Scheduling Real-Time Tasks for Dependability

Yingfeng Oh and Sang H. Son
Dept. of Computer Science
University of Virginia
Thornton Hall, Charlottesville, VA 22903, USA

Real-time systems are increasingly used in applications whose failure may result in large economic and human costs. Since many of the systems operate in environments that are non-deterministic, and even hazardous, it is extremely important that the systems must be dependable, i.e., the deadlines of tasks must be met even in the presence of certain failures. In order to enhance the dependability of a real-time system, we study the problem of scheduling a set of real-time tasks to meet their deadlines even in the presence of processor failures. We first prove that the problem of scheduling a set of non-preemptive tasks on more than two processors to tolerate one arbitrary processor failure is NP-complete even when the tasks share a common deadline. A heuristic algorithm is then proposed to solve the problem. The schedule generated by the heuristic algorithm can tolerate one arbitrary processor failure in the worst case. The analysis and experimental data show that the performance of the heuristic algorithm is near-optimal.

Keywords: combinatorial analysis, dependability, heuristics, scheduling

INTRODUCTION

Many mission-critical and life-critical applications, such as space exploration, aircraft avionics, the operation of nuclear power plants and defense systems, and robotics are not feasible without the support of computer systems. These applications require long duration of reliable operations as well as timeliness of operations. Computer systems that are built to support these applications include SIFT[1], FTMP[2], the space shuttle primary computer system[3], and MAFT[4]. These mission critical systems are mainly parallel or distributed systems that are embedded into complex (or even hazardous) environments. The most sought-after properties of these systems are *timeliness* and *dependability.* Timeliness is the system's ability to meet its specified timing constraints, and dependability is the system's ability to continue its specified operations even in the presence of hardware failures or software errors. A great deal of efforts has been invested to make computer systems highly responsive and dependable, just to cite a few, [5,6,7,8,9.]

Yet, conspicuously lacking in this scenario is an approach towards supporting timeliness (real-time) and dependability (fault-tolerance) simultaneously in a system at the level of task scheduling. Traditional approaches to provide dependability in a real-time system have been to separate the concern of the two issues, i.e., task deadlines are met by real-time scheduling, with the assumption that processors and tasks are free of failures and errors, while the dependability of the system is achieved through redundancy techniques, assuming that task deadlines can be met separately. These two assumptions have been challenged recently by several researchers[10], arguing that real-time and dependability requirements are not orthogonal. Consequently, some efforts[11] have been made to address the joint requirement of the two. However, the approaches adopted so far have either been *ad hoc* or limited to specific case studies. A formal approach which addresses the problem in a top-down or bottom-up manner is needed, because such approach is essential in building timeliness and dependability into a single computer system.

Since most cases of the general real-time scheduling problem are intractable, it is reasonable to expect that many cases of the general real-time fault-tolerant scheduling problem are also intractable. This is indeed the case, as shown by some of the results in this paper. However, this fact neither makes the problem go away nor render our approach ineffective; rather it requires that heuristics be developed where the problem instances are NP-complete. In this paper, we formulate the scheduling problem, and then show that the problem of scheduling a set of real-time tasks with a common deadline on more than two processors for the tolerance of one arbitrary processor failure is NP-complete.

Since NP-complete problems are widely believed to be computationally intractable, a heuristic algorithm is proposed to obtain an approximate solution. The schedule generated by the scheduling algorithm can tolerate, in the worst case, one arbitrary processor failure. Simulation and analysis have been carried out to evaluate the performance of the algorithm, and it is shown that the algorithm finds near-optimal solutions in most of the cases.

RELATED WORK

Though there have been several works in the literature that deal with real-time fault-tolerant scheduling issues, they study the issues under different assumptions and are only remotely related to our work. Here we only mention a few. Krishna and Shin[12] proposed a dynamic programming algorithm that ensures that backup or contingency schedules can be efficiently embedded within the original “primary” schedule to ensure that deadlines continue to be met even in the face of processor failures. Unfortunately, their algorithm has the severe drawback that it is premised on the solution to two NP-complete problems.

Balaji et al[13] presented an algorithm to dynamically distribute the workload of a failed proces-

sor to other operable processors. The tolerance of some processor failures is achieved under the condition that the task set is fixed, and enough processing power is available to execute it. In other words, the guarantee of task deadlines has been assumed beforehand. Bannister and Trivedi[14] considered the allocation of a set of periodic tasks to a number of processors so that a certain number of processor failures can be sustained. All the tasks have the same number of clones (or copies), and for each task, all its clones have the same computation time requirement. An approximation algorithm is proposed, and the ratio of the performance of the algorithm to that of the optimal algorithm, with respect to the balance of processor utilization, is shown to be bounded by $(9m) / (8 (m - r + 1))$, where m is the number of processors to be allocated, and r is the number of clones for each task. However, their allocation algorithm does not consider the problem of minimizing the number of processors used, and the problem of how to guarantee the task deadline on each processor is not addressed. These are very important considerations that our work addresses.

Oh and Son[15, 16] have investigated several special cases of the real-time fault-tolerant scheduling problem. For one special case where the backup copies of the tasks are not allowed to be overlapped, the scheduling problem was proven to be NP-complete[17]. Two heuristic scheduling algorithms were proposed to solve the problem. In this paper, we relaxes the previous requirement that backup copies of tasks are not allowed to be overlapped, and prove that even allowing backup copies of tasks to be overlapped, the scheduling problem is still NP-complete. The complexity result presented here provides the solid evidence that even for a very simple instance, the scheduling problem is NP-complete.

PROBLEM FORMULATION AND COMPLEXITY RESULT

We assume that processors fail in the fail-stop manner and the failure of a processor can be detected by other processors. The means of processor monitoring, failure detection, and failure notification are not considered here. We further assume that all tasks have hard deadlines and their deadlines must be met even in the presence of processor failures. We say that a task meets its deadline if either its primary copy or its backup copy finishes before or at the deadline. Because processor failure is unpredictable and the task deadlines are hard, no optimal dynamic scheduling algorithm exists. We therefore focus on static scheduling algorithms to ensure that task deadlines are met even in the presence of processor failures. The scheduling problem can be formally defined as follows:

A set of n tasks $\Sigma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is given to be scheduled on m processors. Each task is characterized by the tuple $\tau_i = (r_i, c_i, p_i, d_i)$, where r_i, c_i, p_i, and d_i are the release time, the computation time, the period, and the deadline of task τ_i. If p_i is specified as a variable, then the task system is termed an *aperiodic* task system. Otherwise, it is a *periodic* task system. Associated

with each task are a number of primary copies and a number of backup copies. A *k-Timely-Fault-Tolerant* (hereinafter *k*-TFT) schedule is defined as the schedule in which no task deadlines are missed, despite *k* arbitrary processor failures. Then, given a set Σ of *n* tasks, *m* processors, *the scheduling problem* (hereinafter referred to as the TFT scheduling problem) can be defined, in terms of a decision problem, as deciding whether there exists a *k*-TFT schedule for the task set Σ running on *m* processors. In reality, it is more likely that a task set Σ is given, and the scheduling goal is to find the minimum number of processors *m*, such that a *k*-TFT schedule can be constructed for the task set Σ on *m* processors. This then becomes an optimization problem. If a decision problem is NP-complete, then its corresponding optimization problem is NP-hard.

The TFT scheduling problem is a natural extension to the real-time scheduling problem. It is worth noting that tasks in most of the real-time systems are periodic[7,9], and the scheduling of periodic tasks has been the focal point of real-time scheduling theory. On the fault-tolerance aspect, hardware and software redundancy, more specifically processor and task redundancy, are incorporated into the scheduling problem. The scheduling goal is instead captured by the new parameter, *k*-TFT.

In the following, a special case of the TFT scheduling problem is considered. The tasks are assumed to be independent and non-preemptive. Each task has a primary copy and a backup copy, and the scheduling goal is to achieve 1-TFT for processor failure, i.e., the tolerance of one arbitrary processor failure. This case of the TFT problem is chosen to be studied, because it is the simplest case. Our strategy to tackle the TFT scheduling problem is to start with the simplest cases, and then walk our way towards more complicated cases.

The task redundancy scheme specified in this case actually corresponds to the primary-backup copy approach or recovery block approach. Primary-backup copy approach requires the multiple implementation of a specification[6]. The first implementation is called the primary copy, and the other implementations are called the backup copies. The primary and if necessary, the backup copies, execute in series. If the primary copy fails, one of the backup copies is switched in to perform the computation again. This process is repeated until that either correct results are produced or all the backup copies are exhausted. Here we consider a special case of the primary-backup copy approach, i.e., each task has one backup copy only. The following Lemmas guarantee that having one backup copy for each task is sufficient for the tolerance of one arbitrary processor failure. The proofs of these Lemmas can be found in [16].

Lemma 1

In order to tolerate one or more processor failures and guarantee that the deadline of a task is met using primary-backup copy approach, the computation time of the task must be less than or

equal to half of the period of the task, assuming that the deadline coincides with the period.

Lemma 2

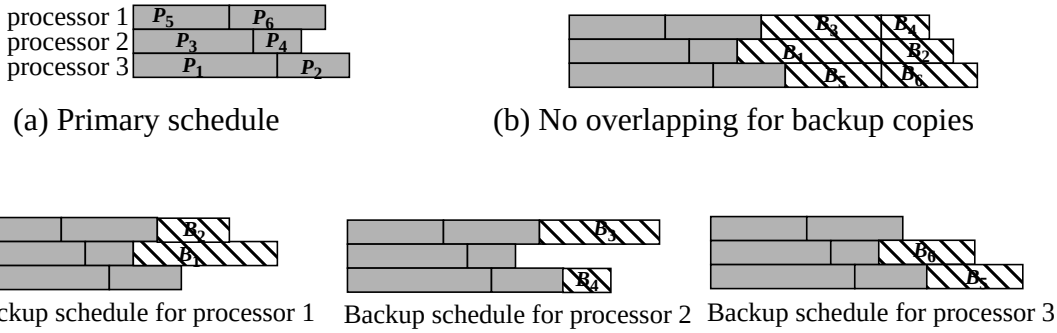
One arbitrary processor failure is tolerated and the deadlines of tasks are met, if and only if the primary copy of a task and its backup copy is scheduled on two different processors such that there is no overlapping in time between their executions, and the backup copies of the tasks whose primary copies are scheduled on a processor must not be overlapped in time for their execution on the same processor.

An obvious implication of Lemma 1 is that for each task, if the computation time of the task is larger than half of its period, it is impossible to find a schedule that is 1-TFT. This is due to the observation that if the primary copy fails at the very end, there will not be enough time left to start a backup copy and finish executing it, assuming that the backup copy has the same computation time requirement as the primary copy. This fact is used implicitly in many situations throughout the paper.

In scheduling the backup copies, we have the options of allowing them to be overlapped or forbidding them from overlapping. Here we consider the case where the backup copies are allowed to be overlapped with each other. What we mean by allowing them to be overlapped is that backup copies of the tasks whose primary copies are scheduled on different processors are allowed to be overlapped in time of their execution on a processor, since, by assumption, only one processor failure is tolerated in the worst case. However, backup copies of the tasks whose primary copies are scheduled on the same processor should not be scheduled to overlap each other in time of their execution on a processor. To illustrate this concept of overlapping, let us consider an example.

Example 1

A set of tasks is given as follows: $\Sigma = \{P_1, P_2, P_3, P_4, P_5, P_6, P_7\}$ is the set of primary copies and the backup copy B_i has the same computation time requirement as its primary P_i. The primary schedule for the task set is given in Figure 1a. For the tolerance of one arbitrary processor failure, a 1-TFT schedule is constructed in Figure 1b, where the backup copies are not allowed to overlap each other. In contrast, the schedule where the backup copies are allowed to overlap each other in time is given in Figure 1c. Let us look at the time after the primary copy P_6 has finished execution on processor 1. In the case of non-overlapping, the backup copy B_3 will be executed if processor 2 fails, while in the case of overlapping, either the backup copy B_2 or the backup copy B_3 will be executed depending on whether processor 3 or 2 fails. Intuitively the length for the schedule allowing overlapping of backup copies is shorter, but the overall schedule is more complex since multiple backup schedules must be generated for each processor.



(a) Primary schedule

(b) No overlapping for backup copies

(c) Overlapping of backup copies

FIG. 1: *Comparison of schedules (overlapping vs non-overlapping of backup copies)*

If the given number of processors is two, there apparently exists an optimal algorithm to schedule a set of tasks having a common deadline so as to tolerate one arbitrary processor failure. However, for more than two processors, the scheduling problem is NP-complete even if the tasks have the same deadline.

Problem (Task Sequencing Using Primary-Backup with a Common Deadline)

Instance: Set Σ of tasks, number of processors $m \geq 3$, for each task $t \in \Sigma$, one primary copy $P(t)$ and one backup copy $G(t)$, a length $c(t) \in Z^+$ (i.e., computation time), a common release time $r \in Z^+$, a common deadline $d(t) = D \in Z^+$, and $c(t) = c(P(t)) = c(G(t))$. Note that overlapping among backup copies of the tasks on different processors is allowed. Z^+ denotes the set of positive integers.

Question: Is there an 1-TFT schedule σ for Σ running on m processors, i.e., for each task $t \in \Sigma$, $\sigma_i(P(t)) + c(P(t)) \leq \sigma_j(G(t))$, and $\sigma_i(G(t)) + c(G(t)) \leq D$, where $i \neq j$, i and j designate the indices of processors.

Theorem 1

The Task Sequencing Problem is NP-complete.

Proof

It is easy to verify that the scheduling problem belongs to NP. We now transform the PARTITION problem[18] to the scheduling problem when the number of processors is 3, i.e., $m = 3$.

The PARTITION problem is defined as follows:

Instance: Finite set A and a size $s(a) \in Z^+$ for each $a \in A$.

Question: Is there a subset $A' \subseteq A$ such that $\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)$?

This problem has been proven to be NP-complete[18] and due to its simple form, it has frequently

been used in the NP-complete proof of other problems.

Given an instance of $A = \{a_1, a_2, \ldots, a_n\}$ of the PARTITION problem, we construct a task set Σ using the primary-backup copy approach to run on three processors for the tolerance of one arbitrary processor failure, such that Σ can be scheduled, if and only if there is a solution to the PARTITION problem. Σ consists of $n + 4$ tasks as follows:

$$r(t) = 0,\ c(t) = a_t,\ d(t) = 3B,$$

for $t = \tau_1, \tau_2, \ldots, \tau_n$, where $\sum_{1 \le i \le n} a_i = 2B$ (This can be assumed without lose of generality). These n tasks are referred to as α-type tasks.

The other four tasks β_1, β_2, β_3, and β_4 are defined as

$$r(\beta) = 0,\ c(\beta) = B,\ d(\beta) = 3B,$$

where $\beta = \beta_1, \beta_2, \beta_3, \beta_4$. These four tasks are referred to as β-type tasks.

It is easy to see that this transformation can be constructed in polynomial time. What we will show in the following is that the set A can be partitioned into two sets S_1 and S_2 such that $\sum_{a \in S_1} s(a) = \sum_{a \in S_2} s(a)$ and $S_1 + S_2 = A$, if and only if the task set can be scheduled to produce an 1-TFT schedule.

First, suppose that A can be partitioned into two sets S_1 and S_2 such that $\sum_{a \in S_1} s(a) = \sum_{a \in S_2} s(a)$ and $S_1 + S_2 = A$. Then for each task $\alpha \in S_1$ whose length is $l(\alpha) = a$, its primary copy is scheduled on processor 2 anywhere during time interval [*B, 2B*), and its backup copy on processor 3 anywhere during time interval [2*B,* 3*B*). For each task $\alpha \in S_2$ whose length is $l(\alpha) = a$, its primary copy is scheduled on processor 3 anywhere during time interval [*B,* 2*B*), and its backup copy on processor 2 anywhere during time interval [2*B,* 3*B*). For the tasks β_1, β_2, β_3, and β_4, they are scheduled in the manner as shown in Figure 2. The schedule thus generated is 1-TFT according to Lemma 2. Therefore, the task set Σ is scheduled on the three processors such that the schedule is 1-TFT.

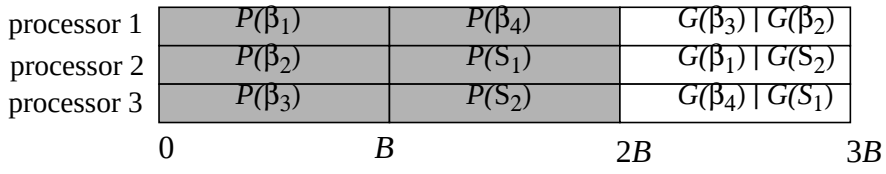


FIG. 2: *Mapping from PARTITION to Task Sequencing*

Conversely, if the task set Σ can be scheduled on three processors such that the schedule is 1-TFT, then the schedule has one of the two forms as given in Figures 3 and 4, if the processors are properly renamed and the tasks properly adjusted. Note that for each processor schedule, shuffling

the primary copies in front of all the backup copies will not violate any scheduling constraint, since primary copies can start earlier than scheduled and backup copies can start later than scheduled, as long as the release time and the deadline constraints are not violated.

Case 1 (Figure 3): The primary copies of the four tasks β_1, β_2, β_3, and β_4 are scheduled on three processors. Let us assume, without lose of generality, that the primary copies of β_1 and β_2 are scheduled on processor 1. Then one of their backup copies must start at time *2B* and complete at the deadline 3*B*, either on processor 2 or on processor 3. It is further assumed that backup copy is scheduled on processor 2. For processor 3, exactly one copy, either primary or backup, of any task among the *n* α-type tasks must be scheduled on it. This is because any 1-TFT schedule for the three processor requires that no idle time exists on any processor, and the primary copy of a task and its backup copy must not be scheduled on the same processor. Therefore, let all the tasks scheduled on processor 2 during time interval [*B*, 2*B*) be the set S_1 (U_2 during [*B*, 2*B*] in the Figure 3), and the tasks on processor 1 during time interval [2*B*, 3*B*) be the set S_2, we have $\sum_{a \in S_1} s(a) = \sum_{a \in S_2} s(a)$ and $S_1 + S_2 = A$. We have solved the PARTITION problem.

Case 2 (Figure 4): The primary copies of the four tasks β_1, β_2, β_3, and β_4 are scheduled on two processors. For the backup copies of the tasks β_1, β_2, β_3, and β_4, there are two cases in which they can be scheduled.

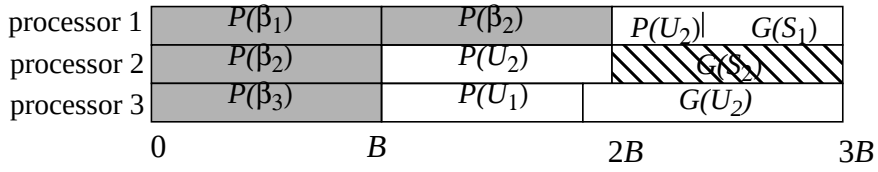


FIG. 3: *Mapping from Task Sequencing to PARTITION*

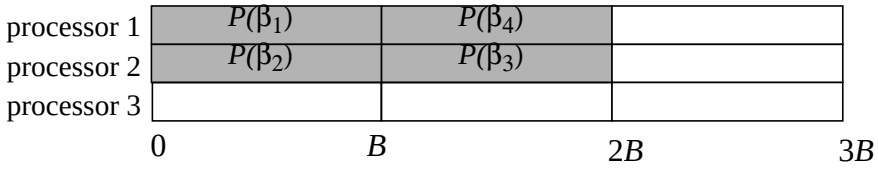


FIG. 4: *Mapping from Task Sequencing to PARTITION*

Case 2.1: The four backup copies are scheduled on processor 3 during time interval [*B*, 3*B*). Then during time interval [0, *B*) for processor 3, only primary copies can be scheduled if any 1-TFT schedule exists. Let all the tasks scheduled on processor 3 during time interval [0, *B)* be the set S_1, and the rest of the *n* tasks be the set S_2, we again have $\sum_{a \in S_1} s(a) = \sum_{a \in S_2} s(a)$ and $S_1 + S_2 = A$.

Case 2.2: Two of the four backup copies are scheduled on processor 1 and processor 2 during time interval [2*B*, 3*B*) respectively. This implies that all the primary copies of the *n* tasks are scheduled on processor 3 (if any of the *n* tasks is scheduled on processor 1 or 2, then there is not enough time for any of the backup copy of β-type tasks to finish). The backup copies of the *n* α-type tasks must be scheduled on processor 1 and 2 during time interval [2*B*, 3*B*). Let all the tasks whose backup copies are scheduled on processor 1 during time interval [2*B*, 3*B*) be the set S_1, and the tasks whose backup copies are scheduled on processor 2 during time interval [2*B*, 3*B*) be the set S_2, we have $\sum_{a \in S_1} s(a) = \sum_{a \in S_2} s(a)$ and $S_1 + S_2 = A$. We have solved the PARTITION problem.

Therefore, the scheduling problem is NP-complete. ■

1-TFT SCHEDULING ALGORITHMS

The NP-completeness result in the previous section suggests that the more generalized problem, where tasks do not share a common deadline, is at least NP-complete. While we would like to design a heuristic for this problem, it is not yet clear to us at this moment that a reasonably efficient heuristic exists for it. The evidences provided by Jeffay, Stanat, and Martel[19] show that a set of periodic tasks may not be feasibly scheduled non-preemptively on a single processor, even if its total utilization is very small, i.e., close to zero. The utilization of a task is defined as the ratio between its computation time and its period, and the total utilization for a set of tasks is the sum of the utilization of each task in the set. The fact that multiple processors are involved further complicates the scheduling problem, let alone the additional requirement of guaranteeing task deadlines even in the presence of processor failures. Therefore, we restrict our attention to the case where all tasks share a common deadline.

In the following, we will first develope a heuristic to solve the scheduling problem as formulated in the previous section, and then evaluate its performance. Though the requirement that all tasks share a common deadline may seem restrictive, the analytic results obtained below can be quite useful. In fact, our results answer the following question as well: Given a set of tasks each with a primary copy and a backup copy (but with no real-time constraints), and the requirement that the failure of any one processor be tolerated, how to schedule the task set, such that the length of the fault-tolerant schedule is minimized, i.e., all the tasks complete execution as early as possible even in the presence of one arbitrary processor failure?

In scheduling a set of tasks on *m* processors, the algorithm must be designed to minimize the schedule length on each processor such that the task set can be successfully scheduled, and at the meantime, to prevent the overlapping of the primary copy of a task and its backup copy. This 1-TFT scheduling problem, at a glance, seems very much to resemble the scheduling problem of min-

imizing the length of a schedule in a multiprocessor system. Since the scheduling to minimize the length of a multiprocessor schedule is NP-complete, several scheduling heuristics have been developed, among which LPT[20] and MULTIFIT[21] are notable ones. However, there are two key issues that set this 1-TFT scheduling problem apart from the problem to minimize the schedule length: the requirement of scheduling primary copies as well as backup copies, and the requirement that the primary copy of a task can not overlap its backup copy, but backup copies of the tasks whose primary copies are scheduled on different processors can be overlapped with each other. The MULTIFIT algorithm, though out-performing LPT in the worst cases, is not easily adapted to solve this 1-TFT scheduling problem. The LPT algorithm is therefore adopted here to serve as the base algorithm upon which a heuristic scheduling algorithm is developed.

The scheduling algorithm starts by sorting the set of tasks in order of non-increasing computation times, and invokes the LPT algorithm to schedule the set of primary copies on the *m* processors. After all primary copies have been scheduled, all the tasks scheduled on any processor are in order of non-increasing computation time, since the LPT algorithm schedules tasks in the same order. Starting from the first processor schedule, we repeatedly apply the *Adapted Largest Processing Time* first (or ALPT) algorithm to the backup copies of the tasks, whose primary copies are scheduled on the same processor, until either the inability of the heuristic algorithm to schedule the task set is reported, or all the *m* processor schedules are exhausted. In the later case, the task set can be scheduled by the heuristic algorithm to produce an 1-TFT schedule on *m* processors. The ALPT algorithm schedules tasks like LPT, except that the tasks (backup copies) may be scheduled a little bit later than they should be in LPT. This modification is to avoid the overlapping of the primary copy of a task and its backup copy.

We use pseudo-code to describe the heuristic algorithm as follows. Note that we sometimes refer to the *m* schedules for *m* processors as one schedule as a whole. Let $s_i(\tau)$ and $f_i(\tau)$ denote the starting time of task τ and its finishing time on processor *i*, respectively. The processors are numbered from 1 to *m*. The function ρ is defined as $\rho(L_y) = y$ for the schedule L_y, or $\rho(\upsilon) = y$ for task υ, where *y* is the index of the processor on which task copy υ is scheduled. L_y denotes the length of schedule or the schedule itself (understood by context) for the processor *y*.

Algorithm OV (Input: Task Set Σ, *m*, 1-TFT; Output: *success*, *schedule*)

(1) Sort the tasks in the order of non-increasing computation time and rename them $\tau_1, \tau_2, \ldots, \tau_n$. Compute $\Omega = \sum_{i=1}^{n} c(\tau_i) = \sum_{i=1}^{n} c_i$. If $\Omega \geq mD$ or $l(T_1) > D/2$, then success := false and report that the task set is not schedulable on *m* processors such that a 1-TFT schedule be produced. Otherwise, go to step (2).

(2) Apply LPT algorithm to schedule the task set on *m* processors.

(3) Let $L_1, L_2, \ldots, L_m$ denote the lengths of the schedules on m processors (initially equal to the lengths of primary schedules). If $\max \{ L_i \mid (1 \le i \le m) \} > D$, then success := false; exit (the task set can't be scheduled); else o to step (4).

(4) (line 1) For processor $i \leftarrow 1$ to m do

Let $\upsilon_1, \upsilon_2, \ldots, \upsilon_{k_i}$ be the k_i tasks (primary copies) scheduled on processor i.

(line 2) For task $j \leftarrow 1$ to k_i do /* ALPT Algorithm */

(line 3) $x := \rho\left(\min \{ L_h \mid h \ne i \wedge 1 \le h \le m \} \right)$;

(line 4) $z := \max \{ f_i(\upsilon_j), L_x \}$;

(line 5) $L_x := z + c(\upsilon_j)$; $s_x(G(\upsilon_j)) := z$;

(line 6) If $L_x > D$ then success := false; exit (the task set is infeasible);

(line 7) success := true; exit.

The scheduling process of algorithm OV can be illustrated by a simple example.

Example 2

The following set of tasks is given to be scheduled on three processors such that one processor failure can be tolerated: $\Sigma = \{\tau_1, \tau_2, \ldots, \tau_7\}$, $\{c(\tau_i) \mid i = 1, \ldots, 7\} = \{10, 8, 8, 7, 6, 3\}$, $r = 0$, and $D = 25$. First, the LPT algorithm is used to schedule the primary copies of the tasks on three processors, as shown by Figure 5. For a processor i, the backup copies of the tasks whose primary copies are scheduled on processor i are scheduled on all the other processors except processor i. The scheduling process is illustrated by Figures 6a, 6b, and 6c. Note that if the number of processors available is two, this task set cannot be scheduled on two processors to produce an 1-TFT schedule.

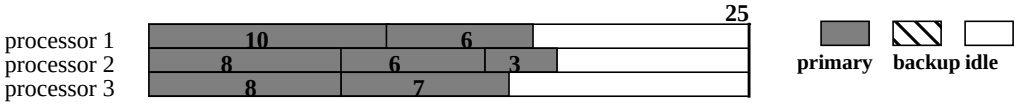
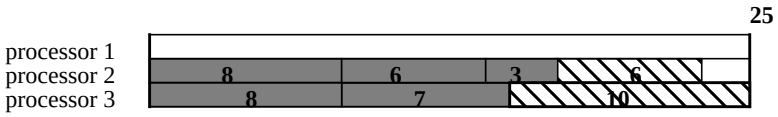
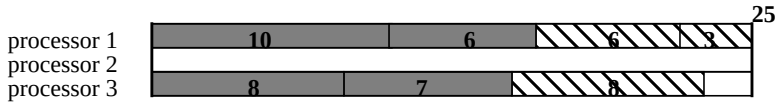


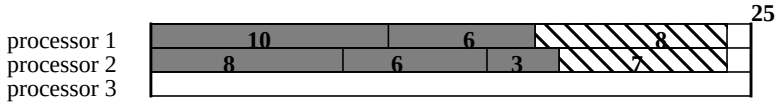
FIG. 5: *Schedule created by LPT*



(a) Schedule created by OV for the backup task copies on processor 1



(b) Schedule created by OV for the backup task copies on processor 2



(c) Schedule created by OV for the backup task copies on processor 3

FIG. 6: *Scheduling process of OV*

The correctness of the schedule generated by OV is guaranteed by the following theorem.

Theorem 2

Algorithm OV generates a 1-TFT schedule.

Proof

According to Lemma 2, what we need to show is that for each task, its primary copy and its backup copy are scheduled on two different processors, such that the starting time for the backup copy is no earlier than the completion time of the primary copy, and its finishing time is no later than the deadline, and that the backup copies of the tasks whose primary copies are scheduled on a processor can not be overlapped in time for their execution in the same processor.

Formally, following the notations used above, we need to show that

$\forall i\,(1 \le i \le m \wedge \forall j\,(1 \le j \le k_i \wedge f_i(\upsilon_j) \le s_x(G(\upsilon_j)) \wedge f_x(G(\upsilon_j)) \le D \wedge i \ne x \wedge \forall j_1\,(j_1 < j \wedge ((\rho(G(\upsilon_{j_1})) = x) \rightarrow (f_x(G(\upsilon_{j_1})) \le s_x(G(\upsilon_j)))))))$ holds, where m is the number of processors, and k_i is the number of primary copies scheduled on processor $i \in [1, m]$.

For each $i \in [1, m]$ and $j \in [1, k_i]$, since $x = \rho\left(\min \{L_h \mid (h \ne i \wedge 1 \le h \le m)\} \right)$ from line 3, $i \ne x$. Since $s_x(G(\upsilon_j)) = z = \max\{f_i(\upsilon_j), L_x\}$, $s_x(G(\upsilon_j)) \ge f_i(\upsilon_j)$. $f_x(G(\upsilon_j)) \le D$ from line 6.

Since L_i is initialized to be the length of the primary schedule on processor i, $f_x(G(\upsilon_{j_1})) \le s_x(G(\upsilon_j))$ since $L_x = z + c(\upsilon_j)$ and $s_x(G(\upsilon_j)) = z = \max\{f_i(\upsilon_j), L_x\} \ge L_x$ from lines 4 & 5, for $j_1 < j$.

Therefore, the schedule thus generated is 1-TFT. ■

Note that algorithm OV is a heuristic solution to the 1-TFT decision problem, where a task set and a certain number m of processors are given, and the question is to determine whether an 1-TFT schedule exists for the task set running on the m processors. As we have previously noted, it is more likely that in reality, a set of tasks is given and the question is to find the minimum number m of processors such that an 1-TFT schedule can be generated for the task set running on the m processors. This problem then becomes the corresponding optimization problem of the 1-TFT decision

problem. For this case, the solution to the optimization problem can be easily derived from the solution to the decision problem. Here we employ the familiar binary search technique to find the minimum number of processors required to schedule a given set of tasks such that the schedule generated is 1-TFT. The algorithm is given as follows:

Algorithm OV-OPT (Input: Task Set Σ, 1-TFT; Output: *m, schedule*)

(1) $lowerB := \left\lfloor \sum_{i=1}^{n} c_i / D \right\rfloor$; $upperB := n$;

(2) $m := \lfloor (lowerB + upperB) / 2 \rfloor$; If ($lowerB = m$) then {$m := m + 1$; exit};

(3) Invoke OV (Σ, *m*, 1-TFT, *success*, *schedule*);

(4) If *success* then $upperB := m$ else $lowerB := m$; goto step 2.

Let us consider an example.

Example 3

Suppose the same task set is given as in Example 2, and the question is to find the minimum number of processors necessary to execute the task set, allowing for one processor failure. The number of processors returned by executing OV-OPT is three, which is in fact equal to the optimal number of processors required.

The time complexity of Algorithm OV is $O(n\log n + n\log m)$, where n is the number of tasks, and m is the number of processors. The sorting process takes $O(n\log n)$ time. The LPT in step 2 takes $O(n\log m)$ time, and the scheduling of backup copies takes $O(n\log(m-1))$, since there are exactly n backup copies. Algorithm OV-OPT takes $O((n\log n + n\log m)\log n)$ time, since the binary search is bounded by $O(\log n)$.

ANALYSIS AND SIMULATION RESULTS

In order to evaluate the performance of the algorithm OV-OPT, we generate task sets randomly, and run OV-OPT on these task sets. Since the 1-TFT decision problem is NP-complete when the number of processors is three, it is hopeless in practice to use enumeration techniques to find the optimal solution even when the number of tasks is small (e.g. 20). However, to find out how well the algorithm performs, we consider the lowest bound possible for each schedule. Since backup copies are allowed to be overlapped, the minimum number of processors required to schedule a task set is given by $\lceil Sum/D \rceil$, where *Sum* is the total computation time of the tasks in the given set, and D is the deadline or period. Therefore, we use $\lceil Sum/D \rceil$ as the lowest bound possible for each schedule.

Our simulation is carried out in the following fashion: first, a common deadline D is chosen. Then a range of values is chosen, from which the computation times of the tasks are randomly gen-

erated according to the uniform distribution. OV-OPT is run for each set of tasks. The ratio between the common deadline D and the maximum computation time of the tasks, i.e., $r = D / \max_i (C_i)$, is kept between 2 and 10. For each different value r, we run OV-OPT for a wide range of task sets. Because of space limit, we only show the result of a typical set of experiments, where $r = 3$ and each data point represents the average value of the number of processors obtained by running 20 independently generated task sets. The result is plotted in Figure 7. It is evident from our extensive simulation that OV-OPT uses less than 5% more processors than the minimum number of processors possible, when the minimum number of processors required is larger than 20. Thus it is concluded that the performance of the algorithm is near-optimal.

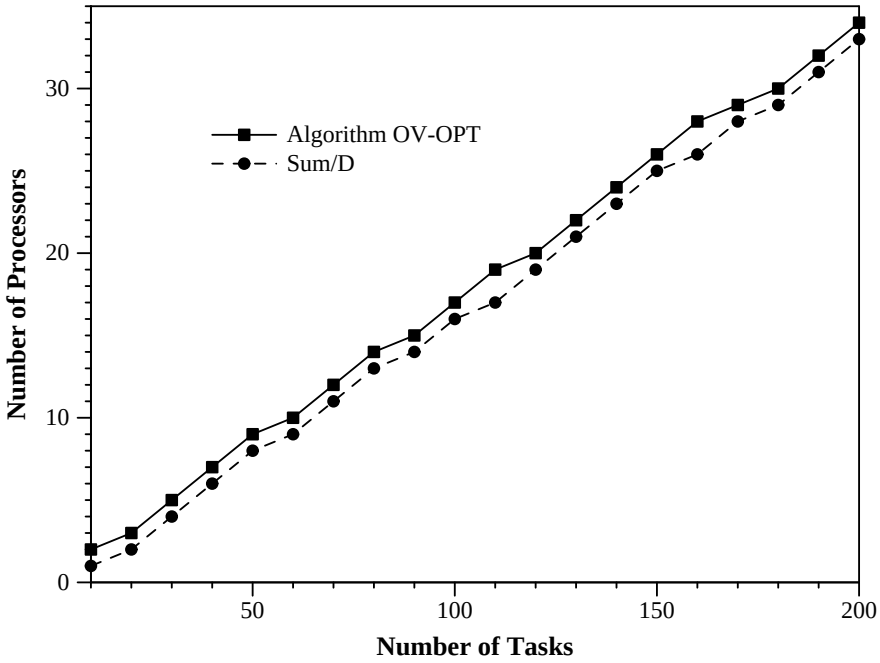


FIG. 7: *Performance of algorithm OV-OPT ($D = 90$, $1 \leq C_i \leq 90$)*

Next we perform an analytic study on the performance of the proposed algorithms. It is apparent that the performance of OV-OPT depends on that of OV, since the former invokes the latter in its execution. Because of the difficulty involved in obtaining a reasonably meaningful upper bound on OV-OPT, we will be instead interested in studying the performance of algorithm OV. Before we proceed any further, let us define what we mean by being optimal for a fault-tolerant schedule with regard to the 1-TFT decision problem and its optimization problem.

For the 1-TFT decision problem, a fault-tolerant schedule is optimal if for all possible processor failure as assumed, its schedule length is the minimum possible. More specifically, let m denote the number of processors in the system, and $W_L(i)$ the length of the fault-tolerant schedule (schedule with primary and backup copies) on the other $m - 1$ processors, assuming that processor P_i has

failed, then the length of the overall fault-tolerant schedule is defined as $W_L = \max_{1 \le i \le m} W_L(i)$. If W_L is the minimum possible, then the schedule is optimal. The algorithm that always generates the optimal schedule is called the optimal algorithm. Let $L_A(\Sigma, m)$ and $L_0(\Sigma, m)$ (or just L_0) denote the length of the overall fault-tolerant schedule generated by heuristic *A* and the optimal schedule length, respectively. Then the ratio

$$\Re_A = \sup\left\{\frac{L_A(\Sigma, m)}{L_0} : \text{all task sets} \Sigma\right\}$$

measures how close a schedule is to an optimal one, in terms of the completion time of the tasks. This metric is an indicator of how good a heuristic scheduling algorithm is. For example, if the ratio $\Re_A$ is 1.2 for heuristic *A*, then as long as the given period *D* is equal to or more than 1.2 of the optimal schedule length, it can always find a feasible schedule. A heuristic with a ratio of 1.2 will always perform no worse than a heuristic with a ratio of 1.8. Note that the ratio is obtained under the worst case.

Similarly, we can define the optimal schedule for the 1-TFT optimization problem as the one that is 1-TFT and for which the minimum number of processors is used. The optimal algorithm is the one that always uses the minimum number of processors to accommodate its 1-TFT schedule. Let $m_B(\Sigma, D)$ and $m_0(\Sigma, D)$ (or just m_0) denote the number of processors required by heuristic *B* to schedule a set of tasks Σ and the minimum number of processors required for the same set of tasks, respectively. Then the ratio

$$r_B = \sup\left\{\frac{m_B(\Sigma, D)}{m_0} : \text{all task sets} \Sigma\right\}$$

measures how well the heuristic *B* performs, as compared with the optimal algorithm, in terms of the number of processors used.

For our algorithms at hand, no proven relationship between $\Re_{\text{OV}}$ and $r_{\text{OV-OPT}}$ has been established. But because of the close relationship between OV and OV-OPT, we have reasons to conjecture that for a very small number $\varepsilon > 0$, $|\Re_{\text{OV}} - r_{\text{OV-OPT}}| \le \varepsilon$. For reader who is interested in finding out how we arrive at this conjecture, please consult the paper by Coffman, Garey, and Johnson[21], where it discusses how a bin-packing heuristic algorithm is modified to solve a similar scheduling problem and the analysis of the performance of the algorithm.

In the following, we seek the value of $\Re_{\text{OV}}$. Note that for any given task set, OV may not be able to schedule it simply because the given deadline *D* is too small or too tight, i.e., the parameter does not play a role in the relative performance of the heuristic algorithm and the optimal algorithm in terms of the completing time of tasks. Hence, in the analysis of the performance of algorithm

OV we disregard *D*, i.e., we omit line 6 of step 4 from algorithm OV.

Let $c_{\max}$ be the largest computation time in a task set and $L(i)$ be the length of the schedule on processor P_i for $i = 1, 2, \ldots, m$. Then we have the following theorem, the proof of which is given in the appendix.

Theorem 3

$\Re_{\text{OV}} \le \frac{3}{2} - \frac{1}{2(m-1)}$, where *m* is the number of processors. $\Re_{\text{OV}} \le 1 + \frac{1}{k} - \frac{1}{k(m-1)}$ if $2 \le k = \lfloor \sum_{\tau} c_i / (m-1) \ / \ c_{\max} \rfloor$.

In the above simulation experiments, since each processor is approximately assigned six tasks and $2 \le k = \lfloor \sum_{\tau} c_i / (m-1) \ / \ c_{\max} \rfloor$, the worst case performance bound for OV should be less than 1 + 1/6 = 1.1667, according to Theorem 3. Since the performance of OV-OPT is closely related to that of OV, we believe that $r_{\text{OV-OPT}}$ assumes a value around 1.1667. In any case, we can claim that the algorithms find schedules that are near-optimal.

CONCLUSIONS

The contribution of this paper is twofold: one is that the NP-completeness result tells us that the TFT scheduling problem is a very hard problem to solve, even in the simple case when there are only three processors and the tasks share a common deadline. Therefore, heuristic approaches are called for to solve the problem. The second contribution is that two heuristic scheduling algorithms are proposed both the decision and the optimization scheduling problems. The algorithms generate schedules that can tolerate one arbitrary processor failure. It is shown through simulation and analysis that the performance of the algorithms is near-optimal.

Many problems remain open, since only a special case of the general real-time fault-tolerant scheduling problem has been considered. The tolerance of more than one processor failures requires that the number of primary copies or backup copies be more than one for each task. Also, good heuristics are needed to obtain approximate solutions to the scheduling problem where tasks have different deadlines. Furthermore, it is interesting to mathematically derive the tight bound for the scheduling algorithms presented in this paper. We are currently investigating these problems.

Acknowledgment – We would like to thank the referees for their encouragement and constructive criticism. Their comments serve to improve the presentation of this paper. This work was supported in part by ONR and by IBM.

APPENDIX

Let us define PS_i as the schedule of primary copies (or *P*rimary *S*chedule) on processor P_i, for

$1 \le i \le m$, Σ_i as the set of tasks whose primary copies are assigned on processor P_i, and $\Sigma_i^- = \Sigma - \Sigma_i$. We further define PM_i as the primary schedule on the other $(m-1)$ processors when processor P_i has failed, i.e., $PM_i = \bigcup_{j=1,\ldots,m, j\neq i} PS_j = \text{LPT}(\Sigma, m) - PS_i$. We assert that the primary schedule PM_i is equivalent to the schedule generated by LPT on the task set Σ_i^- for $(m-1)$ processors. A schedule is equivalent to another schedule if both schedules have the same set of tasks and the starting time of each task (and hence its completion time) is the same in both schedules. The possible difference between two equivalent schedules is that some tasks may be assigned on different processors. The relationship between PS_i and PM_i is illustrated in Figure 8.

Lemma 3

The primary schedule PM_i is equivalent to the schedule generated by LPT on the task set Σ_i^- for $(m-1)$ processors, i.e., $\text{LPT}(\Sigma, m) - PS_i \cong \text{LPT}(\Sigma - \Sigma_i, m-1)$, for $1 \le i \le m$, where LPT(Σ, m) denotes the primary schedule generated by LPT from task set Σ on m processors.

Proof

We first claim that for any task $\tau_j \in \Sigma - \Sigma_i$ with $j \in [1, 2, \ldots, m-1]$, it starts on time zero in LPT(Σ, m) if and only if it starts on time zero in LPT($\Sigma - \Sigma_i$, $m-1$). For LPT($\Sigma - \Sigma_i$, $m-1$), the first $(m-1)$ tasks with the largest computation times are assigned to the $(m-1)$ processors with a starting time of zero. For LPT(Σ, m), the first m tasks with the largest computation times are assigned to the m processors with a starting time of zero. Since one of the first m tasks is deleted, the other $(m-1)$ tasks are the first $(m-1)$ tasks in $\Sigma - \Sigma_i$.

Let $|\Sigma_i| = n_i$, then $\left|\Sigma_i^-\right| = \left|\Sigma - \Sigma_i\right| = n - n_i$. For any task τ_j in $\text{LPT}(\Sigma - \Sigma_i) - PS_i$ with a starting time of $s(\tau_j)$ for $m-1 \le j$, it must be scheduled on a processor other than processor P_i and $s(\tau_j)$ be the earliest idle time among the $(m-1)$ processors. This implies that the starting time for task τ_j in the schedule $\text{LPT}(\Sigma - \Sigma_i, m-1)$ is the same as it is in LPT(Σ, m).

On the other hand, for any task τ_j in $\text{LPT}(\Sigma - \Sigma_i, m-1)$ with a starting time of $s(\tau_j)$ for $m-1 \le j \le n-n_i$, it cannot be scheduled on processor P_i in the schedule LPT(Σ, m), otherwise it would have been deleted in through $(\Sigma - \Sigma_i)$. Therefore, the earliest idle time among the $(m-1)$ processors other than processor P_i is exactly the same as the starting time $s(\tau_j)$ for task τ_j in $\text{LPT}(\Sigma - \Sigma_i, m-1)$. Therefore, the two schedules are equivalent. ■

What Lemma 3 tells us is that every schedule PM_i is equivalent to the schedule generated by LPT on the task set $(\Sigma - \Sigma_i)$. Since OV first schedules the primary copies using LPT and then the backup copies using ALPT, the worst case performance bound is therefore expected to be around $1 + 1/k$ for $k > m$ according to the result by Coffman and Sethi [22]. This is due to the observation that for $k > m$, all the backup copies of the tasks are scheduled immediately after the primary sched-

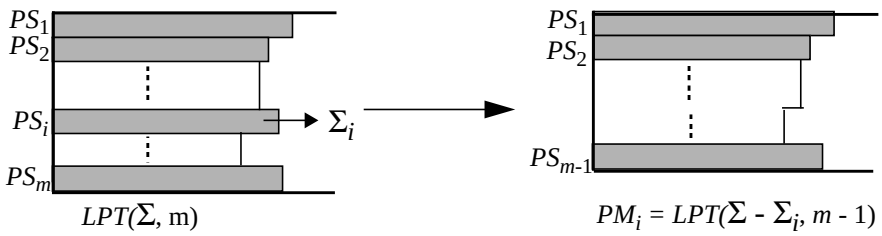


FIG. 8: *Relationship between schedules*

ule on each processor. In the following, we show that our heuristic A has an upper bound which is similar to that for LPT. But it turns out to be non-trivial to show that the upper bounds are tight for heuristic A.

Lemma 4

Let k denote the least number of tasks (primary copies) on any processor or the number of tasks on a processor whose last task terminates the schedule. If $k = 1$, then the schedule is optimal.

Proof

The backup copy of a task will be assigned a starting time no earlier than its primary copy's finishing time. Let τ^* be the task with the minimum computation time requirement c^*, and P^* be the processor on which τ^* is assigned.

For any task τ other than τ^*, its backup copy will be scheduled on processor P^* or any idle processor, with a starting time at c, which is the computation time requirement of task τ. For task τ^*, its backup copy will be assigned to an idle processor with a starting time of c^*, if there is any idle processor, or to the processor on which the task with the second smallest computation time requirement is assigned, with a starting time equal to the finishing time of the task.

Since all the backup copies of the tasks are assigned the earliest starting times as possible, the schedule is therefore optimal. ■

Proof of Theorem 3

Since the backup copy of any task τ must be assigned a starting time no earlier than its primary copy's finishing time, $L_0 \geq 2c$. Let τ^* be the smallest backup copy that finishes last in the fault-tolerant schedule where the processor P_i has failed, and c^* be its computation time requirement. Let P_j be the processor on which τ^* is assigned. Since the primary schedule can be taken as generated by LPT according to Lemma , and the backup schedule by ALPT, we have $L(j) = c^* + s(\tau^*)$, where $s(\tau^*)$ is the starting time of task τ^*. Furthermore, $s(\tau^*) \leq \sum_{\tau \neq \tau^*} c_i / (m-1)$.

$L(j) = c^* + s(\tau^*) \leq c^* + \sum_{\tau \neq \tau^*} c_i / (m-1)$

$$\leq (m-2)\,c^*/(m-1) + \sum_\tau c_j/(m-1)$$

$$\leq (m-2)\,L_0/(2(m-1)) + L_0,$$

since $L_0 \geq \max\{2c^*, \sum_\tau c_i/(m-1)\}$.

Since $\Re_{OV} = \sup\left\{\frac{L_A(\Sigma, m)}{L_0}: \text{all task sets} \Sigma\right\}$, we have $\Re_{OV} \leq \max_{1 \leq i \leq m}\{L(i)/L_0\}$, we have $\Re_A \leq 3/2 - 1/(2(m-1))$.

If $kc_{max} = \sum_\tau c_j/(m-1)$ with $k \geq 2$, then $kc^* \leq kc_{max} \leq \sum_\tau c_j/(m-1) \leq L_0$, where c_{max} is the largest computation time in the task set.

Since $L(i) \leq (m-2)\,c^*/(m-1) + \sum_\tau c_j/(m-1) \leq (m-2)\,L_0/(k(m-1)) + L_0$, we have $\Re_{OV} \leq 1 + 1/k - 1/(k(m-1))$. ■

REFERENCES

1. J.H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostak, and C.B. Weinstock (1978) SIFT: design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE* **66**, 1240-1255.
2. A.L. Hopkins, Jr., T.B. Smith, III, and J.H. Lala (1978) FTMP-A highly reliable fault-tolerant multiprocessor for aircraft. *Proceedings of the IEEE* **66**, 1221-1239.
3. A. Spector and D. Gifford (1984) The space shuttle primary computer system. *CACM* **27**, 874-900.
4. R.M. Kieckhafer, C.J. Walter, A.M. Finn and P.M. Thambidurai (1988) The MAFT Architecture for distributed fault tolerance. *IEEE Transactions on Computers* **37**, 398-405.
5. A. Avizienis (1985) The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering* **11**, 1491-1501.
6. B.W. Johnson (1989) *Design and Analysis of Fault Tolerant Digital Systems.* Addison-Wesley.
7. C.L. Liu and J. Layland (1973) Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM* **10**, 174-189.
8. K. Ramamritham and J.A. Stankovic (1991) Scheduling strategies adopted in Spring: a overview, *Chapter in Foundations of Real-Time Computing: Scheduling and Resource Allocation (ed.) by A.M. van Tilborg and G.M. Koob*, 277-307.
9. L. Sha and J.B. Goodenough (1990) Real-time scheduling theory and Ada. *IEEE Computer* **23**(4), 53-65.
10. K.G. Shin, G. Koob and F. Jahanian (1991) Fault-tolerance in real-time systems. *IEEE Real-Time Systems Newsletter* **7**(3), 28-34.
11. S. Ramos-Thuel and J.K. Strosnider. The transient server approach to scheduling time-critical recovery operations. *12th Symposium on Real-Time Systems*, San Antonio, Texas, 286-295.
12. C.M. Krishna and K.C Shin (1986) On scheduling tasks with a quick recovery from failure.

IEEE Transactions on Computers **35**, 448-454.

13. S. Balaji, L. Jenkins, L.M. Patnaik, and P.S. Goel (1989) Workload redistribution for fault-tolerance in a hard real-time distributed computing system. *FTCS*-**19**, Chicago, Illinois, 366-373.
14. J.A. Bannister and K. S. Trivedi (1983) Task allocation in fault-tolerant distributed systems. *Acta Informatica* **20**, 261-281.
15. Y. Oh and S.H. Son (1991) Multiprocessor support for real-time fault-tolerant scheduling. *IEEE Workshop on Architectural Aspects of Real-Time Systems*, San Antonio, Texas, 76-80.
16. Y. Oh and S.H. Son (1992) An algorithm for real-time fault-tolerant scheduling in multiprocessor systems. *4th Euromicro Workshop on Real-Time Systems*, Athens, Greece, 190-195.
17. Y. Oh and S.H. Son (1994) Scheduling hard real-time tasks with tolerance of multiple processor failures, *Microprocessing and Microprogramming* **40**, 193-206.
18. M.R. Garey and D.S. Johnson (1978) *Computers and Intractability: A guide to the theory of NP-completeness.* W.H. Freeman and Company, NY.
19. K. Jeffay, D.F. Stanat, and C.U. Martel (1991) On non-preemptive scheduling of periodic and sporadic tasks. *12th Symposium on Real-Time Systems*, San Antonio, Texas, 129-139.
20. R. L. Graham (1969) Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.* **17**, 416-429.
21. E.G. Coffman, Jr., M.R. Garey and D.S. Johnson (1978) An application of bin-packing to multiprocessor scheduling. *SIAM J. Comput.* **7**, 1-17.
22. E.G. Coffman, Jr. and R. Sethi (1976) A generalized bound on LPT sequencing. *Revue Francaise d'Automatique Informatique Recherche Operationelle* **10**, Suppl. 17-25.