A Nonpreemptive Real-Time Scheduler with Recovery from Transient Faults and Its Implementation

Daniel Mossé, *Member*, *IEEE Computer Society*, Rami Melhem, *Fellow*, *IEEE*, and Sunondo Ghosh

Abstract—Real-time systems (RTS) are those whose correctness depends on satisfying the required functional *as well as* the required temporal properties. Due to the criticality of such systems, recovery from faults is an essential part of a RTS. In many systems, such as those supporting space applications, single event upsets (SEUs) are the prevalent type of faults; SEUs are transient faults and affect a single task at a time. This paper presents a scheme to guarantee that the execution of real-time tasks can tolerate SEUs and intermittent faults assuming any queue-based scheduling technique. Three algorithms are presented to solve the problem of adding fault tolerance to a queue of real-time tasks by reserving sufficient slack in a schedule so that recovery can be carried out before the task deadline without compromising guarantees given to other tasks. The first algorithm is a dynamic programming optimal solution, the second is a linear-time heuristic for scheduling dynamic tasks, and the third algorithm comprises extensions to address queues with gaps between tasks (gaps are caused by precedence, resource, or timing constraints). We show through simulations that the heuristics closely approximate the optimal algorithm. Finally, the paper describes the implementation of the modified admission control algorithm, the nonpreemptive scheduler, and a recovery mechanism in the FT-RT-Mach operating system.

Index Terms—Fault tolerance, operating system, real-time, scheduling, transient faults.

1 Introduction

Real-time systems (RTS) are those whose correctness depends on satisfying the required logical and functional *as well as* the required temporal properties. It has been recognized for at least a decade that real-time computing systems have to support real-time application programs by maintaining an environment that satisfies timing, reliability, and availability requirements [36]. They encompass operating systems designed and engineered to accommodate the requirements of mission critical systems [1]. For these systems, aside from the functional and timing correctness, the ability to provide noninterrupted service is also essential.

Due to the criticality of RTSs, threads in such systems must always finish within user-specified deadlines. Fault tolerance techniques provide for successful completion of tasks within the deadlines in spite of hardware and software failures [8], [15], [22], [30]. When a fault occurs, extra time is required to handle fault detection and error recovery. For real-time systems in particular, it is essential that the extra time be considered and accounted for on a per-thread basis. Methods explicitly developed for fault tolerance in real-time systems must take into consideration the number and type of faults, while ensuring that timing constraints are obeyed.

Tolerating permanent hardware failures naturally requires redundant processing elements [12], [16], but intermittent and transient failures can often be tolerated using task reexecution on the same hardware [11], [26], [16], which is typically less expensive. This approach was introduced in the recovery block approach of [31].

In this paper, we depart from the traditional approach and decouple runtime recovery from admission control (the procedure that accepts or rejects tasks) for fault tolerance. We assume that some nonpreemptive real-time scheduling policy is used and we focus on mapping a non-fault-tolerant nonpreemptive schedule of real-time threads to a fault-tolerant schedule. Our goal is to guarantee that a thread will complete within its deadline taking into account the time for error recovery.

1.1 Problem Definition

In our model, we start by considering only *transient and intermittent faults*, which are short-lived malfunctions in a hardware component, affecting at most one thread executing on that hardware component. These faults are common in many applications, such as satellite and space stations, where transient faults are called *single event upsets* (SEUs).

We say that a service executes *correctly* if it finishes within the specified deadline and delivers correct results with respect to the functional specification. Otherwise, we say that a *service failure* has occurred. We assume that all inputs occur at the beginning of thread execution, and outputs are generated only at the end of the threads, so that the whole thread can be reexecuted if it has to be aborted due to a fault. Any thread with input or output in the

- *D. Mossé and R. Melhem are with the Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260. E-mail: {mosse, melhem}@cs.pitt.edu.*
- *S. Ghosh is with I2 Technologies, 3700 Lillick Drive #313, Santa Clara, CA 95051. E-mail: sunondo@yahoo.com.*

Manuscript received 20 Oct. 1998; revised 20 Dec. 2002; accepted 11 Feb. 2003.
Recommended for acceptance by W. Sanders.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 108096.

middle of its execution can be broken into smaller threads to satisfy this condition [25], [28], [23].

We also assume error detection capabilities, such as those presented in [6], [13], [29]. Error checking is performed before any change to the environment takes place, and such changes are committed only if no error is detected. When an error is detected, a recovery action takes place. Possible actions include reexecuting the thread, executing a backup thread, executing a recovery block, or rolling-back to a checkpoint. In this paper, we are not concerned with specific recovery policies, but rather with providing a guarantee that the recovery will not cause thread deadlines to be violated, assuming that the worst-case execution time of the recovery action is known.

We consider a real-time system in which a scheduling policy imposes a total ordering on the threads, based either on the timing constraints of the threads (e.g., Earliest Deadline First [17]) or on their priorities (e.g., derived from their importance). This total ordering of threads is implemented in the form of a queue. The algorithms in this paper insert "slack" into that queue to create a fault-tolerant schedule which guarantees that recovery will not cause any thread to miss its deadlines.

When a thread, T_n, arrives in the system, it is inserted into the scheduling queue according to the scheduling scheme used; examples of algorithms that use a queue are Earliest Deadline First (EDF), Least Laxity First (LLF), First In First Out (FIFO), among others. In a fault-free environment, scanning the queue determines if each thread will meet its deadline. However, in a faulty environment, recovery from a fault during T_i's execution may cause the deadline of T_i or of the threads following T_i to be violated. This is because the task execution will be delayed by the time needed for the recovery of T_i.

We model a thread by a tuple $T_i = \langle a_i, r_i, d_i, c_i \rangle$, where a_i is the thread arrival time, r_i is its ready time (earliest start time of the thread), d_i is its deadline, and c_i is its worst-case execution time. For simplicity of presentation, we assume until Section 5 that $\forall i, a_i = r_i$.

We also assume that the worst-case recovery time for thread T_i is $\bar{c}_i$. The *window* of a thread is defined as $d_i - r_i$ and the *window ratio* is defined as $w_i = \frac{d_i - r_i}{c_i}$. It is assumed that $w_i \geq 1 + \frac{\bar{c}_i}{c_i}$ (i.e., $c_i + \bar{c}_i \leq d_i - r_i$) since, without this assumption, it is impossible to recover from a fault and still meet the time constraints. Applications with or without precedence constraints can be abstracted by the queue model used in this paper since the scheduling discipline will impose an ordering on the threads [5].

1.2 Road Map

In the following sections, we describe algorithms that guarantee fault recovery assuming that there exists a $\Delta_f > max_i\{c_i + \bar{c}_i\}$, such that no more that one fault occurs in any time interval of length Δ_f. If no such Δ_f exists, then no guarantees can be given since additional faults can occur during fault recovery. In Section 2, we describe an optimal algorithm that schedules backup slots in a queue of real-time threads, and in Section 3, we describe a greedy algorithm which approximates the optimal one. In Section 4, we evaluate the two algorithms and present simulation results to study the performance of the algorithms when faults are not necessarily separated by Δ_f (i.e., a violation of the fault assumption). In Section 5, we describe the enhancement to the greedy algorithm to consider threads with $r_i \geq a_i$, which may be the result of precedence and/or resource constraints; in that section, we also consider static and dynamic thread requests, negotiated levels of recovery, and periodic threads. Section 6 describes the implementation on the FT-RT-Mach system, including the performance and measured overhead of the system. Section 7 outlines related work while Section 8 concludes the paper.

2 Real-Time Guarantees in the Presence of Faults

Let Q_T be a queue of n threads $T_1, \ldots, T_n$ to be scheduled for nonpreemptive execution starting at the current time, t_0. In the absence of faults, if $\forall i, t_0 + \sum_{j=1}^{i} c_j \leq d_i$, then each thread T_i in Q_T will meet its deadline. In the presence of faults, however, some threads may need to recover and the time needed to complete i threads may be larger than $\sum_{j=1}^{i} c_j$. If le_i is the *latest end* of thread T_i (i.e., the time at which the first i threads in Q_T will complete execution even if faults occur), then T_i will meet its deadline if $le_i \leq d_i$.

A simple estimate for le_i, which allows each thread to recover from a fault is $t_0 + \sum_{j=1}^{i}(c_j + \bar{c}_j)$. This estimate is overly conservative if faults do not occur frequently. Specifically, if we assume that the time between any two faults is at least Δ_f, then it is possible to create a schedule that incorporates recovery slots that are separated by Δ_f and that can be used for fault recovery. That is:

$$le_i = t_0 + \sum_{j=1}^{i} c_j + \sum_{k=1}^{b} lb_k, \tag{1}$$

where $lb_1, \ldots, lb_b$ are the lengths of some slots $B_1, \ldots, B_b$ reserved for recovery.[1] These slots will be called "backup slots." Specifically, if the threads $T_1, \ldots, T_i$ are divided into segments (subsets) $\mathcal{B}_1, \ldots, \mathcal{B}_b$ such that $\mathcal{B}_1 = \{T_1, \ldots, T_{j_1}\}$, $\mathcal{B}_2 = \{T_{j_1+1}, \ldots, T_{j_2}\}, \ldots, \mathcal{B}_b = \{T_{j_{b-1}+1}, \ldots, T_i\}$, and, for $k = 1, \ldots, b$,

$$lb_k + \sum_{T_u \in \mathcal{B}_k} c_u \leq \Delta_f, \tag{2}$$

$$lb_k \geq max\{\bar{c}_u | T_u \in \mathcal{B}_k\}, \tag{3}$$

then le_i given by (1) is the maximum time needed to execute $T_1, \ldots, T_i$, including recovery time. Note that all threads in segment $\mathcal{B}_k$ are assigned to backup slot B_k for recovery.

In the following, we state and prove that the equations above are sufficient to provide guarantees for all threads and recovery from faults when needed.

Definition 1. *Let Q_T be a queue containing n threads at time t_0, that is, $Q_T = \{T_1, T_2, \ldots, T_n\}$. Let le_i be the latest end time of thread T_i in Q_T, assuming that threads recover when faults are detected. A* FT-feasible *schedule is a schedule in which $\forall i, le_i \leq d_i$.*

1. Note that the recovery slots do not use resources, but are simply a guarantee that the schedule will contain enough idle time (slack) for recovery from faults, if needed.

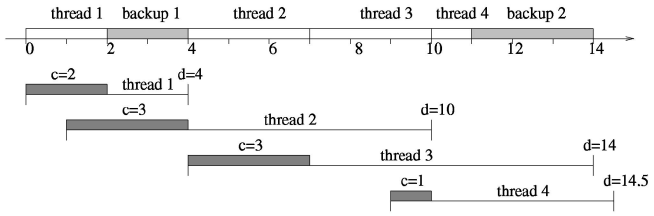


Fig. 1. A schedule which meets the deadlines of the threads in Example 1.

Theorem 1. *For all threads in Q_T, let le_i be computed from (1), (2), and (3), for a given Δ_f ($\Delta_f \geq max_i(c_i + \bar{c}_i)$). If at most one fault occurs in any time interval of length Δ_f, then Q_T is a FT-feasible schedule.*

Proof. Assume that a fault occurs during the execution of a thread in $\mathcal{B}_k$. Condition (2) specifies that there is enough slack for any one thread from $\mathcal{B}_k$ to recover if at most one fault occurs in any time interval of length Δ_f, and condition (3) specifies that the recovery of any thread in $\mathcal{B}_k$ will not require more time than lb_k, which is accounted for in the computation of le_i in (1). □

Note that the values of le_i computed from (1), (2), and (3) are not unique since the constraints in (2) and (3) do not have to be tight. Specifically, the sets $\mathcal{B}_j$ do not have to be the maximal sets that satisfy (2).

Example 1. We intuitively show how backups can be inserted into a queue conforming to Theorem 1, through an example: Threads in a queue are of lengths 2, 3, 3, and 1, their deadlines are 4, 10, 14 and 14.5, respectively; the EDF scheduling policy is used. Assuming that $\Delta_f = 10$, and that $\bar{c}_i = c_i$, Fig. 1 shows the placement of two backups in the queue creating a FT-feasible schedule. The recovery from a fault occurring in T_2, T_3, or T_4 will use the time "reserved" in backup 2. Note that the recovery of these threads will be carried out immediately after the fault is detected, thus postponing the execution of the following threads in the schedule.

We can interpret the insertion of backup slots as a mapping from a non-fault-tolerant queue of threads to a fault-tolerant one. Therefore, the problem we are trying to solve is: **Given a queue of threads, find a mapping which will lead to a FT-feasible schedule**; such mapping is a placement of backups in the schedule and an assignment of tasks to backup slots. Our solution is to apply a dynamic programming algorithm to a queue of threads by creating a layered graph as described next.

2.1 Construction of Graph

Given $Q_T = \{T_1, \ldots, T_n\}$, we construct a layered graph, G, to keep track of the possible positions of backup slots in the queue. The graph has several layers, with layer i corresponding to thread T_i. Layer i has several nodes, each representing a particular placement of thread T_i in the queue and its backup assignment. The jth node on layer i is denoted by $N_{i,j}$ and an edge between a node, $N_{i-1,j}$, in layer $i-1$ and a node, $N_{i,k}$, in layer i is denoted by $E_{i,k}^{i-1,j}$. In addition to the n layers corresponding to the n threads, we create a source node (at layer 0) and sink node (at layer $n+1$). Fig. 2 shows the graph corresponding to the four threads in Example 1.

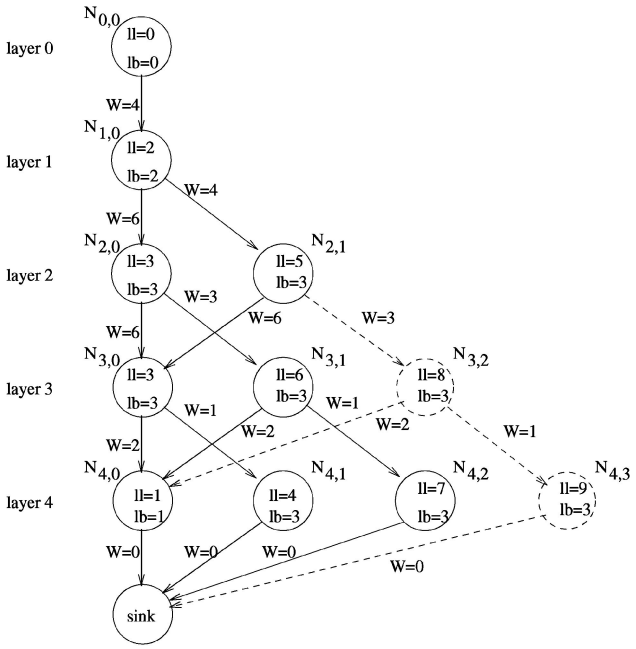


Fig. 2. The layered graph for Example 1. The dashed nodes violate (2) when $\Delta_f = 10$.

The nodes in layer i represent the possible options for placing T_i in the queue given that $T_1, \ldots, T_{i-1}$ are already on the queue. The queue will contain one or more backup slots, with one backup slot placed at the end of the queue. This slot will be called the "last backup." Given a queue configuration containing $T_1, \ldots, T_{i-1}$ and the corresponding backup slots, the first node $N_{i,0}$ in layer i corresponds to placing T_i after the last backup in the queue and creating a new backup slot after T_i. All other nodes in layer i correspond to the placement of T_i before the last backup (i.e., without creating a new backup, if the existing last backup can accommodate the new task). Hence, from each node $N_{i-1,j}$ on layer $i-1$, one edge leads to node $N_{i,0}$ and the other edge leads to a node $N_{i,j+1}$.

Each path in G corresponds to a unique placement of backup slots in the queue. For instance, consider the four tasks in Example 1. The queue configurations corresponding to three different paths in the graph of Fig. 2 are shown in Fig. 3. Fig. 3a shows the queue configuration for the path through nodes $N_{1,0}$, $N_{2,1}$, $N_{3,0}$, and $N_{4,1}$, where moving from node $N_{2,1}$ to node $N_{3,0}$ corresponds to placing T_3 after B_1 and creating a new backup. In Fig. 3b, we show the queue configuration for the path through nodes $N_{1,0}$, $N_{2,1}$, $N_{3,2}$, and $N_{4,0}$, where moving from node $N_{2,1}$ to node $N_{3,2}$ corresponds to placing T_3 before B_1. Fig. 3c is similar, showing path $N_{1,0}$, $N_{2,0}$, $N_{3,0}$, and $N_{4,1}$.

To check the deadline of a task, we use the notion of the *span of a queue configuration*, which is defined to be the completion time of the last thread in the queue associated with that schedule. Each edge in G is assigned a weight such that the weight of a particular path from the source node to the sink node is equal to the span of the queue

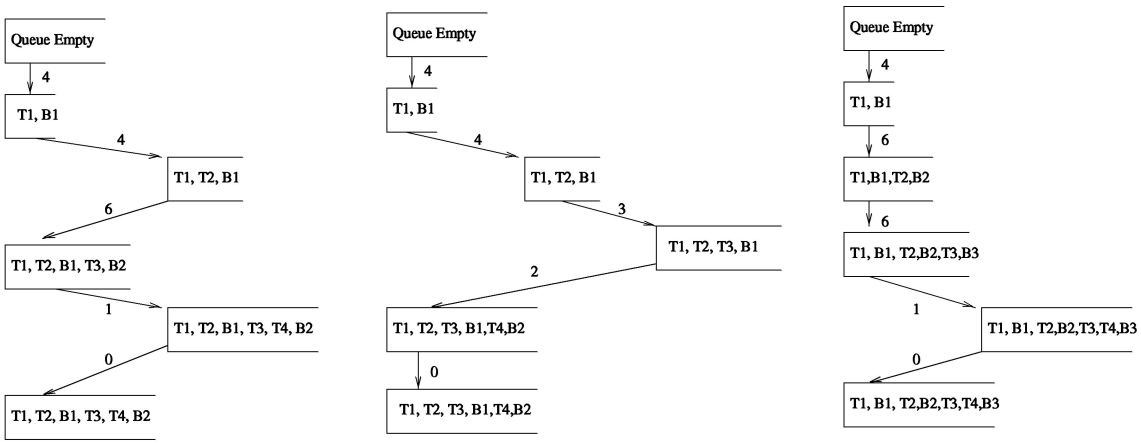


Fig. 3. Queue configurations corresponding to different paths in Fig. 2: (a) schedule span = 15, (b) schedule span = 13, and (c) schedule span = 17.

configuration corresponding to that path. For this, the weight of edge $E_{i,k}^{i-1,j}$, denoted by $W_{i,k}^{i-1,j}$, is set to be equal to the increase in the span of the queue due to the addition of thread T_i. The weight of an edge leading to the sink node has a value of zero (i.e., $W_{n,0}^{n-1,i} = 0$).

In order to simplify the derivation of the edge weights, we compute for each node $N_{i,j}$ in G a label $lb_{i,j}$, which is equal to the length of the last backup in the queue after the placement of T_i. Hence, $lb_{i,j}$ depends on $\bar{c}_i$, the time that needs to be reserved for the recovery of T_i. Specifically, if T_i is inserted after the last backup (node $N_{i,0}$), a new backup of length $\bar{c}_i$ is created and, thus, $lb_{i,j} = \bar{c}_i$. If, however, T_i is inserted before the last backup (nodes $N_{i,1}, N_{i,2} \ldots$), then that last backup increases in length only if it is shorter than $\bar{c}_i$. That is,

$$lb_{i,j} = \begin{cases} 0 & \text{if } i = 0 \\ \bar{c}_i & \text{if } i > 0 \ \& \ j = 0 \\ max(lb_{i-1,j-1}, \bar{c}_i) & \text{if } i > 1 \ \& \ j > 0. \end{cases} \quad (4)$$

The weight $W_{i,k}^{i-1,j}$ of an edge $E_{i,k}^{i-1,j}$ can, then, easily be computed from

$$W_{i,k}^{i-1,j} = \begin{cases} c_i + \bar{c}_i & \text{if } k = 0 \\ c_i + lb_{i,j+1} - lb_{i-1,j} & \text{if } k = j + 1. \end{cases} \quad (5)$$

In other words, if T_i is inserted after the last backup ($k = 0$) and a new backup is created, the span of the queue increases by $c_i + \bar{c}_i$. On the other hand, if a thread is inserted before the last backup ($k = j + 1$), the span of the queue increases by the sum of c_i and the increase in the length of the last backup, if any. The node labels and edge weights are shown in Fig. 2.

The graph constructed so far disregards Δ_f and takes only (3) into consideration and, thus, some nodes in the graph may correspond to queue configurations that violate (2). For the threads in Example 1, if $\Delta_f = 6$, only the leftmost column of Fig. 2 (one backup for each thread) of the graph will correspond to configurations that satisfy (2). If $\Delta_f > 12$, all the nodes in the graph will satisfy (2). To make sure that the span of the queue corresponding to each segment $\mathcal{B}_k$ does not exceed Δ_f, a parameter, $ll_{i,j}$, is computed for each node $N_{i,j}$ to reflect the length of the queue (in units of time) between the last two backup slots (not including either backup slot). For a node $N_{i,0}$ which corresponds to the addition of T_i after the last backup, $ll_{i,0}$ equals the computation time of T_i. For a node $N_{i,j}$, $j \neq 0$, which represents the addition of T_i before the last backup, the time between the last two backups increases by the computation time of T_i. That is, $ll_{i,j}$ is computed as follows:

$$ll_{i,j} = \begin{cases} 0 & \text{if } i = 0 \\ c_i & \text{if } i > 0 \ \& \ j = 0 \\ ll_{i-1,j-1} + c_i & \text{if } i > 1 \ \& \ j > 0. \end{cases} \quad (6)$$

By deleting from the graph the nodes which do not satisfy the condition $ll_{i,j} + lb_{i,j} \leq \Delta_f$, we make sure that (2) is satisfied for all nodes. If this condition is violated at a node $N_{i,j}$, thread T_i and a new backup are forced to be placed at the end of the queue after the last existing backup. The parent of the deleted node $N_{i,j}$ (where $ll_{i,j} + lb_{i,j} > \Delta_f$) has only a single outgoing edge, to node $N_{i,0}$. Thus, this condition restricts the number of nodes at each layer to the number of threads (along with their backup) that can fit within a length of Δ_f in the schedule. For example, in Fig. 2, nodes $N_{3,2}$ and $N_{4,3}$ are removed from the graph if $\Delta_f = 10$ (the dashed nodes and edges). In fact, by computing ll during the generation of the graph, we can avoid the creation of nodes that violate (2).

Our goal is to find the shortest path, ensuring $\forall i, 1 \leq i \leq n, le_i \leq d_i$ (see (1)).

2.2 Shortest Feasible Schedule (SFS)

In this section, we show that the layered graph allows us to find the shortest FT-feasible schedule in $O(n^2)$ time. We describe an algorithm and its proof of correctness.

Definition 2. *Let $m_1, \ldots, m_q$ be all possible mappings from a non-fault-tolerant queue to a FT-feasible schedule. An* optimal *mapping is a mapping m_j, $1 \leq j \leq q$, which minimizes the span of the queue configuration at the sink node.*

In order to find the optimal mapping, all possible placements of backups have to be considered such that the conditions of Theorem 1 are satisfied. Since both possible placements of threads (before or after the last backup) are represented in the graph, exploring all paths in

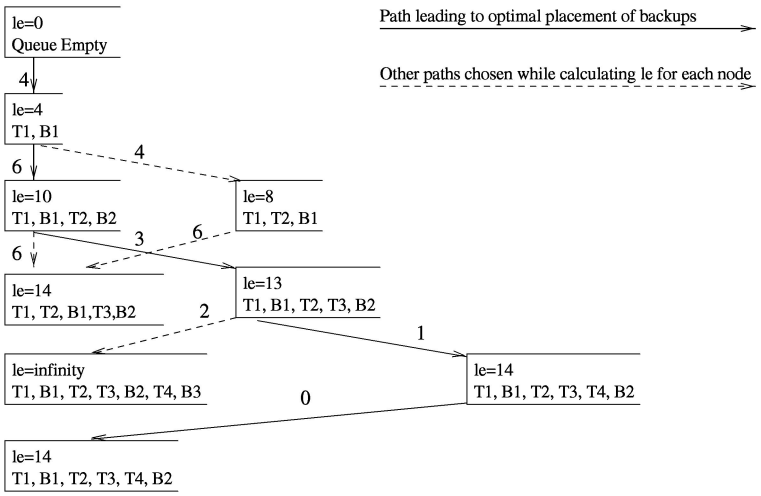


Fig. 4. The shortest queue (span = 14) and the corresponding le values for $\Delta_f = 10$.

the graph will lead to an optimal mapping. To find the path in the graph that leads to an optimal placement of backups, a dynamic programming algorithm is used. The specific sequence of threads and backups can be maintained at each node or the shortest path can be traversed in reverse order to find out the actual placement if needed.

To obtain the optimal mapping of backups in the queue, we associate with each node a value $le_{i,j}$, which corresponds to the minimum span of the queue up to node $N_{i,j}$. Starting with $le_{0,0} = 0$, we may thus recursively compute the following:

$$le_{i,j} = \begin{cases} 0 & \text{if } i = 0 \ \& \ j = 0 \\ min_{k=0}^{i-1}\left\{ le_{i-1,k} + W_{i,0}^{i-1,j-1} \right\} & \text{if } i > 0 \ \& \ j = 0 \\ le_{i-1,j-1} + W_{i,j}^{i-1,j-1} & \text{if } i > 0 \ \& \ j > 0. \end{cases} \quad (7)$$

This algorithm would be sufficient for a queue of non-real-time threads. However, in real-time systems, we also have to take the deadlines of the threads in the queue into consideration. If the deadline of thread T_i is not met when we add it to the queue, that is, if $le_{i,j} > d_i$, the placement of T_i and its backup corresponding to $N_{i,j}$ cannot lead to a FT-feasible schedule. In that case, we assign $le_{i,j} = \infty$ and, thus, that particular backup assignment is deemed infeasible. Consequently, all nodes $N_{i+k,j+k}, k > 0$ will also be infeasible since $N_{i,j}$ is the only parent of these nodes. Thus, the computation of $le_{i,j}$ should be supplemented by:

$$\text{IF } le_{i,j} > d_i, \text{THEN} \qquad le_{i,j} = \infty.$$

Finally, when we reach the nodes at layer n (after considering all n threads in the queue), all nodes with $le_{i,j} \neq \infty$ represent FT-feasible mappings of backups which satisfy $le_i \leq d_i, i = 1, \ldots, n$, where le_i is computed from (1), (2), and (3). Thus, if a feasible mapping exists, the algorithm finds the mapping.

As an example, in Fig. 4, we show $le_{i,j}$ values for nodes corresponding to Fig. 2 when $\Delta_f = 10$. Note that $le_{4,0}$ for node $N_{4,0}$ is assigned a value of ∞ since it would lead to $le_{4,0} = 15$, which means that T_4 would miss its deadline. Also note that, although the path given in Fig. 3b is shorter than the one given in Fig. 4, the former has to go through node $N_{3,2}$, which is removed from the graph when $\Delta_f = 10$.

2.2.1 Complexity and Optimality

Although in our presentation we separated the construction of G and the computation of the shortest path, it is possible to recursively compute the values of $lb_{i,j}$, $W_{i,k}^{i-1,j}$, $ll_{i,j}$, and $le_{i,j}$ for each node from (4), (5), (6), and (7) without explicitly constructing G.

The complexity of this computation is $O(n^2)$ in the worst case. This is because, at every layer of the graph, the algorithm examines each node once computing the values of $lb_{i,j}$, $W_{i+1,k}^{i,j}$, $ll_{i,j}$, and $le_{i,j}$. Since there are n layers, with at most i nodes at layer i and at most two edges for each node, we have to compute at most $O(n^2)$ values. However, in the average case, the complexity of the algorithm is lower since the number of nodes at any layer is specified by the value of Δ_f. If $ll_{i,j} + lb_{i,j} > \Delta_f$, the only edge from node $N_{i,j}$ leads to node $N_{i+1,0}$. On average, the number of nodes in a layer is thus equal to the number of threads that fit into a Δ_f interval. This is, in turn, equal to Δ_f / c_{av}, where c_{av} is the average computation time of the threads in the queue. Thus, the average runtime is $O(\frac{\Delta_f}{c_{av}} n)$.

The SFS algorithm is optimal because it considers all possible combinations of backup placements in the queue, as shown below.

Theorem 2. *By finding the feasible shortest path in G which satisfies $le_{i,j} \leq d_i$ at each node $N_{i,j}$, SFS is optimal in the following sense: If a mapping exists for a given queue with n threads that results in a FT-feasible schedule, the SFS algorithm will find the mapping. Further, if multiple mappings exist, the algorithm finds the mapping that minimizes the span of the queue.*

Proof. We first prove that if there is a feasible path in the graph, SFS will find it. This is true if all possible combinations of backup placements in the queue are considered. We have shown in the previous section that SFS does consider all backup placements and, thus, will always find a FT-feasible schedule if one exists.

```
0   ll = lb = 0 ; le_0 = 0 ;
1   for  i = 1 to n do {
2       if (ll + c_i + max{lb, c̄_i} > Δ_f) {    /* start new segment */
3          lb = c̄_i ;  ll = c_i;
4          le_i = le_{i-1} + c_i + c̄_i;
5       }
6       else {                                  /* add thread to existing segment */
7          le_i = le_{i-1} + c_i + max{lb, c̄_i} − lb;
8          lb = max{lb, c̄_i};
9          ll = ll + c_i;
10      }
11      if (le_i > d_i)                         /* violates deadline */
12         return (FT NOT GUARANTEED)
13  } /* end for */
14  return (FT GUARANTEED)
```

Fig. 5. Algorithm LTH (Linear Time Heuristic).

To prove that the length of the queue found at the sink node is minimum, we note that the shortest queue is given by the shortest path. Since there is a single edge leading into every node $N_{i,j}, 1 \leq j \leq i$ on layer i and the edge $(E_{i,0}^{i-1,k}, 0 \leq k \leq i-1)$ into node $N_{i,0}$ minimizes the queue length to reach $N_{i,0}$, the value $le_{i,0}$ of node $N_{i,0}$ is minimal. □

In a static environment, when all thread arrival times are known beforehand, the $O(n^2)$ complexity of the algorithm is acceptable. However, in a dynamic environment, where the thread arrival times are not known beforehand, a thread should be scheduled as soon as it arrives. This involves inserting the new thread into the existing queue of threads and guaranteeing that the new thread as well as all previously scheduled threads will meet their deadlines even in the presence of faults. In dynamic environments, a lower complexity algorithm should be used to find a FT-feasible schedule without actually building the entire graph. Below, we provide a linear time algorithm to insert backups into a queue of threads.

3 Linear Time Heuristic (LTH)

If Q_T contains threads $T_1, \ldots, T_n$, the algorithm LTH in Fig. 5 can be used to check if the non-fault-tolerant queue can be transformed into a FT-feasible schedule. LTH is equivalent to a greedy way of constructing a single path in the graph of Section 2.2. This path starts with the edge connecting node $N_{0,0}$ to node $N_{1,0}$. From then on, the path follows $N_{2,1}, \ldots, N_{i+1,i}$ until it reaches node $N_{j+1,j}$ that has only a single feasible outgoing edge to node $N_{j+2,0}$ (because $ll_{j+1,j} + lb_{j+1,j} > \Delta_f$). The path has to include this edge. From node $N_{j+2,0}$, the procedure is similar to the one starting at $N_{1,0}$. Finally, the last layer of the graph is reached, and the path ends at the sink node. In this algorithm, the variable ll is used to keep track of the length of the queue between the last two backups, lb is the length of the last backup, and le_i is the length of the queue when i threads have been considered.

Theorem 3. *LTH is correct in the sense that if it returns* FT GUARANTEED *and at most one fault occurs in time* $\Delta_f \geq max_i\{c_i + \bar{c}_i\}$, *then the fault recovery will not cause any thread to miss its deadline.*

Proof. Note that LTH is a special case of SFS, where only some paths are considered in a greedy fashion. This follows from Theorem 1 and the observation that le_i computed by LTH satisfies (4) in lines 3 and 8, (6) in lines 3 and 9, and (7) in lines 4 and 7. □

Since only part of the graph is actually created while running LTH, its complexity is lower than SFS. The worst-case analysis shows that LTH is linear in the number of threads in Q_T (line 1, FOR loop). LTH is greedy because it tries to provide a single backup for as many threads as possible. This may create a schedule which is longer than necessary, and thus may lead to an infeasible schedule. For example, if the EDF scheduling policy is applied to Example 1, with $\Delta_f = 10$ and LTH is used to place the backups, then the queue shown in Fig. 6 is created (corresponding to the path of Fig. 3b). We find that, when T_4 is added to the queue, $le_4 > d_4$ and, thus, LTH returns FT NOT GUARANTEED. This behavior is correct because if a fault occurs in this schedule,

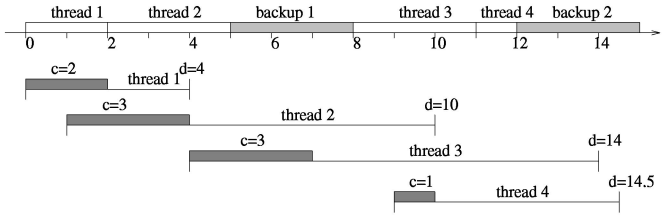


Fig. 6. The non-FT-feasible schedule found by LTH for Example 1.

TABLE 1
Percentage of Feasible Schedules that LTH Deems Infeasible

load	20 threads			50 threads		
	w=5	w=10	w=15	w=5	w=10	w=15
0.3	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0.4	0.3%	0.3%	0.1%	0.3%	0.4%	0.1%
0.5	0.1%	0.4%	0.5%	0.3%	0.2%	0.6%
0.6	0.6%	0.7%	0.5%	0.5%	0.4%	0.6%
0.7	0.3%	0.5%	0.2%	0.3%	0.3%	0.1%
0.8	0.1%	0.3%	0.2%	0.0%	0.1%	0.1%
0.9	0.3%	0.1%	0.1%	0.0%	0.0%	0.0%
1.0	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%

say at time 10.7, T_3 has to recover and T_4 misses its deadline. In this case, LTH fails to find a backup placement when SFS is successful by placing the backup after T_1 (see Fig. 1).

4 SFS and LTH: Evaluation and Analysis

In this section, we present the results of the simulations that were conducted to evaluate SFS and LTH. First, we present a comparison of the two algorithms. SFS is an optimal algorithm, but its complexity is higher than LTH, which is a suboptimal linear time algorithm. Since simulation results show that LTH is very close in terms of schedulability to the optimal SFS, we do not include comparisons with other algorithms. Then, we analyze in depth LTH's performance. We study three scheduling policies in combination with LTH and then select the one with the best results for further evaluation. In all the simulations presented here, we assume that $\forall i, \bar{c}_i = c_i$.

4.1 Comparison of SFS and LTH

Given a queue of threads, we want to determine the loss in performance of LTH in comparison to SFS. We determine the number of times that SFS is able to find a FT-feasible schedule for a queue of threads when LTH fails to do so. To compare the two algorithms, in our simulations, we considered queues containing sets with different number of threads. We generated 1,000 sets of threads for each combination of parameters (such as the load, number of threads in the queue, window ratio w, etc). We found that in the worst case, LTH rejects up to 0.7 percent more threads than SFS (Table 1).

We also found that the small difference in performance depends on the load. If the system is lightly or heavily loaded, then the two algorithms perform almost identically. However, for medium loads (around 0.5 or 0.6), the difference increases slightly. We present in Table 1 the difference in the percentages of thread sets found feasible by SFS and LTH, for 20 and 50 threads in the queue, and for varying average window ratios (recall that $w_i = \frac{d_i - r_i}{c_i}$). The behavior for other values are very similar to those shown in Table 1. We conclude that LTH approximates SFS very well in finding FT-feasible schedules.

4.2 Evaluation of LTH

In dynamic systems, if recovery cannot be guaranteed for a thread when it arrives, it is *rejected*; the user can then abort the thread, continue without fault tolerance, or take some alternative recovery action (see Section 5). A thread which is accepted but misses its deadline is called a *lost thread*. In our approach, threads are only lost because the faults occurred more frequently than expected (violating the assumption that no two faults can occur within an interval of length Δ_f).

Since the two metrics, loss and rejection, are intertwined, in some graphs we combine them so that they can be studied together instead of independently. Specifically, it is clear that the trade off between schedulability and thread loss depends on the importance of each thread (i.e., the cost of missing a deadline). We use a user-defined parameter Ω to represent the ratio of the **cost of losing a thread** after accepting it and the **cost of rejecting** it (i.e., not accepting it when submitted). Thus, we plot some graphs (see Figs. 9 and 10) for the following cost function:

$$\begin{aligned} totalcost = {} & cost\ of\ rejected\ tasks \\ & + \Omega\ \times cost\ of\ lost\ tasks. \end{aligned} \tag{8}$$

Note that a low value of Ω means that missing a deadline is not very critical; for critical-mission systems, system designers should choose a very high value of Ω.

As expected, the simulation results show that the error recovery (i.e., backups) decreases the number of lost threads at the cost of increasing the number of rejected threads. The first goal of our simulation is to estimate this trade off. In addition to the number of rejected threads, we also look at the success of the algorithm from the perspective of lost threads in comparison to the *No-FT* method, in which no backups are added to the schedule.

Another goal of the simulation is to determine the load at which the number of threads rejected and lost are below specified percentages. The system designer can analyze the characteristics of the threads (perhaps with dynamic arrivals) to determine the average window ratio and computation times of those threads. Once the scheduling policy is determined, then the maximum allowed load for a specified schedulability and a specified rate of lost threads can be determined for a given Mean Time To Fault (even though the term MTTF typically refers to *failures*, we use it to refer to the interval between detected transient errors).

4.3 Simulation Parameters

We developed a discrete-event simulator where the events driving the simulation are the arrival, start, and completion of a thread as well as occurrence of faults. To evaluate LTH, for each parameter combination, we generated 100 thread sets of 10,000 threads each and ran each policy on the thread sets, averaging the results. The simulation parameters that can be controlled are (value ranges used in the simulation are between brackets):

- **scheduling discipline**: Many scheduling disciplines can be used; we include the nonpreemptive Earliest Deadline First (EDF), Least Laxity First (LLF), and FIFO.

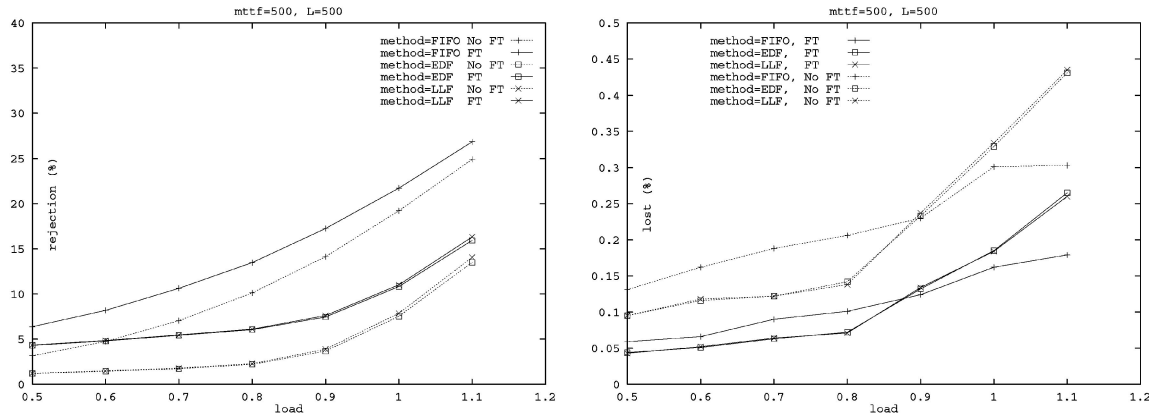


Fig. 7. Percentage of lost and rejected threads as a function of load.

- **average computation time**, c_{av}: The thread computation time is assumed to be uniformly distributed with mean c_{av}. $[= 5]$.
- **load** γ: The thread arrival is Poisson distributed with rate λ_T. The load is then $\gamma = \lambda_T c_{av}$. $[= 0.5, 0.6, \ldots, 1.1]$.
- **maximum window ratio** w_{max}: The window ratio is uniformly distributed between 2 and w_{max}. $[= 5, 10, 15, 20]$.
- **fault interval** $MTTF$: The fault interarrivals are Poisson distributed with mean MTTF. $[= 250, 500, \ldots, 2000]$.
- **length of segment** L: The maximum length of backup separation used in algorithm LTH. $[= 2c_{max}, \ldots, 1.5\text{MTTF}]$.

We ran the experiments for different w_{max} values, but chose to show only $w_{max} = 15$ since the behavior of other values of w_{max} were similar. Note that, if LTH is used to accept threads into the system, a thread can only be lost due to deadline miss if more than one fault occurs within an interval of length L. Note also that L can be larger or smaller than the MTTF; we experiment with this range of values to assess how large the backup separation interval should be.

In the simulations, whether fault tolerance is taken into consideration when a thread is accepted into the system (LTH) or not (No-FT), the thread in which an error is detected is reexecuted. Another possible recovery scheme for No-FT is to simply drop the faulty thread. The advantage of such a "recovery" scheme is that of preventing a domino effect of missed deadlines during transient overloads [10]. We do not consider dropping faulty threads for two reasons:

- Our admission control maintains the system below overloads, even for high offered loads.
- The increase in the number of lost threads would be prohibitive (see discussion below).

4.4 Analysis of Results

We start by analyzing the behavior of LTH in relation to the load with and without error recovery capabilities. Fig. 7 shows the percentage of threads rejected and lost for three scheduling policies, EDF, LLF, and FIFO. The FIFO policy causes more threads to be rejected as compared to the other two and also causes more threads to be lost for lower loads. However, for higher loads, the EDF and LLF policies cause more threads to be lost. This is because the FIFO scheme rejects more threads and hence the system has a lighter load and, thus, a lower number of threads are lost when faults occur.

Even though the percentage of threads rejected by LTH is higher than No-FT for each of the three scheduling policies, the percentage of threads lost is significantly lower. Also, note that the value of L is chosen to be equal to MTTF in Fig. 7. If the number of lost threads is required to be lower, then the value of L has to be smaller. The number of lost threads will approach 0 as L approaches the average length of a thread (which would mean a backup for every thread).

Further, to show why dropping threads is not considered, note that, if we drop a faulty thread, we would lose one thread every MTTF, on average. This translates into a loss increase of $\frac{c_{av}}{MTTF}\ \gamma$. In the graph on the right of Fig. 7, the lost thread percentage would increase by at least double the amount compared to the other recovery schemes (e.g., it would increase by 1 percent for load of 1).

From Fig. 8 on, we will study only the EDF scheduling policy. This is because EDF is more appropriate than FIFO for real-time systems, and the results we obtained for EDF and LLF are almost identical. Fig. 8 shows the percentage of threads rejected and lost for varying values of L as a factor of MTTF. We express L as a fraction of MTTF to be able to compare several different values of L and MTTF in the same graph.

The percentage of threads rejected decreases as the value of L increases since a larger value of L causes fewer backups to be placed in the queue and, thus, more threads can be accepted. On the other hand, the number of threads lost increases with L.

It is interesting to note the difference in rejection rate for the various loads and that the percentage of rejected threads has little variation for varying values of MTTF, for each combination of parameters (e.g., for load = 1.0 or 0.5). This is because threads are mostly rejected due to their timing constraints and not due to the frequency of faults. The two cases in Fig. 8 show that the variation for load = 1.0 is higher

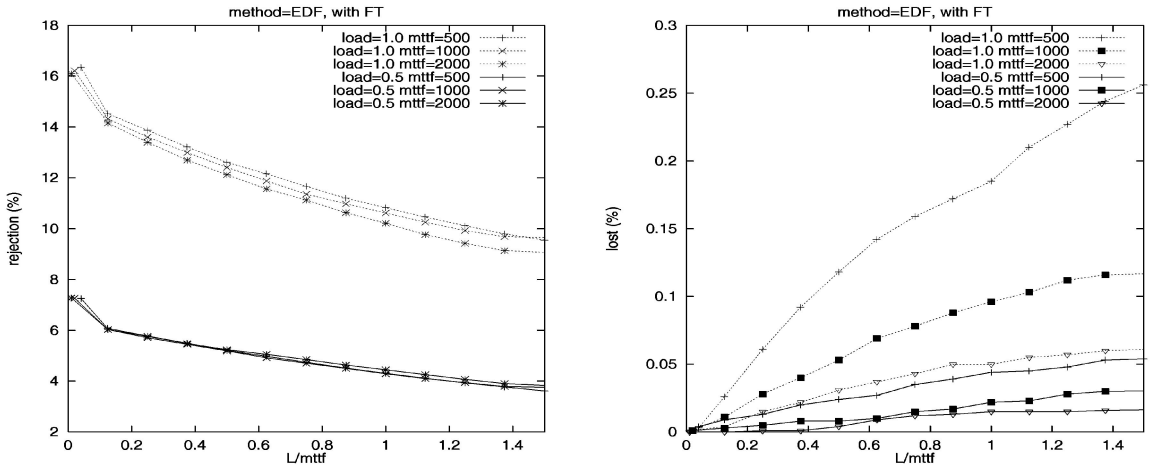


Fig. 8. Percentage of lost and rejected threads for values of L.

than for load = 0.5. For high loads (such as 1.0), when the MTTF is smaller, more backups are reserved and there is less time to schedule threads (increasing slightly the rejection rate). For lower loads, although there are still many backups for small MTTF, the unused processor capacity is still able to accommodate the incoming threads. As for the number of lost threads, it decreases with increased values of MTTF. Also, a higher value of load causes more threads to be lost for the same ratio of $\frac{L}{MTTF}$. Hence, the number of lost threads is a function of L, MTTF, and the load, and there does not seem to be a specific recommended ratio of $\frac{L}{MTTF}$ which is independent of load.

In Figs. 9 and 10, we plot the *totalcost* as defined in (8) for varying values of L and Ω (recall that Ω is the cost ratio of lost and rejected threads). When $\Omega = 0$, the graphs simply show the percentage of rejected threads. However, if $\Omega > 0$, then there is a cost for missing the deadline of an accepted thread and the cost increases with Ω. If the lost threads can cause a catastrophe, the value of Ω is very large, which means that it is better to reject a thread than to accept it and subsequently lose it. Whenever the value of Ω is small, it is preferable not to provide error recovery capabilities at all since the lost threads are not costly, and the number of rejected threads is smaller. However, for larger values of Ω, it is essential to provide error recovery capabilities.

In Fig. 9, we show the total cost as a function of L for different values of Ω. For each value of Ω, No-FT is represented by the straight dotted lines due to its independence of L. When $\Omega = 0$, the total cost (= rejection) decreases as L increases, and No-FT performs better than LTH. As L increases, the total cost of LTH approaches the cost of No-FT. When the value of Ω increases to a certain threshold (slightly less than $\Omega = 150$ in the figure), the total cost becomes almost independent of L; this can be seen by the approximately flat curve of $\Omega = 150$. When Ω increases beyond this threshold, the total cost increases monotonically with L (except for the initial drop, which is caused by an overallocation of backups) and LTH performs better than No-FT. We can also see that when the load is low ($\gamma = 0.5$, left graph), the increase in total cost with L is slower than for a higher load ($\gamma = 1.0$, right graph). Again, since fewer threads are scheduled, fewer threads are lost.

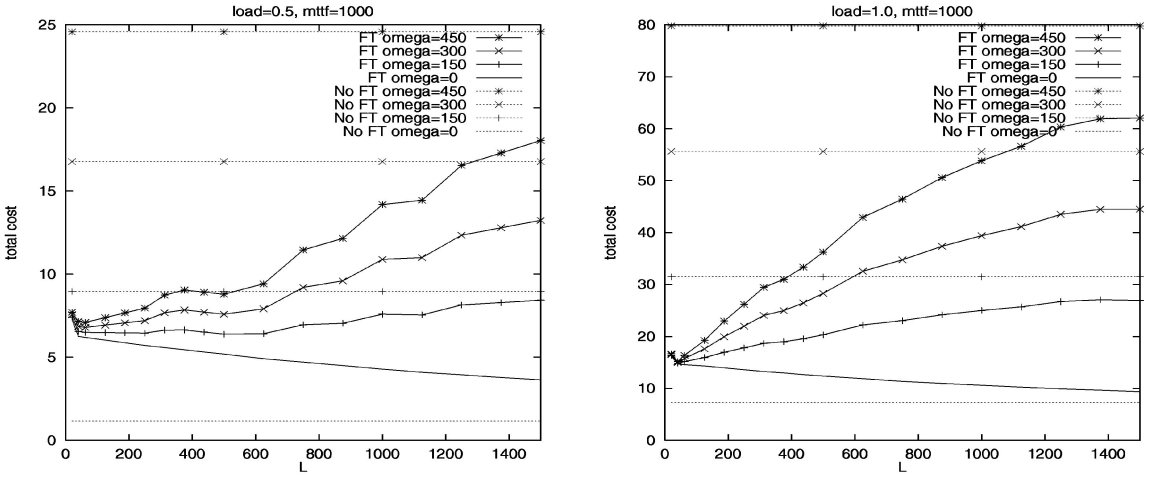


Fig. 9. Total cost for varying values of L and Ω.

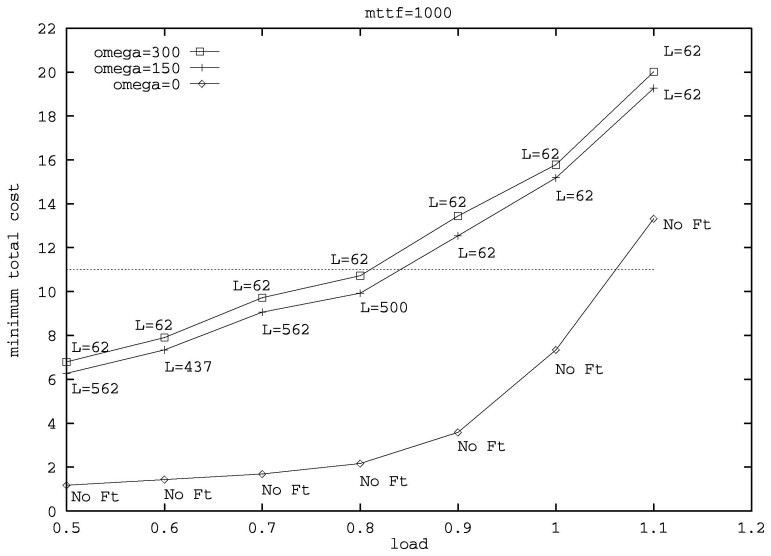


Fig. 10. Total cost with optimal L versus load.

In Fig. 9, there is an optimal value of L that minimizes the total cost of the system, for a given Ω. In Fig. 10, we plot this minimum total cost versus load for varying values of Ω. For a given load and Ω, the total cost varies with L. We find the value of L for which the cost is minimum and plot that value of cost in the graph. The corresponding value of L is specified beside each point in the graph.[2] For small values of Ω, the minimum cost is reached when $\mathrm{L} = \infty$. This is equivalent to the cost when fault tolerance is not provided because a very large L would result in no backups being placed in the queue. So, the curve for $\Omega = 0$ is also the curve representing No-FT. As Ω increases, the value of L at which the cost is minimized decreases. In the figure, we see that, for $\Omega = 150$ and $load \leq 0.8$, the cost is minimum for values of L around MTTF/2. As the load increases, the value of L at which the cost is minimized decreases to MTTF/16. For higher values of Ω (e.g., 300 in the figure), the cost is always minimized at a small value of L (e.g., MTTF/16 in the figure). However, this cost is not minimized at the smallest possible value of L, which is $2c_{max}$. This is because, when L is very small, the number of lost threads is almost 0, but the number of rejected threads increases sharply.

This graph can be used by a system designer to determine the load that can be supported by the system given the value of Ω, and the percentage of rejection and lost threads that the system can tolerate. The value of Ω is determined by the system designer as the number of threads that can be rejected in order to prevent the loss of one thread (by providing guaranteed fault tolerance to that thread thus preventing it from being lost after being accepted). For example, consider a system in which the system designer determines that the rejection can be up to 5 percent, the system can lose up to 0.02 percent of the threads guaranteed with fault tolerance, and the value of Ω for this system is 300. In this case, the total cost is equal to 11 $(= 5 + 300 \times 0.02)$ and, using Fig. 10, we see that a load of less than or equal to 0.8 can meet these specifications if $L = 62$.

2. Each L in the graph is a multiple of $\frac{MTTF}{16}$.

In summary, the following observations can be made from the experiments.

- Given a ratio Ω, system designers can decide what the load in a processor should be, for a given total cost. That is, a system designer can determine the set of threads to be allocated to a particular processor, so that the cost threshold is not exceeded. This is also useful for resource allocation in distributed systems.
- As expected, nonpreemptive EDF with recovery performs better than FIFO, for all Ω values.
- The optimal backup separation depends mainly on the Ω ratio, but it also varies depending on the load; this is due to the existing slack when the processor is underloaded. Further, for high loads, the minimum L is not the best value for backup separation.
- The simple recovery scheme of reexecuting the entire thread when an error is detected is practical enough to warrant its use in embedded and other real-time systems.

5 Scheduling Extensions

There are some simple extensions to our algorithm that allow

1. handling existing idle intervals,
2. dynamic thread arrivals,
3. negotiation of fault tolerance degree, and
4. periodic threads.

Accounting for Gaps in LTH. When threads are not scheduled back-to-back, there will be idle times in the non-fault-tolerant schedule, which we call *gaps*. These gaps in the schedule can be due to resource, precedence, or timing constraints [40], [41]. The algorithm of Section 3 can be easily extended by taking advantage of the existing gaps in the schedule to be able to reduce the amount of inserted slack (backups) to create the fault-tolerant schedule. The main idea is to keep track of how much idle time the gaps provide for fault tolerance.

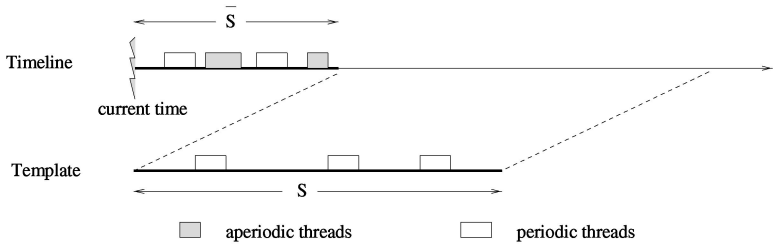


Fig. 11. Scheduling periodic threads on a timeline.

Dynamic Thread Arrivals. When considering a newly arriving thread, the scheduling policy will traverse the queue and find the position, pos, in the queue to insert the new thread. Algorithm LTH (with or without gaps) can be modified so that they start from the correct thread (i.e., pos). The loop (line 1) would simply become `For i =` pos `to` n.

This modification does not affect the algorithm correctness since all other threads (those prior to pos) will not be modified in the schedule. If, because of the new task, some task may miss its deadline, the newly arrived task is rejected and the schedule reverts back to the original one. Note that the average runtime for scheduling dynamic tasks is shorter than the original algorithm, since, on average, each thread will be typically inserted toward the end of the queue.

Negotiating Fault Tolerance. The probability of tolerating a fault is inversely proportional to the backup separation and gives an idea about whether or not the guarantees provided to the user will be valid during thread execution. Clearly, if the separation between backups is large, the probability of tolerating faults is low. In this case, two faults in quick succession may lead to a thread missing its deadline. On the other hand, if the backup separation is small, then frequent faults can be tolerated. In general, for critical threads, the backup separation is required to be low, while for less critical threads, it can be high.

It is possible that the backup separation required by the user leads to an infeasible schedule because there are too many backups in the queue. In such a situation, instead of rejecting the thread set outright, the user can be allowed to negotiate the value of backup separation. To make the choice easier for the user, SFS can provide the smallest value of backup separation which will generate a FT-feasible schedule for all threads in the system. This can be done by doing a depth first search on the graph created in Section 2.1.

Periodic Nonpreemptive Threads. Many current real-time systems are control systems for embedded or plant control applications. These systems typically have a heartbeat (control loop time) and perform several tasks at every heartbeat. The scheduling of each one of these cycles is relatively easy to perform if all threads have harmonic periods. However, when this is not the case, a better approach is needed; below, we present a possible extension to encompass periodic nonpreemptive threads.

It is possible to accommodate periodic threads by generating a schedule for the length of the LCM (least common multiple) of the periods of all threads; this becomes computationally complex, depending on the periods of the threads. Another approach is to find a smaller time interval that enables the scheduling of the minimum computational requirements of each thread (in a scheme similar to [4]); we refer to a *template* when talking about this interval of time with threads. A third approach is to allow *the user to specify* the template size according to the application. Note that the first or second approaches can be used offline to provide the template size for use in the third approach; thus, we assume a fixed template size.

In the following description, we call the scheduling queue a *timeline*; the timeline is an ordered queue of threads with start and end times and can include gaps. Assuming a template size S, a periodic thread with period P_i should be scheduled $\lceil S/P_i \rceil$ times on the template within the intervals $0, P_i, 2P_i, \ldots$. Although the timeline should consist of repeated copies of the template, only one copy of the template is kept on the timeline, thus limiting its span to S. When the execution reaches the end of the timeline, a new instance of the template is copied onto the timeline. Assuming that a new periodic thread is submitted to the system when $\bar{S} < S$ time units are yet to be executed on the timeline (see Fig. 11), then $\lfloor \bar{S}/P_i \rfloor$ instances of the thread should be scheduled on the timeline, and $\lceil S/P_i \rceil$ instances of the thread should be scheduled on the template. With this approach, if $\bar{S}$ is not a multiple of P_i, then the first instance of the thread may be slightly delayed, to make the last instance on the timeline meet the end of time interval $\bar{S}$. The algorithm described earlier for scheduling aperiodic threads can be used to schedule the instances of the periodic thread, and the thread is accepted only if all $\lfloor \bar{S}/P_i \rfloor + \lceil S/P_i \rceil$ instances are successfully scheduled.

Timeline management is uniform even when both periodic and aperiodic threads are to be accommodated in the system concurrently. However, when both types of tasks are present, the timeline should be expanded to accommodate the last scheduled thread. The naive way of achieving this will always maintain a timeline span that is a multiple of S, extending the schedule span so that the time from task submission to task occurrence may be larger than S. When the span of the timeline is to be extended to accept a new aperiodic thread, it is extended by an integer multiple of S and the template is repeatedly copied onto the extension of the timeline before the aperiodic thread is scheduled.

A more sophisticated scheme that saves space by not keeping many instances of periodic threads in the timeline can be used for timeline management. The system keeps a list of aperiodic threads accepted by the system and a set of templates that accommodate each aperiodic thread (i.e., a

template of size S that encompasses the execution window of each aperiodic thread). When a new thread is submitted to the system, if no existing template encompasses the window of the new aperiodic thread, our scheduler creates a template and adds all instances of aperiodic threads to the template. Clearly, the scheduler will first check whether the new aperiodic thread can be scheduled without missing its deadline and without causing other threads to miss their deadlines. Analogously, when a periodic thread is submitted, all existing templates must be checked before accepting the new periodic thread. If the new task causes any deadline to be missed, the task is rejected. This is the scheme adopted in our implementation.

6 Recovery from SEUs in FT-RT-Mach: A Nonpreemptive Implementation

The original RT Mach contains two separate modules for preemptive thread management, namely, a *scheduling policy* and a *dispatcher*, which is based on dispatching the highest priority thread ready to run (there are 32 priority levels [38], [27]). Software timers are used to transition threads between states.

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 the beginning of every thread period, the start timer for the thread goes off and the new instance of the thread is placed in the ready queue.

When an executing thread terminates, it is removed from the ready queue. For periodic threads, start timers are set to the beginning of the next period. Further, an *end timer* exists for each thread: If a thread does not terminate by its deadline, the end timer goes off and invokes a deadline handler thread. The code for the deadline handler is supplied by the user, who has a choice of resuming or killing the thread.

6.1 Adding the Timeline to RT Mach

In order to implement a time-driven nonpreemptive scheduler, several modifications to the kernel data structures, procedures, and API were needed. We call our nonpreemptive policy the *fault-tolerant* (FT) *timeline* (TL) policy. To guarantee that a TL thread will not be preempted under any circumstance, we execute TL threads at the highest priority level in the system. Priorities of all other threads are shifted so that only TL threads can execute at priority 0. To implement the above algorithm, we create a separate data structure in the kernel, which represents the current state of the timeline. Submitted threads are inserted in this timeline (according to an EDF policy) only if they are accepted to the system (see Section 5). In other words, accepted threads are ordered by their deadlines.

To start a TL thread, a timer is created and armed for the start time of the thread (minus the time it takes to handle the timer).[3] When this start timer goes off, the thread is placed in the run queue of priority 0. To simplify and shorten the time of handling the timeline, we reduce the number of timers armed by enforcing that only the thread at the head of the queue has armed timers. In contrast with the original RT-Mach mechanism, the start timer of the thread may change after the thread is admitted. For this reason, we chose to only arm timers of the thread at the head of the timeline. Clearly, this solution has the shortcoming 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 aperiodic TL thread terminates, it is removed from the run queue and from the timeline data structure. Then, timers of the next thread are armed and when its start timer goes off, the thread is moved to the ready queue. Using this approach, we guarantee that the top priority ready queue contains a maximum of one process, namely, the process at the head of the timeline. This bypasses the preemptive nature of the kernel. Periodic thread termination is handled in a way similar to the original RT-Mach.

6.2 Adding Fault Tolerance

To provide fault tolerance through reexecution, we also modified the criteria for accepting a thread. Our admission criteria implements the algorithm in Section 5, taking into consideration both gaps and the slack needed in the system for any one thread to reexecute within the interval Δ_f.

Further, we have also implemented a mechanism for thread recovery (in our implementation, we use reexecution as the recovery technique). Upon termination, the thread *exit* procedure checks a *fault flag*. If the flag is not set, the kernel kills the thread by disarming and destroying its timers, marking its memory as free, and giving back allocated space. If the flag is set, the kernel reexecutes the thread by reseting the start and end timers and restoring the context. In case of reexecution, the timers of the next thread will not be armed until appropriate. This is because threads may be shifted in time and, thus, their timers may need to be adjusted.

In this work, we only provide fault tolerance to user-level applications and not to operating system services.

6.2.1 Error Detection and Injection

Our recovery mechanism relies on an error detection mechanism which sets the *fault flag*. Our error detection is done at three different levels, namely, *hardware level*, *system level*, and *user level*. Since our work does not focus on the hardware, we simply take advantage of the inherent, built-in hardware error detection mechanisms, such as bus errors, divide-by-zero, and the like. These typically trap to the operating system, which in turn sets the fault flag. We have implemented a fault injection mechanism for testing purposes that modifies some memory location in the address space of the running thread. With this fault injection mechanism, we have verified that the hardwar-elevel error detection is indeed working correctly.

At the system level, we use the end of execution timers to detect timing faults. The fault flag is set and a recovery

3. A end-of-execution timer is also created and armed; unlike the original RT Mach, this timer is not set to go off at the deadline, but it is set to go off at the end of the execution of the thread.

TABLE 2
Kernel Execution Times Per Instance of Persistent FTTL and Periodic RMS Threads

	creation (ms)	activation and maintenance (ms)		
		load=15%	load=30%	load=45%
APERIODIC	2.62	2.593	2.558	2.581
PERIODIC	2.58	0.111	0.109	0.113
PERSISTENT	2.53	0.112	0.121	0.114
RMS	2.38	0.099	0.106	0.113
FT-RMS	2.38	0.101	0.123	0.131

policy is invoked. Under the current implementation threads that overrun their worst case execution times more than once are not reexecuted. By doing so, we avoid wasting system resources and let other threads execute within their timing constraints. However, a different policy can be used, combined with the slack inserted in the schedule: after detecting a temporal error, such threads can be reexecuted or allowed to execute for a longer period of time, such as in the imprecise computation paradigm [19], [18].

Lastly, to provide *user-level* error detection and the possibility of fault injection, we implemented a new system call to set the fault flag. This way, the program itself can check the validity of results and trigger thread recovery. The fault flag is reset after recovery and a mechanism that precludes the continuous setting of the fault flag has been implemented (to avoid user mistakes such as continuously reexecuting the thread).

6.3 Periodic Threads

The periodic scheme suggested in Section 5 was also included and tested in the FT-RT-Mach. However, instead of simply implementing the straightforward periodic templates, we provide users with several templates for scenario or mode changes [24] natively in the real-time operating system; clearly, we keep a single *active* template. New templates and activations of existing templates can be done at the user level, through system calls.

The active template is copied to the timeline when the timeline becomes empty or when an aperiodic thread with a deadline beyond the tail of the timeline is submitted for scheduling. There may be one or more copies of the previously active templates already scheduled on the timeline. We implemented the `fttl_thread_create` system call to create periodic threads. When a periodic thread is created using `fttl_thread_create`, it is first scheduled on the timeline starting at the first template boundary. Then, it is scheduled on the currently active template. If there is no copy of a template on the timeline or if the execution point of the timeline has proceeded beyond the first thread of the only template on the timeline, the new periodic thread is not scheduled directly on the timeline. Instead, it is scheduled on the active template. The next time the template is copied to the timeline the new thread will be scheduled for execution along with the rest of the template.

Aperiodic threads are always scheduled on a timeline which has been primed with enough copies of the active template to overlap the execution time of the aperiodic thread. This guarantees that the aperiodic thread will not conflict with a periodic thread. The shortcoming of this approach is that the scheduling of a single aperiodic thread for some distant future time will cause multiple copies of the active template to be added to the timeline. Although our current implementation does not optimize this, it is easy to see that just adding templates to times *around* the aperiodic thread would suffice.

6.4 Performance of FT-RT-Mach

We ran some experiments to measure the execution time of the new kernel functions. The main functions we are interested in are those associated with the creation, activation, and maintenance[4] of threads. These experiments were conducted on a Pentium 200MHz. We used the on-chip Pentium clock to obtain accurate measurements of the execution time of the threads and of the new kernel functions.

We compare the runtimes for three timeline nonpreemptive implementations (aperiodic timeline threads, periodic timeline threads, and aperiodic persistent[5] threads) and the RT Mach preemptive implementations (RMS and its fault tolerant version, FT-RMS). FT-RMS is an RMS derivative that allows for recovery from transient faults by creating enough slack in the system for error recovery to happen before the task's deadline. The theory behind FT-RMS is described in [9] and the implementation is described in [21]; our simulation results have shown that the FT-RMS scheme allows for more tasks to be accepted in the system and less tasks to be lost after acceptance. We include the results from the FT-RMS scheme here so that we can use it as the baseline for comparisons. Periodicity of the aperiodic timeline threads was obtained by creating a persistent thread once and activating it at the beginning of each period through a timer. All experiments were run on the same kernel by selecting the scheduling policy appropriately.

Ten threads were created with varying periods (between 30 and $120ms$); the thread utilizations were changed to obtain different system loads between 0.15 and 0.45 (the system load is limited to 45 percent due to the required fault tolerance). The execution time of kernel functions while running this task set for 120 seconds is shown in Table 2.

From this table, it is clear that the aperiodic implementation of periodic threads is prohibitively costly since, at every instance that is initiated, a thread creation takes place.

4. This includes exit and context switch times for each thread. When a thread exits, it has to return all resources to the kernel.

5. *Persistent threads* are aperiodic threads that remain resident in the system even after completion.

However, the periodic and persistent threads have comparable runtime to the RMS and FT-RMS threads, for all system loads tested (negligibly higher creation time, and comparable maintenance—around 10-15μs). Indeed, for higher system loads, the runtime of FT-RMS is slightly higher than the persistent or the periodic threads since RMS has more context switches than FTTL (due to preemptive scheduling). Lastly, we note that the runtime costs of the FT-RMS scheme when compared to the non-fault-tolerant RMS scheme is acceptable, in particular at lower loads.

7 Related Work

In the general form of the primary-backup (PB) approach, the two processes (a primary and a backup) can execute sequentially [11], [31] or in parallel [2], [12], [26]. When dealing with transient faults, the PB requires only one resource instance and no hardware redundancy, if an error detection mechanism exists. However, PB has a larger latency than hardware redundancy and does not provide instantaneous error recovery (that is, there has to be detection and subsequent recovery). The PB scheme has been included in the well-known *recovery blocks* method [31], which was developed for general-purpose fault-tolerant systems.

The PB scheme is also applicable to fault tolerance in real-time systems, when there is enough time for detection and recovery. For example, it is the basis of the approach in [16], which assumes threads to be harmonic and two instances of each thread (primary and backup) are scheduled on a uniprocessor system. The goal of this heuristic is to maximize the number of backups scheduled, and then to accommodate the maximum number of primaries possible in the schedule. It is assumed that backups execute for less time than primaries and have less accurate results.

In addition to the research projects mentioned above, there have been some attempts for developing *fault-tolerant real-time* systems, such as SIFT [39], FTMP [35], FTP [34], and MARS [14], [12]. These systems use hardware replication and were built for specific applications, customizing designs on an ad hoc per-application basis. The overhead and lack of flexibility of these systems constitute an impediment to utilizing them in a broader range of applications. For example, SIFT, FTMP, and FTP are designed for a specific application, namely, flight control. Also, FTMP, FTP, and MARS require special hardware to perform the fault-tolerance related tasks such as voting. Lastly, to illustrate the overhead of these systems, the fault tolerance tasks in SIFT and FTMP utilize 80 and 60 percent, respectively, of the system processing power.

Further research on tolerance to transient faults have adapted the PB scheme through static and dynamic allocation strategies [32], [33]. Two algorithms are proposed to reserve time for the recovery of periodic real-time threads on a uniprocessor [33]. The RMS analysis of [20] has been extended to include provisions for thread reexecutions. One algorithm allows reserved time to be dedicated to the recovery of some individual threads while the second algorithm allows the reserved time to be shared by all threads. Sharing reserved time for reexecuting has also been explored in [7], in which better schedulability bounds are presented, based on knowledge of the task set.

In [15], processor failures are handled by maintaining contingency or backup schedules, which are invoked in the event of a processor failure. To generate the backup schedule, it is assumed that an optimal schedule exists and the schedule is enhanced with the addition of "ghost" tasks, which function primarily as backup tasks. Since not all schedules will permit such additions, the scheme is optimistic.

Balaji et al. [3] present a best effort approach to provide fault-tolerance in hard real-time distributed computing systems. A primary/backup scheme is used in which both the primary and the backup start execution simultaneously and if a fault affects the primary, the results of the backup are used. Although termed PB, this requires hardware redundancy. This technique of allowing primary and backup tasks to overlap has also been used in [37] to enhance [7].

8 Concluding Remarks

We presented schemes (optimal and greedy) for mapping real-time schedules into guaranteed fault-tolerant real-time schedules. The schemes are based on providing sufficient slack for a backup to execute, if a fault occurs. By carefully manipulating the idle slots, we can minimize the overhead of providing fault tolerance, especially in fault-free situations. We noted that, although we derived an optimal algorithm, the greedy and faster approach in this case performed within 1 percent of the optimal algorithm. These results were obtained through simulations; in the future, we intend to analyze the performance of LTH in comparison with SFS in a theoretical fashion (either deterministic or probabilistic analysis).

Our study presented here can be used to guide real-time system designers on establishing the time interval between consecutive backups (based on the MTTF) and, thus, aid in the analysis of real-time systems in environments subject to SEUs and intermittent faults. Similarly, designers can also determine the load that the system can support given the specific upper bound on rejected and lost threads and given the ratio between the cost of missing a deadline after a thread is accepted for execution and the cost of rejecting that thread.

We also extended and implemented the algorithm to include gaps, periodic threads, static and dynamic systems, and negotiated error coverage. The implementation of FT-RT-Mach required a thorough understanding of the structure of the system, so that the idea of a fault-tolerant nonpreemptive scheduler can be fitted into the preemptive environment of RT Mach. Many of the mechanisms existing in RT Mach were not directly applicable for implementation of timeline-based scheduler, but some were very useful, after relatively few modifications. This implementation, which was included in the RK97a release of RT-Mach[6] led us to identify the locations in the operating system where

6. www.cs.cmu.edu/~rtmach and www.cs.pitt.edu/FORTS.

fault tolerance capabilities need to be inserted, in order to provide for tolerance to SEUs and intermittent faults.

Our future work will focus on enhancing the error detection capabilities, adding support for precedence constraints, and extending the scheme to a distributed implementation.

ACKNOWLEDGMENTS

The authors would like to thank Anthony Egan for the implementation of the FT-RT-Mach primitives, scheduler, and admission control algorithm, and Dan Bauman who implemented and collected timing information for Section 6.4. This work was done at the Department of Computer Science, University of Pittsburgh, and it was supported in part by the US National Science Foundation under an RIA grant CCR-9308886 and by DARPA under contract DABT63-96-C-0044, through the FORTS project. A shorter version of this paper appeared in the 1995 IEEE Real-Time Systems Symposium.

REFERENCES

[1] *Mission Critical Operating Systems,* A.K. Agrawala, K. Gordon, and P. Hwang, eds., IOS Press, 1991.

[2] F. Belli and P. Jedrzejowicz, "An Approach to the Reliability Optimization of Software with Redundancy," *IEEE Trans. Software Eng.,* vol. 17, no. 3, pp. 310-312, Mar. 1991.

[3] S. Balaji, L. Jenkins, L.M. Patnaik, and P.S. Goel, "Workload Redistribution for Fault Tolerance in a Hard Real-Time Distributed Computing System," *Proc. IEEE Fault Tolerance Computing Symp. (FTCS-19),* pp. 366-373, 1989.

[4] L. Dong, R. Melhem, and D. Mossé, "Time Slot Allocation for Real-Time Messages with Negotiable Distance Constrained Requirements," *Proc. Real-Time Technology and Applications Symp.,* 1998.

[5] M. DiNatale and J. Stankovic, "Dynamic End-to-End Guarantees in Distributed Real-Time Systems," *Proc. Real-Time Systems Symp.,* pp. 216-227, Dec. 1994.

[6] J. Gaisler, "Concurrent Error-Detection and Modular Fault-Tolerance in a 32-Bit Processing Core for Embedded Space Flight Applications," *Proc. IEEE Symp. Fault Tolerant Computing (FTCS-24),* pp. 128-130, 1994.

[7] S. Ghosh, D. Mossé, and R. Melhem, "Fault-Tolerant Rate-Monotonic Scheduling," *Proc. Sixth IFIP Conf. Dependable Computing for Critical Applications,* Mar. 1997.

[8] S. Ghosh, D. Mossé, and R. Melhem, "Implementation and Analysis of a Fault-Tolerant Scheduling Algorithm," *IEEE Trans. Parallel and Distributed Systems,* vol. 8, no. 3, pp. 272-284, Mar. 1997.

[9] S. Ghosh, D. Mossé, and R. Melhem, "Tolerant Rate-Monotonic Scheduling," *J. Real-Time Systems,* vol. 15, no. 2, Sept. 1998.

[10] E.D. Jensen, C.D. Locke, and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Operating Systems," *Proc. IEEE Real-Time Systems Symp.,* pp. 112-122, Dec. 1985.

[11] B.W. Johnson, *Design and Analysis of Fault Tolerant Digital Systems.* Addison Wesley, 1989.

[12] 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,* vol. 9, no. 1, pp. 25-40, Feb. 1989.

[13] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner, and J. Reisinger, "Tolerating Transient Faults in MARS," *Proc. Symp. Fault Tolerant Computing (FTCS-20),* pp. 466-473, 1990.

[14] H. Kopetz, "Event-Triggered Versus Time-Triggered Real-Time Systems," *Lecture Notes in Computer Science,* 1991.

[15] C.M. Krishna and K. Shin, "On Scheduling Tasks with a Quick Recovery from Failure," *IEEE Trans. Computers,* vol. 35, no. 5, pp. 448-455, May 1986.

[16] A.L. Liestman and R.H. Campbell, "A Fault-Tolerant Scheduling Problem," *IEEE Trans. Software Eng.,* vol. 12, no. 11, pp. 1089-1095, Nov. 1988.

[17] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in Hard Real-Time Environment," *J. ACM,* pp. 46-61, Jan. 1973.

[18] J.W. Liu, K. Lin, C.L. Liu, and C.W. Gear, "Research on Imprecise Computations in Project QuartZ," *Proc. Workshop Operating Systems for Mission Critical Computing,* Sept. 1989.

[19] K. Lin, S. Natarajan, and J.W. Liu, "Imprecise Results: Utilizing Partial Computations in Real-Time Systems," *Proc. IEEE Real-Time Systems Symp.,* Dec. 1987.

[20] J.P. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," *Proc. IEEE Real-Time Systems Symp.,* pp. 166-171, Dec. 1989.

[21] R. Melhem and D. Mossé, "Forts: Fault Tolerance through Scheduling in Real-Time Systems," http://www.cs.pitt.edu/FORTS, 1998.

[22] D. Mossé, R. Melhem, and S. Ghosh, "Analysis of a Fault-Tolerant Multiprocessor Scheduling Algorithm," *Proc. 24th Int'l Symp. Fault-Tolerant Computing,* June 1994.

[23] D. Mossé, "Design, Development, and Deployment of Fault-Tolerant Applications for Distributed Real-Time Systems, PhD thesis, Univ of Maryland, College Park, 1993.

[24] D. Mossé, "Mechanisms for System-Level Fault Tolerance in Real-Time Systems," *Proc. Int'l Conf. Robotics, Vision, and Parallel Processing for Industrial Automation,* June 1994.

[25] D. Niehaus, "Program Representation and Tranlation for Predictable Real-Time Systems," *Proc. Real-Time Systems Symp.,* pp. 53-63, 1991.

[26] T. Ng and S. Shi, "Replicated Transactions," *Proc. Ninth Int't Conf. Distributed Computer Systems,* June 1989.

[27] T. Nakajima and H. Tokuda, "Implementation of Scheduling Policies in Real-Time Mach," *Proc. Second Int'l Workshop Object Orientation in Operating Systems,* pp. 165-169, Sept. 1992.

[28] V. Nirkhe, S. Tripathi, and A. Agrawala, "Language Support for the Maruti Real-Time System," *Proc. Real-Time Systems Symp.,* pp. 257-266, Dec. 1990.

[29] S.K. Oh and G. MacEwen, "Toward Fault-Tolerant Adaptive Real-Time Distributed Systems," External Technical Report 92-325, Dept. of Computing and Information Science, Queen's Univ., Kingston, Ontario, Canada, Jan. 1992.

[30] Y. Oh and S.H. Son, "Enhancing Fault-Tolerance in Rate-Monotonic Scheduling," *J. Real-Time Systems,* vol. 7, no. 3, pp. 315-329, Nov. 1994.

[31] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. Software Eng.,* vol. 1, no. 2, pp. 220-232, June 1975.

[32] S. Ramos-Thuel, "Enhancing Fault Tolerance of Real-Time Systems through Time Redundancy," PhD thesis, Carnegie Mellon Univ., May 1993.

[33] S. Ramos-Thuel and J.K. Strosnider, "Scheduling Fault Recovery Operations for Time-Critical Applications," *Proc. Fourth IFIP Conf. Dependable Computing for Critical Applications,* Jan. 1994.

[34] T.B. Smith, "Fault-Tolerant Processor Concepts and Operation," *Proc. 14th IEEE Fault-Tolerant Computing Symp.,* June 1984.

[35] T.B. Smith, *The Fault-Tolerant Multiprocessor Computer.* Park Ridge, N.J.: Noyes Publications, 1986.

[36] J.A. Stankovic, "Misconceptions about Real-Time Computing: A Serious Problem for Next-Generation Systems," *Computer,* vol. 21, no. 10, pp. 10-19, Oct. 1988.

[37] T. Tsuchiya, Y. Kakuda, and T. Kikuno, "A New Fault-Tolerant Scheduling Algorithm for Real-Time Multiprocessor Systems," *Real-Time Computing Systems and Applications,* pp. 197-202, 1995.

[38] H. Tokuda, T. Nakajima, and P. Rao, "Real-Time Mach: Toward a Predictable Real-Time System," *Proc. USENIX Mach Workshop,* Oct. 1990.

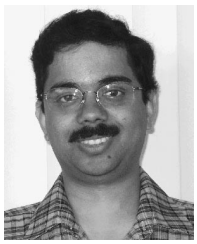
[39] J.H. Wensley et al., "SIFT: Design and Analysis of a Fault Tolerant Computer for Aircraft Control," *Proc. IEEE,* pp. 1240-1255, Oct. 1978.

[40] W. Zhao and K. Ramamritham, "Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints," *J. Systems and Software,* pp. 195-205, 1987.

[41] W. Zhao, K. Ramamritham, and J.A. Stankovic, "Preemptive Scheduling under Time and Resource Constraints," *IEEE Trans. Computers,* vol. 36, no. 8, pp. 949-960, Aug. 1987.

Daniel Mossé received the BS degree in mathematics from the University of Brasilia in 1986, and the MS and PhD degrees in computer science from the University of Maryland in 1990 and 1993, respectively. He joined the faculty of The University of Pittsburgh in 1992, where he is currently an associate professor. His research interest include fault-tolerant and real-time systems, as well as networking. The major thrust of his research in the new millenium is power-aware computing and security. He has served on program committees for all major IEEE sponsored real-time related conferences and as and program and general chairs for RTAS and RT Education Workshop. Typically funded by the US National Science Foundation and DARPA, Dr. Mosse's projects combine theoretical results and implementations. Dr. Mosse is a member of the editorial board of the *IEEE Transactions on Computers,* of the IEEE Computer Society, and the Association for Computing Machinery.

Rami Melhem received the BE degree in electrical engineering from Cairo University in 1976, the MA degree in mathematics, the MS degree in computer science from the University of Pittsburgh in 1981, and a PhD degree in computer science from the University of Pittsburgh in 1983. He was an assistant professor at Purdue University prior to joining the faculty of The University of Pittsburgh in 1986, where he is currently a professor of computer science and electrical engineering and the chair of the Computer Science Department. His research interest include real-time and fault-tolerant systems, optical interconnection networks, high-performance computing and parallel computer architectures. Dr. Melhem served on program committees of numerous conferences and workshops and was the general chair for the third International Conference on Massively Parallel Processing Using Optical Interconnections. He was on the editorial board of the *IEEE Transactions on Computers* and the *IEEE Transactions on Parallel and Distributed Systems*. He is serving on the advisory boards of the IEEE technical committees on parallel processing and on computer architecture. He is the editor for the Kluwer/Plenum book series on computer science and is on the editorial board of the *Computer Architecture Letters*. Dr. Melhem is a fellow of the IEEE and a member of the ACM.



Sunondo Ghosh received the BTech degree in computer science and engineering from the Institute of Technology, Banaras Hindu University (IT-BHU), India, in 1991, and the PhD degree in computer science from the University of Pittsburgh in 1996. He is currently working as a software architect at I2 Technologies. His current interests include distributed systems and enterprise software systems.