Managing Contention and Timing Constraints in a Real-Time Database System

Matthew R. Lehr and Sang H. Son
mrl6a@cs.virginia.edu
son@cs.virginia.edu

Department of Computer Science
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 22903, USA

Abstract

This technical report discusses how current real-time technology has been applied to a database management system to support firm real-time transactions. The report reviews priority-based CPU- and resource scheduling concepts and shows how they are used to avoid the problem of priority inversion in transaction service order, transaction progress, and memory allocation. Next, the appropriateness of optimistic concurrency control to real-time data management is examined, and the implementation of previously proposed methods WAIT-X(S) and Precise Serialization is detailed. Finally, the enforcement of firm deadlines using asynchronous aborts is discussed.

1. Introduction

As real-time applications grow more and more complex, so do the ways in which they maintain and access data. As the amount of data increases, programs typically turn away from application-specific solutions and seek general, adaptable, modular ways to manage data. Conventional systems use Database Management Systems (DBMS) to achieve these ends and DBMS technology is well-understood. Despite all of its features, a conventional DBMS is not quite capable of meeting the demands of a real-time system. Typically, its goals are to maximize transaction throughput, minimize response time, and/or provide some degree of fairness. A real-time DBMS system, however, must adopt goals which are consistent with any real-time system: providing the best service to the most critical transactions and ensuring some degree of predictability in transaction processing.

The StarBase real-time DBMS is an attempt to merge conventional DBMS functionality with real-time technology. StarBase supports the relational database model and understands a simple SQL-like query language. The DBMS maintains a centralized server to which local or remote clients submit transactions. Transactions may execute concurrently and serializability is the correctness criterion. In addition to this conventional functionality, StarBase seeks to minimize the number of high-priority transactions which miss their deadlines. StarBase uses no *a priori* information about transaction workload and discards tardy transactions at their deadline points. In order to realize many of these real-time goals, StarBase has been built on top of RT-Mach, a real-time operating system developed by Carnegie Mellon University [Tok90].

There are essentially three problems with which real-time DBMSs must deal: resolving resource contention, resolving data contention, and enforcing timing constraints. As with other real-time systems, tasks to be performed are stratified according to their relative importance to the system. Priority combines this relative importance with task timing constraints to provide a means to decide which of many tasks should be scheduled at any given moment. The intent is to always grant the highest priority tasks access to resources (CPU, critical sections, etc.). Similarly, StarBase considers each transaction a task in its own right and seeks to provide the best service to the highest priority transactions. The rest of this paper is devoted to addressing how StarBase allocates resources to the highest priority transactions and how it enforces timing constraints. Section 2 begins the discussion by describing the StarBase architecture in general terms. Sections 3 and 4 review solutions proposed for resource and data conflict mediation and show how they have been applied to StarBase. Finally, Section 5 details StarBase's unique methods of enforcing transaction deadlines.

2. Database Overview

The StarBase DBMS is organized as a multi-threaded server [Fig 1.]. It is assumed that database clients are physically disparate from the server, so message-passing is used to communicate between DBMS clients and the server itself. Transaction requests are sent via RT-Mach's Inter-Process Communication (IPC) mechanism and are queued at the server's service port. RT-Mach provides a naming service with which StarBase registers its service port during initialization. Clients look up the service port by querying the name server with StarBase's well-known name.

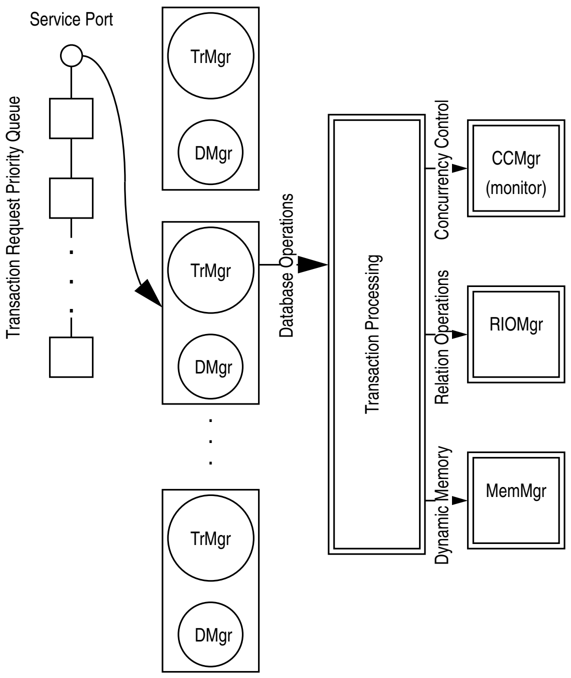

Fig. 1: StarBase Server Architecture

When a request enters service, a *transaction manager* thread of execution is charged with ensuring it is properly processed. The transaction manager executes the appropriate operations (read, write, insert, delete, create, remove) as dictated by the content of the request. At the start of transaction processing, the transaction manager starts a *deadline manager* thread, whose behavior is discussed in Section 5, to enforce the transaction's deadline. A transaction needs certain resources to execute, including mechanisms to acquire memory, read and write data from relations, and ensure that data remains consistent. StarBase provides three resource managers to provide these services: Small Memory Manager (MemMgr), Relation I/O Manager (RIOMgr), and Concurrency Controller (CCMgr). Each resource manager must ensure that transactions access their resources in a consistent and orderly fashion. To prevent mayhem, two of the resource managers are organized as *monitors* to synchronize the actions of different transactions. The services of the RIOMgr, however, are explicitly synchronized by the CCMgr.

StarBase uses a special type of algorithm to ensure data consistency, called *optimistic concurrency control*. Optimistic concurrency control allows transactions to proceed unhindered until they are ready to apply their updates to the database. To maximize the advantages of this, transaction processing code can access the contents of relations directly without proceeding through a Data Manager as in other DBMS.

3. Resource Contention and Transaction Scheduling

For decades a major trend in computing has been to increase efficiency by sharing resources. By providing the abstraction of processes (threads of execution) and a single software entity to control access to resources such as the CPU, memory, and disk, computers provide the illusion of the concurrent execution of different tasks in an orderly fashion. The ultimate arbiter of resources is the operating system, which is charged with resolving which thread of execution gets a particular resource at any given time. The goals of conventional systems, by and large, are to achieve fairness and minimize response time since they are designed to interact with humans. Real-time systems, however, are usually designed for embedded environments and require quick and predictable behavior in response to external mechanical and electrical stimuli (e.g. sensors, actuators, other computing units). Tasks that a real-time system must perform are ranked according to their importance and the most critical tasks are given the best access to resources to ensure they complete.

As with any application, the StarBase real-time DBMS is highly reliant on its native operating system, RT-Mach, to provide the priority-based services necessary for real-time resource scheduling. RT-Mach's services in turn are based on two major ideas (among others) which have been developed to ensure the allocation of resources to more important tasks in real-time systems. Those ideas are *priority-based CPU scheduling* and the *Basic Priority Inheritance Protocol* for non-preemptible resources. With both solutions, tasks to be performed are ranked by their relative priorities (a function of their criticality and/or feasibility), and the highest priority tasks are granted access to the resource in question.

The scheduling of real-time tasks on a single CPU has been studied for years. Many different algorithms have been introduced, each with its own set of requirements. Some require that the task set be completely specified beforehand, and others adapt to a dynamic load. Some use criticality alone to determine priority, and others use feasibility (as measured by deadline, periodicity, slackness, et al.), while others use both criteria to arrive at a priority assignment. In the case of StarBase, the scheduling policies available are those of its native operating system, RT-Mach. RT-Mach provides a variety of priority-based policies, including fixed-priority (round-robin (PRI-RR) and FIFO flavors (PRI-FIFO)), Rate Monotonic (RM), Deadline Monotonic (DM), and Earliest Deadline First (EDF). RT-Mach's real-time thread model distinguishes real-time threads of execution from ordinary ones, requiring the explicit specification of timing constraints and priority on a per-thread basis. The timing and priority information is then used as input to the RT-Mach scheduler [Tok90].

In the case of non-preemptible resources, however, contending threads must wait for the resource in question to be explicitly relinquished (rather than centrally scheduled as is possible with preemptible resources like the CPU) before one can access the resource. Ideally, a high-priority task should never have to wait to acquire a resource from a lower-priority task, but some resources cannot be easily preempted without ensuing chaos. There are cases then, where the goal of allocating resources to high priority tasks cannot be met immediately, and a high-priority task, T_H, has to wait for a low-priority task, T_L, to relinquish a resource, R, before T_H can proceed. This *de facto* blocking is a form of *priority inversion*, since T_L is allowed to execute instead of T_H. As previously mentioned, some priority inversion is unavoidable for non-preemptible resources, but Sha et al. identified a situation where *unbounded* priority inversion can be avoided [Sha90].

For example, consider what would happen if T_H waits for T_L to relinquish R and a third task, T_M, whose priority is between that of T_H and T_L, enters the system. Since T_M has a higher priority than T_L, priority-based CPU scheduling schedules it in preference to T_L (assuming T_M does not also wait to acquire resource R), and when T_M completes, T_L is not be appreciably closer to relinquishing R. T_H is thus indirectly blocked by the introduction of T_M, a lower priority task and suffers from priority inversion for a much longer period than if T_H and T_L executed alone. If more medium priority tasks are introduced into the system besides T_M, the priority inversion interval increases. In fact, T_H may be indefinitely blocked as a stream of medium-priority tasks enter the system.

To remedy this situation, Sha et al. proposed the Basic Priority Inheritance Protocol (BPI) [Sha90]. Simply stated, when a task, T_L, holding resource R effectively blocks other higher-priority tasks waiting to acquire R, it should inherit the highest priority of the waiting tasks. When T_L relinquishes R, it returns to executing at its former (lower) priority. Sha et al. also extended BPI to handle cases where chained waiting on resources can occur, and the resource holder inherits the ceiling of the priorities of all threads it blocks directly or indirectly.

RT-Mach, in turn, has striven to implement these ideas and provides both interprocess communication (RT-IPC) [Kit93] and thread synchronization (RT-Sync) [Tok91] facilities which obey BPI. RT-Mach implements BPI itself as a combination of priority queuing and priority inheritance.

To ensure the propagation of priorities between threads of execution on machines physically disparate from the local one, RT-Mach's RT-IPC package attaches a priority to each message sent. The priority can be either specified explicitly or derived from the sending thread. Incoming messages are queued in priority order at the destination and one of a set of prearranged receiver threads inherits the message priority in accord with BPI if all receivers are busy. Additionally, RT-IPC employs a *priority handoff* scheme distinct from the priority inheritance of BPI. Priority handoff ensures that the receiver thread executes at the priority of the last message it received. Priority inheritance is the temporary boost in priority of a receiver to match the priority of a message which the thread cannot yet receive. Priority handoff, however, occurs precisely at the point when the receiver receives a message. Priority inheritance occurs if the incoming message is of higher priority than some of the receivers, but priority handoff occurs for each message received.

RT-Mach's implementation of RT-Sync is a much more straightforward extrapolation of BPI than RT-IPC. RT-Mach provides a variety of synchronization mechanisms, including real-time mutex (mutual exclusion) and condition variables with which StarBase constructs its monitored MemMgr and CCMgr. Threads attempting to acquire a real-time mutex variable or waiting on a real-time condition variable are queued in priority order, and an inheritance mechanism is used to expedite the corresponding mutex variable holder if it blocks a higher priority thread.

As stated previously, applications must rely heavily on their host operating system to provide services such as scheduling, and StarBase is no exception. StarBase employs RT-Mach's priority-based CPU scheduling and BPI resource handling in several ways: to determine the transaction service order, to provide high-priority transactions the means to progress faster than low-priority transactions, to provide priority-consistent access to facilities such as the small memory manager and concurrency controller. In order to differentiate between high and low priority transactions, each StarBase transaction requires a priority specification. For purposes of uniformity, StarBase adopts the same data type RT-Mach uses to convey priorities: `rt_priority_t`. The use of this data type also lends itself to a straightforward translation of StarBase to RT-Mach priorities. Since the `rt_priority_t` includes a wide range of criticality and timing information, major changes in scheduling policy (e.g. Fixed Priority to Earliest Deadline First) are reduced to simple changes in the functions which compare priorities (e.g. changing the comparison of criticalities to one of deadlines) without any change in the client/server interface. Since the StarBase DBMS itself must make priority-based decisions (e.g. concurrency control), its priority-based comparisons involve priorities expressed in this data type. Of course, which policy is most appropriate differs from application to application, so the type of policy is left as a compile-time constant. Naturally, StarBase must use a consistent transaction scheduling policy across all of its priority-based decisions.

Transaction Service Order

Since performance ultimately degrades as the number of threads of execution in a system increase, and lazy allocation of resources adds unpredictability to the system, StarBase maintains only a fixed number of preallocated transaction manager threads. At the same time, since the StarBase DBMS has no *a priori* knowledge of transaction workload, more transactions may be submitted to the DBMS than it can handle at any given time. In order to throttle the flow in such circumstances, StarBase needs a mechanism to decide which requests to admit into service, and RT-Mach's RT-IPC facilities do just that in a convenient and priority-cognizant manner. To submit a transaction to the StarBase DBMS, a client places the transaction instructions and priority information into a message and uses RT-IPC to send the message to the DBMS server. Since RT-IPC queues incoming messages in priority order, the next available transaction manager receives the next highest priority unreceived message. Requests are therefore served in priority order and only the highest priority outstanding requests are in service at any given time. If a high priority transaction request cannot be serviced immediately because all transaction manager threads are busy serving some lower priority requests, RT-IPC's priority inheritance expedites one or more of the transaction managers so that the high priority request enters service at a time bounded by the minimum of the in-service transaction deadlines.

Transaction Progress

Once transactions enter service, StarBase needs to ensure that high priority transactions progress as quickly as possible. Since transactions require real-time execution, StarBase creates one real-time thread for each transaction manager and relies on RT-Mach's real-time CPU scheduling to schedule them. Transaction manager priorities are not specified explicitly by StarBase, however. Each obtains the correct priority assignment automatically upon receipt of a new transaction via RT-IPC's priority handoff mechanism.

Memory Manager

Transactions, depending on the nature of their operations, require some dynamic allocation of memory during their execution. StarBase maintains a small memory manager to allocate and manage dynamic memory. Since transaction managers of different priorities may attempt to use it simultaneously, entry into the small memory manager is guarded by a real-time mutex variable to avoid the priority inversion problem and to ensure the heap is accessed in mutual exclusion. To provide (relatively) predictable access to memory allocated through the manager, the heap is *wired* so that it cannot be paged out of physical memory.

Concurrency Controller

Although the StarBase concurrency controller is responsible for resolving contention at a higher level (i.e. data contention), it still relies on RT-Mach to provide basic synchronization and avoid the priority inversion problem. In particular, the concurrency controller must keep its own data structures consistent and ensure that transaction commits occur without interference. As such the concurrency controller is organized as a monitor, with a single real-time mutex variable for the monitor lock, and one real-time condition variable for each transaction manager. The precise function of the concurrency controller is detailed in the next section.

4. Data Contention and Concurrency Control

In addition to resources such as the CPU and memory, transactions also compete for access to the data stored in the relations themselves. A DBMS interposes itself between the application and the raw, unstructured storage medium to provide the abstraction of high-level operations called transactions. To obtain reasonable performance, a DBMS must allow multiple transactions to access data concurrently while requiring that the outcome appear as though it were the result of a serial execution of those transactions. Balancing these two behaviors induces a problem which is quite distinct from ordinary contention for operating system resources: contention for data. To enforce data access patterns that are consistent with a serial execution, transactions which read and write the same data must be forcibly reordered by allowing one to proceed while the others are delayed. The procedure which makes these decisions is called a *concurrency control algorithm*, of which there are two major types: *lock-based* and *optimistic*.

Lock-based concurrency control requires that transactions obtain shared or exclusive locks on data items before they may be accessed. If two transactions wish to access a data item in a conflicting way (i.e. at least one requires an exclusive lock), lock-based concurrency control delays granting one or the other of the transactions access to the data. Optimistic concurrency control allows a transaction to access data with no hindrances up to the validation point, when it must certify that it has accessed data in a manner that has not conflicted with other transactions. Should optimistic concurrency control determine that the validator conflicts with other running or recently committed transactions, it aborts one or more of the transactions involved in the conflict. Carey et al. [Car84] notes that locking tends to detect and resolve conflicts over data relatively early in a transaction's execution whereas optimistic concurrency control tends to resolve conflicts late. He also asserts that delaying transactions and maintaining the work they've performed by locking is a much more efficient utilization of system resources than aborting transactions and discarding their work. Carey found that the amount of transaction abortions is the single most important factor in determining the performance of an DBMS.

lock-based		optimistic				
conventional	real-time	conventional			real-time	
2PL	2PL-HP	strict retrospective	mixed	strict prospective	mixed	strict prospective

Fig 2: A Partial Taxonomy of Concurrency Control Methods

StarBase's concurrency control draws heavily from the work of two research groups. First, Haritsa reasoned that optimistic concurrency control can outperform lock-based algorithms in a firm real-time setting [Har91]. He then developed a real-time optimistic concurrency control method, WAIT-X(S), which he found empirically superior, over a wide range of resource availability and system workload levels, to a previously proposed real-time lock-based concurrency control method called 2PL-HP [Har91]. Second, Lee et al. devised an improvement to the conflict detection of optimistic concurrency control in general, which StarBase integrates with Haritsa's WAIT-X(S) [Lee94].

Why Optimistic Concurrency Control?

Experience has shown that in conventional database systems under conditions of high competition for data, locking outperforms optimistic concurrency control. This is attributed to the relative efficiency of the policy each concurrency control method uses to resolve data conflicts. Lock-based concurrency control requires contending transactions to wait until data is free to access, whereas optimistic concurrency control resorts to aborting and restarting some or all of the transactions contending for a particular data item. Locking is deemed more efficient than optimistic concurrency control for two reasons: First, its blocking policy tends to throttle the contention of transactions which still have relatively large amounts of data to access, and secondly, restarts increase contention since operations of transactions which are aborted and subsequently resubmitted must be reperformed.

Surprisingly, however, the advantage that locking enjoys in conventional systems may not be attainable in real-time databases. In addition to the goal of maximizing transaction throughput espoused by conventional DBMS design, real-time systems are required to minimize transaction lateness (or eliminate it entirely). Contention for scarce resources is typically resolved by allocating such resources to those tasks deemed the most critical or likely to complete in time at the expense of tasks of lesser importance or feasibility. Highly loaded systems experience conditions under which an abort/restart policy must be used in order to enforce the timing constraints or performance guarantees for the highest priority tasks. A real-time lock-based concurrency control cannot rely solely on blocking to resolve contention; it must use some form of abort. In particular, Abbot et al. [Abb92] proposed a real-time lock-based concurrency control variant of *two-phase locking* called 2PL-HP. 2PL-HP resolves data contention in favor of more important transactions by aborting transactions holding locks that a higher-priority transaction wishes to acquire exclusively.

Haritsa gives two reasons that optimistic concurrency control may be able to turn the tables on locking in a real-time system. As just mentioned, real-time locking cannot employ contention resolution based solely on blocking--it must abort some transactions to avoid unbounded priority inversion and enforce deadlines. Conversely, the pure abort/restart policy of real-time optimistic concurrency control is dampened by the blocking imposed by real-time priority based resource scheduling. The blurring of these fundamental differences between locking and optimism places the two concurrency control approaches on a more even footing.

Haritsa states that the other deciding factor is precisely when data conflicts are resolved. Lock-based methods such as 2PL-HP can grant a high-priority transaction exclusive access to data very early in the transaction's execution. Other transactions are precluded from using the data and aborted even in the case that the exclusive lock holder cannot feasibly make its deadline. Haritsa terms aborted transactions which result from situations of this sort as *wasted restarts*. Conversely, optimistic concurrency control allows transactions to progress independently up to the point of validation, effectively postponing any conflict resolution. By delaying decisions on data conflicts, only feasible transactions may progress all the way to the validation point while infeasible transactions abort one by one as they miss their deadlines. Thus, by the time a transaction has made it into validation, it has proven its feasibility and the certainty of completing the transaction, should it win all of its data conflict contests, is guaranteed.

One may legitimately ask, "Why attempt a transaction which is obviously infeasible?" The answer is that despite claims to the contrary, determining *a priori* worst-case execution time is difficult. There are cases where small variances may turn a marginally feasible transaction into an infeasible one. At its outset, the transaction may appear feasible but is later rendered infeasible by an unforseen event such as the introduction of a higher priority transaction which blocks it.

WAIT-X(S)

StarBase uses the WAIT-X concurrency control algorithm proposed by Haritsa [Har91]. WAIT-X is optimistic, using *prospective* conflict detection and priority-based conflict resolution. WAIT-X's conflict detection is prospective in the sense that it looks for conflicts between the validator and transactions which may commit sometime in the future (i.e. running transactions). Prospective conflict detection is also referred to as *forward validation* or *broadcast commit*. The attendant advantages of the prospective method are that potential conflictors are readily identifiable, dataset comparisons are simplified, and conflicts are detected much earlier in the execution history. Real-time optimistic methods are precluded, however, from *retrospective* (or *backward validation*) conflict detection, which compares the validator to transactions which committed in the recent past. Since all the transactions which conflict with a validator have committed, there is only one outcome in the face of irreconcilable conflict: abortion of the validator regardless of its priority relative to its conflictors. Prospective conflict detection, on the other hand, allows the concurrency control to choose between aborting the validator or all of its conflictors in a priority-cognizant manner.

When WAIT-X detects conflicts between a validator and some running transactions, it can choose one of three outcomes for the validator. It may abort the validator, it may commit the validator and abort the conflictors, or it may delay the validator slightly in the hope that conflicts resolve themselves in a favorable way. Which course of action to take is a function of the priorities of the validator and conflictors. In particular, Haritsa divides the conflictors into two sets: those conflictors with higher priority than the validator (*CHP*), and those with lower priority (*CLP*). *WAIT-X* blocks the validator until the CHP transactions comprise less than a critical portion, *X%*, of the conflict set:

```
while (CHP transactions in conflict set and
        CHP transactions comprise less than
        X% of conflict set)
    wait;
abort conflict set;
commit validator;
```

Haritsa found experimentally that low values of X tend to minimize the deadline miss ratio for light loads, and high values of X tend to minimize the deadline miss ratio for heavy loads. He established X = 50% as the threshold value which minimizes the overall deadline miss ratio, but applications which require minimization of the highest-priority deadline miss ratio should probably use a greater value for X.

The final aspect of the WAIT-X method deals with handling the abort of the transaction should WAIT-X block it until its deadline. Haritsa claims that transactions which run up against their deadlines while waiting can either be immediately sacrificed by aborting (WAIT-X(S)) or committing (WAIT-X(C)). Sacrifice is preferred over commit since waiters are more likely to be lower priority than most of their conflictors. More importantly, however, commission at the deadline point would effectively extend the execution of a transaction past its deadline, so WAIT-X(C) is not practical for systems requiring firm real-time constraints such as StarBase.

WAIT-X(S) Implementation

The StarBase concurrency control unit implements Haritsa's WAIT-X as a monitor and is a more active entity than other typical concurrency controllers. The concurrency control manager (CCMgr) opens and closes relations on behalf of executing transactions, performs write-throughs to the database, handles asynchronous aborts, and eliminates a potential race condition between the commission of a transaction and the expiration of its deadline. Transaction managers use the six services provided by the CCMgr (`RegisterTransaction`, `RegisterRelationReference`, `UpdateReadSet/WriteSet`, `Validate`, `ScheduleAsynchronous-Abort`, and `AbortSelf`) by calling the corresponding monitor entry procedure. Each monitor entry procedure locks the CCMgr monitor lock to gain access to the monitor and unlocks the monitor lock when exiting. The monitor lock itself is implemented as an RT-Mach mutex variable to control priority inversion between contending transaction managers. Once inside the monitor, of course, operations proceed in a mutually exclusive fashion.

Although on paper WAIT-X consists of a simple test to determine whether a transaction waits or commits, in practice, the test is actually a trigger whose truth value can change at any instant as transactions enter (by reading relations) and exit (by aborting) the validator's conflict set. The CCMgr is a synchronous modification of the asynchronous WAIT-X test, where the validation state corresponds to the testing the trigger, the wait state corresponds to the loop body, and the committed state corresponds to the statements after the `while` loop. Note that validators may be aborted while in the wait state either due to the commitment of other validators or due to the expiration of the validator's deadline.

As previously mentioned, the composition of a validator's conflict set may change from instant to instant. The most frequent case, when a running transaction advances in its read or writeset, is expensive to check because of its frequency and because of the size of the read-/writeset data structures. The CCMgr limits checking the trigger condition to cases where it is reasonably sure conflict sets have changed: when a transaction enters validation for the first time and when a transaction aborts. Note that in this scheme a particular transaction's wait in the CCMgr is strictly bounded by its deadline and waiting transactions retry validation by the earliest deadline of all transactions in the system (subject to the availability of the processor to the transaction with earliest deadline). In order to give precedence to the highest priority transactions, all waiting transactions retry validation in priority order.

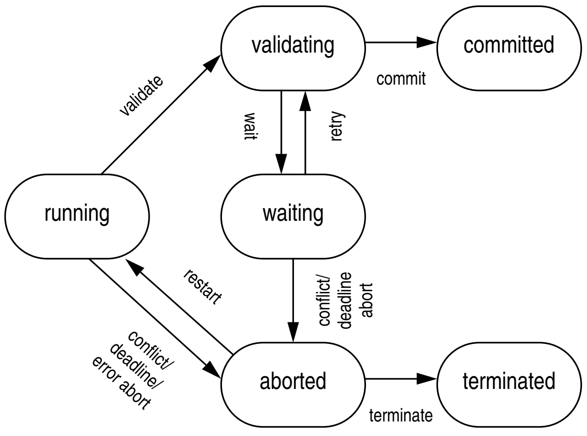

Fig. 3: WAIT-X(S) State Diagram

As is typical, reads and writes are recorded in bitmaps for each relation a transaction references. Comparison of the read- and writesets of transactions during conflict identification can then be expedited by performing a word-wise logical AND on bitmap pairs to detect any overlaps. Since WAIT-X uses prospective validation, only the readset of a potential conflictor need be compared to the writeset of the validator: the potential conflictor is still running so none of its writes are visible to the validator. Prospective validation's conflict detection is simple and relatively low-cost, but it can be improved upon. A method to augment WAIT-X(S)'s conflict detection scheme is discussed in a later section.

Because the bitmap comparison process is not atomic, it is of paramount importance that running transactions be prohibited from advancing in their read- or writesets during the validation of a potential conflictor. Transactions are required to update their read- and writesets prior to actually performing the read from a relation or write into their own private workspace. As the CCMgr retries each validator, it cycles through the validator's datasets marking each relation referenced as locked. Part of the update read-/writeset function checks whether the relation is locked before allowing the operation to proceed. If the relation is locked, the transaction is required to enter the CCMgr monitor, effectively blocking it, and if the transaction is of higher priority than the thread currently executing inside the CCMgr, that thread inherits the transaction's priority. Since no mutual exclusion is used when updating the read- and writesets, there is a race condition between the executing transaction writing bit *i* and the validator reading bit *i*. If the executing transaction reads data item *i* first, writing bit *i* in the process, subsequent validation detects the presence of 1 in the *i*th position in the bitmap and the transaction will be correctly seen as conflicting. If the CCMgr compares a validating transaction's bitmap to that of the executing transaction, marking the relation as locked beforehand, the executing transaction's read will detect the locked relation and attempt to enter the CCMgr monitor, effectively blocking it until the completion of validation.

When the CCMgr computes the conflict set for a given validator, it tallies CHP and CLP transactions. To determine the priority of a conflictor relative to the validator, the CCMgr employs a function of transaction priorities (using RT-Mach's own data type, `rt_priority_t`) which returns TRUE if the first transaction is of higher priority than the second. Note that this function is the same one employed to ensure transactions retry validation in priority order. The exact behavior of the function is arbitrary and fixed at compile-time, but given the current composition of RT-Mach's priority data type, possibilities include Fixed Priority, Earliest Deadline First, and Least Slack First. Once the CHP and CLP have been determined, the CCMgr decides whether the validator can commit or must remain on the waiting list. If the validator commits, the CCMgr schedules aborts for transactions in the validator's conflict set.

The wait state itself is implemented by associating an RT-Mach real-time condition variable appropriately called `waiting` with each transaction. When the CCMgr decides that a validating transaction should wait, the transaction manager is enqueued on a queue of other waiting transactions and suspended on its condition variable. This in turn releases the CCMgr monitor lock and allows other transaction managers to use CCMgr services. The suspended transaction manager is subsequently resumed when another transaction manager calls into the CCMgr to validate or abort. At that point all transactions in the wait queue are retried individually in priority order and if the CCMgr decides that one in particular commits or aborts, it signals the corresponding `waiting` condition variable, unblocking the formerly suspended transaction manager.

Precise Serialization

Precise serialization is a conflict-detection scheme for optimistic concurrency control [Lee94]. The goal of precise serialization is to identify transaction conflicts which strict prospective conflict detection considers irreconcilable but can actually be resolved without aborting the transactions involved. StarBase replaces the prospective conflict detection portion of the WAIT-X(S) scheme with Precise Serialization so that WAIT-X(S) can still enforce transaction serializability while incurring fewer transaction aborts.

In particular, Lee identified the case where a validator, T_V, attempts to commit and write a data item x which another uncommitted transaction T_{CR} has read but not written. Lee terms data conflicts of this type *write-read conflicts*. As mentioned previously, strict prospective validation checks the writeset of the validator against the readset of its potential conflictors, identifying write-read conflicts. If it detects such a conflict, the resolution requires aborting some of the conflicting transactions. Note, however, that if T_{CR} were to commit first, there would be no conflict on data item x. Haritsa noticed the same problem and describes part of the rationale behind the priority wait scheme of WAIT-X as a passive attempt to induce transactions to reserialize themselves in a nonconflicting order. Lee's Precise

Serialization takes a more deterministic tack: it allows T_V to commit while T_{CR} is still running, but requires T_{CR} to behave as if it had committed before T_V! T_{CR} is constrained so that it cannot read any data item written by T_V because it would see a "future" value, and it cannot write any data item read by T_V since T_V has committed and cannot change the past. Finally T_{CR} must discard (as late writes) updates to any data items which T_V wrote during its commit. This pseudo-reserialization of T_V and T_{CR} is called *backward ordering* and its goal is to increase the probability that potential conflictors can complete without either aborting and restarting.

Precise Serialization Implementation

Since Precise Serialization is a conflict-detection scheme, not a full-blown method of concurrency control, it supplements StarBase's WAIT-X implementation rather than replacing it entirely. Precise Serialization modifies the WAIT-X validation conflict detection and requires the addition of a mechanism to detect when a pseudo-reserialized transaction does not behave in accordance with its virtual order in the execution history.

During validation, Precise Serialization partitions the set of conflicting transactions into those which conflict reconcilably and those which conflict irreconcilably. Should the validator be allowed to commit, the reconcilable conflictors must be pseudo-reserialized by backward ordering, while the irreconcilable conflictors must be aborted. To keep track of which are which, StarBase maintains a reserialization candidate set for the validator in addition to the conflict set of the WAIT-X implementation described previously. The conflict set still identifies which transactions conflict irreconcilably with the validator, but the candidate set identifies precisely those datasets among which reconcilable write-read conflicts exist.

To construct the candidate set and the conflict set at the point of validation, the CCMgr cycles through each dataset referenced by the validator, T_V. If T_V has only a write-read conflict with an uncommitted transaction, T_{CR}, on a dataset, then the serialization order should be $T_{CR} \rightarrow T_V$ (backward validation) and the conflicting datasets are added to the reserialization candidate set. If T_{CR} has only a write-read conflict with T_V, then the serialization order should be $T_V \rightarrow T_{CR}$ (forward validation). In this case T_V and T_{CR} are considered to be non-conflicting. If the CCMgr determines that the serialization order should be simultaneously $T_{CR} \rightarrow T_V$ and $T_V \rightarrow T_{CR}$, then T_V and T_{CR} are irreconcilably conflicting, and T_{CR} is added to the conflict set. Note that the CCMgr does not consider write-write conflicts since transactions are required to read tuple locations to determine their values or to establish that they are empty before writing them. Consequently a writeset is always a subset of the readset (for a given transaction and relation) and checking both against a potential conflictor's writeset is redundant.

Once the candidate set and conflict set are completely identified, the CCMgr determines whether the validator should commit or wait according to the WAIT-X commit test. If the validator waits, the conflict and candidate sets are discarded--they will be recomputed if and when the validator retries validation. If the validator commits, the transactions in the conflict set are aborted and the CCMgr must pseudo-reserialize the reconcilable conflictors. Pseudo-reserialization is achieved by attaching copies (or *remnants*) of T_V's datasets to those datasets with which they conflict--note that these dataset pairs are precisely those comprising the reserialization candidate set. Thus when a conflictor later updates its read- and writesets, it can quickly check whether the operation violates its virtual order in the execution history by consulting the dataset remnants attached to the dataset involved in the operation.

Since one of T_V's datasets may conflict with more than one of the conflictors', a remnant is given a reference count rather than physically copied. As conflictors commit or abort one by one, the CCMgr decrements the reference count. When the last conflictor terminates, the CCMgr discards T_V's dataset remnant.

In the same token several transactions may commit even though they conflict with a particular transaction, T_{CR}. The dataset remnants of these transactions attached to T_{CR} are collectively known as the *recently committed conflicting datasets* (or *RCC*s). Pseudo-reserialized transactions such as T_{CR} must check each remnant in the RCCs for a given dataset whenever they read or write to that dataset. As previously mentioned, T_{CR} cannot read anything marked as written in its RCCs, since it would read a "future" value. In most cases T_{CR} cannot write anything marked as read in its RCCs, since it would write a "past" value. The exception occurs when T_{CR} writes a data item that T_V has also written, in which case T_{CR}'s write is discarded as a late write. The net result is that only the value that T_V wrote is visible, consistent with the execution history $T_{CR} \rightarrow T_V$. Unfortunately StarBase's update operation may use past values to compute new ones, precluding the use of late writes for it. The only situation in which the late write phenomenon can be used is one in which the reserialized operation is supposed to have been performed before a delete. Since delete is idempotent, the reserialized operation can be correctly discarded.

5. Enforcing Time Constraints

Each StarBase transaction is accompanied by a deadline specification. Since StarBase is a firm real-time DBMS, it attempts to process the transaction and reply to the application at or before this deadline; no processing should occur after the deadline. Firm deadline transactions may be contrasted with *soft deadline* transactions which are viewed as having some usefulness even if their execution extends beyond the deadline point. Soft deadline transactions typically execute at a reduced priority after the deadline until a point where they reach zero value and are discarded. *Hard deadline* transactions are those transactions whose failure to execute on time is viewed as catastrophic. In general, systems which support hard deadline tasks (such as a DBMS supporting hard deadline transactions) must fix each detail of task execution beforehand, including resource usage and scheduling order. Rather than take the chance that any task misses its hard deadline, such systems massively preallocate and underload resources to ensure temporally precise behavior with no deviations.

Deadline Management

The first step in enforcing firm deadlines is detecting exactly when the deadline expires. As with other real-time functionality, StarBase relies heavily on the RT-Mach operating system to provide supporting mechanisms. RT-Mach provides the concept of a real-time *deadline handler.* Timing faults on a task such as deadline expiration occur asynchronously with regard to the task's thread of execution and are essentially exceptional events. The nature of the actions the handler performs when a timing fault occurs depends on application semantics. Typical actions are to abort the task (firm deadline) or lower its priority (soft deadline). Implementation of a deadline handler requires time-based synchronization in addition to RT-Mach's real-time thread model. In order to ensure the handler action is ready to execute before the deadline, the real-time deadline handler must be eagerly allocated as a real-time thread to execute the deadline handler code. The deadline handler thread then uses a *real-time timer* to block the thread until the deadline expires. A real-time timer is an RT-Mach abstraction which allows real-time threads to synchronize with particular points in time as measured by *real-time clock* hardware devices [Sav93]. Each timer is associated with exactly one clock, but a single clock can support many timers.

RT-Mach provides a default deadline handler constructed from the building blocks discussed above, but it is inadequate for StarBase's purposes. First, the default deadline handler supports only tasks with uniform deadlines, but StarBase, since it assumes no *a priori* information about its transaction workload, treats all transactions as if they have different timing constraints. StarBase requires that its deadline handlers adapt to new transactions and their deadlines as they enter service. Secondly, a RT-Mach default deadline handler forcibly suspends a task when it misses its deadline so that the task does not interfere with the handler's execution. If a task misses its deadline while in the middle of a critical section, the suspended task cannot leave the critical section until it is resumed. Since StarBase uses a critical section to resolve potential race conditions between transaction commit (by the transaction manager) and deadline abort (by the deadline manager), use of a default deadline handler can result in deadlock. Thirdly, default deadline handlers do not allow the transaction and deadline managers to synchronize cooperatively. A deadline manager must know when a transaction completes so that it does not generate a useless abort; a transaction manager must know when the deadline expires, so that it does not commit the aborted transaction. Neither is possible without some shared state which must be accessed consistently (i.e. atomically).

Deadline Management Implementation

The solution, then, is to devise a deadline handler implementation which handles variable deadlines, avoids potential deadlocks, and is eagerly allocated to provide some degree of predictability but at the same time takes precedence over the transaction it manages when the transaction deadline expires.

As mentioned in Section 3, RT-Mach provides RT-Sync real-time thread synchronization facilities. Each transaction and deadline manager pair can be synchronized using RT-Sync to construct a monitor with two real-time condition variables, `newTransaction` and `dmgrCancel`. The transaction manager must be sure that the deadline manager is ready to enforce a new deadline before a new transaction arrives, and the deadline manager must be sure the transaction manager has received a new transaction before it prepares for the new deadline expiration. The condition variable `newTransaction` is used both to wait when one of the managers lags behind the other and to signal the arrival of a new transaction to the deadline manager.

```
rt_mutex_t      monitorLock;
rt_condition_t  newTransaction;
message_t       request;
boolean_t       tmgrReady = FALSE;

while (TRUE)
  {
  rt_mutex_lock (monitorLock, NULL);
  tmgrReady = TRUE;
  if (dmgrReady == FALSE)
    {
    if (dmgrArmed)
      rt_condition_signal (dmgrCancel);
    rt_condition_wait (newTransaction,
                       NULL);
    }
  mach_msg_receive (request);
  rt_condition_signal (newTransaction);
  tmgrReady = FALSE;
  rt_mutex_unlock (monitorLock);
  /* execute transaction */
  }
```

Fig. 4: Transaction Manager

```
rt_condition_t  dmgrCancel;
boolean_t       dmgrArmed = FALSE;
boolean_t       dmgrReady = FALSE;

rt_mutex_lock (monitorLock, NULL);
while (TRUE)
  {
  if (tmgrReady)
    rt_condition_signal (newTransaction);
  dmgrReady = TRUE;
  rt_condition_wait (newTransaction,
                     NULL);
. dmgrReady = FALSE;
  dmgrArmed = TRUE;
  status =
    rt_condition_wait (dmgrCancel,
                       request.deadline);
  dmgrArmed = FALSE;
  if (status == KERN_SUCCESS)
    continue;
  /* abort transaction */
  rt_mutex_lock (monitorLock);
  }
```

Fig. 5: Deadline Manager

The condition variable `dmgrCancel` is used much differently. In order to expedite the processing of subsequent transactions, the transaction manager should be able to actively cancel the deadline manager in the event that the transaction completes well in advance of its deadline. The deadline manager sets a boolean `dmgrArmed` to TRUE whenever it blocks waiting for the deadline to expire so that the transaction manager knows to cancel it. The transaction manager must be able to atomically test and the deadline manager must be able to atomically set `dmgrArmed` (i.e. from within the monitor). Furthermore, `dmgrArmed` must be set atomically with the blocking of the deadline manager, otherwise the transaction manager's cancellation could interleave between the two operations, nullifying it and allowing the deadline manager to wait for the deadline expiration anyway.

Additionally, the deadline manager must release the monitor lock when it blocks so that the transaction manager can enter the monitor and test `dmgrArmed` to determine whether to cancel the deadline manager. StarBase can satisfy all of these requirements by employing a real-time condition variable with a timeout. In particular the deadline manager sets `dmgrArmed` and then waits on the condition `dmgrCancel` with a timeout set to the deadline of the transaction currently in service. The deadline manager unblocks either when the deadline expires (the wait on `dmgrCancel` times out) or when the transaction manager cancels the deadline manager (by signalling on `dmgrCancel`). Naturally if the wait times out, the deadline manager must reenter the monitor.

The transaction and deadline manager behaviors are presented in Figures 4 and 5. This solution allows the deadline handler to deal with deadlines which vary from transaction to transaction since the transaction and deadline managers synchronize before a transaction enters service. The use of a monitor to synchronize the transaction and deadline managers also avoids the deadlock possible were the deadline manager capable of explicitly suspending the transaction manager. Another implementation of the deadline handler involves creating and destroying the deadline manager at the beginning and end of each transaction. Eagerly allocating the deadline manager thread, however, reduces the amount of variability in transaction service times, providing an increased degree of predictability.

Finally, the easiest goal to achieve is that of the deadline manager taking precedence over its transaction manager. Since the deadline handler's execution is considered more critical than the transaction's when the deadline expires, the deadline handler should be assigned a higher priority so that RT-Mach gives it preferential scheduling relative to the transaction whose deadline it handles. At the same time, the execution of the deadline handler should not cause priority inversion by interfering with the transaction managers of higher priority transactions. In order for the deadline handler to function as desired, it should have a slightly higher criticality and slightly tighter timing constraints than its corresponding transaction manager, but a lower criticality and looser timing constraints than transaction managers for higher priority transactions.

Fortunately, the criticality and time spaces are both very large in RT-Mach (at least 2^n where n is the number of bits in a word), which allows StarBase to map the external transaction priorities onto the priorities at which the transaction and deadline manager real-time threads actually run. Furthermore, real-time CPU and resource scheduling generally make decisions on which task to run by simply comparing priorities without quantifying how much they differ. Thus the magnitude of the difference between the priorities of a high-priority and a low-priority task is irrelevant; the high priority task is scheduled until it blocks. The fact that priority spaces are so large and scheduling decisions are based on how rather than how much priorities differ allows StarBase to translate "external" transaction priority values into "internal" priority values used by RT-Mach without radically altering the semantics of the transaction priorities themselves.

Type	(External) Transaction Priority	Transaction Manager Priority	Deadline Manager Priority
criticality	c	2 * c + 1	2 * c
timing constraints (nsec)	t	t	t - 1

Fig. 6: Thread Priority Assignments

The RT-Mach criticality priority space consists of unsigned integers, with 0 being the highest criticality and 2^n-1 being the lowest. The transaction and deadline manager thread criticalities supplied to RT-Mach are gotten by doubling the external transaction priority and adding one to the transaction manager criticality. A deadline manager thus always has a greater criticality than its own transaction manager thread but has a lesser criticality than that of the next highest criticality transaction.

Although time is viewed as continuous and real-valued, RT-Mach's ability to measure it is limited by its clock hardware resolution. RT-Mach, therefore, maintains a data type which represents discretized time in terms of nanoseconds, though its clocks measure time with significantly lower precision. Tighter timing constraints for the deadline manager are gotten by adding one nanosecond to each timing constraint of the corresponding transaction manager. Since RT-Mach's clocks are not accurate to nanosecond precision, distinct transaction managers whose timing constraints differ only by one nanosecond essentially have the same priority (disregarding their criticalities), and tightening a deadline manager's timing constraints by such a small value does not block the processing of an appreciably higher priority transaction.

Asynchronous Aborts

As previously discussed, firm deadlines are handled asynchronously by a deadline handler which is charged with aborting the task in question. In StarBase, the asynchrony between transaction and deadline managers results in a race condition between the commit and deadline abort of a transaction. The concurrency controller (CCMgr) is the authority which permits a transaction to commit and the commit/abort contention is resolved through it. As described in Section 4, the CCMgr is a monitor and threads executing inside of it are capable of atomically determining whether a transaction is in the process of committing or not.

When the deadline expires and the deadline manager must abort the transaction, it calls into the CCMgr. If the transaction has not yet committed, the CCMgr marks the transaction as aborted and disallows it as a potential conflictor with other validators by unlinking it from CCMgr internal data structures. How the CCMgr subsequently notifies the transaction manager of the abort depends on the state of the transaction. If the transaction has not yet entered validation, the transaction manager is notified the next time it updates its read- or writesets; if the transaction has entered validation (i.e entered the wait state), the CCMgr resumes the transaction manager according to the mechanism described in Section 4 with the status that it has failed validation.

In addition to the race condition between the commit and abort of a transaction, there is another race condition between simultaneous aborts. For example, a transaction may discover a semantic error (e.g. relation not found) near the point where the deadline expires or a transaction may abort due to conflicts during validation. Because of the different natures of these aborts, different actions are required on the part of StarBase. The CCMgr again arbitrates which one of multiple aborts takes precedence. The most important is the deadline abort which supersedes all other aborts in order to expedite replying to the client. Semantic errors are next in line and conflict aborts are least critical. Aborts due to deadline expiration and semantic errors must prevail over conflict aborts, since the former require discarding transactions permanently whereas the latter result in restarting transactions.

As described in Section 4, all validating transactions are retried whenever a transaction enters validation for the first time or aborts. Since retrying validation may result in multiple transactions committing or aborting, it may be a fairly lengthy process. Rather than allowing a deadline manager's call into the CCMgr monitor to block it for such a long period of time, the CCMgr maintains a thread which acts as a proxy. When a deadline manager requests that the CCMgr abort its transaction, the deadline manager simply hands off the appropriate priority to the proxy thread and then signals it. The deadline manager is then free to leave the CCMgr monitor and reply to the client while the proxy retries all waiting validators. Note that the deadline manager assigns the priority of the transaction manager rather than its own priority to the proxy so that the deadline manager can proceed unhindered.

6. Conclusion

This paper details the architecture to support a firm real-time DBMS assuming no *a priori* knowledge of transaction workload characteristics. The paper describes how existing real-time technology has been applied to the problems of resource and data contention, and introduces unique methods to enforce deadlines. The next step is to extend these solutions to the situation in which transaction characteristics are at least partially specified beforehand. With prior knowledge, a real-time DBMS can provide better support for periodic transactions. Execution time estimates and off-line analysis can be used to increase DBMS-wide predictability. Temporal consistency [Ram92], where data used to derive new data must be consistent within a certain validity interval, is also a matter to be explored.

Acknowledgements

Youngkuk Kim and Juhnyoung Lee provided many valuable ideas and suggestions on how to improve this paper.

References

[Abb92] Abbott, Robert K., and Garcia-Molina, Hector. “Scheduling Real-Time Transactions: A Performance Evaluation.” *ACM Transactions on Database Systems*, Vol. 17, No. 3, September 1992.

[Car84] Carey, Michael J., and Stonebraker, M. R. “The Performance of Concurrency Control Algorithms for Database Management Systems.” *Proceedings of the Tenth VLDB Conference*, Singapore, August 1984.

[Har91] Haritsa, Jayant R. “Transaction Scheduling in Firm Real-Time Database Systems.” TR1036. Department of Computer Science, University of Wisconsin. August 1991.

[Hua90] Huang, J., Stankovic, J., Towsley, D., and Ramamritham, K. “Real-Time Transaction Processing: Design, Implementation, and Performance Evaluation.” TR90-43. COINS, University of Massachusetts. May, 1990.

[Kim94] Kim, Youngkuk, Lehr, Matthew, George, David and Son, Sang H. “A Database Server for Distributed Real-Time Systems: Issues and Experiences.” *Second IEEE Workshop on Parallel and Distributed Real-Time Systems*, Cancun, Mexico, April 1994.

[Kit93] Kitayama, Takuro, Nakajima, Tatsuo, and Tokuda, Hideyuki. “RT-IPC: An IPC Extension for Real-Time Mach.” *Proceedings of the Second Microkernel Workshop*, September 1993.

[Lee93] Lee, Juhnyoung and Son, Sang H. “Using Dynamic Adjustment of Serialization Order for Real-Time Database Systems.” *14th IEEE Real-Time System Symposium*, December 1993.

[Lee94] Lee, Juhnyoung, and Son, Sang H. “Precise Serialization for an Optimistic Concurrency Control Algorithm.” Submitted for Publication.

[Leh93] Lehr, Matthew R. “StarBase v2.2 Implementation Details.” TR CS-93-48. Department of Computer Science, University of Virginia. July 1993.

[Loe91] Loepere, Keith. *Mach 3 Kernel Principles*. Open Software Foundation and Carnegie Mellon University. 1991.

[Ram92] Ramamritham, Krithi. “Real-Time Databases.” *International Journal of Distributed and Parallel Databases*, Vol. 1, No. 2, April 1993.

[Sav93] Savage, Stefan, and Tokuda, Hideyuki. “Real-Time Mach Timers: Exporting Time to the User.” *Proceedings of the Third USENIX Mach Symposium*, April 1993.

[Sha90] Sha, Lui, Rajkumar, Ragunathan, and Lehoczky, John P. “Priority Inheritance Protocols: an Approach to Real-Time Synchronization.” *IEEE Transactions on Computers*, Vol. 39, No. 9, Sep 1990.

[Son93] Son, Sang H., George, David W., and Kim, Young-kuk. “Developing a Database System for Time Critical Applications on RT-Mach.” Unpublished.

[Sta88] Stankovic, John A. “Misconceptions About Real-Time Computing: a Serious Problem for Next-Generation Systems.” *IEEE Computer*, Vol. 21, No. 10, October 1988.

[Tok90] Tokuda, Hideyuki, Nakajima, Tatsuo, and Rao, Prithvi. “Real-Time Mach: Towards a Predictable Real-Time System.” *Proceedings of the First USENIX Mach Workshop*, October 1990.

[Tok91] Tokuda, Hideyuki, and Nakajima, Tatsuo. “Evaluation of Real-Time Synchronization in Real-Time Mach.” *Proceedings of the Second USENIX Mach Workshop*, October 1991.