Real-time Deterministic Database Management

The objectives of deterministic, predictable database management in the context of real-time application design.

McObject LLC
33309 1st Way South
Suite A-208
Federal Way, WA 98003

Phone: 425-888-8505
E-mail: info@mcobject.com
www.mcoother.com

Copyright 2022-2023, McObject LLC
Abstract—This paper will discuss the objectives of deterministic, predictable database management in the context of real-time application design. We will introduce extensions to a conventional database management systems (DBMS) transaction scheduler that add semantics, enforce database transaction priorities and deadline scheduling. We will then focus on the practical aspects of the design and demonstrate its use in several real-life application patterns in a variety of real-time operating system (RTOS) environments.

The Need for a Deterministic Real-time Database Management System

Embedded devices and software have been a staple of modern life for decades. Long gone are the times when the majority of embedded software was created from scratch. The “build vs. buy” alternative is no longer debated — commercial off-the-shelf (COTS) embedded operating systems, database management systems and other middleware and tools are routinely utilized in almost all areas of embedded systems’ development. Moreover, the last decade has seen commercial software making its way into product designs that have always been considered “out-of-reach” of commercial software vendors — avionics, autonomous systems, critical control systems, and the like. These applications’ missions are often non-interruptible and impose stringent requirements on their timely execution. These requirements are often defined as real-time constraints on the systems’ temporal behavior. For example, an action performed too late or a computation that used temporally inconsistent data can be useless, or worse, harmful even if those actions or computations are functionally correct.

Mission- and safety-critical systems’ manufacturers have recognized that using specialized real-time operating systems and components to fulfill critical-path functionality requirements improves the systems’ stability and reduces time to market. Real-time database management systems that preserve the temporal validity of data through time-cognizant transaction processing that guarantees predictable execution of critical transactions needs to be developed and made available commercially.

Overview of Database Management Systems’ Transaction Processing

A transaction is a unit of work with the database (a single logical operation on the data). Database transactions enforce the ACID properties. The ACID model establishes four requirements that a conventional database management system must achieve to be considered ACID-compliant: Atomicity, Consistency, Isolation, and Durability. Not all DBMS implement the ACID properties. There are other paradigms known as eventual consistency, the CAP theorem, and BASE that are beyond the scope of this paper because they are inapplicable to real-time embedded systems and therefore left as an exercise for the reader. Database management systems enforce the ACID properties by requiring that all database access is done within the context of a transaction. The transaction start and commit methods define a transaction block that is applied as a single database operation (atomicity). The transaction rollback method can be called to discard any data updates that were affected after the start without applying those changes to the database.

Concurrency control in database systems is implemented through database transaction managers. Transaction managers ensure that database ACID-compliant transactions are performed concurrently without violating data integrity (consistency and isolation). Concurrency control generally falls into one of two broad categories: Pessimistic concurrency control and optimistic concurrency control. Pessimistic concurrency control implies proactively locking all or parts of a database to grant access to a task. While locked for updates, the (part of the) database is inaccessible by other tasks. Optimistic concurrency control avoids explicit locks by giving each task a copy of the relevant portion of the database content to work with, which allows greater concurrency but at the cost of greater overhead to create, manage, merge and destroy the copies. Partly because these overhead activities are usually non-deterministic, the real-time transaction extensions discussed here are based on a pessimistic concurrency model. More specifically, the model permits multiple concurrent read-only transactions while a read/write transaction has exclusive access to the database. Such models are found in SQLite and eXtremeDB, among others. This approach to concurrency embraces a greatly simplified locking mechanism, eliminating overhead due to resource-consuming complex lock arbitration and deadlock prevention algorithms. These extremely lightweight transaction managers are
ideal for in-memory databases, and for persistent databases with few concurrent transactions that involve create/update/delete operations.

The real-time (time-cognizant) transaction manager extends the pessimistic concurrency model transaction manager with added temporal provisions discussed below.

**Real-time vs. Non-real-time Database Systems**

Like conventional database systems, real-time database systems are data repositories and provide services for storage, retrieval and manipulation of data. The differences between conventional and real-time databases lie in the temporal requirements of the managed data, timing constraints on transactions, and performance goals. The following aspects are often considered:

a) internal versus external consistency constraints and,

b) transaction schedules and performance metrics.

The design principals of conventional (i.e., non-real-time) ACID-compliant database systems always guarantee strong internal data consistency — a consistent view from all components of the database, avoiding contradictory data in the same database. Real-time database system designers sometimes argue that, for real-time databases, external consistency (the requirement for the database to reflect the current physical environment), is preferable. Indeed, external consistency may be more important than internal consistency for certain applications. However, as a practical matter most applications require preserving their databases’ internal consistency.

Furthermore, the common performance metric for all database systems is response time. For conventional database systems the metric comes down to a number of transactions per time unit; this measurement — the average number of transactions per second (TPS) is used heavily in optimizing the responsiveness of traditional database applications. In contrast, real-time database systems often use firm deadline semantics for transactions — transactions can “meet” (successfully commit) or “miss” (successfully abort) their deadlines, but cannot be “late” (go over their allotted time slot) to commit or abort their execution. A late commit of a real-time transaction can lead to a system’s state being intolerable. Thus, a typical performance metric for real-time databases is the number of transactions that violate their timing constraints (miss their deadline).

The design discussed in this article will focus on the timing constraints of transactions and the associated performance metrics while preserving strong internal consistency through enforcing transactions’ ACID properties.

**Design Principles**

One of the most important differences between the database systems used by real-time and non-real-time systems is that while a conventional (non-real-time) DBMS aims to achieve good throughput or average response time, a real-time DBMS must provide a predictable response time to guarantee the completion of time-critical transactions. Therefore, the design of a real-time database system should avoid using techniques that introduce unpredictable latencies. The ability to meet all deadlines requested by all system events is vital to real-time systems. In order to achieve its goal of guaranteed transaction commit or rollback times, the database runtime relies upon the following assertion:

**The time to reverse any modifications to the database made by a transaction up to any point in the transaction does not exceed the time that was required to apply those modifications.**

First consider read-only (RO) transactions that contain operations such as index lookup and cursor movement operations. These read-only transactions modify internal auxiliary structures. Yet the modifications are independent of the actual transaction workload and effectively consist of incrementing and decrementing internal counters. Therefore, the time required to reverse the runtime modifications made by a RO transaction is:
a) independent of the transaction “workload”, and

b) is less than the time the RO transaction has spent searching through the data or traversing the search result.

From a practical point of view, the actual rollback time for RO transactions is best determined empirically.

Read-write (RW) transactions are more complicated. To illustrate our assumption let us examine the key operations that apply modifications to a database and the time required to reverse those modifications. Hereafter, the examples illustrate implementation details of eXtremeDB to convey the real-time database concepts.

Creating a new object. When a new object is created, new pages are allocated out of the free memory pool, and the page gets formatted i.e., object fields are initialized. The free pool is implemented as a linked list, so the time required to allocate pages is proportional to the number of pages necessary to build the object in the database memory, and hence, to the object size. The reverse operation returns pages to the pool and thus has the same temporal complexity (Figure 1).

Fig. 1. Temporal complexity of creating/removing database objects

Updating objects. When a transaction updates an object for the first time, the database allocates space for a temporary copy of the object and then copies the content of the original object to the newly allocated temporary object (copy-on-write). A transaction rollback re-creates the original object from the copy in the reverse order and then releases the allocated memory pages. The allocation and deallocation operations’ temporal complexity is discussed above. The temporal complexity of the update rollback is no greater than the complexity of the update itself (Figure 2).

Removing an object. Objects do not get deleted until the commit stage of the transaction. Instead, the objects are marked as deleted through the use of atomic operations.
**Adding or removing objects into and from indexes.** Adding and removing index elements are inverse operations and for tree and hash indexes have predictable complexity. The temporal indeterminacy for these indexes arises from rebuilding index structures such as rebalancing a tree index or reallocating hash tables. By design, the database runtime eliminates these activities from the rollback operations. Therefore, the reversal of the index update always takes less time than the original modification that might include rebalancing a tree or reallocating hash tables.

![Diagram of a transaction update in eXtremeDB](image)

**Fig. 2.** A transaction update in eXtremeDB

Some transaction patterns allow for ascertaining the worst-case scenario of the transaction rollback time with a greater precision. An interrupted RW transaction must return the data and the database runtime to the state that it had been prior to the start of the transaction. In the worst-case scenario, the application inside the transaction executed data update operations and each update was applied to different objects. Then the time intervals required to apply and to reverse the modifications are equal as seen below:

```plaintext
trans_start()
{
    object1_update();
    object2_update();
    ...
```
However, in practice this is not a prevalent pattern. Even in the context of RW transactions, most applications perform database lookups, and other RO operations and execute code outside the database runtime or yield to other OS tasks. Moreover, multiple modifications can be made to the same database object. The first object update is the most time-consuming because the object copy is created. Other updates are not “free”, but take less time to complete. The rollback time is independent of the number of updates applied to the object:

```c
trans_start()
{
    object1_update();
    if (lookup()== FOUND)
        object2_update();
    read_external_sensor();
    object3_update();
    object1_update(); // subsequent update of same object does not increase rollback time
...
} transaction_end();
```

All these factors help reduce the threshold rollback time. There is no formalized mechanism to determine the minimum threshold and in practice it is set up empirically.

**Implementation**

For ease of reference let us define some terms used throughout the following sections.

- **transaction deadline** is the total time in ticks (time units) allotted to the database transaction to complete and return control to the application.

- **deadline control point** is the time elapsed from the start of the transaction at which the transaction must start its rollback in order to meet the deadline.

- **deadline control point timer** is a timer set by the application to enforce the control point.

- **deadline control point callback** is an application callback to enforce the control point.

- **transaction queue time interval** is the maximum time that the transaction waits for execution in the transaction queue.

- **transaction rollback time interval** means the time to reverse changes made in the transaction.

- **transaction workload time interval** is the time that the transaction spent doing "real payload work".

- **transaction deadline verification checkpoint** are places in the database runtime code at which a transaction’s elapsed time is tested against the set deadline.

- **transaction queue** represents the eXtremeDB runtime internal structure that register all transactions that have been started.
The transaction manager provides a serializable schedule (“read-write” transactions are executed sequentially, “read-only” in parallel) based on the Earliest Deadline First (EDF) scheduling algorithm. The EDF organizes the transaction queue according to each transactions' absolute deadline, such that the transaction with the earliest deadline is scheduled before transactions with later deadline. Since the absolute deadline of a transaction depends on the current time, the EDF scheduling is dynamic: a transaction which has a higher slot in the queue due to an earlier deadline at one point in time may be pushed down the queue if a new transaction with an even earlier deadline enters the queue. While enabling application-defined transaction deadlines and enforcing those deadlines at runtime, applications have the ability to semantically define firm transaction deadlines so that:

a) transactions that do not complete by their deadlines are considered worthless and get discarded, and

b) transactions are interrupted and forced to initiate a rollback with enough time to satisfy their set deadlines

The database transaction manager has a number of verification checkpoints in the code at which the transaction elapsed time is tested against the deadline. The frequency of the verifications eliminates the possibility of executing the database code long enough to miss the set deadline. If the control point is reached (the transaction ran out of the allotted time slice), the transaction is assigned a special “transaction interrupted” status (MCO_E_INTERRUPTED) and the control is returned back to the application. The application is then expected to rollback the transaction. The transaction manager ensures that all database runtime internals are in a “recoverable” condition, and the subsequent database rollback would restore the database to the consistent state that existed prior to the start of the transaction. Furthermore, the transaction manager guarantees that the rollback is completed within the deadline, provided that the application initiates the rollback when signaled to do so by the database runtime. Thus, the transaction would miss the deadline, but the internal consistency of the database is preserved.

The key to supporting real-time transactions is the ability for applications to safely interrupt the execution of the current transaction. Two methods are available: through an asynchronous event handler or via an application callback that is passed to the database runtime and is invoked periodically during a transaction, signaling the application that the deadline control point was reached.

A. The Callback Method

Let us examine the callback method first. The callback method is normally employed when asynchronous primitives such as a system timer or a hardware watchdog are unavailable. Often the application polls a system clock or responds to hardware interrupts, etc. To use a callback, the application registers a callback function with the database runtime. The eXtremeDB runtime provides a standard method for registering various callbacks (pseudo-code is used for clarity):

```c
mco_db_register.callbacks():
typedef MCO_RET (*mco_db_callback_t)(mco_db_h db);
```
The next step is the callback implementation itself. The database runtime requires that the callback returns a ‘success’ return code (MCO_S_OK) if the transaction can continue running and MCO_E_INTERRUPTED return code if the transaction must be aborted. Note that the callback is invoked often, so its implementation should be as “light” as possible on using system resources (similar to the implementation of interrupt service routines).

```c
/* read the current time and establish the transaction control point */
control_point = MCO_SYSTEM_GET_CURRENT_TIME_MSEC() + deadline / 2;
...

static MCO_RET check_deadline(mco_db_h db)
{
    mco_trans_h t;
    /* obtain a pointer to the currently running transaction */
    if (mco_trans_get_current(myDB, &t) == MCO_S_OK) {
        /* check whether the transaction control point was reached */
        if (control_point < MCO_SYSTEM_GET_CURRENT_TIME_MSEC()) {
            /* yes, return error code to indicate that */
            return MCO_E_INTERRUPTED;
        }
    }
    /*
    * the control point has not been reached yet
    * the transaction can continue running
    */
    return MCO_S_OK;
}
```

The next step is to start the real-time transaction

```c
mco_trans_start_deadline(db... deadline, ... &t);
{
    ...
    ...
    /*
    * transaction workload
    */
    rc = transaction_workload();
    /*
    * If any error including MCO_E_INTERRUPTED was detected, during the 
    * transaction, 
    * the following commit call rolls back the transaction. In other words, even an 
    * attempt to commit a transaction will abort the transaction if the transaction 
    * was in an error state.
    */
    rc = mco_trans_commit(t);
}
```

### B. The Timer Method

The first step in using the timer-based transaction control method is to determine the transaction control point. As discussed, setting the control point to half of the deadline interval is often safe (as discussed earlier this measure could be too coarse and hurt the meet/miss deadline ratio). Then the application starts the
timer, setting the timer period to the control point determined in the first step. Installing a timer is operating system-specific and could be simple or complicated. For example, FreeRTOS semantics in pseudo-code:

```c
static mco_db_h connection;
...
void set_sys_timer(sys_timer_t *timer, mco_db_h db, mco_interval_t deadline) {
    ...
    xTimerCreate( "exdb-timer", pdMS_TO_TICKS(deadline/2), pdFALSE, TimerProc );
}
```

Note the “safe” (deadline/2) interval, and that the TimerProc timer callback function is executed when the timer’s period expires. Also note that the timer callback executes in the context of the timer service task. It is therefore essential that timer callback functions never attempt to block. Similar to the callback method, the timer function must set the MCO_E_INTERRUPTED error code, indicating to the application that the transaction has reached the control point.

```c
void TimerProc( TimerHandle_t xTimer )
{
    mco_trans_h t;
    if (mco_trans_get_current(connection, &t) == MCO_S_OK) {
        mco_trans_set_error(t, MCO_E_INTERRUPTED);
    }
}
```

Finally, a transaction is started:

```c
mco_trans_start_deadline(...deadline,.. &t);
...
rc = transaction_workload();
/*
 * If any error including MCO_E_INTERRUPTED was detected, the following
 * commit request rolls back the transaction
 */
rc = mco_trans_commit(t);
```

Note that in contrast to the callback, the timer handler is invoked just once, when the timer expires and then the transaction error flag is set. The database runtime periodically verifies the flag in an atomic operation.

**Priority Inheritance: An Alternative to Earliest Deadline First**

In most use cases, the earliest deadline first (EDF) algorithm meets the needs of real-time systems’ transaction scheduling. That said, real-time systems transaction scheduling needs are not one-size-fits-all. Sometimes, transaction patterns would benefit from a different nuance. A transaction scheduler based on the Priority Inheritance (PI) algorithm can be superior (result in a superior meet/miss ratio).

The major difference between the EDF and PI transaction schedulers is that in the case of the EDF manager, the database kernel organizes the queue and determines transaction execution order based on deadlines. In the PI case, the execution order is first-in-first-out (FIFO) without respect to deadlines. However, PI raises a lower priority task’s priority to that of a higher priority task when the higher priority task begins a database transaction, allowing the
lower priority task to also preempt tasks that are of lower priority than the new high priority task. This, in turn, allows the lower priority transaction to complete quickly and get out of the way of the high priority task.

Ideally, real-time system developers are able to easily alternate between EDF and PI transaction schedulers in order to empirically measure the effect on the meet/miss ratio.

**Putting it Together — RTOS Examples**

The following samples for all operating systems are functionally organized in the same way. The common “driver” section illustrates the code flow for the database initialization (the main() function), the start and the commit of a real-time transaction (the realtime_job() function) and the database access via the operating system-independent database API. The system-dependent code sections implement the timer and the callback functionality. The system-dependent timer-based approach is demonstrated for FreeRTOS™, INTEGRITY OS and VxWorks®, and the callback method is demonstrated for the AUTOSAR architecture and Deos™ RTOS. The source code is written in C language and omits details for the sake of clarity.

```c
#ifndef MCO_REALTIME_TIMER_H
#define MCO_REALTIME_TIMER_H
#if defined(_LINUX) || defined (_MACOS) /* POSIX */
#include <sys/time.h>
typedef struct itimerval sys_timer_t;
#endif

void timer_handler (mco_db_h db);
void init_sys_timer();
void clear_sys_timer(sys_timer_t *timer);
void set_sys_timer(sys_timer_t *timer, mco_db_h db, mco_interval_t deadline);
#endif
```

Fig. 4
The code fragments above (Figures 4, 5, 6) illustrate the operating system-independent “common sections” for the timer-based implementation.

Figure 7 illustrates the VxWorks implementation via the watchdog timer facility. Any task can create a watchdog timer and use it to run a specified routine in the context of the system-clock ISR, after a specified delay. The hardware timer that drives the watchdog timers generates interrupts at a programmable rate and the associated interrupt service routine scans each timer, decrements counters associated with running timers, and calls the timeout routine associated with any watchdog timers that are found to have expired.
The FreeRTOS implementation of the control timer (Figure 8) uses the software timers service. There is a dedicated ‘Tmr Svc’ (Timer Service or Daemon) task which maintains an ordered list of software timers, with the timer to expire next in front of the list. The Timer Service task is not continuously running: from the Timer List the task knows the time when it must wake up each time a timer in the timer list has expired. When a timer has expired, the Timer Service task calls its callback (the Timer callback). FreeRTOS software timers support two different types of software timers, configured at timer creation time: One-Shot timer such that when the timer expires, it is not restarted again and an Auto-Reload timer that will be automatically restarted when it expires. The sample code runs a one-shot software timer to control the deadline of the transactions. It is required, so FreeRTOS itself must be configured to support the software timers (#define configUSE_TIMERS 1 in FreeRTOSConfig.h) and be given enough priority to serve the events on time.
The **INTEGRITY OS** implementation (Figure 9) uses HighestResStandardClock and Alarm to implement the control timer. A Clock is an Object that allows access to a hardware timer. Clocks can be used to read the time, set the time, or set an alarm. Each Clock possesses its own alarm. This allows multiple alarms to be set on the same hardware timer. In practice, the kernel multiplexes the many alarms onto the hardware timer’s single alarm. HighestResStandardClock is the standard tick timer that is available on every board. This timer is run off of the scheduler tick, which is hardware specific. The Alarm can be set to go off either at a specified time or after the passing of a specified interval. The Alarm can be specified as repeating, in which case after the Alarm goes off another Alarm is automatically set after a specified interval without the program explicitly setting a new Alarm. According to INTEGRITY’s message-based architecture, an Alarm is structured by the kernel like a send of an empty message on its Clock Object. This allows Alarms to be obtained either synchronously or asynchronously. An Alarm is obtained synchronously via a call to SynchronousReceive(), wherein the obtaining Task blocks until the Alarm goes off. An Alarm is obtained asynchronously via a call to AsynchronousReceive(), wherein the obtaining Task can continue about its business until the Alarm goes off, at which point the Task is notified.

The sample code utilizes a synchronous Alarm set on the HighestResStandardClock to implement the timer.

```c
#include <mco.h>
#include "common.h"
#include "timer_implementation.h"

/* system timer operations */
#if defined(_FREERTOS)
#include <FreeRTOS.h>
static mco_db_h myDB;

void TimerProc( TimerHandle_t xTimer ) {
    timer_handler( myDB );
}
void init_sys_timer() {
}

void set_sys_timer(sys_timer_t *timer, mco_db_h db, mco_interval_t deadline) {
    /* Keep the reference to the current database connection */
    myDB = db;
    xTimerCreate( "eXdb-timer", pdMS_TO_TICKS(deadline/2), pdFALSE, TimerProc );
}

void clear_sys_timer(sys_timer_t *timer) {
}
#endif
```

**Fig. 8**
```c
#include <INTEGRITY.h>

Clock AlarmClock;
Task TimerTask;
volatile int stop_timer_task = 0;
mco_db_h timer_connection = 0;

static int TimerProc(Address param) {
    do {
    SynchronousReceive( (Connection)AlarmClock, NULL );
    /* Call the timer handler */
    timer_handler( * ((mco_db_h*)param) );
} while (!stop_timer_task);
return 0;
}

int init_sys_timer() {
    Error E;
    Value P;
    Value W;
    Time res;
    Time AI;
    timer_unit tm;

    E = GetClockResolution(HighestResStandardClock, &res);
    if (E == Success) {
        printf("Timer resolution is %.6f microsec\n", res.Seconds * 1000000.0 +
               res.Fraction / (0xFFFFFFFF/1000000.0) );
    } else {
        printf("GetClockResolution() failed, %u\n", E);
    }

    printf("Testing mco_system_get_current_time()...
");
    tm = mco_system_get_current_time();
    AI.Seconds = 0;
    AI.Fraction = (0xFFFFFFFF/1000) * 123;
    E = SetClockAlarm(HighestResStandardClock, false, NULLTime, &AI);
    if (E != Success) {
        printf("SetClockAlarm(...) == %u\n", E);
    } else {
        E = SynchronousReceive( (Connection)HighestResStandardClock, NULL );
        if (E != Success) {
            printf("SynchronousReceive(...) == %u\n", E);
        }
    }
    tm = mco_system_get_current_time()-tm;
    printf("Sleeping for 123000 microsec took %u microsec\n", tm);

    E = CreateVirtualClock(HighestResStandardClock, CLOCK_READTIME|CLOCK_ALARM,
                           &AlarmClock);
    if (E != Success) {
        printf("CreateVirtualClock(...) == %u\n", E);
        return 0;
    }
```

Fig. 9, pt. 1 of 3
/* setup the task */
E = GetPriorityAndWeight( CurrentTask(), &P, &W );
if ( E == Success ) {

E = CommonCreateTaskWithArgument ( P, (TASKENTRYPOINT)TimerProc, (Address)&timer_connection, 1024, "eXdb-Timer", &TimerTask); /* use
CommonCloseTask()*/
if ( E != Success ) {
    printf("CommonCreateTask(...) == %u\n", E);
    return 0;
}

SetPriorityAndWeight( TimerTask, P, W, true);
RunTask(TimerTask);
} else {
    printf("GetPriorityAndWeight( CurrentTask(), ...) == %u\n", E);
    return 0;
}
return 1;

}

void shutdown_sys_timer() {
    Error E;
    Time  AI;
    AI.Seconds  = 0;
    AI.Fraction = 1;
    stop_timer_task = 1;
    E = SetClockAlarm( AlarmClock, false, NULLTime, &AI );
    if ( E != Success ) {
        printf("SetClockAlarm(...) == %u\n", E);
    }
    SynchronousReceive( (Connection)TimerTask, NULL );
    CommonCloseTask( TimerTask );
    CloseClock( AlarmClock );
}
int set_sys_timer(sys_timer_t *timer, mco_db_h db, mco_interval_t deadline) {
    Error E;
    Time AI;
    Time increment;

    deadline /= 2;

    /* Set the alarm */
    AI.Seconds = deadline / 1000;
    AI.Fraction = (0xFFFFFFFF/1000) * (deadline % 1000);

    E = GetClockResolution(AlarmClock, &increment);
    if (E != Success) {
        printf("GetClockResolution(...) == %u\n", E);
        return 0;
    }

    if ( (increment.Seconds * 1000 + increment.Fraction / (0xFFFFFFFF/1000)) <
        deadline ) {
        timer_connection = db;

        E = SetClockAlarm(AlarmClock, false, NULLTime, &AI);
        if (E != Success) {
            printf("SetClockAlarm(...) == %u\n", E);
            return 0;
        }
    } else {
        printf("Timer resolution %.6f microsec is not enough for the deadline %umsec\n", 1000000.0 * increment.Seconds + increment.Fraction / (0xFFFFFFFF/1000000),
            deadline*2);
        return 0;
    }

    return 1;
}

void clear_sys_timer(sys_timer_t *timer) {
    Value overruns;
    Error E;
    Time AI;

    AI.Seconds = 0;
    AI.Fraction = 1;

    E = SetClockAlarm(AlarmClock, false, NULLTime, &AI);
    if (E != Success) {
        printf("SetClockAlarm(...) == %u\n", E);
    }

    E = ClearClockAlarmOverruns(AlarmClock, &overruns);
    if (E != Success) {
        printf("ClearClockAlarmOverruns(...) == %u\n", E);
    }
}

#if 1

Fig. 9, pt. 3 of 3
The examples for the AUTOSAR architecture and Deos RTOS utilize the callback method. There is a small difference between handling the callback in different operating environments. Some OS-es, like Deos, have the ability to read the wall time, while others, such as AUTOSAR can only support the concept of elapsed time. Omitting details, the following code fragments illustrate the database callback registration and the real-time transaction code flow.

By default, Deos uses the rate-monotonic scheduling algorithm (RMA) to schedule processes. In essence the RMA algorithm simply says that the more frequently a task runs (the higher its frequency), the higher its priority should be. In Deos each process is assigned an execution rate and a CPU budget. Tasks that execute at a higher rate preempt ones that execute at a lower rate. Tasks are expected to complete their execution by explicitly yielding before they run out of their CPU budget. Thus, a task running a database transaction is expected to complete (commit or rollback) the transaction before it runs out of its CPU budget (see Figure 10).

```c
static MCO_RET check_deadline(mco_db_h db)
{
    mco_trans_h t;
    if mco_trans_get_current(myDB, &t) == MCO_S_OK) {
        /* Read the current time and compare with the deadline */
        if (callback_deadline < MCO_SYSTEM_GET_CURRENT_TIME_MSEC()) {
            return MCO_E_INTERRUPTED;
        }
        return MCO_S_OK;
    }
}
int realtime_job(.. db, mco_interval_t deadline)
{
    callback_deadline = MCO_SYSTEM_GET_CURRENT_TIME_MSEC() + deadline / 2;
    mco_trans_start_deadline(..., MCO_READ_WRITE, deadline,..., &t);
    /* read external data and store it in the database */
    read_external_data(&id, &val);
    if (rc == MCO_S_OK) {
        Measurement_id_put(&obj, id);
        Measurement_val_put(&obj, val);
    }
    rc = mco_trans_commit(t);
    return rc;
}
int main ()
{
    int deadline;
    ... register callback(database, check_callback);
    realtime_job(...,deadline...);
    close_database();
    return 0;
}
```

Fig. 10
The **AUTOSAR** implementation (Figure 11) of the control callback makes use of the Time Service module which is a part of the Services Layer. Among other services, the module provides Time measurement services for time-based functionality. Several timer types - so called “Time Service Predef Timers” - are available, if supported by the underlying hardware and enabled in the system. Each Predef Timer has a predefined tick duration (physical time unit) and a predefined number of bits (physical range). If a user wants to implement time-based functionality, no user-specific configuration of the Time Service module is necessary. The program can instantiate any timers (only limited by available memory) and can use the timer instances completely independently. So, hardware timers are reused. All services are called by the program (polling mode). Notifications are not supported.

Each Time Service Predef Timer has its own set of API services which provide simple functionality like a stopwatch: ResetTimer, GetTimeSpan, etc. The sample code uses those services and to mark the time intervals and implement the control callback.
static MCO_RET check_deadline(mco_db_h db) {
    mco_trans_h t;
    timer_unit trans_time;

    if (mco_trans_get_current(myDB, &t) == MCO_S_OK) {

        /* Read the time passed after the last timer reset */
        Tm_GetTimeSpan1us32bit ( deadline_timer, &trans_time );
        if ( callback_deadline < trans_time )
            return MCO_E_INTERRUPTED;
    }
    return MCO_S_OK;
}

int realtime_job(... db, mco_interval_t deadline) {
    ...

    callback_deadline = deadline / 2;

    /* reset the timer and start the transaction */
    Tm_ResetTimer1us32bit( deadline_timer );
    mco_trans_start_deadline(..., MCO_READ_WRITE, deadline,..., &t);

    /* read external data and store it in the database */
    read_external_data(&id, &val);
    rc = Measurement_new(t, &obj);
    if (rc == MCO_S_OK) {
        Measurement_id_put(&obj, id);
        Measurement_val_put(&obj, val);
    }
    rc = mco_trans_commit(t);

    return rc;
}

int main () {
    int deadline;
    ...
    register callback(database, check_callback);

    realtime_job(...,deadline...);

    close_database();
    return 0;
}

Fig. 11
Future Work

Improving the current EDF scheduling implementation to allow preemption based on criteria other than just the transaction deadline value.

A few things that are considered to form the preemption criteria:

- We propose adding an explicit priority parameter to the transaction. Higher priority transactions get the CPU earlier than those with lower priority and a set of rules regulating preemption would be required.

- Consider the time left for the current transaction to complete (based on the deadline)

- Consider the time necessary to abort the current transaction (the deadline/2 by default)

External consistency considerations.

As discussed earlier, for some applications keeping the database external consistency could be more important than other factors. Indeed, a real-time DBMS supports transactions that reflect real world events and as long this reflection is consistent with the demands of the real world, whether the database is internally consistent or not could be irrelevant. Adding external consistency provisions at the expense of relaxing the ACID properties is an option to consider.

Conclusion

New generations of avionics, process control, and similar systems are demanding data management solutions capable of accurately reflecting the state of the environment they control or monitor. Conventional database management systems often employ a wide range of efficient but unpredictable algorithms and scheduling policies that result in timing variability in accessing the data. The need for advanced data management suitable to be deployed in applications with explicit time constraints creates conceptual and engineering challenges for researchers and commercial vendors alike. To meet the fundamental requirements of predictability and temporal consistency of data, DBMS vendors need to look into replacing conventional concurrency control and scheduling methods and instead consider concepts that are ignored in non-real-time database systems.