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

Matthew R. Lehr, Young-Kuk Kim and Sang H. Son

Computer Science Department
University of Virginia
Charlottesville, VA 22903, USA

Abstract

Previous work in real-time database management systems (RT-DBMS) has primarily based on simulation. This paper discusses how current real-time technology has been applied to architect an actual RT-DBMS on a real-time microkernel operating system. A real RT-DBMS must confront many practical issues which simulations typically ignore: race conditions, concurrency, and asynchrony. The challenge of constructing a RT-DBMS is divided into three basic problems: dealing with resource contention, dealing with data contention, and enforcing timing constraints. In this paper, we present our approaches to each problem.

1 Introduction

As real-time applications grow more and more complex, so do the ways in which they maintain and access data. As programs are required to manage larger and large volumes of data, they 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, however, 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 provide some degree of fairness. A RT-DBMS, 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.

*This work was supported in part by ONR.

The StarBase RT-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 that 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 goals, StarBase is constructed on top of RT-Mach, a real-time operating system developed at Carnegie Mellon University [11]. StarBase differs from previous RT-DBMS work [1, 2, 3] in that a) it relies on a real-time operating system which provides priority-based scheduling and time-based synchronization, and b) it deals explicitly with data contention and deadline handling in addition to transaction scheduling, the traditional focus of simulation studies.

There are essentially three problems with which RT-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.

1052-8725/95 $04.00 © 1995 IEEE

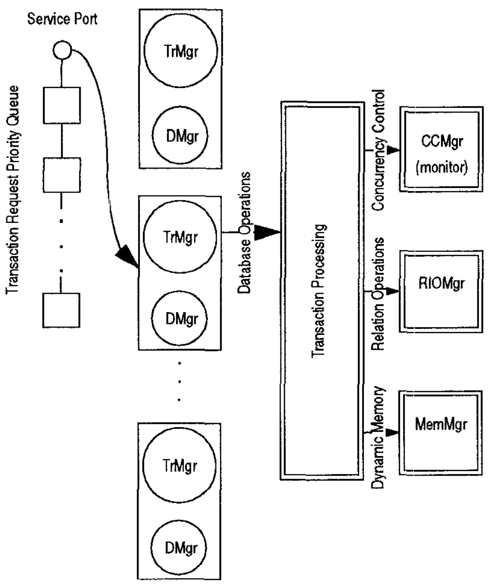

Figure 1: StarBase Server Architecture

2 Database Overview

The StarBase DBMS is organized as a multi-threaded server as shown in Figure 1. It is assumed that database clients are physically disparate from the server, so they pass messages to communicate with the DBMS server. 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.

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 (e.g., read, write) 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's three resource managers provide these services: the Small Memory Manager (MemMgr), the Relation I/O Manager (RIOMgr), and the 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 *optimistic concurrency control* to ensure data consistency, allowing transactions to proceed unhindered until they are ready to apply their updates to the database. The particulars of the concurrency control algorithm are detailed in Section 4.

3 Resource Contention and Transaction Scheduling

For decades the 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. Since they are designed to interact with humans, the goals of conventional systems, by and large, are to achieve fairness and minimize response time. Real-time systems, however, are designed for embedded environments and require quick and predictable behavior in response to external mechanical and electrical stimuli. 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 the highest probability of completing on time.

As with any application, the StarBase RT-DBMS is highly reliant on its native operating system, RT-Mach, to provide the priority-cognizant 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* [7] and the *Basic Priority Inheritance Protocol* (BPI) [9] for non-preemptible resources. With both ideas, 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. RT-Mach provides several priority-based scheduling regimes, including Fixed Priority, Earliest Deadline First, Rate Monotonic, and Deadline Monotonic. RT-Mach's real-time thread model [11] distinguishes real-time threads of execution from ordinary ones, requiring the explicit specification of timing constraints and criticality on a per-thread basis. The timing and priority information is then used as input to the RT-Mach scheduler. RT-Mach also has striven to implement priority-based resource scheduling through its interprocess communication (RT-IPC) [5] and thread synchronization (RT-Sync) [10] facilities. RT-Mach implements BPI itself as a combination of priority queuing and priority inheritance. When a thread blocks on a mutex variable or when a message cannot be immediately received because all potential receivers are busy, RT-Mach queues the waiting thread or message in priority order and then boosts the priority of the thread inside the critical section or the priority of one of the potential receivers in accordance with the BPI protocol.

StarBase employs RT-Mach's priority-based CPU and BPI resource scheduling in several ways: to determine the transaction service order, to provide high-priority transactions the means to progress faster than low-priority transactions, and to provide priority-consistent access to facilities such as the Small Memory Manager and Concurrency Controller. For purposes of uniformity, StarBase adopts the same data type that RT-Mach uses to convey priorities, facilitating the straightforward translation of StarBase to RT-Mach priorities. Since the priority data type, `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. StarBase itself must make priority-based decisions (e.g., concurrency control), so its priority-based comparisons involve priorities expressed as `rt_priority_t`-typed values. Of course, which policy is most appropriate differs from application to application, so the policy to be used is left as a compile-time constant. Naturally, StarBase must use a consistent transaction scheduling policy across all of its priority-based decisions.

3.1 Transaction Service Order

Since performance ultimately degrades as the number of threads of execution in a system increases, 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.

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 [5].

3.2 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.

3.3 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 compete for access to the data stored in the database. To obtain reasonable performance, a DBMS must allow multiple transactions to access data concurrently while requiring that the outcome appears as if it were the result of a serial execution of those transactions. Satisfying these two goals produces a problem which is quite distinct from ordinary contention for operating system resources: contention for data. To resolve data conflicts, StarBase uses a concurrency control implementation which 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 [2]. 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 [2]. Second, Lee and Son devised an improvement to the conflict detection of optimistic concurrency control in general, which StarBase integrates with Haritsa's WAIT-X(S) [6].

4.1 WAIT-X(S)

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*. 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 the conflict set
    and CHP transactions comprise greater
    than X% of the conflict set ) do
  wait;
end
abort the conflict set;
commit the 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 must 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.

4.2 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`, `DeadlineAbort`, 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

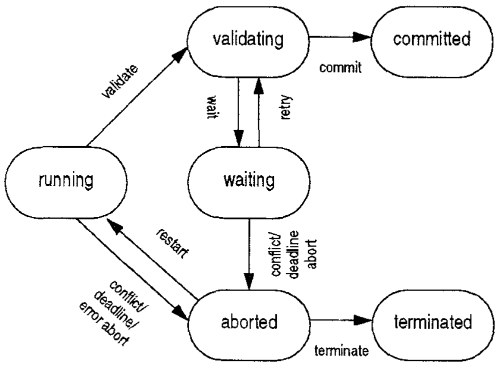

Figure 2: WAIT-X(S) State Diagram

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 (Figure 2).

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.

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, but StarBase ensures at compile time that it is consistent with the CPU scheduling regime under which StarBase is configured to run. 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.

4.3 Precise Serialization

Precise serialization is a conflict-detection scheme for optimistic concurrency control [6]. 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 and decreasing the likelihood of missing transaction deadlines.

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.

4.4 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 tuples 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 *RCCs*). 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 RT-DBMS, it attempts to process the transaction and reply to the application at or before this *firm 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. *Hard deadline* transactions are those transactions whose failure to execute on time is viewed as catastrophic.

5.1 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*, a separate thread of execution which performs application-specific actions when the deadline expires. Typical actions are to abort the thread (firm deadline) or lower its priority (soft deadline). In addition to RT-Mach's real-time threads, implementation of a deadline handler requires time-based synchronization. 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 [8].

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 threads with uniform deadlines. StarBase, since it assumes no *a priori* information about its transaction workload, 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 thread when it misses its deadline so that the thread does not interfere with the handler's execution. If a thread misses its deadline while in the middle of a critical section, it is suspended and cannot leave the critical section until it is resumed. StarBase uses a critical section to resolve potential race conditions between transaction commit (by the transaction manager) and deadline abort (by the deadline manager), so use of a RT-Mach-style 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 in mutual exclusion.

5.2 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 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.

The condition variable `dmgrCancel` is used much differently. The deadline manager must simultaneously block waiting for the deadline to expire or to be cancelled by the transaction manager, whichever comes first. Since RT-Mach provides a time-out on its real-time condition variables, the deadline manager need only wait on `dmgrCancel`, providing the deadline as the time-out value, to block until the deadline. Furthermore, should the transaction manager complete the transaction, it can cancel the deadline manager by signalling on `dmgrCancel`.

The transaction and deadline manager behaviors are presented in Figures 3 and 4. 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

```
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 */
}
```

Figure 3: Transaction Manager

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). Furthermore, real-time CPU and resource scheduling generally make decisions on

```
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);
}
```

Figure 4: Deadline Manager

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

Figure 5: Thread Priority Assignments

which thread to run by simply comparing priorities without quantifying how much they differ. The large priority space and the qualitative priority comparisons allow StarBase to map the external transaction priorities onto new priorities at which the transaction and deadline manager real-time threads actually run.

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, as illustrated in Figure 5.

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. Thus while the timing constraints for the transaction and deadline manager threads are not appreciably different as measured by the hardware clock, scheduling regimes such as Earliest Deadline First will still schedule the deadline manager in preference to the transaction manager.

5.3 Asynchronous Aborts

As previously discussed, firm deadlines are handled asynchronously by a deadline handler which is charged with aborting the thread 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 Conclusions

This paper details the architecture to support a firm RT-DBMS assuming no *a priori* knowledge of transaction workload characteristics. Unlike previous simulation studies, StarBase uses a real-time operating system to provide basic real-time functionality and deals with issues beyond transaction scheduling: resource contention, data contention, and enforcing deadlines. Issues of resource contention are dealt with by employing priority-based CPU and resource scheduling provided by the underlying real-time operating system. Issues of data contention are dealt with by use of a priority-cognizant concurrency control algorithm with a special conflict-detection scheme, called Precise Serialization, to reduce the number of aborts. Issues of deadline-handling are dealt with by constructing deadline handlers which synchronize with the start and end of a transaction and which don't interfere with its execution until the deadline expires.

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 RT-DBMS can preallocate resources and arrange transaction schedules to minimize conflicts, resulting in more predictable service. Execution time estimates and off-line analysis can be used to increase DBMS-wide predictability. Temporal consistency [4], where data used to derive new data must be consistent within a certain validity interval, is also a matter to be explored. Once the basic, real-time, POSIX.4-compliant functionality needed to support a firm real-time database has been established, StarBase can be ported to other platforms.

References

[1] R. Abbott and H. Garcia-Molina. Scheduling Real-Time Transactions: A Performance Evaluation. *ACM Transactions on Database Systems*, 17(3):513–560, September 1992.

[2] J. R. Haritsa. *Transaction Scheduling in Firm Real-Time Database Systems.* PhD thesis, University of Wisconsin–Madison, August 1991.

[3] J. Huang. *Real-Time Transaction Processing: Design, Implementation, and Performance Evaluation.* PhD thesis, University of Massachusetts at Amherst, May 1991.

[4] Young-Kuk Kim. *Predictability and Consistency in Real-Time Transaction Processing.* PhD thesis, Computer Science Department, University of Virginia, May 1995.

[5] T. Kitayama, T. Nakajima, and H. Tokuda. RT-IPC: An IPC Extension for Real-Time Mach. Technical report, Carnegie-Mellon University, August 1993.

[6] J. Lee and S. H. Son. Using Dynamic Adjustment of Serialization Order for Real-Time Database Systems. In *Proc. of the 14th Real-Time Systems Symposium*, pages 66–75, Raleigh-Durham, NC, December 1993.

[7] 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, 1973.

[8] S. Savage and H. Tokuda. Real-Time Mach Timers: Exporting Time to the User. In *Proceedings of the Third USENIX Mach Symposium*, April 1993.

[9] L. Sha, R. Rajkumar, S. H. Son, and C. Chang. A Real-Time Locking Protocol. *IEEE Transactions on Computers*, 40(7):782–800, July 1991.

[10] H. Tokuda and T. Nakajima. Evaluation of Real-Time Synchronization in Real-Time Mach. In *Proc. of the Second USENIX Mach Workshop*, October 1991.

[11] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards Predictable Real-Time Systems. In *Proc. of the USENIX 1990 Mach Workshop*, October 1990.