Enhancing Real-Time Schedules to Tolerate Transient Faults*

Sunondo Ghosh, Rami Melhem, Daniel Mossé
Department of Computer Science,
University of Pittsburgh,
Pittsburgh PA 15260
{ghosh,melhem,mosse}@cs.pitt.edu

Abstract

We present a scheme to guarantee that the execution of real-time tasks can tolerate transient and intermittent faults assuming any queue-based scheduling technique. The scheme is based on reserving sufficient slack in a schedule such that a task can be re-executed before its deadline without compromising guarantees given to other tasks. Only enough slack is reserved in the schedule to guarantee fault tolerance if at most one fault occurs within a time interval. This results in increased schedulability and a very low percentage of deadline misses even if no restriction is placed on the fault separation. We provide two algorithms to solve the problem of adding fault tolerance to a queue of real-time tasks. The first is a dynamic programming optimal solution and the second is a greedy heuristic which closely approximates the optimal.

1 Introduction

"Next-generation" real-time computing systems have to support real-time application programs by maintaining an environment that satisfies timing, reliability, and availability requirements [14]. Due to their criticality, tasks in real-time systems must always finish within the user specified deadlines. Fault tolerance techniques are based on temporal or spatial redundancy and attempt to achieve continuous execution within the deadlines in spite of hardware and software failures [1, 4, 5, 6, 9].

When a fault occurs, extra time is required to handle fault detection and recovery if fault masking is not used. For real-time systems in particular, it is essential that the extra time be accounted for prior to execution on a per task 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.

*Supported in part by NSF grant CCR-9308886 and NSF Cooperative Agreement ASC-8902826

To achieve fault masking of permanent hardware faults, redundant concurrent tasks are used to carry out the computations [3, 5]. On the other hand, to tolerate intermittent and transient faults, a primary/backup (PB) scheme can be used, in which a *primary* process executes computations and outputs results if no errors are detected. In case of an error, a *backup* process assumes the role of the primary process. Some approaches use groups of processes executing sequentially [2, 10], while others have the replicas execute in parallel [3, 7]. Due to the nature of the faults it tries to tolerate, the PB methodology offers small hardware resource requirements, but has larger latency and does not provide fault masking.

Two examples of the PB scheme are presented in [5, 10]. In the *recovery blocks* method [10], a block of commands is executed, and then an acceptance test is performed to detect faults. If there is an error, a recovery block is activated. In [5], the tasks are assumed to be periodic and two instances of each task (primary and backup) are scheduled on a uniprocessor system. The goal of this heuristic is to maximize the number of backups scheduled, after guaranteeing the maximum number of primaries in the schedule.

In this paper we concentrate on mapping a non-fault-tolerant schedule of real-time tasks to a fault-tolerant schedule. Our goal is to guarantee that a task will complete within its deadline even in the presence of transient and intermittent faults.

Problem Definition

We consider a real-time system with a queue-based scheduler, and assume capabilities to detect faults. When a fault is detected, the task is either re-executed or a backup for that task is activated as part of the fault recovery. For simplicity of presentation, we assume that the primary and backup have equal execution times. Our scheme can also be applied when the primary and backup have unequal execution times (e.g., different versions for software fault-tolerance).

In our model we consider only *transient and inter-*

mittent faults, which are short-lived malfunctions in a hardware component, affecting at most one task executing on that hardware component. Since faults need to be identified before being tolerated, fault detection is essential. Therefore, our approach will make use of the fault detection mechanisms that have been developed for various fault models [8, 11, 12].

We assume that a service executes *correctly* if it finishes within the specified deadline and delivers correct results with respect to the specification. Otherwise, we say that a service failure has occurred. We assume that errors are detected at the end of a task and that changes to the environment are committed only if no error is detected. We also assume that all input occurs at the beginning of task execution, and outputs are generated only at the end of the tasks, so that the whole task can be re-executed if it has to be aborted due to a fault. Any task with input or output in the middle of its execution can be broken into smaller tasks to satisfy this condition.

Given a set of tasks and a scheduling policy which is based either on the timing constraints of the tasks (e.g., Earliest Deadline First), or on their priorities (derived from their importance), that policy imposes a total ordering on the tasks. We assume that this total ordering of tasks is implemented in the form of a queue. The algorithms in this paper insert backups into that queue to create a fault-tolerant schedule. The backups are simply a guarantee that there will be enough time (slack) for re-execution of tasks.

We model a task by a tuple $T_i = \langle a_i, d_i, c_i \rangle$, where a_i is the task arrival time (and also its earliest start time), d_i is its deadline, and c_i is its worst case execution time. The maximum possible value of c_i for any task is denoted by c_{max}. The window of a task is defined as $d_i - a_i$ and the *window ratio* is defined as $w_i = \frac{d_i - a_i}{c_i}$. It is assumed that $w_i \geq 2$ since without this assumption it is impossible to re-execute a task after a fault and still meet its time constraints. We assume that the tasks can be independent of each other or have precedence constraints. Both models can be abstracted by the queue model used in this paper. In either case, the scheduling discipline will impose an ordering on the tasks.

In the following sections, first we describe algorithms that guarantee fault-tolerance if faults are separated by some Δ_f. In Section 2, we describe an optimal algorithm that schedules backup slots in a queue of real-time tasks, and in Section 3 we describe a greedy algorithm which approximates the optimal one. In Section 4, we discuss applications of the algorithms, and in Section 5, 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).

2 Real-Time Guarantees in the Presence of Faults

Let Q_T be a queue of n tasks to be scheduled for execution starting at the current time, t_0. In the absence of any fault, each task T_i in Q_T, will meet its deadline if $t_0 + \sum_{j=1}^{i} c_j \leq d_i$. In the presence of faults, however, some tasks may need to be re-executed and the time needed to complete the n tasks may be larger than $\sum_{j=1}^{n} c_j$. If t_i is the time at which the first i tasks in Q_T will complete execution in the presence of faults, then T_i will meet its deadline if $t_i \leq d_i$.

A simple estimate for t_i, which allows each task to be re-executed, is $t_0 + 2\sum_{j=1}^{i} c_j$. This estimate is overly conservative if faults do not occur frequently. Specifically, if any two faults are separated by an interval Δ_f, it is possible to create a schedule that accounts only for backups separated by Δ_f, that is:

$$t_i = t_0 + \sum_{j=1}^{i} c_j + \sum_{k=1}^{b} \beta_k \tag{1}$$

where $\beta_1 \ldots \beta_b$ are the lengths of some slots $B_1, \ldots, B_b$ reserved for backup execution. Specifically, if the tasks $T_1, \ldots, T_i$ are divided into subsets $\mathcal{B}_1, \ldots, \mathcal{B}_b$ such that $\mathcal{B}_1 = \{T_1, \ldots, T_{j_1}\}$, $\mathcal{B}_2 = \{T_{j_1+1}, \ldots, T_{j_2}\}$, ..., $\mathcal{B}_b = \{T_{j_{b-1}+1}, \ldots, T_i\}$, and, for $k = 1, \ldots, b$,

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

$$\beta_k \geq max\{c_u | T_u \in \mathcal{B}_k\}, \tag{3}$$

then t_i given by (1) is the maximum time needed to execute $T_1, \ldots, T_i$ in the presence of faults. Each set $\mathcal{B}_k$ specifies consecutive tasks that are assigned to a backup slot B_k for re-execution. If a fault occurs during the execution of a task in $\mathcal{B}_k$, then this task will re-execute. Condition (2) specifies that at most one task from $\mathcal{B}_k$ will need to re-execute, and condition (3) specifies that the re-execution of any task in $\mathcal{B}_k$ will not require more time than β_k, which is accounted for in the computation of t_i in (1). In other words, the following proposition is true:

Proposition 1 *Let Q_T be a queue containing n tasks at time t_0, that is, $Q_T = \{T_1, T_2, \ldots, T_n\}$. Assume that if a fault is detected during the execution of a task, the task is re-executed. If t_i, $i = 1, \ldots, n$, computed from (1), (2) and (3) satisfy $t_i \leq d_i$, and at most one fault occurs in any time interval of length Δ_f ($\Delta_f \geq 2c_{max}$), then all tasks in Q_T will meet their deadlines.*

Note that the values of t_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). In view of that, we define a *feasible schedule* to be a fault-tolerant schedule that meets all the conditions of Proposition 1 and satisfies all the task deadlines. To intuitively show how backups can be inserted into a queue conforming to Proposition 1, we consider an example: tasks in a queue are of lengths 2, 3, 3, and 1, their deadlines are 4, 10, 14 and 14.5 respectively, and the EDF scheduling policy is used. Assuming Δ_f is 10, Figure 1 shows the placement of two backups in the queue creating a feasible schedule.

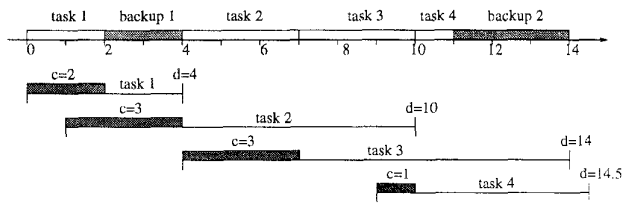

Figure 1: All tasks are accepted

Given a queue of tasks, the problem is to find the placement of backups which will lead to a feasible schedule (as in Figure 1). To solve this problem, we use a layered graph as described next.

2.1 Construction of Graph

Given a queue containing n tasks, $T_1, \ldots, T_n$, a layered graph, G, can be constructed to keep track of the possible positions of backups in the queue. Layer i has several nodes each representing a particular placement of task T_i and its backup in the queue. An edge exists between a node, N, in layer i and a node, N', in layer $i+1$ if the placement of T_{i+1} corresponding to node N' is possible given the placement of T_i corresponding to N. This means that one can view each path in G as a unique queue representing the tasks with corresponding backups. In addition to the n layers corresponding to the n tasks, we create a source node (at layer 0) and sink node (at layer $n+1$). Figure 2 shows the graph corresponding to the tasks in Figure 1.

The j^{th} node on layer i is denoted by $N_{i,j}$, and is labeled by $\langle lb_{i,j} \rangle$[1], where $lb_{i,j}$ is the length of the last backup in the queue after the placement of T_i. The first node in each layer i corresponds to placing T_i after the last backup in the queue and creating a new backup. All other nodes in the layer represent the placement of T_i before the last backup. Since there are exactly two ways of placing this task (either before or after the last backup), there are at most two edges leading out of every node in the graph: $N_{i-1,j}$ to $N_{i,0}$ or to $N_{i,j+1}$. An edge leading from node $N_{i-1,j}$ to node $N_{i,k}$ is denoted by $E_{i,k}^{i-1,j}$. We assign a weight $W_{i,k}^{i-1,j}$ to each edge $E_{i,k}^{i-1,j}$ that is equal to the increase in the length of the queue due to the addition of a new task to the queue, as will be discussed later.

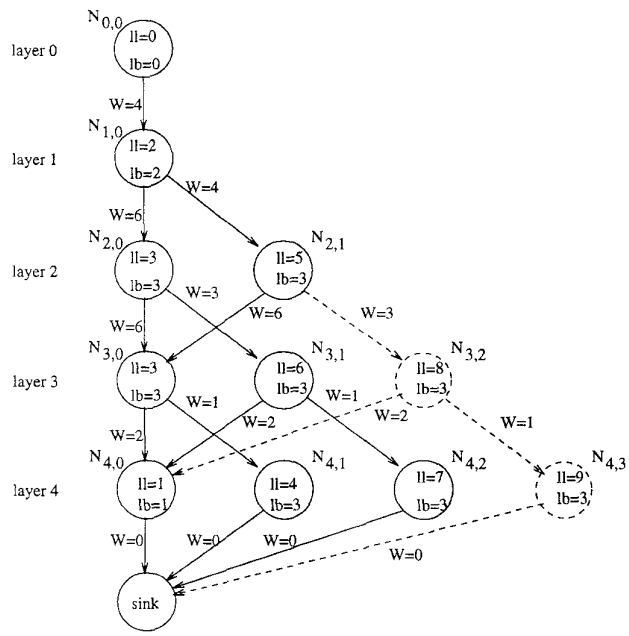

Figure 2: Construction of graph for example in Fig 1.

An example of a task being added before the backup is the edge from node $N_{1,0}$ in Figure 2 to node $N_{2,1}$. This edge ($E_{2,1}^{1,0}$) represents the case where T_2 is placed before the existing backup for T_1, and thus the backup is shared by T_1 and T_2. An example of a task being added after the backup is the transition from node $N_{1,0}$ to node $N_{2,0}$, where T_2 and its new backup are added after the existing backup for T_1.

Now we describe how the value of $lb_{i,j}$ at each node and the value of $W_{i,k}^{i-1,j}$ at each edge are computed. The length of the last backup depends on the length of the task being considered (c_i) and the length of the existing last backup in the queue. If T_i is inserted after the last backup (equivalent to an edge leading to $N_{i,0}$), a new backup of length c_i is created, and the length of the queue increases by $2 * c_i$. On the other hand, if a task is inserted before the last backup, the backup increases in length if the new task is longer than the existing backup (e.g., transition from $N_{1,0}$ to $N_{2,1}$ in Fig 2). The length of the queue increases by the sum of c_i and the increase in the length of the

[1] The meaning of *ll* in the figure is described later.

last backup, if any. The values of $lb_{i,j}$ and $W_{i,k}^{i-1,j}$ are computed using the following formulas, which satisfy (3).

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

$$W_{i,k}^{i-1,j} = \begin{cases} 2 * c_i & \text{if } k = 0 \\ c_i + max(lb_{i-1,j}, c_i) - lb_{i-1,j} & \text{if } k \neq 0 \end{cases} \quad (5)$$

After the graph is constructed, a path from the source node, $N_{0,0}$, to any node, $N_{i,j}$, corresponds to a queue with a unique placement of backups. The sum of weights along a path from the source to a node $N_{i,j}$ is equal to the span of the schedule that the path represents. The shortest path to node $N_{i,j}$ will lead to the shortest schedule.

The graph constructed so far disregarded Δ_f and took only (3) into consideration and thus some nodes in the graph may correspond to queue configurations that violate (2). For the tasks in the example of Figure 1, if $\Delta_f = 3$, only the leftmost column in each layer will satisfy (2) (one backup for each task). If $\Delta_f > 12$, all the nodes in the graph will satisfy (2). To make sure that the length of the queue between two backups does not exceed Δ_f, each node can be labeled with a second parameter $\langle ll_{i,j} \rangle$, where $ll_{i,j}$ is the length (in units of time) from the end of the second last backup to the beginning of the last backup.

The value of $ll_{i,j}$ is updated for each new task added to the queue. If the task is added after the last backup, the value of ll is equal to the computation time of that task. Otherwise, the value of ll increases by the computation time of that task. Specifically,

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

By deleting the nodes from the graph which do not satisfy the condition $ll_{i,j} + lb_{i,j} \leq \Delta_f$, we make sure that (2) is satisfied. If this condition is violated at any node, task T_i and its backup are forced to be placed at the end of the queue after the last backup that already exists. The parent of a 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 tasks (along with their backup) that can fit within a length of Δ_f in the schedule. For example, in Figure 2, nodes $N_{3,2}$ and $N_{4,3}$ are deleted if $\Delta_f = 10$. The dashed nodes and edges are thus removed from the graph.

Our goal is to find the shortest path in the graph, while making sure that $\forall i, 1 \leq i \leq n, t_i \leq d_i$ where t_i is calculated using (1). We use the steps outlined below to find the shortest path in the graph.

2.2 Feasible Shortest Path (FSP)

An *optimal* mapping from a non-fault-tolerant queue to a fault-tolerant queue is an assignment which guarantees that all tasks will meet their deadlines if no two faults occur within Δ_f, and produces the shortest possible queue. In order to find the optimal mapping, all possible placements of backups have to be considered such that the conditions of Proposition 1 are satisfied. Since both possible placements of tasks (before or after the last backup) are represented in the graph, exploring the paths in the graph will lead to an optimal assignment. 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 tasks and backups can be maintained at each node to find out the actual placement if needed.

To obtain the optimal assignment of backups in the queue, we associate with each node a value $V_{i,j}$, which is the minimum length of the queue up to node $N_{i,j}$ (initially, $V_{0,0} = 0$). The value of $V_{i,j}$ ($j \neq 0$) is obtained by adding the value at node $N_{i-1,j-1}$ to the weight $W_{i,j}^{i-1,j-1}$ (since this is the only incoming edge to node $N_{i,j}$). The value of $V_{i,0}$ is obtained by adding the minimum value among all nodes at layer $i-1$ to the weight $W_{i,0}^{i-1,k}$. Note that, from (5), the weights are equal on all edges incident on node $N_{i,0}$, that is, $W_{i,0}^{i-1,k} = W_{i,0}^{i-1,m}$, $0 \leq k, m \leq i-1$. Thus, the value of $V_{i,j}$ can be calculated as follows:

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

In real-time systems, if $V_{i,j} > d_i$, the placement of T_i and its backup corresponding to $N_{i,j}$ cannot lead to a feasible schedule. In that case, we assign $V_{i,j} = \infty$, 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 $V_{i,j}$ at level i should be supplemented by:

$$V_{i,j} = \infty \quad \text{if } V_{i,j} > d_i, \ j = 0, \ldots, i \quad (8)$$

Finally, when we reach the leaf nodes at layer n (after considering all n tasks in the queue), all nodes with $V_{i,j} \neq \infty$ represent feasible assignments of backups which satisfy $t_i \leq d_i, i = 1, \ldots, n$ where t_i is computed from (1), (2), and (3). The sink node at level

$n+1$ is assigned the minimum of all values at level n. Thus, even if a single feasible assignment exists, the algorithm returns FT (FAULT TOLERANCE) GUARANTEED. If the positions of backups in the queue are required, we can follow the edges on the shortest path.

Complexity and Optimality

Although in our presentation we separated the construction of G and the computation of the shortest path, it is possible to simultaneously compute the values of $lb_{i,j}$, $W_{i,k}^{i-1,j}$, $ll_{i,j}$ and $V_{i,j}$ for each node from (4), (5), (6), and (7).

The complexity of this computation is $O(n^2)$ in the worst case. However, in the average case, the complexity of the algorithm is lower, since the number of nodes at any layer is limited 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 tasks 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 tasks in the queue. Thus the average runtime is $O(\frac{\Delta_f}{c_{av}} n)$.

By finding the feasible shortest path in G which satisfies $V_{i,j} \leq d_i$ at each node $N_{i,j}$, FSP is optimal in the following sense: if an assignment of backups exists for a given queue with n tasks such that $t_i \leq d_i$ where t_i is computed from (1), (2) and (3), this algorithm will find the assignment. If multiple assignments of backups exist, the algorithm finds the assignment that minimizes the length of the queue.

In a static environment, when all task arrival times are known beforehand, the $O(n^2)$ complexity of the algorithm is acceptable. However, in a dynamic environment, a task should be scheduled as soon as it arrives. This involves inserting the new task into the existing queue of tasks, and guaranteeing that the new task as well as all previously scheduled tasks will meet their deadlines even in the presence of faults. A lower complexity algorithm should be used in this situation. Such an algorithm should be able to find a path from the source to the sink of the graph without actually building the entire graph. In the next section, we provide a linear time algorithm to insert backups into a queue of tasks.

3 Linear Time Heuristic (LTH)

If Q_T contains tasks $T_1, \ldots, T_n$, the following algorithm can be used to check if the queue is schedulable without violating the deadlines of the n tasks even in the presence of faults. In this algorithm, the variable δ is used to keep track of the length of the queue between the last two backups, β is the length of the last backup, and t_i is the length of the queue when i tasks have been considered.

Algorithm LTH (Linear Time Heuristic):
$\delta = 0$; $\beta = 0$; $t_0 = 0$;
For $i = 1, \ldots, n+1$ do
  If $\delta + c_i + max\{\beta, c_i\} \leq \Delta_f$
    Then $\delta = \delta + c_i$; $t_i = t_{i-1} + c_i$; $\beta max\{\beta, c_i\}$;
    Else $t_i = t_{i-1} + c_i + \beta$; $\delta = c_i$; $\beta = c_i$;
  If $t_i + \beta > d_i$, return (FT NOT GUARANTEED);
return (FT GUARANTEED)

The following proposition follows directly from Proposition 1 and the observation that t_i computed by LTH satisfies (1), (2) and (3).

Proposition 2 *If LTH returns* FT GUARANTEED *and at most one fault can occur in time* $\Delta_f \geq 2c_{max}$, *then the n tasks in* Q_T *are guaranteed to complete execution before their deadlines.*

The worst case analysis shows that LTH is linear with the number of tasks in Q_T. In the context of the layered graph described in Section 2.1, the placement of backups chosen by LTH is equivalent to the path $N_{0,0}$, $N_{1,0}$, $N_{2,1}$, $N_{3,2}$, ... which reaches a node $N_{j+1,j}$ that has only a single outgoing edge (to node $N_{j+2,0}$). The path has to include this edge. From node $N_{j+2,0}$, the same procedure of including edges to $N_{j+3,1}$ and so on is continued. Finally the last row is reached, and the path ends at the sink.

LTH is greedy (thus not optimal) because it tries to provide a single backup for as many tasks as possible. This may create a schedule which is longer than necessary, and thus may lead to an infeasible schedule. For example, if EDF scheduling policy is applied to Figure 1 and LTH is used to place the backups, then the queue shown in Figure 3 is created. We find that when T_4 is added to the queue, $t_4 > d_4$, and thus LTH returns FT NOT GUARANTEED. This is correct, because if a fault occurs, say at time 10.7, T_3 is re-executed, and T_4 misses its deadline. In this case, LTH fails to find a backup placement when FSP is successful by placing the backup after T_1 (see Figure 1).

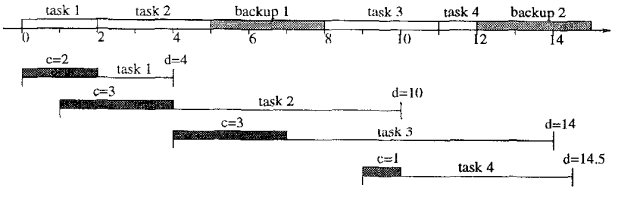

Figure 3: Fault Tolerance is not guaranteed by LTH

4 Applications

The two algorithms described above (FSP and LTH) can be used in several different scenarios in real-

time systems: whenever the system is prone to transient or intermittent faults, and a non-fault-tolerant queue has to be transformed into a fault-tolerant queue.

4.1 Static and Dynamic Systems

In a static system, all tasks are analyzed and scheduled when the system is being built. If all the task arrival times are known beforehand, the optimal FSP algorithm can be used to ensure that they will meet their deadlines even in the presence of faults. If the tasks in the system are hard real-time, and can cause a catastrophe if not completed in time, they *should* be analyzed while building the system, and sufficient slack for fault tolerance should be provided at that time. If all critical tasks cannot be scheduled with fault tolerance, then more computation power should be added (as suggested in [9]), or a few less important tasks can be removed from the set by the system designers. Once the tasks are accepted, they must finish before their deadlines even in the presence of faults.

Even if tasks arrive dynamically, it is possible to statically determine that they will meet their deadlines in the presence of faults. To do so, we can consider their worst case arrival pattern, which will occur if several or all of the dynamic tasks, triggered by various events, arrive simultaneously. In such a situation, the tasks can be queued in a predetermined order. This queue will be provided with backups using FSP.

However, if the tasks need to be scheduled and guaranteed dynamically, (e.g., "essential" tasks in the Spring system [13]), then LTH can be used due to its lower complexity. In this case, the newly arrived task has to be inserted in the queue of existing tasks based on any scheduling policy determined by the user. Once the queue is ready, LTH can be run on the queue to insert backups. If the backups cannot be inserted at the required separation, an alternative action has to be taken. Either the backup separation can be increased until the queue is feasible, or the user informed about the infeasiblity of the queue. If the schedule is infeasible, the user has the option of continuing without fault tolerance, aborting the task (if it is within the user's control), or taking over manual control (e.g., switching from autopilot to manual control of an aircraft). It is important to remember that once the newly arrived task is accepted, all tasks previously accepted and the new task are guaranteed to finish before their deadlines even in the presence of faults.

4.2 Negotiating Fault Tolerance

The probability of tolerating a fault is inversely proportional to the backup separation (Δ_f as defined in Proposition 1) and gives an idea of whether or not the guarantees provided to the user will be valid during task 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 task missing its deadline. On the other hand, if the backup separation is small, then frequent faults can be tolerated. In general, for critical tasks the backup separation is required to be low, while for less critical tasks, 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 task set outright, the user can be allowed to negotiate the value of backup separation. To make the choice easier for the user, FSP can provide the smallest value of backup separation which will generate a feasible schedule for all tasks in the system. This can be done by doing a depth first search on the graph created in Section 2.1. The exact procedure is not described here due to lack of space.

5 Simulation and Analysis of Results

In this section, we present the results of the simulations that were conducted to evaluate FSP and LTH. First, we present a comparison of the two algorithms. FSP is an optimal algorithm, but its complexity is higher. LTH is a linear time algorithm, but is not optimal. We show how the two compare in terms of schedulability. We thereafter analyze LTH's performance exclusively, since their behavior is similar, and the complexity of LTH is lower. We study three scheduling policies in combination with LTH, and then select the one with the best results for further studies.

5.1 Comparison of FSP and LTH

Given a queue of tasks, we want to determine what is the loss in performance of LTH in comparison to FSP. We determine the number of times that FSP is able to find a feasible schedule for a queue of tasks when LTH fails to do so. In our simulations to compare the two algorithms, we considered queues containing sets of different number of tasks. We generated 1000 tasks sets for each combination of parameters (such as the load[2], number of tasks in the queue, window ratio w). We found that in the worst case, LTH rejects up to 0.7% more tasks sets than FSP.

We also found that the difference between the two algorithms depends on the load. If the system is lightly loaded or heavily loaded, then the two algorithms perform almost identically. However, for

[2] The load is approximated by the sum of computation times of the tasks divided by the time span from the earliest task arrival time to the last deadline. This concept will be explained further in Section 5.3.

load	20 tasks			50 tasks		
	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

Table 1: Percentual Difference in number of task sets guaranteed by FSP and LTH.

medium loads (around 0.5 or 0.6), the difference goes up. We present in Table 1 the difference in the percentages of task sets guaranteed by FSP and LTH, for varying window ratios, and for 20 and 50 tasks in the queue. We conclude from the results obtained that LTH approximates FSP very well in finding feasible schedules for queues of real-time tasks.

5.2 Evaluation of LTH

In dynamic systems, if a task cannot be guaranteed fault tolerance by the system when it arrives, it is *rejected* (and the user can abort the task, continue without fault tolerance, or switch to manual control). A task which cannot meet its deadline because the faults occurred more frequently than expected (violating the fault assumptions) is called a *lost task*. We use a parameter Ω to represent the ratio of the cost of losing a task after accepting it and the cost of rejecting it (not accepting it when submitted).

As expected, the simulation results show that the fault tolerance capability decreases the number of lost tasks at the cost of increasing the number of rejected tasks. The first goal of our simulation is to estimate this tradeoff for different parameters. In addition to the number of rejected tasks, we also look at the success of the algorithm from the perspective of lost tasks in comparison to the algorithm which does not provide fault-tolerance. We refer to the method in which no fault tolerance is provided as the *No-FT* method.

Another goal of the simulation is to determine the load at which the number of tasks rejected and lost are below specified percentages. The system designer can analyze the characteristics of the tasks that may arrive dynamically to determine the average window ratio and computation times of those tasks. Once the scheduling policy is determined, then the maximum allowed load for a specified schedulability and a specified rate of lost tasks can be determined for a given Mean Time To Failure (MTTF).

5.3 Simulation Parameters

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

- **scheduling discipline**: Many scheduling disciplines can be used including Earliest Deadline First (EDF), Least Laxity First (LLF), First In First Out (FIFO).
- **average computation time, c_{av}**: The task computation time is assumed to be uniformly distributed with mean c_{av}. [=5]
- **load γ**: The probability of task arrival is Poisson distributed with rate $\lambda_T = \frac{\gamma}{c_{av}}$. [=0.5,0.6,...,1.1]
- **maximum window ratio w_{max}**: The window ratio is uniformly distributed between 2 and w_{max}. [=5,10,15]
- **fault rate λ_f**: The probability of fault arrival is Poisson distributed with rate λ_f. [$=\frac{1}{250}, \frac{1}{500}, \cdots, \frac{1}{2000}$]
- **backup separation** L: Δ_f defined earlier is taken to be equal to L. [$=2c_{max}, \ldots,$ 1.5MTTF]

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.

In the simulation, whether fault tolerance is taken into consideration when a task is accepted into the system (LTH) or not (No-FT), the task in which an error is detected is re-executed. Note that if LTH is used to accept tasks into the system, a task can be lost due to deadline miss only if more than one fault occurs within an interval of length L (violating the fault model).

5.4 Analysis of results

We start by analyzing the behavior of LTH in relation to the load. Our scheme can be used with any scheduling policy, and we determine one which produces the best results using this study. Figure 4 shows the percentage of tasks rejected and lost for three scheduling policies, EDF, LLF, and FIFO. The FIFO policy causes more tasks to be rejected as compared to the other two, and also causes more tasks to be lost for lower loads. However, for higher loads the EDF and LLF policies cause more tasks to be lost. This is because the FIFO scheme rejects more tasks and hence fewer tasks execute in the system. Thus a lower number of tasks are lost when faults occur.

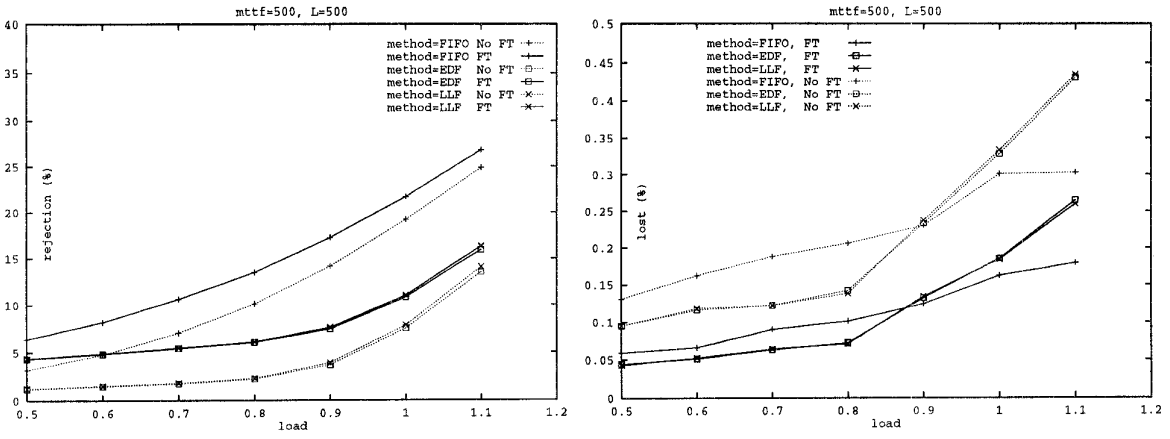

Figure 4: Percentage of lost and rejected tasks for varying load

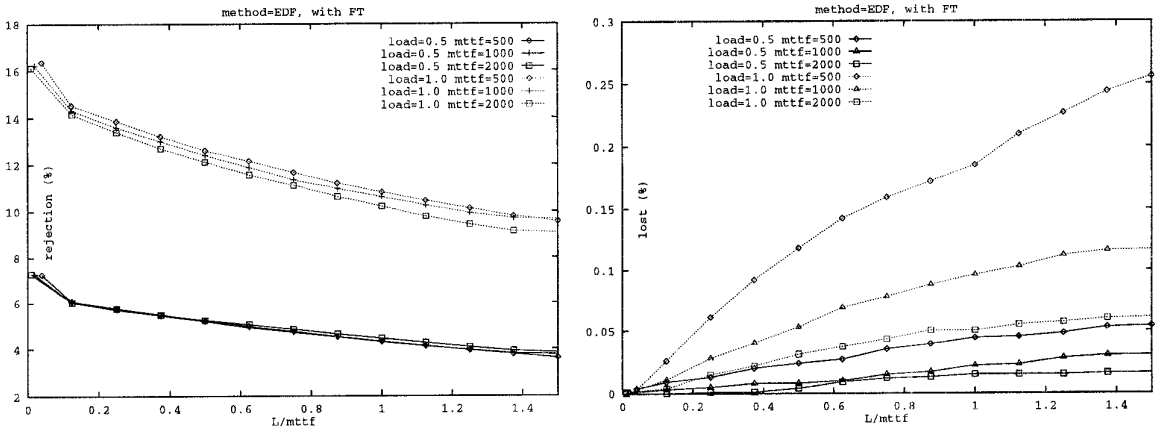

Figure 5: Percentage of lost and rejected tasks for values of L as a fraction of MTTF

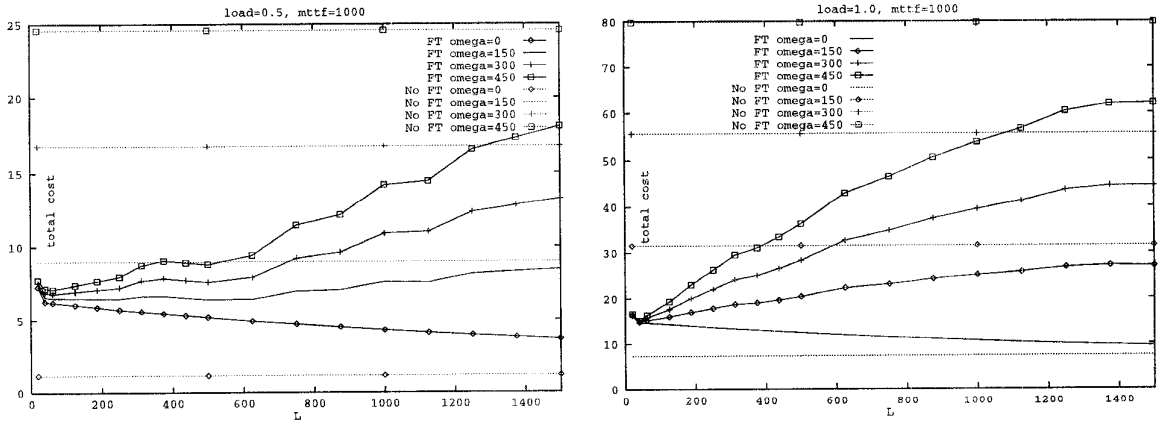

Figure 6: Total cost for varying values of L and Ω

Even though the percentage of tasks rejected by LTH is higher than No-FT for all three scheduling policies, the percentage of tasks lost is lower. Also note that the value of L is chosen to be equal to MTTF in Figure 4. If the number of lost tasks is required to be lower, then the value of L has to be smaller. The number of lost tasks will approach 0 as L approaches the average length of a task (which would mean a backup for every task). Note that the choice of L=MTTF is not the one that minimizes the rejection and cost, as we will see later.

From Figure 5 onwards, 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. Figure 5 shows the percentage of tasks rejected and lost for varying values of L as a factor of MTTF. The percentage of tasks rejected decreases as the value of L increases compared to the MTTF. This is because a larger value of L causes fewer backups to be placed in the queue, and thus more tasks can be accepted. On the other hand, the number of tasks lost increases with L, with the slope being quite steep for higher loads.

It is interesting to note the difference in rejection rate for the various loads, and that the percentage of rejected tasks has little variation for varying values of MTTF, for each set of parameters (e.g., for load = 1.0 or 0.5). This is because tasks are rejected due to their timing constraints and not due to the frequency of faults. The two cases shown in Figure 5 show that the variation for load = 1.0 is higher than for load = 0.5. For high loads (such as 1.0), when the MTTF is smaller, more backups are reserved and therefore leave less time for the primaries to execute (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 tasks. As for the number of lost tasks, it decreases with increased values of MTTF. Also a higher value of load causes more tasks to be lost for the same ratio of $\frac{L}{MTTF}$. Hence the number of lost tasks 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 each of the sets of graphs presented above, the system designer has to deal with two parameters, rejected tasks and lost tasks, to decide on the value of *load* and L. To simplify the analysis, we combine the two parameters so that they can be studied together instead of independently. Specifically, it is clear that the tradeoff between schedulability and task loss depends on the importance of each task (the cost of missing a deadline). As mentioned earlier, the parameter Ω is the ratio of the costs of losing a task and of rejecting it. So we plot all future graphs according to to following cost function: $totalcost = rejected + \Omega \times lost$.

If $\Omega = 0$, then the graph simply shows the percentage of rejected tasks. However, if $\Omega > 0$ then there is a cost for accepting a task and then missing its deadline, and the cost increases with Ω. If the lost tasks can cause a catastrophe, the value of Ω is very large, which means that it is acceptable to reject a task, but it is very costly to lose an accepted task. We have observed that whenever the value of Ω is small, it is preferable not to provide fault tolerance at all, since the lost tasks are not costly, and the number of rejected tasks is smaller. However, for larger values of Ω, it is essential to provide fault tolerance.

In Figure 6, we show the total cost for different values of Ω. When $\Omega = 0$, the total cost (equal to the rejection) decreases as L increases. For this value of Ω, No-FT performs better that LTH. As L becomes large, the total cost (rejection) approaches the cost of No-FT. When the value of Ω increases to a certain threshold (slightly less than Ω=150 in the figure), the total cost becomes almost independent of L. When Ω increases beyond this threshold, the total cost increases monotonically with L (except for the initial drop), and LTH performs better than No-FT. We can also see that when the load is low (0.5), the increase in total cost with L is slower than for a higher load (1.0). Since less tasks are scheduled, less tasks are lost.

An interesting observation is that for large values of Ω, the total cost is minimum not for the smallest possible value of L (which is $L = 2 * c_{max}$), but for a slightly larger value of L. This is because when L is very small, the number of lost tasks is almost 0, but the number of rejected tasks increases sharply.

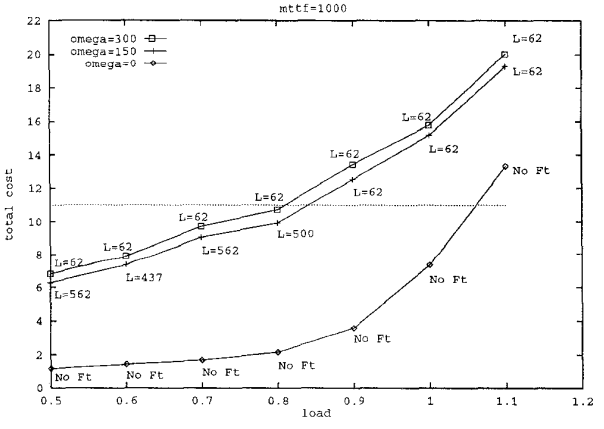

Figure 7: Total cost with optimal L vs. load

In Figure 7, we plot the 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[3] is specified beside each point in the graph. For small values of Ω, the minimum cost is reached when L=∞. 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 for no fault-tolerance. 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).

This graph can be used to determine the load that can be supported by the system given the percentage of rejection and lost tasks that the system designer can tolerate, and the value of Ω. The value of Ω can be determined by the system designer to be the number of tasks that may be rejected in order to prevent the loss of one task (by providing guaranteed fault tolerance to that task thus preventing it from being lost after being accepted).

For example, consider a system that can tolerate a rejection of below 5%, can lose up to 0.02% of the tasks guaranteed for fault tolerance, and the value of Ω is specified as 300. In this case, the total cost is equal to 11 ($= 5 + 300 \cdot 0.02$), and using Figure 7 we see that a load of less than 0.8 can meet these specifications.

6 Concluding Remarks

We presented a scheme for providing placement of backups and real-time scheduling guarantees in the presence of transient and intermittent faults. The scheme is based on providing sufficient slack for each task to re-execute (or backup to be activated) if a fault occurs. By carefully manipulating the idle slots, we can ensure that these slots are reclaimed whenever they are no longer needed. This minimizes the overhead of providing fault tolerance, especially when no faults occur.

We analyzed the effect of varying the interval between two consecutive backups, based on the expected transient/intermittent fault inter-arrival time. This study can be used to guide real-time system designers on establishing the time interval L between consecutive backups (based on Δ_f), and thus aid in the analysis of real-time task sets in environments subject to transient and intermittent faults. Similarly, designers can also determine the load that the system can support given the specific upper bound on rejected and lost tasks, and given the ratio between the cost of missing a deadline after a task is accepted for execution, and the cost of rejecting that task.

If faults are separated by Δ_f, our scheme can guarantee that no accepted tasks will be lost (miss its deadline). If this fault separation assumption is not valid, then we have shown, using simulation, that task loss is minimal. This fault tolerance capability comes at a price of a decrease in schedulability.

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

References

[1] S. Ghosh, R. Melhem, and D. Mossé. Fault-Tolerant Scheduling on a Hard Real-Time Multiprocessor System. In *Int'l Parallel Processing Symp*, April 1994.

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

[3] H. Kopetz et al. Distributed Fault-Tolerant Real-Time Systems: The MARS Approach. *IEEE Micro*, 9(1):25–40, Feb. 1989.

[4] C. M. Krishna and K. G. Shin. On Scheduling Tasks with a Quick Recovery from Failure. *IEEE Trans on Computers*, 35(5):448–455, May 1986.

[5] A. L. Liestman and R. H. Campbell. A Fault-tolerant Scheduling Problem. *Trans Software Engineering*, SE-12(11):1089–1095, Nov 1988.

[6] D. Mossé, R. Melhem, and S. Ghosh. Analysis of a Fault-Tolerant Multiprocessor Scheduling Algorithm. In 24^{th} *Fault-Tolerant Computing Symp*, June 1994.

[7] T. Ng, S. Shi. Replicated Transactions. *9th Int'l Conf on Dist Comput Syst*, Chicago, IL, June 1989.

[8] S. K. Oh and G. MacEwen. Toward Fault-tolerant Adaptive Real-Time Distributed Systems. CIS-TR 92-325, Queen's University, Canada, Jan 1992.

[9] Y. Oh. *The Design and Analysis of Scheduling Algorithms for Real-Time and Fault-Tolerant Computer Systems.* Ph.D. thesis, Univ of Virginia, 1994.

[10] B. Randell. System Structure for Software Fault Tolerance. *IEEE Trans. on Software Engineering*, SE-1(2):220–232, June 1975.

[11] T. B. Smith. *The Fault-Tolerant Multiprocessor Computer.* Noyes Publications, Park Ridge, NJ, 1986.

[12] T. K. Srikanth. *Designing Fault Tolerant Algorithms for Distributed Systems Using Communication Primitives.* PhD thesis, Cornell University, Feb 1986.

[13] J. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-Time Systems. *IEEE Software*, 8(3):62–72, May 1991.

[14] J. A. Stankovic. Misconceptions about Real-Time Computing: A Serious Problem for Next-Generation Systems. *IEEE Computer*, 21(10):10–19, Oct. 1988.