Scheduling Hard Real-Time Tasks with Tolerance of multiple processor failures

Yingfeng Oh and Sang H. Son

Technical Report No. CS-93-28
May 24, 1993

Scheduling Hard Real-Time Tasks with Tolerance of Multiple Processor Failures

Yingfeng Oh and Sang H. Son[1]

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract

Real-time systems are being extensively used in applications that are mission-critical and life-critical, such as space exploration, aircraft avionics, and robotics. Since these systems are usually operating in environments that are non-deterministic, and even hazardous, it is extremely important that hard deadlines of tasks be met even in the presence of certain failures. To tolerate processor failures in a real-time multiprocessor system, the problem of scheduling a set of hard real-time tasks with duplication is studied. We first prove that the problem of scheduling a set of non-preemptive tasks on $m \geq 3$ 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 scheduling algorithm can tolerate, in the worst case, one arbitrary processor failure, but in the best case $\lfloor m/2 \rfloor$ processor failures, where m is the number of processors in the system. Experimental data and analysis show that the performance of the algorithm is near-optimal. The research described in this paper is a part of our on-going research effort to address the problem of supporting timeliness and dependability simultaneously in a system.

Keywords: real-time scheduling, parallel processing, fault-tolerance

---

1. This work was supported in part by ONR, by DOE, and by IBM.

I. Introduction

The support of computer systems is indispensable to many applications that are mission-critical and life-critical, such as space exploration, aircraft avionics, and robotics. These applications require not only long duration of reliable services, but also timeliness of operations. Computer systems that are built to support these applications include SIFT [28], FTMP [9], the space shuttle primary computer system [26], and MAFT [11]. These mission critical systems are mainly parallel or distributed systems that are embedded into complex, even hazardous environments, under tight constraints on timeliness and dependability of operations. A great deal of efforts has been invested to make computer systems highly dependable and predictable, just to cite a few, [1] [10] [12] [17] [18] [21] [22] [24] [27].

Yet, conspicuously lacking in this scenario is a formal approach towards supporting timeliness (real-time) and dependability (fault-tolerance) simultaneously in a system at the level of task scheduling. Traditional approaches to provide fault-tolerance and real-time in a system have been to separate the concern of the two issues, i.e., the timeliness of tasks are ensured through real-time scheduling, with the assumption that processors and tasks are fault-free, while the dependability of processors or tasks is achieved through redundancy techniques, assuming that task deadlines can be met separately. These two assumptions have been challenged recently by several researchers [25], arguing that real-time and fault-tolerant requirements are not orthogonal. Consequently, some efforts [23] have been made to address the joint requirement of timeliness and dependability. 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.

Our approach, which we deem formal, is to formalize this real-time fault-tolerant problem at the level of task scheduling, and systematically study the various cases of this general scheduling problem. In the formation, tasks are characterized not only by timing constraints, but also by degree of redundancy. The scheduling goal is twofold: timeliness and dependability. The general approach of studying scheduling problems is used, i.e., the complexity of the various cases of the scheduling problems is examined, and then optimal algorithms or heuristic algorithms are devised to solve them. By studying the general scheduling problem, we hope that we will be able to answer such critical questions in designing highly responsive and resilient computer systems as the following one: given the overall system requirement of dependability and timeliness, and the characteristics of a task set (possibly with duplicated tasks or different versions) and those of processors, can the task set be scheduled such that the overall system requirement be met? If so, how

to schedule it?

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 present a formal definition of the scheduling problem, followed by our major results on a special case of the scheduling problem. It is shown that the problem of scheduling a set of real-time tasks with a common deadline on $m \geq 3$ 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, but in the best case, $\lfloor m/2 \rfloor$ processor failures, where m is the number of processors in the system. Simulation and analysis have been carried out to evaluate the performance of the algorithm, and it is shown that the algorithm finds the optimal solution in most of the cases.

The organization of the paper is as follows. The related work is described in Section II. The scheduling problem is defined and a special case of it proven to be *NP-complete* in Section III. The scheduling heuristic is presented in Section IV. The analysis of the performance of the algorithm and the simulation results for the algorithm are given in Section V. We conclude this paper in Section VI with a look at future work.

II. Related Work

In this section, we focus our review on those studies that are related to the real-time fault-tolerant scheduling problem. It is obvious from our studies that research efforts in this area has been quite limited, and it should be noted that these resultes reviewed below are remotely related to the problem we are considering. Balaji et al [2] presented an algorithm to dynamically distribute the workload of a failed processor 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. Bannister and Trivedi [3] considered the allocation of a set of periodic tasks, each of which has the same number of clones, onto a number of processors, so that a certain number of processor failures can be sustained. An approximation algorithm is proposed, and the ratio of the performance of the algorithm to that of the optimal solution, with respect to the bal-

ance 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. Their allocation algorithm is based on the assumption that sufficient processors are available to accommodate the scheduling of tasks.

Krishna and Shin [13] proposed a dynamic programming algorithm that ensures that backup, or contingency, schedules can be efficiently embedded within the original, "primary" schedule to ensure that hard deadlines continue to be met even in the face of processor failures. They assume that a given algorithm *P*, which finds the optimal nonfault-tolerant schedule for systems, can be split into two subalgorithms: *P1*, which finds the optimal allocation of tasks to processors and also the optimal schedule, by calling *P2*, which is an optimal scheduler for a single processor. Each task in the task set has a cost function associated with it. The scheduling goal is therefore to minimize the total cost of the system. The fault-tolerant algorithm *Q* is derived from algorithm *P*, such that for each processor, *Q* takes as its inputs the set of primary clones and the set of backup clones, and produces as its output a fault-tolerant schedule that can sustain up to N_{sust} processor failures, while the total cost of the system is minimized. However, the algorithm in [13] has a severe drawback for the following reason: the problem to schedule a set of independent and preemptive tasks with different release times and deadlines and different weighted functions to minimize the total cost on a single processor is *NP-hard* [14]. This implies that it is unlikely to find an efficient algorithm *P*, which was assumed to exist and used as the base algorithm for *Q*. Furthermore, no such algorithm as *P* has yet been found, that can be split into two subalgorithms that are both optimal.

We have investigated several special cases of the real-time fault-tolerant scheduling problem. Two scheduling algorithms [19] [20] have been proposed to obtain approximate solutions to those special cases. The complexity result presented in this paper is the first solid evidence that even for a very simple case of the scheduling problem, it is intractable. The heuristic thus devised is an improvement over the previous ones.

III. 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 algorithm 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 $\Re = \{\tau_1, \tau_2, \ldots, \tau_n\}$ 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 is the release time of task *i*, c_i is the computation time of task *i*, p_i is the period of task *i*, and d_i is the deadline of task *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 $\Re$ 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 schedule, which is *k-TFT* for the task set $\Re$ on *m* processors. In reality, it is more likely that a task set $\Re$ 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 $\Re$ on *m* processors. This then becomes an optimization problem. If a decision problem is *NP-complete*, then its corresponding optimization problem is at least *NP-complete.*

The TFT scheduling problem is a natural extension to the real-time scheduling problem. Figure 1 depicts the structure of the TFT problem. On the real-time dimension, the parameters are essentially the same as those used in real-time scheduling. On the fault-tolerance dimension, hardware and software redundancy, more specifically processor and task redundancy, are incorporated into the scheduling problem. The scheduling goal is represented by the *k-TFT* parameter, the meaning of which is given above.

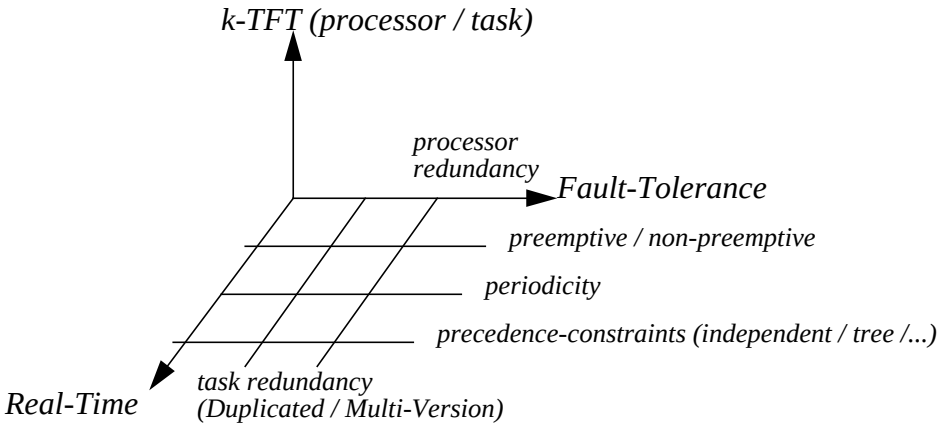

Figure 1: The TFT Scheduling Problem

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 the above case actually corresponds to the primary-backup copy approach or recovery block approach. Primary-backup copy approach requires the multiple implementation of a specification [10]. 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 [20].

Lemma 1: In order to tolerate one or more processor failures and guarantee that the deadline of a task is met using the 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 and the backup copy of each task is scheduled on two different processors and there is no overlapping in time between their executions.

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 which 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 complete a backup copy, assuming that the backup copy has the same computation time requirement as the primary copy. This fact is used implicitly in many situations throughout this 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 not allowed to be overlapped with each other. What we mean by disallowing them to be overlapped is that backup copies of the tasks whose primary copies are scheduled on different processors are not allowed to overlap in time of their executions on a processor. For obvious reasons, backup copies of the tasks whose primary copies are scheduled on the same processor should not be scheduled to overlap in time of their executions on a processor. When the given number of pro-

cessors is two, there obviously 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 when the tasks have the same deadline.

Task Sequencing Using Primary-Backup with a Common Deadline(Non-Overlapping of Backups)

Instance: Set $\Re$ of tasks, number of processors $m \geq 3$, for each task $t \in \Re$, one primary copy $P(t)$ and one backup copy $G(t)$, a length $l(t) \in Z^+$ (the set of natural numbers), a common release time $r \in Z^+$, a common deadline $d(t) = D \in Z^+$, and $l(P(t)) = l(G(t)) = l(t)$. No overlapping of backup copies is allowed.

Question: Is there an m-processor schedule σ for $\Re$ that is *1-TFT*, i.e., for each task $t \in \Re$, $\sigma_i(P(t)) + l(P(t)) \leq \sigma_j(G(t))$, and $\sigma_i(G(t)) + l(G(t)) \leq D$, where $i \neq j$, i and j designate the index of processors.

Theorem 1: *Task Sequencing Using Primary-Backup with a Common Deadline* is *NP-complete*.

Proof: It is sufficient to prove that this scheduling problem is *NP-complete* even in the case of $m = 3$. It is easy to verify that this problem is in *NP*. We next transform the PARTITION problem – an *NP-complete* problem – to the scheduling problem.

The PARTITION problem [7] is stated as follows: Given a finite set A and a "size" $s(a) \in Z^+$ for each $a \in A$, is there a subset $A' \subseteq A$ such that $\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)$?

Given an instance of A = $\{a_1, a_2, \ldots, a_n\}$ of the PARTITION problem, we construct a task set $\Re$ using the primary-backup copy approach to run on three processors for the tolerance of a single arbitrary processor failure, such that $\Re$ can be scheduled, if and only if there is a solution to the PARTITION problem. $\Re$ consists of $n + 1$ tasks as follows:

$r(t) = 0, l(t) = a_t,$

$d(t) = 2B$, where $t \in \{\tau_1, \tau_2, \ldots, \tau_n\}$, $\sum_{1 \leq i \leq n} a_i = 2B$ (this can be assumed without loss of generality);

and one other task β:

$r(\beta) = 0, l(\beta) = B, d(\beta) = 2B.$

It is easy to see that this transformation can be constructed in polynomial time. What we need to show 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.

First, suppose that A can be partitioned into two sets S_1 and S_2 such that $\sum_{a \in S_1} s(a) = \sum_{a \in S_2} s(a) = B$ and $S_1 + S_2 = A$. Then we schedule, for each $\alpha \in S_1$, the primary copy of the task α with $l(\alpha) = a$ on processor 2 anywhere between time interval [*0, B)*, and its backup copy on processor 3 anywhere between time interval *[B, 2B*). For each task $\alpha \in S_2$ with $l(\alpha) = a$, the

primary copy of task α is scheduled on processor 3 anywhere between time interval [*0, B)* and its backup copy on processor 1 anywhere between time interval *[B, 2B*). Therefore, the *2*n* copies of the *n* tasks can be scheduled on processors satisfying the condition set in ***Lemma 2***. For task β, its primary copy is scheduled on processor 1 during time period [*0, B),* and its backup copy is scheduled on processor 2 between time period *[B, 2B*), as shown in Figure 2. Thus, the task set $\Re$ can

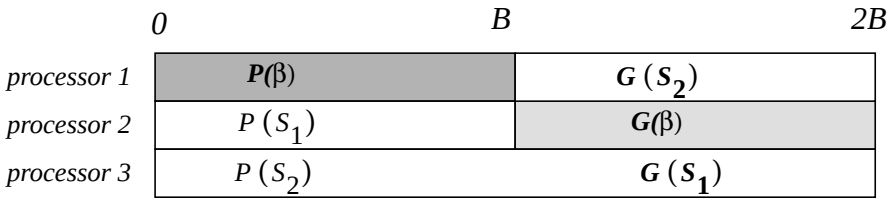

Figure 2: Mapping from PARTITION to Task Sequencing

be scheduled on three processors such that the schedule is *1-TFT*.

Conversely, if the task set $\Re$ is scheduled on three processors such that the schedule is *1-TFT*, we claim that for all tasks scheduled between the time interval *[0, B*) on processor 2, the sum of the tasks' lengths is *B*, i.e., $\sum_{a \in S_1} s(a) = B$. To be able to tolerate one arbitrary processor failure, the primary copy of a task and its backup copy must be scheduled on two different processors and their execution time must not be overlapped. This later requirement is guaranteed by the primary-backup copy approach. Since the common deadline is *2B* and the total task execution time is *2*(2B+B) = 6B*, any *1-TFT* schedule should have no idle time during the time interval *[0, 2B)* on all three processors. Therefore, any *1-TFT* schedule must be equivalent to the schedule shown in Figure 3, if processors are properly renamed and the primary copies are moved in front of all the backup copies for each processor. 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. For processor 3, exactly one copy, either primary or backup, of any task among the *n* 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 never be scheduled on the same processor. Therefore, we let all the tasks

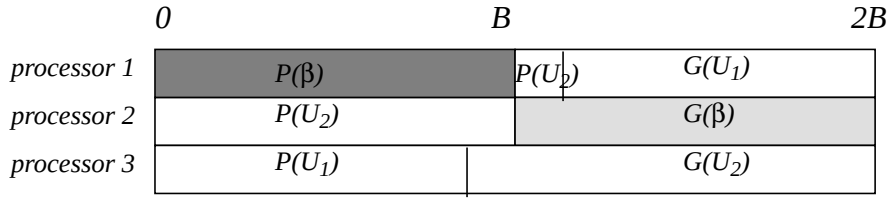

Figure 3: Mapping from Task Sequencing to PARTITION

scheduled on processor 2 between time interval *[0, B)* be the set S_1, and the tasks on processor 1

between time interval *[B, 2B)* be the set S_2. We then 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. The scheduling is therefore *NP-complete*. ■

IV. A *1-Timely-Fault-Tolerant* Scheduling Algorithm

Since the scheduling problem is *NP-complete*, a heuristic scheduling algorithm is presented in this section to obtain approximate solution.

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 scheduling problem, at a first glance, seems very much to resemble the scheduling problem of minimizing the makespan of a schedule in a multiprocessor system. Since the scheduling to minimize the makespan of a schedule is *NP-complete*, several scheduling heuristics have been developed, among which LPT [8] and MULTIFIT [6] are notable ones. However, there are two key issues that set this scheduling problem apart from the one to minimize the makespan: 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, and backup copies of different tasks can not overlap each other in execution either. The MULTIFIT algorithm, though out-performing LPT in the worst cases, is not easily adapted to solve the *1-TFT* scheduling problem. The LPT algorithm is therefore adopted here to serve as the base algorithm upon which a scheduling heuristic is developed.

The algorithm starts by first scheduling the primary copies on the *m* processors using the LPT algorithm. It then schedules the backup copies, by following several rules described below, such that the primary copy of a task and its backup copy are scheduled on different processors, and the backup copies of those tasks, whose primary copies are scheduled on a processor, are also scheduled on one processor. The algorithm is given as follows. Note that *D* is the common deadline of the tasks.

Algorithm 1 (Input: Task Set $\Re$, *m*, *1-TFT*; Output: *success*, *schedule*)

Step 1: Sort the tasks in order of non-increasing computation times and rename them $T_1, T_2, \ldots, T_n$. Compute $\Omega = \sum_{i=1}^{n} l(T_i)$. If $\Omega > (mD)/2$ or $l(T_1) > D/2$, then report that the task set can not be scheduled on *m* processors by this algorithm such that a *1-TFT* schedule can be produced. Otherwise, go to **Step 2**.

Step 2: Apply the LPT algorithm to schedule the task set on *m* processors.

Step 3: Sort the primary schedules for the *m* processors in order of non-increasing sched-

ule lengths. Duplicate the primary schedules to form *m* backup schedules and append them at the end of the primary schedules (Figure 4a).

Step 4: Swap the backup schedules according to the swapping rules defined below (Figure 4b). Shift the backup schedules to obtain the mixed schedules according to the shifting rules defined below (Figure 4c).

Step 5: Find the maximum length among the mixed schedules and compare it to *D*. If it is longer than *D***,** the task set cannot be scheduled. Otherwise, the mixed schedules generated in **Step 4** are the schedules which are *1-TFT* as a whole.

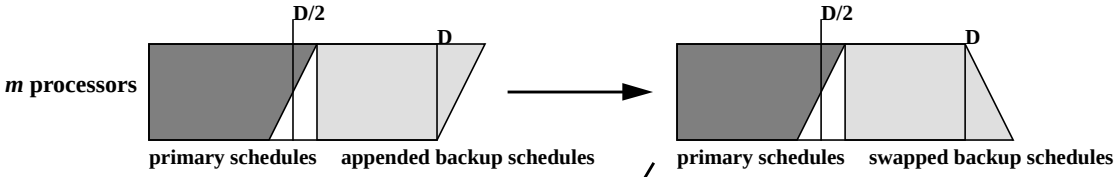

Figure 4a: Schedules after Appending

Figure 4b: Schedules after Swapping

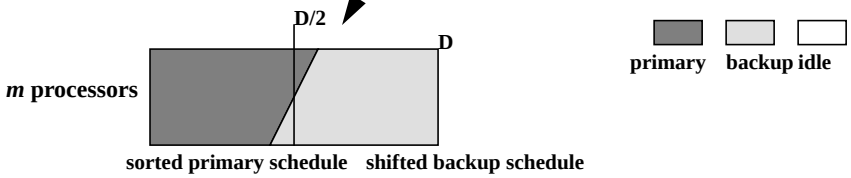

Figure 4c: Schedules after Shifting

The functioning of the algorithm is illustrated by the following simple example.

Example 1: Using Algorithm 1 to schedule the following task set on four processors: $\Re = \{\tau_1, \tau_2, \ldots, \tau_7\}$, $\{l(\tau_i) \mid i = 1, \ldots, 7\} = \{10, 8, 8, 7, 6, 6, 3\}$, r = 0, and $D = 25$. First, the LPT algorithm is used to schedule the primary copies of the tasks on four processors, as shown by Figure 5a. Secondly, the four primary schedules are sorted in non-increasing order. Thirdly, the primary schedules are duplicated to form the backup schedules, which are then appended to the back of the primary schedules. Finally, the backup schedules are swapped and shifted appropriately. The final result is shown in Figure 5b. Note that if the number of processors available is three, the task set cannot be scheduled by this algorithm.

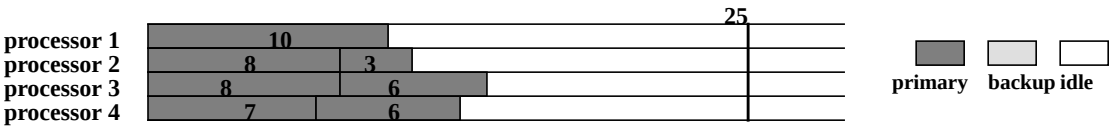

Figure 5a: Schedule Generated by LPT

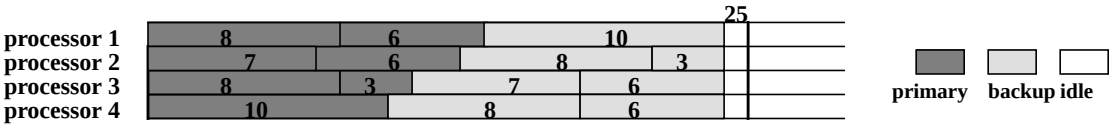

Figure 5b: Schedule Generated after Swapping and Append-

The reason to sort the primary schedules before appending is to minimize the maximum length of the mixed schedule along with the swapping and shifting processes in the later stages. The swapping process makes sure that the backup copy of a task is not scheduled on the same processor as its primary copy. The purpose of shifting is to minimize the finishing time of the mixed schedule as well as to avoid the overlapping of backup copies among different tasks. To elaborate on the swapping and shifting processes, we formally define the swapping and shifting rules.

Swapping Rules:

(1) If the number of processors *m* is even, the longest backup schedule is appended behind the shortest primary schedule, and the second longest backup schedule is appended behind the second shortest primary schedule, and so forth.

(2) If *m* is odd, then the backup schedules of the three central processors are appended in acyclic fashion. The three central processors are the ones whose positions are in the middle. The backup schedules of the rest of the processors are swapped by following swapping rule (1).

To define the shifting rules, we need the following definitions.

***Definition 1*:** Two processors are called twin processors if backup copies of the tasks in the primary schedule on a processor are appended after the primary schedule of the other processor. The two schedules on twin processors are called twin schedules. For example, in Figure 5b, processors 1 and 4 are twin processors, so are processors 2 and 3.

Definition 2: For the primary schedule of a processor i, $l_p(i)$ is defined as its primary schedule length. $l_q(i)$ is defined as the computation time of the first task in the primary schedule. Obviously, $l_p(i) \geq l_q(i)$. Though $l_p(i)$ denotes the length of a schedule, it will also be used to denote the corresponding time interval whose length is $l_p(i)$.

***Shifting Rules*:**

Suppose the backup schedule of processor j is appended behind the primary schedule of processor i.

(1) If $l_p(i) \leq D/2$ and $l_p(j) \leq D/2$, then the tasks in $l_p(j)$ are shifted together ahead of time such that the starting time of the first task in $l_p(j)$ is $max\{l_p(i), l_p(j)\}$. If

$l_p(j) \neq L_q(j)$, the starting time of the first task in $l_p(j)$ can be moved to $l_p(i)$ and the rest of the backup copies can be moved ahead accordingly.

(2) If $l_p(i) \leq D/2$ and $l_p(j) > D/2$, the tasks in $l_p(j)$ are shifted together ahead of time such that the starting time of the first task in $l_p(j)$ is $l_p(i)$.

(3) If $l_p(i) > D/2$ and $l_p(j) \leq D/2$, the tasks in $l_p(j)$ are shifted together ahead of time such that the starting time of the first task in $l_p(j)$ is $l_p(i)$.

(4) If $l_p(i) > D/2$ and $l_p(j) > D/2$, the tasks in $l_p(j)$ are shifted together ahead of time such that the starting time of the first task in $l_p(j)$ is $l_p(i)$.

(5) Apply the above rules to every schedule on the processors.

The schedule thus generated by ***Algorithm 1*** is *1-TFT*, as shown by the following theorem.

Theorem 2*:* ***Algorithm 1*** produces a *1-Timely-Fault-Tolerant* schedule.

Proof*:* Since any primary copy of a task and its backup copy are scheduled on two different processors, as guaranteed by the Swapping Rules, we need only show that there is no overlapping between the primary copy of a task and its backup copy. Obviously, there is no overlapping between the primary copy of a task and its backup copy after the swapping process, but before the shifting process. What we need to show is that no overlapping occurs when the shifting is carried out. There are four cases to consider.

Case 1: $l_p(i) \leq D/2$ and $l_p(j) \leq D/2$: Since the starting time for the first task in $l_p(j)$ is $max\{l_p(i), l_p(j)\}$, there is no overlapping between the primary copies of the task scheduled on processor *j* and their corresponding backup copies on processor *i*. If $l_p(j) \neq l_q(j)$, there must be at least two tasks in the primary schedule on processor *j*. Also, the inequality $l_p(i) > l_q(j)$ must hold. If not, the second task on processor *j* should be scheduled on processor *i* according to the LPT algorithm. Since $l_p(i) > l_q(j)$, no overlapping can occur between any primary copy and its corresponding backup copy.

Case 2: $l_p(i) \leq D/2$ and $l_p(j) > D/2$: Since $l_p(j) > l_p(i)$, there must be at least two tasks in the primary schedule on processor *j*. Following similar argument used in Case 1 yields that no overlapping can occur between any primary copy and its corresponding backup copy.

Case 3: $l_p(i) > D/2$ and $l_p(j) \leq D/2$: No overlapping can possibly occur between any primary copy and its corresponding backup copy in this case.

Case 4: $l_p(i) > D/2$ and $l_p(j) > D/2$: Obviously, no overlapping can possibly occur between any primary copy and its corresponding backup copy in this case. If this case occurs, no *1-TFT* schedule can be generated.

Since **Step 5** in the scheduling algorithm ensures that any backup copy finishes before the

deadline *D*, the schedule thus generated is *1-TFT*. The theorem holds. ■

Observation: The schedule generated by ***Algorithm 1*** is *1-TFT* in the worst case and $\lfloor m/2 \rfloor$*-Fault-Tolerant* in the best case, where *m* is the number of processors. The schedule is *1-TFT* by ***Theorem*** *2*. The schedule is $\lfloor m/2 \rfloor$*-Fault-Tolerant*, because the failure of up to $\lfloor m/2 \rfloor$ number processors can be sustained, if none of the $\lfloor m/2 \rfloor$ processors that fail has its twin among them. In the schedule generated by ***Algorithm 1*** as shown in Figure 5b, if processors *1* and *2* fail, their twin processors – processors *3* and *4* can execute the backup copies such that none of the task deadline is missed.

V. Analysis and Performance Evaluation

In order to evaluate the performance of the scheduling algorithm, we develop another heuristic algorithm that calls the above algorithm to solve its corresponding optimization problem. In other words, we assume that the number of processors is not known and the scheduling goal is to find the minimum number of processors required to execute a set of tasks. Then this is the optimization problem corresponding to the schedule problem described above. We use the typical 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 2 (Input: Task Set $\Re$, *1-TFT*; Output: *m* and *schedule*);

Step 1: LowerB := $\left\lfloor [\sum_{i=1}^{n} l(T_i)] / D \right\rfloor$; UpperB := n;

Step 2: m := $\lfloor (LowerB + UpperB)/2 \rfloor$; IF (LowerB=m) THEN {m := m + 1; EXIT};

Step 3: Invoke ***Algorithm 1*** ($\Re$, m, *1-TFT*, success, *schedule*);

IF success THEN UpperB := m ELSE LowerB := m; Goto **Step 2**.

Example 2: Suppose the same task set is given as in Example 1, 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 ***Algorithm 2*** is four, which is in fact equal to the optimal number of processors required.

The time complexity of ***Algorithm 1*** 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. ***Algorithm 2*** takes $O((n\log n + n\log m)\log n)$ time, since the binary search is bounded by $O(\log n)$.

To evaluate the performance of the algorithms – ***Algorithm 1***, we generate task sets randomly, and run ***Algorithm 2***. Since the scheduling problem is *NP-complete*, it is hopeless in prac-

tice to use enumeration techniques to find the optimal solution even when the number of tasks is small. However, to find out how well the algorithms perform, we consider the lowest bounds possible for each schedule. Since backup copies are allowed to overlap, the minimum number of processors required to schedule the task set is $\lceil 2Sum/D \rceil$, where *Sum* is the total computation time of the tasks, and *D* is the deadline or period. The factor of 2 comes from the fact that no overlapping of backup copies is allowed. Therefore, we use $\lceil 2Sum/D \rceil$ 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 generated. ***Algorithm 2*** is run for each set of tasks, the number of which is incremented for each run. The ratio between the common deadline and the maximum computation time of the tasks is kept between 2 and 7. For each different value *D*, we invoke 100 task sets, each of which differs from the previous one in number of one. Eighty different values of *D* are chosen from the range of 20 and 99. For a particular value of *D* (*D = 90*), the performance of the scheduling algorithm is given in Figure 6. It is evident from our extensive simulation that the difference between the number of processors computed by this algorithm and the lowest bound possible is only one or two. Thus it is concluded that the performance of the algorithm is near-optimal.

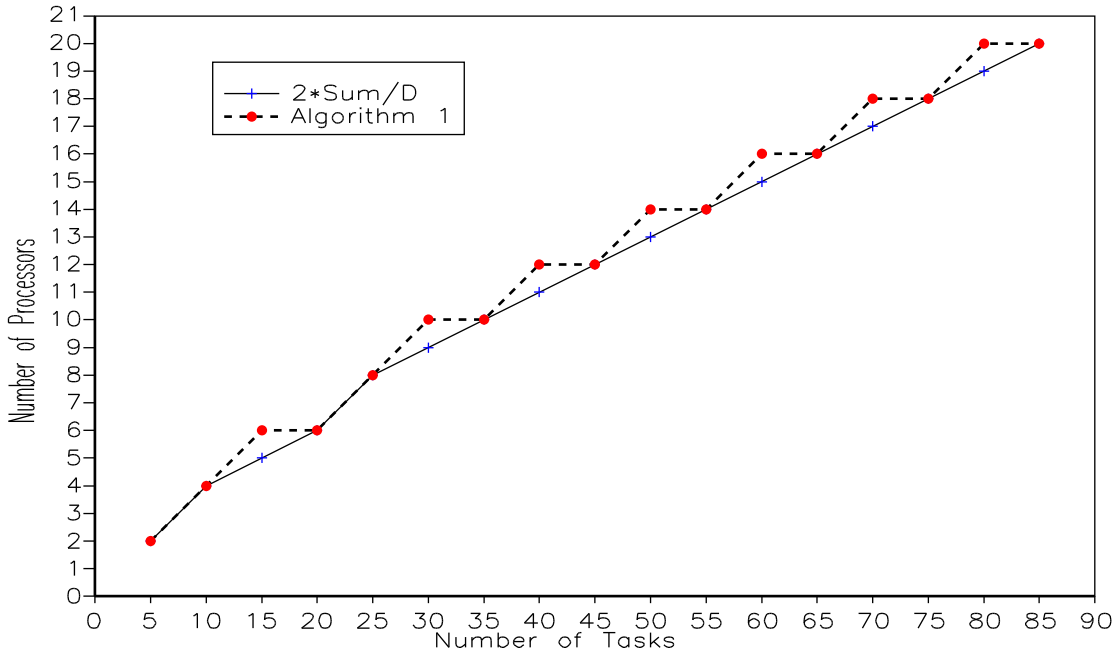

Figure 6: Performance of the Scheduling Algorithm ($D = 90$, $1 \le C_i \le 90$)

The performance of the scheduling heuristic - ***Algorithm 1*** may seem surprisingly good at the first glance. However, it is not surprising at all if we take a closer look at the performance of the heuristic. Graham [8] proved that the worst case performance of LPT was tightly bounded by

4/3 - 1/3m, where *m* is the number of processors. However, that bound is only achievable by a pathological example, where, with the exception of one processor, the number of tasks scheduled on each processor is only two. Coffman and Sethi [5] later generalized Graham's bound to be *(k+1)/k - 1/(km)*, where *m* is the number of processors, and *k* is the least number of tasks on any processor, or *k* is the number of tasks on a processor whose last task terminates the schedule. This result shows that the worst case performance bound for LPT approaches unity approximately as *1 + 1/k*. The worst case performance of ***Algorithm 1*** is therefore expected to be better than *1 + 1/k*.

In our experiments, each processor is approximately assigned five tasks, and thus the worst case performance bounds for both heuristics are expected to be less than 1 + 1/5 = 1.2, according to the above analysis. Also, it is quite unlikely to randomly generate a task set, which can coincide with the worst cases for the heuristic.

VI. Conclusion

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 a scheduling heuristic is proposed to generate a schedule that can tolerate one arbitrary processor failure. It is shown that the performance of the algorithm is near-optimal.

Many problems remain open, since we only consider a special case of the general real-time fault-tolerant scheduling problem. 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 worst-case performance bound of the scheduling algorithm presented in this paper. We are currently investigating these problems.

References

[1] Avizienis, A. "The N-version approach to fault-tolerant software," IEEE Transactions on Software Engineering 11, 1985, pp. 1491-1501.

[2] Balaji, S. et al. "Workload redistribution for fault-tolerance in a hard real-time distributed computing system," FTCS-19, Chicago, Illinois, June 1989, pp. 366-373.

[3] Bannister, J.A. and K. S. Trivedi. “Task allocation in fault-tolerant distributed systems,” Acta Informatica, 20, Springer-Verlag, 1983, pp. 261-281.

[4] Coffman, E.G., Jr. Computer and Job Shop Scheduling Theory, New York: Wiley, 1975.

[5] Coffman, E.G., Jr. and R. Sethi. “A generalized bound on LPT sequencing,” Revue Francaise d’Automatique Informatique Recherche Operationelle, Vol. 10, No. 5, 1976, Suppl., pp. 17-25.

[6] Coffman, E.G., Jr., M.R. Garey, and D.S. Johnson. “An application of bin-packing to multiprocessor scheduling,” SIAM J. Computing 7, 1978, pp. 1-17.

[7] Garey, M.R. and D.S. Johnson. Computers and Intractability: A guide to the theory of NP-completeness, W.H. Freeman and Company, NY, 1978.

[8] Graham, R.L. “Bounds on multiprocessing timing anomalies,” SIAM J. Appl. Math., 17, 1969, pp. 416-429.

[9] Hopkins, A.L. et al. “FTMP-A highly reliable fault-tolerant multiprocessor for aircraft,” Proceedings of the IEEE, Vol. 66, No. 10, October, 1978.

[10] Johnson, B.W. Design and Analysis of Fault Tolerant Digital Systems, Addison-Wesley, 1989.

[11] Kieckhafer, R.M., C.J. Walter, A.M. Finn, and P.M. Thambidurai. “THe MAFT Architecture for distributed fault tolerance,” IEEE Transactions on Computers, Vol. 37, No. 4, April 1988, pp. 398-405.

[12] Knight, J.C. and P.E. Ammann. “Design fault tolerance,” Reliability Engineering and System Safety 32, 1991, pp. 25-49.

[13] Krishna, C.M. and K.C Shin. “On scheduling tasks with a quick recovery from failure,” IEEE Transactions on Computers, C-35(5), May 1986, pp. 448-454.

[14] Labetoulle, J., E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. “Preemptive scheduling of uniform machines subject to release dates,” Report BW 99, Mathematisch Centrum, Amsterdam, 1979.

[15] Leung, J.Y.T. and J. Whitehead. “On the complexity of fixed-priority scheduling of periodic, real-time tasks,” Performance Evaluation, Vol. 2, pp. 237-250, 1982.

[16] Liestman, A.L. and R.H. Campbell. “A fault tolerant scheduling problem,” IEEE Transactions on Software Engineering, SE-12(11), November 1986, pp. 1089-1095.

[17] Liu, C.L., and J. Layland. “Scheduling algorithms for multiprogramming in a hard real-time environment,” JACM 10(1), 1973.

[18] Liu. J.W.S., K-J. Lin, W-K. Shih, A. Yu, A-Y. Chung, and W. Zhao “Algorithms for scheduling imprecise computations,” Computer, Vol. 24, No. 5, May 1991, pp. 58-69.

[19] Oh, Y., and S.H. Son. “Multiprocessor support for real-time fault-tolerant scheduling,” IEEE 1991 Workshop on Architectural Aspects of Real-Time Systems, San Antonio, Texas, pp. 76-80, Dec. 3, 1991.

[20] Oh, Y., and S.H. Son. “An algorithm for real-time fault-tolerant scheduling in multiprocessor systems,” 4th Euromicro Workshop on Real-Time Systems, Athens, Greece, June 1992.

[21] Pradhan, D.K. Fault-Tolerant Computing -- Theory and Techniques, Volumes I and II, Prentice-Hall, Englewood Cliffs, N.J., 1986.

[22] Ramamritham, K. and J.A. Stankovic. “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, 1991.

[23] Ramos-Thuel, S., and J.K. Strosnider. “The transient server approach to scheduling time-critical recovery operations,” RTSS, 1991, pp. 286-295.

[24] Sha, L., and J.B. Goodenough. “Real-time scheduling theory and Ada,” Computer, April 1990, pp. 53-65.

[25] Shin, K.G., G. Koob, and F. Jahanian. “Fault-tolerance in real-time systems,” IEEE Real-Time Systems Newsletter, Vol. 7, No. 3, 1991, pp. 28-34.

[26] Spector, A., and D. Gifford. “The space shuttle primary computer system,” CACM, September 1984, pp. 874-900.

[27] Stankovic, J.A. “Misconception of real-time computing,” IEEE Computer, October 1988, pp. 10-19.

[28] Wensley, et.al. “SIFT: design and analysis of a fault-tolerant computer for aircraft control,” Proc.of the IEEE, Vol. 66, No. 10, October 1978, pp. 1240-1255.