Fault-Tolerant Rate-Monotonic Scheduling*

Sunondo Ghosh, Rami Melhem and Daniel Mossé
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
telephone: (412) 624-8923, fax: (412) 624-8854
{ghosh, melhem, mosse}@cs.pitt.edu

Abstract

Due to the critical nature of the tasks in hard real-time systems, it is essential that faults be tolerated. Several studies have shown that space applications, which have very high reliability requirements, have also very high transient faults frequency. Therefore, tolerance to this type of faults is essential in such applications.

In this paper, we present a scheme which can be used to tolerate faults during the execution of preemptive real-time tasks. We describe a recovery scheme which can be used to re-execute tasks in the event of single and multiple transient faults and discuss conditions that must be met by any such recovery scheme. We then extend the Rate Monotonic Scheduling (RMS) scheme to provide tolerance for single and multiple transient faults. We derive schedulability bounds for sets of real-time tasks given the desired level of fault tolerance for each task or subset of tasks. Finally, we analyze and compare the bounds derived as a function of the amount of processing time reserved for the re-executions.

1 Introduction

In recent years real-time systems have been widely used in several time critical applications ranging from fly-by-wire aircraft and space shuttles, to industrial process control and smart automobiles. The tasks in such applications are *hard* real-time tasks, and have stringent timing constraints. The consequences of missing a deadline of such tasks may be catastrophic.

Since the tasks in hard real-time systems are critical in nature, it is very important that their deadlines be met even in the presence of faults. Faults can be of three kinds: permanent, intermittent, and transient. Permanent faults are caused by the total failure of a computing unit, and are typically tolerated by using hardware redundancy, such as spares processors. Transient faults

*Supported in part by NSF grant CCR-9308886 and DARPA under contract DABT63-96-C-0044.

are temporary malfunctions of the computing unit or any other associated components which causes an incorrect result to be computed. Intermittent faults are repeated occurrences of transient faults.

The possible causes of transient faults include limitations in the accuracy of electromechanical devices, electromagnetic radiation received by interconnections (such as long buses acting like receiving antennas), power fluctuations not properly filtered by the power supply, and effects of ionizing radiation on semiconductor devices [6]. Several studies in the last two decades have concluded that transient faults are significantly more frequent than permanent faults [6, 12, 35]. In [35], measurements showed that transient faults are 30 times more frequent than permanent faults, while in [12], 83% of all faults were determined to be transient or intermittent. In [6], tables were provided to show comparative rates of permanent and transient faults on several systems: the mean time to error (transient fault) varied from 44 hours to 328 hours, while the mean time to failure (permanent fault) varied from 800 to 4900 hours.

In some real-time systems such as satellites and space shuttles, transient faults occur at a much higher frequency than in general purpose systems [5]. In [5], an orbiting satellite containing a microelectronics test system was used to measure error rates in various semiconductor devices including microprocessor systems. The number of errors, caused by protons and cosmic ray ions, mostly ranged between 1 and 15 in 15-minute intervals, and was measured to be as high as 35 in such intervals. Due to the high occurrence of transient faults in some applications, and because physical redundancy can be used to tolerate permanent faults in real-time systems [27, 28], we consider only the problem of transient and intermittent faults in this paper.

To avoid the catastrophic and costly consequences of system failure in critical real-time systems, it is essential that faults be detected and tolerated. Extensive research has been conducted on detecting faults using both hardware [8, 24, 26, 34] and software [13, 33, 36]. In [26], three error detection mechanisms are described which can be used to tolerate transient faults caused by heavy-ion radiation and power-supply disturbances. A survey of various types of watchdog processors for concurrent error detection is presented in [24]. In [8], a processing core used for embedded space flight applications is provided with internal concurrent error-detection, mainly to detect transient faults. In [34], an approach is presented for the on-line detection of control flow errors caused by transient and intermittent faults using special built-in hardware. The concept of recovery blocks is introduced in [33] and techniques to detect control flow errors of programs are presented in [13, 36].

Transient faults in real-time systems are generally tolerated using time redundancy, which involves the retry or re-execution of any task running during the occurrence of a transient fault [10, 15, 17, 31]. This is a relatively inexpensive method of providing fault-tolerance since not much extra hardware is required. For example, in space and aviation applications, reducing the hardware is important since that decreases weight, size, power consumption, and cost.

To guarantee that real-time tasks will meet their deadlines, scheduling techniques can be used to provide a predictable response time. A fundamental result in real-time scheduling was developed by Liu and Layland [22], in which periodic preemptive tasks were studied. They proved that, if tasks have fixed priorities that are derived from their periods, then any set of n tasks with a total utilization below $n(2^{1/n} - 1)$ is schedulable on a uniprocessor system. This policy was called Rate Monotonic Scheduling (RMS). In recent years, RMS has been used to schedule real-time task sets in a variety of critical applications ranging from navigation of satellites [7] to avionics [11, 18] to industrial process control [3]. However, RMS does not provide mechanisms for managing time redundancy, so that real-time tasks will complete within their deadlines even in the presence of faults. The goal of this paper is to derive new utilization bounds which include provisions for the re-execution of faulty tasks scheduled using RMS.

We are aware of three previous attempts to incorporate time redundancy into rate monotonic scheduling [4, 30, 32][1]. In [30], Pandya and Malek analyze the schedulability of a set of periodic tasks that are scheduled using RMS and tolerate a single fault. In the event of a fault, all unfinished tasks are re-executed. They prove that no task will miss a single deadline under these conditions even in the presence of a fault if the utilization of the processor does not exceed 50%. The type of faults handled are those that can be handled by re-execution, e.g., intermittent and transient hardware faults. In [31, 32], Ramos-Thuel and Strosnider present static and dynamic allocation strategies to provide fault tolerance. Two algorithms are proposed to reserve time for the recovery of periodic real-time tasks on a uniprocessor [32]. The RMS analysis of [19] has been extended to include provisions for task re-executions. Another study done recently by Burns, Davis and Punnekkat [4] provides exact schedulability tests for fault-tolerant task sets. Time redundancy is employed to provide a predictable performance in the presence of failures. The response time analysis of [1] is extended to the fault tolerance approach. However, neither study [4, 32] derives utilization bounds[2] for the task sets, which is the main goal of this paper.

The bounds developed for scheduling aperiodic tasks along with periodic tasks [20, 2] can also be used for tolerating faults by considering the re-executing task to be an aperiodic task which arrives when the task fails. However, there are some problems with this approach. First of all, the papers

[1]Other studies have dealt with real-time fault-tolerant scheduling using a timeline and a primary-backup approach [10, 16, 21, 25]. Since these works are not related to RMS, we do not discuss them here. We also do not discuss [29], which deals with fault-tolerant scheduling using RMS on a multiprocessor, since it only presents a discussion for permanent faults.

[2]A utilization bound U allows a simple test of the form $\sum U_i \leq U$ to determine a task set's schedulability. For example, in RMS, $U = n(2^{1/n} - 1)$.

dealing with aperiodic tasks generally do not give importance to the deadline of those tasks, but try to improve their response time. For fault tolerance, the deadline of the re-executing task is important. Secondly, the recovery scheme to be used with the aperiodic schemes is not clear. Thirdly, we have compared the bounds developed for scheduling aperiodic tasks with the bounds developed in this paper, and have found a significant improvement [9]. Finally, the implementation of the recovery schemes described in the aperiodic servers have more overhead due to the complexity of those schemes. For these reasons, we do not discuss the bounds developed for aperiodic tasks in this paper.

Assumptions and Definitions

In this paper, we adopt the usual RMS assumptions, as follows. We consider sets of periodic, preemptive tasks. The tasks are eligible for execution at the beginning of the period, and have to complete before the end of the period. We assume that there are n tasks in the system, and are denoted by $\tau_1, \tau_2, \ldots, \tau_n$. The tasks are independent, that is, have no precedence constraints. The computation times and periods of task τ_i are denoted by C_i and T_i respectively, and $T_i \leq T_{i+1},\ i = 1, \ldots, n-1$. The *utilization* U_i of τ_i is C_i/T_i, and the *utilization factor* of the system is the fraction of processor time spent in the execution of the task set and is equal to $\sum_{i=1}^{n} U_i$.

We assume that faults are transient or intermittent, that is, only a single task is affected by each fault. Therefore, faults can be tolerated by re-execution. We assume that either the hardware or the application can detect faults. The cost of fault detection and preemption are assumed to be negligible. However, in Section 6, suggestions are made to relax these assumptions.

We consider different fault models in this paper. First we consider the case where only a single fault can occur within $T_n + T_{n-1}$. Then we consider multiple faults within $T_n + T_{n-1}$, which can occur in different tasks or during the execution and re-execution of the same task.

The goal of this paper is twofold: first we describe a recovery scheme for the re-execution of faulty tasks, including a scheme to add slack to the schedule, and then we derive schedulability bounds for sets of tasks which require fault tolerance through task re-execution. Towards this end, we extend the rate monotonic scheduling (RMS) result of [22] to include provision for tolerance of transient faults. In Section 2, we describe how temporal redundancy can be incorporated into RMS such that guarantees can be provided for tasks to re-execute. In Sections 3 and 4, we derive utilization bounds for task sets that need to tolerate single and multiple transient faults, respectively. We compare and analyze some of the bounds in Section 5. Finally, we conclude the paper with a discussion of practical considerations in Section 6.

2 Providing Fault-Tolerance

Our general approach to fault tolerance is to make sure that there is enough slack in the schedule so that if a fault occurs while any of the tasks is executing, that task can be re-executed. Tasks are executed following the usual RMS scheme if no faults occur (and the slack is not used). However, when a fault occurs in a task, a recovery scheme is used to re-execute that task.

In the RMS and other schemes with utilization bounds less than 100%, there is a natural amount of slack in each processor. To reason about such slack we show that the amount of slack available when a fault occurs can be deterministically computed. The slack is distributed such that the amount of slack available over an interval of time is proportional to the length of that interval. The ratio of slack S available over an interval of time L is thus constant and can be imagined to be the utilization of a backup task B, where S/L is the backup utilization. Formally, if the backup utilization is U_B, and the backup time (or slack) available during an interval L is denoted by BT_L, then

$$BT_L = U_B L \tag{1}$$

The backup task can be envisioned as partially occupying backup slots between every two consecutive period boundaries, where a period boundary is the beginning of any period. Therefore, the length of backup slot between the k^{th} period of τ_i and l^{th} period of τ_j is given by $U_B(lT_j - kT_i)$, where there is no period boundary of any tasks in the system between times kT_i and lT_j. It is important to note that the backup task is just an abstraction to help in reasoning about the existing (or inserted) slack in the processor.

As an example, consider two tasks with $C_1 = 1.5, T_1 = 5, C_2 = 2$, and $T_2 = 8$ (see Figure 2). The two utilizations are 30% and 25% respectively. Assuming that $U_B = 30\%$, the lengths of backups from 0 to T_1 is 1.5, from T_1 to T_2 is 0.9, from T_2 to $2T_1$ is 0.6, and so on. The utilization factor of the system is 55%. Note that the reserved slack is shown in the figure at the beginning of each period. However, if no fault occurs then the next eligible task is executed at that time (see Section 2.2).

Note the use of *overloading* of the backups. Overloading, which is the scheduling of backups for more than one task in the same time slot has been used in non-preemptive aperiodic environments [27]. In our scheme, the same slack is being used as the backup for all the tasks in the system. If a separate backup is scheduled for each task, the utilization of each task would double, and would reduce the maximum schedulability to half of the initial value ($\approx$ 34.5% in the general case, which assumes that tasks are schedulable if their utilization is less than $\approx$ 69% [22]).

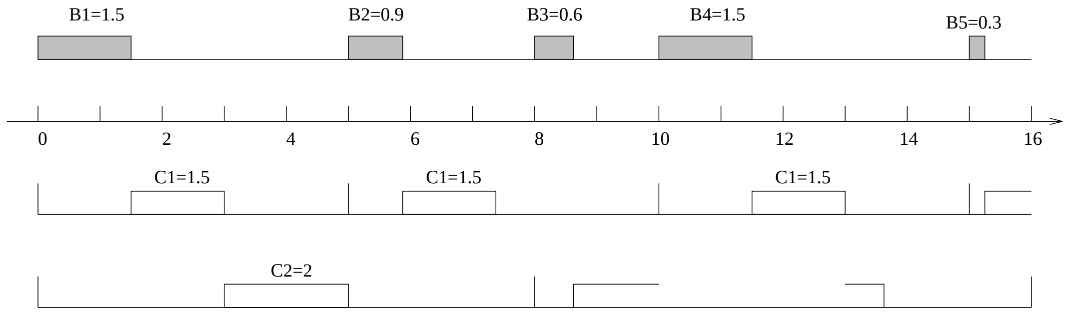

Figure 1: Two tasks and the backup tasks.

2.1 Conditions for Recovery

In this section, we state conditions which are required to ensure recovery from single and multiple faults in a preemptive, periodic system.

Let us first assume that there are no faults in the system. If the set of tasks passes one of the fault-tolerant schedulability tests described in later sections, then each task will finish before its deadline. We will show later that the utilization factor of tasks with a backup computed using our formulas is always smaller than the Liu and Layland bound of [22]. Therefore, if there are no faults, the slack is not used for executing any task, the task set will satisfy the RMS bound, and is thus schedulable.

In addition to the fault-free case above, we need to ensure the schedulability of the task set even when the backups are used for the re-execution of some task. This translates into deriving a schedulability bound. One easy way of deriving such a bound is to decrease the utilization bound of the system by the backup utilization. If the resulting task set still meets this new bound, then it is schedulable. If the Liu and Layland bound is denoted by U_{LL}, then this naive bound can be determined as follows:

$$U_{naive} = U_{LL} - U_B \tag{2}$$

For large values of n, $U_{LL} \approx 69\%$. Thus, if the backup utilization is 25% (for example), then according to the above formula, the utilization bound for any task set with large n can be 44%. This is obviously not a tight bound, since [30] has shown the bound to be 50%. Our goal is to develop a recovery (re-execution) scheme for the faulty tasks and the corresponding tight bound on task schedulability given a set of tasks and the utilization of the backup.

A recovery scheme that ensures the re-execution of any task after a fault has been detected must satisfy the following conditions:

[S1]: There should be sufficient slack for every instance of each task to re-execute. In other words, the slack between kT_i and $(k+1)T_i$ should be at least C_i for any value of k and i.

[S2]: When any instance of τ_i finishes executing, all the slack available within its period (at least C_i if **[S1]** holds) should be available for the re-execution of τ_i in case a fault is detected.

[S3]: When a task re-executes, it should not cause any other task to miss its deadline.

[S1] ensures the availability of sufficient slack for a task to re-execute, and **[S2]** ensures that this slack can be used after a task finishes executing to re-execute that task before its deadline if a fault is detected. **[S3]** ensures that all tasks meet their deadlines even when a high priority task needs to re-execute.

If these three conditions are met, then it is possible to re-execute a faulty task and meet its deadline. However, we still need a recovery scheme which will explain how the slack should be used. We will show below that the recovery scheme needed is very straightforward: *the faulty task should be simply re-executed at its own priority.*

2.2 Single Fault

In this section, we study a fault model in which *at most one fault occurs within any interval of length* $T_n + T_{n-1}$. In the next section we also consider multiple faults.

Lemma 1 *If* $U_B \geq C_i/T_i,\ i = 1, \ldots, n$, *then* **[S1]** *is satisfied.*

Proof: From (1), the amount of slack available within any period T_j is $U_B T_j$. Since $U_B \geq U_i, i = 1, \ldots, n$, the amount of idle time within T_j is $T_j U_B \geq T_j C_j / T_j = C_j$. □

To show that **[S2]** is met, we use the concept of *swapping*. We define swapping to be the usage of slack for the execution of a task, leading to the creation of slack of equal length following the task. The slack is effectively shifted later in time, that is, it is swapped with the task's computation time[3]. Swapping is illustrated in Figure 2. The black areas in the figure show the currently running part of a task, while the gray areas indicate a backup. The computation of a task τ_1 gets swapped with backup time B_1 (shown in Figure 2(b)) and then a part of a second task τ_2 gets swapped with the previously swapped backup B_1 (shown in Figure 2(c)). The second part of τ_2 executes next, swapping with B_1, since τ_1 is not yet ready, and so on.

Lemma 2 *If* **[S1]** *is satisfied, and swapping takes place, then* **[S2]** *is satisfied.*

[3]Note that swapping, like the backups or slack, is just an abstraction, and is used simply to demonstrate that the scheme works. Swapping will automatically take place in the fault-free case in RMS, that is, whenever the slack is not used for re-executing a task.

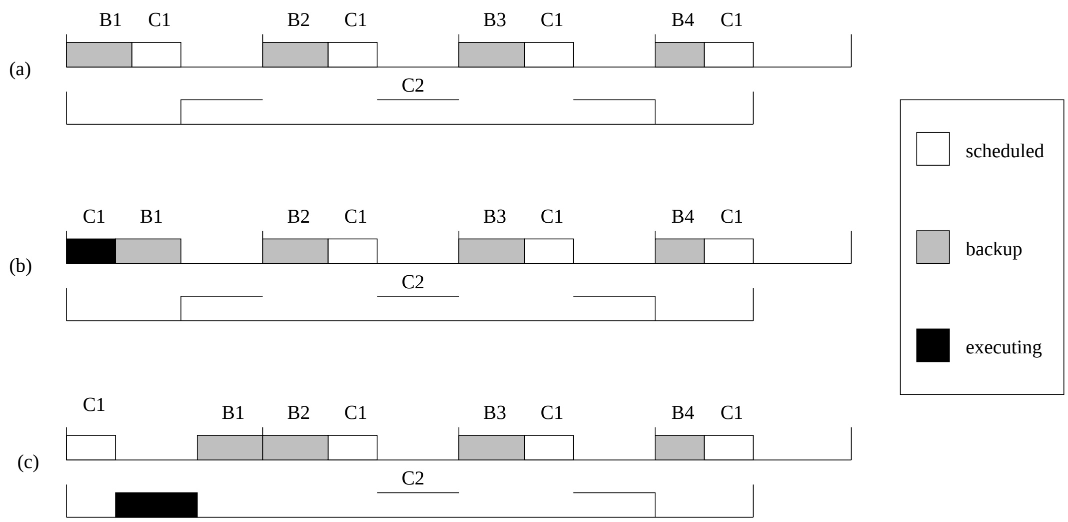

Figure 2: Swapping of C_1 and B_1 (b) and then C_2 with B_1 (c).

Proof: Since **[S1]** is satisfied, we know that there is enough slack within the period of every task τ_i, but need to prove that all the required slack is available when τ_i finishes executing.

First, consider the task with the smallest period (τ_1), and thus with the highest priority. Since **[S1]** is true, the total slack available within T_1 is at least C_1. When τ_1 executes, assuming no faults have occurred within $T_1 + T_n$, then C_1 gets swapped with the backup available at the beginning of T_1, and an equivalent amount of slack is created following its execution. Thus, when any instance of τ_1 finishes executing, all the slack within its period is available for τ_1 to re-execute.

Note that this is true for any task $\tau_i, 1 \leq i < n$. That is, when any instance of task τ_i finishes executing, all the slack within its period is available for τ_i to re-execute. If there is no fault within $T_{i+1} + T_n$, then none of the backup available within T_{i+1} is used up. Thus C_{i+1} gets swapped with all the backups available within T_{i+1} (which add up to C_{i+1} since **[S1]** holds), and the total slack available at the end of its execution is at least C_{i+1}. Thus, **[S2]** is true. $\Box$

Figure 3 shows the first two instances of C_1 and C_2 from Figure 2 have completed their executions, and demonstrates that **[S2]** is satisfied for C_2.

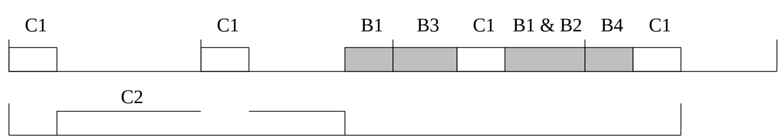

Figure 3: Swapping of C_2 with backups complete.

Lemma 3 *If* **[S1]** *and* **[S2]** *are satisfied, and the faulty task is re-executed at its own priority, then* **[S3]** *is satisfied.*

Proof: If a task is re-executed at its own priority, it can never cause higher priority tasks to miss their deadlines. Therefore, we only need to prove that the re-executing task will not cause lower priority tasks to miss their deadlines. Let the period of the re-executing task be T_r, and the period of some lower priority task be T_l (see Figure 4). We know that $T_r \leq T_l$. Let us consider three scenarios: (1) T_r starts before the beginning of T_l and ends within T_l, (2)T_r is fully contained within T_l, and (3) T_r starts within T_l and ends after T_l finishes.

In cases 1 and 2, when T_r finishes before the end of T_l, it is easy to prove that τ_r will not cause τ_l to miss its deadline. This is because there is sufficient slack within T_r to complete the re-execution of τ_r (since Lemmas 1-3 hold). If τ_r re-executes, it simply uses up the slack already available before the end of T_r, and recovery completes without displacing τ_l. Note that if τ_r does not re-execute, swapping occurs and τ_l will finish earlier than expected.

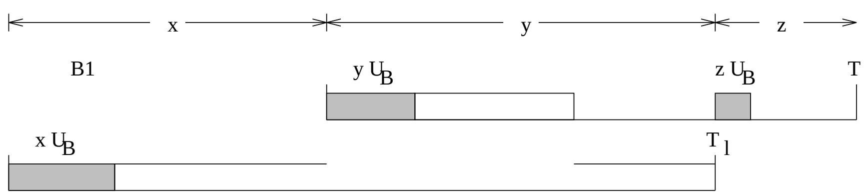

Figure 4: Task τ_r finishes after τ_l.

Case 3, when the end of T_r is after the end of T_l, is slightly more complex. Consider the case shown in Figure 4, where τ_l will just meet its deadline if RMS is used. Task τ_r will need up to $U_B T_r$ to re-execute, which is greater than the backup available between the beginning of T_r and the end of T_l. Thus, it appears that the re-execution of τ_r at its own priority could push τ_l beyond its deadline. However, we can prove that τ_l will still meet its deadline.

Let x, y, and z be the lengths of time as shown in Figure 4. Since we assume that only one fault can occur within $T_n + T_{n-1}$, and a fault occurs during τ_r's execution, then any re-executing task will not use the backup (xU_B) within T_l. Thus, τ_l is able to use the slack of length xU_B available between the beginnings of T_l and T_r, and finish earlier by xU_B. The re-execution of τ_r delays τ_l by $(y+z)U_B$, which is zU_B more than the slack scheduled before the end of T_l. Even then τ_l will meet its deadline because it has used up xU_B slack available earlier, and $x > z$ (since $T_l > T_r \Rightarrow x + y > y + z$). □

Theorem 1 *One fault can be tolerated within* $T_n + T_{n-1}$ *if slack is distributed according to (1) and* **[S1]**, **[S2]** *and* **[S3]** *are true.*

Proof: Directly from Lemmas 1, 2, and 3. □

However, a single fault being tolerated within $T_n + T_{n-1}$ is only the worst case scenario, and occurs either when $U_n = U_B$ and τ_n is re-executed, or when $U_{n-1} = U_B$, and τ_{n-1} is re-executed. In most cases, more than one fault can be tolerated within $T_n + T_{n-1}$, but guarantees cannot be provided. For example, if the faults occur only while τ_1 is executing, then at least one fault can be tolerated in each T_1.

2.3 Multiple Faults

In this section, we consider multiple faults within $T_n + T_{n-1}$. The utilization of the backup has to be the total task utilization for which fault tolerance is required. Let us assume that the schedule has m backups $B_j,\ j = 1, \ldots, m$ with a corresponding utilizations U_{Bj}. Let $U_{BT} = \sum_{j=1}^{m} U_{Bj}$.

If $U_{BT} = U_i + U_j$ $(1 \leq i, j \leq n)$ and $U_i < U_j$, then any task τ_k with utilization between U_i and U_j (i.e., $U_i < U_k \leq U_j$) will be able to tolerate a single fault within $T_n + T_{n-1}$, while any task with utilization less than U_i will be able to tolerate an additional fault within the same interval of time. If $U_{BT} = 2U_i$, then any task with utilization less than U_i can tolerate two faults within $T_n + T_{n-1}$. Using this approach, multiple faults can be tolerated within the $T_n + T_{n-1}$.

Various kinds of multiple faults can be tolerated in the following ways:

- If $U_{Bj} \geq U_i\ \forall j : 1 \leq j \leq m, \forall i : 1 \leq i \leq n$, then m faults can be tolerated within $T_n + T_{n-1}$. The minimum value of U_{BT} in this case is $m \times max\{C_j/T_j\}, 1 \leq j \leq n$.
- If $U_{BT} = mC_j/T_j$, for some j, then all tasks τ_i with $U_i \leq U_j$ can tolerate m faults within $T_n + T_{n-1}$.
- If only a single fault must be tolerated for all tasks in the system, but k additional faults must be tolerated for a set R_C of m critical tasks, then we should have $U_{BT} = max\{C_i/T_i\} + k \times max\{C_j/T_j \mid \tau_j \in R_C\}$.

Clearly, for any desired level of fault tolerance for various tasks, a different value of U_{BT} can be derived. The proof for each case is similar to that described in the case of single faults within $T_n + T_{n-1}$. The recovery scheme is also the same, that is, the re-executing task runs at its own priority. Multiple faults will be further discussed in Section 4.

3 RMS Bounds to Tolerate Single Faults

Now that we have described a recovery scheme that uses slack to guarantee fault tolerance in RMS, we need to study the effect of the added slack on the utilization factor $(\sum_{i=1}^{n} U_i)$. Clearly, the utilization factor will decrease when a fraction of the processing capacity is reserved for re-execution. First we state some definitions from [22] that will facilitate the following discussion.

A processor is said to be *fully utilized* when an increase in the computation time of any of the tasks in the set will make the set unschedulable. The *least upper bound* of the utilization factor is the minimum of all utilization factors over all sets of tasks that fully utilize the processor. Therefore, if a task set has a utilization factor smaller than the least upper bound, then it is schedulable.

The problem we need to solve can be stated as follows: Given a set of tasks $\tau_1, \tau_2, \ldots, \tau_n$, and a backup utilization, U_B, determine the least upper bound of the utilization factor. To determine this bound, we follow the proof strategy used in [22]. Specifically, the following steps are used:

- Step 1: Determine the critical instant. The critical instance occurs when the response time of each task is maximum, and thus the utilization factor is minimized.
- Step 2: Determine the value of T_i with respect to T_j and C_j ($j = 1, \ldots, i-1$) for which the upper bound of the utilization factor is minimized. Assume that $T_1 \leq T_i \leq 2T_1,\ i = 2, \ldots, n$.
- Step 3: For the periods T_i determined in Step 2, find the relationship between the backup utilization U_B, the number of tasks n, and the least upper bound of the utilization factor.
- Step 4: Generalize the bound for arbitrary task periods, removing the restriction in Step 2, deriving a fault-tolerant utilization bound.

Once we have a fault-tolerant utilization bound, we can determine whether or not a given set of tasks is schedulable with fault tolerance. In this section, we consider only the case of single faults within $T_n + T_{n-1}$.

3.1 Step 1: Determining critical instant

Liu and Layland [22] have shown that, for a set of periodic preemptive tasks, the critical instant occurs when all tasks arrive simultaneously. If no faults occur, then the arguments used by [22] are applicable to our system. However, we also need to consider the case where a fault occurs and a task has to be re-executed at its own priority, thus adding to the response time of lower priority tasks. If a single fault occurs, the maximum response time for τ_i occurs when some task τ_j with higher priority ($T_j < T_i$) and with the maximum computation time among all tasks with priority higher than i ($C_j = max\{C_k\}, k = 1, \ldots, i-1$) needs to re-execute. In this case, τ_i gets delayed by at most C_j due to the re-execution. This additional factor is maximum when τ_i arrives at the same time as τ_j, so that the τ_i gets delayed by exactly C_j. Thus, the maximum response time of a task occurs when it arrives with all other tasks of higher priority.

3.2 Step 2: Minimizing U for varying task periods

In this section, we assume that $T_1 \leq T_i \leq 2T_1,\ i = 1, \ldots, n$. We determine the values of C_i's for which the utilization factor is minimized, while maintaining a fully utilized processor. Let $C_1, C_2, \ldots, C_n$ be the computation times such that the processor is fully utilized. Using (1), the length of the backup within the first instance of T_1 is $B_1 = T_1 U_B$, and the lengths of backups between the end of T_i and the end of T_{i+1} is $B_{i+1} = (T_{i+1} - T_i)U_B$ (see Figure 5). Therefore, $C_1 = T_2 - T_1 - B_2 = (T_2 - T_1)(1 - U_B)$.

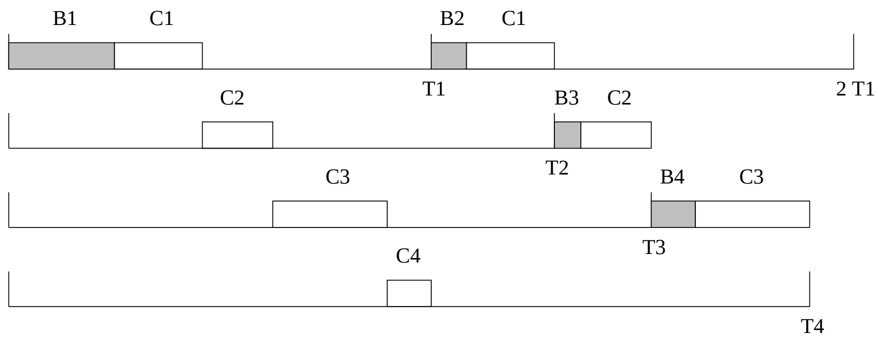

Figure 5: Four tasks with $T_{i+1} = T_i + B_{i+1} + C_i$.

First we determine the effect of varying only C_1 and C_2 on the utilization factor. If the value of C_1 increases by $\Delta(1 - U_B)$, then C_2 has to be decreased by a value of $\Delta(1 - U_B)$ to ensure that the processor is still fully utilized. This new value of utilization is

$$U' = \frac{C_1 + \Delta(1 - U_B)}{T_1} + \frac{C_2 - \Delta(1 - U_B)}{T_2} + \ldots + \frac{C_n}{T_n}$$

The change in utilization is positive: $U' - U = (1/T_1 - 1/T_2)\Delta(1 - U_B) > 0$.

Similarly, if the value of C_1 decreases by $\Delta(1 - U_B)$, then C_2 has to be increased by $2\Delta(1 - U_B)$. The new utilization is

$$U' = \frac{C_1 - \Delta(1 - U_B)}{T_1} + \frac{C_2 + 2\Delta(1 - U_B)}{T_2} + \ldots + \frac{C_n}{T_n}$$

As before, the change in utilization is positive: $U' - U = (-1/T_1 + 2/T_2)\Delta(1 - U_B) > 0$.

Therefore, if U is minimum, then $C_1 = (T_2 - T_1)(1 - U_B)$. Similarly, given that $B_{i+1} = (T_{i+1} - T_i)U_B,\ i = 1, \ldots, n-1$, it can be shown (see [9]) that U is minimum when $C_i = T_{i+1} - T_i - B_{i+1}$ which yields

$$C_i = (T_{i+1} - T_i)(1 - U_B), \quad i = 1, \ldots, n-1 \tag{3}$$

Since the processor is fully utilized, $T_n = 2(C_1+C_2+\ldots+C_{n-1})+C_n$. Therefore

$$\begin{aligned} C_n &= T_n - 2(C_1 + C_2 + \ldots + C_{n-1}) - T_n U_B \\ &= (2T_1 - T_n)(1 - U_B) \end{aligned} \tag{4}$$

An example of a schedule for 4 tasks satisfying (3) and (4) is shown in Figure 5. A similar schedule can be drawn up for n tasks.

3.3 Simple FT-RMS bound

As the U_{naive} bound (2) shows, a straightforward way of computing a schedulability bound for fault-tolerant RMS would be to decrease the bound in [22] by U_B, (i.e., $U_{LL} - U_B$). However, using (3) and (4) and following the proof in [22], we obtain the following improved result which is a general fault tolerance bound for RMS (G-FT-RMS):

$$U_{G\text{-}FT\text{-}RMS} = n(2^{1/n} - 1)(1 - U_B) = U_{LL}(1 - U_B) \tag{5}$$

This equation is the result of adding a backup with utilization U_B distributed uniformly across the schedule which is equivalent to scheduling the tasks $\tau_i, 1 \leq i \leq n$ on a virtual processor with only $(1 - U_B)$ of the processor's computation power.

Consider an example to show how (5) is used to determine whether a task set is schedulable or not. Consider a set of 3 tasks with utilizations $U_1 = 10\%, U_2 = 20\%$, and $U_3 = 25\%$. If we assume that a single fault needs to be tolerated, then $U_B = 25\%$. (5) gives us a bound of 58.5% while the sum of utilizations of the task is 55%. Since $\sum U_i < U_{G\text{-}FT\text{-}RMS}$, the tasks are schedulable.

The above equation is a general equation applicable to RMS for any value of U_B. If $U_B = max\{U_i\}, 1 \leq i \leq n$ (as it is the case in the example), then all the tasks in the system can tolerate a single fault within $T_n + T_{n-1}$. We discuss in Section 4 how the bound $U_{G\text{-}FT\text{-}RMS}$ (5) can be slightly modified to account for multiple faults.

3.4 Step 3: Determining the minimum U for a given U_B

The bound $U_{G\text{-}FT\text{-}RMS}$ was derived by simply reserving enough slack for tasks to re-execute. However, by using additional information, we can achieve a better bound. This additional information takes the form of a reasonable assumption: *the value of U_B is equal to one of the task utilizations, that is, $U_B = U_k$, for some $k, 1 \leq k \leq n$.* In practice, this is easily achieved, since all the task utilizations are known. Surprisingly, this simple assumption helps us to significantly improve upon the $U_{G\text{-}FT\text{-}RMS}$ bound.

Assume that $U_B = U_k = \frac{C_k}{T_k}$, for some $k \neq 1$ and $k \neq n$. The cases of $k = 1$ and $k = n$ are the boundary cases, and have a derivation similar to the general case. Substituting the value of C_k in (3) gives us

$$T_k = T_{k+1}(1 - U_B) \tag{6}$$

Using (3), (4) and (6), the utilization factor is

$$\begin{aligned}
U = \sum_{i=1}^{n} \frac{C_i}{T_i} &= \sum_{i=1;i\neq k}^{n-1} [\frac{(T_{i+1} - T_i)(1 - U_B)}{T_i}] + \frac{(2T_1 - T_n)(1 - U_B)}{T_n} + U_k \\
&= \sum_{i=1;i\neq k}^{n-1} [(\frac{T_{i+1}}{T_i} - 1)(1 - U_B)] + (\frac{2T_1}{T_n} - 1)(1 - U_B) + U_B \\
&= U_{VAR}(1 - U_B) - (n-1)(1 - U_B) + U_B \\
\text{where} \quad U_{VAR} &= \sum_{i=1;i\neq k}^{n-1} [\frac{T_{i+1}}{T_i}] + \frac{2T_1}{T_n} \qquad (7)
\end{aligned}$$

From here onwards, we consider only U_{VAR}, since the other terms contributing to U are not dependent on the periods $T_1, \ldots, T_n$. Let $h_i = \frac{T_i}{T_n}$, $i = 1, \ldots, n$. Using (6) and (7) we get

$$U_{VAR} = \sum_{i=1;i\neq k-1;i\neq k}^{n-1} \frac{h_{i+1}}{h_i} + \frac{h_{k+1}(1 - U_B)}{h_{k-1}} + \frac{2h_1}{h_n} \tag{8}$$

Taking the derivatives of U_{VAR} with respect to each h_i, $i = 1, \ldots, n-1$ and $i \neq k$ (note that $i \neq k$ from (7)), and making them equal to 0 to find the minima, we get the following set of equations. Note that, by definition, $h_n = 1$.

$$\begin{aligned}
\frac{dU}{dh_1} = 0 \;&\Rightarrow\; -\frac{h_2}{(h_1)^2} + 2 = 0 && \Rightarrow 2(h_1)^2 = h_2 \\
\frac{dU}{dh_i} = 0 \;&\Rightarrow\; \frac{1}{h_{i-1}} - \frac{h_{i+1}}{(h_i)^2} = 0 && \Rightarrow (h_i)^2 = h_{i-1}h_{i+1}, \\
& \quad i = 2, \ldots, k-2, k+2, \ldots, n-1 && \\
\frac{dU}{dh_{k-1}} = 0 \;&\Rightarrow\; \frac{1}{h_{k-2}} - \frac{h_{k+1}(1 - U_B)}{(h_{k-1})^2} = 0 && \Rightarrow (h_{k-1})^2 = h_{k-2}h_{k+1}(1 - U_B) \\
\frac{dU}{dh_{k+1}} = 0 \;&\Rightarrow\; \frac{1 - U_B}{h_{k-1}} - \frac{h_{k+2}}{(h_{k+1})^2} = 0 && \Rightarrow (h_{k+1})^2 = \frac{h_{k-1}h_{k+2}}{(1 - U_B)}
\end{aligned}$$

The above equations can be solved to find that

$$\frac{h_2}{h_1} = \frac{h_3}{h_2} = \ldots = \frac{h_{k-1}}{h_{k-2}} = \frac{h_{k+1}(1 - U_B)}{h_{k-1}} = \frac{h_{k+2}}{h_{k+1}} = \ldots = \frac{h_n}{h_{n-1}} = \frac{2h_1}{h_n} \tag{9}$$

and

$$\frac{h_n}{h_{n-1}} = \frac{1}{h_{n-1}} = [2(1 - U_B)]^{(\frac{1}{n-1})} \tag{10}$$

Using (7), (8), (9) and (10), we get

$$U_{1-FT-RMS} \quad = \quad (n-1)(1-U_B)([2(1-U_B)]^{\frac{1}{n-1}}-1)+U_B \qquad (11)$$

We have derived [9] the same bound (11) for the case $U_B = U_1$ and $U_B = U_n$.

We proved above that (11) determines the least upper bound of utilization factor for all task sets which can tolerate single faults within $T_n + T_{n-1}$ with an assumption that $\forall i : T_1 \leq T_i \leq 2T_1$. Therefore, if a task set meets the condition $\sum_{i=1}^{n} U_i < U_{1-FT-RMS}$, then it is schedulable, and any task τ_i with $U_i < U_B$ can tolerate one fault within $T_n + T_{n-1}$.

3.5 Step 4: Minimizing U for all task sets

To prove that $U_{1-FT-RMS}$ is the least upper bound of the utilization factor for all task sets, we also need to consider tasks sets with $T_i > 2T_1$ for one or more $i = 1, \ldots, n$. This can be done by using a variation of the theorems used in [22] and in [29]. Due to lack of space, we do not include this proof in this paper. However, we can state that (11) gives the least upper bound of the utilization factor for all tasks sets with n tasks and backup utilization U_B.

4 RMS Bounds to Tolerate Multiple Faults

In this section, we consider two ways in which multiple transient faults can occur. The first is when each fault occurs while different tasks are executing (Section 4.1), and the second is when a fault occurs when the primary and again when the backup of the same task are executing (Section 4.2). If the total backup utilization is U_{BT}, then a general bound for the task set can be derived by replacing U_B with U_{BT} in (5), that is, the new bound will be $U_{LL}(1 - U_{BT})$. Again, if we use additional information, we can improve this bound.

4.1 Multiple faults in different tasks

Assume m backups and each has its utilization equal to one of the task utilizations, that is, for $j = 1, \ldots, m, U_{Bj} = U_i$ for some $1 \leq i \leq n$. Then we can produce a mapping from each backup index j to some task index i. Let $i = \sigma(j)$ denote the mapping from backup index j to task index i, and S be the set of task indices i such that U_i is equal to some U_{B_j}, that is, $\exists j, U_{Bj} = U_i = U_{\sigma(j)}$. Formally, $S = \{\ i \mid i = \sigma(j) \text{ for } j = 1, \ldots, m\}$

For example, if $U_{B_1} = U_4$ and $U_{B_2} = U_7$, then $\sigma(1) = 4$, $\sigma(2) = 7$, and S = $\{4,\ 7\}$. The task indices in set S represent task utilizations which are known, and therefore constant[4]. In that case, we have the following equations:

$$U_{Bj} = U_{\sigma(j)} = \frac{C_i}{T_i},\ j = 1, \ldots, m \qquad (12)$$

[4]Recall that in Liu and Layland [22], all task utilizations were variable.

$$C_i = \begin{cases} (T_{i+1} - T_i)(1 - U_{BT}), & \text{if } i < n \\ (2T_1 - T_n)(1 - U_{BT}), & \text{if } i = n \end{cases} \quad (13)$$

Using (13) the utilization factor can be derived as in the case of single fault[5]:

$$\begin{aligned} U = \sum_{i=1}^{n} \frac{C_n}{T_n} &= U_{VAR}(1 - U_{BT}) - (n - m)(1 - U_{BT}) + U_{BT} \\ \text{where} \quad U_{VAR} &= \sum_{i=1; i \notin S}^{n-1} [\frac{T_{i+1}}{T_i}] + \frac{2T_1}{T_n} \end{aligned} \quad (14)$$

As before, we consider only U_{VAR} from here onwards. The set of task indices S can be broken into smaller sets $S_1, S_2, \ldots, S_p$, such that each subset has consecutive task indices. If no two indices in S are consecutive, then there are m subsets. Let the number of indices in S_k be b_k, $k = 1, \ldots, p$ $(i.e., |S_k| = b_k)$ and $b_1 + b_2 + \cdots + b_p = m$.

Substituting the values of C_i from (13) into (12), and using $\sigma(j) = i$ (when $i \in S$) we get

$$T_{\sigma(j)} = T_{\sigma(j)+1} \frac{1 - U_{BT}}{1 - U_{BT} + U_{Bj}}, \; j = 1, \ldots, m \quad (15)$$

Let us assume that $\forall l : i, i + b_l + 1 \notin S, \quad i + 1, i + 2, \ldots, i + b_l \in S$, and $b_l = |S_l|$ such that $i + 1, i + 2, \ldots, i + b_l \in S_l$. Also assume that $\sigma(x_i) = i + 1, \ldots, \sigma(x_i + b_l - 1) = i + b_l$. Then, using (15), we get

$$T_{i+1} = T_{i+b_l+1} \frac{(1 - U_{BT})^{b_l}}{\prod_{j=x_i}^{x_i+b_l-1} (1 - U_{BT} + U_{Bj})} \quad (16)$$

Let $h_k = \frac{T_k}{T_n}$, $k = 1, \ldots, n$. Also, let $F_i = \frac{(1-U_{BT})^{b_l}}{\prod_{j=x_i}^{x_i+b_l-1}(1-U_{BT}+U_{Bj})}$ for $l = 1, \ldots, p$, $i + 1 \in S$ and $\sigma(x_i) = i + 1$. Then, applying (16) to (14), we get

$$U_{VAR} = \sum_{i=1; i \notin S; i+1 \notin S}^{n-1} [\frac{h_{i+1}}{h_i}] + \sum_{i=1; i \notin S; i+1 \in S}^{n-1} [\frac{h_{i+b_l+1}}{h_i} F_i] + \frac{2h_1}{h_n} \quad (17)$$

Thereafter, we take the derivatives of U_{VAR} with respect to each h_i, $1 \leq i \leq n - 1$, $i - 1 \notin S, i \notin S, i + 1 \notin S$, and also with respect to h_i and h_{i+b_l+1}, where $i \notin S$, $i + 1 \in S$ and $i + 1 \in S_l$. We then make the derivatives equal to 0 to find the minima. These equations can be solved to find the utilization bound for multiple faults:

$$U_{M\text{-}FT\text{-}RMS} = (n - m)(1 - U_{BT})([2 \prod_{j:\sigma(j) \in S} (\frac{1 - U_{BT}}{1 - U_{BT} + U_{Bj}})]^{\frac{1}{n-m}} - 1) + U_{BT} \quad (18)$$

[5]We assume that $n \notin S$ as we assumed $n \neq k$ in the single fault case, for simplicity of presentation. A similar derivation can be done for $n \in S$ to generate an identical result.

where $U_{BT} = \Sigma_{j=1}^{m} U_{Bj}$. Note that (18) reduces to (11) when $m = 1$.

Now let us consider a specific example to show how to use (18). Consider a set of 5 tasks with utilizations $U_1 = 4\%, U_2 = 5\%, U_3 = 10\%, U_4 = 17\%$, and $U_5 = 20\%$. Let us assume that one backup is needed for all the tasks, and an additional backup is needed for the critical tasks τ_1 and τ_2. Therefore, $U_{B1} = 20\%$ and $U_{B2} = 5\%$. For this situation, (18) gives a bound of 56.4%, while $\sum U_i = 56\%$. Since $\sum U_i < U_{M\text{-}FT\text{-}RMS}$, the tasks are schedulable with the required degree of fault tolerance. On the other hand, (5) gives us a bound of 55.8%, and thus, using the general bound (5), the tasks would be considered unschedulable.

4.2 Multiple Faults in Any Task

In this section, we also consider the case where a fault might occur during the re-execution of a faulty task. In certain cases we might need more than one backup for some critical tasks (or task sets), which must be completed within its deadline even if faults occur during their re-execution. If there are k_i backups for task τ_i, then (12) can be written as:

$$U_{Bj} = k_i U_i = k_i \frac{C_i}{T_i} \tag{19}$$

The above derivation can be repeated for this new case, and the general form of the fault tolerant RMS bound found to be is:

$$U_{FT\text{-}RMS} = (n-m)(1-U_{BT})([2 \prod_{i \in S} (\frac{1-U_{BT}}{1-U_{BT}+U_{Bi}/k_i})]^{\frac{1}{n-m}} - 1) + U_{BT} \tag{20}$$

The utilization bounds obtained above in (18) and (20) can be used to tolerate multiple faults. Equation (20) allows great flexibility with different tasks (or task sets) being able to tolerate variable number of faults. If there are only a few critical tasks in the system, and their utilization is small, then multiple backups can be provided for that set of critical tasks. The bound is so flexible that it can determine the least upper bound of the utilization factor for any combination of backups, for example, one backup for all tasks in the system (with utilization equal to the maximum utilization of all tasks), an additional backup for the critical tasks (with utilization equal to the maximum utilization of the critical tasks), and a third backup for the most critical task in the system.

5 Study of the Schedulability Bounds

In this section we analyze the behavior of some of the bounds derived above. Figure 6 shows that for different values of U_B, $U_{1\text{-}FT\text{-}RMS}$ (11) is always larger (and thus enables the acceptance of more task sets) than $U_{G\text{-}FT\text{-}RMS}$ (5). However, for small values of U_B, the two bounds generate close results.

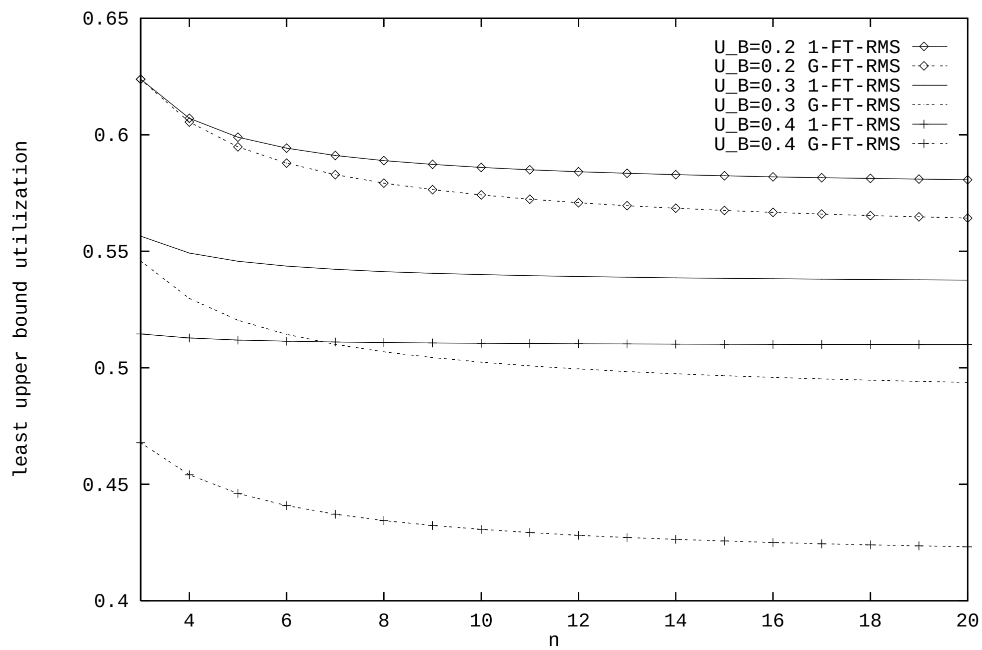

Figure 6: Comparison of General and Single Fault Equations.

The graph in Figure 7 shows how (18) is a better bound than (5) using two backups ($m = 2$), and fixing the utilization of one of the two backups at 10%. The utilization of the other backup and the total number of tasks are varied. For other values of m and other specific backup values, the graphs obtained are similar. Note that the larger the backup utilizations, the larger the gap between the two bounds.

For a given number of tasks, the bound $U_{1-FT-RMS}$ depends on U_B and varies between a minimum and a maximum value. The minimum can be found by taking the derivative of $U_{1-FT-RMS}$ (11) with respect to U_B and setting $\frac{dU_{1-FT-RMS}}{dU_B} = 0$ we obtain:

$$U_B = 1/2$$

Substituting $U_B = 0.5$ in (11), we get $U_{1-FT-RMS} = 0.5$. Therefore, the minimum utilization of the processor is 0.5, and occurs when one of the tasks has a utilization factor of 50%.

Next we determine the maximum value of $U_{1-FT-RMS}$. Since U_B is equal to the maximum of all task utilizations, U_B cannot be less than the value in (21) if the processor is fully utilized. We know that for $i = 1, \ldots, n$, $\frac{C_i}{T_i} \leq U_B$ and it can be proven that maximum utilization will be $U = nU_B$ which will occur when $\frac{C_i}{T_i} = U_B$, $i = 1, \ldots, n$. Using $U = nU_B$ and (11),

$$(n-1)2^{\frac{1}{n-1}}(1-U_B)^{(\frac{n}{n-1})} + nU_B - (n-1) = nU_B$$

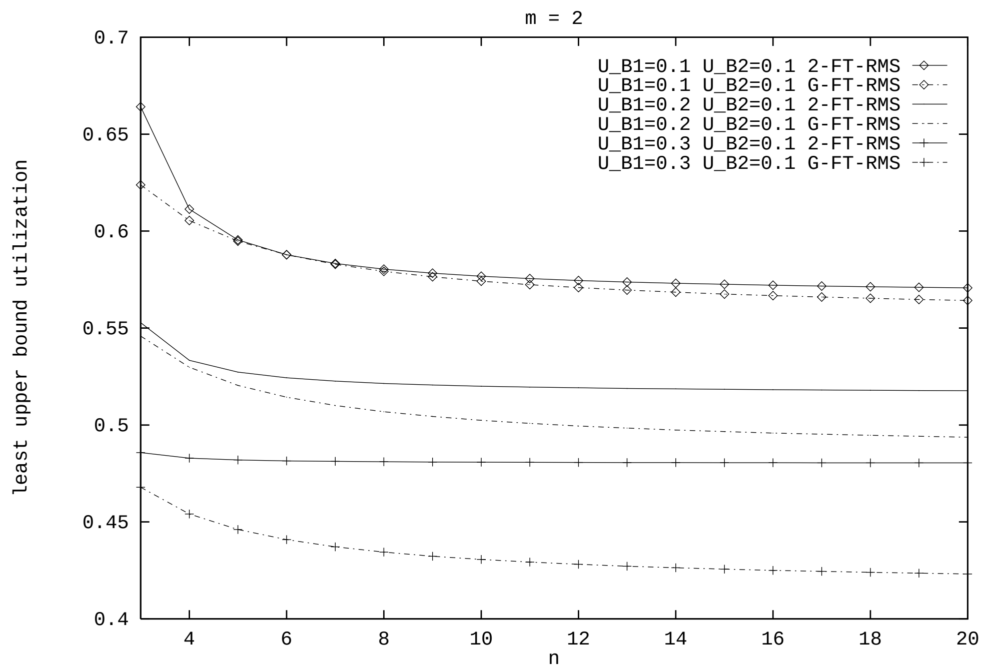

Figure 7: Comparison of General and Multiple Fault Equations.

$$\Rightarrow \quad 2^{\frac{1}{n-1}}(1-U_B)^{(\frac{n}{n-1})} - 1 = 0$$

$$\Rightarrow \quad U_B = 1 - (\frac{1}{2})^{\frac{1}{n}} \tag{21}$$

As a result, the maximum value of $U_{1-FT-RMS}$ for a given value of n can be obtained by substituting (21) in (11) to obtain

$$\text{Maximum value of } U_{1-FT-RMS} = n[1 - (\frac{1}{2})^{\frac{1}{n}}] \tag{22}$$

The plot of this equation can be seen in Figure 8.

Thus, depending on n and on the task with maximum utilization, the value of $U_{1-FT-RMS}$ varies between 50% and $\approx$ 69%. The bound is low for high values of U_B, and high for low values of U_B. This means that, if any one of the tasks in the system has high utilization (closer to 50%), then the utilization factor of the system is lower. On the other hand, if the utilization of all tasks in the system are close to each other, then the utilization factor of the system is higher. For example, consider a task set with 10 tasks. If the highest utilization of all tasks (equal to U_B) is 40%, then the least upper bound of the utilization factor ($U_{1-FT-RMS}$) is 51%, if U_B is 25%, then $U_{1-FT-RMS}$ is 56%, and if U_B is 10%, then $U_{1-FT-RMS}$ is 65%. The minimum value of U_B for 10 tasks is 6.7% (from (21)). For that value of U_B, $U_{1-FT-RMS}$ is 67%, which is the maximum possible for 10 tasks. As a comparison, the corresponding value of U_{LL} for 10 tasks is 71%.

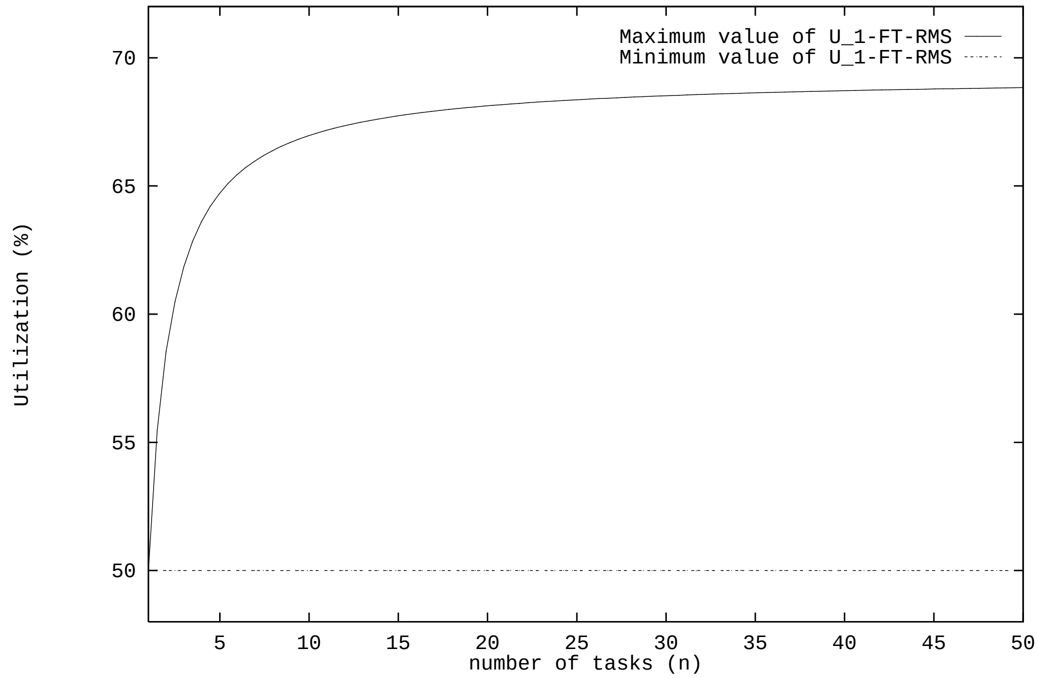

Figure 8: Maximum and minimum $U_{1-FT-RMS}$ for a given n.

The above equations are valid when fault tolerance is required for all tasks in the system. However, we may chose to provide fault tolerance to only some of the tasks by making U_B equal to the maximum of the utilizations of all tasks that need fault tolerance. The utilizations of the remaining tasks can actually be larger than U_B. The bounds (11) and (20) are valid for such cases since its derivation only assumes that $U_B = U_k$ for some $1 \leq k \leq n$.

Note that if we choose $U_B = 0$ (no fault tolerance), then we get $U_{1-FT-RMS} = (n-1)(2^{\frac{1}{n-1}} - 1)$ which is the same as U_{LL} for $n-1$ tasks. We obtain U_{LL} for $n-1$ tasks because, by assuming that $U_B = 0$, we also imply that some task, τ_k, has a utilization $U_k = U_B = 0$, effectively reducing the cardinality of the offered task set by 1.

Finally, we show that our bound is always smaller than U_{LL} for the same number of tasks. Equation (22) gives us the maximum value of $U_{1-FT-RMS}$ for all possible values of U_B. And this maximum value of $U_{1-FT-RMS}$ is equal to

$$\begin{aligned} \text{Maximum value of } U_{1-FT-RMS} &= n[2^{1/n} - 1]/2^{1/n} = U_{LL}/2^{1/n} \\ \Rightarrow U_{1-FT-RMS} &\leq U_{LL}/2^{1/n} < U_{LL} \end{aligned} \tag{23}$$

Note that the maximum value of $U_{1-FT-RMS}$ asymptotically approaches U_{LL}.

6 Discussion

6.1 Contributions

In this paper, we have studied an approach for providing fault tolerance while dispatching tasks according to the rate monotonic scheduling approach. The main contributions of this paper are:

- Introduction of a new scheme by which slack is added to a schedule of periodic tasks in a way that every instance of each task is guaranteed to tolerate single or multiple faults. The number of faults tolerated depends on the amount of slack reserved.
- Statement and proof of conditions required to ensure recovery from single and multiple faults.
- Description of a simple recovery scheme in which the faulty task is re-executed at its own priority. As a result of this, any system which uses rate monotonic scheduling can use this fault tolerance approach.
- Derivation of a general utilization bound (5) by using the new approach for distributing slack, and applying the derivations of [22].
- Use of a simple technique by which the backup's utilization is made equal to one of the task utilizations. This step leads to new fault-tolerant RMS utilization bounds (11) and (20) which are better than (5). These bounds account for the re-execution of tasks when faults occur. Given a set of tasks, these bounds can be used to determine if the tasks are schedulable with fault tolerance.

6.2 Revisiting Assumptions

Now we discuss some issues of practical importance. First, any critical real-time system must tolerate permanent faults in addition to transient faults. Secondly, transient faults need to be detected before they can be tolerated. We propose a simple approach by which both of these goals can be achieved considering only crash faults for the permanent fault case. A combination of time and space redundancy can be used to detect and tolerate both transient and permanent faults. Two processors are used to execute the tasks, and at the end of the tasks' execution, their results are compared. If their results do not match, then a transient fault has occurred. Using the slack reserved when the tasks were scheduled, the faulty task is re-executed on both processors, and the results compared again. If the fault is transient, then the re-execution will allow a correct result to be generated before the task's deadline. During the comparison, if one of the processors does not generate a result, then a permanent fault has occurred and the results from the non-faulty processor can be used. A spare processor can be used to replace the faulty processor, if desired.

Furthermore, we assumed in this paper that preemption and fault detection costs are negligible. In practice, these costs are not negligible. The fault detection cost for the scheme described above will include the communication cost between the two processors, and the comparison time. The communication and comparison times are more or less constant and have to be significantly lower than the task computation times to be cost effective. This additional time can be added to the computation time of each task *a priori*, so that fault detection times are taken into account. The cost of preemption in RMS has been studied in [14]. Our next step in this research is to apply those results in our scheme.

6.3 Uses for FT-RMS Bounds

In addition to the hardware faults discussed, the recovery scheme presented in this paper can also be used to tolerate software faults by re-executing a different version of the task if the first version fails to meet its acceptance test. The scheduling scheme and the schedulability bounds remain the same, regardless of what software version is used.

Our recovery scheme can be used in two special cases: (1) to tolerate timing faults, which occur when a task exceeds its assigned maximum execution time, and (2) for the implementation of the imprecise computation paradigm [23]. In such cases, the reserved slack can be used to complete the task's execution, and thus tolerate the timing faults up to a certain limit (equal to the backup length or at least twice the execution time). Similarly, if the worst case execution time for most real-time tasks in a system is much longer than the average execution time, then the average case can be used as its computation time while scheduling the task. The maximum difference between the worst case and average computation times or the maximum difference between optional and mandatory parts of imprecise computations (of all tasks) can be assigned as a backup. If any of the tasks exceed their assigned computation time, then the backup can be used to execute the task to completion. If the probability of such timing faults is low, then the number of backups can be small, since not many tasks will exceed their computation time. If this probability is large, then several backups may be needed. The number of backups is also a function of the criticality of the tasks.

We believe the fault-tolerant RMS scheme is particularly useful in applications such as avionics [18] where RMS is used as a scheduling technique and retry is used as one of the fault tolerance mechanisms. The scheme is also useful in satellites which have a high rate of transient errors [5] and where RMS is used for navigation payload software [7].

References

[1] N.C. Audsley, A. Burns, M.F. Richardson, and K. Tindell. Applying New

Scheduling Theory to Static Priority Pre-emptive Scheduling. *Software Engineering Journal*, 8(5):284–292, Sept 1993.

[2] L. Sha B. Sprunt and J. Lehoczky. Aperiodic Task Scheduling for Hard-Real-Time Systems. *Journal of Real-Time Systems*, pages 27–60, 1989.

[3] J.T. Baldwin. Predicting and Estimating Real-Time Performance. *Embedded Systems Programming*, 8(2), Feb. 1995.

[4] A. Burns, R. Davis, and S. Punnekkat. Feasibility Analysis of Fault-Tolerant Real-Time Task Sets. In *8th Euromicro Workshop on Real-Time Systems*, Jun 1996.

[5] A. Campbell, P. McDonald, and K. Ray. Single Event Upset Rates in Space. *IEEE Trans. on Nuclear Science*, 39(6):1828–1835, Dec 1992.

[6] X. Castillo, S.R. McConnel, and D.P. Siewiorek. Derivation and Caliberation of a Transient Error Reliability Model. *IEEE Trans. on Computers*, C-31(7):658–671, July 1982.

[7] L. Doyle and J. Elzey. Successful Use of Rate Monotonic Theory on a Formidable Real Time System. In *11th IEEE Workshop on Real-Time Operating Systems and Software*, pages 74–78. IEEE, May 1994.

[8] J. Gaisler. Concurrent Error-detection and Modular Fault-tolerance in a 32-bit Processing Core for Embedded Space Flight Applications. In *Symp. on Fault Tolerant Computing (FTCS-24)*, pages 128–130. IEEE, 1994.

[9] S Ghosh. *Guaranteeing Fault Tolerance through Scheduling in Real-Time Systems*. PhD thesis, University of Pittsburgh, Aug 1996.

[10] S. Ghosh, R. Melhem, and D. Mossé. Enhancing Real-Time Schedules to Tolerate Transient Faults. In *Real-Time Systems Symposium*, Dec 1995.

[11] M.S. Hecht, J.B. Hammer, C.D. Locke, J.D. Dehn, and R. Bohlmann. Rate Monotonic Analysis of a Large, Distributed System. In *IEEE Workshop on Real-Time Applications*, pages 4–7. IEEE, July 1994.

[12] R.K. Iyer, D.J. Rossetti, and M.C. Hsueh. Measurement and Modeling of Computer Reliability as Affected by System Activity. *ACM Trans. on Computer Systems*, 4(3):214–237, Aug. 1986.

[13] J.R. Kane and S.S. Yau. Concurrent Software Fault Detection. *IEEE Transactions on Software Engineering*, SE-1(1):87–99, Mar. 1975.

[14] D. Katcher. *Engineering and Analysis of Real-Time Operating Systems*. Ph.D. Thesis, Carnegie Mellon University, Aug 1994.

[15] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner, and J. Reisinger. Tolerating Transient Faults in MARS. In *Symp. on Fault Tolerant Computing (FTCS-20)*, pages 466–473. IEEE, 1990.

[16] C. M. Krishna and K. Shin. On Scheduling Tasks with a Quick Recovery from Failure. *IEEE Trans on Computers*, 35(5):448–455, May 1986.

[17] C.M. Krishna and A.D. Singh. Reliability of Checkpointed Real-Time Systems Using Time Redundancy. *IEEE Trans. on Reliability*, 42(3):427–435, Sept 1993.

[18] R. Lachenmaier and T. Stretch. The IEEE Scalable Coherent Interface: An Approach for a Unified Avionics Network. In *Advanced Packaging Concepts for Digital Avionics*. IEEE, June 1994.

[19] J. P. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proc. IEEE Real-Time Syst. Symp.*, pages 166–171, Dec. 1989.

[20] J. P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced Aperiodic Responsiveness in Hard-Real-Time Environments. In *Real-Time Systems Symposium*, pages 261–270, 1987.

[21] A.L. Liestman and R.H. Campbell. A Fault-tolerant Scheduling Problem. *Trans Software Engineering*, SE-12(11):1089–1095, Nov 1988.

[22] C. L. Liu and J. Layland. Scheduling Algorithm for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM.*, 20(1):46–61, Jan. 1973.

[23] J. W. Liu, K. Lin, C. L. Liu, and C. W. Gear. Research on Imprecise Computations in Project QuartZ. In *Proc. of Workshop on Operating Systems for Mission Critical Computing*, Sept. 1989.

[24] A. Mahmood and E. J. McCluskey. Concurrent Error Detection Using Watchdong Processors - A Survey. *IEEE Transactions on Computers*, 37(2):160–174, Feb. 1988.

[25] A. Mehra, J. Rexford, H.S. Ang, and F. Jahanian. Design and Evaluation of a Window-Consistent Replication Service. In *Proc. Real-Time Technology and Applications Symp.*, 1995.

[26] G. Miremadi and J. Torin. Evaluating Processor-Behavior and Three Error-Detection Mechanisms Using Physical Fault-Injection. *IEEE Transactions on Reliability*, 44(3):441–453, Sept. 1995.

[27] D. Mossé, R. Melhem, and S. Ghosh. Analysis of a Fault-Tolerant Multiprocessor Scheduling Algorithm. In 24^{th} *Int'l Symposium on Fault-Tolerant Computing*, Austin, TX, June 1994. IEEE.

[28] Y. Oh. *The Design and Analysis of Scheduling Algorithms for Real-Time and Fault-Tolerant Computer Systems*. Ph.D. Thesis, University of Virginia, May 1994.

[29] Y. Oh and S.H. Son. Enhancing Fault-Tolerance in Rate-Monotonic Scheduling. *The Journal of Real-Time Systems*, 7(3):315–329, Nov 1994.

[30] M. Pandya and M. Malek. Minimum Achievable Utilization for Fault-tolerant Processing of Periodic Tasks. Technical Report TR 94-07, Univ of Texas at Austin, Dept of Computer Science, 1994.

[31] S. Ramos-Thuel. *Enhancing Fault Tolerance of Real-Time Systems through Time Redundancy*. Ph.D. Thesis, Carnegie Mellon University, May 1993.

[32] S. Ramos-Thuel and J.K. Strosnider. Scheduling Fault Recovery Operations for Time-Critical Applications. In *4th IFIP Conference on Dependable Computing for Critical Applications*, Jan 1995.

[33] B. Randell. System Structure for Software Fault Tolerance. *IEEE Trans. on Software Engineering*, SE-1(2):220–232, June 1975.

[34] M.A. Schuette and J.P. Shen. Processor Control Flow Monitoring Using Signatured Instruction Streams. *IEEE Transactions on Computers*, C-36(3):264–275, Mar. 1987.

[35] D.P. Siewiorek, V. Kini, H. Mashburn, S. McConnel, and M.Tsao. A Case Study of C.mmp, Cm*, and C.vmp: Part 1 - Experiences with Fault Tolerance in Multiprocessor Systems. *Proceedings of the IEEE*, 66(10):1178–1199, Oct. 1978.

[36] S.S. Yau and F.C. Chen. An Approach to Concurrent Control Flow Checking. *IEEE Transactions on Software Engineering*, SE-6(2):126–137, Mar. 1980.