Implementation of a Transient-Fault-Tolerance Scheme on DEOS
— A Technology Transfer from an Academic System to an Industrial System*

Libin Dong, Rami Melhem, Daniel Mossé
Dept of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
{dong, melhem, mosse}@cs.pitt.edu

Sunondo Ghosh, Walt Heimerdinger, Aaron Larson
Honeywell Technology Center
Minneapolis, MN 55418
{sghosh, walt, alarson}@htc.honeywell.com

Abstract

Fault tolerance is important in real-time systems where correct execution of tasks must satisfy certain temporal constraints. Since transient faults occur more frequently than permanent faults, this paper focuses on a transient-fault-tolerance algorithm, FTRMS. The FORTS group of the University of Pittsburgh derived the FTRMS algorithm and implemented it on FT-RT-Mach. The same mechanism is transferred to a commercial system, DEOS, at the Honeywell Technology Center. By describing and contrasting the implementation of FTRMS on the two systems, this paper illustrates the difference between an academic system and an industrial system. An application example shows the effect of the new transient fault tolerance scheme on the system.

1 Introduction

In recent years, real-time systems have been used in many time-critical applications, such as industrial process controls and aircraft autopilots. The tasks in such applications have stringent timing constraints. They must generate correct results before their deadlines even in the presence of faults. As a result, the ability to tolerate faults is a critical need in real-time systems.

Faults can be either permanent, intermittent or transient in nature. Permanent faults are caused by the complete failure of a hardware component, and are typically tolerated by using hardware redundancy. Transient faults are short-lived malfunctions of a computing component that cause an incorrect result. Intermittent faults are repeated occurrences of transient faults.

The possible causes of transient faults include limitations in the accuracy of electromechanical devices, electromagnetic radiation received by interconnections (such as long buses acting like receiving antennas), power fluctuations not properly filtered by the power supply, and the effects of ionizing radiation on semiconductor devices [2]. Our focus is on a *transient fault tolerance (TFT)* scheme because it has been shown that transient faults are significantly more frequent than permanent faults [2, 6]. For example, in [10], measurements showed that transient faults are 30 times more frequent than permanent faults. Transient faults in real-time systems are generally tolerated using time redundancy, which involves the re-execution of a task if a transient fault occurred during its first execution. This is a relatively inexpensive method of providing fault-tolerance.

In real-time systems, scheduling techniques are used to provide a predictable response time, and an admission control condition is applied to guarantee that the real-time tasks will meet their deadlines once they are admitted. The Rate Monotonic Scheduling (RMS) algorithm [8] is used in a variety of critical applications ranging from navigation of satellites to avionics. In order to guarantee the timing constraints of all the tasks in case of faults, a Fault-Tolerant Rate Monotonic Scheduling (FTRMS) scheme was derived to add TFT to the original RMS algorithm [5] .

RT-Mach is a micro-kernel-based real-time operating system, developed at the Carnegie Mellon University. The FORTS group at the University of Pittsburgh added a TFT scheme to RT-Mach by applying the FTRMS algorithm. Some application examples implemented on the new system, *FT-RT-Mach* have shown that it can effectively schedule critical real-time tasks while including time for recovery from transient faults [3]. In order to explore whether this scheme is practical and applicable to real world applications, we implemented FTRMS on a commercial real-time operating system, the Digital Engineering Operating System^{TM} (DEOS), developed by Honeywell initially for avionics applications.

This paper describes and contrasts the implementation of FTRMS on the two systems, especially dealing with admission control, fault detection, fault recovery and state information maintenance (check-pointing). We also analyze

*This work was supported in part by DARPA contract DABT63-96-C0044. The technology transfer was accomplished when the first author was a summer intern at the Honeywell Technology Center.

the design differences between an academic system and an industrial embedded system.

2 System Description

Both FT-RT-Mach and DEOS support real-time computing. We first present an overview of the two systems without TFT functionalities, focusing on their real-time capabilities. This provides the foundation for the description of the TFT implementation on the two systems.

2.1 FT-RT-Mach

The objective of FT-RT-Mach is to support a predictable real-time computing environment. It is based on a kernel which is responsible for thread management, thread scheduling, real-time synchronization, and real-time interprocess communication [9]. In this paper, we will emphasize thread management and scheduling, which affect the implementation of our TFT scheme.

2.1.1 FT-RT-Mach Thread Management

The thread is the finest-grained execution unit in FT-RT-Mach. FT-RT-Mach supports a variety of thread types, and provides a uniform system interface to all kinds of threads. In order to create a certain thread, say T_i, the thread attributes are specified, including the following ones [9].

- *Real-time thread or non-real-time thread.*
- *Periodic or aperiodic.*
- *A priority value.* It is used if the fixed priority scheduler is used in the system.
- *Timing attributes.* These include the worst case execution time C_i, the start time of the thread S_i, the period of a periodic thread P_i or the deadline of an aperiodic thread D_i, and the abort time A_i if the thread is soft real-time.

The thread also needs to specify a stack descriptor and a procedure name. The stack descriptor defines the size and the address of the thread local stack region. In the case of real-time periodic threads, the procedure describes the execution of the thread instance. It is invoked and terminated in every period by the kernel. Since the procedure does not need any FT-RT-Mach related control primitives for its own purpose, the application programmers who write the thread procedures do not need to know about the RT properties.

After defining the attributes, the *rt_thread_create* primitive is called to create a structure of the new thread and allocate user and kernel stack regions according to the stack descriptor. The kernel also creates and arms a wake-up-timer and a deadline-timer for the new thread. The wake-up-timer generates an interrupt at the next ready time of the periodic thread, and the deadline-timer generates an interrupt if the thread overruns its deadline. At the beginning of every thread period, the wake-up-timer expires and the new instance of the thread is placed in the ready queue.

When a thread terminates its execution, it is removed from the ready queue. The *exit* primitive resets its wake-up-timer and deadline-timer to the beginning and the end of the next period, respectively. Moreover, the SP (stack pointer) is reset to the top of the stack and the IP (instruction pointer) is reset to the beginning of the thread procedure, so the next execution of the thread begins with a new context.

Although the FT-RT-Mach thread model makes it easy for the application programmer to write the thread procedure, it has two disadvantages. Since the SP is reset and the execution stack is cleared at each invocation of a thread instance, the new instance cannot access the previous results, unless global memory is used. Also, since the IP is reset by the kernel to the beginning of the procedure for all the instance invocations, it becomes cumbersome to define some special behavior for a particular instance, such as initializing the local variables in the first instance.

2.1.2 FT-RT-Mach Thread Scheduling

FT-RT-Mach provides an integrated time-driven scheduler (ITDS), whose objective is to provide predictability, flexibility and modifiability for managing both hard and soft real-time activities [11]. ITDS provides an application with a set of scheduling policies, including RR (Round Robin), TL (Time-Line), FP (Fixed Priority), RM (Rate Monotonic), RM/DS (RM with Deferrable Server) and RM/SS (RM with Sporadic Server). In order to predict whether a given thread set can meet its deadline or not, FT-RT-Mach adopts some well-known schedulability analysis schemes for several scheduling policies. For example, it provides the RMS and EDF schedulability bounds for a set of periodic, independent threads in a single processor environment, based on [8].

2.2 Digital Engine Operating System (DEOS)

DEOS is a micro-kernel based, general purpose, deterministic hard real-time operating system certified for use in commercial airplanes. [1] DEOS supports space partitioning (guarantees that one process cannot violate another's memory space) at a process level and time partitioning (guarantees that one thread cannot violate the timing guarantees

[1] More precisely, DEOS has appeared as a component of a certified aircraft. DEOS has not been certified per se, since the FAA and other certification authorities do not certify stand-alone software.

of another thread) at a thread level. Other features include standard services for mutexes (Highest Locker Protocol), mailboxes, semaphores, shared memory, and events.

2.2.1 DEOS Thread Management

DEOS was built around a periodic thread model. All threads are assigned a rate and a CPU budget for that period. The CPU budget is the same as the worst case execution time. Periodic threads run until they have consumed their CPU budget, and then are suspended until their next period. A combination of fixed budget and slack scheduling [1] applied on the periodic thread model can be used to achieve a wide range of hard, soft, and non-real-time responses.

DEOS requires application threads to be *harmonic*. Two periods are defined to be harmonic if the length of the larger period is an integer multiple of the smaller one. A set of threads is said to be harmonic if their periods are all harmonic.

Like FT-RT-Mach, a DEOS application also needs a stack descriptor and a procedure name for a thread. On thread creation, DEOS creates a stack and initializes the thread IP to the beginning of the thread procedure. However, the meaning of the thread procedure is different from FT-RT-Mach. In FT-RT-Mach, the procedure describes the behavior of every thread instance, and is terminated at the end of each instance. In DEOS, the procedure describes the whole life of the thread. A typical thread is written as an infinite loop; once the procedure is terminated, the thread is destroyed.

In a DEOS thread procedure, a system call *waitUntilNextPeriod* suspends the execution of the current instance. The thread will not be invoked until the next period, when the thread resumes its execution from the first statement following the *waitUntilNextPeriod* call in the procedure. This architecture accommodates the standard initialization, normal execution, and termination modes of a thread used by most applications in a very natural way. Compared with the thread model of FT-RT-Mach, it has two advantages. First, since the thread procedure never terminates and the SP is not controlled by the thread management, the computation results of the previous instance can be directly accessed from the stack. Second, the usage of *waitUntilNextPeriod* makes it easy to define different functionalities for different instances. It gives the application programmer more freedom to describe some special instances. The pseudo-code example in Figure 1 illustrates the convenient usage.

2.2.2 DEOS Thread Scheduling

DEOS uses the RMS algorithm to schedule periodic real-time threads. For threads with the same period, DEOS permits the applications to specify a partial order (the "schedule before" directive) to schedule their execution. The objective of the partial order is to reduce the end-to-end latency for data passed between threads in the same period.

```
procedure thread1(argument)
{
    data definition
    {
        behavior of the 1st instance
        waitUntilNextPeriod()
    }
    {
        behavior of the 2nd instance
        waitUntilNextPeriod()
    }
    while (1) {
        periodic behavior of all later instances
        waitUntilNextPeriod()
    }
}
```

Figure 1. A Procedure Example in DEOS

Because of the harmonic feature of the thread set, DEOS provides the near-100% processor utilization. The restriction to harmonic periods simplifies the admission tests, general implementation, and furthermore, matches the periodic nature of most applications in embedded control systems.

2.2.3 DEOS Application Registry

DEOS requires the use of an off line database called the *registry* to specify kernel object attributes and all timing parameters, and to check their validity before the application is submitted to the system. So when an application is submitted in the system, almost all of the application parameters such as the set of harmonic periods allowed for all the threads have been fixed in the registry information.

The registry is used to prevent the kernel from breaking down because of some invalid kernel object attributes, and to prevent some critical thread from being rejected at run time because of unacceptable parameters. Another important purpose of the registry is to isolate applications from platform dependent parameters. For example, the value of a thread budget changes when porting from one platform to another. In an environment that involves certification, it is highly desirable *not* to change source code, or even re-link executable files, because any change would involve additional verification costs. In DEOS, time and space partitioning, the usage of registry, and various other functionalities are instrumental in reducing verification costs and increasing the robustness of the system.

3 Need for Transient-Fault-Tolerance (TFT) in DEOS

DEOS provides the infrastructure for applications requiring a high degree of dependability in the face of both transient and permanent faults. For example, the Primus Epic^{TM} system [4] described below already includes a number of fault tolerance mechanisms that can be enhanced by our TFT mechanisms.

3.1 Fault tolerance mechanisms in Primus Epic

Primus Epic is an avionics system developed by Honeywell for commuter aircraft and business jets. A variety of applications such as the flight management system (FMS), autopilot, displays and utility functions run on the Primus Epic system. This system uses several processing modules called Modular Avionics Units (MAUs) each of which uses DEOS as its operating system.

Faults are usually tolerated through the use of hardware redundancy. All critical applications have one or more backups running concurrently on multiple MAUs (hot spares). Faults in MAUs are detected using a combination of techniques including watchdog timers, software monitors, etc. If one MAU fails, control is transferred to a different copy of the application running on an alternate MAU, and the system continues normal operation. The fault tolerance mechanisms in Primus Epic generally target permanent faults but can also be used to recover from transient faults. However, as explained later, these mechanisms involve a substantial reconfiguration of system resources and in some circumstances can be visible to the user.

3.2 Benefits of targeting transient faults

As mentioned earlier, many faults encountered are known to be transient in nature. It is very rare that a processor, such as an MAU in Primus Epic, will fail completely, while it is more likely that a transient fault will occur causing incorrect computations or process failures.

Recovery from transient faults can be achieved by restoring the original state of the affected thread (through checkpointing) and re-executing it with the same input data as provided before the fault. To tolerate a wider class of transient faults, an alternate algorithm can be used if the primary thread fails. The mechanisms used to recover from major permanent faults are more expensive than necessary for such transient faults, and hence we have explored the possibility of adding a much more limited fault tolerance mechanism to deal with these transient faults. The benefits of providing recovery mechanisms specifically designed to tolerate transient faults are many:

- **Faster recovery time and improved availability:** For transient faults, time redundancy can provide faster recovery than hardware redundancy. If hot spares are used as in Primus Epic, a fault will cause a switch in control to the spare, and then the faulty processor must be restarted, re-initialized, and must finally catch up to the other processor. If transient faults can be tolerated on the primary processor, the availability of the system also improves.

- **Limited impact:** The TFT mechanisms can be used to identify the specific thread or process affected by a transient fault. If such techniques are not used, then the whole processor may be considered faulty thus shutting down all threads and processes running on that processor.

- **Reduced risk:** There is an inherent risk associated with switching control from one processor to another. Also, the faulty processor must be recovered using state data from the other processor thus causing further risk until the faulty processor is up and running independently of the other processor. All these risks can be mitigated by tolerating transient faults on the same processor.

- **Reduce rate of threads:** In many control applications, the rate of a thread is set much higher than needed to make sure a few missed deadlines do not severely impact the system. If recovery time is guaranteed to the thread within its period when required, then the rate of the thread may be reduced, thus allowing more threads to be scheduled in the system.

- **Pilot comfort:** When the control switches from one processor to another, the pilot may observe the switchover, which may reduce satisfaction with the system.

- **Lower hardware requirements and potential cost savings:** The hot spares used in any fault-tolerant system are ideal for processor failures, but may be overkill for transient faults. Transient faults can be tolerated by using time redundancy thus reducing the hardware redundancy requirements. If the number of faults being tolerated using extra hardware decreases, some of the processors in the system may be eliminated. This may result in some cost savings.

- **Potential for single string systems and systems using a small number of processors:** DEOS can potentially be used in less critical systems where only a single processor is available (single string systems) or a small number of processors are being used, and there are no backups available for many threads. These systems either have lower availability requirements and thus do not need the high cost of hardware redundancy or they have stringent weight and volume constraints (such as space systems) and need to manage with a small number of processors. In such systems, it may be useful to tolerate transient faults which would significantly improve their availability.

3.3 Objective of TFT

The objective of the TFT scheme described in this paper is to ensure that FT threads produce correct output even in the presence of transient faults. Note that the TFT scheme cannot tolerate all transient faults, for example, transient faults affecting the operation of the operating system or the thread code being executed cannot be tolerated. These can be treated as permanent faults, and tolerated using other fault tolerance mechanisms already existing in the system. However, we believe that a number of transient faults can be tolerated by re-executing or executing an alternate version of the thread. We call the class of transient faults that can be tolerated through TFT *correctable transient faults*.

Thus, TFT aims to provide the following attributes to all threads that require FT in DEOS:

- Data integrity: through restoration of state and re-execution of the thread.
- Timely execution: by reserving time for re-execution *a priori*.

This problem of providing data integrity in a timely manner can be converted into a scheduling problem where the schedule guarantees recovery from correctable transient faults by providing sufficient time for re-execution.

4 FTRMS Scheme Implemented on FT-RT-Mach

To provide TFT functionality applying FTRMS, FT-RT-Mach has the following three parts in addition to the original RT-Mach kernel: the FTRMS admission control scheme, fault detection scheme, and fault recovery scheme. Once a new periodic thread satisfies the admission control condition, it is guaranteed to meet its deadline, even in presence of transient faults, as long as the assumed fault model is satisfied. When there is no fault, the kernel uses the original RMS to schedule the currently active threads in the system. At the end of every thread execution, a fault detection mechanism determines if a fault has occurred during the thread execution. If so, the thread is re-executed in the same period using a certain recovery scheme. This general idea is applied on both FT-RT-Mach and DEOS. But due to the different characteristics of the two systems, the implementation issues are different, which will be shown in the later sections.

4.1 Fault Model

In order to guarantee the re-execution of a thread to meet its deadline, we define the fault model by the following assumptions:

A1 Transient faults are at least ΔF time units apart. It means that two faults cannot occur within a ΔF time interval and at most one thread can experience the fault in this interval.

A2 ΔF is greater than the largest period of all the thread periods. That is, $\forall i, \Delta F > P_i$. This means that the transient fault will not affect the re-execution of a faulty thread.

Most of the time, the above two assumptions are true in the real world, because a system with frequent faults will not be used to support critical real-time applications.

4.2 Admission Control

Given a set of n periodic threads $\{T_1, ..., T_n\}$, let U_i be the utilization of T_i, that is, $U_i = C_i/P_i$. Liu and Layland have proven that a set of periodic threads scheduled using the original RMS scheduling algorithm will meet their deadlines if the summation of each thread utilization is not larger than a certain threshold [8]. However, in order to tolerate transient faults by re-executing the thread within the same period, the system needs to reserve some extra utilization for the re-execution, such that the timing constraints are not violated in case of a transient fault. It is proven in [5] that the set of threads are guaranteed to meet their deadlines under the fault model assumptions if the following condition is satisfied.

$$\sum_{i=1}^{n} U_i \leq U_{LL}(1 - U_{max}) \qquad (1)$$

In Equation (1), U_{LL} denotes the Liu and Layland bound, and U_{max} is the maximum thread utilization among all threads being protected from faults. A new thread is rejected if the above condition is violated.

4.3 Fault Detection

Fault detection can be implemented at the application level, kernel level, or hardware level. In FT-RT-Mach, a variable, *fault flag*, is added to the kernel to indicate the appearance of a fault. This *fault flag* can be set by an exception handler or by a user thread, such as a fault injecting thread for testing purposes. When a fault occurs, *fault flag* is set to be equal to the current thread id. Upon termination of a thread instance, the *exit* primitive first checks whether *fault flag* is equal to its own thread id, which allows the fault to be recognized at the end of the thread execution.

4.4 Fault Recovery

The *exit* primitive terminates the thread normally as described in Section 2.1.1 if the *fault flag* indicates no fault occurrence. Otherwise, it starts re-execution immediately by setting the wake-up-timer to the current time. In this case, the deadline-timer is not changed but remains equal to the finishing time of the current period, indicating that the re-execution must be finished within the current period.

Since the re-execution of the faulty thread must not influence other threads' timing constraints, it cannot be executed at its original priority. Thus, the re-execution is not scheduled using the traditional RMS algorithm, though all the other threads still are. [5] shows that by reserving U_{max} utilization for re-execution, there is $U_{max}L$ backup slack available during any interval of length L. So there is a certain amount of slack between every two consecutive period boundaries, where a period boundary is the beginning of any thread period. When there is no fault, the threads are executed using RMS scheduling, such that the slack can be envisioned as being swapped with the execution of a thread. Thus, the scattered slack is effectively shifted and accumulated following the thread. When a fault occurs, the thread reclaims the slack in the current period for re-execution. It is proven in [5] that there is enough slack for re-execution. The following steps describes the fault recovery procedure of reclaiming slack for re-execution.

1. When a thread has detected a fault at the end of its execution, it immediately starts re-execution with the highest priority for S time units, where S is equal to the amount of slack that has been swapped with the thread execution. Assume the period of the faulty thread instance begins at time t_0, and the earliest period boundary after the primary execution is at time t_1. Then S is calculated as $S = (t_1 - t_0)U_{max}$. The faulty thread is suspended after re-executing for S time units, and all the other threads are scheduled using the RMS algorithm.

2. Whenever a wake-up-timer goes off indicating a period boundary, say, at time t_2, the faulty thread is scheduled to run at the highest priority for S' time units and suspended, where $S' = (t_3 - t_2)$ and t_3 is the time of the next period boundary.

3. Step 2 is repeated until the re-execution is finished.

The following figures illustrate how the re-execution is carried out. In figure 2 it is assumed that a fault occurred during the execution of the j^{th} instance of a thread, T_f. The black box represents the primary (faulty) execution of T_f, and the shaded boxes represent the re-execution of T_f. The short vertical lines above the schedule indicate the period boundaries of the thread set.

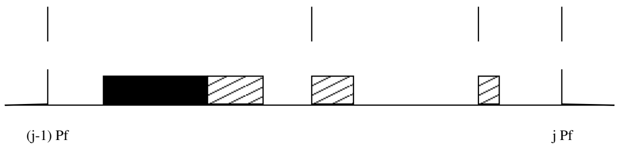

Figure 2. Re-execution of a faulty thread in FT-RT-Mach

4.5 State Information

As described in section 2.1.1, the thread procedure in FT-RT-Mach is invoked and terminated for each thread instance. At each invocation, the SP is reset and the execution stack is cleared. So the FT-RT-Mach thread model design is not suitable for state information maintenance. The only way to maintain some state information on FT-RT-Mach is to define the state information as global variables that are shared by all the threads and are not cleared by any thread invocation.

In the TFT scheme, the procedure body of a thread instance is defined as two blocks, the primary block and the recovery block. According to the current execution state, the appropriate block is chosen to run. If state information maintenance is needed, user level check-pointing can be implemented by the application to save the state information in the primary block, and restore it in the recovery block.

5 TFT Scheme Implemented on DEOS

We apply the same general FTRMS mechanism as described in the previous section. We also assume the same fault model.

5.1 Admission Control

Since the threads on DEOS have harmonic periods, the RMS utilization threshold is up to 100%. Applying Equation (1), a set of n harmonic threads is guaranteed to meet all deadlines under the fault model assumptions if the following condition is satisfied:

$$\sum_{i=1}^{n} U_i \leq 1 - U_{max} \quad (2)$$

If the system records the summation of the current thread utilization and U_{max}, then the time complexity of admitting a new thread is $O(1)$.

The admission control condition in Equation (2) is conservative, because it does not make full use of the fault model assumption (A2). For example, assume two threads are submitted to the system with the above admission control condition. $U_1 = 50\%$, $U_2 = 25\%$ and $P_2 = 2P_1$. Obviously, this set of threads will be rejected according to (2). However, in the following figures showing all of the three possible cases in which a transient fault may occur, we observe that these two threads cannot miss their deadlines as long as there is only one transient fault in the largest period. In the figures, a black box denotes the primary execution of a thread, and a shaded box denotes the re-execution of a thread.

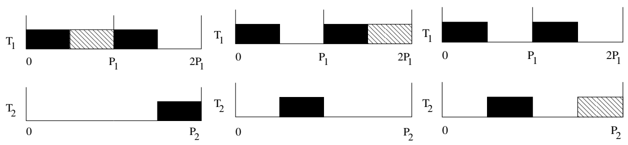

Figure 3. Left: T1 re-executes in its first period; Center: T1 re-executes in its second period; Right: T2 re-executes in its period

To obtain a higher system utilization, a new sufficient and necessary fault-tolerant admission control condition which is similar to the exact characterization analysis [7] may be used. Recall that in DEOS, the set of possible periods supported by the application is fixed by the registry before the application is submitted to the system. Assume there are ρ harmonic periods defined in the system and they are sorted in increasing order. We define the following notation:

- P^k: the k^{th} period, $(1 \leq k \leq \rho)$.
- $\Gamma_{l,k}$: the ratio of P^l over P^k, $\Gamma_{l,k} = P^l/P^k$, $(1 \leq l, k \leq \rho)$. Due to the harmonic feature, $\Gamma_{l,k}$ is always an integer if $l > k$.
- $\mathcal{A}^k$: the set of threads whose periods are equal to P^k, $\mathcal{A}^k = \{j|P_j = P^k\}$.
- $sumC^k$: the summation of the execution time of all the threads in the set $\mathcal{A}^k$, $sumC^k = \sum_{i\in\mathcal{A}^k} C_i$.
- $maxC^k$: the maximum of the execution time of all the threads in the set $\mathcal{A}^k$, $maxC^k = MAX_{i\in\mathcal{A}^k}\{C_i\}$, where MAX denotes the operation of calculating the maximum of a set of numbers.

The formula, $sumC^1 + maxC^1$, computes the worst case execution demand that must be finished within P^1, including both the primary execution demand of all the threads in $\mathcal{A}^1$ and the re-execution of an arbitrary thread in $\mathcal{A}^1$. The following formula estimates the worst case execution demand that must be finished within P^2.

$$\Gamma_{2,1}sumC^1 + sumC^2 + MAX(maxC^1, maxC^2) \quad (3)$$

The first two terms of Equation (3) calculate the primary execution demand of all the threads in $\mathcal{A}^1$ and $\mathcal{A}^2$ within a period of P^2. The third term estimates the worst case re-execution time of a thread that must be finished within P^2. Following the same analysis, the following formula computes the execution demand within a period P^k $(1 \leq k \leq \rho)$.

$$\sum_{j=1}^{k}\Gamma_{k,j}sumC^j) + MAX(maxC^1, .., maxC^k) \quad (4)$$

In order to enforce the time constraints of the threads, the execution demand within a certain period cannot be larger than the period. So the threads in the system are guaranteed to meet their deadlines under the fault model if and only if the following condition is satisfied:

$$\forall k \in [1, \rho],$$

$$\sum_{j=1}^{k}\Gamma_{k,j}sumC^j + MAX(maxC^1, .., maxC^k) \leq P^k \quad (5)$$

The correctness of this condition can be proven in a manner analogous to the exact characterization in [7]. If we apply this admission control scheme to the example shown in the previous subsection, the set of two threads will be accepted. This scheme utilizes the CPU better than the previous scheme, but it is more complicated to implement than the first one, especially when threads can be dynamically created or deleted. If the system keeps track of the values of $sumC^k$'s and $maxC^k$'s, the time complexity for admitting a new thread is $O(\rho)$, because the left side of an inequality in (5) can be computed from the previous inequality in time $O(1)$.

5.2 Fault Detection

The fault detection scheme is similar to the scheme on FT-RT-Mach. We detect a fault at the end of each thread periodic execution by checking the validity of the computation results via an acceptance test function. Since the acceptance test is closely related to the application, the application needs to define the acceptance test function for a new thread before creating it. Upon termination of the thread, the function is called to detect faults.

5.3 Fault Recovery

When a fault is detected by the acceptance test function, we need to recover from the fault by re-executing the thread before the deadline. Because of the harmonic feature of the thread set, we can prove that all the threads are guaranteed to meet their deadlines if the priority of the re-execution is the same as its primary execution. The proof of correctness is as follows.

Theorem Given a set of n harmonic threads $\{T_1, .., T_n\}$ where $P_1 \leq P_2 ... \leq P_n$, if the admission control condition (2) is satisfied, and the faulty thread is re-executed at its own priority, then all threads are guaranteed to meet their deadlines under the fault model assumptions.

Proof: We use contradiction to prove the theorem. Let T_f denote the faulty thread that needs to be re-executed. Assume the re-execution causes a thread, say T_m, to miss its deadline. Since the re-execution is carried on at the original priority of T_f, all the threads which have higher priority than T_f are not influenced by the re-execution. Thus, it is true that $P_f \leq P_m$.

The execution demand that must be finished within the period P_m is equal to $\sum_{i=1}^{m} \frac{P_m}{P_i} C_i + C_f$, including both the primary executions of the threads and the re-execution of T_f. A missed deadline indicates that this execution demand exceeds P_m, that is, the following must be true.

$$\sum_{i=1}^{m} \frac{P_m}{P_i} C_i + C_f > P_m \tag{6}$$

(6) is equivalent to the following condition.

$$\sum_{i=1}^{m} U_i + \frac{C_f}{P_m} > 1 \tag{7}$$

Obviously, (7) contradicts the admission control condition (2). □

The correctness of re-executing at the same priority under admission control condition (5) can be proven similarly. On DEOS, the recovery scheme is much simpler than the one implemented on FT-RT-Mach, which needs to reclaim backup slack for re-execution. As a result, the re-execution overhead of DEOS is smaller than that of FT-RT-Mach, because there is no extra computation effort in addition to the original RMS scheduling scheme. Thus, the harmonic feature of DEOS not only simplifies the implementation of the kernel, but it improves the fault recovery scheme, as well.

5.4 State Information

Since most DEOS applications need some state information from previous executions, state information maintenance is a very important issue in DEOS. Because the invocation of a DEOS thread instance does not clear the execution stack, the previous values of the state variables can be accessed directly from the thread stack when there is no fault. So the design of the DEOS thread model is more suitable than FT-RT-Mach for applications which need to maintain state information.

When a transient fault occurs and the thread needs to re-execute, some state information may have been changed during the primary (faulty) execution. In order to obtain correct results for the re-execution, all the essential state information needs to be restored to the same initial value for the primary execution. To achieve this, we maintain two copies of state information of the thread, a primary and a backup copy. A thread can modify the primary copy of the state information during its execution, while the backup copy is a clean copy that can be updated only by kernel at the end of its execution in each period if no fault is detected. The backup copy of the state information is updated using the correct results of the primary copy. If a fault is detected, then before the beginning of the re-execution, the primary copy of the state information is restored from the backup copy, which was updated at the end of the previous period and carries the correct initial values for the re-execution. Instead of user level check-pointing as in FT-RT-Mach, the state information is maintained by the kernel.

We will next discuss several issues about the design and implementation of state information maintenance.

- **What can be considered as state information?** Since state information is represented by some variables used and changed in the periodic computation, the first option is to consider all the variables stored in the thread execution stack to be state information. The advantage of this scheme is simplicity, and the concept of state information is completely transparent to the application designer. However, the disadvantage of this scheme is the large time and space overhead. Since state information may only be a small part of the execution stack, maintaining a copy of the whole stack is time consuming and memory intensive. We chose another option which allows the application designer to analyze the application and declare selected variables as state information. Though this partial-check-point scheme requires extra work by the application designer, it is widely applicable for industrial applications, due to its small time overhead and space overhead.

- **How to allocate and store the backup copy?** After the application declares a set of variables as the state information, the kernel allocates a continuous block of memory space from the thread virtual address space for the storage of the backup copy of the state information. This backup memory space is divided into two parts: a state information table and the values of all the state variables. The state information table maintains an array of records each of which represents one state variable. The record maintains the addresses of the primary and backup copies of the state variable and the variable size. Given the state information table, we can copy the value of a variable from one location to the other for a number of bytes equal to the variable size.

There is one issue on restoring state information of re-executing threads that we have not yet dealt with. In DEOS, it is possible for threads running at different rates to share their state using shared memory. If the state of a faulty thread is used by another thread before the former can recover, then the latter's state may also get corrupted. To avoid this, we need to ensure that the state of a thread does not get used until it is determined to be fault free. This can be done either by scheduling the consumer of state information after the producer has completed, or by locking the memory of the producer until the state is determined to be fault free. These techniques have not yet been implemented in DEOS, and are left as future work. Currently, we assume that threads running at different rates do not share state information.

6 An Application Example in DEOS

We used a simulated airplane flight control application to test the implemented DEOS fault tolerance scheme. This is a realistic simulation of a commercial business jet that is used to determine various parameters such as velocity, acceleration, pitch angle, pitch angular rate and change in altitude of the aircraft given an input from the pilot stick, and the current values of altitude and air speed. This example models the pitch axis of a real aircraft that is valid at an altitude of 10,000 feet and air speed of 230 knots.

6.1 Threads

There are three threads in the sample program running at the same rate with a period of 100 milliseconds.

- The first thread, called *pilot*, simulates the behavior of the pilot. At periodic intervals, this thread toggles the *stick* from 0 to 1 and vice versa. When the stick is 0, the aircraft angle of attack (angle between nose and velocity vector) does not change, but the airplane loses altitude gradually due to the value of thrust being used in this model. When the stick is 1, the pilot is commanding the nose of the aircraft to pitch up, resulting in an increase in altitude. The output of this thread is the stick value.
- The second thread, called *control* is the pitch autopilot control law. The pilot stick command is combined with velocity and altitude sensor data to provide an elevator position command that gives a desirable aircraft response while maintaining passenger comfort.
- The third and final thread, called *airplane* models the entire aircraft. This thread takes the elevator command (in degrees) as input and simulates the response from the aircraft by modifying the values of velocity, pitch angular rate, pitch angle, and altitude.

6.2 Detecting and Tolerating Faults

Faults are detected using acceptance tests. The acceptance test used in this system is that the value of altitude should not change by more than 10 feet within one thread period. If it does, then the state of the thread has been corrupted, and recovery is needed.

Recovery is achieved by restoring the state of the thread saved at the end of the previous period, and re-executing the thread using the saved inputs. Once the re-execution ends, the acceptance test is used again. If the fault has been corrected, then the application continues operation as normal.

If the retry also generates an incorrect output, it may be due to a more serious fault, and then the redundant application running on a different computing module is allowed to take control.

6.3 Fault injection

We first ran the navigation program without faults and observed its output as shown in Figure 4. Next, we injected faults into the control law which caused the value of altitude to change abruptly as shown in the left part of Figure 5. Without using the TFT scheme, the application state remained incorrect. However, on using the TFT scheme, the fault was detected and corrected as shown in the right part of Figure 5. The spikes in the figure are due to the injected faults showing both the faulty and corrected outputs from the DEOS machine. In a real system, only the corrected output will be sent out, and the spikes will be eliminated. Effectively, the corrected output will look similar to the fault-free output.

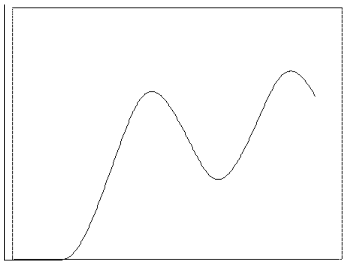

Figure 4. Aircraft's altitude v.s. time with no faults.

7 Conclusion

In this paper we presented the implementation of a TFT scheme on FT-RT-Mach and DEOS. Although both environments support hard periodic real-time tasks, they impose different constraints on the TFT scheme:

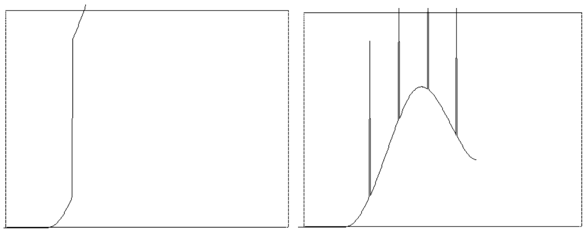

Figure 5. Aircraft's altitude v.s. time with faults. Left part: Faults injected and not corrected; Right part: Faults injected and corrected

- FT-RT-Mach is a general-purpose operating system developed for research purposes. In order to explore and implement all kinds of research results related with real-time systems, flexibility and modifiability are very important design policies for FT-RT-Mach. This is why FT-RT-Mach supports a large variety of thread types and scheduling policies. Because of the generic thread model, the fault recovery scheme implemented on FT-RT-Mach is not straightforward. The check-pointing of state information and the recovery block are implemented at user level. We are currently working on kernel modifications to include this.
- DEOS is a certified operating system for critical avionics applications. Correctness, robustness and integrity of the system implementation are the most important issues. The simplicity of the system results in a lower possibility of errors and bugs. With this goal in mind, the system only supports the periodic harmonic thread model and the RMS scheduling algorithm, which greatly simplify the kernel. By observing these limitations, we can implement a TFT scheme that not only provides time for re-executing a thread that encounters a transient fault, but also provides a check-pointing capability to restore essential state information.

  Practical application requirements are another important concern in the design of DEOS. The TFT technique implemented on DEOS has an optimal admission control condition and a simple fault recovery scheme. State information is maintained automatically by the kernel.

References

[1] Pam Binns. Incremental Rate Monotonic Scheduling for Improved Control System Performance. *Proceedings of IEEE Real-time Technology and Applications Symposium*, 6 1997.

[2] X. Castillo, S.R. McConnel, and D.P. Siewiorek. Derivation and Caliberation of a Transient Error Reliability Model. *IEEE Trans. on Computers*, C-31(7):658–671, 6 1982.

[3] A. Egan, D. Kutz, D. Mikulin, R. Melhem, and D. Mosse. Fault-Tolerant RT-Mach(FT-RT-Mach) and its Application to Real-Time Train Control. *Software Practice and Experience*, 1997.

[4] F. George. Introducing Primus Epic. *Business and Commercial Aviation*, pages 116–120, 11 1996.

[5] S. Ghosh, R. Melhem, D. Mosse, and J. Sen Sarma. Fault Tolerant, Rate Monotonic Scheduling. *Journal of Real-Time Systems*, 15(2), 9 1998.

[6] R.K. Iyer, D.J. Rossetti, and M.C. Hsueh. Measurement and Modeling of Computer Reliability as Affected by System Activity. *ACM Trans. on Computer Systems*, 4(3):214–237, 8 1986.

[7] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization And Average Case Behavior. *Proceedings of IEEE Real-time Systems Symposium*, pages 166–171, 12 1989.

[8] C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):47–61, 1973.

[9] T. Nakajima, T. Kitayama, and H. Tokuda. Experiments with Real-Time Servers in Real-Time Mach. *Proceedings of the USENIX Mach III Symposium*, pages 1–19, 4 1993.

[10] D. P. Siewiorek, V. Kini, H. Mashburn, S. McConnel, and M. Tsao. A case Study of C.mmp, Cm* and C.vmp: Part 1-Experiences with Fault Tolerance in Multiprocessor Systems. *Proceedings of the IEEE*, 66(10):1178–1199, 10 1978.

[11] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. *Proceedings of USENIX Mach Workshop*, pages 73–82, 10 1990.