Preemptive Scheduling of Tasks with Reliability Requirements in Distributed Hard Real-Time Systems

Yingfeng Oh and Sang H. Son

Technical Report No. CS-93-25
May 24, 1993

Preemptive Scheduling of Tasks with Reliability Requirements in Distributed Hard Real-Time Systems

Yingfeng Oh and Sang H. Son

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. 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. It is therefore extremely important that these hard real-time systems must be reliable, i.e., task deadlines be met even in the presence of certain faults or failures. In this paper, we address the problem of supporting timeliness and dependability in a real-time system at the level of task scheduling. We consider the problem of scheduling a set of tasks, each of which, for fault-tolerance purpose, has multiple versions, on a number of processors, such that the number of processors used is minimized. Two scheduling algorithms are proposed and evaluated using simulation. It is shown that the algorithms produce near-optimal schedules. The results presented in this paper is a part of our on-going research effort to address the problem of supporting timeliness and fault-tolerance in a distributed / parallel system.

Keywords: fault-tolerance, task scheduling, distributed real-time systems.

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

I. Introduction

Current trend towards building a practical real-time system is to apply parallel or distributed computing with the support of fault-tolerance techniques. The need for parallel or distributed computing is the direct result of the increasing degree of sophistication with respect to the goals established for industrial real-time systems. In some instances, parallel processing may be the "natural" way of viewing a problem or class of problems to be solved. In other cases, due to heavy computing demands, parallel processing can be the best, perhaps only, means of providing sufficient processing power to meet critical real-time deadlines. Since most real-time systems are embedded into unpredictable, even hazardous environments under tight constraints on timeliness of operations, the dependability of these operations must be strongly supported. This presents a serious problem for both real-time and fault-tolerance research communities to support timeliness and dependability simultaneously in a parallel or distributed real-time system.

Task deadlines in a hard real-time system must be guaranteed, otherwise catastrophic consequences may occur. The guarantee of task deadlines is mostly through real-time scheduling. Since many real-time operations or tasks are periodic, the guarantee of periodic task deadlines has been a major concern, and many scheduling algorithms [23] [29] have been proposed. However, all these results are based on the assumption that neither tasks may contain errors nor processors may fail, and yet the meeting of task deadlines may be rendered impossible without that assumption. Since processor failures are inevitable in a system, especially when a large number of processors are employed, and the correctness of a single copy of most software (or tasks) can not be easily guaranteed using current software engineering technology, it is imperative that the capability to tolerate processor failures or task errors in a real-time system must be provided.

Fault-tolerance can be achieved through redundancy techniques [13] [27]. Software and hardware redundancy can be used. For the tolerance of task errors, multiple versions of a task are provided and executed. For the tolerance of processor failures. multiple processors are used to execute the same task set. We argue that for real-time systems, there exist cases that fault-tolerance can not be well supported by separating the real-time scheduling issue from that of employing redundancy. On one hand, it is very difficult to dichotomize the functional specification from the timing specification in a real-time system, since each function is coupled with a timing constraint. Real-time and fault-tolerance can also compete with each other. For example, software redundancy and error recovery routines will enhance fault-tolerance, but may cause tasks to miss deadlines.

On the other hand, a simple approach of applying existing fault-tolerant methods does not fully address the problem. For example, an approach to accomplish fault-tolerance in a real-time

system may be suggested as follows: First, a set of real-time tasks is scheduled on a number of processors such that the task deadlines are guaranteed. Then, a triple redundancy scheme is used, in which, for each processor in the original allocation, is replaced by a group of three processors, such that one processor failure in a group of three can be tolerated. This approach certainly accomplishes fault-tolerance to certain extent. However, it has several drawbacks: (1) Task errors can not be tolerated using this approach. (2) This approach works well only when the number of processors is small. When the number of processors is large, processor failures become more frequent. When the number of processor failures reaches a threshold number, the benefit of using more processors is totally offset. In order to effectively tolerate processor failures, the number of processors used must be minimized. (3) This approach also suffers from the effect of clustering failures of processors. If a processor group fails, then the tasks executed on that group will miss their deadlines. Besides, this approach may under-utilize the processor resource, since some tasks may be less important than others. For the less important tasks, fewer processors can be used to executed them. Therefore, the scheduling of real-time tasks and the employment of redundancy must be considered together as one big problem.

In this paper, we directly tackle this problem by first formulizing it as a scheduling problem, and then proposing two algorithms to solve it. The scheduling problem is defined in a very general manner. For fault-tolerance purpose, multiple versions of a task is executed on different processors. In some cases, the versions of a task may be merely copies of a single version, the purpose of using which is for the tolerance of processor failures. In other cases, the versions of a task may be truly different implementations of a task, the reason of using which is for the tolerance of task errors. The computation times of versions belonging to a task may be different, and the number of versions a task can have may also be different. There arises a challenging problem of scheduling a set of tasks with such a diverse set of requirements, on a number of processors, such that the task deadlines are guaranteed, and the fault-tolerance capability is provided. Furthermore, the number of processors must be minimized in order to support fault-tolerance effectively. In a tautology, for the support of timeliness and dependability, which we dubbed “timely-fault-tolerance” (*TFT* in short), the problem becomes one to schedule a set of real-time tasks using a minimum number of processors, such that for each task, its multiple versions are executed on different processors for fault-tolerance, while its deadline is guaranteed for timeliness. To our best knowledge, this is the first attempt to address *TFT* problem, by defining a scheduling problem in such general terms

The organization of this paper is as follows. The *TFT* scheduling problem is formulated in Section II, followed by a review of related work. Since the scheduling problem itself is intractable, a scheduling algorithm is presented in Section III to solve it. In order to evaluate the perfor-

mance of the algorithm, we use simulation techniques. The simulation methodology is given in Section IV, as well as the performance of the first algorithm. The performance of the first algorithm suggests that there is space for improvement, which prompts the design of the second algorithm in Section V. A short discussion is given in Section VI. In the final section, we conclude by summarizing our major results.

II. Problem Formulation and Related Work

In order to present any scheduling results, it is sensible to state the assumptions beforehand. Along with the assumptions, the justification of them will be given. The effect of relaxing some of the assumptions will be discussed in a Section VI. The presentation of these assumptions follows the format used by Liu and Layland [23].

(A) Each task has κ number of versions, where κ is a natural number. The κ versions of a task may or may not have the same computation time requirement, and the κ versions may be merely copies of one implementation or truly versions of different implementations.

(B) For each task, all its versions must be executed on different processors.

(C) The requests of all tasks for which hard deadlines exist are periodic, with constant interval between request. The request of a task consists of the requests of all its versions — i.e., all versions of a task are ready for execution when its request arrives.

(D) Each task must be completed before the next request for it arrives — i.e., all its versions must be completed at the end of each request period.

(E) The tasks are independent in that the requests of a task do not depend on the initiation or the completion of requests for other tasks.

(F) For each task, the computation times of all its versions are constant and do not vary with time. The computation time here refers to the time which is taken by a processor to execute the task without interruption.

(G) Any non-periodic tasks in the system are special, and do not have hard deadlines. The fault-tolerance of these tasks can be compromised since they are not critical to the overall system performance.

Assumptions (A) and (B) make a rather general statement about the redundancy schemes used by each task, and represent what is widely practised in building fault-tolerant systems. The term “version” has been used in *N*-version programming [2] to denote multiple implementations

of a task. However, for the sake of convenience, it is used here to denote both true versions of a task and more copies of a single task version. In the case of using merely duplicated copies, the errors produced by a task can not be tolerated, since all the versions, or more specifically copies, produce the same results. But processor failures can be tolerated by using more copies of a task. Here we are not concerned ourselves with details about what faults are to be tolerated or how faults are tolerated, rather we just make the general statement that for fault-tolerance purpose, each task has a number of versions, and for each task, all its versions are to execute on different processors. Note that the number of versions used by each task may be different, i.e., the values κ assumes for different tasks may be different.

Assumptions (C), (D), and (F) have been argued [23] to have close resemblance to many industrial real-time systems, and have thus been used by many [18] [29] [33] in studying and building real-time systems. Though Assumption (E) does exclude the situation where certain tasks have precedence of execution before others, it nevertheless is a good model for many real-time systems.

Assumption (G) may appear to be overly restrictive. In fact, with the rapid advance of real-time scheduling techniques [29], Assumption (G) can be totally omitted. It is put in here so that we can focus ourself on periodic tasks at the moment. The ways to schedule non-periodic, hard deadline tasks along with periodic tasks will be discussed in Section VI.

The ***TFT* Scheduling Problem**: Given a set of n tasks $\Re = \{\tau_1, \tau_2, \ldots, \tau_n\}$, where

$\tau_1 = ((c_{11}, c_{12}, \ldots, c_{1\kappa_1}), r_1, d_1, p_1)$

……

$\tau_i = ((c_{i1}, c_{i2}, \ldots, c_{i\kappa_i}), r_i, d_i, p_i)$

……

$\tau_n = ((c_{n1}, c_{n2}, \ldots, c_{n\kappa_n}), r_n, d_n, p_n)$, and

$c_{i1}, c_{i2}, \ldots, c_{i\kappa_i}$ are the computation times of the κ_i versions of task i, r_i, d_i, and p_i are the release time, deadline, and period of task i, respectively.

what is the minimum number of processors that are sufficient to run all the tasks such that all the task deadlines are met and all versions of a task execute on different processors?

In this paper, we only consider the case of preemptive scheduling. According to Assumption (D), the deadline of each task coincides with its next arrival. For periodic task scheduling, it has been proven [23] that the release times of tasks do not affect the scheduling in the long run. Therefore, release time r_i and deadline d_i can be safely omitted when we consider solutions to the problem.

In order to guarantee the deadlines of periodic tasks, the Earliest Deadline First (EDF) algorithm was proven to be optimal for a single processor system by Liu and Layland [23]. The EDF algorithm schedules tasks according to their deadlines, and always assigns the processor to the task having the earliest deadline, preempting other tasks if necessary. A set of *n* periodic tasks is schedulable using EDF if and only if $\sum_{i=1}^{n} C_i/P_i \leq 1$, where C_i and P_i are the computation time and period of task *i*, respectively. For a system with priority scheduling, dynamic assignment of priorities to tasks is necessary in order to make the EDF algorithm work correctly.

Since dynamic priority assignment involves large overhead and complexity in actual system implementation, fixed priority assignment scheme is often considered more practical. For fixed priority assignment, the scheme called "intelligent" fixed priority by Serlin [28] or rate-monotonic fixed priority by Liu and Layland [23] was proven to be optimal. The rate-monotonic algorithm assigns priorities to tasks according to their periods, the shorter the period of a task, the higher its priority is. The deadlines of a set of *n* periodic tasks are guaranteed to be met by the rate-monotonic algorithm if $\sum_{i=1}^{n} C_i/P_i \leq n\,(2^{1/n} - 1)$, or $ln2 = 0.693$ when $n \to \infty$ [23] [28], where C_i and P_i are the computation time and period of task *i*, respectively. Lehoczky et al [19] later gave the necessary and sufficient for the rate-monotonic fixed priority assignment. In this paper, we will study the scheduling problem under both fixed and dynamic priority assignment schemes. For convenience, we refer the condition $\sum_{i=1}^{n} C_i/P_i \leq 1$ for dynamic priority assignment as *EDF condition*, and the $\sum_{i=1}^{n} C_i/P_i \leq n\,(2^{1/n} - 1)$ condition as *RM condition*.

To schedule periodic tasks on a multiprocessor system, the scheduling problem is *NP*-hard [21]. In other words, neither the EDF algorithm nor the rate-monotonic algorithm is optimal. Several heuristics based on rate-monotonic fixed priority assignment strategy [8] [9] [10] have been proposed. The guaranteed performance of two heuristics — Rate-Monotonic-Next-Fit and Rate-Monotonic-First-Fit [10] are upper bounded by 2.67 and 2.2, respectively.

In order to tolerate processor failures, Balaji et al [3] 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. Krishna and Shin [17] 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. Yet their algorithm has the severe drawback that it is premised on the solution to two intractable problems. Oh and Son [25] [26] have investigated several special cases of the *TFT* scheduling problem where the tasks are assumed to be non-preemptive. Several complexity results have been obtained, and two scheduling algorithms have been proposed to obtain approximate solutions to those special cases.

Bannister and Trivedi [4] 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. All the tasks have the same number of clones, 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 solution, 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. Their allocation algorithm is based on the assumption that sufficient processors are available to accommodate the scheduling of tasks.

III. The Design of the First Algorithm

Since the real-time version of the *TFT* scheduling problem — the problem of scheduling a set of periodic tasks with a single version on a minimum number of processor is *NP*-complete [21], the *TFT* problem is therefore at least *NP*-complete. A heuristic scheduling algorithm is proposed in this section to solve it. Since the EDF and rate-monotonic algorithms are optimal algorithms for scheduling periodic tasks on a single processor, they can be used to schedule a subset of tasks that is assigned to each processor. The *TFT* scheduling problem can be solved as a whole if the following requirements are satisfied:

(a) For each task, all its versions are scheduled on different processors.

(b) The number of processors allocated is minimized.

Requirement (a) can be met by keeping track of the processors that previous versions of a task have been allocated. To meet requirement (b), several allocation heuristics may be available, yet it is hard to tell which is better.

The approach to adapt the allocation algorithm proposed by Bannister and Trivedi [4] for the solution to the *TFT* problem seems to be attractive. Even though no schedulability test is introduced into their allocation algorithm, it can be added easily. The result of *(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, is particularly attractive, since it indicates that the allocation algorithm can balance the workload among the processors evenly. Besides, different versions of a task are assigned to different processors using their algorithm. The only problem left is to find the minimum number of processors to accommodate the task set. We accomplish this by using a binary search technique.

The design of the heuristic consists of two steps: First, assuming *m* number of processors is sufficient for the execution of the task set $\Re$, the following algorithm — ***Algorithm 0*** is invoked

to assign versions of tasks onto different processors such that the versions of a task are assigned to different processors, and the set of assigned tasks on each processor is schedulable under EDF or rate-monotonic algorithms. Second, a binary search technique — ***Algorithm 1*** is used to find the minimum number of processors that is possible under ***Algorithm 0*** to execute the task set.

Algorithm 0 (Input: task set $\Re$, *m*; Output: *success*)

(1) Initialize $U_i = 0$ for $1 \le i \le m$, and $t = 1$.

(2) Assign the κ_t versions of task *t* simultaneously to the κ_t least utilized processors, and increment the utilization for each processor *i* that a version of task *t* has been assigned to by c_{tj} / p_t, where $j \in \{1, 2, \ldots, \kappa_t\}$. If $U_i > l(2^{1/l} - 1)$ for rate-monotonic algorithm, where *l* is the number of versions having being assigned to processor *i* (or $U_i > 1$ for EDF), then *success = FALSE*, return. Otherwise, increment $t = t + 1$.

(3) If t > n then *success = TRUE*, return. Otherwise, go to (2).

Algorithm 1 (Input: task set $\Re$; Output: *m*)

(1) LowerBound = $\sum_{i=1}^{n} (\sum_{j=1}^{\kappa_i} c_{ij}) / p_i$; UpperBound = $n \times max_{(1 \le i \le n)} \{\kappa_i\}$;

(2) *m* = (LowerBound + UpperBound) / 2; IF (LowerBound = *m*) THEN {*m* = *m* + 1; EXIT};

(3) Invoke ***Algorithm 0*** ($\Re$, *m*, *success*); IF *success* THEN UpperBound = *m* ELSE LowerBound = *m*. Goto (2).

Algorithm 0 can be used to schedule multi-version, periodic tasks on processors, either for fixed or dynamic priority assignment schemes. The only difference between the two schemes appearing in ***Algorithm 0*** is the difference between the guaranteed bounds used: one — EDF is 1 and the other — rate-monotonic is $l(2^{1/l} - 1)$, depending on the number of tasks assigned. The lower bound for the number of processors that is sufficient to execute the task set is given by $\sum_{i=1}^{n} (\sum_{j=1}^{\kappa_i} c_{ij}) / p_i$, which is the total computation time requirement of the task set, without the fault-tolerant constraint (a). The upper bound of the number of processors is given by $n \times max_{(1 \le i \le n)} \{\kappa_i\}$, which is the total number of tasks *n* times the maximum number of versions a task has among the *n* tasks. The upper bound is true regardless whether the deadlines of tasks are guaranteed using dynamic priority assignment scheme, e.g., EDF, or fixed priority assignment scheme, e.g., rate-monotonic, on each individual processor.

The correctness of the algorithm is somewhat self-evidence. The condition check at step (2) in ***Algorithm 0*** ensures that all the tasks assigned to a processor can meet their deadlines. When the deadline of the last task is guaranteed, the deadlines of all the tasks are guaranteed. The fault-tolerant requirement — constraint (a) is also met at step (2) in ***Algorithm 0***. The complexity of ***Algorithm 0*** is $O(\kappa n \log m)$, where κ is the maximum number of versions a task has among the *n*

tasks. There are a total of n tasks, and for each task, $O(\kappa \log m)$ operations are needed to find the κ least utilized processors. The complexity of ***Algorithm 1*** is give by $O((\kappa n \log m) \times \log(\text{Upper-Bound-LowerBound}))$.

IV. Simulation

Since heuristic algorithms are usually developed to obtain approximate solutions for problems that are intractable or likely to be intractable, they tend to have the property of producing good solutions sometimes, and bad solutions some other times. In order to evaluate how well a heuristic algorithm performs, there are two major approaches: analytical and experimental. For some heuristics, their worst-case, or even asymptotic performance can be obtained using formal analysis. For others, formal analysis can be hopelessly complicated, and thus rendered impractical. Where formal analysis fails, simulation may be the only way to evaluate the performance of heuristics. In our case, a full analysis of the performance of the scheduling heuristic seems to require a significant amount of time and efforts, and it has possibility that it can not be done in the near future. We therefore resort to simulation studies.

Our simulation studies consist of two steps: 1. Generate sets of tasks with random distributions. 2. Fit the task sets into ***Algorithm 1*** to produce the results. To generate a task set, the following parameters must be chosen: the number of the tasks, the number of versions for each task, the computation time for each version, and the period for each task. The performance of ***Algorithm 1*** is shown in Figure 1. The total utilization is given by $\sum_{i=1}^{n} (\sum_{j=1}^{\kappa_i} c_{ij}) / p_i$. The total utilization corresponds to the least number of processors needed to execute the task set. It is a lower bound on the number of processors to be computed.

The periods of tasks are randomly generated between the range of 10 and 50. The number of versions each task has is randomly generated between the range of 1 and 5. The computation time of each version is randomly generated between a range of 1 and its period. The performance of the algorithm under *EDF condition* is better than that of *RM condition*. This is expected since the total utilization for each processor is bounded by $n(2^{1/n} - 1)$ under *RM condition*, where n is the number of tasks having been assigned to a processor, while it is bounded by 1 under the *EDF condition*. The *RM condition* is dynamic, in the sense that as more tasks are assigned to a processor, the bound gets lower. The EDF bound, however, is fixed, independent of the number of tasks assigned to a processor.

In some cases, the number of processors retuned by ***Algorithm 1*** is more than twice the total utilization of task set. Even though the total utilization may be far smaller than the optimal num-

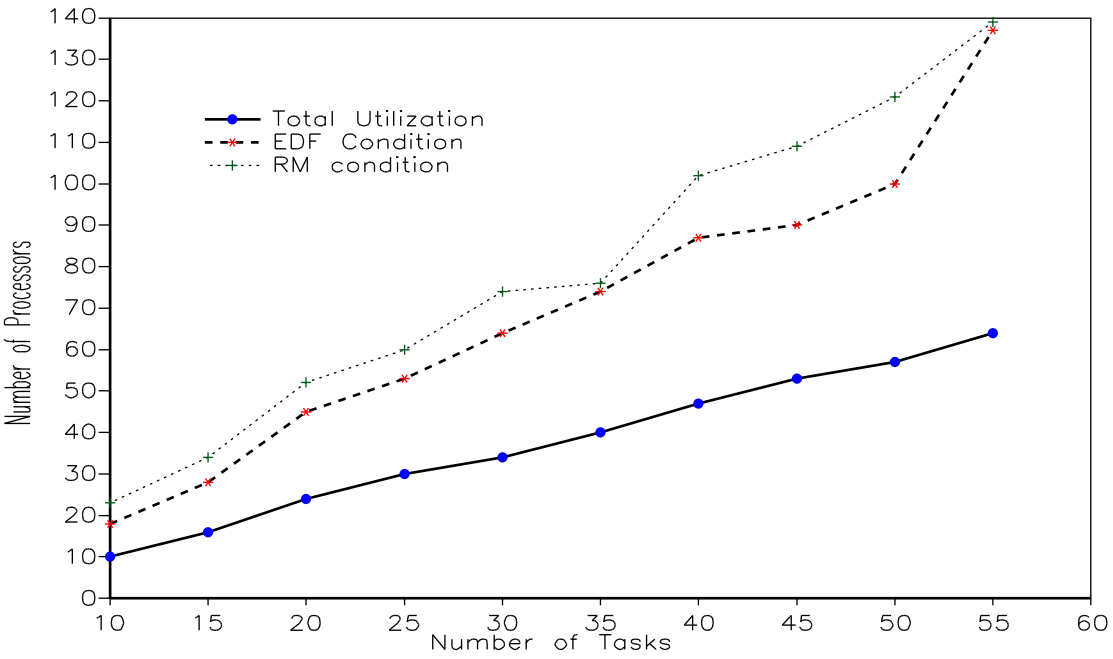

Figure 1: Performance of the First Algorithm under *RM/EDF Conditions*

ber of processors necessary to execute the task set under fault-tolerance constraint, it still comes as a surprise to us that the number of processors used by the algorithm is still large.

Looking for ways to improve the performance of the algorithm, we consider two options: (1) for each task, its versions are assigned to processors according to the largest computation time first strategy. In other words, the version with the largest computation time requirement is always assigned to the processor that is the least utilized. (2) The tasks are assigned to processors in the order of decreasing utilization. For this to work, the task set must be sorted first and then ***Algorithm 1*** run. Apparently, there are four ways to arrange the input data:

1. VD-TD: Versions of each task are sorted in Decreasing order of computation time (VD), and Tasks are sorted in the order of Decreasing utilization (TD).
2. VD: Versions of each task are sorted in Decreasing order of computation time (VD) only.
3. TD: Tasks are sorted in the order of Decreasing utilization (TD) only.
4. US: The task set is unsorted as it is randomly generated.

For the same set of inputs, the performance of ***Algorithm 1*** under both *RM* and *EDF conditions* is given in Figure 2 and Figure 3, respectively. The improvement of performance is quite significant. It is obvious that by assigning tasks to processors in the order of decreasing task utilization gives the best performance. What is surprising is that the order in which the versions of a

task is assigned to processors does not yield much performance.

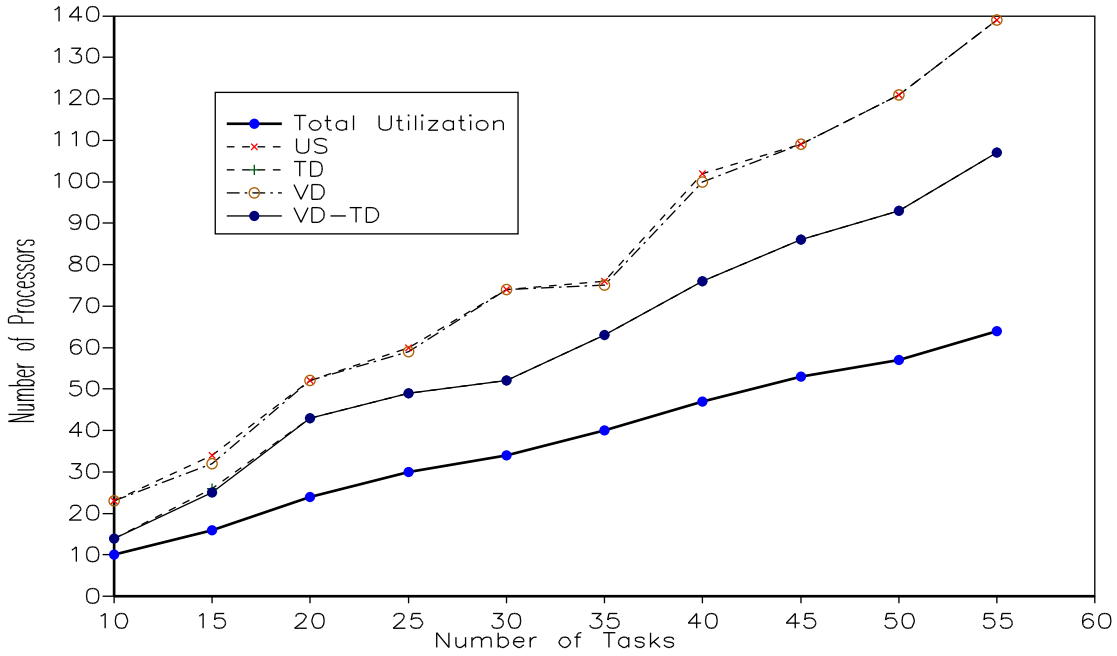

Figure 2: Performance of the First Algorithm under *RM condition*

Using the total utilization of the task set as a baseline for measuring the performance of the

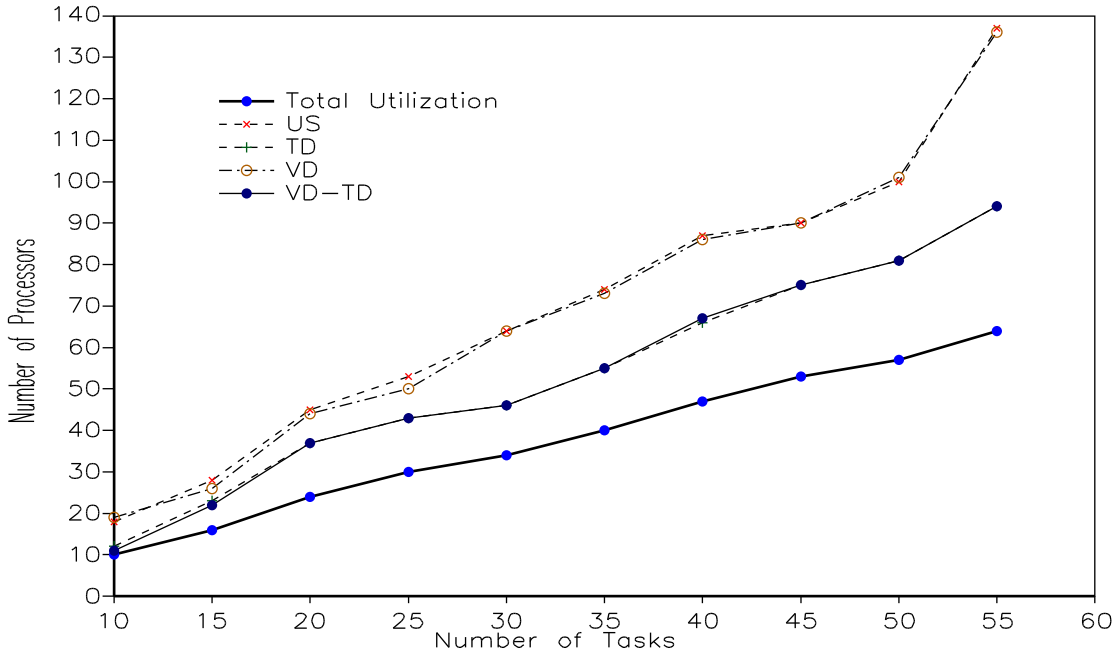

Figure 3: Performance of the First Algorithm under *EDF condition*

algorithm is less illuminating, since the optimal number of processors may differ from the total

utilization greatly in some cases, and little in other cases. The ideal solution is to find the optimal number of processors for those task sets. However, finding the optimal number of processors will require at least exponential time with respect to the number of tasks, since the *TFT* scheduling problem is at least *NP-complete*. In the following, we present a method to randomly generate task sets, such that, each of these task sets fully utilizes a known number of processors, using either the rate-monotonic algorithm or the EDF algorithm.

Given *m* number of processors, and the average number of task versions to be run on each processor, we generate a set of tasks that fully utilizes *m* processors, at the meantime satisfies the timing and fault-tolerant constraints of the tasks. Firstly, *m* arrays of random numbers are generated. Secondly, each item in an array is divided by the sum of all items in its array to obtain a number between 0 and 1, which later corresponds to the utilization of a version. Thirdly, for each task, a number *v*, which corresponds to the number of the versions is has, is randomly generated, confirming with the average as given. Then *v* number of computation times is randomly selected from the *m* arrays of numbers (corresponding to computation time). This process is repeated until all the items in the *m* arrays are picked. The periods of all tasks are chosen to be one. Since the release time of task is immaterial in periodic task scheduling, it is not considered in our experiment. A simple example is given below to illustrate the process of the random generation of task sets.

Example 1: Given that the number of processors is 4, the average number of versions a task has is 3, four arrays of numbers are randomly generated. The number of task versions per processor is also a random variable. In this example, it is chosen between a range of 1 and 6. The results of the first two steps are given in Figure 4. The final set of tasks is given as follows:

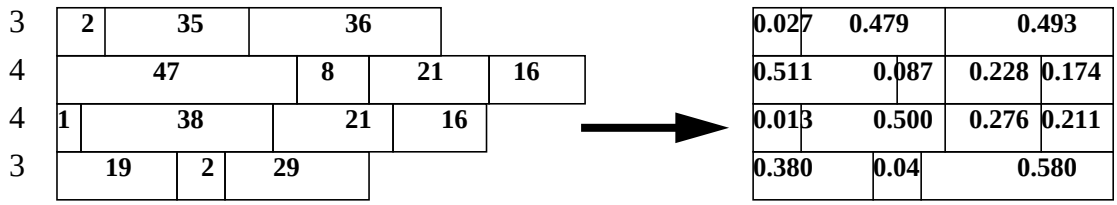

Figure 4: Random Generation of Task Sets

$\Re = \{\tau_1, \tau_2, \tau_3, \tau_4\}$

$\tau_1 = ((0.210, 0.027, 0.013, 0.479, 0.174), 1)$

$\tau_2 = ((0.276, 0.228, 0.493), 1)$

$\tau_3 = ((0.500), 1)$

τ_4 = ((0.511, 0.380, 0.087, 0.580, 0.040), 1)

The performance of ***Algorithm 1*** is given in Figure 5 and Figure 6, respectively. Here a different performance metric called "percentage of extra processors" is used. It is defined as $(N - N_0) / N_0$, where N is the number of processors computed by ***Algorithm 1***, and N_0 is the optimal number of processors.

The task sets used by ***Algorithm 1*** under the *RM* and *EDF condition*s are the same. Figure 5 and Figure 6 show again that the performance of ***Algorithm 1*** depends heavily on the order in which tasks are assigned to processors. The performance of the algorithm under *EDF condition* is expected to be better than that under *EDF condition*. The number of processors used under *EDF condition* is about 78% of that under *RM condition*, confirming with the fact that on the average, the *RM condition* provides bounds at 0.78. An interesting result of this experiment is that there is fluctuation in performance, implying that the algorithm is very sensible to input data.

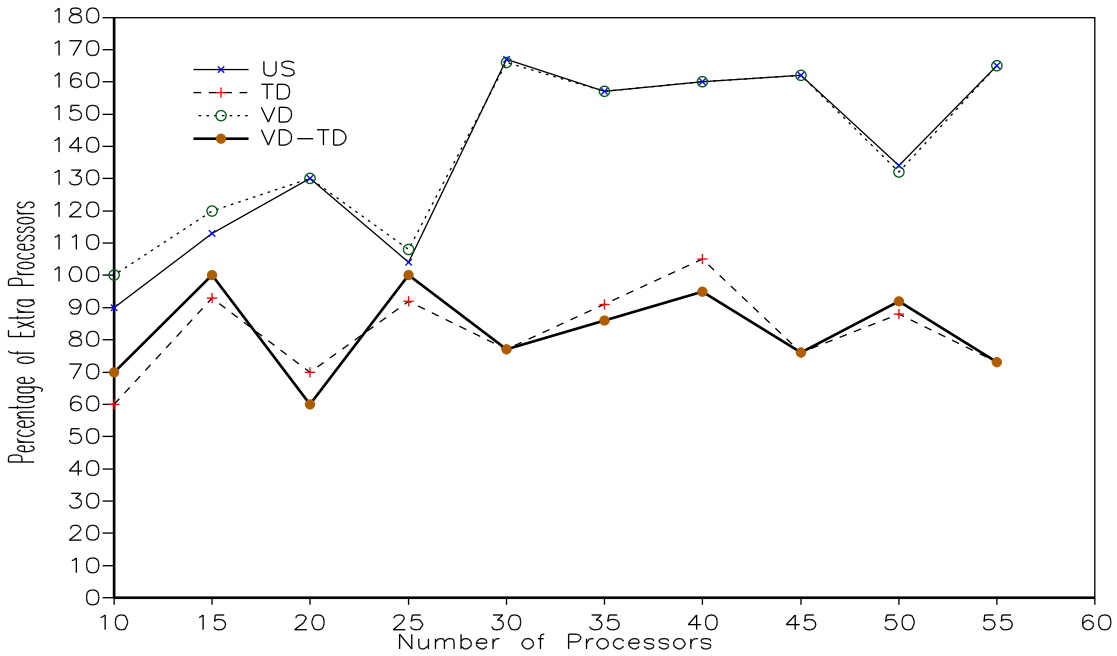

Figure 5: Performance of the First Algorithm under *RM condition* (Average versions/task = 3, Average Task Versions/processor in optimal assignment=5)

V. The Design of the Second Algorithm

The performance of ***Algorithm 1*** is somewhat out of our expectation, given the well-balanced result of *(9m) / (8 (m - r + 1))*, where *m* is the number of processors to be allocated, and *r* is the number of copies for each task, by Bannister and Trivedi [4]. The relatively poor performance may come from the fact that versions of a task may have different computation time requirements,

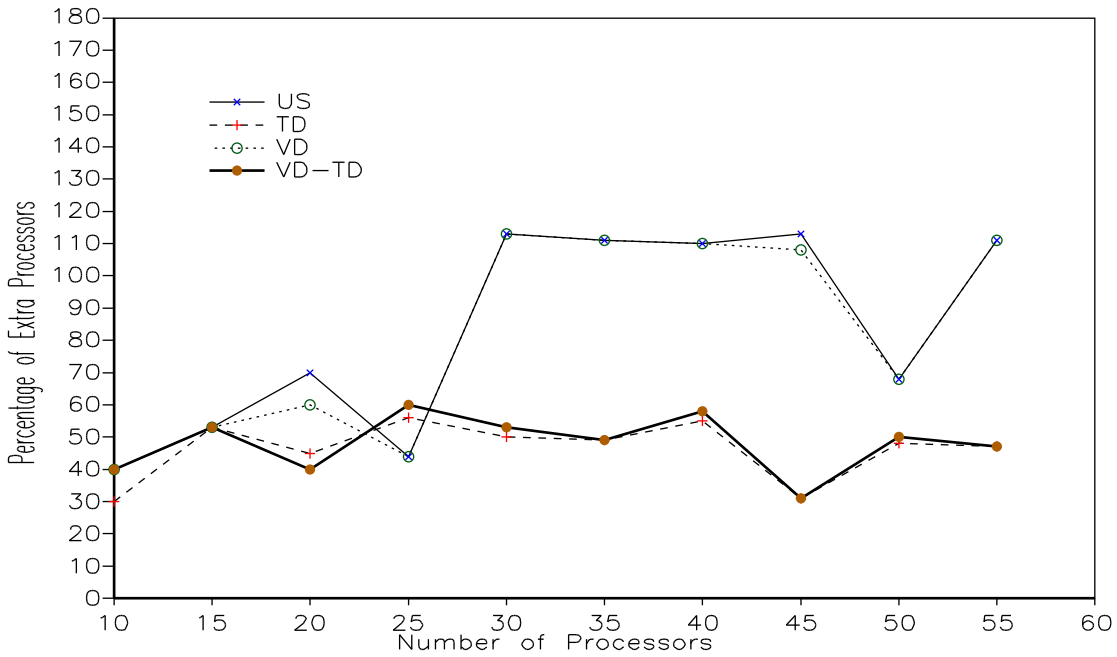

Figure 6: Performance of the First Algorithm under *EDF condition* (Average versions/task = 3, Average Task Versions/processor in optimal assignment =5)

contrasting with the assumption that clones of a task have the same computation time requirement, which was used by Bannister and Trivedi in obtaining their result. A new algorithm which uses a different scheduling strategy is thus developed to obtain better performance. This new algorithm schedules tasks in the similar manner as bin-packing.

Bin-packing algorithms [7] are a class of well-studied heuristic algorithms, which perform well for the assignment of items into fixed-size bins. The First-Fit (FF) and First-Fit-Decreasing (FFD) algorithms have very low asymptotic bounds. For FF, the asymptotic bound is 1.7, while for FFD, it is 11/9 [14]. Since each item in the bin-packing problem is independent of other items, it can be assigned to any bin. This does not confirm to our problem definition. In other words, none of the bin-packing algorithms can be directly used to solve the *TFT* scheduling algorithm. Modifications have to be made in order to make them work.

The new algorithm that is based on the bin-packing idea is given as follows:

Algorithm 2 (Input: task set $\Re$; Output: m)

1. Set $i = 1$ and $m = 1$. /* i denotes the ith task, m the number of processors allocated*/
2. (a) Set $l = 1$. /* l denotes the lth version of task i */

   (b) Set $j = 1$. If the lth version of task i together with the versions that have been assigned to processor P_j can be feasibly scheduled on P_j according to the *RM condition* (or *EDF condition*) for a single processor, and no version of task i has been

previously assigned to processor P_j, assign the lth version of task i to P_j. Otherwise, increment $j = j + 1$ and go to step 2(b).

(c) If $l > \kappa_i$, i.e., all versions of task i have been scheduled, then go to Step 3. Otherwise, increment $l = l + 1$, and go to Step 2(b).

3. If $j > m$, set $m = j$. IF $i > n$, i.e., all tasks have been assigned, then return. Otherwise increment $i = i + 1$ and go to step 2(a).

When the algorithm terminates, m is the number of processors required to execute the given set of tasks $\Re$. The correctness of this algorithm can be similarly argued as that of ***Algorithm 1***. The complexity of this algorithm is upper bounded by $O(\kappa nm)$.

For the same set of inputs as used to test the performance of ***Algorithm 1*** in Figure 1, the performance of ***Algorithm 2*** is given in Figure 7. The performance of ***Algorithm 1*** is also plotted in the same figure for comparison. Again, the task sets are not sorted. Obviously, there is a significant improvement in performance for ***Algorithm 2*** over ***Algorithm 1***. While ***Algorithm 1*** uses about 140% more processors under *RM condition* (also see Figure 5), ***Algorithm 2*** uses about 32% more processors than the optimal ones (also see Figure 8 below). ***Algorithm 2*** is a clear win over ***Algorithm 1***.

The performance of ***Algorithm 2*** with respect to input patterns is given in Figure 8 and Figure 9. It is interesting to note that ***Algorithm 2***, under either *RM condition* or *EDF condition*, is sensitive to neither the order in which tasks are assigned to processors, nor the order in which versions of a task are assigned to processors.

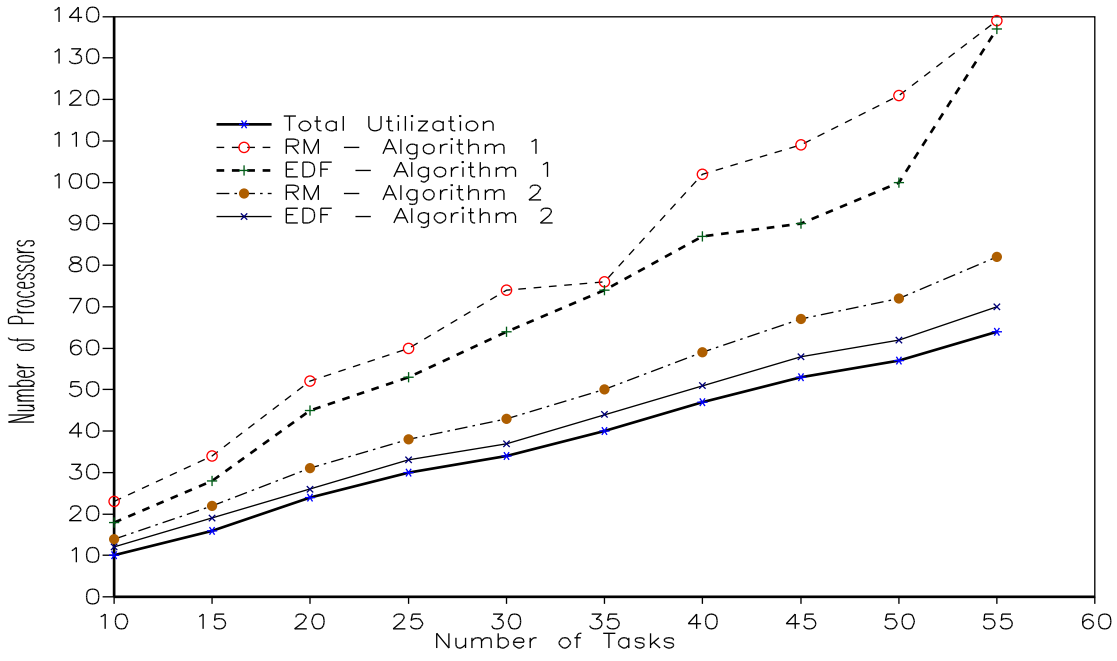

Figure 7: Performance of the Second Algorithm, compared with the First One

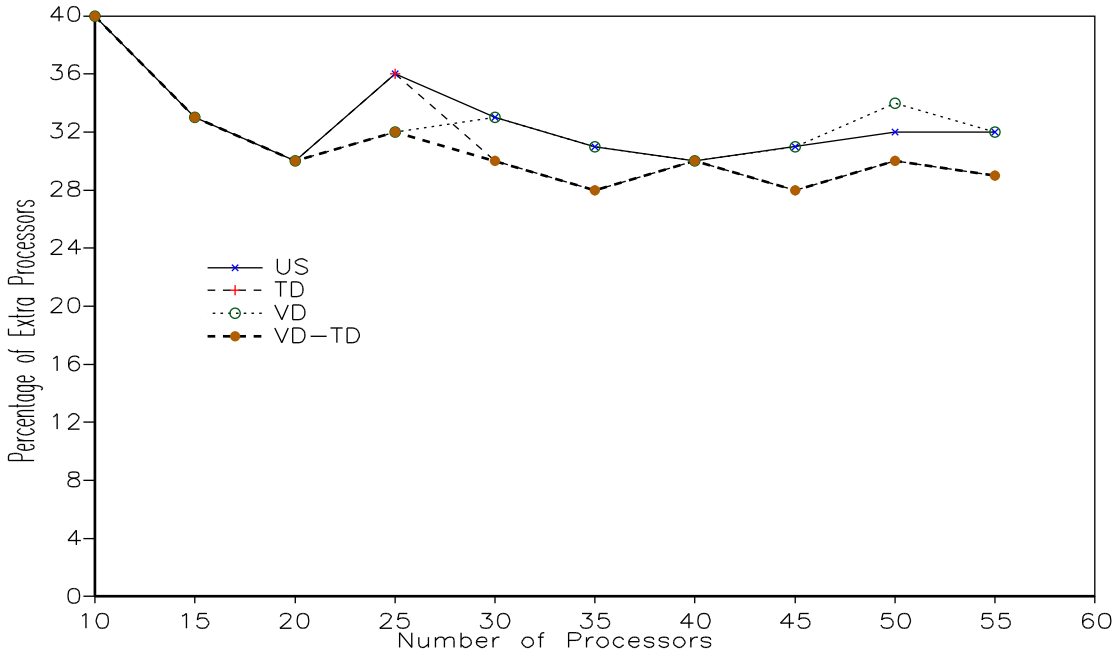

Figure 8: Performance of the Second Algorithm under *RM condition* (Average versions/task = 3, Average Task Versions/processor in optimal assignment=5)

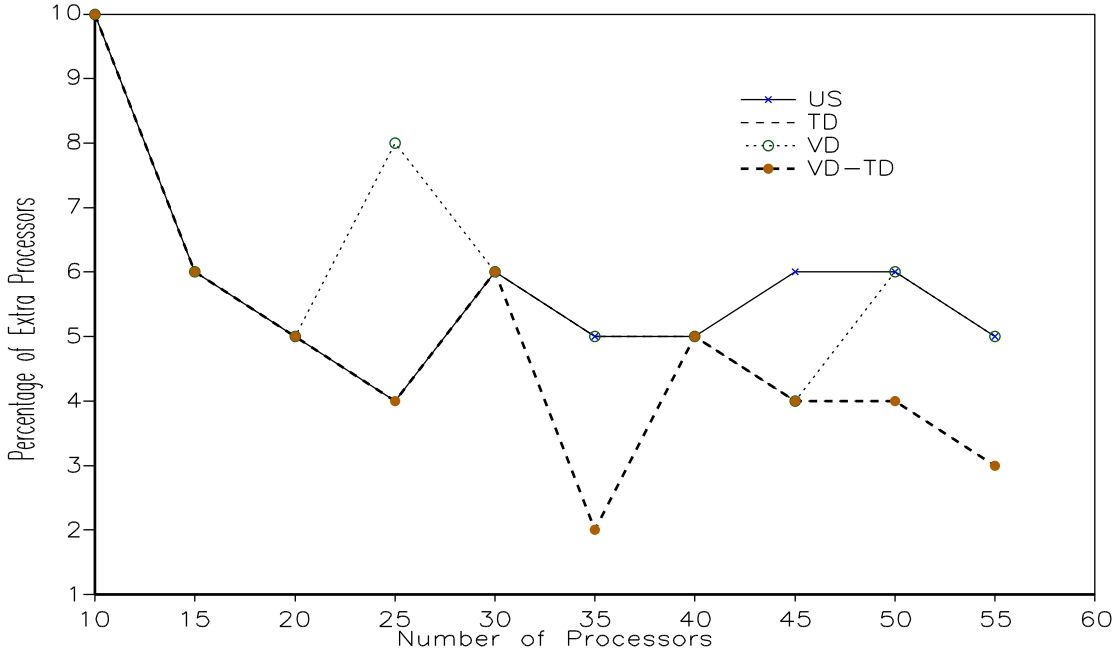

Figure 9: Performance of the Second Algorithm under *EDF condition* (Average versions/task = 3, Average Task Versions/processor in optimal assignment=5)

VI. Discussion

In the previous sections, we have developed two algorithms to solve the *TFT* scheduling problem. In all the simulated experiments, we have chosen the number of versions a task can have to be a random number within certain range, and the computation times of versions for a task to be also randomly distributed. In reality, there are situations where the number of versions a task can have is the same among all the tasks, or even more, all versions belonging to a task have the same computation time requirement (different tasks still have different computation time requirements). We have therefore also simulated both algorithms under these situations.

Figure 10 gives the performance of both algorithms with the number of versions per task is fixed at three, and the computation times of the versions are randomly generated. The performance of both algorithms — ***Algorithm 1*** and ***Algorithm 2*** is improved, while more improvement is gained by ***Algorithm 1***. Even more improvement is obtained when all tasks have the same number of versions, and all versions of a task have the computation time requirement. This is illustrated in Figure 11. The result obtained by Bannister and Trivedi [4] does make sense under these situations.

In summary, our simulation studies show that

1. The order of assigning tasks to processors affects the performance ***Algorithm 1*** significantly, regardless of which condition is used.
2. The order of assigning different versions of a tasks to processors does not have much impact on the performance of ***Algorithm 1***.
3. ***Algorithm 2*** is insensitive to the order of assigning tasks to processors and the order of assigning versions of a task to processors.
4. ***Algorithm 2*** outperforms ***Algorithm 1*** in all the experiments we have carried,
5. The performance of ***Algorithm 2*** is near-optimal. The 32% more processors used by the rate-monotonic algorithm is almost inevitable because of its decreasing bound $n\,(2^{1/n}-1)$, where n is the number of tasks.

The superiority of ***Algorithm 2*** is reflected not only in its out-performance over ***Algorithm 1,*** but most importantly, it can be used as an dynamic or on-line algorithm. The insensitivity property it has with regard to the order tasks and versions are assigned makes it even more attractive when it is used on-line.

We have mentioned in Section II that some of the assumptions in our problem statement can be relaxed, Assumption (G) is one of them. The strategy to schedule a task system which has both periodic and sporadic tasks, is to schedule the periodic tasks first to meet their deadlines, then the sporadic tasks. Several algorithms [18] [33] [34] have also been proposed to integrate sporadic or

aperiodic tasks into the scheduling of periodic tasks. They are Background, Polling, Priority Exchange, Extended Priority Exchange, Sporadic Server, and Deferrable Server. When multiple versions of sporadic tasks are used for fault-tolerance purpose, both algorithms we present here can be used with little modification.

There are several other cases that we like to consider in the near future. One case is how to schedule a set of periodic, multi-version tasks which synchronize in order to achieve "timely-fault-tolerance". Other cases include: (1) task deadlines are not equal to task periods; (2) urgent tasks have longer periods. The deadline transformation algorithms [29] can be used.

One of limitations of our results is that the precedence constraints of tasks are not considered. The reason for not incorporating precedence constraints into our problem definition is not because they do not exist in real world problems, but rather the difficulty of finding a solution to such a complicated problem as guaranteeing task deadlines and tolerating errors or failures simultaneously.

VII. Conclusion

Rate-monotonic scheduling algorithm has played a vital role in real-time scheduling. Because of its beauty to guarantee the deadlines of a set of periodic tasks, many scheduling results have been obtained, and many real-time operating systems have been built, using it as a "backbone". However, one of its major drawbacks to be used in mission critical applications is that it is not fault-tolerant. In this paper, we not only have presented a way to make a real-time system using rate-monotonic algorithm fault-tolerant, but also have developed two scheduling algorithms to achieve that. Simulation studies indicate that the second algorithm performs nearly optimal.

Our future work will focus on the extension of our current results to task sets that have simple precedence constraints, such as chain or tree.

References

[1] Audsley, N.C. "Deadline monotonic scheduling," Doctoral Thesis, Dept. Computer Science, University of York, 1990.

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

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

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

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

[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] Coffman, E.G. Jr., M.R. Garey, D.S. Johnson. “Approximate algorithms for bin packing - An updated survey,” In Algorithm Design for Computer System Design, G. Ausiello, M. Lucertini, and P. Serafini (Eds), Springer-Verlag, New York, 1985, pp. 49-106.

[8] Davari, S., and S.K. Dhall. “An on line algorithm for real-time tasks allocation,” IEEE Real-Time Systems Symposium, December 1986.

[9] Davari, S., and S.K. Dhall. “On a periodic real-time task allocation problem,” Proc. of 19th Annual International Confernce on System Sciences, 1986, pp. 133-141.

[10] Dhall, S.K., and C.L. Liu. “On a real-time scheduling problem,” Operations Research, Vol. 26, 1978, pp. 127-140.

[11] 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.

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

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

[14] Johnson, D.S. Near-Optimal Bin Packing Algorithms, Doctoral Thesis, MIT, 1973

[15] 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.

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

[17] 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.

[18] Lehoczky, J.P., L. Sha, and J.K. Stronider. “Enhanced aperiodic responsiveness in hard real-time environments,” IEEE RTSS, 1987, pp. 261-270.

[19] Lehoczky, J., L. Sha, and Y. Ding. “The rate monotonic scheduling algorithm: Exact characterization and average case behavior,” IEEE RTSS, 1989, pp. 166-171.

[20] Lehoczky, J.P. “Fixed priority scheduling of periodic task sets with arbitrary deadlines,” IEEE RTSS, 1990, pp. 201-209.

[21] 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.

[22] Liestman, A.L. and R.H. Campbell. “A fault tolerant scheduling problem,” IEEE Transac-

tions on Software Engineering, SE-12(11), November 1986, pp. 1089-1095.

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

[24] 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.

[25] 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.

[26] 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.

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

[28] Serlin, O. “Scheduling of time critical processes,” Proceedings of the Spring Joint Computers Conference 40, 1972, pp. 925-932.

[29] Sha, L., J.P. Lehoczky, and R. Rajkumar. “Solution for some practical problems in prioritized preemptive scheduling,” IEEE RTSS, 1986, pp. 181-191.

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

[31] Shih, W-K., J.W.S. Liu, and C.L. Liu. “Scheduling periodic jobs with deferred deadlines,” Report No. UIUCDCS-R-90-1583, University of Illinois, 1990.

[32] 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.

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

[34] Sprunt, B., J.P. Lehoczky, and L. Sha. “Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm,” IEEE RTSS, 1988, pp. 251-258.

[35] Sprunt, B. Aperiodic Task Scheduling for Real-Time Systems, Doctoral Thesis, Carnegie Melon University, 1990.

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

[37] 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.

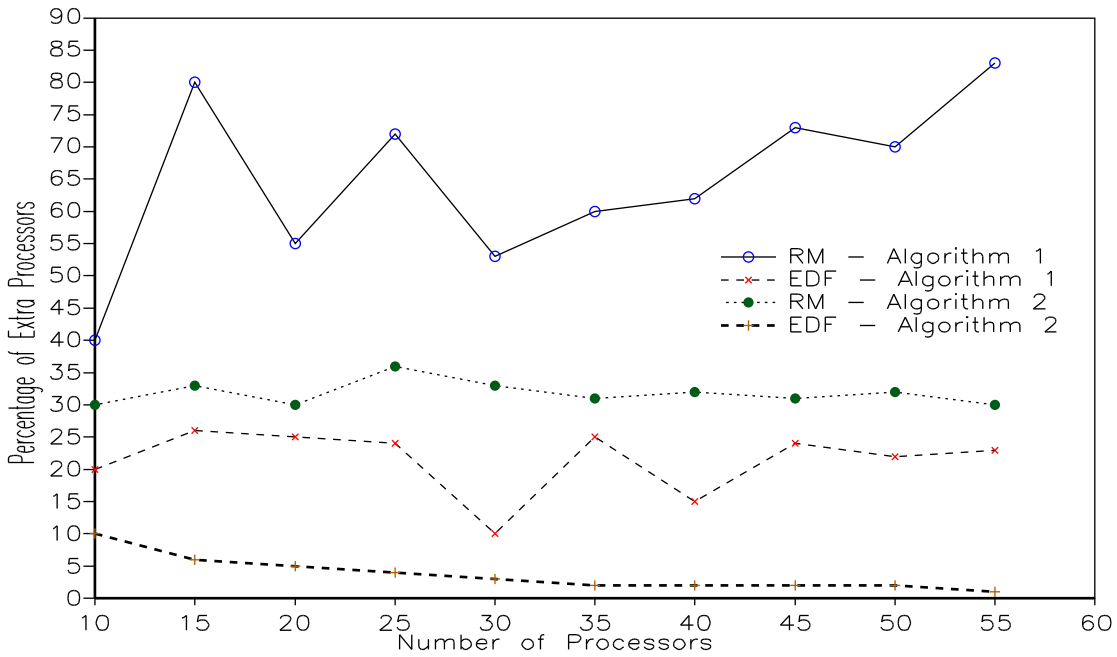

Figure 10: Performance of both Algorithms (Versions/task = 3, computation times of each version vary)

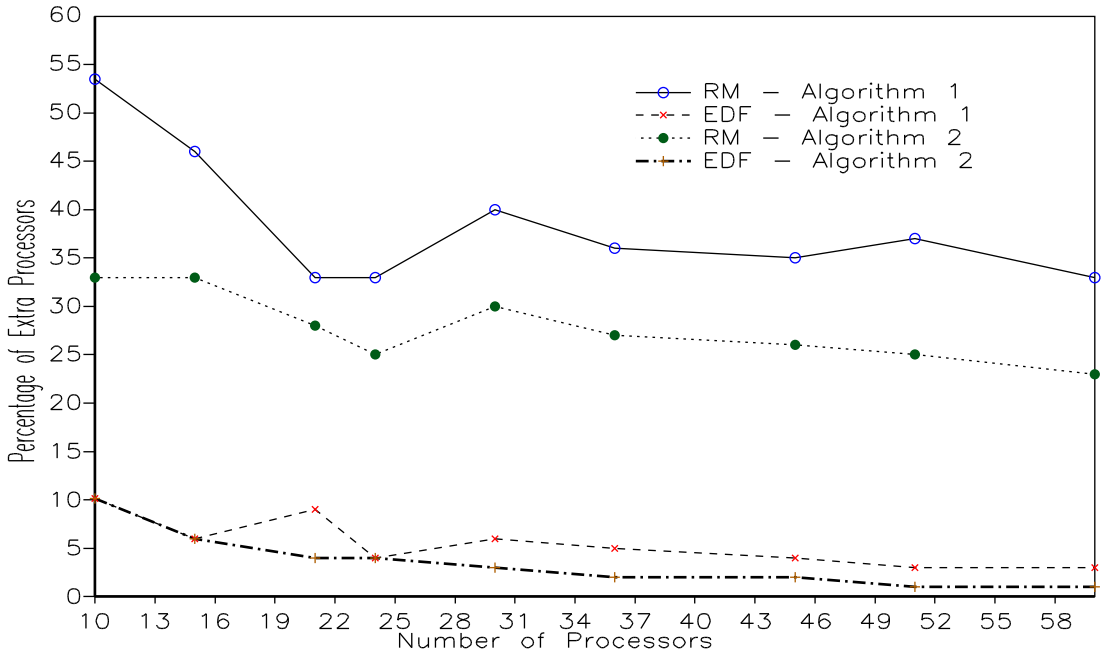

Figure 11: Performance of both Algorithms (Versions/task = 3, versions for a task have the same computation times)