A Fault-tolerant Real-time Scheduling Algorithm for Precedence-Constrained Tasks in Distributed Heterogeneous Systems

Xiao Qin, Hong Jiang and David R. Swanson

Department of Computer Science and Engineering
University of Nebraska-Lincoln
{xqin, jiang, dswanson}@cse.unl.edu

Technical Report No. TR-UNL-CSE 2001-1003
September 2001
University of Nebraska-Lincoln
Lincoln, NE 68588-0115

A Fault-tolerant Real-time Scheduling Algorithm for Precedence-Constrained Tasks in Distributed 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 propose and evaluate a fault-tolerant real-time scheduling algorithm that can tolerate one processor's permanent fault in a heterogeneous distributed system. Workload in this study consists of a stream of real-time jobs where each job contains multiple precedence-constrained tasks with individual deadlines. A Primary Backup (PB) model is employed, where each real-time task has two copies, i.e. a primary one and a backup one, that are allocated to two different processors. The backup copy executes only if the primary copy fails due to the failure of its assigned processor. The proposed scheduling algorithm also takes the reliability measure into account, in order to further enhance the reliability of the heterogeneous system. In addition, the detection time for permanent fault is incorporated into the scheduling scheme so as to make the scheduling result more realistic and accurate. Simulation results show that the proposed algorithm provides significantly improved reliability and schedulability.

Keywords: Real-time scheduling, fault-tolerance, heterogeneous system, reliability cost

1. Introduction

Scheduling algorithms, be they at the job-level [7] or at the instruction-level [8], play an important role in obtaining high performance in parallel and distributed systems [6]. As a result, a wide variety of scheduling algorithms have been proposed and studied in the literature. The objective of scheduling algorithms is to map tasks onto processors and order their execution so that overall performance goals are achieved.

In general, a scheduling algorithm is designed to operate in the presence (or absence) of a number of operational conditions, depending on the workload, operational environment, complexity and cost constraints. These operational conditions include (but not limited to): (1) whether the operation is static [13] or dynamic [7] (i.e. off-line vs. on-line), (2) whether the workload has real-time constraints [30], (3) whether fault-tolerance [9] is considered, (4) whether the underlying system is heterogenous [25] or homogeneous, and (5) whether precedence constraints exist among tasks to be scheduled [3]. Unfortunately, most scheudling algorithms in the literature only consider one or two such conditions in isolation from others, thus limiting their power and applicability. Due to the ever increasing complexity and diversity of parallel and distributed systems in terms of workload, systems and operational characteristics, it has become increasingly desirable and necessary to design scheduling algorithms that can operate in the presence of all necessary conditions so as to satisfy any specific parallel/distributed computing requirement.

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

Thus, we are motivated in this study to investigate the possibility of designing scheduling algorithms that take the above five conditions into consideration simultaneously, superseding conditions set for most existing algorithms. More specifically, to make the scheduling algorithm more powerful and practical, we propose a scheduling scheme with which *real-time* tasks with *precedence constraints* can be *statically* scheduled to *tolerate the failure* of one processor in a *heterogeneous* parallel and distributed environment. While extending the proposed algorithm to incorporate on-line (*dynamic*) scheduling is possible (and is currently being developed), it should also be noted that the proposed algorithm can be easily "downgraded" to relax one or more of the imposed operational conditions. The purpose of this paper is to present and analyse such a scheduling algorithm.

The issue of scheduling on heterogeneous systems has been studied in many papers [25][21]. Ranaweera and Agrawal present a task-duplication based scheduling scheme (TDS) to schedule tasks of a directed acyclic graph (DAG) onto a heterogeneous system. A scalable scheduling scheme called STDS for heterogeneous systems is devised in [24]. In [5] and [29], reliability cost is incorporated into scheduling algorithms for tasks with precedence constraints. However, these algorithms either do not support fault-tolerance or assume that tasks in the system are not real-time.

Previous work has been done to facilitate real-time computing in heterogeneous systems. A solution for the dynamic resource management problem in real-time heterogeneous systems is proposed in [10]. In [27], a probabilistic model for a clients/server heterogeneous multimedia system is presented. In our previous work, both static [22] and dynamic [21] real-time scheduling schemes for heterogeneous systems were developed. The above algorithms, however, are not able to tolerate any permanent processor failure.

Since occurrences of faults are often unpredictable, fault-tolerance must be considered in the design of real-time scheduling algorithms to make systems more reliable [14][17]. Liberato et al. proposed a necessary and sufficient feasibility-check algorithm for fault-tolerant scheduling, assuming an earliest-deadline-first policy for aperiodic preemptive tasks [15]. A delayed scheduling algorithm using a passive replica method was developed in [2]. This scheme has a relatively small overhead for backup processes. However, the above algorithms share the same assumption that the underlying system is homogeneous with identical processors and uniform interconnection networks.

Algorithms proposed in [1][4][16][19][20][23][29] are the closest to the proposed algorithms in this paper in terms of operational conditions. The main difference between Abdelzaher and Shin's work and this study is that the algorithm studied in [1] assumes that there is no failure in processors, and thus does not support fault-tolerance. The proposed algorithm, however, is able to tolerate any one processor's failure, thus making the system more dependable. Dima et al. devised an offline real-time and fault-tolerant scheduling algorithm using replication of operations and data communications [4]. The difference between Dima et al.'s work and our proposed algorithm is that, while the former [4] must execute the backup copy simultaneously with the primary copy, ours schedules the backup copy after its primary copy, thus avoiding the unnecessary execution (and processor resource consumption) if the primary copy completes successfully. Manimaran et al. [16] and Mosse et al. [18][19] proposed dynamic algorithms to schedule real-time tasks with resource and fault-tolerance requirements on multiprocessor systems. There main differences between their work and ours lies in the workload where tasks considered in their algorithms are independent among one another and are scheduled on-line, whereas tasks considered in our algorithm are confined by precedence constraints and scheduled off-line. Oh and Son studied a real-time and fault-tolerant scheduling algorithm that statically schedules a set of independent tasks, and can tolerate one processor failure [20]. The major difference between Oh and Son's work [20] and ours is that their algorithm assumes no message communication among tasks, whereas our algorithm allows real-time tasks to communicate with one

other by passing messages. Furthermore, the above algorithms [1][4][16][18][19][20] are devised for homogeneous systems, thus limiting their applicability. Although Srinivasan and Jha's algorithm [29] and ours are both designed for heterogeneous systems, there are two main differences between them. Unlike our algorithm, theirs does not consider fault-tolerance, and tasks considered are non-real-time.

To the best of our knowledge, most scheduling algorithms do not consider fault-tolerance and reliability issues, only consider homogeneous systems, or assume independent tasks. In this paper, we attempt to address all these issues in a unified algorithm. The rest of the paper is organized as follows. Section 2 presents the system model and assumptions. Section 3 describes some scheduling principles behind the fault-tolerant real-time scheduling algorithm proposed in Section 4. Performance analysis is presented in Section 5. Section 6 concludes the paper by summarizing the main contributions of this paper and by commenting on future directions for this work. Finally, the appendix contains proofs for all the theorems and presents a simple example to elucidate the proposed algorithm.

2. The System Model

In this study, we consider non-pre-emptive real-time tasks running on a heterogeneous system, where processors may operate at different speeds and communication channels may have different bandwidths. Tasks are related to one another by their precedence constraints. Thus, these real-time tasks are modelled by Directed Acyclic Graphs (DAGs). When one processor fails, it takes extra, non-negligible time to detect and handle the fault, and we assume that the fault detection time is *dt*. Most scheduling models in the literature assume *dt* to be zero. To make the real-time scheduling more precise and realistic, we have incorporated the detection time into the scheduling algorithm.

To achieve the tolerance of permanent faults in one processor, multiple versions of tasks on different processors may be used. The *Primary Backup* (PB) model is one of many schemes. In this model, two copies of each task are executed serially on two different processors. For simplicity of presentation, we assume that primary and backup copies of a task are identical. This implies that two copies have the same execution times. It should be noted that our approach can also be applied when primary and backup copies of a task have different execution times. Since the focus of this work is to investigate a scheduling scheme that tolerates permanent failures in any one processor, failures of communication links connecting processors are not considered in this model. Therefore, we assume that the proposed scheduling scheme is based on reliable communication channels. Real-time and fault-tolerant communication protocols [11] can be applied to guarantee that messages arrive from one processor to another even in the presence of channel failures. Since the issue of fault-tolerant communication is beyond the scope of the current study, we will not discuss it further in this paper.

A real-time DAG is defined as a pair $T = \{V, E\}$, where $V = \{v_1, v_2,...,v_n\}$ represents the set of real-time tasks, and the set of weighted and directed edges E represents communication between pairs of real-time tasks. $e_{ij} = (v_i, v_j) \in E$ indicates a message sent from task v_i to v_j, and $|e_{ij}|$ denotes the volume of data sent between these tasks. In order to support fault-tolerance, each task has a primary and a backup copy, denoted v^P and v^B. Fig. 11 in Appendix E shows an example DAG which we will use throughout the paper.

The heterogeneous system is modeled by a set of processors $P = \{p_1, p_2,..., p_m\}$, where p_i is a processor with local memory. Processors in the system are connected with one other by a high-speed interconnection network. A processor communicates with other processors through message passing, and the communication time between two tasks assigned to the same processor is assumed to be zero.

A measure of *computational heterogeneity* is modeled by a function, $C: V \times P \rightarrow R$, which represents the execution time of each task on each available processor in the distributed system. $c_j(v_i)$

denotes the execution time of task v_i on processor p_j. Since we assume that for each task v_i, its primary copy v_i^P and backup copy v_i^B have the same version, we get $c_j(v_i^P) = c_j(v_i^B) = c_j(v_i)$. A measure of *communicational heterogeneity* is modeled by a function $M: E \times P \times P \rightarrow R$. That is, the communication time for sending a message e from task v_s on processor p_i to task v_r on processor p_j is determined by $w_{ij}*|e|$. w_{ij} is the weight on the edge between processors p_i and p_j , representing the delay involved in transmitting a message of unit length between the two processors.

For each 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 task v is allocated. These parameters are subject to constraints: $f(v) = s(v) + c_i(v)$ and $s(v) \leq d(v) - c_i(v)$, where $p(v) = i$. A parallel job has a feasible schedule if for each real-time task $v \in V$, it satisfies constraints $s(v^P) \leq d(v) - c_i(v)$, and $s(v^B) \leq d(v) - c_j(v)$, where $p(v^P) = i$, $p(v^B) = j$.

To make the system more reliable without any extra hardware cost, we also introduce a reliability model, which is similar to those proposed by Srinivasan et al. [29] and Qin et al. [21][22]. In this model, processor failures are assumed to be independent, and follow a Poisson Process with a constant failure rate. The reliability cost of a task v_i on a processor p_j is the product of p_j's failure rate λ_j and v_i's execution time on p_j. It should be noted that the notion of *reliability heterogeneity* is implied in the reliability cost by virtue of heterogeneity in $c_j(v_i)$ and λ_j. Thus, the reliability cost of a task schedule is the summation over all tasks' reliability costs based on the given schedule. Given a heterogeneous system P, the reliability cost is defined below. $RC_0(T)$, which denotes the reliability cost with no failure in the system, is defined as:

$$RC_0(T) = \sum_{j=1}^{m} \sum_{p(v_i^P)=j} \lambda_j c_j(v_i) \quad (1)$$

Here reliability is represented by $e^{-RCo(T)}$. To achive a high overall reliability, it is intuitive to schedule a task with a longer execution time to a more reliable processor.

3. The Scheduling Principles

The algorithm to be presented in Section 4 relies heavily on the knowledge and understanding of relationships among the primary and backup copies, and predecessors and successors of tasks. In order to qualify or quantify some of these relationships, thus facilitating the development of the algorithm, in the subsections that follow we will present the necessary definitions, assumptions, theorems and corollaries.

3.1 Definitions and Assumptions

To facilitate the presentation of the proposed scheduling principles, additional definitions and assumptions are introduced.

DEFINITION 1. Given a processor $p_i \in P$, $\mathbf{@(p_i, t)}$ denotes that the processor has no failures at time $t' \leq t$, whereas $\mathbf{\sim@(p_i, t)}$ represents a processor with software or hardware failures at time $t' > t$.

DEFINITION 2. Given a task $v \in V$, $\mathbf{\Theta v}$ signifies that task v successfully executes on processor $p(v)$, whereas $\mathbf{\sim\Theta v}$ signifies that v does not work.

DEFINTION 3. Given two tasks v_i and v_j, v_i is ***schedule-preceding*** v_j, denoted $\mathbf{v_i \succ v_j}$, if and only if $s(v_j) \geq f(v_i)$.

DEFINITION 4. Given two tasks v_i and v_j, v_i is ***message-preceding*** v_j, denoted $\mathbf{v_i \Rightarrow v_j}$, if and only if $(v_i, v_j) \in E$, and v_i sends a message to v_j.

DEFINITION 5. Given two tasks v_i and v_j, there is an ***execution-precedence*** relationship between v_i and v_j, denoted $\mathbf{v_i \mapsto v_j}$, if and only if $(v_i, v_j) \in E$, Θv_i, Θv_j, $v_i \Rightarrow v_j$ and $v_i \succ v_j$.

DEFINITION 6. $\boldsymbol{D(v) \subset V}$ denotes the set of predecessors of task v, $D(v) = \{ v_i \in V \mid (v_i, v) \in E \}$. That is, v cannot start until it receives messages from all tasks in $D(v)$ due to precedence constraints. $\boldsymbol{S(v) \subset V}$ denotes the set of successors of task v, $S(v) = \{ v_i \in V \mid (v, v_i) \in E \}$.

From the above definitions, it is clear that, given three tasks v_i, v_j and v_k, if $v_i \succ v_j$ and $v_j \succ v_k$, then $v_i \succ v_k$. It is noted that the execution-precedence relationship "$\mapsto$" and message-precedence relationship "$\Rightarrow$" are different from schedule-precedence relationship "$\succ$". The last is transitive, whereas none of the first two is transitive. The execution-precedence relationship is the strongest in that it can imply the other two. The schedule-precedence relationship, on the other hand, is the weakest. This is described in Property 1 as follows.

Property 1. Given two tasks v_i and v_j, if v_i is execution-preceding v_j, then v_i is message-preceding v_j. If v_i is message-preceding v_j, then v_i is schedule preceding v_j. Thus, $\boldsymbol{(v_i \mapsto v_j) \rightarrow (v_i \Rightarrow v_j) \rightarrow (v_i \succ v_j)}$.

If task v_i is a predecessor of task v_j, then the primary copy of v_i must send a message to the primary copy of v_j. This is described formally in the following property.

Property 2. Given two tasks v_i and v_j, if v_i is a predecessor of v_j, then v_i^P must be message-preceding v_j^P, that is, $\boldsymbol{\forall v_i, v_j \in V: (v_i, v_j) \in E \rightarrow v_i^P \Rightarrow v_j^P}$.

Given a task v, the backup copy of v executes if the primary copy does not work. There are two cases in which v^P could not work. (1) Before time $f(v^P)$, processor $p(v^P)$ fails. In other words, $\exists\, t < f(v^P)$: $\sim@(p(v^P), t) \rightarrow \sim\Theta v^P$. (2) Processor $p(v^P)$ has no failures before time $f(v^P)$, but v^P can not receive messages from all its predecessors. Case (2) is illustrated by a simple example in Fig. 1 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 $t < f(v_j^P)$, processor $p(v_j^P)$ fails, then v_j^B should execute. Since 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 7. 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 $f(v^P))$, thus, $\Theta v^B \rightarrow \sim@(p(v^P), f(v^P))$. Alternatively, given a task v, v^P is a strong primary copy, if and only if that no failures of $p(v^P)$ at time $f(v^P))$ imply the execution of v^P, i.e. $@(p(v^P), f(v^P)) \rightarrow \Theta v^P$.

3.2 The Principles

We use the primary/backup approach to satisfy the fault-tolerance requirement. In this approach, it should be obvious that the primary and backup copies of each task must be scheduled on two different processors.

The backup copy of a task executes only when the primary copy could not be finished. This requires that the start time of the backup copy be later than the finish time of the primary copy. This requirement is described in the following proposition and its proof can be found in [18].

Proposition 1. One processor's failure is tolerable, only if for each task v, there is no overlapping time between v^P and v^B, and the start time of v^B is greater than or equal to the sum of the finish time of v^P and the fault detection time. Thus, $\boldsymbol{\text{(One processor's failure is tolerable)} \rightarrow \forall v \in V: s(v^B) \geq f(v^P) + dt}$.

Suppose that the scheduling result can tolerate one processor's failure, one essential precondition is that, for each task v, the sum of the execution times of v's primary/backup copies and the fault detection time is less than or equal to its deadline. It is given as the proposition below. The proof of proposition 2 is straightforward, and it is omitted in this section.

Proposition 2. One processor failure is tolerable, only if (a) primary and backup copies are allocated to different processors, and (b) the sum of execution times for v^P and v^B and fault detection time is less than or equal to the deadline, that is,

(One processor's failure is tolerable) $\rightarrow \forall v \in V: (p(v^P) = i \neq p(v^B) = j) \wedge c_i(v^P) + c_j(v^B) + dt \leq d(v)$.

Given two tasks v_i and v_j, $(v_i, v_j) \in E$. The relationships among v_i^P, v_j^P, v_i^B, and v_j^B are the key issues to be considered in the scheduling algorithm. Our main task is to determine the execution-precedence and the message-precedence relationships among these copies. Suppose the primary copy of task v_j 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, v_i^B will never be execution-preceding v_j^P. This statement is described and proved in Theorem 1. This case is also illustrated in Fig 2.

Theorem 1. 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. That is, $v_j \in V$, $(v_i, v_j) \in E$: $p(v_i^P) = p(v_j^P) = p \wedge (v_i^P$ *is the strong primary copy)* $\rightarrow \sim(v_i^B \mapsto v_j^P)$.

Proof: See Appendix. □

The result of Theorem 1 is very interesting and useful for the scheduling algorithm. It suggests that if a task v_j is allocated to the same processor as its predecessor v_i, and v_i^P is a strong primary copy, then the backup copy of v_i only needs to send messages to the backup copy of v_j.

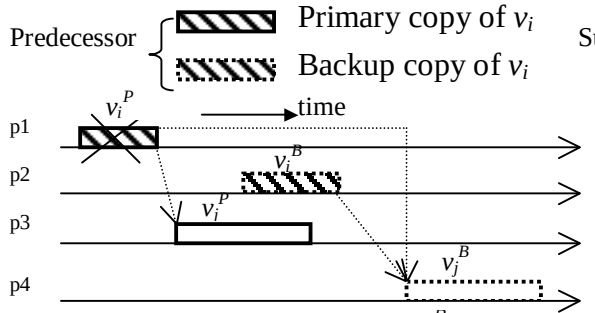

Fig. 1 Since processor p1 fails, v_i^B executes. Becuase v_j^P can not receive message from v_i^B, v_j^B must execute instead of v_j^P.

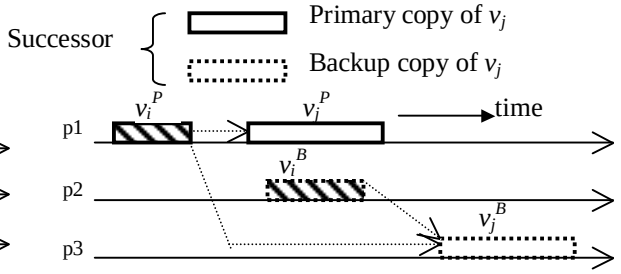

Fig. 2 v_i is the predecessor of v_j, v_i^P and v_j^P are scheduled on the same processor, and v_i^P is the strong primary copy. In this case, v_i^B is not execution-preceding v_j^P.

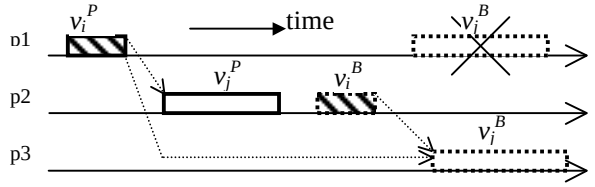

Fig. 3 $(v_i, v_j) \in E$, v_i^B,is not schedule-preceding v_j^P and v_i^P is a strong primary copy. v_j^B can not be scheduled on the processor on which v_i^P is scheduled.

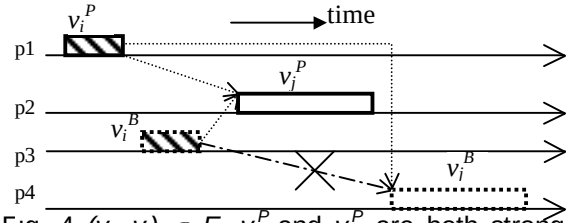

Fig. 4 $(v_i, v_j) \in E$, v_i^P and v_j^P are both strong primary copies, and v_i^P and v_j^P are scheduled on two different processors. v_i^B is not execution-preceding v_j^B.

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. 3 illustrates this scenario.

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. Thus, $\forall v_i, v_j \in V$, $(v_i, v_j) \in E$: $\sim(v_i^B \Rightarrow v_j^P) \wedge v_i^P$ is a strong primary copy $\rightarrow p(v_i^P) \neq p(v_j^B)$.

Proof: See Appendix. □

Since the proposed primary/backup model tolerates single-processor failures only, we assume that

only one processor in the system will encounter a permanent failure. Based on this assumption, we observe that if task v_i is a predecessor of task v_j, and the primary copies of both tasks are strong primary copies, then the backup copy of v_i is not execution-preceding the backup copy of v_j. Fig. 4 illustrates a scenario of the case, which is presented formally in the theorem below.

Theorem 3. Given two tasks v_i and v_j, v_i is a predecessor of v_j. If v_i^P and v_j^P are both strong primary copies, and $p(v_i^P) \neq p(v_j^P)$, then v_i^B is not execution-preceding v_j^B. In other words, $\forall v_i, v_j \in V$, $(v_i, v_j) \in E$: (v_i^P *and* v_j^P *are two strong primary copies)* $\wedge\, p(v_i^P) \neq p(v_j^P) \rightarrow \sim(v_i^B \mapsto v_j^B)$.

Proof: See Appendix. □

Theorem 3 suggests that if task v_i is a predecessor of task v_j, and the primary copies of both tasks are strong primary copies, then the backup copy of v_i^B does not need to send messages to v_j^B.

The notion of strong primary copy appears in Theorems 1, 2, and 3, it is therefore necessary to be able to determine whether a task has a strong primary copy. It is straightforward to prove that a task without any predecessor has a strong primary copy. Base on this fact, Theorem 4, below, suggests an approach to determining whether a task with predecessors 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 schedule-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: See Appendix. □

4 The Scheduling Algorithm

The propositions, theorems and corollaries provided in section 3 help us establish important qualitative relationships among primary and backup copies of tasks, as well as among predecessors and descendents of tasks. In this section, we will use these relationships to determine necessary quantitative relationships in order to calculate essential values for the proposed algorithm. These values include the earliest available time and the earliest start time of primary and backup copies, the earliest available time and the earliest start time of messages.

4.1 Notations

To help present the algorithm clearly, we define the following notations.

EAT*$_i$*(v, v*$_j$*) denotes the earliest available time for the primary/backup copy of task v if message e sent from $v_j \in D(v)$ represents the only precedence constraint.

The primary copy of a task v must receive messages from all primary copies of the tasks in set $D(v)$. Hence, the earliest available time of v^P on processor p_i is determined by the primary copies of its predecessors and messages sent from $D(v)$.

EAT*$_i^P$*(v) denotes the maximum (or the latest) of $EAT_i(v^P, v_j^P)$ at processor p_i, for all messages sent from the primary copies of the tasks in $D(v)$. Thus,

$$EAT_i^P(v) = MAX_{v_j \in D(v)}\{EAT_i(v^P, v_j^P)\} \quad (2)$$

Suppose we have a pair of tasks v_i and v_j, $(v_i, v_j) \in E$. Based on Theorem 3, we know that if v_i^P and v_j^P are both strong primary copies and $p(v_i^P) \neq p(v_j^P)$, then v_i^B is not execution-preceding v_j^B.

U(v) $\subset$ ***V*** denotes the set of *v*'s predecessor tasks allocated on different processors and with strong primary copies, namely, $U(v) = \{v_i \mid \{ (v_i, v) \in E, v_i^P$ and v^P are both strong primary copies and $p(v_i^P) \neq p(v^P)\}$. In other words, the backup copy of $v_i \in U(v)$ does not need to send message to the backup copy of *v*, based on Theorem 3.

B(v) $\subset$ ***V*** denotes the set of *v*'s predecessor tasks that are not in *U(v)*, namely, $B(v) = D(v) - U(v)$. Thus, $B(v) = \{ (v_i, v) \in E \mid v_i^P$ *is not a strong primary copy* $\vee$ v^P *is not a strong primary copy* $\vee$ $p(v_i^P) = p(v^P)\}$. This means that the backup copy of $v_i \in B(v)$ must send message to the backup copy of *v*, based on Theorem 3. It is straightforward to note that if v^P is not a strong primary copy, all *v*'s predecessor tasks are in *B(v)*. In other words, $B(v) = D(v)$.

According to the proposed scheduling algorithm, the primary copy of a task is allocated before its corresponding backup copy is scheduled. Therefore, given two tasks *v* and $v_i \in D(v)$, the primary and backup copies of v_i should have been allocated when the algorithm starts scheduling v^B. Obviously, v^B must receive message from v_i^P. In addition, v^B also needs to receive message from v_i^B, for $v_i \in B(v)$. Therefore, the maximum earliest available time of v^B on processor p_i is determined by the primary copies of its predecessors, the backup copies of tasks in *B(v)* and messages sent from these tasks. The maximum earliest available time of v^B is given below.

EAT$_i^B$(v) denotes the maximum (or the latest) of earliest available time on processor p_i, where $i \neq p(v^P)$. It is determined as,

$$EAT_i^B(v) = MAX\{ f(v^P) + \delta,\ MAX_{v_j \in D(v)}(EAT_i(v^B, v_j^P)), MAX_{v_j \in B(v)}(EAT_i(v^B, v_j^B))\}$$

$$= MAX_{v_j \in D(v), v_k \in B(v)}\{ f(v^p) + \delta,\ EAT_i(v^B, v_j^P), EAT_i(v^B, v_j^B)\} \quad (3)$$

VQ$_i$ = { v_1, v_2, …, v_q } is the task queue of which all tasks are scheduled to processor p_i. The subscripts indicate the increasing order in time in which these tasks are scheduled on the processor.

EST$_i^P$(v) and ***EST^P(v)*** represent, respectively, the earliest start time for the primary copy of *v* on processor p_i, and the earliest start time for the primary copy of task *v* on any processor. Thus,

$$EST^P(v) = MIN_{p_i \in P}\{ EST_i^P(v)\} \quad (4)$$

EST$_i^B$(v) and ***EST^B(v),*** similarly represent, respectively, the earliest start time for the backup copy of *v* on processor p_i, and the earliest start time for the backup copy of task *v* on any processor. That is,

$$EST^B(v) = MIN_{p_i \in F(v)}\{ EST_i^B(v)\} \quad (5)$$

In (5), *F(v)* denotes a set of feasible processors to which the backup copy of *v* can be allocated. Proposition 2 suggests that $p(v^p) \notin F(v)$. Set *F(v)* is easily determined based on both Proposition 2 and Theorem 2. The primary copy of a task *v* can be scheduled to a processor p_i if processor p_i has an idle time slot, defined to be the idle time period between any two consecutively scheduled tasks on p_i , that satisfies the following two conditions: (a) it starts later than v^P's earliest available time $EAT_i^P(v)$, (b) it is large enough to accommodate the task's execution.

Similarly, the backup copy of a task *v* can be scheduled to a processor p_i if processor p_i has an idle time slot that satisfies the following conditions: (a) it starts later than v^B's earliest available time $EAT_i^B(v)$, (b) it is large enough to accommodate the task's execution.

$EPT_i^P(v)$ and $EPT_i^B(v)$ denote, respectively, two sets of scheduled tasks whose separating time gaps constitute all idle time slots that meet the conditions for scheduling primary and backup copies of tasks. Thus, $EPT_i^P(v) = \{v_k \in VQ_i \mid s(v_{k+1}) - MAX[f(v_k), EAT_i^P(v)] \geq c^i(v)\}$ (6)

$$EPT_i^B(v) = \{v_k \in VQ_i \mid s(v_{k+1}) - MAX[f(v_k), EAT_i^B(v)] \geq c^i(v)\} \quad (7)$$

The earliest start time for the primary and backup copies of *v* on processor p_i are determined by the following expressions.

$$EST_i^P(v) = MAX\{EAT_i^P(v), MIN_{v_k \in EPT^P{}_i(v)}(f(v_k))\} \quad (8)$$

$$EST_i^B(v) = MAX\{EAT_i^B(v), MIN_{v_k \in EPT^B{}_i(v)}(f(v_k))\} \quad (9)$$

In equations (8) and (9), both $EAT_i^P(v)$ and $EAT_i^B(v)$ can be derived from $EAT_i(v, v_j)$ using expressions (2) and (3). $EAT_i(v, v_j)$ is determined by task v_j 's finish time and the arrival time of message $e = (v_j, v) \in E$. If v_j and v are allocated to the same processor, the communication time of message e is assumed to be zero. If v_j and v are allocated to two different processors, the message arrival time will be determined by the finish time of the parent task and the volume of data being transmitted. Thus, we have the following expressions for message start time and message arrival time.

Given a message $e= (v_j, v)\in E$, $\boldsymbol{MST_{ki}(e)}$ and $\boldsymbol{MAT_{ki}(e)}$ represent the message start time and the message arrival time, respectively, where

$$MAT_i(e) = \begin{cases} MST_{ki}(v_j,v) + w_{ij}\,|(v_j,v)| & \text{if } p(v_j) = k \neq i \\ f(v_j) & \text{otherwise} \end{cases} \quad (10)$$

A message e can be scheduled to the communication channel between two processors if the channel has an idle time slot that: (a) starts later than the sender's finish time, and (b) is large enough to accommodate e's transmission.

Given a message $e = (v_a,v_b)$, $\boldsymbol{MPT_{ij}(e)}$ is a set of messages between processor p_i and p_j $(i \neq j)$ whose separating time gaps constitute all the idle time slots that satisfy conditions (a) and (b) above. In other words, $e_k \in MPT_{ij}(e)$, then the time interval $ms(e_{k+1}) - mf(e_k)$ is an idle slot time satisfying the two conditions. Thus, $MPT_{ij}(e)=\{e_k \in MQ_j |\ MST(e_{k+1}) - MAX[f(v_a), MAT_j(e_k)] \geq |e|\}$ (11)

Then, $MST_{ij}(e)$ can be defined as follows, $MST_{ij}(e) = MAX\{f(v_a), MIN_{e_k \in MPT_{ij}(e)}(mf_{ij}(e_k))\}$ (12)

The objectives of the algorithm, presented below, are to minimize the schedule length (and thus maximize guarantee ratio) and maximize the reliability. For each primary copy v_i^P, it is allocated on the processor with the minimum reliability cost. However, if the schedule length Lp for primary copies on a processor is too long, it may result in the possibility that some backup copies can not meet their deadlines.

For each backup copy v_i^B, it should satisfy four conditions, namely, (1) it is allocated on the processor that is different than the one assigned for its primary copy, (2) it is allocated on the processor that leads to the minimum increase in reliability cost, (3) v_i^B can finish before deadline, and (4) v_i^b can receive messages from all its predecessors.

4.2 Reliability Cost Driven Scheduling

Reliability cost driven (RCD) schemes were studied in [22], [23], [28] and [29] to maximize system reliability. It must be noted that the tasks considered in [23] and [28] do not have any precedence constraints. In [29] and [22], RCD-based scheduling algorithms for tasks with precedence constraints are proposed for non-real-time and real-time, respectively. However, algorithms in [22] and [29] do not take fault-tolerance into account, neither do algorithms in [23] and [28]. We have adapted the basic idea of RCD scheme to accommodate these differences. The objectives of the algorithm are twofold: to minimize the schedule length, thus maximizing the schedulability, and to maximize the reliability by incorporating reliability cost and fault-tolerance in the scheduling. The basic idea of the proposed algorithm follows below.

First, real-time tasks are sorted in non-decreasing order of their deadlines, and are processed for scheduling in that order. Hence, the tasks with earlier deadline will be given a higher priority to be scheduled. The primary copies of tasks are allocated first, followed by their backup copies. For each given copy, primary or backup, of a task, the candidate processor for allocation is chosen following a

greedy process, thus resulting in the processor that leads to the smallest reliability cost being chosen. In the case that two processors lead to the same smallest reliability cost, selecting the processor that gives rise to the earlier EST breaks the tie. The fault-tolerant RCD scheduling algorithm is formally described below. An example to illustrate how the proposed algorithm works is given in Appendix.

FTRCD Algorithm:

1. **if** $\exists v$: $2MIN_{1\le i\le m}\{c_i(v)\} > d(v)$ **then return(**FAIL**);** /* Pre-check the deadlines based on proposition 2 */
2. **Sort tasks in *V* by their deadlines in non-decreasing order, subject to precedence constraints of the DAG, and put them into an ordered list *OL*;**
3. **for each task *v* in *OL* do** /* Schedule primary copies */
4. $s(v^P) \leftarrow \infty$; *find* $\leftarrow$ *NO*; $rc \leftarrow \infty$; /* Initialize start time, reliability cost (rc) for v^P */
5. **for each processor p_i do** /* Determine reliability cost of v^P on each processor */
6. $rc_i \leftarrow \lambda_i \times c_i(v)$; /* Calculate the reliability cost of v^P on p_i */
7. **if** $(EST^P_i(v) + c_i(v) \le d(v))$ **then** /* Calculate the *EST* of v^P on p_i using Equation (8), and check deadline */
8. **find** $\leftarrow$ **YES;** /* Indicate that there is a possible schedule */
9. **if** $((rc_i<rc)$ **or** $(rc_i=rc$ **and** $EST^P_i(v)< s(v^P)))$ /* Find the minimum reliability cost */
10. **then** $s(v^P) \leftarrow EST^P_i(v)$; $p \leftarrow p_i$; $rc \leftarrow rc_i$; /* Assign start time and rc */
11. **end if**
12. **end for**
13. **if find = NO then return** (FA IL) /* If no possible schedule is available for v^P, algorithm terminates */
14. $p(v^P) \leftarrow p$; $f(v^P) \leftarrow s(v^P) + c_i(v)$, **where** $p_i = p$; /* Determine the finish time */
15. **Dispatch** v^P **to processor** $p(v^P)$;
16. **Update information of message in the *MQ*;**
17. **end for** /* Finish scheduling the primary copies */
18. **for each task *v* in *OL* do** /* Schedule the backup copies */
19. $s(v^B) \leftarrow \infty$; *find* $\leftarrow$ *NO*; $rc \leftarrow \infty$; /* Initialize start time, reliability cost for v^P */
20. **for each possible processor $p_i \in F(v)$, subject to Proposition 2 and Theorem 2, do**
21. **Determine whether v^P is a strong primary copy (using Theorem 4)**;
22. $c_i \leftarrow \lambda_i \times c_i(v)$; /* Calculate the reliability cost of v^P on p_i */
23. **if** $(EST^B_i(v) + c_i(v) \le d(v))$ **then** /* Calculate the *EST* of v^B on p_i using Equation (9), and check deadline */
24. **find** $\leftarrow$ **YES;** /* Indicate that there is a possible schedule */
25. **if** $((rc_i<rc)$ **or** $(rc_i=rc$ **and** $EST^B_i(v)< s(v^B)))$ **then** /* Find the minimum reliability cost */
26. $s(v^B) \leftarrow EST^B_i(v)$; $p \leftarrow p_i$; $rc \leftarrow rc_i$; /* Assign start time and rc */
27. **end if**
28. **end for**
29. **if find = NO then return** (FAIL); /* If no possible schedule is available, algorithm terminates */
30. $p(v^B) \leftarrow p$; $f(v^B) \leftarrow s(v^B) + c_i(v)$, **where** $p_i = p$; /* Determine the finish time */
31. **Dispatch** v^B **to processor** $p(v^B)$;
32. **Update information of each message in *MQ*;**
33. **for each task** $v_j \in U(v)$ **(using Theorem 3) do** /* Determine messages based on $U(v)$ obtained by Theorem 3 */
34. **if** $p(v^P) \neq p(v_j^P)$ **or** v^P **is not a strong primary copy then**
35. v_j^B **sends message to** v^P **if possible (based on Theorem 1);**
36. **end for** /* Finish scheduling the backup copies */
37. **return** (SUCCEED);

5. Performance Evaluation

This section presents results from our extensive simulation experiments that evaluate the performance of the proposed algorithm. For the purpose of comparison, we modified the FTRCD algorithm to form a non-reliability-driven algorithm, called FTAEAP (*f*ault-*t*olerant, schedule *a*s *e*arly *a*s *p*ossible), by eliminating the consideration of reliability cost in scheduling. More specifically, we change lines 8 and 24 in the FTRCD algorithm so that the *EST* values now become the only criteria

for selecting candidate processors. Therefore, the FTAEAP algorithm, greedy in nature, tolerates permanent failures in any one processor, but does not consider reliability cost.

We run the two algorithms on a variety of task graphs that are generated randomly. Since binary tree and lattice are two typical DAGs, representative of many real-life parallel and real-time programs, we consider them in the simulations. In addition, DAGs with random precedence constraints are also considered to stress-test the proposed algorithm. All these DAGs are randomly generated. Random graphs have been widely used by several researchers in the past [22][28][29].

In addition to the DAGS, input parameters for the simulation include: (1) number of real-time tasks in each DAG, (2) number of processors and their failure rates, (3) fault detection time, (4) a randomly generated computational heterogeneity vectors $C(v) = \{c_1(v), c_2(v), \ldots, c_m(v)\}$ for each real-time task, and (5) communicational heterogeneity vectors. Together, they represent the system model and workload precisely.

The performance measures used in this study are: *reliability cost (RC)*, defined in Equation (1), and schedule *guarantee ratio (GR)*, defined to be the percentage of real-time jobs (DAGs) that are schedulable. While the former gives a measure of system reliability as a result of a particular schedule, the latter quantifies the effectiveness and scheduling power of a scheduling algorithm. For each of the two scheduling algorithms, it is invoked with the above inputs and, if the task set is schedulable, then $RC_0(\Omega)$ is calculated based on the resultant schedule.

In this simulation study, 100,000 task sets were generated independently for scheduling algorithm. Workload parameters are chosen in such a way that they are either based on those used in the literature or represent reasonably realistic workload and provide some stress tests for the algorithms.

Table 1 Parameters for simulation experiments

Parameter	Explanation	Value (Fixed)	(Varied)
FR	Failure rate of processors	-	
C	Range of execution time	5 - 50	5-50,5-100,5-120
D	Range for generating the deadline	1- 10	1-100,1-300,…, 1-1100
W	Range of communication weight	0.5 - 1.5	
V	Range of communication volume	1 - 10	
M	Number of processor	8	7, 8
N	Number of tasks in a DAG	Btree/Random 10,20,…,70 Lattice: 9,16,…,81	
δ	Fault Detection time	5	5, 10 , 15, …,35

The parameters used in the simulation studies are summarized in Table 1 [22][29]. Real-time DAGs for the simulation are generated as follows:

(1) For each real-time task, the computation time in the execution time vector is randomly chosen, uniformly from the range *E*. The scale of this range approximates the level of computational heterogeneity.

(2) 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 chosen as follows: $d(v_i) = md*\{max(d(v_j) + 1 + |e_{ij}| \times w_{lk} + max\{c^k(v_i)\} + \delta\}$, where $e_{ij} = (v_j, v_i) \in E(J)$, $k \in [1, m]$, *md* is a factor for defining deadlines. δ is the fault detection time that is randomly computed according to a uniform distribution,. When *md* is set to a high value, real-time tasks will have loose deadlines, whereas *md* with a low value results in tight deadlines for tasks.

Data communication among real-time tasks is simulated as follows:

(1) Communication weight (w_{ij}) is chosen uniformly from the range *W*. The scale of this range approximates the level of communicational heterogeneity.

(2) Communication volume between two real-time tasks is uniformly selected from the range *V*. This range reflects the variance in message size.

A random DAG is generated in four steps as below:

(1) The number of tasks *N* and the number of messages *U* are chosen. In this simulation study, it is assumed that $U = 4N$. (2) The execution time for each task is chosen randomly. (3) The

communication time for each message is generated randomly and its sender and receiver selected randomly, subject to the condition that such selection does not generate any circle in the graph. (4) A deadline for each task is generated randomly.

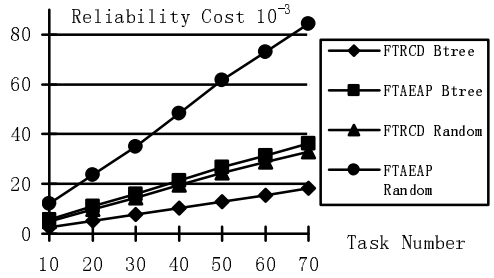

Fig. 5 Reliability costs of FTRCD and FTAEAP. The C range for Btree and for Random Graph is [5,50] and [5,120]

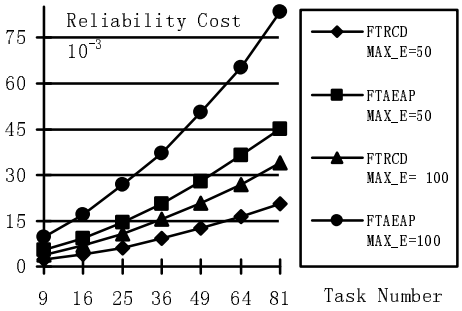

Fig. 6 Reliability costs of FTRCD and FTAEAP. Task graph is lattice

5.1 Reliability Cost

RC is used as one of the objective functions for scheduling tasks. *RC* results of the simulation running FTAEAP and FTRCD are presented in this subsection. Since the purpose of this experiment is to study the impact of *RC*, schedulability (i.e. *GR*) is not considered here. Therefore, deadlines of real-time tasks are assumed to be so large that all tasks' deadlines can be guaranteed. To satisfy this assumption, the *md* factor is set to be arbitrarily large. The number of processors is set to 8, and maximum execution time is chosen to be 50 and 100, respectively, other parameters are selected as shown in Table 1.

RC is computed according to Equation (1). Simulation results of binary trees and random DAGs are shown in Fig. 5, and results of lattice DAGs are plotted in Fig. 6. We observe that FTRCD outperforms FTAEAP consistently and significantly in the *RC* measure. As the number of tasks increases, the advantage of FTRCD over FTAEAP becomes more significant. This is because FTAEAP does not consider *RC* in its scheduling scheme while FTRCD makes efforts to allocate tasks to processors on which their execution times are minimum, since it is the execution time that dominates the *RC* product.

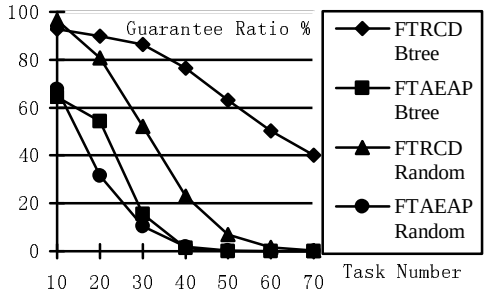

Fig. 7 Guarantee ratio of FTRCD and FTAEAP for Btree and random graph

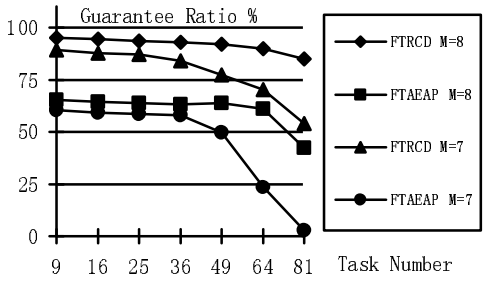

Fig. 8 Guarantee ratio of FTRCD and FTAEAP. Task graph is lattice

5.2 Schedulability

The guarantee ratio, formally defined to be: *GR = Number of task sets that are schedulable * 100 / Total task sets*, is used in this subsection to measure the scheduling power of the proposed algorithm. In this experiment, the maximum execution time is fixed to 50, the number of processors is selected to be 8 and 10 for binary trees, and is chosen as 7 and 8 for lattices. Other parameters are chosen as shown in Table 1.

Fig. 7 and Fig. 8 illustrate the impact of the job size on *GR* for binary trees, random graphs and lattices, respectively. As the number of tasks in a job increases, *GR* decreases. This is intuitive, because under a fixed computing resource, the more tasks there are in the system, the fewer of them can be guaranteed to satisfy their deadlines. It is very interesting that this impact is more significant for binary trees than for both random graphs and lattices. We conclude from this result that the schedulability of the system does not only depend on the scheduling algorithm, but also on the structure of the DAG. Again, FTRCD gives rise to a relatively higher guarantee ratio compared with that of the FTAEAP. This is because FTRCD tends to allocate tasks to processors that would execute tasks the fastest, or equivalently, fastest processors tend to be picked.

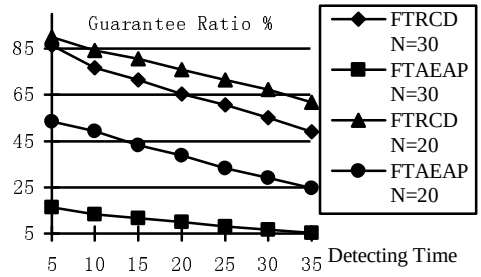

Fig. 9 Impact of detection time on guarantee ratio.Task graph is binary tree

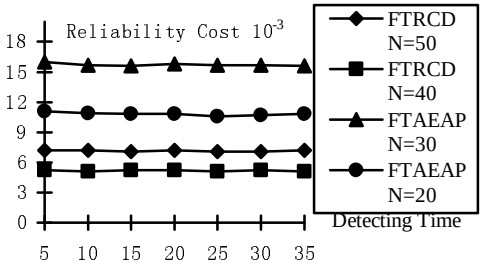

Fig. 10 Impact of detecting time on reliability cost.Task graph is binary tree

5.3 Impact of The Fault Detection Time

In the simulation experiment, the fault detection time is increased from 5 to 35 in steps of 5. Other parameters assume the default values given in Table 1. To make the results more discernable, the *N* value is chosen for the two algorithms in such a way that performance curves are evenly spread in the plots while the proposed algorithm (FTRCD) is stress-tested. Thus, for algorithm FTRCD, *N* is set to be 50 and 40, respectively, whereas for algorithm FTAEAP, *N* is chosen to be 20 and 30. Notice that FTRCD is loaded almost twice as much as FTAEAP. In this experiment, we only present the result for binary tree since other DAGs behave similarly. Figs. 9 and 10 illustrate the impact of fault detection time on *GR* and *RC*, respectively.

Fig. 9 shows the impact of the fault detection time on *GR* for different values of *N*. As fault detection time increases, *GR* decreases significantly. It clearly suggests that decreasing fault detection time can significantly improve *GR* of the system. This result strongly supports the intuition that using a high-speed network to detect fault in a short amount of time can substantially enhance the system performance. From Fig. 10, we observe that *RC* remains roughly the same with increasing detection time. It is clear that fault detection time has no significant impact on *RC*. This is because a larger fault detection time only delays the earliest start time of the tasks. This means that fault detection time does not play any role in local *RC* calculation, which in turn determines to which processor a task should be allocated.

6 Conclusion

In this paper, we present a fault-tolerant scheduling algorithm for a heterogeneous distributed system executing real-time tasks with precedence constraints. To provide the system with fault-tolerance, each task is associated with a primary and backup copy so that, if the primary copy of the task does not work due to the permanent failure of one processor, its backup copy can continue the task's execution. In the proposed approach, it is required that both primary and backup copies be scheduled to complete before their deadlines. In addition, conditions under which the primary and

backup copies of a task should send message to the primary and backup copies of its successors are investigated.

When a processor fails, it takes an extra amount of time to handle fault and fault detection. Nevertheless, most related research work in the literature assumes that fault detection time is zero. The proposed approach, however, takes fault detection time into account, and studies how fault detection time affects scheduling performance quantitatively. The experimental result suggests that shortening fault detection time improves the schedulability of the algorithm.

Another contribution of this paper is that the proposed algorithm is devised for heterogenous distributed systems that are increasingly being used for real-time applications. We adopt the notion of reliability cost, which is a good metric to evaluate the reliability of a heterogenous system. The proposed algorithm employs a reliability-driven scheme to minimize the reliability cost without any extra hardware cost. To evaluate the performance of the reliability-driven, fault-tolerant scheme, we compare the proposed algorithm with one of its variations that does not consider reliability cost when scheduling tasks. The result shows that the reliability-driven algorithm not only generates the schedule with a minimized reliability cost, but also has superior schedulability compared with non-reliability-driven algorithms.

As an extension of this work, it would be interesting to study more DAG types that represent more applications. We are planning to find out whether DAG types make significant impact on the performance of the algorithm. Devising a dynamic fault-tolerant scheduling algorithm based on FTRCD algorithm will also be a challenging task. In the proposed scheduling model, it is assumed that backup copies are not allowed to be overlapped with one another. One possible extension of the current work would be to study the possibility of allowing backup copies to be overlapped in some fashion.

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] K. Ahn, J. Kim, S. Hong, "Fault-Tolerant Real-Time Scheduling using Passive Replicas," In *Proc. Of the 1997 Pacific Rim International Symposium on Fault-Tolerant Systems*, Taipei, TAIWAN, December 15-16, 1997.

[3] N.M. Amato, P. An, "Task Scheduling and Parallel Mesh-Sweeps in Transport Computations," *Technical Report TR00-009*, Department of Computer Science, Texas A&M University, January 2000.

[4] C. Dima, A. Girault, C. Lavarenne, and Y. Sorel, "Off-Line Real-Time Fault-Tolerant Scheduling," In *Proc. Of the Euromicro Workshop on Parallel and Distributed Processing*, pp. 410—417, Mantova, Italy, February, 2001.

[5] 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.

[6] D.G. Feitelson, "Scheduling Parallel Jobs on Clusters," In *High Performance Cluster Computing, Vol 1: Architectures and Systems*, pp. 519-533, Prentice-Hall, 1999.

[7] D.G. Feitelson and A. W. Mu'alem, "On the Definition of ``On-Line'' in Job Scheduling Problems," Technical Report 2000-32, Hebrew University of Jerusalem, Mar 2000.

[8]R. Govindrarajan, Narasimba Rao, E.R. Altman and Guang R. Gao, An Enhanced Co-Scheduling Method using Reduced MS-State Diagrams, In the Proceedings of the International Parallel Processing Symposium, pp 168-175, Orlando, Florida, April 1998.

[9] K. Hashimoto, T. Tsuchiya, and T. Kikuno, "A New Approach to Realizing Fault-Tolerant Multiprocessor Scheduling by Exploiting Implicit Redundancy," In *Proc. Of the 27th Intl Symposium on Fault-Tolerant Computing*, Seattle, WA, June 25-27, 1997.

[10] 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.

[11] W. Jia, Wei Zhao, D. Xuan, and G. Xu, "An Efficient Fault-Tolerant Multicast Routing Protocol with Core-Based Tree Techniques," *IEEE Transactions on Parallel and Distributed Systems*, 10(10), pp 984 – 1000, Oct. 1999.

[12] D. Kebbal, E.G Talbi, and J.M Geib, "Building and scheduling parallel adaptive applications in heterogeneous environments," *In Proc. Of the 1st IEEE Computer Society International Workshop on Cluster Computing*, pp.195-201, Melbourne, Australia, 1999.

[13] Y.K. Kwok and I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors ACM Computing Surveys Vol.31 , No. 4, 1999, pp.406-471.

[14] F. Liberato, S. Lauzac, R. Melhem and D. Mosse, "Fault Tolerant Real-Time Global Scheduling on Multiprocessors," *Proc.of Euromicro Workshop in Real-Time Systems*, 1999.
[15] 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.
[16] 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.
[17] P. Mejia Alvarez and D. Mosse, "A Responsiveness Approach for Scheduling Fault Recovery in Real-Time Systems," *Proceedings of Fifth IEEE Real-Time Technology and Applications Symposium*, Canada, pp.1-10, June 1999.
[18] D. Mosse, R. Melhem and S.Ghosh, "Analysis of a Fault-Tolerant Multiprocessor Scheduling Algorithm," In Proc. Of the 24th Fault-tolerant Computing Symposium. Austin, TX, 1994.
[19] 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
[20] 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.
[21] Xiao Qin, and Hong 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.
[22] Xiao Qin, Hong 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.
[23] Xiao Qin, Z.F. Han, H. Jin, L.P. Pang and S.L. Li., "Real-time Fault-tolerant Scheduling in Heterogeneous Distributed Systems," *in Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA, June 26-29, 2000. Vol. I, pp.421-427.
[24] S. Ranaweera, D.P. Agrawal, "A scalable task duplication based scheduling algorithm for heterogeneous systems,"*In Proc. Of the 29th International Conference on Parallel Processing*, pp. 383 –390, 2000.
[25] 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.
[26] S. Ranaweera and D.P. Agrawal, "A Task Duplication Based Scheduling Algorithm for Heterogeneous Systems," *in 14th International Parallel and Distributed Processing Symposium* (IPDPS'00), 1 – 5 May 2000, Cancun, Mexico.
[27] R.M. Santos, J. Santos, and J. Orozco, "Scheduling heterogeneous multimedia servers: different QoS for hard, soft and non real-time clients," *In Proc. Of the 12th Euromicro Conference on Real-Time Systems*, pp.247-253, 2000.
[28]S.M. Shatz, J.P. Wang, and M. Goto, "Task Allocation for Maximizing Reliability of Distributed Computer Systems," *IEEE Trans. Computers*, vol.41, no.9, pp.1156-1168, Sept. 1992.
[29] 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.
[30] J. Stankovic, M. Spuri, K. Ramamritham and G.C. Buttazzo, "*Deadline Scheduling for Real-time systems: EDF and Related Algorithms,"* Kluwer Academic Publishers, 1998

Appendix.

<u>Please Note:</u> The proofs of the theorems and the scheduling example used in this paper are provided in this appendix for the convenience of the reviewers. We understand that the conference has a page limit and do not intend to publish all the material presented here in te proceedings if the paper is accepted. The materials will, however, be made available as a technical report.

A. Proof of Theorem 1.

Proof: By contradiction: Assume v_i^B *is execution-preceding* v_j^P, thus, v_j^P must execute (Def.5). Since v_i^P is a strong primary copy, processor $p(v_i^P)$ must have failures by time $f(v_i^P)$ (Def.7). 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. □

B. Proof of Theorem 2.

Proof: 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 $p(v_i^P)$ does not work at time $f(v_i^P)$ (Def. 7). Since $f(v_i^P) < f(v_j^B)$, it also

implies that $p(v_i^P)$ does not work at time $f(v_j^B)$. Thus, we have $\sim@(p(v_i^P), f(v_j^B))$. Since v_i^B executes, either v_i^B is execution-preceding v_j^P or v_i^B is execution-preceding v_j^B. But $\sim(v_i^B \Rightarrow v_j^P)$, v_i^B cannot be execution-preceding v_j^P. Hence, we only have $v_i^B \mapsto v_j^B$. This implies that v_j^B executes (Def. 7), which means $@(p(v_j^B), f(v_j^B))$, a contradiction. Thus, we have proven that $p(v_j^B) \neq p(v_i^B)$. □

C. Proof of Theorem 3.

Proof: By contradiction: Suppose $v_i^B \mapsto v_j^B$. We have Θv_i^B and Θv_j^B (Def. 5), which implies $\sim@(p(v_i^P), f(v_i^P))$ (Def.7) and $\sim@(p(v_j^P), f(v_j^P))$ (Def.7). Since v_i^P is schedule-preceding v_j^P, thus $f(v_i^P) < f(v_j^P)$, implying $\sim@(p(v_i^P), f(v_j^P))$. This means that at time $t > f(v_j^P)$, two processors in the system have failed. A contradiction. □

D. Proof of Theorem 4.

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, we have $@(p(v_i^P), f(v_i^P))$. 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. In other words, v_i^P must execute since $p(v_j^P)$ has not failed by time $f(v_i^P)$. Therefore, according to Definition 7, v_i^P is a strong primary copy. □

E. Scheduling Example

We use an example to illustrate how the proposed algorithm, FTRCD, works, by scheduling the task graph (DAG) of Fig. 11 on a three-processor system. The scheduling result is depicted in Fig. 12.

After sorting tasks by their deadlines in non-decreasing order, subject to their precedence constraints, the task sequence in the list *OL* is, $OL = (v_1, v_3, v_2, v_5, v_4, v_6)$. From step 3 to 17, primary copies of the six tasks are scheduled. For each primary copy, its reliability costs on three processors are calculated (see step 5-12), and the processor which gives rise to the minimum reliability cost is determined in step 9 and 10. For example, the primary copy of v_1 is allocated to p_2, because the reliability cost of v_1 on p_2 is the smallest among the three processors, namely, 0.95×10^{-6}, compared to 2×10^{-5} and 1.05×10^{-5} on p_1 and p_3, respectively. Step 13 terminates the algorithm if one primary copy's deadline cannot be guaranteed. The six backup copies are scheduled following steps 18 through 35 of the algorithm. *RCs* of each backup copy on all feasible candidate processors are generated following steps 20 through 28. All candidate processors for the backup copy of v are stored in $F(v)$. For example, $F(v_2) = \{p_3\}$. In this particular example, p_3 happens to be the only suitable candidate for v_2. This is because, on the one hand, the primary and backup copies of v_2 are allocated to different processors, due to Proposition 2. Hence, $p(v_2^P) = p_1 \notin F(v_2)$. On the other hand, v_1 has a strong primary copy and v_1^B is not schedule-preceding v_2^P, which, according to Theorem 2, implies that v_1^P and v_2^B cannot be allocated to the same processor, thus $p(v_1^P)=p_2 \notin F(v_2)$.

Before calculating *RCs* of each backup copy, step 20 determines whether a task has a primary copy based on Theorem 4. For example, v_1^P is a strong primary copy (Theorem 4(a)), and v_2^P is not a strong primary copy due to the fact that $p(v_1^P) \neq p(v_2^P) \wedge \sim(v_2^B \Rightarrow v_1^P)$ (Theorem 4(b)). Like step 13, step 29 terminates the algorithm if one backup copy's deadline cannot be satisfied. The last step after

scheduling each backup copy of task v is to schedule messages from v_j^B ($v_j \in U(v)$) to v^P, if applicable, based on Theorem 1 (See step 33-35).

Finally, the reliability cost and schedule length generated by the FTRCD algorithm are 1.11×10^{-3} and 87, respectively.

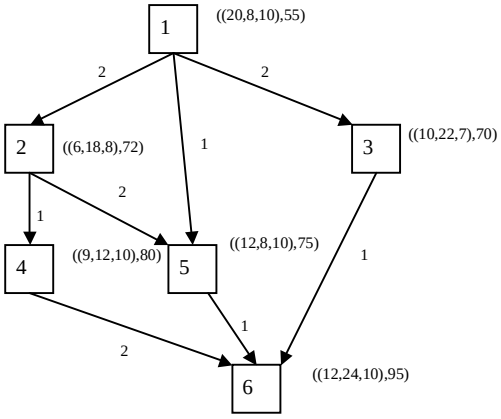

Fig. 11 DAG task graph. Assume a 3-processor system and each real-time task is denoted by $v_i = ((c_{i1}, c_{i2}, c_{i3},), d_i)$, where c_{ij} is the execution time of v_i on p_j, and d_i is the deadline. i = [1, 6], j = [1, 3]. Communication weights are: $w_{12} = w_{21} = 1$, $w_{13} = w_{31} = 3$, $w_{23} = w_{32} = 3$.

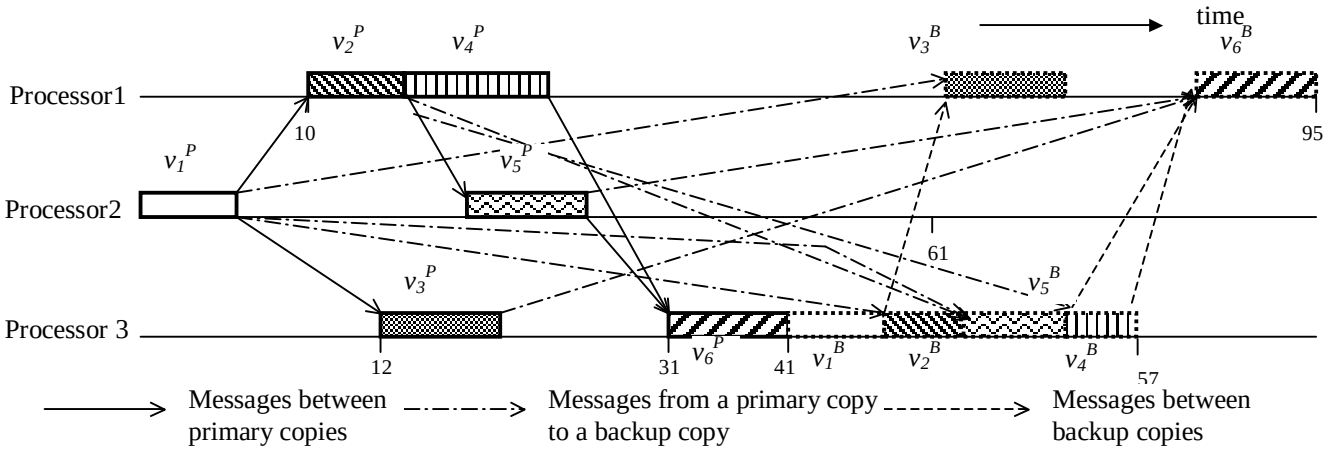

Fig.12 Schedule Produced by the FTRCD algorithm. Reliability cost = 1.1×10^{-4}, schedule length = 95 $\lambda_1 = 1*10^{-6}$, $\lambda_2=0.95*10^{-6}$, $\lambda_3 = 1.05*10^{-6}$