used to derive new data must be temporally consistent with each other. This requires that a set of data items used to derive a new data item form a *relative consistency set* R.

Data item d is *temporally consistent* if and only if d is absolutely consistent and relatively consistent [82]. Every data item in the real-time database consists of the current state of the object (i.e. current value stored in that data item), and two timestamps. These timestamps represent the time when this data item was last accessed by the committed transaction. These timestamps are used in the concurrency control method to ensure that the transaction reads only from committed transactions and writes after the latest committed write. Formally,

Definition 2.1 *a Data item in the real-time database is denoted by* $d : (value, RTS, WTS, avi)$*, where* d_{value} *denotes the current state of* d*,* d_{RTS} *denotes when the last committed transaction has read the current state of* d*,* d_{WTS} *denotes when the last committed transaction has written* d*, i.e., when the observation relating to* d *was made, and* d_{avi} *denotes* d*'s absolute validity interval, i.e., the length of the time interval following* R_{WTS} *during which* d *is considered to have absolute validity.*

A set of data items used to derive a new data item form a relative consistency set R. Each such set R is associated with a *relative validity interval*. Assume that $d \in R$. d has a correct state if and only if [82]:

1. d_{value} is logically consistent, i.e., satisfies all integrity constraints.

2. d is temporally consistent:

   - Data item $d \in R$ is absolutely consistent if and only if $(current_time - d_{observationtime}) \leq d_{absolutevalidityinterval}$.
   - Data items are relatively consistent if and only if $\forall d^{'} \in R|d_{timestamp} - d^{'}_{timestamp}| \leq R_{relativevalidityinterval}$.

In this section we do not consider temporal data or temporal constraints. A good book on temporal databases can be found in [99]. A discussion on integration of temporal and real-time database systems can be found from [85]. Finally, temporal consistency maintenance is discussed in [107, 110].

2.2 Transactions in Real-Time Database System

In this section, transactions are characterized along three dimensions; the manner in which data is used by transactions, the nature of time constraints, and the significance of executing a transaction by its deadline, or more precisely, the consequence of missing specified time constraints [1].

To reason about transactions and about the correctness of the management algorithms, it is necessary to define the concept formally. For the simplicity of the exposition, it is assumed that each transaction reads and writes a data item at most once. From now on the abbreviations *r, w, a* and *c* are used for the read, write, abort, and commit operations, respectively.

Definition 2.2 *A transaction T_i is partial order with an ordering relation $\prec_i$ where [11]:*

1. $T_i \subseteq \{r_i[x], w_i[x] \mid$ *x is a data item* $\} \cup \{a_i, c_i\}$*;*

2. $a_i \in T_i$ *if and only if* $c_i \notin T_i$*;*

3. *if* t *is* c_i *or* a_i*, for any other operation* $p \in T_i$*,* $p \prec_i t$*; and*

4. *if* $r_i[x], w_i[x] \in T_i$*, then either* $r_i[x] \prec_i w_i[x]$ *or* $w_i[x] \prec_i r_i[x]$.

Informally, (1) a transaction is a subset of read, write and abort or commit operations. (2) If the transaction executes an abort operation, then the transaction is not executing a commit operation. (3) if a certain operation t is abort or commit then the ordering relation defines that for all other operations precede operation t in the execution of the transaction. (4) if both read and write operation are executed to the same data item, then the ordering relation defines the order between these operations.

A *real-time transaction* is a transaction with additional real-time attributes. We have added additional attributes for a *real-time transaction*. These attributes are used by the real-time scheduling algorithm and concurrency control method. Additional attributes are the following [51]:

- Timing constraints - e.g. deadline is a timing constraint associated with the transaction.
- Criticalness - It measures how critical it is that a transaction meets its timing constraints. Different transactions have different criticalness. Furthermore, criticalness is a different concept from deadline because a transaction may have a very tight deadline but missing it may not cause great harm to the system.
- Value function - Value function is related to a transaction's criticalness. It measures how valuable it is to complete the transaction at some point in time after the transaction arrives.
- Resource requirements - Indicates the number of I/O operations to be executed, expected CPU usage, etc.
- Expected execution time. Generally very hard to predict but can be based on estimate or experimentally measured value of worst case execution time.
- Data requirements - Read sets and write sets of transactions.
- Periodicity - If a transaction is periodic, what its period is.
- Time of occurrence of events - In witch point in time a transaction issues a read or write request.
- Other semantics - Transaction type (read-only, write-only, etc.).

Based on the values of the above attributes, the availability of the information, and other semantics of the transactions, a real-time transaction can be characterized as follows [54]:

- Implication of missing deadline: *hard*, *critical*, or *soft (firm)* real-time
- Arrival pattern: *periodic*, *sporadic*, or *aperiodic*.
- Data access pattern: predefined (*read-only*, *write-only*, or *update*) or *random*
- Data requirement: *known* or *unknown*
- Runtime requirement i.e. pure processor or data access time: *known* or *unknown*
- Accessed data type: *continuous*, *discrete*, or both

The real-time database system apply all three types of transactions discussed in the database literature [8]:

- *Write-only transactions* obtain the state of the environment and write into the database.
- *Update transactions* derive a new data item and store it in the database.
- *Read-only transactions* read data from the database and transmit that data or derived actions based on that data to the controlling system.

The above classification can be used to tailor the appropriate concurrency control methods [51]. Some transaction-time constraints come from temporal consistency requirements and some come from requirements imposed on system reaction time. The former typically take the form of periodicity requirements. Transactions can also be distinguished based on the effect of missing a transaction's deadline.

Transaction processing and concurrency control in a real-time database system should be based on priority and criticalness of the transactions [97]. Traditional methods for transaction processing and concurrency control used in a real-time environment would cause some unwanted behavior. Below the four typified problems are characterized and priority is used to denote either scheduling priority or criticality of the transaction:

- **wasted restart:** A wasted restart occurs if a higher priority transaction aborts a lower priority transaction and later the higher priority transaction is discarded when it misses its deadline.
- **wasted wait:** A wasted wait occurs if a lower priority transaction waits for the commit of a higher priority transaction and later the higher priority transaction is discarded when it misses its deadline.
- **wasted execution:** A wasted execution occurs when a lower priority transaction in the validation phase is restarted due to a conflicting higher priority transaction which has not finished yet.
- **unnecessary restart:** An unnecessary restart occurs when a transaction in the validation phase is restarted even when history would be serializable.

Traditional two-phase locking methods suffer from the problem of wasted restart and wasted wait. Optimistic methods suffer the problems of wasted execution and unnecessary restart [67].

3 Transaction Processing in the Real-Time Database System

This section presents several characteristics of transaction and query processing. Transactions and queries have time constraints attached to them and there are different effects of not satisfying those constraints [82]. A key issue in transaction processing is *predictability* [95]. If a real-time transaction misses its deadline, it can have catastrophic consequences. Therefore, it is necessary to able to *predict* that such transactions will complete before their deadlines. This prediction will be possible only if the worst-case execution time of a transaction and the data and resource needs of the transaction is known.

In a database system, several sources of unpredictability exist [82]:

- Dependence of the transaction's execution sequence on data values
- Data and resource conflicts
- Dynamic paging and I/O
- Transactions abort and the resulting rollbacks and restarts

- Communication delays and site failures on distributed databases

Because a transaction's execution path can depend on the values of the data items it accessed, it may not be possible to predict the worst-case execution time of the transaction. Similarly, it is better to avoid using unbounded loops and recursive or dynamically created data structures in real-time transactions. Dynamic paging and I/O unpredictability can be solved by using main memory databases [2]. Additionally, I/O unpredictability can be decreased using deadlines and priority-driven I/O controllers (e.g. [4, 94]).

Transaction rollbacks also reduce predictability. Therefore, it is advisable to allow a transaction to write only to its own memory area and after the transaction is guaranteed to commit write the transaction's changes to the database [67].

3.1 Scheduling Real-Time Transactions

A *transaction scheduling policy* defines how priorities are assigned to individual transactions [1]. The goal of transaction scheduling is that as many transactions as possible will meet their deadlines. Numerous transaction scheduling policies are defined in the literature. Only a few examples are quoted here.

Transactions in a real-time database can often be expressed as *tasks* in a real-time system [1]. Scheduling involves the allocation of resources and time to tasks in such a way that certain performance requirements are met. A typical real-time system consists of several tasks, which must be executed concurrently. Each task has a *value*, which is gained to the system if a computation finishes in a specific time. Each task also has a *deadline*, which indicates a time limit, when a result of the computing becomes useless.

In this section, the terms *hard, soft*, and *firm* are used to categorize the transactions [1]. This categorization tells the *value* imparted to the system when a transaction meets its deadline. In systems which use priority-driven scheduling algorithms, value and deadline are used to derive the priority [33, 100].

A characteristic of most real-time scheduling algorithms is the use of priority based scheduling [1]. Here transactions are assigned 'priorities', which are implicit or explicit functions of their deadlines or *criticality* or both. The criticality of a transaction is an indication of its level of importance. However, these two requirements sometimes conflict with each other. That is, transactions with very short deadlines might not be very critical, and vice versa [12]. Therefore, the criticality of the transactions is used in place of the deadline in choosing the appropriate value to priority. This avoid the dilemma of priority scheduling, yet

integrates criticality and deadline so that not only the more critical transactions meet their deadlines. The overall goal is to maximize the net worth of the executed transactions to the system.

Whereas arbitrary types of value functions can be associated with transactions [14, 36, 45], the following simple functions occur more often (see also Figure 1):

- *Hard* deadline transactions are those which may result in a catastrophe if the deadline is missed. One can say that a large negative *value* is imparted to the system if a hard deadline is missed. These are typically safety-critical activities, such as those that respond to life or environment-threatening emergency situations (e.g. [75, 60]).

- *Soft* deadline transactions have some value even after their deadlines. Typically, the value drops to zero at a certain point past the deadline (e.g. [42, 50]).

- *Firm* deadline transactions impart no value to the system once their deadlines expire, i.e., the value drops to zero at the deadline (e.g. [22, 36]).

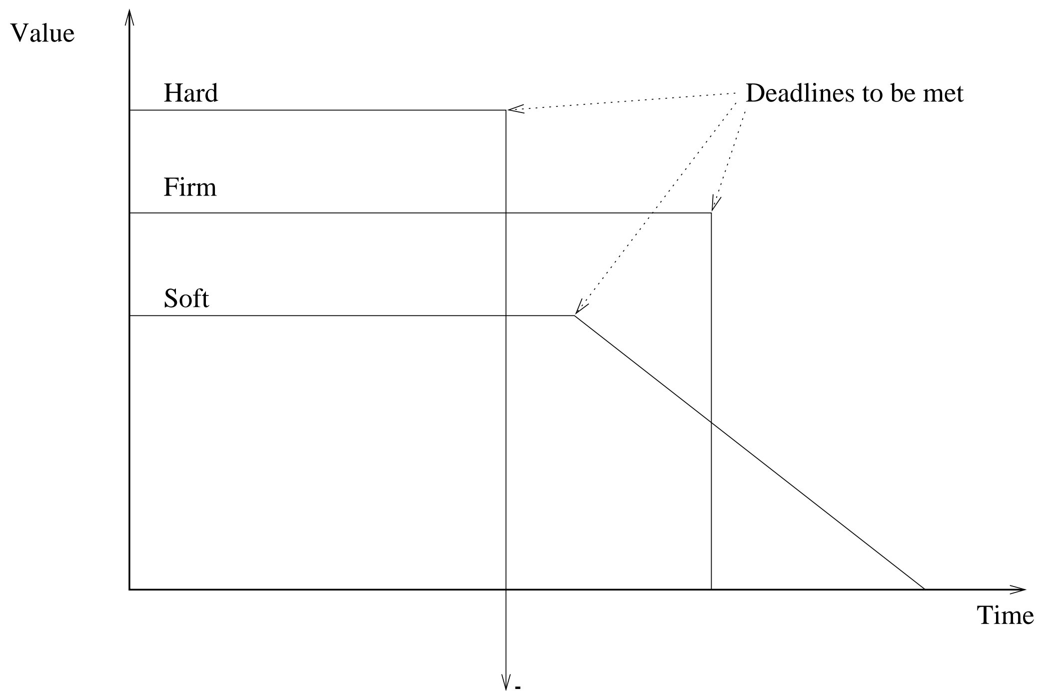

Figure 1: The deadline types.

3.2 Scheduling Paradigms

Several scheduling paradigms emerge, depending on a) whether a system performs a schedulability analysis, b) if it does, whether it is done statically or dynamically, and c) whether the result of the analysis itself produces a schedule or plan according to which tasks are dispatched at run-time. Based on this the following classes of algorithms can be identified [83]:

- Static table-driven approaches: These perform a static schedulability analysis and the resulting schedule is used at run time to decide when a task must begin execution.
- Static priority-driven preemptive approaches: These perform a static schedulability analysis but unlike the previous approach, no explicit schedule is constructed. At run time, tasks are executed using a highest priority first policy.
- Dynamic planning-based approaches: The feasibility is checked at run time, i.e., a dynamically arriving task is accepted for execution only if it is found feasible.
- Dynamic best effort approaches: The system tries to do its best to meet deadlines.

In the *earliest deadline first* (EDF) [75] policy, the transaction with the earliest deadline has the highest priority. Other transactions will receive their priorities in descending deadline order. In the *least slack first* (LSF) [1] policy, the transaction with the shortest slack time is executed first. The slack time is an estimate of how long the execution of a transaction can be delayed and still meet its deadline. In the *highest value* (HV) [33] policy, transaction priorities are assigned according to the transaction value attribute. A survey of transaction scheduling policies can be found in [1].

3.3 Priority Inversion

In a real-time database environment resource control may interfere with CPU scheduling [3]. When blocking is used to resolve a resource allocation such as in 2PL [24], a *priority inversion* [2] event can occur if a higher priority transaction gets blocked by a lower priority transaction.

Figure 2 illustrates an execution sequence, where a priority inversion occurs. A task T_3 executes and reserves a resource. A higher priority task T_1 pre-empts task T_3 and tries to allocate a resource reserved by task T_3. Then, task T_2 becomes eligible and blocks T_3. Because T_3 cannot be executed the resource remains reserved suppressing T_1. Thus, T_1 misses its deadline due to the resource conflict.

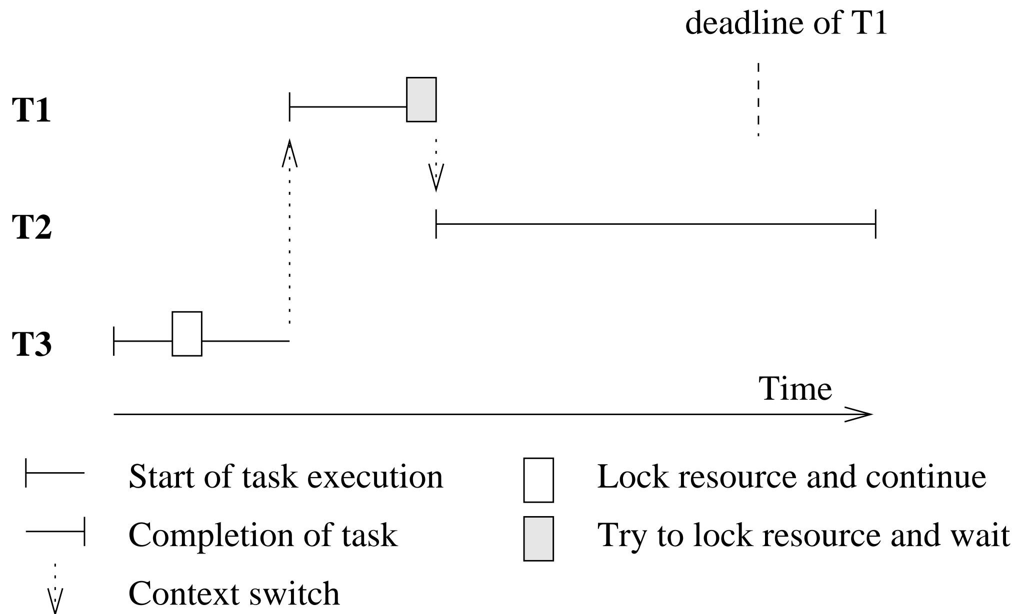

Figure 2: Priority inversion example.

In [87], a *priority inheritance* approach was proposed to address this problem. The basic idea of priority inheritance protocols is that when a task blocks one or more higher priority task the lower priority transaction inherits the highest priority among conflicting transactions.

Figure 3 illustrates, how a priority inversion problem presented in figure 2 can be solved with the priority inheritance protocol. Again, task T_3 executes and reserves a resource, and a higher priority task T_1 tries to allocate the same resource. In the priority inheritance protocol task T_3 inherits the priority of T_1 and executes. Thus, task T_2 cannot pre-empt task T_3. When T_3 releases the resource, the priority of T_3 returns to the original level. Now T_1 can acquire the resource and complete before its deadline.

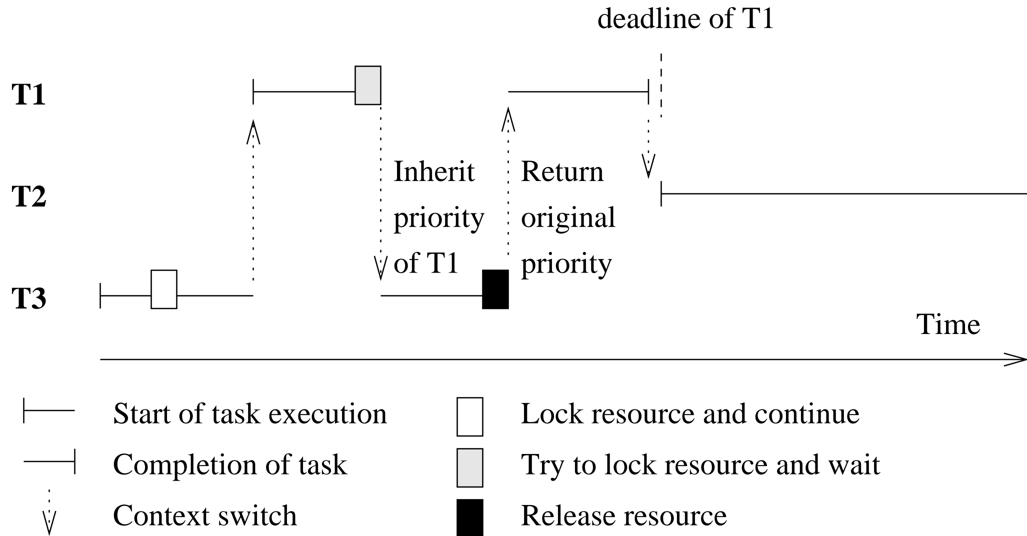

Figure 3: Priority inheritance example.

4 Concurrency Control in Real-Time Databases

A *Real-Time Database System* (RTDBS) processes transactions with timing constraints such as deadlines [82]. Its primary performance criterion is timeliness, not average response time or throughput. The scheduling of transactions is driven by priority order. Given these challenges, considerable research has recently been devoted to designing concurrency control methods for RTDBSs and to evaluating their performance (e.g. [2, 35, 40, 43, 61, 58] Most of these methods are based on one of the two basic concurrency control mechanisms: *locking* [24] or *optimistic concurrency control* (OCC) [56].

In real-time database systems transactions are scheduled according to their timing constraints. Task scheduler assigns a priority for a task based on its timing constraints or criticality or both [82]. Therefore, high priority transactions are executed before lower priority transactions. This is true only if a high priority transaction has some database operation ready for execution. If no operation from a higher priority transaction is ready for execution, then an operation from a lower priority transaction is allowed to execute its database operation. Therefore, the operation of the higher priority transaction may conflict with the already executed operation of the lower priority transaction. In non-pre-emptive methods a higher priority transaction must wait for the release of the resource. This is the

priority inversion problem presented earlier. Therefore, data conflicts in concurrency control should also be based on transaction priorities or criticalness or both. Hence, numerous traditional concurrency control methods have been extended to the real-time database systems.

There are many ways in which the schedulers can be classified [11]. One obvious classification criterion is the mode of database distribution. Some schedulers that have been proposed require a fully replicated database, while others can operate on partially replicated or partitioned databases. The schedulers can also be classified according to network topology. The most common classification criterion however is the synchronization primitive, i.e. those methods that are based on mutually exclusive access to shared data and those that attempt to order the execution of the transactions according to a set of rules [80]. There are two possible views: the pessimistic view that many transactions will conflict with each other, or the optimistic view that not too many transactions will conflict with each other. Pessimistic methods synchronize the concurrent execution of transactions early in their execution and optimistic methods delay the synchronization of transactions until their terminations [105]. Therefore, the basic classification is as follows:

- Pessimistic Methods
  - Timestamp Ordering Methods [79, 101]
  - Serialization Graph Testing [11]
  - Locking Methods [88, 41, 7, 62, 39, 63]
- Optimistic Methods
  - Backward Validation Methods [32]
  - Forward Validation Methods [34, 58, 59, 111, 68]
  - Serialization Graph Testing [11, 79, 70]
  - Hybrid Methods [69]
- Hybrid Methods [40, 92, 112, 27, 61]

In the locking-based methods, the synchronization of transactions is acquired by using physical or logical locks on some portion or granule of the database. The timestamp ordering method involves organizing the execution order of transactions so that they maintain mutual and internal consistency. This ordering is

maintained by assigning timestamps to both the transactions and the data that are stored in the database [80].

The state of a conventional non-versioning database represent the state of a system at a single moment of time. Although the contents of the database change as new information is added, these changes are viewed as modification to the state. The current contents of the database may be viewed as a snapshot of the system. Additionally, conventional DBSs provide no guarantee of transaction completion times.

In the following sections recent related work on locking and optimistic methods for real-time databases are presented.

4.1 Locking Methods in Real-Time Databases

In this section we present some well known pessimistic concurrency control methods. Most of these methods are based on 2PL.

2PL High Priority In the 2PL-HP (2PL High Priority) concurrency control method [2, 5, 41] conflicts are resolved in favor of the higher priority transactions. If the priority of the lock requester is higher than the priority of the lock holder, the lock holder is aborted, rolled back and restarted. The lock is granted to this requester and the requester can continue its execution. If the priority of the lock requester is lower than that of the lock holder, the requesting transaction blocks to wait for the lock holder to finish and release its locks. High Priority concurrency control may lead to the cascading blocking problem, a deadlock situation, and priority inversion.

2PL Wait Promote In 2PL-WP (2PL Wait Promote) [3, 41] the analysis of concurrency control method is developed from [2]. The mechanism presented uses shared and exclusive locks. Shared locks permit multiple concurrent readers. A new definition is made - the priority of a data object, which is defined to be the highest priority of all the transactions holding a lock on the data object. If the data object is not locked, its priority is undefined.

A transaction can join in the read group of an object if and only if its priority is higher than the maximum priority of all transactions in the write group of an object. Thus, conflicts arise from incompatibility of locking modes as usual. Particular attention is given to conflicts that lead to priority inversions. A priority inversion occurs when a transaction of high priority requests and blocks for an

object which has lesser priority. This means that all the lock holders have lesser priority than the requesting transaction. This same method is also called 2PL-PI (2PL Priority Inheritance) [41].

2PL Conditional Priority Inheritance Sometimes High Priority may be too strict policy. If the lock holding transaction T_h can finish in the time that the lock requesting transaction T_r can afford to wait, that is within the slack time of T_r, and let T_h proceed to execution and T_r wait for the completion of T_h. This policy is called 2PL-CR (2PL Conditional Restart) or 2PL-CPI (2PL Conditional Priority Inheritance) [41].

Priority Ceiling Protocol [86, 87] the focus is to minimize the duration of blocking to at most one lower priority task and prevent the formation of deadlocks. A real-time database can often be decomposed into sets of database objects that can be modeled as atomic data sets. For example, two radar stations track an aircraft representing the local view in data objects O_1 and O_2. These objects might include e.g. the current location, velocity, etc. Each of these objects forms an atomic data set, because the consistency constraints can be checked and validated locally. The notion of atomic data sets is especially useful for tracking multiple targets.

A simple locking method for elementary transactions is the two-phase locking method; a transaction cannot release any lock on any atomic data set unless it has obtained all the locks on that atomic data set. Once it has released its locks it cannot obtain new locks on the same atomic data set, however, it can obtain new locks on different data sets. The theory of modular concurrency control permits an elementary transaction to hold locks across atomic data sets. This increases the duration of locking and decreases preemptibility. In this study transactions do not hold locks across atomic data sets.

Priority Ceiling Protocol minimizes the duration of blocking to at most one elementary lower priority task and prevents the formation of deadlocks [86, 87]. The idea is that when a new higher priority transaction preempts a running transaction its priority must exceed the priorities of all preempted transactions, taking the priority inheritance protocol into consideration. If this condition cannot be met, the new transaction is suspended and the blocking transaction inherits the priority of the highest transaction it blocks.

The priority ceiling of a data object is the priority of the highest priority transaction that may lock this object [86, 87]. A new transaction can preempt a lock-

holding transaction if and only if its priority is higher than the priority ceilings of all the data objects locked by the lock-holding transaction. If this condition is not satisfied, the new transaction will wait and the lock-holding transaction inherits the priority of the highest transaction that it blocks. The lock-holder continues its execution, and when it releases the locks, its original priority is resumed. All blocked transactions are awaked, and the one with the highest priority will start its execution.

The fact that the priority of the new lock-requesting transaction must be higher than the priority ceiling of all the data objects that it accesses, prevents the formation of a potential deadlock. The fact that the lock-requesting transaction is blocked only at most the execution time of one lower priority transaction guarantees, the formation of blocking chains is not possible [86, 87].

Read/Write Priority Ceiling The Priority Ceiling Protocol is further advanced in [88], where the Read/Write Priority Ceiling Protocol is introduced. It contains two basic ideas. The first idea is the notion of priority inheritance. The second idea is a total priority ordering of active transactions. A transaction is said to be active if it has started but not completed its execution. Thus, a transaction can execute or wait caused by a preemption in the middle of its execution. Total priority ordering requires that each active transaction execute at a higher priority level than the active lower priority transaction, taking priority inheritance and read/write semantics into consideration.

4.2 Optimistic Methods in Real-Time Databases

Optimistic Concurrency Control (OCC) [32, 56], is based on the assumption that conflict is rare, and that it is more efficient to allow transactions to proceed without delays. When a transaction desires to commit, a check is performed to determine whether a conflict has occurred. Therefore, there are three phases to an optimistic concurrency control method:

- *Read phase*: The transaction reads the values of all data items it needs from the database and stores them in local variables. Concurrency control scheduler stores identity of these data items to a read set. However, writes are applied only to local copies of the data items kept in the transaction workspace. Concurrency control scheduler stores identity of all written data items to a write set.

- *Validation phase*: The validation phase ensures that all the committed transactions have executed in a serializable fashion. For a read-only transaction, this consists of checking that the data values read are still the current values for the corresponding data items. For a transaction that has writes, the validation consists of determining whether the current transaction has executed serializable way.

- *Write phase*: This follows the successful validation phase for transactions including write operations. During the write phase, all changes made by the transaction are permanently stored into the database.

In the following we introduce some well known optimistic methods for real-time database systems.

Broadcast Commit For RTDBSs, a variant of the classical concurrency control method is needed. In Broadcast Commit, OPT-BC [35], when a transaction commits, it notifies other running transactions that conflict with it. These transactions are restarted immediately. There is no need to check a conflict with committed transactions since the committing transaction would have been restarted in the event of a conflict. Therefore, a validating transaction is always guaranteed to commit. The broadcast commit method detects the conflicts earlier than the conventional concurrency control mechanism, resulting in earlier restarts, which increases the possibility of meeting the transaction deadlines [35].

The main reason for the good performance of locking in a conventional DBMS is that the blocking-based conflict resolution policy results in conservation of resources, while the optimistic method with its restart-based conflict resolution policy wastes more resources [35]. But in a RTDBS environment, where conflict resolution is based on transaction priorities, the OPT-BC policy effectively prevents the execution of a low priority transaction that conflicts with a higher priority transaction, thus decreasing the possibility of further conflicts and the waste of resources is reduced. Conversely, 2PL-HP loses some of the basic 2PL blocking factor due to the partially restart-based nature of the High Priority scheme.

The delayed conflict resolution of optimistic methods aids in making better decisions since more information about the conflicting transactions is available at this stage [35]. Compared to 2PL-HP, a transaction could be restarted by, or wait for, another transaction which is later discarded. Such restarts or waits are useless and cause performance degradation. OPT-BC guarantees the commit and thus the completion of each transaction that reaches the validation stage. Only validating

transactions can cause the restart of other transactions, thus, all restarts generated by the OPT-BC method are useful.

First of all, OPT-BC has a bias against long transactions (i.e. long transactions are more likely to be aborted if there is conflicts), like in the conventional optimistic methods [35]. Second, as the priority information is not used in the conflict resolution, a committing lower priority transaction can restart a higher priority transaction very close to its validation stage, which will cause missing the deadline of the restarted higher priority transaction [34].

OPT-SACRIFICE In the OPT-SACRIFICE [34] method, when a transaction reaches its validation phase, it checks for conflicts with other concurrently running transactions. If conflicts are detected and at least one of the conflicting transactions has a higher priority, then the validating transaction is restarted, i.e. sacrificed in favor of the higher priority transaction. Although this method prefers high priority transactions, it has two potential problems. Firstly, if a higher priority transaction causes a lower priority transaction to be restarted, but fails in meeting its deadline, the restart was useless. This degrades the performance. Secondly, if priority fluctuations are allowed, there may be the mutual restarts problem between a pair of transactions (i.e. both transactions are aborted). These two drawbacks are analogous to those in the 2PL-HP method [34].

OPT-WAIT and WAIT-X When a transaction reaches its validation phase, it checks if any of the concurrently running other transactions have a higher priority. In the OPT-WAIT [34] case the validating transaction is made to wait, giving the higher priority transactions a chance to make their deadlines first. While a transaction is waiting, it is possible that it will be restarted due to the commit of one of the higher priority transactions. Note that the waiting transaction does not necessarily have to be restarted. Under the broadcast commit scheme a validating transaction is said to conflict with another transaction, if the intersection of the write set of the validating transaction and the read set of the conflicting transaction is not empty. This result does not imply that the intersection of the write set of the conflicting transaction and the read set of the validating transaction is not empty either [34].

The WAIT-50 [34] method is an extension of the OPT-WAIT - it contains the priority wait mechanism from the OPT-WAIT method and a wait control mechanism. This mechanism monitors transaction conflict states and dynamically decides when and for how long a low priority transaction should be made to wait for

the higher priority transactions. In WAIT-50, a simple 50 percent rule is used - a validating transaction is made to wait while half or more of its conflict set is composed of transactions with higher priority. The aim of the wait control mechanism is to detect when the beneficial effects of waiting are outweighed by its drawbacks [34].

We can view OPT-BC, OPT-WAIT and WAIT-50 as being special cases of a general WAIT-X method, where X is the cutoff percentage of the conflict set composed of higher priority transactions. For these methods X takes the values infinite, 0 and 50 respectively.

4.3 Validation Methods

The validation phase ensures that all the committed transactions have executed in a serializable fashion [56]. Most of the validation methods use the following principles to ensure serializability. If a transaction T_i is before transaction T_j in the serialization graph(i.e. $T_i \prec T_j$), the following two conditions must be satisfied [67]:

1. No overwriting. The writes of T_i should not overwrite the writes of T_j.

2. No read dependency. The writes of T_j should not affect the read phase of T_i.

Generally, condition 1 is automatically ensured in most optimistic methods because I/O operations in the write phase are required to be done sequentially in the critical section [67]. Thus most validation schemes consider only condition 2. During the write phase, all changes made by the transaction are permanently installed into the database. To design an efficient real-time optimistic concurrency control method, three issues have to be considered [67]:

1. which validation scheme should be used to detect conflicts between transactions;

2. how to minimize the number of transaction restarts; and

3. how to select a transaction or transactions to restart when a nonserializable execution is detected.

In *Backward Validation* [32], the validating transaction is checked for conflicts against (recently) committed transactions. Conflicts are detected by comparing the read set of the validating transaction and the write sets of the committed transactions. If the validating transaction has a data conflict with any committed transactions, it will be restarted. The classical optimistic method in [56] is based on this validation process.

In *Forward Validation* [32], the validating transaction is checked for conflicts against other active transactions. Data conflicts are detected by comparing the write set of the validating transaction and the read set of the active transactions. If an active transaction has read an object that has been concurrently written by the validating transaction, the values of the object used by the transactions are not consistent. Such a data conflict can be resolved by restarting either the validating transaction or the conflicting transactions in the read phase. Optimistic methods based on this validation process are studied in [32]. Most of the proposed optimistic methods are based on Forward Validation.

Forward Validation is preferable for the real-time database systems because Forward Validation provides flexibility for conflict resolution [32]. Either the validating transaction or the conflicting active transactions may be chosen to restart. In addition to this flexibility, Forward Validation has the advantage of early detection and resolution of data conflicts. In recent years, the use of optimistic methods for concurrency control in real-time databases has received more and more attention. Different real-time optimistic methods have been proposed.

Forward Validation (OCC-FV) [32] is based on the assumption that the serialization order of transactions is determined by the arriving order of the transactions at the validation phase. Thus the validating transaction, if not restarted, always precedes concurrently running active transactions in the serialization order. A validation process based on this assumption can cause restarts that are not necessary to ensure data consistency. These restarts should be avoided.

The major performance problem with optimistic concurrency control methods is the late restart [67]. Sometimes the validation process using the read sets and write sets erroneously concludes that a nonserializable execution has occurred, even though it has not done so in actual execution [92] (see Example 1). Therefore, one important mechanism to improve the performance of an optimistic concurrency control method is to reduce the number of restarted transactions.

Example 1 *Consider the following transactions T_1, T_2 and history H_1:*
T_1: $r_1[x]c_1$
T_2: $w_2[x]c_2$

H_1: $r_1[x]w_2[x]c_2$

Based on the OCC-FV method [32], T_1 *has to be restarted. However, this is not necessary, because when* T_1 *is allowed to commit such as:*

$H_2 : r_1[x]w_2[x]c_2\mathbf{c_1}$,

then the schedule of H_2 *is equivalent to the serialization order* $T_1 \rightarrow T_2$ *as the actual write of* T_2 *is performed after its validation and after the read of* T_1*. There is no cycle in their serialization graph and* H_2 *is serializable.*

One way to reduce the number of transaction restarts is to dynamically adjust the serialization order of the conflicting transactions [67]. Such methods are called *dynamic adjustment of the serialization order* [67]. When data conflicts between the validating transaction and active transactions are detected in the validation phase, there is no need to restart conflicting active transactions immediately. Instead, a serialization order of these transactions can be dynamically defined.

Definition 4.1 *Suppose there is a validating transaction* T_v *and a set of active transactions* $T_j (j = 1, 2, ..., n)$*. There are three possible types of data conflicts which can cause a serialization order between* T_v *and* T_j *[67, 73, 92]:*

1. $RS(T_v) \cap WS(T_j) \neq \emptyset$ *(read-write conflict)*
   A read-write conflict between T_v *and* T_j *can be resolved by adjusting the serialization order between* T_v *and* T_j *as* $T_v \rightarrow T_j$ *so that the read of* T_v *cannot be affected by* T_j*'s write. This type of serialization adjustment is called forward ordering or forward adjustment.*

2. $WS(T_v) \cap RS(T_j) \neq \emptyset$ *(write-read conflict)*
   A write-read conflict between T_v *and* T_j *can be resolved by adjusting the serialization order between* T_v *and* T_j *as* $T_j \rightarrow T_v$*. It means that the read phase of* T_j *is placed before the write of* T_v*. This type of serialization adjustment is called backward ordering or backward adjustment.*

3. $WS(T_v) \cap WS(T_j) \neq \emptyset$ *(write-write conflict)*
   A write-write conflict between T_v *and* T_j *can be resolved by adjusting the serialization order between* T_v *and* T_j *as* $T_v \rightarrow T_j$ *such that the write of* T_v *cannot overwrite* T_j*'s write (forward ordering).*

OCC-TI The OCC-TI [67, 65] method resolves conflicts using the timestamp intervals of the transactions. Every transaction must be executed within a specific time slot. When an access conflict occurs, it is resolved using the read and write sets of the transaction together with the allocated time slot. Time slots are adjusted when a transaction commits.

OCC-DA OCC-DA [58] is based on the Forward Validation scheme [32]. The number of transaction restarts is reduced by using dynamic adjustment of the serialization order. This is supported with the use of a dynamic timestamp assignment scheme. Conflict checking is performed at the validation phase of a transaction. No adjustment of the timestamps is necessary in case of data conflicts in the read phase. In OCC-DA the serialization order of committed transactions may be different from their commit order.

OCC-DATI Optimistic Concurrency Control with Dynamic Adjustment using Timestamp Intervals (OCC-DATI) [74]. OCC-DATI is based on forward validation. The number of transaction restarts is reduced by dynamic adjustment of the serialization order which is supported by similar timestamp intervals as in OCC-TI. Unlike the OCC-TI method, all checking is performed at the validation phase of each transaction. There is no need to check for conflicts while a transaction is still in its read phase. As the conflict resolution between the transactions in OCC-DATI is delayed until a transaction is near completion, there will be more information available for making the choice in resolving the conflict. OCC-DATI also has a new final timestamp selection method compared to OCC-TI.

5 Summary

The field of real-time database research has evolved greatly over the relatively short time of its existence. The future of RTDB research is dependent of continues progress of this evolution. Research on this field should continue to pursue state-of-the-art applications and apply both existing techniques to them as well as develop a new ones when needed.

References

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions. *ACM SIGMOD Record*, 17(1):71–81, March 1988.

[2] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In *Proceedings of the 14th VLDB Conference*, pages 1–12, Los Angeles, California, USA, 1988. Morgan Kaufmann.

[3] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions with disk resident data. In *Proceedings of the 15th VLDB Conference*, pages 385–396, Amsterdam, The Netherlands, 1989. Morgan Kaufmann.

[4] R. Abbott and H. Garcia-Molina. Scheduling I/O requests with deadlines: A performance evaluation. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 113–124. IEEE Computer Society Press, 1990.

[5] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database Systems*, 17(3):513–560, September 1992.

[6] Brad Adelberg, Ben Kao, and Hector Garcia-Molina. Overview of the stanford real-time information processor (strip). *SIGMOD Record*, 25(1):34–37, 1996.

[7] D. Agrawal, A. E. Abbadi, and R. Jeffers. Using delayed commitment in locking protocols for real-time databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 104–113. ACM Press, 1992.

[8] I. Ahn. Database issues in telecommunications network management. *ACM SIGMOD Record*, 23(2):37–43, June 1994.

[9] Mehdi Amirijoo, Jörgen Hansson, and Sang H. Son. Specification and management of qos in real-time databases supporting imprecise computations. *IEEE Trans. Computers*, 55(3):304–319, 2006.

[10] Sten Andler, Jörgen Hansson, Joakim Eriksson, Jonas Mellin, Mikael Berndtsson, and Bengt Eftring. Deeds towards a distributed and active real-time database system. *SIGMOD Record*, 25(1):38–40, 1996.

[11] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[12] S. R. Biyabani, J. A. Stankovic, and K. Ramamritham. The integration of deadline and criticalness in hard real-time scheduling. In *Proceedings of the 8th IEEE Real-Time Systems Symposium*, pages 487–507, Huntsville, Alabama, USA, 1988. IEEE Computer Society Press.

[13] A. Buchmann. *Real Time Database Systems*. Idea Group, 2002.

[14] A. P. Buchmann, D. R. McCarthy, M. Hsu, and U. Dayal. Time-critical database scheduling: A framework for integrating real-time scheduling and concurrency control. In *Proceedings of the 5th International Conference on Data Engineering*, pages 470–480, Los Angeles, California, USA, 1989. IEEE Computer Society Press.

[15] Alejandro P. Buchmann, Holger Branding, Thomas Kudrass, and Jürgen Zimmermann. Reach: a real-time, active and heterogeneous mediator system. *IEEE Data Eng. Bull.*, 15(1-4):44–47, 1992.

[16] Michael J. Carey, Rajiv Jauhari, and Miron Livny. Priority in dbms resource scheduling. In *VLDB*, pages 397–410, 1989.

[17] S. Cha, B. Park, S. Lee, S. Song, J. Park, J. Lee, S. Park, D. Hur, and G. Kim. Object-oriented design of main-memory dbms for real-time applications. In *2nd Int. Workshop on Real-Time Computing Systems and Applications*, pages 109–115, Tokyo, Japan, October 1995.

[18] S. Chen, J. A. Stankovic, J. F. Kurose, and D. Towsley. Performance evaluation of two new disk scheduling algorithms for real-time systems. *The Journal of Real-Time Systems*, 3(3):307–336, September 1991.

[19] Oracle Corp. http://www.oracle.com/database/timesten.html.

[20] Lockheed Martin Corporation. http://www.lockheedmartin.com/products/ eaglespeedrealtimedatabasemanager/index.html.

[21] A. Datta and S. Mukherjee. Buffer management in real-time active database systems. In *Real-Time Database Systems - Architecture and Techniques*, pages 77–96. Kluwer Academic Publishers, 2001.

[22] A. Datta, S. Mukherjee, P. Konana, I. Viguier, and A. Bajaj. Multiclass transaction scheduling and overload management in firm real-time database systems. *Information Systems*, 21(1):29–54, March 1996.

[23] A. Datta, S. H. Son, and V. Kumar. Is a bird in the hand worth more than two in the bush? limitations of priority cognizance in conflict resolution for firm real-time database systems. *IEEE Transactions on Computers*, 49(5):482–502, May 2000.

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

[25] M. H. Graham. Issues in real-time data management. *The Journal of Real-Time Systems*, 4:185–202, 1992.

[26] M. H. Graham. How to get serializability for real-time transactions without having to pay for it. In *Proceedings of the 14th IEEE Real-time Systems Symposium*, pages 56–65, 1993.

[27] P. Graham and K. Barker. Effective optimistic concurrency control in multiversion object bases. *Lecture Notes in Computer Science*, 858:313–323, 1994.

[28] Thomas Gustafsson and Jörgen Hansson. Data freshness and overload handling in embedded systems. In *RTCSA*, pages 173–182, 2006.

[29] U. Halici and A. Dogac. An optimistic locking technique for concurrency control in distributed databases. *IEEE Transactions on Software Engineering*, 17(7):712–724, July 1991.

[30] H. Han, S. Park, and C. Park. A concurrency control protocol for read-only transactions in real-time secure database systems. In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications*, pages 458–462. IEEE Computer Society Press, 2000.

[31] J. Hansson and S. H. Son. Overload management in rtdbs. In *Real-Time Database Systems - Architecture and Techniques*, pages 125–140. Kluwer Academic Publishers, 2001.

[32] T. Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, 1984.

[33] J. Haritsa, M. Carey, and M. Livny. Value-based scheduling in real-time database systems. Tech. Rep. CS-TR-91-1024, University of Winconsin, Madison, 1991.

[34] J. R. Haritsa, M. J. Carey, and M. Livny. Dynamic real-time optimistic concurrency control. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 94–103. IEEE Computer Society Press, 1990.

[35] J. R. Haritsa, M. J. Carey, and M. Livny. On being optimistic about real-time constraints. In *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, pages 331–343. ACM Press, 1990.

[36] J. R. Haritsa, M. J. Carey, and M. Livny. Data access scheduling in firm real-time database systems. *The Journal of Real-Time Systems*, 4(2):203–241, June 1992.

[37] J. R. Haritsa, K. Ramamritham, and R. Gupta. Real-time commit processing. In *Real-Time Database Systems - Architecture and Techniques*, pages 227–244. Kluwer Academic Publishers, 2001.

[38] Tian He, John A. Stankovic, Michael Marley, Chenyang Lu, Ying Lu, Tarek F. Abdelzaher, Sang H. Son, and Gang Tao. Feedback control-based dynamic resource management in distributed real-time systems. *Journal of Systems and Software*, 80(7):997–1004, 2007.

[39] D. Hong, S. Chakravarthy, and T. Johnson. Locking based concurrency control for integrated real-time database systems. In *Proceedings of the First International Workshop on Real-Time Databases: Issues and Applications*, Newport Beach, California, March 7-8, 1996.

[40] J. Huang, J. A. Stankovic, K. Ramamritham, and D. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *Proceedings of the 17th VLDB Conference*, pages 35–46, Barcelona, Catalonia, Spain, September 1991. Morgan Kaufmann.

[41] J. Huang, J. A. Stankovic, K. Ramamritham, and D. Towsley. On using priority inheritance in real-time databases. In *Proceedings of the 12th IEEE*

Real-Time Systems Symposium, pages 210–221, San Antonio, Texas, USA, 1991. IEEE Computer Society Press.

[42] J. Huang, J. A. Stankovic, K. Ramamritham, D. Towsley, and B. Purimetla. Priority inheritance in soft real-time databases. *The Journal of Real-Time Systems*, 4(2):243–268, June 1992.

[43] J. Huang, J. A. Stankovic, D. Towsley, and K. Ramamritham. Experimental evaluation of real-time transaction processing. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pages 144–153, Santa Monica, California, USA, December 1989. IEEE Computer Society Press.

[44] Svein-Olaf Hvasshovd, Øystein Torbjørnsen, Svein Erik Bratsberg, and Per Holager. The ClustRa telecom database: High availability, high throughput, and real-time response. In *Proceedings of the 21th VLDB Conference*, pages 469–477, San Mateo, Calif., 1995. Morgan Kaufmann.

[45] E. D. Jensen, C. D. Locke, and Tokuda. H. A time-driven scheduling model for real-time systems. In *Proceedings of the 5th IEEE Real-Time Systems Symposium*, pages 112–122, San Diego, California, USA, 1985. IEEE Computer Society Press.

[46] B.-S. Jeong, D. Kim, and S. Lee. Optimistic secure real-time concurrency control using multiple data version. In *Lecture Notes in Computer Science*, volume 1985, 2001.

[47] Abhay Kumar Jha, Ming Xiong, and Krithi Ramamritham. Mutual consistency in real-time databases. In *RTSS*, pages 335–343, 2006.

[48] Kyoung-Don Kang, Sang H. Son, and John A. Stankovic. Managing deadline miss ratio and sensor data freshness in real-time databases. *IEEE Trans. Knowl. Data Eng.*, 16(10):1200–1216, 2004.

[49] B. Kao and R. Cheng. Disk scheduling. In *Real-Time Database Systems - Architecture and Techniques*, pages 97–108. Kluwer Academic Publishers, 2001.

[50] B. Kao and H. Garcia-Molina. Deadline assigment in a distributed soft real-time system. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 428–437, Pittsburgh, PA, USA, 1993. IEEE Computer Society Press.

[51] B. Kao and H. Garcia-Molina. An overview of real-time database systems. In S. H. Son, editor, *Advances in Real-Time Systems*, pages 463–486. Prentice Hall, 1995.

[52] W. Kim and J. Srivastava. Enhancing real-time dbms performance with multiversion data and priority based disk scheduling. In *Proceedings of the 12th IEEE Real-Time Systems Symposium*, pages 222–231, Los Alamitos, Calif., 1991. IEEE Computer Society Press.

[53] Young-Kuk Kim and S. H. Son. Developing a real-time database: The StarBase experience. In A. Bestavros, K. Lin, and S. Son, editors, *Real-Time Database Systems: Issues and Applications*, pages 305–324, Boston, Mass., 1997. Kluwer.

[54] Young-Kuk Kim and Sang H. Son. Predictability and consistency in real-time database systems. In *Real-Time Database Systems - Architecture and Techniques*, pages 509–531. Kluwer Academic Publishers, 1995.

[55] J. Kiviniemi, T. Niklander, P. Porkka, and K. Raatikainen. Transaction processing in the RODAIN real-time database system. In A. Bestavros and V. Fay-Wolfe, editors, *Real-Time Database and Information Systems*, pages 355–375, London, 1997. Kluwer Academic Publishers.

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

[57] T.-W. Kuo and K.-Y. Lam. Conservative and optimistic protocols. In *Real-Time Database Systems - Architecture and Techniques*, pages 29–44. Kluwer Academic Publishers, 2001.

[58] K.-W. Lam, K.-Y. Lam, and S. Hung. An efficient real-time optimistic concurrency control protocol. In *Proceedings of the First International Workshop on Active and Real-Time Database Systems*, pages 209–225. Springer, 1995.

[59] K.-W. Lam, K.-Y. Lam, and S. Hung. Real-time optimistic concurrency control protocol with dynamic adjustment of serialization order. In *Proceedings of the IEEE Real-Time Technology and Application Symposium*, pages 174–179. IEEE Computer Society Press, 1995.

[60] K.-W. Lam, V. Lee, S.-L. Hung, and K.-Y. Lam. An augmented priority ceiling protocol for hard real-time systems. *Journal of Computing and Information, Special Issue: Proceedings of Eighth International Conference of Computing and Information*, 2(1):849–866, June 1996.

[61] K.-W. Lam, S. H. Son, and S. Hung. A priority ceiling protocol with dynamic adjustment of serialization order. In *Preceedings of the 13th IEEE Conference on Data Engineering*, Birmingham, UK, 1997. IEEE Computer Society Press.

[62] K.-Y. Lam and S.-L. Hung. Concurrency control for time-constrained transactions in distributed databases systems. *The Computer Journal*, 38(9):704–716, 1995.

[63] K.-Y. Lam, S.-L. Hung, and S. H. Son. On using real-time static locking protocols for distributed real-time databases. *The Journal of Real-Time Systems*, 13(2):141–166, September 1997.

[64] K.-Y. Lam and T.-W. Kuo. Mobile distributed real-time database systems. In *Real-Time Database Systems - Architecture and Techniques*, pages 245–258. Kluwer Academic Publishers, 2001.

[65] J. Lee. *Concurrency Control Algorithms for Real-Time Database Systems*. PhD thesis, Faculty of the School of Engineering and Applied Science, University of Virginia, January 1994.

[66] J. Lee and S. H. Son. An optimistic concurrency control protocol for real-time database systems. In *3rd International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 387–394, 1993.

[67] J. Lee and S. H. Son. Using dynamic adjustment of serialization order for real-time database systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pages 66–75, Raleigh-Durham, NC, USA, 1993. IEEE Computer Society Press.

[68] U. Lee and B. Hwang. Optimistic concurrency control based on timestamp interval for broadcast environment. In *Advances in Databases and Information Systems, 6th East European Conference, ADBIS 2002, Bratislava, Slovakia*, volume 2435 of *Lecture Notes in Computer Science*, 2002.

[69] V. C. S. Lee and K-W. Lam. Optimistic concurrency control in broadcast environments: Looking forward at the server and backward at the clients. In H. V. Leong, W-C. Lee, B. Li, and L. Yin, editors, *First International Conference on Mobile Data Access*, Lecture Notes in Computer Science, 1748, pages 97–106. Springer Verlag, 1999.

[70] V. C. S. Lee and K.-W. Lam. Conflict free transaction scheduling using serialization graph for real-time databases. *The Journal of Systems and Software*, 55(1):57–65, November 2000.

[71] V. C. S. Lee, K-Y. Lam, and S-L. Hung. Virtual deadline assignment in distributed real-time database systems. In *Second International Workshop on Real-Time Computing Systems and Applications*, 1995.

[72] K.-J. Lin and M.-J. Lin. Enhancing availability in distributed real-time databases. *ACM SIGMOD Record*, 17(1):34–43, March 1988.

[73] Y. Lin and S. H. Son. Concurrency control in real-time databases by dynamic adjustment of serialization order. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 104–112, Los Alamitos, Calif., 1990. IEEE Computer Society Press.

[74] J. Lindström and K. Raatikainen. Dynamic adjustment of serialization order using timestamp intervals in real-time databases. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, pages 13–20. IEEE Computer Society Press, 1999.

[75] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[76] McObject LLC. http://www.mcobject.com/.

[77] D. Locke. Applications and system characteristics. In *Real-Time Database Systems - Architecture and Techniques*, pages 17–26. Kluwer Academic Publishers, 2001.

[78] Solid Infomation Technology Ltd. http://www.solidtech.com/en/products/ relationaldatabasemanagementsoftware/embed.asp.

[79] K. Marzullo. Concurrency control for transactions with priorities. Tech. Report TR 89-996, Department of Computer Science, Cornell University, Ithaca, NY, May 1989.

[80] M. T. Özsu and P. Valduriez. *Principles of Distributed Database System*. Prentice Hall, second edition, 1999.

[81] B. Purimetla, R. M. Sivasankaran, K. Ramamritham, and J. A. Stankovic. Real-time databases: Issues and applications. In S. H. Son, editor, *Advances in Real-Time Systems*, pages 487–507. Prentice Hall, 1996.

[82] K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1:199–226, April 1993.

[83] K. Ramamritham and J. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, 82(1):55–67, January 1994.

[84] Krithi Ramamritham, Sang H. Son, and Lisa Cingiser DiPippo. Real-time databases and data services. *Real-Time Systems*, 28(2-3):179–215, 2004.

[85] R. Ramamritham, R. M. Sivasankaran, J. A. Stankovic, D. F. Towsley, and M. Xiong. Integrating temporal, real-time, and active databases. *SIGMOD Record*, 25(1):8–12, 1996.

[86] L. Sha, R. Rajkumar, and J. P. Lehoczky. Concurrency control for distributed real-time databases. *ACM SIGMOD Record*, 17(1):82–98, March 1988.

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

[88] L. Sha, R. Rajkumar, S. H. Son, and C.-H. Chang. A real-time locking protocol. *IEEE Transactions on Computers*, 40(7):793–800, January 1991.

[89] LihChyun Shu, John A. Stankovic, and Sang H. Son. Achieving bounded and predictable recovery using real-time logging. *Comput. J.*, 47(3):373–394, 2004.

[90] M. Sivasankaram, R. and J. Ramamritham, K. an Stankovic. System failure and recovery. In *Real-Time Database Systems - Architecture and Techniques*, pages 109–124. Kluwer Academic Publishers, 2001.

[91] R. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 19(9):35–42, September 1986.

[92] S. H. Son, J. Lee, and Y. Lin. Hybrid protocols using dynamic adjustment of serialization order for real-time concurrency control. *The Journal of Real-Time Systems*, 4(2):269–276, June 1992.

[93] S. H. Son, D. Rasikan, and B. Thuraisingham. Improving timeliness in real-time secure database systems. *SIGMOD Record*, 25(1):25–33, 1996.

[94] B. Sprunt, D. Kirj, and L. Sha. Priority-driven, preemptive i/o controllers for real-time systems. In *Proceedings of the International Symposium on Computer Architecture*, pages 152–159, Honolulu, Hawaii, USA, 1988. ACM.

[95] J. Stankovic and K. Ramamritham. Editorial: What is predictability for real-time systems? *The Journal of Real-Time Systems*, 2:247–254, 1990.

[96] J. A. Stankovic, S. H. Son, and J. Hansson. Misconceptions about real-time databases. *IEEE Computer*, 32(6):29–36, June 1999.

[97] J. A. Stankovic and W. Zhao. On real-time transactions. *ACM SIGMOD Record*, 17(1):4–18, March 1988.

[98] John A. Stankovic, Sang H. Son, and Jörg Liebeherr. Beehive: Global multimedia database support for dependable, real-time applications. In *RTDB*, pages 409–422, 1997.

[99] A. Tansel, J. Clifford, S. Jojodia, A. Segev, and R. (ed.) Snodgrass. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings, 1994.

[100] S-M. Tseng, Y. H. Chin, and W-P. Yang. Scheduling real-time transactions with dynamic values: A performance evaluation. In *Proceedings of the Second International Workshop on Real-Time Computing Systems and Applications*, Tokio, Japan, 1995. IEEE Computer Society Press.

[101] Ö. Ulusoy and G. Belford. Real-time transaction scheduling in database systems. *Information Systems*, 18(6):559–580, 1993.

[102] Ö. Ulusoy and G. G. Belford. A simulation model for distributed real-time database systems. In *Proceedings of the 25th Annual Simulation Symposium*, pages 232–240, Los Alamitos, Calif., April 1992. IEEE Computer Society Press.

[103] Yuan Wei, Vibha Prasad, Sang H. Son, and John A. Stankovic. Prediction-based qos management for real-time data streams. In *RTSS*, pages 344–358, 2006.

[104] Yuan Wei, Sang H. Son, and John A. Stankovic. Maintaining data freshness in distributed real-time databases. In *ECRTS*, pages 251–260, 2004.

[105] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann, 2002.

[106] V. Wolfe, L. DiPippo, J. Prichard, J. Peckham, and P. Fortier. The design of real-time extensions to the open object-oriented database system. Technical report TR-94-236, University of Rhode Island, Department of Computer Science and Statistics, February 1994.

[107] M. Xiong, J. A. Stankovic, R. Ramamritham, D. F. Towsley, and R. M. Sivasankaran. Maintaining temporal consistency: Issues and algorithms. In *RTDB*, pages 1–6, 1996.

[108] Ming Xiong and Krithi Ramamritham. Deriving deadlines and periods for real-time update transactions. *IEEE Trans. Computers*, 53(5):567–583, 2004.

[109] Ming Xiong, Krithi Ramamritham, Jayant R. Haritsa, and John A. Stankovic. Mirror: a state-conscious concurrency control protocol for replicated real-time databases. *Inf. Syst.*, 27(4):277–297, 2002.

[110] Ming Xiong, Krithi Ramamritham, John A. Stankovic, Donald F. Towsley, and Rajendran M. Sivasankaran. Scheduling transactions with temporal constraints: Exploiting data semantics. *IEEE Trans. Knowl. Data Eng.*, 14(5):1155–1166, 2002.

[111] I. Yoon and S. Park. Enhancement of alternative version concurrency control using dynamic adjustment of serialization order. In *Proceedings of the Second International Workshop on Real-Time Databases: Issues and Applications*, Burlington, Vermont, USA, September 18-19, 1997.

[112] P. S. Yu and D. M. Dias. Analysis of hybrid concurrency control for a high data contention environment. *IEEE Transactions on Software Engineering*, SE-18(2):118–129, February 1992.