Effective Fault-tolerant Scheduling Algorithms for Real-time Tasks on Heterogeneous Systems

Yi-Hsuan Lee, Ming-Dien Chang, and Cheng Chen
Department of Computer Science and Information Engineering
National Chiao Tung University, Hsinchu, Taiwan, R.O.C.
E-mail: {yslee, mdchang, cchen}@csie.nctu.edu.tw

Abstract

Real-time systems are being increasingly used in several applications which are time critical in nature. Tasks corresponding to these applications have deadlines to be met. Due to the catastrophic consequences of not tolerating faults, fault-tolerance is also an important requirement of such systems. In this paper, we use the common *Primary/Backup* (*PB*) scheme to propose three algorithms, which are used to schedule real-time tasks with fault-tolerant requirements on heterogeneous systems. Our algorithms are modified and extended from existed algorithms, and aim at enhancing their performances without increasing the time complexity. Based on our simulation, all of them can achieve expected results. Besides, one of them also can use lower scheduling cost to obtain reasonable performances, which make it become much effectively and efficiently.

Keywords: *Real-time, Fault-tolerant, Task scheduling, Heterogeneous system*

1 Introduction

Real-time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced [3, 6, 9]. It can be broadly classified into three categories [12]. Among them the hard real-time system is most strict, and in this paper we will focus on it.

The demand for complex real-time applications, which require high computational needs with timing constrains and fault-tolerant requirements, has led to the choice of multi-processor systems. Due to the critical nature of tasks in a hard real-time system, every admitted task must complete its execution even in the presence of processor failures [3, 6]. In multi-processor systems, fault-tolerance can be provided by scheduling multiple versions of tasks on different processors [4, 8]. Four different schemes have evolved for fault-tolerant scheduling of real-time tasks [3, 5, 7, 11]. Among them we choose the *Primary/Backup* (*PB*) scheme, which is the most popular one.

Some effective scheduling algorithms used for real-time multiprocessor system have been proposed, but most of them are designed for homogeneous system [1-3, 6, 13]. In this paper, we propose three fault-tolerant task scheduling algorithms to schedule real-time tasks on heterogeneous system. These algorithms are modified and extended from existed algorithms [6, 14], and aim at enhancing their performances without increasing the time complexity. Based on our simulation, all of them can achieve

expected results. Moreover, our third algorithm can use lower scheduling cost to achieve reasonable performances, which make it become much effectively and efficiently.

The remainder of this paper is organized as follows. Section 2 surveys the fundamental background. Design issues and principles of our algorithms are introduced in Section 3. In Section 4, some experimental results are given. Finally, we give some conclusions in Section 5.

2 Fundamental Background

2.1 System, Task, and Fault Models [3, 6, 14]

The *heterogeneous system* consists of m heterogeneous processors, $P_1 \ldots P_m$, connected by a network. Every processor may fail due to hardware fault which results in task's failures. The faults can be transient or permanent and are independent. Each independent fault results in failing of only one processor. This assumption can be relaxed by grouping processors [1, 6].

Many real-rime task scheduling algorithms assume that tasks are independent, because [5] have proven that precedence constraints can be actually removed. Thus, we simply assume tasks are aperiodic, independent, non-preemptive, and not parallelizable. Every task T_i has following attributes: *ready time* (r_i), *computation time* on processor P_j (c_{ij}), and *deadline* (d_i). Each task T_i has two versions, *primary copy* (Pr_i) and *backup copy* (Bk_i), which have identical attributes. Since we assume tasks are not parallelizable, $d_i - r_i$ should be long enough so that both copies of T_i can be scheduled within this interval.

2.2 Basic Terminologies [3, 6, 14]

Definition 2.1 The *feasible schedule S* ensures that the timing and fault-tolerant constraints of all tasks are met. A *partial schedule* is one which does not contain all tasks.

Definition 2.2 For a task T_i, $st(Pr_i)$ and $ft(Pr_i)$ are *scheduled start time* and *scheduled finish time* of its primary copy Pr_i respectively.

Definition 2.3 For a task T_i, $st(Bk_i)$ and $ft(Bk_i)$ are *scheduled start time* and *scheduled finish time* of its backup copy Bk_i respectively.

Definition 2.4 The primary and backup copies of a task T_i are said to be *mutually exclusive in time* if $st(Bk_i) \geq ft(Pr_i)$.

Definition 2.5 For a task T_i, $proc(Pr_i)$ and $proc(Bk_i)$ are processors where its two copies Pr_i and Bk_i are scheduled.

Definition 2.6 The primary and backup copies of a task T_i are said to be *mutually exclusive in space* if $proc(Pr_i) \neq proc(Bk_i)$.

A task T_i is *feasible* in a fault-tolerant schedule if it satisfies the following two conditions. The first one is $r_i \leq st(Pr_i) < ft(Pr_i) \leq st(Bk_i) < ft(Bk_i) \leq d_i$, because both copies must satisfy the timing constraints and time exclusion. The second one is Pr_i and Bk_i should mutually exclude in space, which is necessary to tolerate permanent processor failures.

2.3 Related Work

Because PB scheme schedules two copies of a task on different processors, the entire schedulability is obviously decreased. Thus, *BB-overloading* technique, proposed in [3], describes that Bk_i and Bk_j scheduled on the same processor can overlap if $proc(Pr_i) \neq proc(Pr_j)$. Notice that applying this technique must assume that only one processor may fail at a time. Under this assumption, the principle of BB-overloading technique is quite trivial because $proc(Pr_i)$ and $proc(Pr_j)$ will not fail together.

Recently, many heuristic scheduling algorithms have been proposed to schedule real-time basks on multiprocessor system and support fault-tolerance [1-3, 6, 13]. Nevertheless, only a few of them focus on heterogeneous system. In the following we introduce *efficient Fault-tolerant Reliability Cost Driven* (*eFRCD*) [14], *Myopic Algorithm* [10], and *Distance Myopic Algorithm* (*DMA*) [6], which we choose to modify and design our methods.

eFRCD is a static algorithm to schedule real-time tasks on heterogeneous system. Unlike most real-time task scheduling algorithms, it considers factors such as precedence constraints among tasks, fault-tolerance, and reliability cost simultaneously. In *eFRCD*, tasks are ordered in non-decreasing order of deadlines at first. Then, Pr_i and Bk_i are allocated to processors with minimum *reliability costs* at the same time. Although *eFRCD* is entirely simple and efficient, it usually suffers from lower schedulability.

Myopic Algorithm (*MA*) is a heuristic search algorithm that schedules real-time tasks on multiprocessor system. It still orders tasks in non-decreasing order of deadlines, and uses two features doing scheduling. The first one is using a *feasibility check window* to contain the first K tasks in the task queue. The larger the feasibility check window, the higher the scheduling cost and the more the look-ahead nature. The second one is using an integrated heuristic function to select task. It will rearrange the sequence of tasks being scheduled, which can improve the schedulability by selecting the most appropriate task. Moreover, *MA* has the capability of *backtracking*. If the current schedule cannot be extended any more, it will deallocate the last scheduled task and try to schedule another one.

Distance Myopic Algorithm (*DMA*) is extended from *MA* to support fault-tolerance. It treats Pr_i and Bk_i of task T_i as separate tasks and constructs a task queue according to deadlines and variable *distance*. Unlike *eFRCD*, Pr_i and Bk_i will be scheduled separately in *DMA*. Basically, *DMA* can yield higher schedulability. But *DMA* is only designed for homogeneous system, and one of its main drawbacks is the difficulty to select variables K and *distance*.

3 Proposed Effective Algorithms

In this section, we introduce our three algorithms. All of our algorithms are integrated with BB-overloading technique.

3.1 Modified eFRCD Algorithm

The first *Modified eFRCD* algorithm is simplified from *eFRCD* to match our system and task models. Since we ignore precedence constraints and reliability cost, some variables used in *eFRCD* are redefined. The pseudo code of *Modified eFRCD* is shown in Figure 1.

Definition 3.1 For a task T_i, the *Latest Finish Time*(*LFT*), which is the time that the task copy must complete before, of its two copies are defined as follows.

$$\begin{cases} LFT(Pr_i) = d_i - \max_j c_{ij} \\ LFT(Bk_i) = d_i \end{cases}$$

Definition 3.2 For a task T_i, $EST_j(Pr_i)$ is the *Earliest Start Time* on processor P_j that can complete its primary copy before $LFT(Pr_i)$. $EST_j(Pr_i)$ should be within time interval $[\max(r_i, avail(j)), LFT(Pr_i) - c_{ij}]$, where $avail(j)$ is the time that P_j is available to execute Pr_i.

Definition 3.3 For a task T_i, $EST_j(Bk_i)$ is the *Earliest Start Time* on processor P_j that can complete its backup copy before $LFT(Bk_i)$.

1. **1.** Order tasks in non-decreasing order of deadlines
2. **for** (the unscheduled task T_i with minimal deadline)
   (a) **for** (each processor P_j) Calculate $EST_j(Pr_i)$
   (b) **if** (there exists processors that can complete Pr_i in-time)
       Allocate Pr_i to processor P_k with minimal $EST_k(Pr_i)$
   (c) **else** Reject T_i and go to Step 3
   (d) **for** (each processor P_j except $proc(Pr_i)$) Calculate $EST_j(Bk_i)$
   (e) **if** (there exists processors that can complete Bk_i in-time)
       Allocate Bk_i to processor P_k with minimal $EST_k(Bk_i)$
   (f) **else** Reject T_i and deallocate Pr_i from the schedule
3. Repeat Step 2 until all tasks are scheduled

Figure 1. Modified eFRCD Algorithm.

$EST_j(Bk_i)$ should be within time interval [**max** $(ft(Pr_i), avail(j))$, $LFT(Bk_i) - c_{ij}$], where $avail(j)$ is the time that P_j is available to execute Bk_i.

3.2 Heterogeneous Distance Myopic Algorithm

Modified eFRCD is actually very efficient, but it lacks for look-ahead nature. Furthermore, *Modified eFRCD* schedules task copies according to their earliest start time, which is not appropriate in heterogeneous system.

We have introduced that *DMA* uses the feasibility check window and an integrated heuristic function doing scheduling. Using feasibility check window can achieve the look-ahead nature, so *DMA* won't suffer from above drawback. Besides, authors of *MA* have proven that the integrated heuristic function performs better than simple heuristic function. Although *DMA* is quite effective, however, it is only designed for homogeneous system. Hence, our second *Heterogeneous Distance Myopic Algorithm* (*HDMA*) is extended from *DMA* to support the heterogeneous system.

Definition 3.4 For a task T_i, $EFT_j(Pr_i)$ and $EFT_j(Bk_i)$ indicate the *Earliest Finish Time* on processor P_j of its two copies respectively.

$$\begin{cases} EFT_j(Pr_i) = EST_j(Pr_i) + c_{ij} \\ EFT_j(Bk_i) = EST_j(Bk_i) + c_{ij} \end{cases}$$

Definition 3.5 For a task T_i, the *Earliest Finish Time* (*EFT*) of its two copies are defined as follows. Notice that if Pr_i is not yet scheduled, $EFT(Bk_i)$ is set to infinite.

$$\begin{cases} EFT(Pr_i) = \min_j EFT_j(Pri) + c_{ij} \\ EFT(Bk_i) = \min_j EFT_j(Bki) + c_{ij} \end{cases}$$

Definition 3.6 A partial schedule is *strongly feasible* if a feasible schedule can be generated by extending the current partial schedule with each task of the feasibility check window.

We redefine the integrated heuristic function $H = LFT(Pr_i)/LFT(Bk_i) + W \times EFT(Pr_i)/EFT(Bk_i)$, where W is an input parameter. Except the heuristic function, other scheduling mechanisms of *HDMA* are retained from *DMA*. Figure 2 shows the pseudo code of our *HDMA*.

3.3 Fault-Tolerant Myopic Algorithm

Since *HDMA* is directly extended from *DMA*, it retains two drawbacks. The first one is the difficulty of selecting variables *distance* and *K*. Authors of *DMA* have pointed out that the right combination offers the best scheduling performance, but this combination will depend on characteristics of the input task set [6].

1. Order tasks (primary copies) in non-decreasing order of deadlines and insert backup copies
2. **for** (all tasks in the feasibility check window) Calculate their $EFT(Pr_i/Bk_i)$
3. Check for strong feasibility:
4. **if** (strongly feasible or no more backtracking is possible)
   (a) Calculate heuristic values for tasks in the feasibility check window
   (b) Choose the task Pr_i/Bk_i with the smallest heuristic value to extend the schedule
   (c) **if** (the chosen task Pr_i/Bk_i can meet its latest finish time)
   $proc(Pr_i/Bk_i) = P_k$, where P_k is the processor with $EFT_k(Pr_i/Bk_i) = EFT(Pr_i/Bk_i)$
   (d) **else** Reject Pr_i/Bk_i
   (e) **if** (the rejected task is a primary copy) Delete its backup copy from the schedule
   (f) **else** Deallocate its primary copy from the schedule
5. **else** backtrack and try the task with the next smallest heuristic value. Goto Step 8
6. Move the feasibility check window by one task to the right
7. Repeat Steps 3 ~ 7 until all tasks in the task queue are considered

Figure 2. Heterogeneous Distance Myopic Algorithm (HDMA).

The second one is much implicit. In *DMA*, task copies are moved into the feasibility check window based on the predefined sequence. But actually, a backup copy becomes schedulable only after its primary copy has been scheduled yet. Therefore, the feasibility check window may contain unschedulable backup copies. In this situation, the feasibility check window seems to be smaller than its actual size, which may decrease the schedulability indirectly.

In order to overcome above drawbacks, we propose *Fault-Tolerant Myopic Algorithm* (*FTMA*). Like *DMA*, *FTMA* also follows scheduling mechanisms from *MA*. Our design features focus on task queue construction and feasibility check window movement, which are described in detail below.

In *FTMA*, primary and backup copies are separately ordered in non-decreasing order of deadlines. This feature can avoid the difficulty of selecting *distance*. During each scheduling step, we calculate heuristic values of the first primary and backup copies. The heuristic function is the same as *DMA*, and the task copy with smaller heuristic values will be moved into the feasibility check window. From Definition 3.5, only if a primary copy is scheduled, the heuristic value of its backup copy will not equal to infinite. That is, only schedulable backup copies will be moved into the feasibility check window. From above two features, *FTMA* can obviously overcome both drawbacks of *HDMA* without increasing the time complexity. Hence, we expect that *FTMA* could outperform *HDMA*. Figure 3 is the pseudo code of *FTMA*.

4 Performance Studies

4.1 Simulation Environment

In this section, we construct a simulation environment to evaluate above algorithms. Instead of randomly generating task sets, we construct a task set generator, similar to the one provided in *MA*, to guarantee schedulable task sets. Also, the tasks are generated to guarantee the (almost) total utilization of the processors. Notices that the schedule generated by the generator is used only for the purpose of generating a feasible task set of tasks which are then input to the scheduling algorithm, i.e., the scheduling algorithms have no knowledge of the

1. Order tasks in non-decreasing order of deadlines to construct primary and backup task queues
2. Put the first K tasks of primary task queue into the feasibility check window
3. **for** (all tasks in the feasibility check window) Calculate their $EFT(Pr_i/Bk_i)$
4. Check for strong feasibility
5. **if** (strong feasible or no more backtracking is possible)
   (a) Calculate heuristic values for tasks in the feasibility check window
   (b) Choose the task Pr_i/Bk_i with the smallest heuristic value to extend the schedule
   (c) **if** (the chosen task can meet its latest finish time)
   $proc(Pr_i/Bk_i) = P_k$, where P_k is the processor with $EFT_k(Pr_i/Bk_i) = EFT(Pr_i/Bk_i)$
   (d) **else** Reject Pr_i/Bk_i
   (e) **if** (the rejected task is a primary copy) Delete its backup copy from the task queue
   (f) **else** Deallocate its primary copy from the schedule
6. **else** backtrack and try the task with the next smallest heuristic value. Goto Step 10
7. Calculate heuristic values for the first primary copy Pr_i and the first backup copy Bk_i
8. Move Pr_i or Bk_i with smaller heuristic value into the feasibility check window
9. Repeat Steps 4 ~ 9 until all tasks in both primary and backup task queues are considered

Figure 3. Fault-Tolerant Myopic Algorithm (FTMA).

schedule itself but are only given the tasks and their requirements.

About the evaluating metric, we use *guarantee ratio* (*GR*) as *DMA* [6]. It is defined by the percentage of tasks arrived in the system whose deadlines are met. Hence, if the guarantee ratio is higher, the scheduling ability of that algorithm is higher, since the input task set is guaranteed schedulable. In the following we present three parts of simulation results, which highlight effects of parameters *laxity*, size of feasibility check window (K), and *distance* (d).

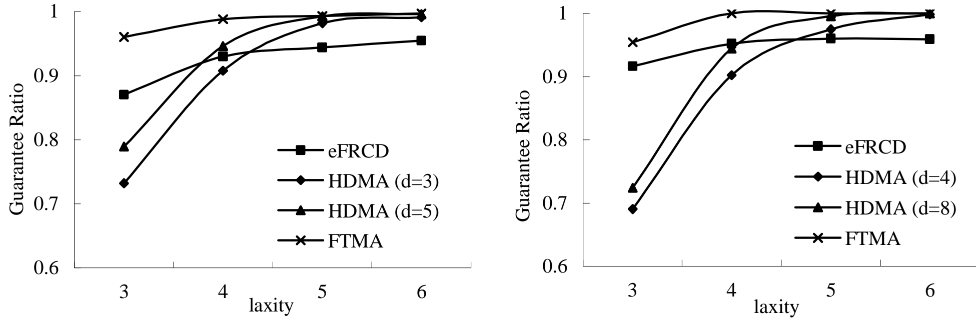
4.2 Experimental Results

At first, we discuss the effect of the weight value W in the integrated heuristic function. From simulation results, effects of different W values are little significance. Hence, we simply set $W = 1$ in the following evaluations.

The effect of *laxity* is studied in Figure 4. As the *laxity* increases, the guarantee ratio also increases for all three algorithms. This result seems trivial. Because larger *laxity* indicates the time interval between deadline and ready time of a task is longer, every algorithm has much flexibility to be more effective. Among them, *FTMA* is obviously the most effective one.

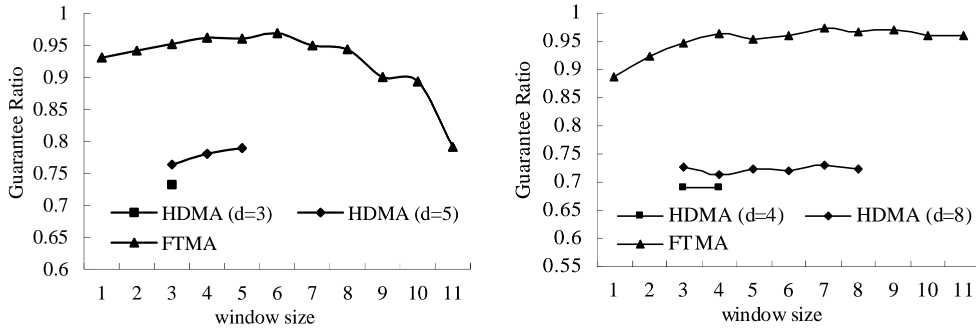
Figure 5 shows the effect of varying the size of feasibility check window for *HDMA* and *FTMA*. Authors of *DMA* recommend that if there are m processors, *distance* may be in the range [m/2, m] and K may be less than *distance*. In *HDMA* we follow this suggestion, and in *FTMA* K is unrestricted because it doesn't use *distance*. Similar as Figure 4, *FTMA* achieves much better results than that of *HDMA*. Moreover, guarantee ratio influenced by K is not significant, which indicates smaller feasibility check window also can obtain reasonable results, especially in *FTMA*. Since smaller K means lower scheduling cost, this feature can lead *FTMA* to a much efficient algorithm.

Finally, we study the effect of *distance*. This time we let $K = d$, which achieves better results theoretically. In Figure 6, whether the number of processor is, guarantee ratio increases



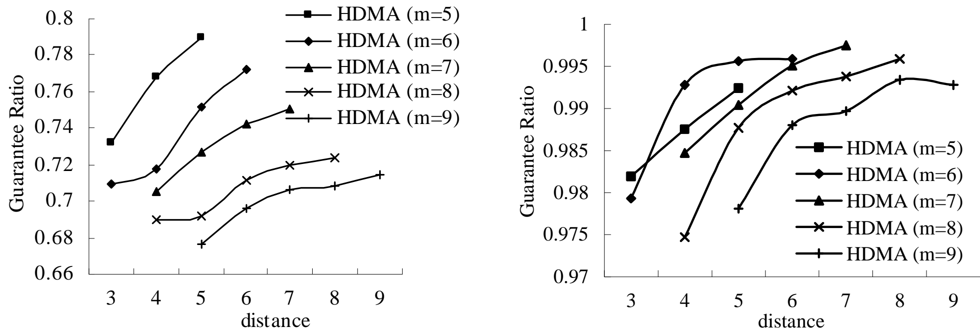
(a) (b)

Figure 4. Effect of task laxity. (a) 5 processors, (b) 8 processors.



(a) (b)

Figure 5. Effect of feasibility check window. (a) 5 processors, (b) 8 processors.



(a) (b)

Figure 6. Effect of task distance. (a) laxity = 3, (b) laxity = 5.

with increasing *distance*. The reason is larger *distance* make more consecutive primaries (or backups) be considered simultaneously, which helps to fully utilize all processors.

5 Conclusions

In this paper, we propose three fault-tolerant task scheduling algorithms used in heterogeneous system and construct a simulation environment to evaluate them. These algorithms are modified and extended from existed algorithms and aim at enhancing the guarantee ratio without increasing the time complexity. Based on our simulation, all of them can reach expected results. Besides, *FTMA* can obtain reasonable performances using smaller

feasibility check window, which means it is an effective and efficient algorithm.

References

[1] R. Al-Omari, G. Manimaran, and Arun K. Somani, "An Efficient Backup-overloading for Fault-tolerant Scheduling of Real-time Tasks", *Proc. of International Parallel and Distributed Processing Symposiu*m, pp. 1291-1295, 2000.

[2] R. Al-Omari, Arun K. Somani, and G. Manimaran, "A New Fault-tolerant Technique for Improving Schedulability in Multiprocessor Real-time Systems", *Proc. of International Parallel and Distributed Processing Symposium*, pp. 32-39, April 2001.

[3] S. Ghosh, R. Melhem, and D. Mosse, "Fault-tolerance Through Scheduling of Aperiodic Tasks in Hard Real-time Multiprocessor Systems", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 8, No. 3, pp. 272-284, March 1997.

[4] A. L. Liestman and R. H. Campbell, "A Fault-tolerant Scheduling Problem", *IEEE Trans. on Software Engineering*, Vol. 12, No. 11, pp. 1089-1095, Nov. 1988.

[5] J. W. S. Liu, W. K. Shih, K. J. Lin, R. Bettati, and J. Y. Chung", "Imprecise Computations", *Proc. of IEEE*, Vol. 82, No. 1, pp. 83-94, Jan. 1994.

[6] G. Manimaran and C. Siva Ram Murthy, "A Fault-tolerant Dynamic Scheduling Algorithm for Multiprocessor Real-time Systems and Its Analysis", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 9, No. 11, pp. 1137-1152, Nov. 1998.

[7] L. V. Mancini, "Modular Redundancy in a Message Passing System", *IEEE Trans. on Software Engineering*, Vol. 12, No. 1, pp. 79-86, Jan. 1986.

[8] Y. Oh and S. Son, "Multiprocessor Support for Real-time Fault-tolerant Scheduling", *Proc. of IEEE Workshop Architectural Aspects of Real-time Systems*, pp. 76-80, Dec. 1991.

[9] K. Ramamritham and J. A. Stankovic, "Scheduling Algorithms and Operating Systems Support for Real-time Systems", *Proc. of IEEE*, Vol. 82, No. 1, pp. 55-67, Jan. 1994.

[10] K. Ramamritham, J. A. Stankovic, and P.-F. Shiah, "Efficient Scheduling Algorithms for Real-time Multiprocessor Systems", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 1, No. 2, pp. 184-194, April 1990.

[11] P. Ramanathan, "Graceful Degradation in Real-time Control Applications Using (m, k)-firm Guarantee", *Proc. of IEEE Fault-tolerant Computing Symposiu*m, pp. 132-141, 1997.

[12] K. G. Shan and P. Ramanathan, "Real-time Computing: A New Discipline of Computer Science and Engineering", *Proc. of IEEE*, Vol. 82, No. 1, pp. 6-24, Jan. 1994.

[13] T. Tsuchiya, Y. Kakuda, and T. Kikuno, "A New Fault-tolerant Scheduling Technique for Real-time Multiprocessor Systems", *Proc. of 2nd International Workshop on Real-time Computing Systems and Applications*, pp. 197-202, Oct. 1995.

[14] X. Qin, H. Jiang, and D. R. Swanson, "An Efficient Fault-tolerant Scheduling Algorithm for Real-time Tasks with Precedence Constraints in Heterogeneous Systems", *Proc. of International Conference on Parallel Processing*, pp. 360-368, 2002.