SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper., **29**(4), 379–395 (1999)

Fault-Tolerant RT-Mach (FT-RT-Mach) and an Application to Real-Time Train Control

A. EGAN, D. KUTZ, D. MIKULIN, R. MELHEM AND D. MOSSÉ*
Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, USA
(email: {melhem, mosse}@cs.pitt.edu)

SUMMARY

Even though real-time systems have the stringent constraint of completing tasks before their deadlines, many existing real-time operating systems do not implement fault tolerance capabilities. In this paper we summarize fault tolerant real-time scheduling policy for dynamic tasks with ready times and deadlines. Our focus in this paper is the implementation, which includes fault-tolerant scheduling, re-scheduling, and recovery mechanisms in the FT-RT-Mach operating system, a fault-tolerant version of RT-Mach. A real-time train control application is then implemented using the FT-RT-Mach operating system. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: fault tolerance; operating system; real-time system; process control

1. INTRODUCTION

In train control, correct execution of tasks is determined not only by logical correctness, but also by satisfying certain temporal constraints. For example, when a train is moving, real-time collision path algorithms as well as velocity and break control must be carried out within very stringent timing constraints. Tasks with temporal constraints are *hard real-time* when failure to produce the desired results on time may lead to catastrophic consequences (such as loss of life or high monetary loss in the case of a train accident).

Many theoretical and practical solutions have been proposed for RT issues, both for embedded and non-embedded systems. One of them is the *RT-Mach* operating system, which focuses on solving the problem of executing real-time periodic threads in uniprocessor systems [1]. Several early real-time scheduling policies were implemented such as Rate-Monotonic Scheduling (RMS), and Earliest Deadline First (EDF) [2], while later implementations added support for continuous media (reserves [3], netphone [4], etc.). Such technology for dealing with real-time constraints has also been used in other commercial operating systems (e.g. QNX, VxWorks, and Mach-RT). All of the above are representatives of *preemptive* systems.

For non-preemptive operating systems, Spring [5,6] and Maruti [7] use explicit timing constraints to schedule real-time tasks, which are then guaranteed to execute within their

*Correspondence to: D. Mossé, Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, USA.
Contract/grant sponsor: DARPA; Contract/grant number: DABT52-96-C-0044.

CCC 0038–0644/99/040379–17$17.50

Received 6 May 1998
Revised 17 August 1998, 24 November 1998
Accepted 24 November 1998

deadlines. These hard deadlines are met by preallocating required resources on a timeline to provide a deterministic environment. The advantage of non-preemptive systems is that synchronization is simpler to achieve, and there is less overhead during execution of threads.

In either preemptive or non-preemptive systems, since missing hard deadlines may lead to catastrophic failures, issues of fault tolerance must be addressed in conjunction with real-time. Fault tolerance can be achieved either by hardware redundancy or by reserving time for recovery of tasks. The former technique usually executes modules concurrently, and results are voted upon. Although this scheme can mask faults, its cost may be prohibitive. The latter technique applies an alternative scheme which re-executes[†] tasks in case of faults. However, to enforce the real-time constraints of tasks, enough time should be reserved for re-execution of any failed task before its deadline.

Few experimental real-time operating systems have explored support of fault tolerance under real-time conditions. The Alpha approach to fault tolerance [8,9] is to provide replication, migration, and atomic transactions at the kernel level, and to allow implementations of policies on a higher level. The emphasis of Alpha is on providing a dependable kernel that controls priority inversion problems, avoids the occurrence of deadlocks, and tackles the problems of transient overloads. In essence, it is a soft real-time system, in that deadlines are *not guaranteed*.

HARTOS [10,11] is an operating system designed for fault-tolerant communication through the use of the inherently redundant interconnection structure of the hexagonal mesh. MARS [12,13] provides hardware-level fault tolerance capabilities; however, such capabilities are not user-transparent, nor can they be applied in dynamic systems, and MARS is highly dependent on the hardware platform of the embedded system it is built for.

The Maruti system [7] uses replicas to tolerate faults, combining replicas with a resource allocation scheme to guarantee the real-time behavior. An object-based model [14] enhances objects to enable the specification of real-time and fault-tolerant characteristics, which include timing and resource requirements, a monitoring hierarchy for information flow, and a set of error handlers.

Precursor works that involve fault tolerance based on the Mach operating system include References [15–21]. ARTS [20,21] is a RT-Mach system that provides support for exception-handling as well as timing errors. The work in Russinovich [15] has an implementation of fault tolerance facilities that can be used at the user level, and caters for general purpose user applications. In contrast, the work in Ramos-Thuel and Strosnider [16,17] is of theoretical nature, evaluated through simulations; it is a table-driven approach to computing the slack of the tasks in the system. After such slack is computed, it may be used for recovery from faults in the system; it is possible that lower-priority tasks miss their deadlines in the case the recovery of the faulty task takes higher priority. In our implementation, FT-RT-Mach, we adopt a non-preemptive strategy for the hard real-time tasks, with the ability to create preemptive tasks at lower priorities. This is motivated by the train control application and to simplify the resource management (e.g. no need for semaphores, and therefore no priority inversion concerns). Finally, our work on preemptive fault-tolerant real-time schedule, FT-RMS, developed bounds for rate monotonic scheduling with provisions for fault recovery. Simulations have shown that the amount of schedulability lost is not significant [19], and our implementation has shown that the operating system overhead is minimal [18].

The thrust of this paper is twofold: first, the description of our fault-tolerant scheduler and its implementation, based on the timeline paradigm. Our new FT-RT-Mach allows users to

[†]Other forms of recovery, such as recovery blocks, are also possible. We mention re-execution for simplicity of presentation.

specify non-preemptive threads with or without tolerance to transient faults. Secondly, we show how our thread model is used for the implementation of a train control system on FT-RT-Mach. In that application, transient faults are tolerated by FT-RT-Mach while permanent faults are tolerated at the user level by allocating tasks on primary and secondary processors, each of which runs FT-RT-Mach.

2. THE FAULT-TOLERANT ALGORITHM

2.1. Model

We consider a system in which tasks[‡] are dynamically submitted, that is, tasks may arrive in the system while other tasks are executing. The characteristics of a real-time task, T_i, include the task's ready time or earliest start time, r_i, its the worst case execution time, c_i, and its deadline, d_i. All inputs occur at the beginning of the task and all outputs are generated in the end of the task, so that the whole task can be re-executed in case of a fault. Inputs are buffered before execution to enforce equal inputs to recovering threads (in case receiving messages have consuming semantics).

Tasks are scheduled *non-preemptively* on a timeline [22], where the start time s_i and the end time $e_i = s_i + c_i$ for each task are recorded and associated with the task. This implies that a total ordering is imposed on the tasks that are ready to execute. Thus, the timeline can be seen as a queue sorted by start times.

At this stage in the implementation, the operating system tolerates *only transient faults*, which are short-lived malfunctions in a hardware component. A prevalent example of transient faults is timing faults, such as a task exceeding its worst case execution time or a task missing its deadline due to transient overloads in the system. Our focus is on transient faults since they are significantly more frequent than permanent faults [23–25]. For example, in Siewiorek *et al.* [25], measurements showed that transient faults are 30 times more frequent than permanent faults, while in Iyer *et al.* [26], 83 per cent of all faults were determined to be transient or intermittent. At the user level, hardware redundancy can be used to tolerate permanent faults.

We assume a detection mechanism that detects errors at the end of a task. Our model assumes that a task executes correctly if it finishes before the specified deadline and produces correct results according to its specification. Otherwise we say that a fault occurred. Recovery from faults is accomplished by re-executing the task in which the error was detected.

Transient faults are assumed to be at least ΔF time units apart. It means that two faults cannot occur within a ΔF time interval and at most one task can experience a fault in this interval. We also assume that $\forall i$, $\Delta F > 2c_i$, which means that transient faults will affect only a single task and that a fault will not affect a task during its re-execution. This last assumption needs to be appropriately modified if a recovery technique other than re-execution is used.

In this paper, we are concerned only with faults that occur in the user code, or while the operating system is executing system calls on behalf of some user application. This is because the operating system executes operating system functions a relatively small percentage of the CPU time, and such faults can be considered permanent faults, since they usually can corrupt the internal structures of the system.

[‡]We call executable units either tasks or threads in this paper.

2.2. Algorithm

Let Q_T be a queue of n tasks to be scheduled for execution at the current time t_0. If Q_T contains tasks $T_1, \ldots, T_n$ the algorithm below can be used to check if the queue is schedulable without violating the deadlines of the n tasks, even in the presence of faults. The way the Q_T is created is not important; it can be any non-preemptive strategy, for example a slack-based policy [27], a bidding policy [28], or Earliest Due Date (EDD) [29].

When a set of tasks arrives, an admission procedure determines which tasks are accepted and guaranteed to execute within their deadlines. The admission procedure consists of two parts: (a) construction of the schedule; and (b) verification that the schedule satisfies real-time and fault tolerance constraints. As mentioned before, schedule construction is done by adding the new task to the existing queue or by completely reordering the tasks in the queue. Schedule verification extends the algorithm in Ghosh *et al.* [27] to encompass ready times, and is discussed below. A task T_i cannot start before its ready time r_i, and if $r_i > t_0$, gaps may be introduced in the timeline.

Upon arrival of a new task T_i in the system, the scheduler is invoked. Assume that the scheduler inserts the new task in the queue at position *current*, such that $e_i \leq d_i - c_i$, which allows T_i to re-execute in case of a fault. However, guaranteeing that $e_i \leq d_i - c_i$ is not enough to ensure that T_i will complete before its deadline. This is because some other task(s) that is (are) scheduled before T_i may re-execute, and may cause T_i to miss its deadline.

We assume that the scheduler does not change the order of the tasks in the queue, and that an admission control algorithm is invoked for every task added to the queue. Then at the time of adding a new task, tasks T_1 though $T_{current-1}$ need not be considered because those tasks were already guaranteed to meet their deadlines, and insertion of the new task does not affect them (the start and end times of these tasks are not changed). In order to tolerate *a single fault* throughout the lifetime of the system, the verification procedure needs only to account for a single re-execution, which would be accomplished by checking that $\forall i \geq current$, $le_i \leq d_i$, where le_i is the latest possible end time of T_i, and is defined as $le_i = e_i + \max_{j \leq i}\{c_j\}$. The last factor accounts for the re-execution of the longest task in the task set before the incoming task.

However, more than a single fault may occur. In particular, if a fault may occur every ΔF time units, the time reserved for re-executions (slack) must accommodate this fault model. For that, we divide the queue into segments, each of which is related to the fault interval ΔF. A *segment* S^k corresponds to a set of tasks in the schedule that can be accommodated within ΔF, with re-execution. The tasks in a segment are consecutive tasks from the total ordering imposed by the timeline. Due to the hard real-time constraints, and since tasks are non-preemptive, the algorithm defines segments conservatively.

Specifically, a segment is formed by adding tasks to fill the time interval ΔF, taking into account the possibility of re-execution of any one task in the segment. That is, if β^j is the number of tasks in all segments S^1 to S^j, a segment S^k is the maximal set of consecutive tasks defined as follows:

$$S^k = \left\{ T_i \,|\, i > \beta^{k-1} \wedge i \leq \beta^k \wedge \sum_{i=\beta^{k-1}+1}^{\beta^k} c_i + \max_{i \in S^k}\{c_i\} \leq \Delta F \right\}$$

The above definition of a segment takes care of adding slack, but assumes that all tasks are ready at current time t_0, and thus the schedule does not contain gaps. The potential slack is introduced due to the re-execution of faulty tasks, and thus the tasks start times may


```
1  for pos = current to n do {
2       if ((le_{pos-1} + c_pos > end_segment) AND
3           (max(r_pos, e_{pos-1} + 2 * c_pos) > end_segment)
4       {                                        /* start new segment */
5           ls_pos = max(r_pos, le_{pos-1});
6           e_pos = ls_pos + c_pos;
7           le_pos = ls_pos + 2 * c_pos;
8           end_segment = ls_pos + ΔF;
9       }
10      else {                                   /* add task to existing segment *
11          ls_pos = max(r_pos, e_{pos-1})
12          e_pos = ls_pos + c_pos
13          le_pos = max(le_{pos-1} + c_pos, ls_pos + 2 * c_pos)
14      }
15 if (le_pos > d_pos)                           /* violates deadline */
16      return REJECT
17 } /* end for */
18 return ACCEPT
```

Figure 1. Algorithm `Verification`

potentially change. To verify whether tasks will complete within their deadlines, we need first to define ls_i, the latest potential start time of the task in case faults occur in the system before or during the execution of the task. The computation of ls_i is straightforward if the original schedule does not contain gaps (i.e. if the tasks are ready as soon as they arrive). However, when the schedule contains gaps, ls_i is defined as follows. For the first task in a segment, $ls_i = \max(le_{i-1}, r_i)$, and for each subsequent task, $ls_{i+1} = \max(ls_i + c_i, r_{i+1})$. If $le_i = \max(s_i + 2c_i, le_{i-1} + c_i)$, then a segment is redefined as follows (note that le_i and ls_i take care of both original gaps in the schedule and added slack):

$$S^k = \{T_i | i > \beta^{k-1} \wedge i \leq \beta^k \wedge le_i - ls_{\beta^{k-1}+1} \leq \Delta F\}$$

In the algorithm `Verification` (see Figure 1), we let n be the number of tasks already in the queue, and *current* be the position that the new task is inserted in the queue. Also, *end_segment* denotes the end of the segment, which is the start time of the first task of a segment plus the fault interval (i.e. $end\_segment = ls_{\beta^{k-1}+1} + \Delta F$).

Note that algorithm `Verification` does not change the schedule, it only constructs a worst case scenario in which each task T_i will start at ls_i rather than s_i. Thereafter, the algorithm checks if tasks will miss their deadlines in this worst case scenario. The condition on lines 2 and 3 checks whether it is time to start a new segment. This happens when the latest end time of the current task goes beyond the current segment boundary. If this is the case, a new segment is initialized (lines 5–8), by placing the new task to start after the *latest end* of the last task in the previous segment. Otherwise the task is placed after the end time of the previous tasks (lines 11–13).

Clearly, in case of faults, this start time may change due to re-execution of one of the tasks before the current task. On lines 15 and 16 we check if the latest end of the current task is greater than its deadline. If it is, we cannot admit the new task since, due to the re-execution of some task in the segment, the current task may not be able to meet its deadline. If, for

each task considered (line 1), the latest end is less than or equal to its deadline, the algorithm accepts the new task (line 18).

2.3. Summary of algorithm evaluation

In this section we summarize (from Ghosh *et al.* [30]) the performance of the algorithm above. Our simulation studies have considered different workloads and different scheduling policies that were adapted to be used with the fault-tolerant scheduling policy. We noted that the percentage of rejected tasks for the various loads has little variation for varying values of Δ_F. This is because tasks are rejected due to their timing constraints and not due to the separation between faults.

We have shown that, as expected, the fault tolerance capability decreases the number of tasks that miss their deadlines (called *lost tasks*) at the cost of increasing the number of *rejected tasks*. Note that tasks are only lost if there are more faults than the model assumes.

We have also studied the behavior of the system according to the ratio of the costs of losing a task after accepting it and of rejecting a task (not accepting it when submitted). We note that only when this ratio is very high, it pays off to use redundancy in order to tolerate faults; in particular, our simulations have shown that when the ratio is less than approximately 100, providing for error recovery is not cost effective. Clearly, the value of this ratio is intrinsically linked to the value chosen for Δ_F.

The study in Ghosh *et al.* [30] also provides the system designer with a methodology to determine the maximum load he/she should allow in the system, before the cost of losing/rejecting tasks becomes prohibitive. Naturally, this value depends on the characteristics of the workload being offered to the system.

3. IMPLEMENTATION OF FT TL IN RT-MACH

In this section we discuss the implementation of the fault-tolerant (FT) timeline (TL) scheduling scheme in FT-RT-Mach. We present a macro view of the current system and the changes needed to accommodate the new paradigm. We also discuss how the scheduler takes into consideration the ability to tolerate faults, creating the FT TL paradigm.

3.1. Design issues

3.1.1. RT-Mach design

The scheduling and dispatching of tasks in RT-Mach is implemented as two separate modules, namely a *scheduling policy* and a *dispatcher*, which is based on dispatching the highest priority thread ready to run. The ready queue of RT-Mach is implemented as a multi-priority queue with 32 priority levels, with 0 being the highest priority. RT-Mach algorithms assume that tasks in each priority level can be executed continuously until the task ends or until a higher priority task preempts it. When a thread is *created*, the user area, stack, and data structures are initialized, and a *start timer* for that thread is created and armed. The thread is enabled and added to the ready queue. Subsequent invocations of this thread (in later periods) are done similarly: at every thread period, the start timer for the thread goes off and the new instance of the thread is placed in the run queue.

When an executing thread terminates, it is removed from the run queue. Tasks which are periodic are not destroyed, but simply put to sleep and the corresponding start timer is set to the beginning of the next period. To detect tasks that are still running at their deadlines, an *end timer* is created and armed for each task. If a task does not terminate by its deadline, the end timer goes off and invokes a deadline handler thread (whose code is supplied by the user). The user writing the deadline handler can kill or resume the thread in the next invocation.

3.1.2. Adding timelines

To implement a non-preemptive time driven scheduler, several modifications to the kernel data structures, procedures, and API were needed. We call our non-preemptive policy the *timeline* (TL) policy. To guarantee that a TL thread will not be preempted we execute TL threads at the highest priority level in the system. All other threads are demoted so that only TL threads can execute at priority 0. To implement the above, we create a separate queue in the kernel, which represents the current state of the timeline. Tasks are put in this queue sorted by their deadlines, and the verification algorithm is invoked (see section 2).

For all timeline threads, a timer is created and armed for the start time of the thread (minus the estimated time it takes to handle the timer; in the timeline approach, this is a constant amount). When this start timer goes off, the thread is placed in the run queue of priority 0. A difference between the FT-RT-Mach mechanism and the one in RT-Mach is that in the former the start time of the thread may change after the thread is admitted. It depends upon new incoming tasks and also on whether a fault occurred in the system or not (see below). In either case, the tasks may be shifted, and their timers modified. For this reason we chose to arm only timers of the thread at the head if the timeline. Clearly, this solution has the drawback of needing to disarm timers of the current head of the queue, in case a new thread is inserted earlier than the head of the timeline.

When an executing thread terminates, it is removed from the run queue and from the timeline queue. At this point, timers of the next thread are armed and when its start timer goes off, the thread is moved to the run queue. Using this approach we guarantee that the top priority run queue contains a maximum of one process, namely the process at the head of the timeline.

3.1.3. FTTL and re-execution

To provide fault tolerance through re-execution, we also modified the criteria for accepting a task. Our implementation of the admission criteria implements algorithm `Verification` from section 2, taking into consideration the slack needed for any one task to re-execute within the interval ΔF; effectively, the timeline becomes FTTL. Further, we have also implemented a mechanism for task re-execution. Upon termination, the task executes the `exit` procedure, which either kills the thread (disarms and destroys its timers, marks its memory as free, gives back space allocated, etc.) or re-executes it (resets the start and end timers, restore context, etc.).

We assume that a *fault flag* will be set in case a fault occurs; this fault flag can be set by a system error detection mechanism[§] or by a user performing, for example, acceptance tests. Re-execution will take place if the flag is set. In case of re-execution, the timers of the next task will not be armed until appropriate. For example, the start time of a thread T_i will be

[§]This error detection can be done in hardware or by the operating system

changed if the faulty task needs to re-execute within $[s_i, e_i]$; another example is the start time of a task after a gap: it does not need to be changed, unless the gap is shorter than the task re-executing.

3.1.4. Persistent threads

Beta testing of our first FT TL implementation uncovered an overhead that heavily impacted our implementation. In many embedded applications, the same task may be repeated at irregular intervals in response to some asynchronous event, such as an incoming train onto a train yard. This would require the creation of a new FTTL thread every time the event occurred. Most of the overhead was in the thread creation part, which could not be withstood by some low latency applications (e.g. our train control application). To avoid this overhead, we introduced *persistent* timeline threads.

A persistent thread is not scheduled at creation time. When it is *created*, its stack is allocated and other data structures are initialized. It is then suspended. A subsequent call is needed to *activate* the persistent thread, specifying $\langle r, c, d \rangle$, which inserts an instance of a persistent thread on the timeline. Clearly, a persistent thread may be FT or non-FT. The admission mechanism above works identically for threads that are persistent or not; the only difference is that the admission algorithm is run upon activation for persistent threads and upon creation for non-persistent threads.

When a persistent thread terminates, its resources are not deallocated, but the thread is suspended and its instruction pointer reset, so that another instance may be activated at a later time. More than one instance of a persistent thread may be activated and thus appear simultaneously in the timeline. Clearly, the overhead to activate a persistent thread is much smaller than to create a new thread (see section 3.3).

3.1.5. Non-fault-tolerant threads

The primary intention of the FT TL scheme is to provide fault tolerance. However, there are situations when it is preferable to avoid (or it is not necessary to have) scheduler-level fault tolerance. For example in a distributed system with a centralized scheduler, we may wish to schedule backup tasks on different processors to tolerate only permanent faults. The automatic transient fault handling of our FT TL would add unnecessary overhead. To accommodate this scenario, we added an attribute to the thread structure, namely a flag `thread_isFT`, to distinguish between FT and non-FT threads. For non-FT threads there is no provision for re-execution. This relaxes the restrictions on start and end times: the only requirement that needs to be checked for T_i is $ls_i + c_i \leq d_i$.

Our scheduler/admission algorithm allows for intermixing of FT and non-FT threads under one unified execution/scheduling model. The factor $\max\{c_{pos}\}$ from line 3 of Figure 1 is not needed when the task is non-FT. The fault interval ΔF is only relevant to FT threads, and therefore the segments and corresponding fault-tolerant admission control algorithms are only performed for the FT threads.

3.1.6. Non-real-time threads

Since RT-Mach is a microkernel, much of the system functionality resides at user-level threads (e.g. the Unix server). Non-real-time threads can be accommodated in a real-time system by rewriting the threads to comply with the real-time specifications (e.g. assigning ∞

to the deadlines and/or periods). To avoid this inconvenience to users, we implemented the system to revert to a non-real-time policy (e.g. standard timesharing) whenever there is no timeline thread scheduled. The timeline mechanism does not preclude the non-real-time user threads from executing.

In doing so, FT-RT-Mach allows any necessary periodic, I/O, and other types of threads to be processed as usual. Such threads have no guaranteed temporal behavior, since the TL threads always have highest priority when they are ready to run.

3.2. Kernel and RT library modifications

The fault tolerance mechanism and its implementation required modifications in both the microkernel (RT-Mach internals) and the real-time thread library. We start by describing the changes that were made in the real-time library code, the new functions added, and the changes in the data structure. Then we describe the kernel modifications.

3.2.1. Additions to the RT library

A function `fttl_thread_attribute_init()` for initialization of thread attributes was added to enhance the current API to accept TL threads. Real-time thread attributes structure is required by `syscall_rt_thread_create()` system call, which already exists in RT Mach and which is invoked to create new real-time threads. For that, two structures, `rt_thread_attr_t` and `thread_t`, used by `syscall_rt_thread_create()` had to be modified to include the timeline attributes. We added three new flags, as follows: `tl_flag` to indicate whether a thread is a timeline thread; `thread_isFT` to determine whether it is fault-tolerant or not; and `tl_isPersistent` to indicate whether it is persistent TL thread or not.

A new scheduling policy `sched_policy_ft_timeline` was added to `sched_policy.h` to allow FT TL scheduling to be enabled using the standard `set_scheduling_policy` system call. The default fault interval ΔF is defined in `ft_timeline.c` to be very large (in our implementation it is set to 50 seconds, which in practice means no fault tolerance capabilities). This value may (and should) be changed using the new `fttl_set_fault_interval` system call.

3.2.2. Kernel modifications

Modifications to the kernel consist of two major parts: *admission control* and *dispatching*. A new admission policy was added to the microkernel (called fault-tolerant timeline or `ft_timeline`). Under RT-Mach kernel source tree in *kernel/rt/sched_policy* files, `ft_timeline.c` and `ft_timeline.h` hold the admission control code which implements the algorithm described above.

To allow for the execution of the non-TL tasks, the basic dispatching code was taken from the files which implement the time sharing policy, which is default RT-Mach scheduling policy (in file `mk.c` in directory *kernel/rt/sched_policy*). During dispatching, preference is given to threads at priority 0, that is, TL threads. We added checks in the kernel to prevent users from specifying TL threads without changing the scheduling policy to TL. This is a necessary change in the kernel, since the different scheduling policies have conflicting models. We still allow other types of threads to be created and executed within the FTTL

paradigm (e.g. rate-monotonic periodic threads can still be specified, and will be run periodically and preemptively), but no temporal guarantees are given.

The dispatching also had to be modified. In addition to usual dispatching, TL tasks are non-preemptive, and FT TL threads may be re-executed due to transient faults. Recall that either a user-level or system-level error detection mechanism can set an error state in the thread structure (`fault flag`), and the dispatching mechanism consults it at thread termination. We have used both random and user-generated fault injection for testing, debugging, and evaluating the algorithm.

To enforce re-execution of faulty threads, we modified the function `rt_thread_exit()`, which is invoked when a thread terminates. Since FT TL is the only policy in FT-RT-Mach that deals with faults, only FT TL threads are restarted in case of transient faults. To restart a thread we use the `rt_thread_reset()` function to set up machine-specific context. This function resets the user stack with initial values stored in each thread's control block and sets the program counter to point to the first instruction of a thread. The overhead to reset a thread is very low. Specifically, there are a total of 12 single-word assignment statements in C in `rt_thread_reset()`.

In addition to `rt_thread_reset()`, we also needed to modify time specifications of both start and end timers and rearm both of them. The start timer gets a current value of the system clock, and the end timer is set to start timer plus execution time. The re-execution of tasks is guaranteed to avoid violations of tasks' timing constraints and the dispatcher currently simply restarts a faulty thread. This is because our admission control guarantees that there is enough time to re-execute at least one thread every ΔF time units. However, the system can be made to monitor for violations of the fault model, by decreasing the available slack at each task re-execution, and allowing other tasks to re-execute only if there is still some slack left. We have left this for future implementations.

The existing RT-Mach deadline handler library functions, `rt_thread_deadline_handler` and `rt_thread_rtdeadline_handler`, may also be used with FT TL threads. They allow the thread to be notified when a FT TL thread exceeds its worst case execution time. Since the FT TL scheduler automatically includes enough slack to re-execute the thread, the deadline handler function can request re-execution of the thread, as follows. The deadline handler sets the fault flag on behalf of the thread and `thread_terminate()` is invoked. This will signal the scheduler to re-execute the thread. If a non-recoverable error is encountered, a thread may be terminated by using `thread_terminate()` without setting the fault flag.

Due to the non-preemptive nature of the TL threads, our deadline handlers are set to go off at the scheduled end of the thread execution, e_i. However, if the fault model includes timing errors (e.g. a task executing longer than its c_i), our scheme is flexible enough to allow tasks to resume their executions and not only re-execute. The system can allow threads to 'overexecute' for as much as the reserved slack. This can be used in several situations, for example to implement the imprecise computation model [31], to allow for recovery from timing errors, and to allow for more efficient use of resources (by allowing users to specify the average execution time instead of the worst case execution time of a task, and allowing the thread to 'overexecute'). In such situations, the system integrator must ensure that the combined overtime requested by tasks cannot exceed the slack reserved for 're-execution'.

To ensure that the timeline schedule is not compromised (i.e. protection against malicious or malformed programs), the deadline handler is itself preempted if it exceeds the deadline of its parent thread. For the same reason, the deadline handler is not called if an FT TL thread exceeds its execution time during re-execution. Similarly, if an FTTL thread does not have a

deadline handler or if a thread is not FT, the thread will be terminated automatically by the system when the thread exceeds its execution time.

Finally, two new system calls were added to activate and deactivate persistent threads, namely `fttl_activate_persist_thread`, and `fttl_deactivate_persist_thread`. Since several instances of a persistent thread may be scheduled on the timeline, an instance number parameter was added to the call. A persistent thread can identify itself by using the added `fttl_instance_self` call.

3.3. Experimental evaluation of overhead

We ran some experiments to measure the overhead of the new kernel functions. The main functions we are interested in are those associated with the creation, activation, and maintenance[¶] of threads. These experiments were conducted on a Pentium-Pro 200MHz. We used the on-chip Pentium clock to obtain accurate measurements of the execution time of the threads and the overhead of the system functions.

Table I. System overhead per instance of persistent and non-persistent FTTL threads

		Total overhead (ms)		
	Creation (ms)	util = 15%	util = 30%	util = 45%
Persistent	2.48	0.138	0.145	0.139
Non-persistent	2.49	2.593	2.558	2.581

In the first experiment we compare the overhead of sequentially creating 150 threads using `fttl_thread_create` and the overhead of a single creation and 150 sequential activations using `fttl_persistent_thread_activate`. The per-instance overhead is shown for thread utilization of 15 per cent, 30 per cent and 45 per cent in Table I. Note that in this experiment we could not exceed a utilization of 50 per cent due to the assumption on the utilization of the fault-tolerant threads. In the first column of Table I, we show the average time for creating each thread. This time (approximately 2.5 ms) depends very slightly on the scheduling policy. As clear from the table, the overhead in the non-persistent threads is much higher than the persistent threads, since in the non-persistent case, a thread is created for each instance.

In a second experiment we compare the system overhead of the timeline implementation and the RT-Mach preemptive implementation; for this purpose, we run periodic preemptive threads (using RMS) and non-preemptive persistent threads (using FTTL). Periodicity of each FTTL thread was obtained by creating a persistent thread once and activating it at the end of the thread execution. Both experiments were run on the same kernel by selecting the scheduling policy appropriately.

For this experiment, 10 threads were created with varying periods (between 30 and 120 ms); the thread utilizations were changed to obtain different system loads between 0.15 and 0.6 (RMS can only guarantee correct executions up to approximately 0.7 load). The overhead of running this task set for 120 seconds is shown in Table II. From this table, it is clear that the FTTL threads have an overhead comparable to the RMS threads, for all system loads tested.

[¶]This includes exit and context switch times for each thread. For example, when a thread exits, it has to return all resources to the kernel.

Table II. System overhead per instance of persistent FTTL and periodic RMS threads

		Activation and maintenance (ms)			
	Creation (ms)	load = 15%	load = 30%	load = 45%	load = 60%
FTTL	2.48	0.111	0.113	0.113	0.113
RMS	2.38	0.099	0.106	0.113	0.115

Indeed, for higher system loads, the RMS has slightly higher overhead than FTTL, since RMS has more context switches than FTTL (due to preemptive scheduling).

4. A TRAIN CONTROL APPLICATION IN FT-RT-MACH

In this section we describe an application implemented on FT-RT-Mach for train control. As will become apparent, there are several real time and fault tolerance requirements in this application. We concentrate on a specific part of the system, which is restricted to a *train yard*. A yard is a collection of tracks that merge and fan-out a restricted physical location; a yard is used for temporary storage, maneuvering, loading/unloading, and parking of trains.

Further, when implementing this application on FT-RT-Mach, we only took advantage of the tolerance to transient faults. Permanent faults (processor crashes, mainly) were dealt with by spatial redundancy; more specifically, as in the train yard, we use a watchdog and a duplicated standby spare as a backup processor.

4.1. Environment

A train yard contains several miles of tracks. Spread along the tracks are *sensors* and *actuators*, which are collectively called *data points*. There are approximately 950 data points of 12 different types in the train yard we consider. Sensors include (a) wheel detectors distributed along the tracks, which detect whether there is a train car in a particular segment of the tracks; (b) switch detectors, that report the position of a track switch; and (c) semaphore detectors, which note the value of a signal, used for (dis)allowing flow in a particular track segment. Actuators, on the other hand, include retarders, to which commands can be given to accelerate, decelerate, or stop a train, and switch machines, to change the course (or track) of a train.

In the train yard studied, each activity takes place within a 50 ms cycle, and there are about 40 inputs per second per processor. This means that each input and each output has a deadline of about 1.25 ms. Note that these figures are for the implementation of the train yard for which we have data (with somewhat outdated equipment); a more modern facility would be able to handle a higher number of data points per processor.

With data from the entire yard, a controller program is able to make decisions concerning the position, velocity, routing, and movement of trains. These decisions are translated into specific commands and sent back out to the actuators to modify the state of the system.

Trains come into the yard and under control of our system through an *incoming delta*, which is a set of tracks that feeds into the yard from different locations. Trains are stopped in the tracks and a message is sent from another Information System (IS) to hand off the control of the incoming train. A handshake protocol is implemented to ensure the hand-off is

successful. After the train is successfully transferred to our system, it enters the *lead track*, which concentrates all traffic into the yard. Similarly, trains leaving the yard are moved onto the outgoing lead track, parked in an *outgoing delta track* and a message is sent to another IS, which takes control of the particular train.

4.2. Functional requirements

The controller, which is typically implemented by a process, must collect data from hundreds of (sometimes more than a thousand) data points. The data collected may need to be processed, carrying out, for example, data fusion used for merging streams of data (both for bandwidth savings and for functional necessity). Some data points must be read periodically (scanned), while some others generate data in an aperiodic fashion (interrupt-driven). We need to implement a system that provides collection and treatment in real-time, to preserve integrity of the system. As mentioned above, the cycle for transmitting the data to the controller and receiving actuator commands – such as retarder commands – is 50 milliseconds.

The requirements include processing within stringent timing constraints. If data is dropped before reaching the controller, an error has occurred, and an error handler must be invoked. Notifications must be sent to the handling thread; if treatment takes longer than expected, extra time is allowed, but this fact is recorded, since it should not occur frequently. If the extra time is not sufficient, the system may fail. In this case, there are two possible recovery actions, depending on the gravity of the fault: either the operator manually recovers from the fault, or all trains receive commands to stop, to avoid a catastrophe. This fail-safe approach is extremely expensive, since stopping trains causes and interruption of service and restarting of trains requires skilled personnel; thus, stopping/restarting is avoided as much as possible.

In terms of tolerance to hardware faults, our system must tolerate faults of data collection and faults of processing at the controller. These faults may be transient (i.e. no need for external intervention for repair) or permanent (need external intervention). We assume that transient faults last for very short periods of time, and affect at most one function; for example, a fault will affect the reading of a single data point, the processing of a set of data points, or the sending of commands to the actuators.

4.3. Implementation

In our design, shown in Figure 2, data points are partitioned into sets, and each set of data points is monitored/controlled by a processor, called the *IO-remote*. The IO-remote collects the data from the data points, processes such data (reformatting and pre-processing), carries out data fusion (collecting data from multiple data points and merges it into a single message), and relays the potentially fused data to a *controller*. Each IO-remote typically manages tens or hundreds of data points. The controller is a processor that possesses a global view of the yard, after collecting data from the IO-remotes.

We implemented a simulated environment for input from and output to the data points. The data used for input is a set of traces from an actual train yard, collected over the course of a full hour (data was obtained from a leader in train control system worldwide). All sensor data was stored in a file, and this file feeds the IO-remotes.

Since some of the data points need to be scanned, and some are interrupt-driven, we implemented different threads to handle these cases. Scanning was carried out with a periodic thread, while each interrupt-driven I/O operation invoked a new activation of a persistent thread (recall that the persistent threads are created *a priori* and activated as needed).

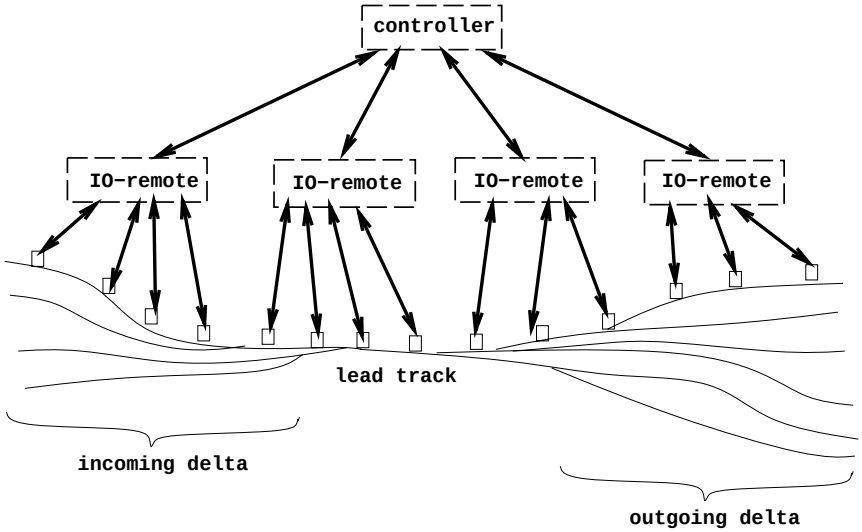

Figure 2. Design for a Train Control System using FT-RT-Mach; each node, represented by a dashed box, is detailed in Figure 3

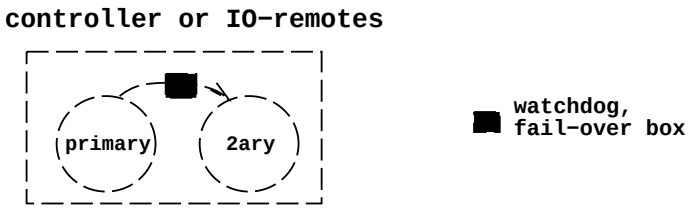

Figure 3. Architecture for each node in a Train Control System using FT-RT-Mach

Further, the controller was implemented to generate responses and stimuli to the actuators. The output generated was displayed to the consoles of the FT-RT-Mach machines running the controller as well as to the consoles of the processors running IO-remotes. Our future plans include developing a graphical interface for this controller to show the movement of the trains in the yard.

To provide tolerance to transient faults, the FT-RT-Mach scheme described in section 2 is used. However, the user application handled the tolerance to permanent faults, as follows. Each processor in the system is considered a dual: a *primary* and a *secondary* processor, with a hardware fail-over box connecting the two (see Figure 3). The threads running in the primary IO-remotes and controller have backup threads running in the secondary processor. Further, a watchdog thread was implemented to monitor the health of the dual-processor nodes and simulate the fail-over box. When a fault is detected (we use the fail-stop model for processor failures), the secondary processor is notified by the watchdog and its state is changed to *online*. When online, the processor actually sends out responses to the actuators; this is unlike in the *offline* mode, in which the processor receives the inputs but does not generate any output. This dual-processor architecture is applicable to both IO-remote and controller nodes.

Since each processor (primary or secondary) is running FT-RT-Mach, and our application takes advantage of its tolerance to transient faults, deadlines at the primary processor had to

be adjusted. In the worst case, the primary processor fails only at the end of the execution of a task, and this task needs to be executed before its deadline. Since detection and notification take some time (t_i), the deadlines at the primary are adjusted to

$$d_i' = d_i - 2c_i - t_i$$

With this adjustment, the secondary processor still has enough time to run the thread, suffer a transient fault, and re-execute the thread. Clearly, there are several policies to adjust these deadlines [32]; we implemented a simple one and will leave experimentation for future work. The ready times for the secondary processor must be similarly adjusted.

4.4. Lessons learned

Through the implementation of the train control system, we have learned that some of the features of FT-RT-Mach helped and some hindered the progress and effectiveness of the implementation.

4.4.1. Positive lessons

The non-preemptive nature of the FT TL threads makes it simple to work with critical sections and avoid the complexity and delays of using locks. Priority inversion, a well-known problem in real-time implementations [33], is prevented. Further, the threads with real-time capabilities helped in that we did not have to worry about explicitly and manually scheduling the threads, as is done in the previous frame-based implementation.

The guaranteed automatic re-execution of the threads due to transient faults, and the possibility of resuming or re-executing without violating deadlines when execution times were exceeded have proved very valuable. It simplified the design, and allowed for the implementation to progress at the higher level, without having to be concerned with the added complexity of programming own timers.

Finally, the fault tolerance admission control provided the implementors with guaranteed and deterministic behavior which can be exploited dynamically. Many current real-time operating systems (even the commercial ones) do not provide such facilities. Usually, admission control is carried out *a priori* and then threads are loaded onto the system.

4.4.2. Shortcomings and future work

Even though our system allows the user to treat missed deadlines with deadline handlers, in order to guarantee the determinism of the computations, we disallow such deadline handling a second time. It was clear to us when implementing functions in the data collection part, that a notification from deadline handlers *must* be issued. This is because some tasks need to clean up before they can ‘leave the scene of the crime’ (e.g. setting up flags and global variables for subsequent invocations of the same or different threads). Our current implementation simply terminates threads the second time a deadline is missed, a situation that should be remedied.

We also needed to implement tolerance to permanent faults, in the user code. This should be done through a system-provided facility and we are currently working on a middleware that provides this functionality. The users specify only the threads to be executed, and the system finds the appropriate node for the execution, according to load and probability of successful termination. We also plan to include the specification of the machines to run the threads, since

it may be the case that performing certain functions can only be done in specific machines. For example, each IO-remote can only be connected to a particular set of data points.

Finally, we have noted the need for non-preemptive periodic RT threads (our implementation uses persistent threads and activates the next instance at the end of each instance). From a design and programmer's perspective, using persistent threads is certainly not the best way of implementing periodic threads. A periodic scheduling for non-preemptive tasks (compatible with the FTTL approach) is underway.

5. CONCLUSION

In this paper we presented an implementation of a non-preemptive fault-tolerant real-time operating system, which can be used for dynamic and static task sets (but is geared towards aperiodic tasks). In the transient fault arena, we extended previous scheduling algorithms to include ready times and implemented it in the RT-Mach operating system, creating the FT-RT-Mach system. The scheme is based on providing sufficient slack for each task to re-execute in case of transient and intermittent faults. If faults are separated by at least ΔF, our policy guarantees that no accepted task will miss its deadline.

The implementation of FTTL on RT-Mach required a thorough understanding of the structure of the system. Many of the existing mechanisms were used after relatively few modifications. We also created *persistent threads* to decrease the overhead of thread creation.

Finally, we validated our approach with a control application, implementing a system that carries out train control in a train yard. The threads were created, activated and run under FT-RT-Mach, and the inputs to the system consisted of real trace data, collected in an actual train yard. We have seen that the facilities provided by FT-RT-Mach enhanced and facilitated the implementation, and we learned from this exercise that more functionality is needed. For example, the implementation of tolerance to permanent faults was done at the user-level, and an implementation of an automated facility is currently underway, through a middleware layer.

Our future work will focus on adding enhanced detection capabilities, considering precedence constraints and extending the scheme to a distributed implementation. For this last issue, we want to compare the performance of transparent error recovery at the operating system level and non-transparent treatment at the user level implementing a middleware for dealing with permanent faults.

REFERENCES

1. T. Nakajima and H. Tokuda, 'Implementation of scheduling policies in Real-Time Mach', *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, September 1992, pp. 165–169.
2. C. L. Liu and J. W. Layland, 'Scheduling algorithms for multiprogramming in hard real-time environment', *Journal of the ACM*, 46–61 (January 1973).
3. C. W. Mercer, S. Savage and H. Tokuda, 'Processor capacity reserves for multimedia operating systems', *Technical Report CMU-CS-93-157*, School of Computer Science, Carnegie Mellon University, 1993.
4. C. Lee, K. Yoshida, C. Mercer and R. Rajkumar, 'Predictable communication protocol processing in Real-Time Mach', *Proceedings of IEEE Real-time Technology and Applications Symposium*, June 1996.
5. J. Stankovic and K. Ramamritham, 'The Spring kernel', in A. Agrawala, K. Gordon and P. Hwang (eds.), *Mission Critical Computer Systems*, IOS Press, 1992, pp. 86–117.
6. K. Ramamritham and J. Stankovic, 'Scheduling strategies adopted in Spring: An overview', in A. van Tilborg and G. Koob, (eds), *Foundations of Real-Time Computing: Scheduling and Resource Management*, Kluwer Academic, 1991.

7. M. Saksena, J. da Silva and A. K. Agrawala, 'Design and implementation of Maruti-II', in Sang Son (ed.), *Principles of Real-Time Systems*, Prentice Hall, 1994.
8. J. D. Northcutt, *Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel*, Academic Press, 1987.
9. E. D. Jensen, C. D. Locke and H. Tokuda, 'A time-driven scheduling model for real-time operating systems', *Proc. IEEE Real-Time Syst. Symp.*, December 1985, pp. 112–122.
10. D. D. Kandlur, D. L. Kiskis and K. G. Shin, 'A real-time operating system for HARTS', *Proc. of Workshop on Operating Systems for Mission Critical Computing*, September 1989.
11. D. D. Kandlur and K. G. Shin, 'Reliable broadcast algorithms for HARTS', *ACM Trans. on Computer Systems*, **9**(4), 374–398 (November 1991).
12. H. Kopetz, *Event-Triggered Versus Time-Triggered Real-Time Systems*, *Lecture Notes in Computer Science*, Springer-Verlag, 1991.
13. H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft and R. Zainlinger, 'Distributed fault-tolerant real-time systems: The MARS approach', *IEEE Micro*, **9**(1), 25–40 (February 1989).
14. D. Mossé, 'Creating resilient real-time applications', *20th IFAC/IFIP Workshop on Real-Time Programming*, November 1995.
15. M. Russinovich, 'Application-transparent fault management', PhD thesis, Carnegie Mellon University, 1994.
16. S. Ramos-Thuel and J. K. Strosnider, 'Scheduling fault recovery operations for time-critical applications', *4th IFIP Conference on Dependable Computing for Critical Applications*, January 1994.
17. S. Ramos-Thuel and J. K. Strosnider, 'The transient server approach to scheduling time-critical recovery operations', *Real-Time Systems Symposium*, San Antonio, TX, December 1991, pp. 286–295.
18. R. Melhem and D. Mossé, Forts: Fault tolerance through scheduling in real-time systems. Available: http://www.cs.pitt.edu/FORTS.
19. S. Ghosh, D. Mossé and R. Melhem, 'Fault-tolerant rate-monotonic scheduling', *Journal of Real-Time Systems*, **15**(2) (September 1998).
20. H. Tokuda and C. W. Mercer, 'ARTS: A distributed real-time kernel', *ACM SIGOPS Operating System Review*, **23**(4), 29–53 (July 1989).
21. H. Tokuda and C. W. Mercer, 'The ARTS Kernel: Towards predicatable distributed real-time systems', *Proc. of Workshop on Operating Systems for Mission Critical Computing*, September 1989.
22. S.-T. Levi and A. Agrawala, *Real Time System Design*, McGraw-Hill, 1990.
23. X. Castillo, S. R. McConnel and D. P. Siewiorek, 'Derivation and calibration of a transient error reliability model', '*IEEE Trans. on Computers*, **C-31**(7), 658–671 (July 1982).
24. R. K. Iyer and D. J. Rossetti, 'A measurement-based model for workload dependence of CPU errors', *IEEE Trans. on Computers*, **C-35**(6), 511–519 (June 1986).
25. 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', *Proc. IEEE*, **66**(10), 1178–1199 (October 1978).
26. 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 (August 1986).
27. S. Ghosh, D. Mossé and R. Melhem, 'Implementation and analysis of a fault-tolerant scheduling algorithm', *IEEE Trans. Parallel and Distributed Systems*, **8**(3), 272–284 (March 1997).
28. W. Zhao and K. Ramamritham, 'Distributed scheduling using bidding and focused addressing', *Proc. IEEE Real-Time Syst. Symp.*, December 1985, pp. 103–111.
29. D. Verma, H. Zhang and D. Ferrari, 'Delay jitter control for real-time communication in a packet switching network', *Proc. TriComm.*, 1991, pp. 35–43.
30. S. Ghosh, R. Melhem and D. Mossé, 'Enhancing real-time schedules to tolerate transient faults', *Real-Time Systems Symposium*, December 1995.
31. K. Lin, S. Natarajan and J. W. Liu, 'Imprecise results: Utilizing partial computations in real-time systems', *Proc. IEEE Real-Time Syst. Symp.*, December 1987.
32. M. DiNatale and J. Stankovic, 'Dynamic end-to-end guarantees in distributed real-time systems', *Real-Time Systems Symposium*, 1994.
33. R. Rajkumar, *Synchronization in Real-Time Systems: A Priority Inheritance Approach*, Kluwer Academic, 1991.

