A Processor-Efficient Scheme for Supporting Fault-Tolerance in Rate-Monotonic Scheduling

Yingfeng Oh and Sang H. Son

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract

We address the issue of supporting fault-tolerance in a real-time system, where the deadlines of periodic tasks are guaranteed by the Rate-Monotonic algorithm. The problem is stated as one to minimize the total number of processors required to execute a set of periodic tasks, each of which, for fault-tolerance purposes, has multiple versions. A simple but effective algorithm is proposed to solve the task allocation problem. The algorithm is shown to have superior worst-case and average-case performance.

I. Introduction

Rate-Monotonic (RM) scheduling [12, 16] is becoming a viable scheduling discipline for real-time systems. Through the years, researchers have successfully developed a host of scheduling techniques out of this discipline to solve many practical real-time problems, such as task synchronization, bus scheduling, joint scheduling of periodic and aperiodic tasks, transient overload, and parallel processing [14, 15]. The rate-monotonic scheduling discipline has been widely used in a number of applications. For example, it has been specified for use with software on board the Space Station as the means for scheduling multiple independent task execution; it will be built into the on-board operating system [6]. Many Ada compilers also support the scheduling discipline [15].

Rate-monotonic algorithm is the optimal algorithm for scheduling periodic tasks to meet their deadlines. It is optimal in the sense that if a set of periodic tasks can be feasibly scheduled by a fixed-priority algorithm, it can be scheduled by the rate-monotonic algorithm. In many mission-critical and life-critical applications, periodic tasks are often executed in a hard real-time environment, where missing a task deadline may result in catastrophic consequences. Rate-monotonic scheduling, with its ease of implementation and its optimality, is a powerful vehicle in guaranteeing task deadlines in such hard real-time environments. However, it can only do so as long as the processors never fail or the tasks never produce any erroneous results. Unfortunately, processors do fail in reality, especially in a system that support long duration of operations; and tasks do produce erroneous results, especially in a large complex system. In order to support the dependability or fault-tolerance of these real-time systems, we study the problem of guaranteeing task deadlines even in the presence of processor failures and task errors. In particular, we will show how to meet task deadlines even in the presence of processor failures or task errors in a periodic task system, where task deadlines are guaranteed by the rate-monotonic algorithm. The problem is formulated as an optimization problem and a simple algorithm is devised to solve it.

Much work has been done in addressing the fault-tolerance of real-time systems; but due to space limit, we mention only a few. Note that these papers address the issues under different assumptions and some of them are only remotely related to our work. Krishna and Shin 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 [9]. Unfortunately, their algorithm has the drawback that it is premised on the solution to two NP-complete problems. Balaji et al presented an algorithm to

dynamically distribute the workload of a failed processor to other operable processors [1]. 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 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 [2]. 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 obtained. However, their allocation algorithm does not consider the problem of minimizing the number of processors used, and the problem of how to guarantee task deadlines on each processor is not addressed. These are the important considerations that our work addresses.

We make the general assumption that for fault-tolerance purposes, a task has multiple versions, each of which must be executed on a different processor. The problem, dubbed the *Fault-Tolerant Rate-Monotonic Multiprocessor Scheduling* (or FT-RMMS), is stated as one that the minimum number of processors is used to guarantee task deadlines and copies or versions of each task are assigned to different processors for fault-tolerance purposes. While the need to guarantee task deadlines is well known for a real-time system, the importance of minimizing the total number of processors is often not fully understood. First, using more processors will increase the probability of having more processor failures. Second, using more processors will affect the cost, weight, size, and power consumption of the whole system, the increase of any of which may jeopardize the success of an entire operation. For certain applications of real-time systems, such as aerospace and submarine control, these factors could be very important.

Although the process to assign versions of a task to different processors could be straightforward, it is non-trivial to minimize the number of processors used to accommodate them. In fact, the simplest version of the FT-RMMS problem, which is the *Rate-Monotonic Multiprocessor Scheduling* (or RMMS) problem, has been proven to be NP-complete [10]. Hence, any practical solution to the problem presents a trade-off between computational complexity and performance. It has been shown that heuristic algorithms can deliver near-optimal solutions to NP-complete problems with limited computational overhead [7, 14].

Although a number of heuristic algorithms have been proposed to solve the RMMS problem [3, 4, 5, 14], there is only one solution to the FT-RMMS problem so far. A heuristic algorithm, called FT-RM-FF, is developed in [13]; it applies a variant of the bin-packing heuristic, the First-

Fit, to assign tasks to processors. The algorithm is a dynamic or on-line one, and its worst-case performance is shown to be upper bounded by 2.33, i.e., the maximum ratio between the number of processors required to schedule a set of tasks and the minimum number of processors required to schedule the same task set is 2.33 for FT-RM-FF.

Our approach for developing a heuristic algorithm for FT-RMMS is quite different from the previous one. Rather than increasing the level of sophistication of the bin-packing heuristic, we base our new scheme on two novel ideas: a divide-and-conquer strategy and a tighter schedulability condition. The divide-and-conquer strategy allows us to separate the requirements of ensuring versions of a task be assigned to different processors and guaranteeing the tasks assigned to a processor meet their deadlines. The schedulability condition allows us to assign more tasks to each processor by enabling re-ordering of the tasks for assignments. The combination of these two ideas results in a scheme called FT-RM-NF, which is superior in the worst-case and average-case performance, and fast in assigning tasks to processors. FT-RM-NF has a time complexity of $O(\kappa n \log n)$ and a worst-case bound of 2 and $1/(1-\alpha)$ for $\alpha < 1/2$, where α is the maximum allowable utilization of any version among all tasks, n is the total number of tasks, and κ is the maximum number of versions a task can have. In most practical applications, α is usually small, i.e., $\alpha < 1/2$, thus the worst-case performance bound is much better than 2. Simulation results show that FT-RM-NF uses less than 10% extra processors when $\alpha \leq 0.2$.

The rest of the paper is organized as follows: problem formulation and background information are described in the next section, which includes the definition of the problem and the criteria under which the performance of the heuristic algorithm is measured. In section III, we present the FT-RM-NF algorithm and analyze its performance, followed by the empirical results on the performance of FT-RM-NF in section IV. We conclude in section V with a discussion of the limitations and contributions of this work.

II. Problem Formulation

We assume that processors are homogeneous, i.e., they have identical speed. For a set of n tasks, we assume:

(A) Each task has κ_i versions, where κ_i is a natural number. The κ_i versions of a task may have different computation time requirements, and the κ_i versions may be merely copies of one implementation or truly versions of different implementations.

(B) All versions of a task must be executed on different processors for fault-tolerance pur-

poses.

(C) The requests of all tasks are periodic with constant intervals between requests. 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 the sense that the requests of a task do not depend on the initiation or the completion of requests for other tasks.

For a task τ with a period of T, the request of task τ comprises the readiness of task τ for execution and its CPU request. If the initial release time of task τ is R, then the task will arrive in the system at time $R + kT$, where k is a positive integer, and we say that there is a request from task τ every T time units for execution. According to assumption (D), a request from task τ at time instance $(R + kT)$ must be completed before the next request of the same task arrives, i.e., before time instance $[R + (k + 1)T]$, where k is a positive integer. A task meets its deadline if all requests of all its versions meet their deadlines.

Note that assumptions (C), (D), and (E) are the same as those in the task model for rate-monotonic scheduling; they represent a simplified model for most practical real-time applications. As we have noted above, much work has been done in successfully generalizing the original model for rate-monotonic scheduling.

Assumptions (A) and (B) make a rather general statement about the redundancy schemes used by each task. The term “version” has been used in N-version programming 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 mere copies of a single task version. In the case of using merely duplicated copies, the errors produced by a task cannot be tolerated, since all the versions, or merely duplicated copies, produce the same results. But processor failures can be tolerated by using mere copies of a task and executing them on different processors. To tolerate task errors, different versions of a task need not execute on different processors. If the employment of multiple versions for a task is for the tolerance of task errors only, we can treat all its versions as independent tasks in this model. Here we are not concerned with details about what faults are to be tolerated or how faults are tolerated, rather we make the general statement that for fault-tolerance purposes, each task has a number of versions that must be executed on different processors. Note that the number of versions used by each task can be different, depending on the applications; the smallest is one (ver-

sion) and the largest is κ. In other words, κ_i may assume a different value, ranging from 1 to κ.

Since the computation time requirement of a request for each version may be different, we use the worst-case estimate as the computation time requirement for each version in making scheduling decisions. The time to switch a processor from one task to another is assumed to be zero, but in practice, the switching time can be included in the worst-case time estimate.

We say that a set of tasks is feasible if it can be scheduled by some algorithms such that all task deadlines are met. If a set of periodic tasks can be feasibly scheduled on a single processor, then the *Rate-Monotonic* (*RM*) [12] or *Intelligent Fixed Priority* algorithm [16] is optimal, in the sense that if a set of periodic tasks is feasible with a fixed-priority algorithm, then it is feasible with the RM algorithm. The RM algorithm assigns priorities to tasks according to their periods, where the priority of a task is in inverse relationship to its period. In other words, a task with a shorter period is assigned a higher priority. The execution of a low-priority task will be preempted if a high-priority task arrives. Since each task has a potentially infinite number of requests, it would be rather time-consuming to manually check that each request finishes before its corresponding deadline. Fortunately, Liu, Layland, and others have provided us with simple schedulability conditions that we can use for that purpose. Because RM is the best fixed-priority algorithm in the same sense as its optimality, and its ease of implementation due to the fixed-priority manner, we will use it in guaranteeing task deadlines on each processor. The FT-RMMS problem can thus be formulated as follows

Fault-Tolerant-Rate-Monotonic-Multiprocessor-Scheduling (FT-RMMS) Problem: A set of n tasks $\Sigma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is given with $\tau_i = ((C_{i,1}, C_{i,2}, \ldots, C_{i,\kappa_i}), R_i, D_i, T_i)$ for $i = 1, 2, \ldots, n$, where $C_{i,1}, C_{i,2}, \ldots, C_{i,\kappa_i}$ are the computation times of the κ_i versions of task τ_i. R_i, D_i, and T_i are the release time, deadline, and period of task τ_i, respectively. The question is to schedule the task set Σ using the least number of processors such that all task deadlines are met and all versions of a task execute on different processors.

An optimal algorithm is the one that always uses the minimum number of processors to execute any given task set. According to assumption (D), the deadline of each task coincides with its next arrival. For periodic task scheduling, it has been proven [12] that the release times of tasks do not affect the schedulability of the tasks. Therefore, release time R_i and deadline D_i can be safely omitted when we consider solutions to the problem. Let $u_{i,j} = C_{i,j} / T_i$ be the utilization (or load) of the jth version of task τ_i and $u_i = \sum_{j=1}^{\kappa_i} C_{i,j}/T_i$ be the utilization (or load) of task τ_i. Define

$\kappa = \max_{1 \le i \le n} \kappa_i$ and $\alpha = \max_{i,j} (C_{i,j} / T_i)$, i.e., α is the maximum allowable utilization of any version among all tasks.

The general solution to such a problem as FT-RMMS or RMMS involves two algorithms: one to schedule tasks assigned on each individual processor, and the other to assign tasks to the processors. Typically, the task assignment schemes apply variants of well-known bin-packing heuristics where the set of processors is regarded as a set of bins. The decision whether a processor is full is determined by a schedulability condition. Unlike bin-packing, the schedulability conditions for RM scheduling on a single processor are usually non-linear functions of task utilizations; therefore, more complexity is involved in designing and analyzing heuristics for these problems.

The performance of a task assignment algorithm is often evaluated by providing worst-case bounds, i.e., finding the maximum ratio $N_A(\Sigma) / N_0(\Sigma)$ across all task sets, where $N_A(\Sigma)$ (or just N_A) and $N_0(\Sigma)$ (or just N_0) are the number of processors required by the heuristic A and the optimal number of processors required to schedule a given task set Σ, respectively. Worst-case bounds are determined by

$$\Re_A = \inf \left\{ r \ge 1 : \frac{N_A(\Sigma)}{N_0} \le r \text{ for all task sets } \Sigma \right\}.$$

A slightly different measurement called the asymptotic performance bound, which is more of theoretical interest, can also be used:

$$\Re_A^{\infty} = \inf \left\{ r \ge 1 : \text{for some } N \in Z^+, N_0 \ge N, \frac{N_A(\Sigma)}{N_0} \le r \text{ for all task sets } \Sigma \right\}$$

It is apparent that the smaller the $\Re_A$'s value is, the better the heuristic algorithm A performs in terms of the worst-case scenario. In other words, the smaller the $\Re_A$'s value is, the closer the heuristic solution is to the optimal one. Hence, we want to minimize $\Re_A$ as much as possible when we design a heuristic algorithm. For our particular problem of FT-RMMS, we derive a bound that is stricter than $\Re_A$; it is the maximum ratio between the number of processors required by the heuristic A to schedule a given task set Σ and the total utilization (or load) of the task set Σ. Since the optimal number of processors required to execute a task set must be no smaller than its total utilization, $\Re_A$ can be immediately obtained as well. Obtaining the bound with regard to the total utilization of a task set provides us with more information, since the optimal number of processors cannot be found in reality when the task set is reasonably large, while the total load of a task set can always be calculated.

As it was previously noted, in order to tolerate processor failures or task errors, redundant

processors need to be introduced. In general, using more processors will increase the fault-tolerance of a system. However, there is a limit to which adding more processors will decrease the overall fault-tolerance of the system, since using more processors increases the probability of having more processor failures. An interesting question to which a system designer would want to know the answer is the trade-off between the number of redundant processors used and the fault-tolerance (e.g., reliability and availability) of the system. Though we cannot answer this question completely in this work (since it requires application-specific information), we provide a partial answer to it by proposing a heuristic algorithm to schedule the tasks such that we can put bounds on the total number of processors used for a certain task set in terms of the maximum degree κ of task redundance and the total utilization of the task set. The current bounds are the lowest (hence the best) ones for this problem. Without such bounds, it would be impossible to calculate the fault-tolerance (e.g., reliability and availability) of the system.

III. The Design and Analysis of FT-Rate-Monotonic-Next-Fit

To solve the FT-RMMS problem, we need to address two issues: the scheduling on each processor and the assignment of tasks to processors. In general, the performance of a heuristic algorithm to solve this problem depends on the strategies taken to solve these two issues. One simple solution to the problem will be to use one processor for the execution of each task version. Although this solution guarantees that each request of each version of a task meets its deadline and that versions of a tasks be assigned to different processors for fault-tolerance purposes, it is very inefficient in processor usage. As we have argued before, our goal is to use as few processors as possible to accommodate a given set of tasks. In the following, we will first present the schedulability condition that will be used in FT-RM-NF and then the strategy to assign tasks to processors.

A set of tasks $\Sigma = \{\tau_i = (C_i, T_i) \mid i = 1, 2, \ldots, n\}$ is given to be scheduled on a single processor. Liu and Layland provides us with a schedulability condition that if $\sum_{i=1}^{n} C_i/T_i \leq n\left(2^{1/n} - 1\right)$, then all the n tasks can be scheduled to meet their deadlines by the RM algorithm. We would like to use this condition in determining the feasibility of a set of tasks, but the performance of the heuristics using this condition in solving the RMMS problem is not as good, as shown by the work done in [4, 5, 14]. This is because the condition is a worst-case one. There are some task sets that are feasible with the RM algorithm, but cannot be determined to be feasible by the condition.

A necessary and sufficient condition has been found for the RM algorithm [8, 11]. It is appar-

ent that the upper bounds of the performance of heuristic algorithms using the necessary and sufficient condition are not higher than those of the algorithms using any sufficient condition. However, the time complexity in using the necessary and sufficient condition for the RM algorithm is data-dependent. Due to the stringent requirements of hard real-time systems, it is not practical to employ the necessary and sufficient condition in general for these systems. Hence, heuristic algorithms using simple sufficient conditions are often sought in the solution of such problems as FT-RMMS.

In the following, we will be using a new sufficient schedulability condition for the scheduling of tasks; it is given in Theorem 1, and in a simpler form, in Corollary 1 [3].

Theorem 1: *Let* $\{\tau_i = (C_i, T_i) \mid i = 1, 2, \ldots, n\}$ *be a set of n tasks. Define*

$$V_i = \log_2 T_i - \lfloor \log_2 T_i \rfloor \text{ for } i = 1, 2, \ldots, n. \quad (1)$$

Then sort the V_i*s in the order of increasing value and rename them to be* V_i *for* $i = 1, 2, \ldots, n$. *If* $\sum_{i=1}^{n} C_i/T_i \le \sum_{i=1}^{n-1} 2^{V_{i+1}-V_i} + 2^{1+V_1-V_n} - n$, *then the task set can be feasibly scheduled by the RM algorithm.*

This condition is superior to Liu and Layland's condition in performance, since it always yields a processor utilization no lower than that by the Liu and Layland's condition [12]. It is also superior to the necessary and sufficient condition in time complexity, since the former runs only in time linear to the number of tasks, while the time complexity of the latter is data-dependent and is at least linear to the number of tasks. The higher processor utilization is achieved by this new condition, since it uses more information from the task set. The new condition, which is referred to as the Period-Oriented (or PO) condition, takes into account the relative values of task periods T_i, and it is derived under the worst-case situation in term of C_i, while Liu and Layland's condition does not consider T_i and C_i specifically. Under the worst-case situation of T_i, the new condition degrades to the worst-case condition, i.e., $\sum_{i=1}^{n} C_i/T_i \le n\left(2^{1/n} - 1\right)$. The proof of the theorem can be found in [14].

It is obvious that for each V value, $0 \le V_i < 1$. We can simplify the PO condition by defining $\beta = V_n - V_1$, where V_n is the largest and V_1 the smallest among all the V values (since they are sorted). By this definition, we immediately obtain the following corollary.

Corollary 1: *Let* $\{\tau_i = (C_i, T_i) \mid i = 1, 2, \ldots, n\}$ *be a set of n tasks and* V_i *be defined as in (1). Define* $\beta = \max_{1 \le i \le n} V_i - \min_{1 \le i \le n} V_i$. *If* $\sum_{i=1}^{n} C_i/T_i \le (n-1)\,(2^{\beta/(n-1)} - 1) + 2^{1-\beta} - 1 \le \max\{\ln 2, 1 - \beta \ln 2\}$, *then the task set can be feasibly scheduled by the RM algorithm.*

In the above condition, it is apparent that the smaller the value of β is, the higher the processor utilization is, since the function $f(\beta) = \max\{\ln 2, 1 - \beta \ln 2\}$ increases as β decreases. In order to achieve higher processor utilization, we shall try, in the design of a new scheme, to minimize β as much as possible by assigning tasks with V values close to each other.

Now that we have a tighter schedulability condition, we need to find a better way to assign tasks to processors. Since versions of a task must be assigned to different processors, we observe that a simple way to separate the concerns of ensuring versions of a task being assigned on different processors and using the least number of processors is to divide the processors into κ classes. To simplify the matter of presentation, we assume without loss of generality that each version of a task is numbered from 1 to $\kappa_i \le \kappa$. The processors are also divided into κ classes. The ith version of a task will be assigned to a processor in the ith class. Since the utilization bound increases as β decreases in the PO condition, a natural way to minimize the number of processors is to first sort the tasks in the order of increasing or decreasing V values and then schedule them using the PO condition. In such a way, we assign the tasks having close V values together, i.e., a small β value, thus increasing the total utilization of a processor. The FT-RM-NF algorithm is described as follows:

Fault-Tolerant-Rate-Monotonic-Next-Fit (FT-RM-NF) (Input: task set Σ; Output: m)

(1) Sort the task set such that $0 \le V_1 \le \ldots \le V_n < 1$.

(2) $i := 1$; $N_j := 1$ for $j = 1, \ldots, \kappa$;

(3) To schedule the κ_i versions of task τ_i, assign the jth ($1 \le j \le \kappa_i$) version of task τ_i to processor P_{j, N_j} in the jth processor class if this version together with the versions that have been assigned to P_{j, N_j} can be feasibly scheduled on it according to the PO condition, i.e.,
$U_{j, N_j} + u_{i, j} \le \max\{\ln 2, 1 - \beta_j \ln 2\}$ where $\beta_j = V_i - S_{j, N_j}$.
If not, assign the version to $P_{j, N_j + 1}$ and $N_j := N_j + 1$, $S_{j, N_j} := V_i$.

(4) If $i < n$, then $i := i + 1$ and go to (3); else stop.

The initial value of S_{j, N_j} is given by the first version assigned to a processor in the jth processor class. The number of versions assigned to each processor class may be different because each task may have different number of versions. U_{j, N_j} denotes the total utilization of the versions that have been assigned to processor P_{j, N_j}. The Next-Fit strategy is used to assign versions to processors within a processor class. The sorting process takes $O(n \log n)$ and the assignment process

takes $O(\kappa n \log n)$. Since $\kappa \geq 1$, FT-RM-NF runs at $O(\kappa n \log n)$ and hence it is an effective algorithm. The algorithm is apparently a static (or off-line) one.

Though this algorithm is quite simple, it can deliver very efficient worst-case performance, as shown by the following theorem. It performs even better as $\alpha = \max_{i,j}(C_{i,j}/T_i)$ decreases. It may also be true that the worst-case performance of other algorithms increases as α decreases, but the increase in the performance of FT-RM-NF is very rapid, as shown below.

Theorem 2: *Let N be the number of processors required by FT-RM-NF to feasibly schedule a given set of tasks with a total utilization of* $U = \sum_{j=1}^{n} u_j$ *and* $\alpha = \max_{i,j}(C_{i,j}/T_i)$. *If* $\alpha \geq 1/2$, *then* $N \leq 2U + \kappa(1 + \ln 2)$. *If* $\alpha < 1/2$, *then* $N < U/(1-\alpha) + \kappa[1 + \ln 2/(1-\alpha)]$.

Proof: In the completed FT-RM-NF schedule, let us first establish the relationship between the N_l processors used in the lth processor class and the total utilization of all the versions assigned to these N_l processors, i.e., between N_l and $U_l = \sum_{j=1}^{N_l} U_{l,j}$ for $l = 1, \ldots, \kappa$. Then we will combine the relationships together to obtain the final result.

For the jth processor $P_{l,j}$ of the lth processor class, let $\tau_{l,j,1}, \tau_{l,j,2}, \ldots, \tau_{l,j,s_j}$ be the $s_{l,j}$ versions (tasks) that are assigned to it and $U_{l,j} = \sum_{k=1}^{s_{l,j}} u_{l,j,k}$ for $j = 1, \ldots, N_l$, where $u_{l,j,k}$ denotes the utilization of the kth version (task) assigned to processor $P_{l,j}$. Furthermore, let $V_{l,j,i}$ be the V value of task $\tau_{l,j,i}$ and $\beta_{l,j} = V_{l,j+1,1} - V_{l,j,1}$. Then $\sum_{j=1}^{N_l} \beta_{l,j} \leq 1$. According to FT- RM-NF, we have

$$\sum_{k=1}^{s_{l,j}} u_{l,j,k} + u_{l,j+1,1} > \max\{\ln 2, 1 - \beta_{l,j} \ln 2\} \geq 1 - \beta_{l,j} \ln 2 \qquad (2)$$

for $j = 1, \ldots, (N_l - 1)$. Since $U_{l,j+1} \geq u_{l,j+1,1}$, we have

$$U_{l,j} + U_{l,j+1} \geq 1 - \beta_{l,j} \ln 2 \qquad (3)$$

from (2), where $j = 1, \ldots, (N_l - 1)$. Summing up the $(N_l - 1)$ inequalities in (3) yields

$$2\sum_{j=1}^{N_l} U_{l,j} - U_{l,1} - U_{l,N_l} \geq (N_l - 1) - \ln 2 \sum_{j=1}^{N_l - 1} \beta_{l,j} \geq (N_l - 1) - \ln 2$$

since $\sum_{j=1}^{N_l - 1} \beta_{l,j} \leq \sum_{j=1}^{N_l} \beta_{l,j} \leq 1$. In other words, $2\sum_{j=1}^{N_l} U_{l,j} > (N_l - 1) + U_{l,1} + U_{l,N_l} - \ln 2 \geq N_l - 1 - \ln 2$.

Summing up all κ classes of processors yields

$$2\sum_{l=1}^{\kappa} \sum_{j=1}^{N_l} U_{l,j} \geq \sum_{l=1}^{\kappa} N_l - \kappa(1 + \ln 2)$$

Since $N = \sum_{l=1}^{\kappa} N_l$ and $\sum_{l=1}^{\kappa} \sum_{j=1}^{N_l} U_{l,j} = \sum_{j=1}^{n} u_j = U$, i.e., the total utilization of all the processors is equal to the total utilization of the task set, we have

$$N \le 2U + \kappa(1 + \ln 2).$$

If $\alpha = \max_{i,j} (C_{i,j}/T_i)$, then $\alpha \ge u_{l,j+1,1}$ and

$$U_j + \alpha > 1 - \beta_j \ln 2 \tag{4}$$

from (2), where $j = 1, \ldots, (N_l - 1)$. Summing up the $(N_l - 1)$ inequalities in (4) yields $\sum_{j=1}^{N_l} U_{l,j} + (N_l - 1)\,\alpha > (N_l - 1) - \ln 2 \sum_{j=1}^{N_l - 1} \beta_{l,j} > (N_l - 1) - \ln 2$, since $\sum_{j=1}^{N_l - 1} \beta_{l,j} \le \sum_{j=1}^{N_l} \beta_{l,j} \le 1$.

In other words, $\sum_{j=1}^{N_l} U_{l,j} - \alpha > N_l (1 - \alpha) - 1 - \ln 2$. Then

$$N_l < \frac{1}{1-\alpha} \sum_{j=1}^{N_l} U_{l,j} + \left(1 + \frac{\ln 2}{1-\alpha}\right)$$

Summing up all κ classes of processors yields

$$\sum_{l=1}^{\kappa} N_l < \frac{1}{1-\alpha} \sum_{l=1}^{\kappa} \sum_{j=1}^{N_l} U_{l,j} + \kappa\left(1 + \frac{\ln 2}{1-\alpha}\right)$$

Since $N = \sum_{l=1}^{\kappa} N_l$ and $\sum_{l=1}^{\kappa} \sum_{j=1}^{N_l} U_{l,j} = \sum_{j=1}^{n} u_j = U$, we have

$$N < \frac{U}{1-\alpha} + \kappa\left(1 + \frac{\ln 2}{1-\alpha}\right).$$

Hence, we have proven the theorem. ■

Since the optimal number of processors required to execute a task set is no smaller than the total utilization of the task set, i.e, in term of the above notations, $N_0 \ge \sum_{j=1}^{n} u_j$, we have the following corollary immediately.

Corollary 2: *Let N and N_0 be the number of processors required by FT-RM-NF and the minimum number of processors required to feasibly schedule a given set of tasks, respectively. Define* $\alpha = \max_{i,j} (C_{i,j}/T_i)$. *If* $\alpha \ge 1/2$, *then* $N \le 2N_0 + \kappa(1 + \ln 2)$. *If* $\alpha < 1/2$, *then* $N/N_0 < 1/(1-\alpha) + [1 + \ln 2/(1-\alpha)]\,\kappa/N_0$.

Having proven the upper bounds for the FT-RM-NF algorithm, we next show that the upper bounds are tight, asymptotically speaking.

Theorem 3: $\Re^{\infty}_{\text{FT-RM-NF}} = 2$. $\Re^{\infty}_{\text{FT-RM-NF}}(\alpha) = \frac{1}{1-\alpha}$ *for* $\alpha < 1/2$.

Proof: Since $N \le 2N_0 + 1 + \ln 2$ for $\kappa = 1$, $\Re^{\infty}_{\text{FT-RM-NF}} \le 2$ according to the definition. Since

$\frac{N}{N_0} < \frac{1}{1-\alpha} + \left(1 + \frac{\ln 2}{1-\alpha}\right)\frac{1}{N_0}$ for $\kappa = 1$ and $\alpha < 1/2$, we have $\Re^{\infty}_{\text{FT-RM-NF}}(\alpha) \le \frac{1}{1-\alpha}$. We need to prove that when $\kappa = 1$, $\Re^{\infty}_{\text{FT-RM-NF}} = 2$ and $\Re^{\infty}_{\text{FT-RM-NF}}(\alpha) = \frac{1}{1-\alpha}$ for $\alpha < 1/2$. Then we can conclude that the bounds are tight.

To prove that the above bounds are tight, we need only to construct task sets that require the upper-bounded numbers of processors when they are scheduled by RM-FF-NF.

Let $n = 4k$ where k is a positive integer and ε be an arbitrarily small number such that $0 < \varepsilon << 1/n$. Furthermore, define $\delta > 0$ such that $2^{n\delta} < 1 + \varepsilon$.

Then for the first bound, the set of n tasks $\Sigma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is constructed as follows:

$$\tau_i = (C_i, T_i) = (1/2, 2^{i\delta}) \text{ for } i = 2j \text{ and } j = 0, 1, \ldots, 2k-1;$$

$$\tau_i = (C_i, T_i) = (\varepsilon, 2^{i\delta}) \text{ for } i = 2j+1, j = 0, 1, \ldots, 2k-1.$$

Since $V_{i+1} - V_i = \delta$, the tasks are in the order of increasing V values.

We first claim that $2k$ processors are required to schedule the task set by RM-FF-NF.

According to the schedulability condition used by RM-FF-NF,

$$\frac{1/2}{2^{2j\delta}} + \frac{\varepsilon}{2^{(2j+1)\delta}} + \frac{1/2}{2^{(2j+2)\delta}} > \frac{1/2}{2^{n\delta}} + \frac{\varepsilon}{2^{n\delta}} + \frac{1/2}{2^{n\delta}} > 1 > 1 - 2\delta\ln 2 = 1 - \beta\ln 2,$$

where $\beta = 2\delta$ and $j = 0, 1, \ldots\ldots, 2k-1$.

Hence, tasks τ_{2j+1} and τ_{2j} are assigned to a processor, for $j = 0, 1, \ldots, 2k-1$, in the completed RM-FF-NF schedule. Then a total number of $2k$ processors is required by RM-FF-NF.

We next claim that $k + 1$ processors are needed to schedule the same task set in the optimal schedule.

Since $1/2 + 1/2 = 1 \le 2^{i\delta}$ for $i = 2j$ and $j = 0, 1, \ldots, 2k-1$, any two of these $2k$ tasks can be scheduled on a processor. Yet any three of these tasks cannot be scheduled on a processor since $1/2 + 1/2 + 1/2 > 1 + 1/n > 1 + \varepsilon > 2^{n\delta}$. Therefore, exactly k processors are needed to schedule these $2k$ tasks. For the other $2k$ tasks with $\tau_i = (C_i, T_i) = (\varepsilon, 2^{i\delta})$ for $i = 2j+1$, $j = 0, 1, \ldots, 2k-1$, and $\varepsilon > 0$, one processor is needed to schedule them since $n\varepsilon << 1 \le 2^{i\delta}$.

Let N and N_0 be the number of processors required by RM-FF-NF and the minimum number of processors required to schedule this task set, respectively. Then $N = 2k$ and $N_0 = k + 1$. Hence $\Re^{\infty}_{\text{FT-RM-NF}} = 2$.

For the second bound, task sets can be similarly constructed to prove that the upper-bounded

number of processors is required by RM-FF-NF in each case. Hence we can conclude that

$$\Re^{\infty}_{\text{FT-RM-NF}}(\alpha) = \frac{1}{1-\alpha}.$$ ■

IV. Empirical Study of FT-RM-NF

In the above section, the performance bound of the new algorithm was derived under worst-case assumptions. While a worst-case analysis assures that the performance bounds are satisfied for any task set, it does not provide the insight into the average-case behavior of the algorithm. In order to answer such questions as whether the algorithm performs on the average close to its worst-case performance, one can analyze the algorithm with probabilistic assumptions, or conduct simulation experiments. We resort to simulation.

The simulation is conducted by running the algorithm on a large number of computer generated sample task sets and averaging the results over a number of runs. The input data of all parameters for a task set are generated according to uniform distribution. The periods of tasks are generated in the range of $1 \le T_i \le 500$. The number of versions for each task is uniformly distributed in the range of $1 \le \kappa_i \le 5$. The computation time of each version is in the range of $1 \le C_{i,j} \le \alpha T_i$, where $\alpha = \max_{i,j}(C_{i,j}/T_i)$ .The output parameter for the algorithm is the percentage of extra processors used to accommodate a set of tasks, with regard to the total utilization (or load) of the task set. The total load of a task set is given by $U = \sum_{i=1}^{n}\left(\sum_{j=1}^{\kappa_i} C_{i,j}\right)/T_i$, which is a lower bound on the number of processors needed to execute the task set. In other words, the optimal number of processors needed to schedule a task set with a load of U is at least U. Suppose that $N(\Sigma)$ is the number of processors required by FT-RM-NF to schedule a task set Σ with a load of U, then the percentage of extra processors is given by $100 \times \frac{N(\Sigma) - U}{U}$.

The simulation results are plotted in Figure 1 and Figure 2 with two values of α. The number of runs for each data point is chosen to be 20, since for our experiments, 20 runs is large enough to counter the effect of "randomness". In order to make comparisons, we also run the same data through the on-line algorithm FT-RM-FF [13]. When α is small, FT-RM-NF consistently outperforms FT-RM-FF. On the average, FT-RM-NF uses less than 20% extra processors when $\alpha \le 0.5$, and less than 10% extra processors when $\alpha \le 0.2$. The superiority of this algorithm is quite obvious.

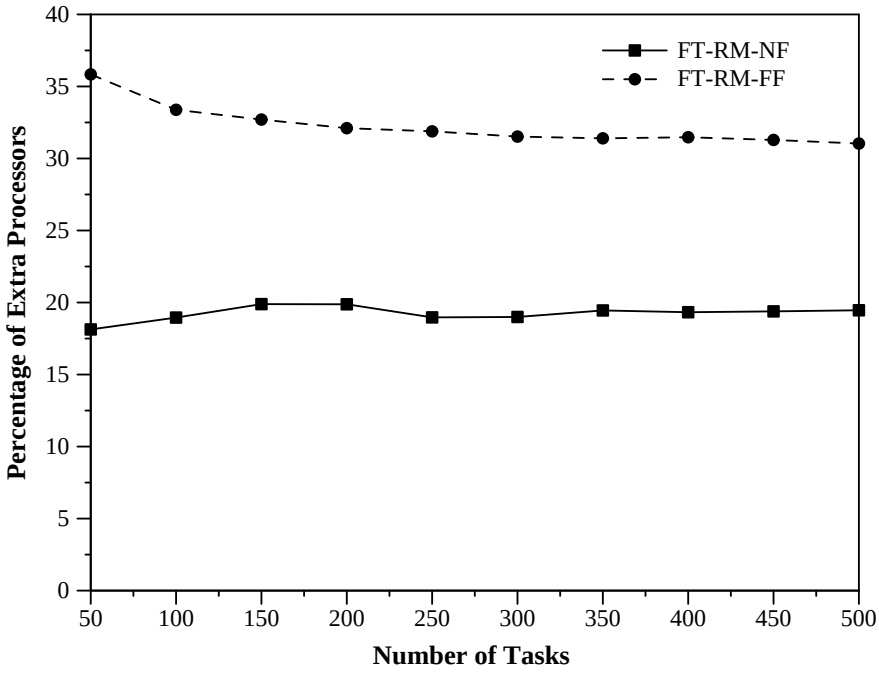

Figure 1: Performance of FT-RM-NF and FT-RM-FF (α = 0.5)

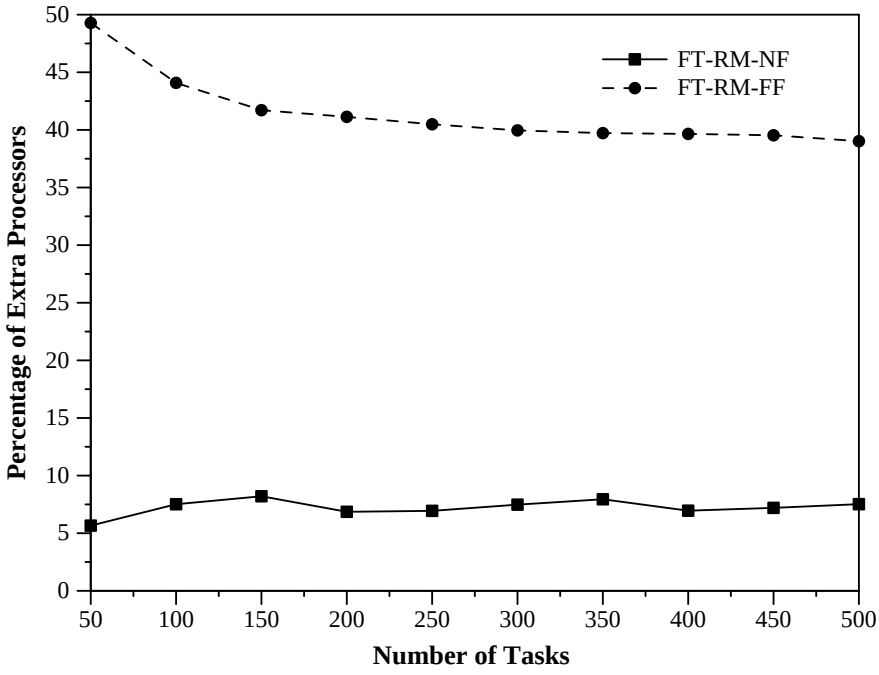

Figure 2: Performance of FT-RM-NF and FT-RM-FF (α = 0.2)

V. Conclusions

In this paper, we have investigated the problem of providing fault-tolerance in a real-time

system where rate-monotonic scheduling is used to guarantee task deadlines. Although our approach does not address specific fault-tolerant mechanisms or implementation issues, its merit lies in providing a theoretical bound on how many processors are required to tolerate a certain number of processor failures or task errors. Although it is a static algorithm, the proposed algorithm performs consistently better than a previous algorithm proposed for the same problem. The worst-case analysis shows that the algorithm FT-RM-NF has a worst-case performance bound of no more than 2. The performance of the algorithms can be significantly improved when the maximum allowable utilization of a task is small. Simulation results also indicate that the algorithm performs very well on the average. The superior performance of the algorithm is achieved through employing a new and more effective schedulability condition for the rate-monotonic scheduling and a clever allocation scheme.

There are some limitations in this work. First, for some practical applications, certain tasks must be associated with certain peripheral devices (or processors). This implies that we may not have as much freedom in assigning tasks to processors as we have in our algorithm. Second, the means of voting are not considered here, although they can be treated as a part of the operations for each version. These seem to be practical issues and their solutions are more likely to be application-specific. For future work, it would be interesting to do a case study based on this algorithm to obtain the fault-tolerance parameters (such as reliability and availability) of a practical system, and to extend this work to consider the situation where some tasks may share resources.

Due to the inevitable employment of multiprocessors in many real-time applications and the widespread use of the rate-monotonic scheduling algorithm for guaranteeing task deadlines, the problem of meeting task deadlines in such systems even in the presence of some processor failures or task errors becomes an urgent one to address. Despite certain limitations, we believe that the problem studied in this paper is an important one and the presented results are timely ones for the research community and for practitioners at large.

Acknowledgments: This work was supported in part by ONR, CIT, and Loral Federal Systems.

References

[1] Balaji, S., L. Jenkins, L.M. Patnaik, and P.S. Goel. Workload Redistribution for Fault-tolerance in a Hard Real-time Distributed Computing System. *FTCS-19*, Chicago, Illinois, 366-373 (1989).

[2] Bannister, J.A. and K. S. Trivedi. Task Allocation in Fault-tolerant Distributed Systems,

Acta Informatica, 20:261-281, 1983.

[3] Burchard, A., J. Liebeherr, Y. Oh, and S.H. Son. Assigning Real-time Tasks to Homogeneous Multiprocessor Systems, *IEEE Transactions on Computers*, to appear.

[4] Davari, S. and S.K. Dhall. An On Line Algorithm for Real-time Tasks Allocation, *IEEE Real-Time Systems Symposium*, 194-200 (1986).

[5] Dhall, S.K. and C.L. Liu. On A Real-time Scheduling Problem, *Operations Research*, 26: 127-140 (1978).

[6] Gafford, J.D. Rate-monotonic Scheduling, *IEEE Micro*, 34-39 (June 1991).

[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] Joseph, M. and P. Pandya. Finding Response Times in a Real-Time System, *The Computer Journal*, 29(5): 390-395 (1986).

[9] Krishna, C.M. and K.C. Shin. On Scheduling Tasks with A Quick Recovery from Failure, *IEEE Transactions on Computers*, 35(5):448-454 (1986).

[10] Leung, J. Y. T. and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks, *Performance Evaluation*, 2: 237-250 (1982).

[11] Lehoczky, J., L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior, *IEEE Real-Time Systems Symposium*, 166-171 (1989).

[12] Liu, C.L. and J. Layland. Scheduling Algorithms for Multiprogramming in A Hard Real-time Environment, *Journal of ACM*, 10(1):46-61 (1973).

[13] Oh, Y. and S. H. Son. Enhancing Fault-tolerance in Rate-monotonic Scheduling, *Journal of Real-Time Systems*, 7(3): 315-329 (1994).

[14] Oh, Y. The Design and Analysis of Scheduling Algorithms for Real-time and Fault-tolerant Computer Systems, *Ph.D. Dissertation*, Department of Computer Science, University of Virginia (1994).

[15] Sha, L., and J.B. Goodenough. Real-time Scheduling Theory and Ada, *Computer*, 53-65 (April 1990).

[16] Serlin, O. Scheduling of Time Critical Processes, *Proceedings of the Spring Joint Computers Conference*, 40:925-932 (1972).