An Improved Feasible Shortest Path Real-Time Fault-Tolerant Scheduling Algorithm

Hyungil Kim, Sungyoung Lee and Byeong-Soo Jeong
School of Electronics and Information, Kyung Hee University
Kyungki-Do, Youngin-City, Kiheung-Eup, Seocheon-Ri 1
E-mail : { hikim, sylee }@oslab.kyunghee.ac.kr

Abstract

Fault tolerance is an important aspect of real-time computer systems, since timing constraints must not be violated. For real-time single processor environment, Ghosh proposed two queue-based scheduling techniques: an FSP (Feasible Shortest Path) algorithm and LTH(Linear Time Heuristics). Even though the FSP algorithm can produce optimal fault-tolerant schedules, it is not practical due to its time complexity. The LTH algorithm is a greedy heuristics that closely approximates the optimal. However, since Ghosh's algorithm assumes that there is at most one fault within time interval Δf and does not consider inter-fault time it can deteriorate real-time scheduling performance due to unnecessary backup scheduling. In this paper, we have proposed an improved FSP algorithm on the more realistic assumption that there is no additional fault during minimum inter-fault time ΔF after one fault occurs. The proposed algorithm can improve system performance by including more primary tasks in a fault-tolerant schedule and also reduce time complexity in generating backup schedules.

1. Introduction

In hard real-time applications, a guarantee is required that tasks will meet their deadlines despite the presence of faults, because it causes critical situations if tasks cannot finish within the user specified deadlines. Avoiding faults is not always possible, since the system designer does not know when a fault will occur or what the faults will be. Thus, fault-tolerant real-time scheduling has been widely studied for a long time. Such studies have been performed in two different environments, viz., single processor environments [1, 2] and multiprocessor environments [3, 4, 5, 6, 7]. Generally, fault-tolerant scheduling cannot solve the permanent faults occurring in a single processor system. Thus, most research in this environment tries to solve the problems with the assumption that a fault is intermittent and transient in a task or a processor. The basic approach for fault-tolerant scheduling is to efficiently provide backup period where the faulted task is re-executed when a fault occurs in a task. Such an approach also assumes that a fault does not occur continuously during the re-execution of the faulted task.

Ghosh and Mossé [1] suggest a queue-based fault-tolerant scheduling scheme that overcomes larger latency and fault masking problems in the previous Primary/Backup scheme [2]. Their 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. Ghosh provides 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 called FSP (Feasible Shortest Path). The FSP algorithm can generate a fault-tolerant schedule that satisfies deadlines of all tasks if at most one fault occurs within a given time interval, Δf . An optimal fault-tolerant schedule is made by mapping a non-fault-tolerant schedule of real-time tasks to a fault-tolerant schedule by way of queue mapping. The second Ghosh algorithm is LTH (Linear Time Heuristics). It guarantees that all tasks in a queue can be completed within their deadlines if at most one fault occurs within time interval $\Delta f \geq C_{max}$ (C_{max} is the maximum execution time of tasks). The goal of this heuristics is to maximize the number of backups scheduled, after guaranteeing the maximum number of primaries in the schedule.

However, in Ghosh's algorithms, since the assumption that there is at most one fault within time interval Δf does not reflect real fault occurring situations, it can degrade scheduling efficiency due to unnecessary

This work was partly supported by the Ministry of Information and Communication of Korea (AB-97-G-0655)

backup scheduling. In this paper, in order to remove such inefficiencies we propose an improved FSP real-time fault-tolerant scheduling algorithm while assuming more realistic fault occurring conditions in which a new fault does not occur within time interval ΔF (inter-fault time) after the fault occurs in a task. Such a realistic assumption makes it possible to schedule more primary tasks within same time interval because it does not need to provide a backup schedule during time interval ΔF after a fault occurs. Figure 1 shows that inter-fault time can be much smaller than time interval Δf according to Ghosh's assumption. On the Ghosh assumption, two faults can happen close to each other as in Figure 1. In that case, for the backup schedule of fault 1 and 2, primary real-time tasks cannot be scheduled during the next time interval Δf. This causes real-time performance degradation. By contrast, Figure 2 shows an example of our assumption that the next fault (fault 2) occurs only after inter-fault time ΔF since fault 1 occurred. This is very similar to the minimum inter-arrival time of sporadic tasks that have the characteristics of hard and non-periodic real-time tasks. Thus, our real-time fault-tolerant scheduling algorithm can be efficiently applied to sporadic task scheduling.

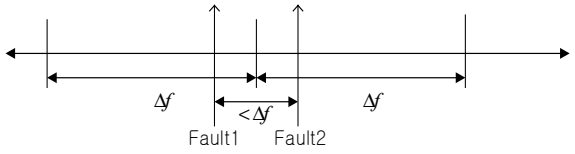

Figure 1. Example of Ghosh's fault occurring assumption

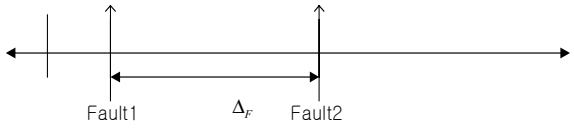

Figure 2. Example of minimum inter-fault time ΔF

Our proposed algorithm goes through three phases to produce a final backup schedule. In the first phase, it generates an optimal backup schedule for at least one task set (H_i). In the second phase, corresponding backup schedules are generated while assuming that the fault occurs in a task other than the first task in H_i. Among these backup schedules, we select the longest one and remove the tasks that could not meet their deadlines in that schedule. After that, the procedure is repeated again from the first phase. If all tasks in a schedule can meet their deadlines, the backup schedule for a task set (H_i) is determined. In the third phase, the above steps are applied to remaining tasks excluding task set (H_i) and finally the global backup schedule is determined. In contrast with Ghosh's assumption that there can be only one fault during the time interval Δf without regard to the time of the fault, we consider the more realistic assumption that there is no additional fault during the minimum inter-fault time ΔF after one fault occurs (This assumption is more meaningful in view of the fault-tolerant recovery of sporadic real-time tasks). The proposed algorithm not only improves system performance by including more primary tasks in fault-tolerant schedules but also reduces scheduling cost so that it may be well applied to dynamic environments.

The remainder of the paper is organized as follows. In Section 2 we briefly describe Ghosh's algorithms and mention their drawbacks. In Section 3 we propose an improved FSP algorithm that can overcome the problems of Ghosh's algorithm. Simulation results are represented in Section 4. In Section 5 we discuss future work and provide some concluding remarks.

2. Overview of FSP and LTH Algorithms

In our fault-tolerant real-time system model, we consider only transient and intermittent faults which are short-lived malfunctions in a hardware component, affecting at most one task executing on that hardware component. We assume that all input occurs at the beginning of task execution, and outputs are generated only at the end of 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 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 of tasks. We assume that this total ordering of tasks is implemented in the form of a queue. FSP and LTH algorithms are devised to efficiently insert backups into that queue so that a fault-tolerant schedule is possible.

The FSP algorithm is based on an analytical model that will now be described. In this model, a task is represented by a tuple $T_i = <a_i, d_i, c_i>$, where a_i is task arrival time (and 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}. 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 $t_0 + \Sigma C_j < d_i$. 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 ΣC_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 $t_i < d_i$.

If any two faults are separated by an interval Δf, the execution time t_i of task T_i in a task queue Q_t can be described by formula (1):

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

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

$T_1,...,T_i$ are divided into subsets $ß_1,...,ß_b$ such that $ß_1 = \{T_1,...,T_{j1}\}$, $ß_2 = \{T_{j1+1},...,T_{j2}\},\ldots, ß_b = \{T_{jb-1},...,T_i\}$, and, for $k = 1,...,b$,

$$ß_k + \sum_{Tu \in Bk} c_u \leq \Delta f \qquad (2)$$

$$ß_k \geq \max \{c_u \mid T_u \in B_k\} \qquad (3)$$

then t_1 given by (1) is the maximum time needed to execute $T_1,...,T_i$ in the presence of faults. Each set $ß_k$ specifies consecutive tasks that are assigned to a backup slot β_k for re-execution. If a fault occurs during the execution of a task in $ß_k$, then this task will re-execute. Condition (2) specifies that at most one task from $ß_k$ will need to re-execute, and condition (3) specifies that the re-execution of any task in $ß_k$ will not require more time than β_k, which is accounted for in the computation of t_1 in (1). The above description can be summarized by the following proposition, and the FSP algorithm is based on this proposition.

[**Proposition 1**] *Let Q_t be a queue containing n tasks at time t_0, that is, $Q_t = \{T_1,T_2,...,T_n\}$. Assume that if a fault is detected during the execution of a task, the task is re-executed. If t_i i = 1,...,n, computed from (1), (2) and (3) satisfy $t_i < d_i$, and at most one fault occurs in any time interval of length Δf ($\Delta f >= 2C_{max}$), then all tasks in Q_t will meet their deadlines.*

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 illustrate how backups can be inserted into a queue conforming to **Proposition 1**, let's consider an example: tasks in a queue are lengths 2, 3, 3, and 1, their deadlines are 4, 10, 14, and 14.5 respectively, and the EDF scheduling policy is used. If Δf is 10, the tasks can be divided into subsets, $ß_1 = \{T_1\}$ and $ß_2 = \{T_2, T_3, T_4\}$, and backup slot β_i and β_2 will be 2 and 3 respectively. By using formula (1), the execution time t_2 of task T_2 is calculated in formula (4), and we know that t_2 can meet its deadline d_2.

$$t_2 = t_0 + \sum_{j=1}^{2} c_j + \sum_{j=1}^{2} \beta_k = 0 + c_1 + c_2 + \beta_i + \beta_2$$
$$= 2 + 3 + 2 + 3 = 10 \leq d_2 \qquad (4)$$

The execution time of other tasks can be calculated by the same formula and all tasks can meet their deadlines. Figure 3 shows how a fault-tolerant schedule can be obtained in each case of task fault.

As can be seen in Figure 3, every task can complete within time 12, no matter which task a fault occurs in. At this point, if we assume that deadline of task T_4 is 12, the execution sequence of tasks will be T_1, T_2, T_3, T_4 since the arrival time of T_4 is the latest among the tasks even though the deadline of T_4 is earlier than T_3. Here, according to formula (1), the execution time of T_4 can be calculated as follows:

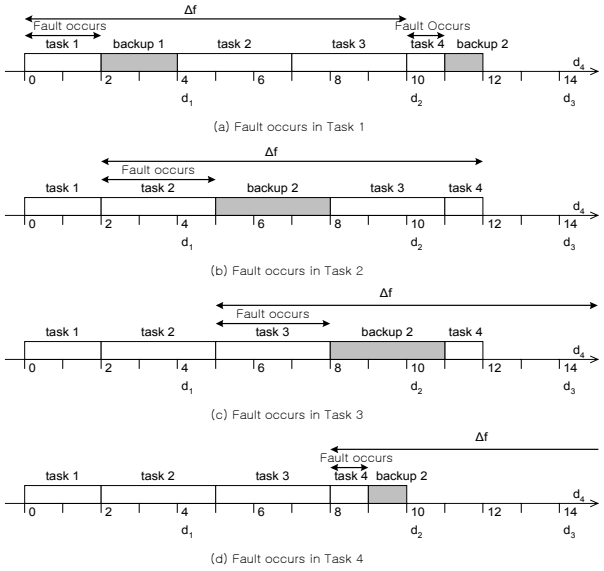

Figure 3. Example of Fault-Tolerant Schedule

$$t_4 = t_0 + \sum_{j=1}^{2} c_j + \sum_{k=1}^{2} \beta_k = 0 + c_1 + c_2 + c_3 + c_4 + \beta_i + \beta_2$$
$$= 2 + 3 + 3 + 1 + 2 + 3 = 14 > d_4 \qquad (5)$$

Therefore, even though a feasible (fault-tolerant) schedule (as in Figure 3) can be possible in the case where T_4's deadline is 12, the FSP algorithm does not produce a feasible schedule since the conditions in **Proposition 1** are not satisfied as can be seen in formula (5). The reason is that the FSP algorithm does not consider the realistic situations that a fault does not occur within time interval Δf after a first fault occurs. In order to overcome such problems, we propose an improved FSP algorithm while considering a more realistic assumption about fault occurrence.

3. Improved FSP algorithm

The basic approach of our proposed algorithm can be described as follows. The whole procedure of the algorithm consists of three phase. In the first phases, with the assumption that a fault occurs in any one task, a backup schedule is determined. At this time, a backup schedule of other tasks that can be completed within inter-fault time ΔF does not need to be considered because we assume that there are no additional faults during the minimum inter-fault time ΔF after one fault occurs. By doing this, we can get an optimal backup schedule for at least one task set. In the second phase, with the assumption that a fault occurs in the next task to that assumed in the first phase, we generate

a backup schedule. In this case, we can also apply the same assumption. That is, since there is no additional fault within the minimum inter-fault time ΔF, we don't need to consider any backup schedule for tasks that arrive within ΔF time before a fault occurs in a task. In the second phase, by selecting the longest backup schedule among them, we can get an optimal fault-tolerant schedule for one task set that can be completed within inter-fault time ΔF. In the third phase, we apply the above procedures to the remaining tasks while dividing tasks into subsets that have the same time interval ΔF. Finally, we can get a global fault-tolerant real-time schedule that can reduce rejected tasks as much as possible.

We now turn to describing our algorithm in more detail with a concrete system assumption and explanation about the notation used in the algorithm. First, we assume a system environment as follows:

A_1) The system is a hard real-time fault-tolerant system with a single processor.

A_2) Tasks are non-periodic and scheduled in non-preemptive manner when they arrive.

A_3) Every task is independent of every other. In the case where a precedence relation exists between tasks, the task is divided into independent tasks that have preparation time and closing time.

A_4) The execution time of a backup task is the same as that of the primary task.

A_5) There is no restriction on system resources.

A_6) There is a mechanism to promptly detect fault in a task.

In the next improved FSP algorithm, the following notations are used:

- A : A set of tasks that can be scheduled.
- H_i: A set of tasks that are included in the ith scheduling period, which means a set of tasks that do not need backup scheduling after a fault occurs in the first task in H_i.
- $W_{i,j}$: The scheduling time when a fault occurs in the jth task in the ith scheduling period.
- W_i: The longest schedule in the ith scheduling period.
- $C_{i,j}$: The execution time of task j in the ith scheduling period.
- $T_{acc}K$: The Boolean value to indicate whether the kth task is accepted or not (rejected).
- d_j : Deadline of task j

The improved FSP algorithm, by using the assumption that the minimum inter-fault time is always larger than ΔF, determines a set of tasks (H_i) that do not need backup after the first fault occurs (line 3~4). In line 6, with the assumption that the other task in H_i has a fault, it calculates scheduling time continuously. Among these schedules, it selects a schedule that has the longest scheduling time as a final schedule of the first scheduling period (line 10). At this time, if there are tasks that cannot meet their deadlines, they are removed from a task set A (line 7~9). As can be seen in the algorithm, the time complexity of our proposed algorithm is $O(n)$ because it executes a *For Loop* in line 2. It is comparable to the FSP algorithm whose time complexity is $O(n^2)$.

```
1   A = ∅; i=1 ; j=1 ; f_flag =False ;
      n = number of task ;                    (7)
2   For k=1,…,n Do {
3     determine H_k ;
4     W_k – W_k-1 ;
5       For j = min(H_k),…,max(H_k) Do {
6         W_k,j = W_k-1 + 2c_j + Σ_{l=1, l≠j, Taccept≠false}^{max(Hk)} C
7         If (W_k,j > d_j) Then
8           T_acc k = False ;
9         Else A ← T_j
10        W_k = max(W_k + W_k,j) ;
11        }
12  }
13  Return (A);
```

Figure 4. Improved FSP algorithm

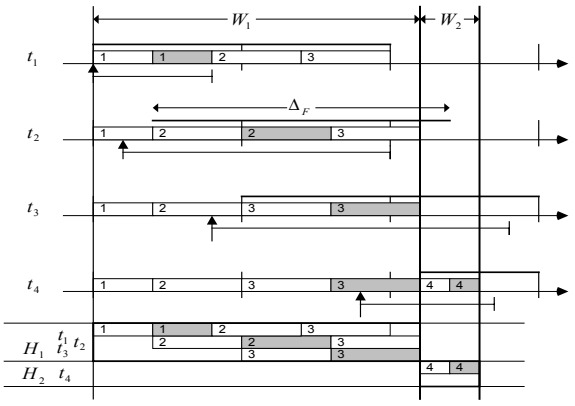

Figure 5. Example of scheduling with the improved FSP algorithm

In order to describe our algorithm more clearly, we will explain by simple example (Figure 5). Let's suppose that there are four tasks, T_1, T_2, T_3, and T_4. Their time values (arrival time, execution time, deadlines) are T_1(0,2,4), T_2(1,3,9), T_3(4,3,14), and T_4(9,1,13.5) respectively. We assume that ΔF is 10. First of all, we decide H_i that can be scheduled within ΔF with the assumption that a fault occurs in T_1. H_i will be $\{T_1, T_2, T_3\}$. After this, with the assumption that a fault occurs in T_2 or T_3, we calculate the corresponding scheduling time and select the longest one (W_1). In this example, the length of W_1 will be 11. In the next step, after a set of tasks H_2 is

decided (here, $H_2 = \{T_4\}$), the same procedure is repeated.

4. Performance Evaluation

In order to show the improvement offered by our scheduling algorithm, we did some simple experimentation with analytical analysis. In the experiment, we randomly generated 500 tasks having the average computation time (40) and the average deadline (120). We also generated faulted tasks with the fault rate (0.2) so that they had the minimum inter-fault time ΔF (200). Figure 6 shows the simulation results that estimate the number of rejected tasks between Ghosh's LTH algorithm and our EFSP (Extended FSP). We found that the number of rejected tasks is greatly affected by the ΔF value. In our experimentation, approximately half of tasks were rejected in case of LTH. On the contrary, many fewer tasks are rejected in our EFSP algorithm.

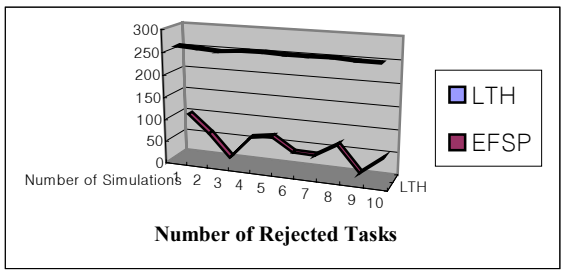

Figure 6. Simulation Results

5. Conclusion

Ghosh proposed two fault-tolerant scheduling algorithms for a real-time single processor environment: FSP (Feasible Shortest Path) algorithm and LTH (Linear Time Heuristics). Even though Ghosh devised a formal model for fault-tolerant real-time scheduling and developed scheduling algorithms based on that model, there is a drawback, since Ghosh's algorithm assumes that there is at most one fault within time interval Δf and does not consider inter-fault time. In this paper, we have proposed an improved FSP algorithm on the more realistic assumption that there is no additional fault during the minimum inter-fault time ΔF after one fault occurs. The proposed algorithm can improve system performance by including more primary tasks in a fault-tolerant schedule and is also more applicable to sporadic real-time task scheduling due to its similarity of task arrival pattern. The next step in our work is to clarify and improve our algorithm through careful quantitative analysis.

6. Reference

[1] S. Ghosh, R. Melhem, and D. Mosse. "Enhancing Real-Time Schedules to Tolerate Transient Faults," *Proceedings of Real-Time Systems Symposium*, pp. 120-129, Dec 1995.

[2] A. L. Liestman and R. H, Campbell, "A Fault-Tolerant Scheduling Problem", *IEEE Transactions on Software Engineering*, SE-12(11):1089-1095, Nov. 1986.

[3] T. Tsuchiya, Y. Kakuda, and T. Kikuno, "A New Fault-Tolerant Scheduling Technique for Real-Time Multiprocessor Systems," *Proc. '95 Real-Time Conference System Application*, pp. 197-202, 1995.

[4] D. Mosse, R. Melhem, and S. Ghosh, "Analysis of a Fault-Tolerant Multiprocessor Scheduling Algorithm," *Proc. 24th International Symposium on Fault-Tolerant Computing*, pp. 16-25, 1994.

[5] S. Ghosh, R. Melhem, and D. Moss , "Fault-Tolerant Scheduling on a Hard Real-Time Multiprocessor System," *Proc. 8th International Parallel Processing Symposium*, pp. 775-782, 1994.

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

[7] Y. Oh and S. Son, "Multiprocessor Support for Real-Time Fault Tolerant Scheduling," *Proc. Workshop Architectural Aspects of Real-Time Systems*, pp. 76-80 San Antonio, Tex., Dec. 1991

[8] Y. Oh and S. Son, "Fault-Tolerant Real-Time Multiprocessor Scheduling," *Technical Report TR-92-09*, University of Virginia, April 1992