Dual Token-Based Fault-Tolerant Scheduling for Hard Real-Time Multiprocessor Systems

Sungyoung Lee[1], Sam Kweon Oh[2] and Chul Hee Woo[1]
[1]Department of Computer Engineering, Kyung Hee University, Korea
{sylee@oslab.kyunghee.ac.kr}
[2]School of Computing, Hoseo University, Korea
{ohsk@dogsuri.hoseo.ac.kr}

Abstract

Real-time multiprocessor systems frequently assume that there exists a dedicated processor for task allocation that never fails. This assumption is, however, too strong in the sense that all the physical objects are subject to failure. Moreover, once the dedicated processor fails, the whole multiprocessor system will fail. As a way to solve this problem, we propose a fault-tolerant scheduling algorithm based on moving dual-token. While the primary processor holding a primary token performs task allocation, the backup processor holding a backup token, in case that the primary processor has failed, does primary processor creation. Since no dedicated processor for task allocation exists in this scheme, failure of the whole multiprocessor system due to that of the dedicated processor can be avoided. Meanwhile the deadline-based scheduling policy used for backup task allocation, compared to heuristic scheduling, allows easier implementation and improved scheduling predictability. Simulation results show that the proposed dual-token based algorithm yields low rejection rates over those with dedicated processor for task allocation.

footnote: Research supported in part by contract AB-97-G-0655 from the Department of Information and Communication in Korea

1. Introduction

Real-time systems usually have stringent performance and reliability requirements. Since failures in these systems may cause severe consequences, they are required to be *fault-tolerant, i.e.* tolerant of component failures. Task scheduling play crucial roles in maintaining high system reliability and good system performances. Due to diversified and expanded application domains, real-time systems tend to have highly complex and dynamic environments. In these systems, usually both periodic and aperiodic tasks exist. While periodic tasks can be scheduled at pre-run-time, aperiodic tasks in general cannot, due to unpredictable arrival times. An approach to scheduling such aperiodic tasks is to make run-time scheduling attempt at their arrival times.

Several scheduling algorithms have been developed for real-time systems. Since the general problem of optimal scheduling of tasks on a uniprocessor or a multiprocessor system is NP-complete, different heuristics have been used to schedule real-time tasks with the aim of maximizing performance measures such as acceptance ratio and processor utilization. Among these heuristics, only a few scheduling algorithms have attempted to provide fault tolerance in real-time systems [1, 2, 3, 4, 5, 6, 7].

Liestman and Campbell [1] proposed a fault-tolerant scheduling algorithm to handle transient faults. The tasks are assumed to be periodic, and short(backup) copies of all tasks are scheduled on a uniprocessor system to guarantee minimum performance for each task. This approach assumes that the task periods are multiples of each other. Krishna and Shin[KS86] suggest a run-time scheduling algorithm for periodic tasks in multiprocessor systems. Their algorithm handles processor failures by maintaining contingency backup schedules. These schedules are used in the event of processor failure. To generate the backup schedule, an optimal schedule is assumed to exist, and the schedule is enhanced with the addition of *"ghost"* tasks each of which functions as a backup task. Also Oh and Son [3, 4] describe a fault-tolerant scheduling strategy for periodic tasks in multiprocessor systems. In this strategy, a backup schedule is created for tasks in the primary schedule. The tasks are then rotated such that the primary and backup schedules are on different processors and do not overlap. Thus, it is possible to tolerate up to one processor failure in the worst case. To provide a single failure-tolerant schedule, this scheme requires twice the number of processors required for a non-fault-tolerant schedule. All of these algorithms deal with periodic tasks.

A fault-tolerant scheduling algorithm for aperiodic tasks in multiprocessor systems is suggested by Ghosh *et al.* [5]. This algorithm, unless another processor failure occurs until a previously failed processor has been recovered, guarantees a successful completion of scheduled tasks. In addition, by scheduling multiple backup tasks at the same time slot on the same processor and by releasing the resources reserved for a backup task when the primary task has been successfully completed, it allows scheduling of more tasks. In case of a processor failure, however, it requires backup tasks to have enough laxities for their scheduling and executions. Tsuchiya *et al.* [7] propose an algorithm with which successful task completions can be guaranteed even with insufficient laxities. This algorithm is, however, heuristic.

The scheduling algorithms for aperiodic tasks in multiprocessor systems have two problems in relation to processor failures. First, in case that a processor on which tasks are being executed and/or scheduled for execution fails, the backup copies of these tasks must be scheduled and executed on the other processors. Second these algorithms assume that a processor is dedicated for task allocation and that it never fails. This assumption is, however, too strong in the sense that all the physical objects are subject to failure. Moreover, once the dedicated processor fails, the whole multiprocessor system will fail. As a way to solve these problems, a fault-tolerant non-preemptive task scheduling algorithm based on moving dual-token is proposed. In this algorithm, the processor holding a primary token, so called the primary processor, takes the role of allocating tasks, and the backup processor holding a backup token performs processor selection and gives the processor selected a primary token in case that the primary processor has failed. The primary copy of a task is scheduled on the primary processor and the backup copy on the backup processor. Since the tokens are moving - i.e., there is no dedicated processor for task allocation - in this scheme, failure of the whole multiprocessor system due to that of the dedicated processor can be avoided.

The remainder of this paper is organized as follows. Section 2 presents a system model. Section 3 describes scheduling algorithms based on moving dual-token. Section 4 presents simulation results. Section 5 gives conclusion.

2. System Model

The system consists of N processors with shared memory. Figure 1 shows the system model. Three different types of queues exists in the system: a *global task queue,* three *processor queues (active, ready, and idle queues),* and N *local task queues*; all these queues are in the shared memory. While the processors with currently-executing tasks are queued in the order of *earliest completion time* in the active processor queue, the processors with the tasks that have been scheduled but have not been executed yet, are queued in the order of *largest free time-slot* in the ready queue. Lastly, the idle processors whose local task queue is empty are queued in the idle processor queue. The assumptions used in this paper are as follows:

A1) All tasks are independent and no resource limitations exist in the system.

A2) A primary task and its backup task are identical; i.e. they have the same execution times.

A3) The time between two successive processor failures is not less than the time between the start time of a primary task and the completion time of its backup task.

A4) We consider processor failures only.

A5) The primary token and the backup token can never be lost while they are being passed.

A6) All the processors in the system can participate in task scheduling, in addition to task allocation.

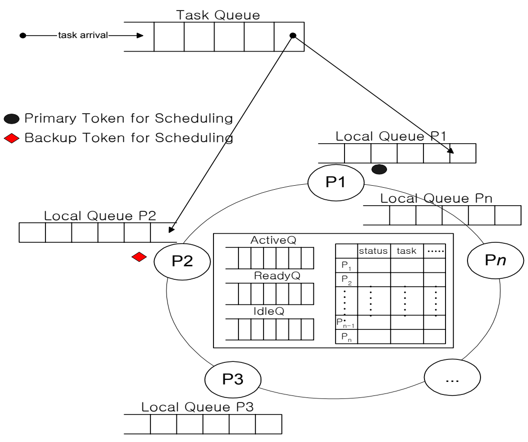

Figure 1. System model

The notations and terminologies used in this paper are as follows:

Tok(P): Primary token.

Tok(B): Backup token

PC(T): Task T's primary copy.

BC(T): Task T's backup copy

P: Processor ($P_{TOK(P)}$, $P_{TOK(B)}$, $P_{PC(T)}$, $P_{BC(T)}$)

T_a: Task T's arrival Time.

T_d: Task T's deadline.

T_c: Task T's computation time.

Q_T: Global task queue

Q_P: Processor queue (idle(***IdleQ***$_P$), ready(***ReadyQ***$_P$), active(***ActiveQ***$_P$))

RP(BC(T)): Redundant part of task T's backup copy.

BP(BC(T)): Recovery part of task T's backup copy

BP(T_c): The computation time of the backup copy of task T.

FS_{PP}: Primary processor's free time-slot.

FS_{BP}: Backup processor's free time-slot.

3. Dual Token-Based Fault-Tolerant Scheduling

3.1 The Basic Concept

The proposed scheduling algorithm consists of two parts: one for token allocation and the other for backup task scheduling. To allocate a primary token, the system first examines the *idle queue.* If there exists an idle processor in the queue, a primary token is assigned to that processor. If not, the *ready* and *active* queues are being searched; a primary token is assigned to the processor having the largest free time-slot in this case. The way how a backup token is allocated is similar to that of the primary token allocation. The backup processor that holds a backup token, in case that the primary processor has failed, performs processor selection and assigns the processor selected a primary token. In this way, failure of the whole multiprocessor system due to that of the primary processor can be avoided. Deadline-based scheduling approach is taken for the scheduling of backup tasks. In this approach, once the primary copy of a task is scheduled on the primary processor, the backup scheduler attempts to schedule the backup copy as close as possible to its deadline. By doing this, when the primary copy has been successfully completed, the backup copy can be deallocated; hence, the deallocated cpu resource can be used for scheduling other tasks.

The proposed dual-token based scheduling algorithm can be summarized as follows. The primary processor checks whether it can accept the primary copy of a task for scheduling. If so, the backup processor checks whether it can schedule the backup copy. Only when the scheduling of both copies are possible, they are scheduled for execution on the primary processor and the backup processor respectively. After the scheduling (allocation) of a task copy, token passing follows immediately. In this way, not only failure of the whole multiprocessor system due to that of a dedicated processor for task allocation can be avoided (since the tokens are moving, there's no dedicated processor for task allocation) but also the system utilization can be improved. Of course, a processor failure can also be tolerated since the primary copy and the backup copy of a task are scheduled on two different processors. Figure 2 shows the logical flow of the proposed scheduling algorithm.

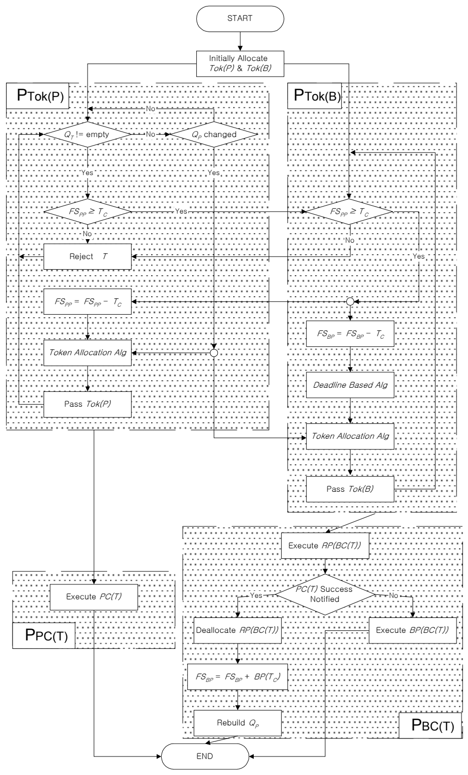

Figure 2. Logical flow of dual-token based scheduling algorithm

Since all the processors are idle at the beginning, the primary token and the backup token are given to two arbitrarily chosen processors respectively. There are four processor states: (***$P_{TOK(P)}$, $P_{TOK(B)}$, $P_{PC(T)}$, $P_{BC(T)}$***). In ***$P_{TOK(P)}$***, the processor performs task allocation; if there exists a task for scheduling in the global queue, it schedules the task if there exists a free-time slot enough for the scheduling of that task (***$FS_{PP} \geq T_c$***) (at this point in time, the backup processor also schedules the backup copy of that task in exactly the same way). Unless both the primary and the backup copies can be scheduled, the task is rejected. Once the task is scheduled, the primary processor decreases its free time-slot by the execution time of the task (***FS***$_{PP}$= ***FS***$_{PP}$ - ***T***$_c$). Then it passes the primary token to a processor with the largest free time-slot, by executing the token allocation algorithm. In ***P***$_{TOK(B)}$, the processor takes the role of the backup processor; it schedules backup task copies using a deadline-based non-preemptive scheduling algorithm. In case that a failure of the primary processor has been detected, it performs recovery by creating a new primary processor. When the primary copy of a task is accepted for scheduling by the primary processor, the backup processor checks whether it can schedule the backup copy (***FS***$_{BP} \geq$ ***T***$_c$). If so, the backup copy is scheduled as close as possible to its deadline. Then the token allocation algorithm is invoked to pass the backup token. The task that is not schedulable is rejected. In ***P***$_{PC(T)}$, the processor executes the primary copy of a task. On successful completion of the primary copy, the backup processor is notified of it, and hence it can perform deallocation of the backup copy. In ***P***$_{BC(T)}$, the processor, when informed of the completion of the primary copy of a task, performs the deallocation of the backup copy. When not informed of the completion of the primary copy, it assumes failure of the primary copy and hence it will execute the backup copy until completion.

3.2 Token Allocation Algorithm

The token allocation algorithm allocates primary and backup tokens to processors. A pseudo code for this algorithm is shown in Figure 3.

```
1: if a token-holding processor P has failed
2: then if P is the primary processor
3:        then Create a new primary token;
4:        else Create a new backup token;
            //P is the backup processor//
5: else Get P's token;
6: If IdleQ_p is not empty
7: then Choose a processor P' from
            the head of IdleQ_p;
8: else if ReadyQ_p is not empty
9:        then Choose a processor P' from
                 the head of ReadyQ_p;
10:       else Choose a processor P' from
                 the head of ActiveQ_p;
11: If processor P' has already had a token
12: then Ignore P' and Goto 6;
13: else allocate a token to P';
```

Figure 3. Token Allocation Algorithm

The pseudo code consists of two parts. At the first part (lines 1-5), a token that is to be passed is obtained. If the primary processor is shown to have failed, a new primary token is created (line 3). If the backup processor is shown to have failed, a new backup token is created (line 4). Otherwise (no failure), the token that is currently held by processor P is chosen. At the second part (lines 6-13), the obtained token is passed to a newly selected processor. Processor selection is done by inspecting the states of processor queues, in the order of idle, ready, and active queues; a processor with the largest free time-slot is chosen (lines6-10). At line 11, the processor selected is checked to see whether it has already had a token, which is necessary to prevent the processor from having both the primary and the backup token. Figure 4 shows a flow chart of the token allocation algorithm.

In case that the primary (or backup) processor has failed, the backup (or primary) processor performs token allocation to create a new primary (or backup) processor. Figure 5 shows this recovery process.

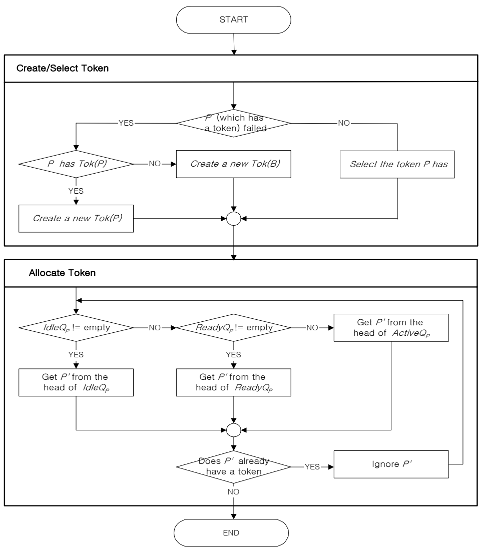

Figure 4. Flow chart of the token allocation algorithm

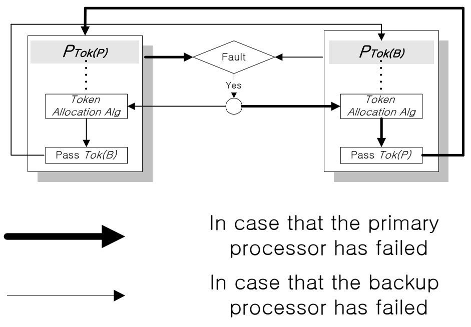
P_Tok(P)
Fault
P_Tok(B)
Yes
Token Allocation Alg
Token Allocation Alg
Pass Tok(B)
Pass Tok(P)
In case that the primary processor has failed
In case that the backup processor has failed

Figure 5. Recovery of the primary and backup processors

3.3 Deadline-based task scheduling

The basic idea is to make the backup processor schedule each backup task as close as to its deadline. The pseudo code is shown in Figure 6. First, it updates the arrival time of a task T (line 1). If there is no free-time slot ($\boldsymbol{FS_{BP}}$) large enough to schedule $\boldsymbol{T}$, it adjusts $\boldsymbol{T_a}$ in a way to get a larger free-time slot (lines 2-3). Once found, it sets $\boldsymbol{T_a}$ to be the start time of the $\boldsymbol{FS_{BP}}$ found (line 4). And the task ***BC(T)*** is queued in the local task queue. The flow chart of this algorithm is shown in Figure 7.

```
1: Update $T_a$ by $T_d$ - $T_c$;
2 : If $FS_{BP}$ which can hold BC(T) not exist
3 :     then Find the proper $FS_{BP}$;
4 :          Adjust $T_a$ to the start time of
                  the found $FS_{BP}$;
5 : Insert BC(T) into local $Q_T$;
```

Figure 6. Deadline-based task scheduling algorithm

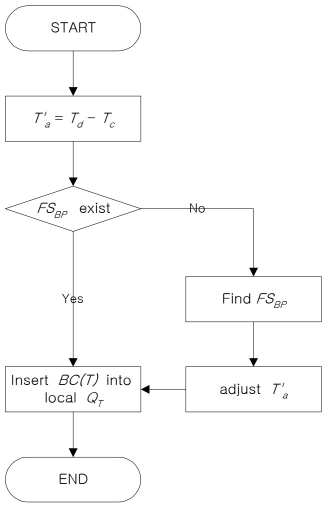

Figure 7. Flow chart of the deadline-based scheduling algorithm

Figure 8 shows an example. The primary copy of a task is scheduled as early as possible, while the backup copy scheduled as late as possible. By scheduling in this way, the chances of deallocating the backup task copies can be increased.

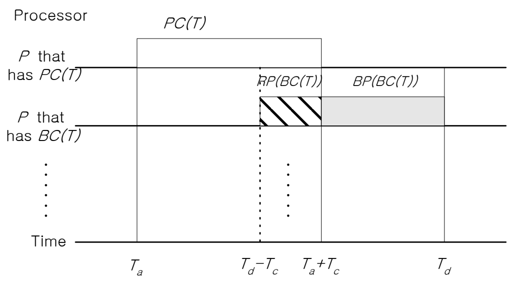

Figure 8. An example: task scheduling with deadline-based policy

4. Simulation Results

We compare our algorithm with those [5, 6] having a dedicated processor for task allocation. The parameters used for simulation are as follows:

- The number of processors (N): The number of processors is set to 6.
- The average task execution time (c): The task arrival is assumed to be uniformly distributed and the average execution time is set to 5.
- Processor load (γ): The total amount of cpu time used for task execution. The inter-task

arrival time becomes shorter as load γ increases. The inter-task arrival time is assumed to be uniformly distributed with average α $(=c/\gamma * P)$.

- The number of tasks is set to 1000.
- The task window w_i is defined to be the task deadline (d_i) - the task ready time (r_i).

The window size is uniformly distributed with average c $\times$ β.

Figure 9 shows the rejection ratios of tasks with varying window size and load. As shown in this figure, the dual-token based algorithm yields low rejection rates over that with a dedicated processor for task allocation. We can also observe that the rejection ratio increases with increasing load and with decreasing window size. However, the overhead incurred due to token passing has not been accounted for in this simulation. Further work and analysis will be pursued with such overhead taken into consideration.

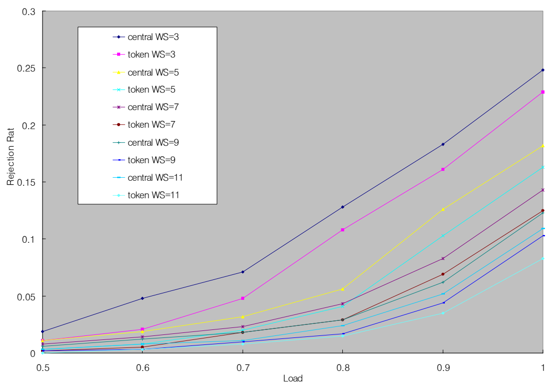

Figure 9. Rejection ratios with varying load and window size

5. Conclusion

A dual-token-based scheduling algorithm for fault-tolerant real-time multiprocessor systems has been proposed. Since no dedicated processor for task allocation exists in this scheme, failure of the whole multiprocessor system due to that of the dedicated processor can be avoided. In addition, the system utilization can be improved by utilizing all the processors in the system for task scheduling. Lastly, the deadline-based scheduling policy used for backup task allocation, compared to heuristic scheduling, allows easier implementation and improved scheduling predictability.

References

[1] A. L. Liestman and R. H. Campbell, "A Fault-Tolerant Scheduling Problem," IEEE Trans. Software Eng., vol. 12, no. 11, pp. 1,089-1,095 Nov. 1986.

[2] 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

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

[4] Y. Oh and S. Son, "Fault-Tolerant Real-Time Multiprocessor Scheduling," Technical

Report TR-92-09, University of Virginia, April 1992

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

[6] D. Mossé, 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.

[7] T. Tsuchiya, Y. Kakuda, and T. Kikuno, "A New Fault-Tolerant Scheduling Technique for Real-Time Multiprocessor Systems," International Workshop on Real-Time Computing Systems and Applications, pp. 197-202, 1995.