An Efficient Fault-tolerant Scheduling Algorithm for Real-time Tasks with Precedence Constraints in Heterogeneous Systems [*]

Xiao Qin Hong Jiang David R. Swanson
Department of Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588-0115, {xqin, jiang, dswanson}@cse.unl.edu

Abstract

In this paper, we investigate an efficient off-line scheduling algorithm in which real-time tasks with precedence constraints are executed in a heterogeneous environment. It provides more features and capabilities than existing algorithms that schedule only independent tasks in real-time homogeneous systems. In addition, the proposed algorithm takes the heterogeneities of computation, communication and reliability into account, thereby improving the reliability. To provide fault-tolerant capability, the algorithm employs a primary-backup copy scheme that enables the system to tolerate permanent failures in any single processor. In this scheme, a backup copy is allowed to overlap with other backup copies on the same processor, as long as their corresponding primary copies are allocated to different processors. Tasks are judiciously allocated to processors so as to reduce the schedule length as well as the reliability cost, defined to be the product of processor failure rate and task execution time. In addition, the time for detecting and handling of a permanent fault is incorporated into the scheduling scheme, thus making the algorithm more practical. To quantify the combined performance of fault-tolerance and schedulability, the performability measure is introduced. Compared with the existing scheduling algorithms in the literature, our scheduling algorithm achieves an average of 16.4% improvement in reliability and an average of 49.3% improvement in performability.

1. Introduction

Heterogeneous distributed systems have been increasingly used for scientific and commercial applications, including real-time safety-critical applications, in which the system depends not only on the results of a computation, but also on the time instants at which these results become available. Examples of such applications include aircraft control, transportation systems and medical electronics. To obtain high performance for real-time heterogeneous systems, scheduling algorithms play an important role. While a scheduling algorithm maps real-time tasks to processors in the system such that deadlines and response time requirements are met, the system must also guarantee its functional and timing correctness even in the presence of faults.

The proposed algorithm, referred to as eFRCD *(efficient Fault-tolerant Reliability Cost Driven Algorithm),* endeavors to comprehensively address the issues of fault-tolerance, reliability, real-time, task precedence constraints, and heterogeneity. To tolerate one processor permanent failure, the algorithm uses a Primary/Backup technique to allocate two copies of each task to different processors. To further improve the quality of the schedule, a backup copy is allowed to overlap with other backup copies on the same processor, as long as their corresponding primary copies are allocated to different processors. As an added measure of fault-tolerance, the proposed algorithm also considers the heterogeneities of computation and reliability, thereby improving the reliability without extra hardware cost. More precisely, tasks are judiciously allocated to processors so as to reduce the schedule length as well as the reliability cost, defined to be the product of processor failure rate and task execution time. In addition, the time for detecting and handling of a permanent fault is incorporated into the scheduling scheme, thus making the algorithm more practical.

The rest of the paper is organized as follows. Section 2 briefly presents related work in the literature. Section 3 describes the workload and the system characteristics. Section 4 proposes the eFRCD algorithm and the main principles behind it, including theorems used for presenting the algorithm. Performance evaluation is given in Section 5. Section 6 concludes the paper by summarizing the main contributions of this paper.

[*] This work was supported by an NSF grant (EPS-0091900) and a Nebraska University Foundation grant (26-0511-0019)

2. Related work

The issue of scheduling on heterogeneous systems has been studied in the literature in recent years. A scheduling scheme, STDP, for heterogeneous systems was developed in [16]. In [3,17], reliability cost was incorporated into scheduling algorithms for tasks with precedence constraints. However, these algorithms neither provide fault-tolerance nor support real-time applications.

Previous work has been done to facilitate real-time computing in heterogeneous systems. In [7], a solution for the dynamic resource management problem in real-time heterogeneous systems was proposed. These algorithms, however, cannot tolerate any processor failure. Fault-tolerance is considered in the design of real-time scheduling algorithms to make systems more reliable.

In paper [6], a mechanism was proposed for supporting adaptive fault-tolerance in a real-time system. Liberato et al. proposed a feasibility-check algorithm for fault-tolerant scheduling [8]. The well-known Rate-Monotonic First-Fit assignment algorithm was extended in [2]. However, both of the above algorithms assume that the underlying system either is homogeneous or consists of a single processor.

The algorithm in [1] is a real-time scheduling algorithm for tasks with precedence constraint, but it does not support fault-tolerance. Manimaran et al. [9] and Mosse et al. [4] have proposed dynamic algorithms to schedule real-time tasks with fault-tolerance requirements on multiprocessor systems, but the tasks scheduled in their algorithms are independent of one another and are scheduled on-line. Martin [10] devised an algorithm on the same system and task model as that in [4]. Oh and Son studied a real-time and fault-tolerant scheduling algorithm that statically schedules a set of independent tasks [12]. Two common features among these algorithms [4,8,11, 12] are that (1) tasks are independent from one another and (2) they are designed only for homogeneous systems. Although heterogeneous systems are considered in both [17] and eFRCD, the latter considers fault-tolerance and real-time tasks while the former does not consider either.

Very recently, Girault et al. proposed a real-time scheduling algorithm for heterogeneous systems that considers fault-tolerance and tasks with precedence constraints [5]. This study is by far the closest to eFRCD that the authors have found in the literature. The main differences between [5] and eFRCD are three-fold: (a). eFRCD considers heterogeneities in computation, communication and reliability that will be defined shortly, whereas the former only considers computational heterogeneity. These hetero-geneities. (b). The former does not take reliability cost into consideration, whereas eFRCD is reliability-cost driven; and (c). The former allows the concurrent execution of primary and backup copies of a task while eFRCD allows backup copies of tasks whose primary copies are scheduled on different processors to overlap one another.

In the authors' previous work, both static [14,15] and dynamic [13] scheduling schemes for heterogeneous real-time systems were developed. One similarity among these algorithms is that the *Reliability Cost Driven Scheme* is applied. With the exception of the FRCD algorithm [15], other algorithms proposed in [13,14] cannot tolerate any failure. In this paper, the FRCD algorithm [15] is extended by relaxing the requirement that backup copies of tasks be not allowed to be overlapped.

3. Workload and system characteristics

A real-time job with dependent tasks can be modelled by *Directed Acyclic Graph (DAG)*, $T = \{V, E\}$, where $V = \{v_1, v_2, ..., v_n\}$ is a set of tasks, and a set of edges E represents communication among tasks. $e_{ij} = (v_i, v_j) \in E$ indicates a message transmitted from task v_i to v_j, and $|e_{ij}|$ denotes the volume of data being sent. To tolerate permanent faults in one processor, a primary-backup technique is applied. Thus, each task has two copies, namely, v^P and v^B, executed sequentially on two different processors. Without loss of generality, it is assumed that two copies of a task are identical. The proposed approach also is applied when two copies of each task are different.

The heterogeneous system consists of a set $P = \{p_1, p_2, ..., p_m\}$ of heterogeneous processors connected by a network. A processor communicates with other processors through message passing. A measure of *computational heterogeneity* is modeled by a function, $C: V \times P \rightarrow Z^+$, which represents the execution time of each task on each processor. Thus, $c_j(v_i)$ denotes the execution time of v_i on p_j. A measure of *communicational heterogeneity* is modeled by a function $M: E \times P \times P \rightarrow Z^+$. Communication time for sending a message e_{sr} from v_s on p_i to v_r on p_j is determined by $w_{ij} * |e_{sr}|$, where $|e_{sr}|$ is the communication cost and w_{ij} is the weight on the edge between p_i and p_j, representing the delay involved in transmitting a message of unit length between the two processors.

Given a task $v \in V$, $d(v)$, $s(v)$ and $f(v)$ denote the deadline, scheduled start time, and finish time, respectively. $p(v)$ denotes the processor to which v is allocated. These parameters are subject to constraints: $f(v) = s(v) + c_i(v)$ and $f(v) \leq d(v)$, where $p(v) = p_i$. A real-time job has a *feasible schedule* if for all $v \in V$, it satisfies both $f(v^P) \leq d(v)$, and $f(v^B) \leq d(v)$.

A k-timely-fault-tolerant *(k-TFT)* schedule is defined as the schedule in which no task deadlines are missed [12], despite k arbitrary processor failures. The goal of eFRCD is to achieve 1-TFT.

The reliability cost of task v_i on p_j is defined as the product of failure rate, λ_j, of p_j and v_i's execution time on p_j. It should be noted that *reliability heterogeneity* is implied in the reliability cost by virtue of heterogeneity in $c_j(v_i)$ and λ_j. Let RC_0 (R, Ψ) and $RC_i(R,\Psi)$ $(1 \le i \le m)$ be the reliability cost when no processor fails and when p_i fails, where Ψ is a given schedule and $R = \{\lambda_1, \lambda_2, \ldots, \lambda_m\}$ is a set of failure rates for the processors. RC_0 and RC_i are determined by equation (1) and (2), respectively.

$$RC_0(R,\Psi) = \sum_{i=1}^{m} \sum_{p(v^P)=i} \lambda_i c_i(v) \qquad (1)$$

$$RC_i(R,\Psi) = \sum_{j=1, j\neq i}^{m} \sum_{p(v^P)=j} \lambda_j c_j(v) + \sum_{j=1, j\neq i}^{m} \sum_{p(v^P)=i, p(v^B)=j} \lambda_j c_j(v)$$
$$= \sum_{j=1, j\neq i}^{m} \left(\sum_{p(v^P)=j} \lambda_j c_j(v) + \sum_{p(v^P)=i, p(v^B)=j} \lambda_j c_j(v) \right) \qquad (2)$$

In equation (2), the first summation term on the right hand side represents the reliability cost due to tasks whose primary copies reside in fault-free processors, while the second summation term expresses the reliability cost due to the backup copies of the tasks whose primary copies reside in the failed processor.

Reliability, given in the following expression, captures the ability of the system to complete parallel jobs in the presence of one processor permanent failure.

$$RL(R,\Psi) = e^{-RC(R,\Psi)} \qquad (3)$$

4. Scheduling algorithms

In this section, we present the eFRCD algorithm, which has three objectives, namely, (1) total schedule length is reduced so that more tasks can complete before their deadlines; (2) permanent failures in one processor can be tolerated; and (3) The system reliability is enhanced by reducing the overall reliability cost of the schedule.

4.1 An outline

The key for tolerating a single processor failure is to allocate the primary and backup copies of a task to two different processors such that the backup copy subsequently executes if the primary copy fails to complete due to its processor failure. Not all backup copies need to execute, even in the presence of a single processor failure. Since only tasks allocated to the failed processor are affected and need their backup copies to be executed, certain backup copies can be scheduled to overlap with one another. More precisely, a v^B is allowed to overlap with other backup copies on the same processor, if the corresponding primary copies are allocated to the different processors to which the v^P is not allocated. Thus, in a feasible schedule, the primary copies of any two tasks must not be allocated to the same processor if their backup copies are on the same processor and there is an overlap between two the backup copies. This statement is formally described as below.

Proposition 1. $\forall v_i, v_j \in V : \left(p(v_i^B) = p(v_j^B)\right) \wedge$
$\left(\left(s(v_i^B) \le s(v_j^B) < f(v_i^B)\right) \vee \left(s(v_j^B) \le s(v_i^B) < f(v_j^B)\right)\right) \rightarrow p(v_i^P) \ne p(v_j^P)$

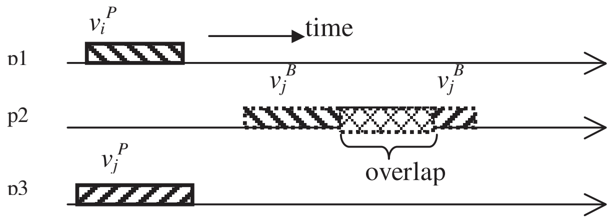

Fig. 1 Primary copies of v_i and v_j are allocated to p_1 and p_3, respectively, and backup copies of v_i and v_j are both allocated to p_2. These two backup copies can be overlapped with each other.

Fig. 1 shows an example illustrating this case. In this example, v_i^P and v_j^P are allocated to p_1 and p_3, respectively, and backup copies of v_i and v_j are both allocated to p_2. These two backup copies can be overlapped with each other because at most one of them will ever execute in the single-processor failure model.

The algorithm schedules tasks in the following three main steps. First, tasks are ordered by their deadlines in non-decreasing order, such that tasks with tighter deadlines have higher priorities. Second, the primary copies are scheduled. Finally, the backup copies are scheduled in a similar manner as the primary copies, except that they may be overlapped on the same processors to reduce schedule length. More specifically, in the second and third steps, the scheduling of each task must satisfy the following three conditions: (1) its deadline should be met; (2) the processor allocation should lead to the minimum increase in overall reliability cost among all processors satisfying condition (1); and (3) it should be able to receive messages from all its predecessors. In addition to these conditions, each backup copy has three extra conditions to satisfy, namely, (i) it is allocated on the processor that is different than the one assigned for its primary copy, (ii) its start time is later than the finish time of its primary copy plus the fault detection time δ and (iii) it is allowed to overlap with other backup copies on the same processor if their primary copies are allocated to different processors. Condition (i) and (ii) can be formally described by the following proposition.

Proposition 2. *A* schedule is 1-TFT
$$\rightarrow \forall v \in V : \left(p(v^P) \ne p(v^B)\right) \wedge \left(s(v^B) \ge f(v^P) + \delta\right).$$

4.2 The eFRCD algorithm

To facilitate the presentation of the algorithm, necessary notations are listed in the following table.

Table 1. Definitions of Notation

Notation	DEFINITION
D(v)	The set of predecessors of task *v*. $D(v) = \{v_i \mid (v_i, v) \in E\}$

$S(v)$	The set of successors of task v, $S(v) = \{v_i \mid (v, v_i) \in E\}$
$F(v)$	The set of feasible processors to which v^B can be allocated, determined in part by Theorem 2.
$B(v)$	The set of predecessors of v's backup copy, determined by Expression (7).
VQ_i	The queue in which all tasks are scheduled to p_i, $s(v_{q+1}) = \infty$, and $f(v_0) = 0$
$VQ_i'(v)$	The queue in which all tasks are scheduled to p_i, and cannot overlap with the backup copy of task v, where $s(v_{q+1}) = \infty$, and $f(v_0) = 0$
$v_i \succ v_j$	v_i is *schedule-preceding* v_j, if and only if $s(v_j) \geq f(v_i)$.
$v_i \Rightarrow v_j$	v_i is *message-preceding* v_j, if and only if v_i sends a message to v_j. Note that $v_i \Rightarrow v_j$ *implies* $v_i \succ v_j$ but not inversely.
$v_i \mapsto v_j$	v_i *execution-preceding* v_j, if and only if both tasks execute and $v_i \Rightarrow v_j$ *Note that* $v_i \mapsto v_j$ *implies* $v_i \Rightarrow v_j$ *and* $v_i \succ v_j$
$EAT_i^P(v)$	The *earliest available time* on p_i for v^P
$EAT_i^B(v)$	The earliest available time on p_i for v^B.
$EST_i^P(v)$	The earliest start time for v^P on processor p_i.
$EST_i^B(v)$	The earliest start time for v^B on processor p_i.

A detailed pseudocode of the eFRCD algorithm is presented below.

The eFRCD Algorithm:

Input: $T = \{V, E\}$, P, C, M, R /* DAG, Distributed System, Computational, Communicational and Reliability Heterogeneity */

Output: Schedule feasibility of T, *and* a viable schedule Ψ if it is feasible.

```
1. Sort tasks by the deadlines in non-decreasing order, subject to
precedence constraints, and generate an ordered list $OL$;
2. /* Schedule primary copies of tasks */
  for each task $v$ in $OL$, following the order, schedule $v^P$ do
 . 2.1 $s(v^P) \leftarrow \infty$; $rc \leftarrow \infty$; $VQ_i = \emptyset$;
  2.2 for each processor $p_i$ do /* Check if $v$ can be allocated to $p_i$ */
   /* Calculate $EST^P_i(v)$, where $VQ_i = \{ v_1, v_2, ..., v_q \}$ is the queue in */
    /* which all tasks are scheduled to $p_i$, $s(v_{q+1}) = \infty$, and $f(v_0) = 0$ */
    2.2.1 /*Compute the earliest start time of $v$ on $p_i$ */
        for ($j = 0$ to $q + 1$) do
        /* check if the unoccupied time intervals, interspersed */
        /* by currently scheduled tasks, can accommodate $v$ */
            if $s(v_{j+1}) - MAX\{f(v_j), EAT_i^P(v)\} \geq c_i(v)$ then
                $EST^P_i(v) = MAX\{f(v_j), EAT_i^P(v)\}$; break;
        end for
    2.2.2 /* Determine the earliest $EST_i$ based on Equation (6) */
        if $v^P$ starts executing at $EST^P_i(v)$ and can be
        completed before $d(v)$ then
          Determine reliability cost $rc_i$ of $v^P$ on $p_i$;
          /* Find the minimum reliability cost */
          if (($rc_i < rc$) or ($rc_i = rc$ and $EST^P_i(v) < s(v^P)$)) then
              $s(v^P) \leftarrow EST^P_i(v)$; $p \leftarrow p_i$; $rc \leftarrow rc_i$;
     end for
  2.3 if no proper processor is available for $v^P$, then return(FAIL);
  2.4 Assign $p$ to $v$, where the reliability cost of $v^P$ on $p$ is the minimal;
     $VQ_i \leftarrow VQ_i + v^P$;
  2.5 Update information of messages;
  end for
3. /*Schedule backup copies of tasks */
  for each task $v$ in the ordered list, schedule the backup copy $v^B$ do
  3.1 $s(v^B) \leftarrow \infty$; $rc \leftarrow \infty$;
     /* Determine whether the $v^B$ should be allocated to processor $p_i$ */
  3.2 for each feasible processor $p_i \in F(v)$, subject to Proposition 2 and
  Theorem 2, do          /* identify backup copies already scheduled */
   3.2.1 for ($v_j \in VQ_i$) do /* on $p_i$ that can overlap with $v^B$ */
            if ($v_j$ is a primary copy) or (($v_j$ is a backup copy) and
            ($p(v_j) = p(v)$)) then /* subject to Proposition 1 */
              copy $v_j$ into task queue $VQ_i'(v)$;
   3.2.2 Determine if $v^P$ is a strong primary copy (using Theorem 4);
   3.2.3 for (all $v_j$ in task queue $VQ_i'(v)$) do /*check the unoccupied */
        /* time intervals, and time slots occupied by backup copies */
       /* that can overlap with $v^B$, can accommodate $v^B$ */
         if $s(v_{j+1}) - MAX\{f(v_j), EAT_i^B(v)\} \geq c_i(v)$ then
           $EST^B_i(v) = MAX\{f(v_j), EAT_i^B(v)\}$; break;
        end for
   3.2.4 /*Determine the earliest $EST_i$ based on Equation (9) */
        if $v$ starts executing at $EST^B_i(v)$ and can be completed
         before $d(v)$ then
           Determine reliability cost $rc_i$ of $v^P$ on $p_i$;
           /* Find the minimum $rc$ */
           if (($rc_i < rc$) or ($rc_i = rc$ and $EST^B_i(v) < s(v^B)$)) then
             $s(v^B) \leftarrow EST^B_i(v)$; $p \leftarrow p_i$; $rc \leftarrow rc_i$;
        end if
     end for
  3.3 if no proper processor is available $v^B$, then return(FAIL);
  3.4 Find and assign $p \in F(v)$ to $v$, where the reliability cost of $v^B$ on $p$
is the minimal; $VQ_i \leftarrow VQ_i + v^B$;
  3.5 Update information of messages;
  3.6 for each task $v_j \in B(v)$ do  /* avoid redundant messages */
        $v_j$ sends message to $v^B$ if possible; (based on Theorem 1 and
        Expression (7) )
  3.7 for each task $v_j \in S(v)$ do  /* avoid redundant messages */
        if $p(v^P) \neq p(v_j^P)$ or $v^P$ is not a strong primary copy then
          $v^B$ sends message to $v_j^P$ if possible; (based on Theorem 3)
  end for
  return (SUCCEED);
```

4.3 The scheduling principles

Recall that $EST(v)$ and $EAT(v)$ are important to determine a proper schedule for a given task v. While both EAT and EST indicate a time when all messages from v's predecessors have arrived, EST additionally signifies that the processor to which v is allocated is now available for v to start execution. In the following, we present a series of derivations that lead to the final expressions for $EAT(v)$ and $EST(v)$.

If only one of v's predecessors $v_j \in D(v)$is considered, then the earliest available time $EAT_i(v, v_j)$ for the primary/backup copies of task v depends on the finish time $f(v_j)$, the earliest message start time $MST_{ik}(e)$, and the transmission time $w_{ik}*|e|$, for message e sent from v_j to v, where $p_k = p(v_j)$. Thus,

$$EAT_i(v, v_j) = \begin{cases} f(v_j) & \text{if } p_i = p_k \\ MST_{ik}(e) + w_{ik}*|e| & \text{otherwise} \end{cases} \quad (4)$$

Now consider all predecessors of v. Clearly v must wait until the last message from all its predecessors has arrived. Thus the earliest available time for v^P on p_i, $EAT_i^P(v)$ is the maximum of $EAT_i(v, v_j)$ over all the predecessors.

$$EAT_i^P(v) = MAX_{v_j \in D(v)} \left\{ EAT_i(v^P, v_j^P) \right\} \quad (5)$$

Based on expression (5), $EST_i^P(v)$ on p_i can be computed by checking the queue VQ_i to find out if the processor has an idle time slot that starts later than task's $EAT_i^P(v)$ and is large enough to accommodate the task. This procedure is described in step 2.2.1 in the algorithm. $EST_i^P(v)$ is applied to derive $EST^P(v)$, the earliest start time for v^P on any processor. Expression for $EST^P(v)$ is given below.

$$EST^{P}(v) = \underset{p_i \in P'}{MIN} \{EST_i^{P}(v)\} \quad (6)$$

where $P' = \left\{ p_i \in P'' \mid c_i(v) \times \lambda_i = \underset{p_{i_j} \in P''}{MIN} \{c_j(v) \times \lambda_j\} \right\}$, and

$P''=\{p_i \in P \mid EST_i^P(v) + c_i(v) < d(v)\}$.

$EST^B(v)$, the earliest start time for v^B, is computed in a more complex way than $EST^P(v)$. This is because the set of predecessors of v^P, $D^P(v)$, contains exclusively the primary copies of v's predecessor tasks, whereas the set of predecessors of v^B, $B(v)$, may contain a certain combination of the primary and backup copies of v's predecessor tasks. In order to decide $B(v)$, it is necessary to introduce the notion of *strong primary copy* as follows.

Note that there are two cases in which v^P may fail to execute: (1) $p(v^P)$ fails before time $f(v^P)$, and (2) v^P fails to receive messages from all its predecessors. Case (2) is illustrated by a simple example in Fig. 2 where dotted lines denote messages sent from predecessors to successors. Let v_j be a predecessor of v, and $p(v) \neq p(v_j)$. Suppose at time $t < f(v_j^P)$, $p(v_j^P)$ fails, then v_j^B should execute. Suppose v_j^B is not schedule-preceding v^P, v^P can not receive any message from v_j^B. Hence, even if $p(v^P)$ does not fail, v^P still can not execute. The primary copy of a task that never encounters case (2) is referred to as a *strong primary copy*, as formally defined below.

Definition 1. Given a task v, v^P is a strong primary copy, if and only if the execution of v^B implies the failure of $p(v^P)$ before time $f(v^P)$). Alternatively, given a task v, v^P is a strong primary copy, if and only if no failures of $p(v^P)$ at time $f(v^P)$) imply the execution of v^P.

Recall that one assumption is that only one processor will encounter permanent failures, we observe that if v_i is a predecessor of v_j, and the primary copies of both tasks are strong primary copies, then v_i^B is not message-preceding v_j^B. Fig. 3 illustrates a scenario of the case, which is presented formally in the theorem 1 that is helpful in determining the set of predecessors for a backup copy (See step 3.6).

Theorem 1. Given two tasks v_i and v_j, v_i is a predecessor of v_j. v_i^B is *not message-preceding* v_j^B, meaning that v_i^B does not need to send message to v_j^B, if v_i^P and v_j^P are both strong primary copies, and $p(v_i^P) \neq p(v_j^P)$.

Proof: Since v_i^P and v_j^P are both strong primary copies, according to Definition 1, v_i^B and v_j^B can both execute if and only if both v_i^P and v_j^P have failed to execute due to processor failures. But v_i^P and v_j^P are allocated to two different processors, an impossibility. Thus, at least one of v_i^B and v_j^B will not execute, implying that no messages need to be sent from v_i^B to v_j^B. □

Let $B(v) \subset V$ be the set of predecessors of v^B. It is defined as follows.

$$B(v) = \{ v_i^P \mid v_i \in D(v)\} \cup \{v_i^B \mid v_i \in D(v) \wedge (v_i^P \text{ is not a strong primary copy} \vee v^P \text{ is not a strong primary copy} \vee p(v_i^P) = p(v^P))\} = D^P(v) \cup D^B(v) \quad (7)$$

In the eFRCD algorithm, the primary copy is allocated before its corresponding backup copy is scheduled. Hence, given a task v and its predecessor $v_i \in D(v)$, two copies of v_i should have been allocated when the algorithm starts scheduling v^B. Obviously, v^B must receive

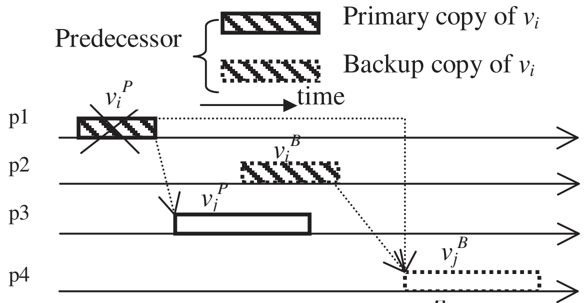

Fig. 2 Since processor p_1 fails, v_i^B executes. Becuase v_j^P can not receive message from v_i^B, v_j^B must execute instead of v_j^P.

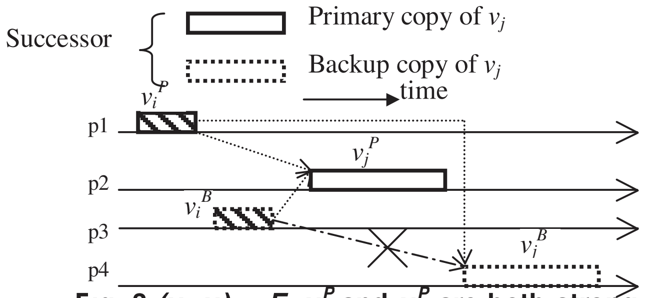

Fig. 3 $(v_i, v_j) \in E$, v_i^P and v_j^P are both strong primary copies, and v_i^P and v_j^P are scheduled on two different processors. v_i^B is not execution-preceding v_j^B.

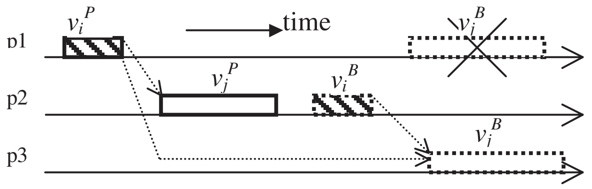

Fig. 4 $(v_i, v_j) \in E$, v_i^B is not schedule-preceding v_j^P and v_i^P is a strong primary copy. v_j^B can not be scheduled on the processor on which v_i^P is scheduled.

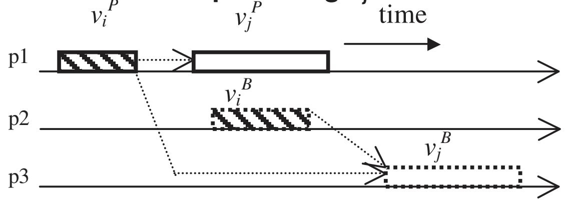

Fig. 5 v_i is the predecessor of v_j, v_i^P and v_j^P are scheduled on the same processor, and v_i^P is the strong primary copy. In this case, v_i^B is not execution-preceding v_j^P.

message from v_i^P and all $v_i^B \in D^B(v)$. Therefore, the maximum earliest available time of v^B on p_i is determined by the primary copies of its predecessors, the backup copies of tasks in $D^B(v)$ and messages sent from these tasks. $EAT_i^B(v)$ is given in the expression below, where δ is a certain amount of time to detect and handle the fault.

$$EAT_i^B(v) =$$
$$MAX\left\{f(v^P)+\delta, MAX_{v_j^P \in D^P(v)}\left(EAT_i(v^B, v_j^P)\right), MAX_{v_j^B \in D^B(v)}\left(EAT_i(v^B, v_j^B)\right)\right\}$$
$$= MAX_{v_j^P \in D^P(v), v_k^B \in D^B(v)}\left\{f(v^p)+\delta, EAT_i(v^B, v_j^P), EAT_i(v^B, v_j^B)\right\} \quad (8)$$

$EST_i^B(v)$ and $EST^B(v)$ denote the earliest start time for v^B on p_i, and the earliest start time for v^B on any processor, respectively. The computation of $EST_i^B(v)$ is more complex than that of $EST_i^P(v)$, due to the need to judiciously overlap some backup copies on the same processors. The computation of $EST_i^B(v)$ can be found from step 3.2.3 in the above algorithm. The *backup-overlapping scheme* (BOV) is implemented in step 3.2, which attempts to reduce schedule length by selectively overlapping backup copies of tasks. The expression for $EST^B(v)$ is given below,

$$EST^B(v) = \underset{p_i \in P'}{MIN}\left\{EST_i^B(v)\right\} \qquad (9)$$

where $P' = \left\{p_i \in P'' \mid c_i(v)\times\lambda_i = \underset{p_{i_j} \in P''}{MIN}\left\{c_j(v)\times\lambda_j\right\}\right\}$, and

$P'' = \{p_i \in F(v) \mid EST_i^B(v) + c_i(v) < d(v)\}$.

The candidate processor p_i in P'' is not chosen directly from the set P. Instead, it is selected from $F(v)$, a set of feasible processors to which the backup copy of v can be allocated. Obviously, $p(v^P)$ is not an element of $F(v)$. Given a task v, it is observed that under some special circumstance, v^B cannot be scheduled on the processor where the primary copy of v's predecessor v_i^P is scheduled (Fig. 4 illustrates this scenario). The set $F(v)$ can be generated will help of Theorem 2.

Theorem 2. Given two tasks v_i and v_j, $(v_i, v_j) \in E$, if v_i^B is not schedule-preceding v_j^P, and v_i^P is a strong primary copy, then v_j^B and v_i^P can not be allocated to the same processor.

Proof: Suppose the theorem is incorrect, thus, v_j^B and v_i^P are allocated to the same processor. Assume that v_i^B executes instead of v_i^P. This, combined with the fact that v_i^P is a strong primary copy, implies that $p(v_i^P)$ has failed before time $f(v_i^P)$. Since $f(v_i^P) < f(v_j^B)$, it also implies that $p(v_i^P)$ has failed before time $f(v_j^B)$, indicating that v_j^B will not execute. v_i^B cannot be execution-preceding v_j^P, since v_i^B is not schedule-preceding v_j^P. Hence, v_i^B must be execution-preceding v_j^B, impling that v_j^B does execute. A contradiction. □

Recall that $EAT_i(v, v_j)$ in expression (4) is a basic parameter used to derive $EAT_i^P(v)$ in expression (5) and $EAT_i^B(v)$ in expression (8). $EAT_i(v, v_j)$ is determined by the start time $MST_{ik}(e)$ of message e sent from $p_i = p(v)$ to $p_k = p(v_j)$. A message is allocated to a link if the link has an idle time slot that is later than the sender's finish time and is large enough to accommodate the message. $MST_{ik}(e)$ is computed by the following procedure, where $e = (v_j, v)$, $MST(e_{r+1}) = \infty$, $MST(e_0) = 0$, $|e_0| = 0$, and $MQ_i = \{e_1, e_2, \ldots, e_r\}$ is the message queue containing all messages scheduled to the link from p_i to p_k. This procedure behaves in a similar manner as the procedure for computing $EST_i^P(v)$ in step 2.2.1.

Computation of $MST_{ik}(e)$:
```
1.  for (g = 0 to r + 1) do /* Check whether the idle time slots */
    /* If the idle time slots can accommodate v, return the value */
2.   if $MST_{ik}(e_{g+1}) - MAX\{MST_{ik}(e_g) + w_{ik}*|e_g|, f(v_j)\} \geq w_{ik}*|e|$ then
3.        return $MAX\{MST_{ik}(e_g) + w_{ik}*|e_g|, f(v_j)\}$;
4.   end for
5. return ∞;  /* No such idle time slots is found, MST is set to be ∞ */
```

In scheduling messages, the proposed algorithm tries to avoid sending redundant messages in step 3.7, which is based on theorem 3. Suppose v_j^P has successfully executed, either v_i^P is execution-preceding v_j^P or v_i^B is execution-preceding v_j^P. We observe that, in some special cases illustrated in Fig 5, v_i^B will never be execution-preceding v_j^P. This statement is described Theorem 3.

Theorem 3. Given two tasks v_i and v_j, $(v_i, v_j) \in E$, if the primary copies of v_i and v_j are allocated to the same processor and v_i^P is a strong primary copy, then v_i^B *is not execution-preceding* v_j^P, meaning that sending a message from v_i^B to v_j^P would be redundant.

Proof: By contradiction: Assume v_i^B *is* execution-preceding v_j^P, thus, both v_i^B and v_j^P must execute (Table 1). Since v_i^P is a strong primary copy, processor $p(v_i^P)$ must have failed before time $f(v_i^P)$ (Def. 1). But v_i^P and v_j^P are allocated to the same processor and v_i^P is schedule-preceding v_j^P, implying that v_j^P also could not execute. A contradiction. □

The notion of strong primary copy appears in Theorems 1-3, it is therefore necessary to be able to determine whether a task has a strong primary copy. Theorem 4, applied to eFRCD in step 3.2.2, suggests an approach to determining whether a task has a strong primary copy. In this approach, we assume that we already know if all the predecessors have strong primary copies or not. By using this approach recursively, starting from tasks with no predecessors, we are able to determine whether a given task has a strong primary copy.

Theorem 4. (a) A task with no predecessors has a strong primary copy. (b) Given a task v_i and any of its predecessors v_j, if they are allocated to the same processor and v_j has a strong primary copy, or, if they are allocated on two different processors and the backup copy of v_j is message-preceding the primary copy of v_i, then v_i has a strong primary copy. That is, $\forall v_j \in V, (v_j, v_i) \in E$: $((p(v_i^P) = p(v_j^P) \wedge (v_j^P$ *is a strong primary copy*$)) \vee (p(v_i^P) \neq p(v_j^P) \wedge (v_j^B \Rightarrow v_i^P)) \rightarrow (v_i^P$ *is a strong primary copy*$)$.

Proof: As the proof of (a) is straightforward from the definition, it is omitted here. We only prove (b). Suppose before time $f(v_i^P)$, processor $p(v_i^P)$ does not fail. Let v_j be a predecessor of v_i. There are two possibilities: (1) $p(v_i^P) = p(v_j^P)$, we have $f(v_j^P) < f(v_i^P)$, implying that processor $p(v_j^P)$ does not fail before $f(v_j^P)$. Because v_j^P is a strong primary copy, v_j^P must execute. (2) $p(v_i^P) \neq p(v_j^P)$ and $v_j^B \Rightarrow v_i^P$, implying that even if one processor fails, v_i^P can still receive message from task v_j (recall that $v_j^P \Rightarrow v_i^P$). Based on (1) and (2), we have proven that v_i^P can receive messages from all its predecessors. Thus, v_i^P must execute since $p(v_j^P)$ has not failed by time $f(v_i^P)$. Therefore, according to Defnition 1, v_i^P is a strong primary copy. □

5. Performance evaluation

In this section, we compare the performance of the proposed algorithm with three other algorithms in the literature, namely, OV [12], FGLS [5], and FRCD [15] by extensive simulations.

Three performance measures are used to capture three important but different aspects of the algorithms. The first measure is *schedulability (SC)*, defined to be the percentage of parallel real-time *jobs* that have been successfully scheduled among all submitted jobs. The second is *reliability (RL)*, defined in expression (3). To combine the performances of the first two measures, the third measure, *performability (PF)*, is defined to be a product of *SC* and *RL*.

It is noted that the four algorithms differ in some aspects. First, OV assumes independent tasks and homogeneous systems, whereas FRCD, eFRCD and FGLS consider tasks with precedence constraints that execute on heterogeneous systems. Second, among FRCD, eFRCD and FGLS, while the former two incorporate computational, communicational and reliability heterogeneities into the scheduling, the latter considers only computational heterogeneity. To make the comparison fair, FGLS, FRCD and eFRCD are downgraded to handle independent tasks that execute on homogeneous systems.

Similarly, in Sections 5.4, the eFRCD algorithm is downgraded by assuming communicational homogeneity, while the FGLS algorithm is adapted to include reliability heterogeneity.

5.1 The workload

Workload parameters are chosen either based on those used in the literature [14,17] or represent realistic workload. In each simulation experiment, 100,000 real-time DAGs were generated independently for the scheduling algorithm as follows: First, determine the number of real-time tasks *N,* the number of processors *m* and their failure rates *R*. Then, the computation time in the execution time vector *C* is randomly chosen and uniformly distributed in a given range. Third, data communication among real-time tasks and communication weights are uniformly selected from *1* to *10*. Fourth, the failure rates were uniformly selected from a given range. Finally, the fault detection time δ is randomly computed according to a uniform distribution. Real-time deadlines can be defined in two ways:

1. A single deadline associated with a real-time job, which is a predetermined set of tasks with or without precedence constraints. Such a deadline, referred to as a *common deadline* [10,11,12], was employed in OV. To make a fair comparison, the common deadline is applied to FGLS, FRCD and eFRCD in simulation studies reported in Sections 5.2 and 5.3.
2. Individual deadlines associated with tasks within a real-time job. This deadline definition is often used for the dynamic scheduling of independent real-time tasks [4,8]. In sections 5.4, this deadline definition was adapted for tasks with precedence constraints. More specifically, given $v_i \in V$, if v_i is on p_k and v_j is on p_l, then v_i's deadline is determined by: $d(v_i) = MAX\{d(v_j)\} + |e_{ij}| \times w_{lk} + MAX\{c_k(v_i)\} + t$, where $e_{ij} \in E$, $k \in [1, m]$, t is chosen uniformly from a given range *H* that represents the individual relative deadline.

5.2 Schedulability

This experiment evaluates performance in terms of schedulability among the four algorithms, namely, OV, FGLS, FRCD and eFRCD, using the *SC* measure. The workload consists of sets of independent real-time tasks that are to be executed on a homogeneous distributed system. The size of the homogeneous system is fixed at 20, and a common deadline of 100 is selected. The failure rates are uniformly selected from the range between $0.5*10^{-6}$ and $3.0*10^{-6}$. Execution time is a random variable uniformly distributed in the range [1, 20]. *SC* is first measured as a function of task set size as shown in Fig. 6.

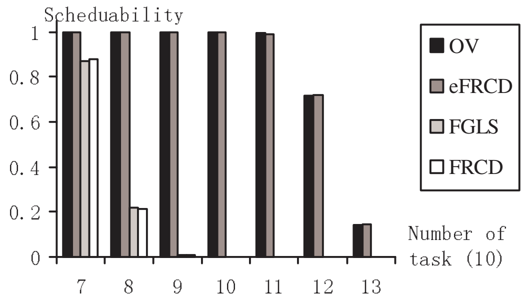

Fig. 6 SC as a function of N. Deadline= 100, m=16.

Fig. 6 shows that the SC performances of OV and eFRCD are almost identical, and so are FGLS and FRCD. Considering that eFRCD had to be downgraded for comparability, this result should imply that eFRCD is more powerful than OV, because eFRCD can also schedule tasks with precedence constraints to be executed on heterogeneous systems, which OV is not capable of.

The results further reveal that both OV and eFRCD significantly outperform FGLS and FRCD in *SC*, suggesting that both FGLS and FRCD are not suitable for scheduling independent tasks. The poor performance of FGLS and FRCD can be explained by the fact that they do not employ the BOV scheme. The consequence is twofold. First, FGLS and FRCD require more computing resources than eFRCD, which is likely to lead to a relatively low *SC* when the number of processors is fixed. Second, the backup copies in FGLS and FRCD cannot overlap with one another on the same processor, and this may result in a much longer schedule length.

5.3 Reliability performance

In this experiment, the reliability of the OV, FGLS, FRCD and eFRCD algorithms are evaluated as a function of maximum processor failure rate, shown in Fig. 7. To stress the reliability performance, *SC*s of all the four algorithms are assumed to be 1.0, by assigning an extremely loose common deadline. The task set size and system sizes are 200 and 20, respectively. Execution time of each task is chosen uniformly from the range between 500 and 1500. The failure rates were uniformly selected from range [$1.0*10^{-6}$, MAX_F], where MAX_F varies from $3.5*10^{-6}$ to $7.5*10^{-6}$ per hour with increments of $0.5*10^{-6}$.

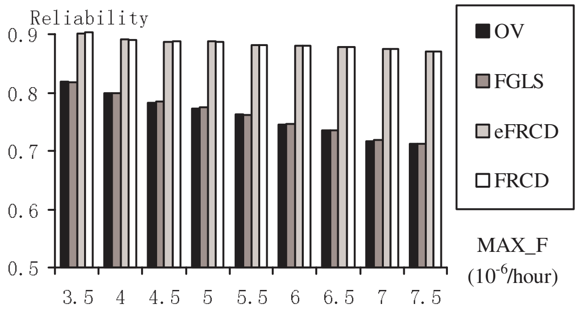

Fig. 7 Reliability as function of MAX_F. N = 50, m = 20.

As can be observed in Fig. 7, the *RL* of OV and FGLS are very close, and so are those of FRCD and eFRCD. FRCD and eFRCD perform considerably better than both OV and FGLS, with *RL* values being approximately from 10.5% to 22.3% higher than those of OV and FGLS. The FRCD and eFRCD algorithms have much better reliability simply because OV and FGLS do not consider reliability in their scheduling schemes while both FRCD and eFRCD take reliability into account. This experimental result validates the use of the proposed FRCD and eFRCD algorithm to enhance the reliability of the system, especially when tasks either have loose deadlines or no deadlines.

5.4 Effect of computational heterogeneity

The computational heterogeneity is reflected by the variance in execution times of the computation time vector *C*, and therefore a metric $\eta=(\alpha,\beta)$ is introduced to represent the computational heterogeneity level, where α =(MIN_E+MAX_E)/2 is the average value for execution time in *C*, and $\beta = \alpha$ - MIN_E is the deviation of *C*. Clearly, the higher the value of β, the higher the level of heterogeneity. To study the effect of the heterogeneity level on the *PF* of FGLS and eFRCD, α is fixed to 20 and β is chosen from 0 to 28 with increments of 4. Fig. 8 shows *PF* as a function of β, the heterogeneity level.

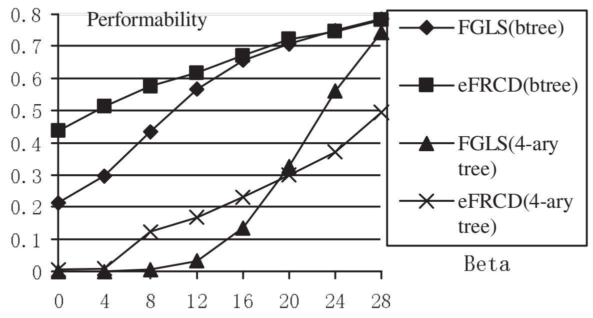

Fig. 8 PF of btree and 4-ary tree as a function of heterogeneity level. H=[1,100], N=150, m=20, alpha=20

The first observation from Fig. 8 is that the value of *PF* increases with the heterogeneity level. This is because *PF* is a product of *SC* and *RL*, and both *SC* and *RL* become higher when the heterogeneity level increases. These results can be further explained by the following reasons. First, though the individual relative deadlines are not affected by the change in computational heterogeneity, high variance in task execution times does affect the absolute deadlines, making the deadlines looser and the *SC* higher. Second, high variance in task execution times also provides opportunities for more tasks to be packed in with the fixed number of processors, giving rise to a higher *SC*. Third, *RC* decreases as the heterogeneity level increases, implying an increasing *RL*.

A second interesting observation is that eFRCD outperforms FGLS with respect to *PF* at low heterogeneity levels while the opposite is true for high heterogeneity levels. This is because when heterogeneity levels are low, both *SC* and *RL* of eFRCD are considerably higher than those of FGLS. On the other hand, eFRCD's *SC* is lower than that of FGLS at a high heterogeneity level, and *RLs* of two algorithms become similar when heterogeneity level increases. Therefore, eFRCD's *PF*, the product of *SC* and *RL*, is lower than that of FGLS at high heterogeneity levels. This result suggests that, if *SC* is the only objective in scheduling, FGLS is more suitable for systems with relatively high levels of heterogeneity, whereas eFRCD is more suitable for scheduling tasks with relatively low levels of heterogeneity. In contrast, if *RL* is the sole objective, eFRCD is consistently better than FGLS. In addition, Fig.8 indicates that performability of FGLS increases much more rapidly with heterogeneity level than that of eFRCD, implying that FGLS is more sensitive to the change in computational heterogeneity than eFRCD.

6. Conclusion

In this paper, an efficient fault-tolerant and real-time scheduling algorithm (eFRCD) for heterogeneous systems executing tasks with precedence constraints is studied. The fault-tolerant capability is incorporated in the algorithm by using a Primary/Backup (PB) model, in which each task is associated with two copies that are allocated to two different processors. eFRCD relaxes the requirement in FRCD [15] that forbids the overlapping of any backup copies to allow such overlapping on the same processor if their corresponding primary copies are allocated to different processors. The system reliability is further enhanced by reducing overall reliability cost while scheduling tasks. Moreover, the algorithm takes system and workload heterogeneity into consideration by explicitly accounting for computational, communicational, and reliability heterogeneity.

To the best of our knowledge, the proposed algorithm is the first of its kind reported in the literature, in that it most comprehensively addresses the issues of fault-tolerance, reliability, real-time, task precedence constraints, and heterogeneity. To assess the performance of eFRCD, extensive simulation studies were conducted to quantitatively compare it with the three most relevant existing scheduling algorithms in the literature, OV [12], FGLS [5], and FRCD [15]. The simulation results indicate that the eFRCD algorithm is considerably superior to the three algorithms in the vast majority of cases. There are two exceptions, however. First, the FGLS outperforms eFRCD marginally when task parallelism is low. Second, when computational heterogeneity is high, the eFRCD algorithm becomes inferior to the FGLS algorithm.

References

[1] T.F. Abdelzaher and K.G. Shin., "Combined Task and Message Scheduling in Distributed Real-Time Systems," *IEEE Transaction on Parallel and Distributed Systems*, Vol. 10, No. 11, Nov. 1999.

[2] Alan A. Bertossi, Luigi V. Mancini, Federico Rossini, "Fault-Tolerant Rate-Monotonic First-Fit Scheduling in Hard-Real-Time Systems," *IEEE Trans. Parallel and Distributed Systems*, 10(9), pp. 934-945, 1999.

[3] A. Dogan,F. Ozguner, "Reliable matching and scheduling of precedence-constrained tasks in heterogeneous distributed computing,"*In Proc. of the 29th International Conference on Parallel Processing*, pp. 307-314, 2000.

[4] 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, 1997

[5] A. Girault, C. Lavarenne, M. Sighireanu and Y. Sorel, "Fault-Tolerant Static Scheduling for Real-Time Distributed Embedded Systems," *In Proc. of the 21st International Conference on Distributed Computing Systems*(ICDCS), Phoenix, USA, April 2001.

[6] O. Gonzalez, H. Shrikumar, J.A. Stankovic and K. Ramamritham, "Adaptive Fault Tolerance and Graceful Degradation Under Dynamic Hard Real-time Scheduling," *In Proc. of the 18th IEEE Real-Time Systems Symposium*, San Francisco, California, December 1997.

[7] E.N. Huh, L.R. Welch, B.A. Shirazi and C.D. Cavanaugh, "Heterogeneous Resource Management for Dynamic Real-Time Systems," *In Proc. of the 9th Heterogeneous Computing Workshop*, 287-296, 2000.

[8] F. Liberato, R. Melhem, and D. Mossé, "Tolerance to Multiple Transient Faults for Aperiodic Tasks in Hard Real-Time Systems," *IEEE Transactions on Computers*, Vol. 49, No. 9, September 2000.

[9] G. Manimaran and C. Siva Ram Murthy, "A Fault-Tolerant Dynamic Scheduling Algorithm for Multiprocessor Real-Time Systems and Its Analysis," *IEEE Transactions on Parallel and Distributed Systems*, 9(11), November 1998.

[10] M. Naedele, "Fault-Tolerant Real-Time Scheduling under Execution Time Constraints," Sixth International Conference on Real-Time Computing Systems and Applications, Hong Kong, China, 13 - 15 December, 1999.

[11] Y.Oh and S.H.Son, "An algorithm for real-time fault-tolerant scheduling in multiprocessor systems," *4th Euromicro Workshop on Real-Time Systems*, Greece, 1992, pp.190-195.

[12]Y. Oh and S. H. Son, "Scheduling Real-Time Tasks for Dependability," Journal of Operational Research Society, vol. 48, no. 6, pp 629-639, June 1997.

[13]X. Qin, and H. Jiang, "Dynamic, Reliability-driven Scheduling of Parallel Real-time Jobs in Heterogeneous Systems," *In Proc. of the 30th International Conference on Parallel Processing*, Valencia, Spain, pp.113-122, 2001.

[14]X. Qin, H. Jiang, C.S. Xie, and Z.F. Han, "Reliability-driven scheduling for real-time tasks with precedence constraints in heterogeneous distributed systems," *In Proc. of the 12th International Conference Parallel and Distributed Computing and Systems,* pp.617-623, November 2000.

[15]X. Qin, H. Jiang, and D.R. Swanson, "A Fault-tolerant Real-time Scheduling Algorithm for Precedence-Constrained Tasks in Distributed Heterogeneous Systems," *Technical Report TR-UNL-CSE 2001-1003*, Department of Computer Science and Engineering, University of Nebraska-Lincoln, September 2001.

[16] S. Ranaweera, and D.P. Agrawal, "Scheduling of Periodic Time Critical Applications for Pipelined Execution on Heterogeneous Systems," *In Proc. of the 2001 International Conference on Parallel Processing*, Valencia, Spain, Sept 4-7, 2001, pp. 131-138.

[17] S. Srinivasan, and N.K. Jha, "Safty and Reliability Driven Task Allocation in Distributed Systems," *IEEE Trans. Parallel and Distributed Systems*, 10(3), pp. 238-251, 1999.