Minimum Achievable Utilization for Fault-Tolerant Processing of Periodic Tasks [*]

Mihir Pandya Miroslaw Malek
Department of Electrical and Computer Engineering,
The University of Texas at Austin,
Austin, Texas 78712.

March 23, 1994

Abstract

Rate Monotonic Scheduling (RMS) policy is a widely accepted scheduling strategy for real-time systems due to strong theoretical foundations and features attractive to practical uses. For a periodic task set of n tasks with deadlines at the end of task periods, it guarantees a feasible schedule on a single processor as long as the utilization factor of the task set is below $n(2^{1/n}-1)$ which converges to 0.69 for large n. We analyze the schedulability of the set of periodic tasks that are scheduled by RMS policy and are susceptible to a single fault. The recovery action is the re-execution of all uncompleted tasks. The priority of the RMS policy is maintained even during recovery. Under these conditions, we guarantee that no tasks will miss a single deadline even in the presence of a fault if the utilization factor on the processor does not exceed 0.5. Thus 0.5 is the minimum achievable utilization that permits recovery from faults before the expiration of the deadlines of the tasks. This bound is larger than $0.69/2 = 0.345$ that would be obtained if computation times were doubled to provide for re-executions in RMS analysis. This result provides scheduling guarantees for tolerating a variety of intermittent and transient hardware and software faults that can be handled simply by re-execution. In addition, we demonstrate how permanent faults can be tolerated efficiently by maintaining common spares among a set of processors that are independently executing periodic tasks.

Keywords: fault tolerance, minimum achievable utilization, periodic tasks, rate monotonic scheduling, real-time systems.

[*]This material is based in part upon research supported by ONR Contract N00014-88-K-0543, ONR Grant N0014-91-J-1858 and Texas Advanced Technology Program Grant No. 386

1 Introduction

In the realm of real-time computation, we frequently encounter systems where the tasks are required to execute periodically. Applications where this requirement is common are often found in, for example, process control, space applications, avionics and others. Even when the external events that trigger tasks are not periodic, many real-time systems sample the occurrence of these events periodically and execute the associated tasks during the time slots reserved for them. The sampling rate depends on the expected frequency of the external event. The reason why aperiodic or sporadic tasks are executed in a periodic manner is because the periodic execution is well understood and predictable.

A variety of scheduling policies for periodic real-time systems have been studied. A scheduling policy is defined as optimal if it can schedule any feasible set of tasks if any other policy can also do the same. A system is called a fixed-priority system if all the tasks have fixed priorities and these priorities do not change during run time. Rate Monotonic Scheduling (RMS) has been proven to be an optimal scheduling policy for scheduling a set of fixed priority tasks on a uniprocessor. Earliest-Deadline-First (EDF) is the optimal scheduling policy for a variable priority system. Note that a priority of a task is different from its criticality. The former is some measure that is assigned to the tasks by the scheduling policy to facilitate scheduling whereas the latter is the measure of importance of the task as defined by the application.

RMS is widely used in practice because it can be easily implemented. It is a preemptive policy where the priority of the tasks are assigned in increasing order of their periods and the task of a particular priority preempts any lower priority task. Liu and Layland proved that as long as the utilization factor of a task set consisting of n tasks is less than $n(2^{1/n}-1)$, the task set is guaranteed a feasible schedule on a uniprocessor [1]. This bound approaches 0.69 as n goes to infinity. However, there may exist task sets which have utilization factors above this bound and still may be feasibly scheduled. The stochastic analysis of the breakdown utilization factor for randomly generated task sets is presented in [2].

The problem for scheduling periodic tasks on multiprocessors is considered in [3] [4] [5]. It is easy to demonstrate that neither the RMS nor the EDF algorithms are optimal for scheduling a set of periodic tasks on a multiprocessor system among fixed and variable priority algorithms respectively [3]. In fact, no scheduling policy is proven to be optimal for a multiprocessor system.

Another issue in real-time computing that is currently gaining increased attention of researchers is fault tolerance. Computers are being introduced to a great extent in critical applications and more reliance is being placed on them while reducing human intervention to a minimum. In situations where the demand for hard real-time processing merges with catastrophic consequences of failures, it is not difficult to imagine why fault tolerance must be provided. Responsive systems [6] which must perform computations to successfully meet their deadlines even in the presence of faults are indispensable in many applications. This paper contributes to an evolving framework for the design and implementation of responsive systems. Our goal in this paper is to investigate the issues of fault tolerance in a system of real-time periodic tasks employing Rate Monotonic Scheduling. Previous work has usually addressed software faults where each task has primary and an alternate code. In [7], an off-line scheduling strategy is considered for periodic tasks where the period of a particular task is an integral multiple of the next lower task period. The alternates are scheduled by RMS policy first and then an effort is made to include the maximum number of primary executions in the schedule. A similar problem of scheduling alternate versions of programs called ghosts is considered in [8]. Dynamic programming is used to perform scheduling and an attempt is made to minimize a cost function. A load balancing scheme is presented for periodic task sets scheduled by RMS in [9] where the neighbors of a faulty processor on a ring take over its tasks which are then eventually distributed to the other processors. However, there is no consideration of missing deadlines due to an overload caused by task migration in response to a fault.

In this paper, we address the schedulability criterion of a set of periodic tasks for fault-tolerant processing. Specifically, we prove that the minimum achievable utilization is 0.5 for a set of periodic tasks executing in an environment that is susceptible to the occurrence of a single fault where the recovery action is to recompute all the partially executed tasks. This result guarantees that all the tasks will meet their deadlines even in the presence of a fault if the utilization factor of the task set on a processor is less than 0.5. The classes of faults that can be tolerated include intermittent and transient hardware and software faults. In addition, permanent crash and incorrect computation faults can also be handled by providing spares to perform recovery and subsequent execution of the task set.

The paper is organized as follows: in Section 2, we provide the background, explain the problem and declare the assumptions. In the following section we present the proof of our assertion that the minimum achievable utilization is 0.5. In Section 4, we address practical and implementation issues. Our conclusions are given in the final section.

2 Background, problem statement and assumptions

As has been mentioned in the Introduction, RMS has a strong theoretical foundation and is widely used in practice due to its simplicity. Rate Monotonic Scheduling policy assigns priorities to tasks in the increasing order of their periods. Consider a set S of n tasks. Each Task i is described by a tuple (C_i, T_i, R_i) where C_i is the execution time of the task, T_i is the period and R_i is its release time, i.e., the time when the first invocation of the task occurs. Thus

$$S = \{(C_i, T_i, R_i) | i = 1, \ldots, n\}$$

We will assume that tasks are labeled in such a manner that $T_1 < T_2 < \ldots < T_n$. A task is expected to complete its computation prior to the end of its period. Thus the j^{th} *instance* $(j = 1, 2, \ldots)$ of the Task i is ready for execution at time $R_i + (j-1) * T_i$ and has a deadline for completion at $R_i + j * T_i$. We assume that we are dealing with a hard real-time system and the aim is to meet the deadlines under all conditions as opposed to soft real-time systems where the deadlines may be missed and the aim is to reduce the delay. In this paper, when not explicitly mentioned, all R_i's are assumed to be zero.

The execution of the tasks is preemptive, i.e., during the execution of a Task i, if any higher priority Task k is ready for execution, the computation of Task i is interrupted and it remains suspended until Task k completes its execution. Then Task i continues from the state at which it was suspended, provided no other task of higher priority is waiting for execution. It is usually assumed that the time to swap tasks is negligible, or that it is accounted for in the computation time. Note that the definition of preemption is recursive, i.e, if Task k has interrupted Task i, it can itself be interrupted by another task of still higher priority. The RMS is a fixed priority policy since the priorities of tasks remain static and do not change during the course of execution of the tasks. The priorities are assigned in the increasing order of the task periods. The task with the smallest period is assigned the highest priority and the task with the largest period the lowest. We will call the *arrival time* of the task as that instant at which it is ready for execution, i.e., $R_i, R_i + T_i, R_i + 2T_i, \ldots$ and its deadline as the next arrival of the same task. The *departure time* of a task is defined as the time instant when the task completes its execution. Thus the arrival time of the j^{th} instance of Task i is $R_i + (j-1)T_i$ but its departure time cannot be defined easily because it depends on the parameters of higher priority tasks.

The utilization factor U of a task set is defined as

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n}$$

For a single processor system, a task set is said to fully utilize the processor under a scheduling algorithm if the task set can be feasibly scheduled using the algorithm but increasing any of the C_is will cause the schedule to be infeasible. The least upper bound of the utilization factor is the minimum of the utilization factors for all possible task sets that fully utilize the processor [1] and is also called the minimum achievable utilization [3]. If the task set has a utilization factor which is less than the minimum achievable utilization, then it is guaranteed a feasible schedule. From [1], for a task set with n tasks, the minimum achievable utilization is $n(2^{1/n} - 1)$. As $n \to \infty$, the minimum achievable utilization converges to ln2 which is approximately 0.69.

2.1 Fault classification

In any discussion on fault tolerance, it is necessary to consider the issue of fault assumptions because it has a significant impact on the design of the system. Under a *crash fault* model, the processor is either operating correctly or, if a fault occurs, does not respond at all to any event, internal or external. An *incorrect computation fault* assumption considers that the processor may fail to produce a correct result in response to correct inputs. For issues related to fault diagnosis and consensus in fault-tolerant processing, the reader can refer to [10].

In addition, faults are also classified as permanent, intermittent and transient [11]. A permanent or hard fault is an erroneous state that is continuous and stable. An intermittent fault occurs occasionally due to unstable nature of hardware. A transient fault results from temporary environmental conditions. A permanent fault can be tolerated only by providing spares which take over the tasks of a primary processor when the fault occurs. Intermittent and transient faults can be tolerated by repeating the computations.

2.2 Analysis of the problem

In general, scheduling problem is concerned with allocating shared resources to multiple processes who need the resources simultaneously. This allocation is performed while attempting to achieve certain prespecified goals. In traditional computers, the goal is usually to minimize the total time or increase the response time for all the requests. However, in real-time systems, the goal is simply to allocate the resources in such a manner that the deadlines associated with the tasks are met. In this paper, as we are dealing with scheduling tasks for execution, the resources are the processors. For hard real-time systems, the scheduler has to be such that all tasks are guaranteed to be completed before their deadlines.

When real-time systems are to be used for critical applications, it is necessary that the system survives in spite of faults that may arise in the system. Unlike non-real-time systems where the occurrence of faults and subsequent recovery may be permitted to cause delays, it is imperative that the results of computations in real-time systems meet the deadline even in presence of faults. Thus the notion of guaranteeing a feasible schedule has to be extended to cover the random events of fault occurrences. This is a challenging endeavor which has to be addressed nevertheless. In this paper, we will consider fault tolerance strategies for a set of periodic tasks executed under RMS policy which will *guarantee that no task will miss even a single deadline* due to the occurrence of a fault at any random moment subject to the fault assumptions explicitly stated therein *and maintaining the priority of the RMS policy.*

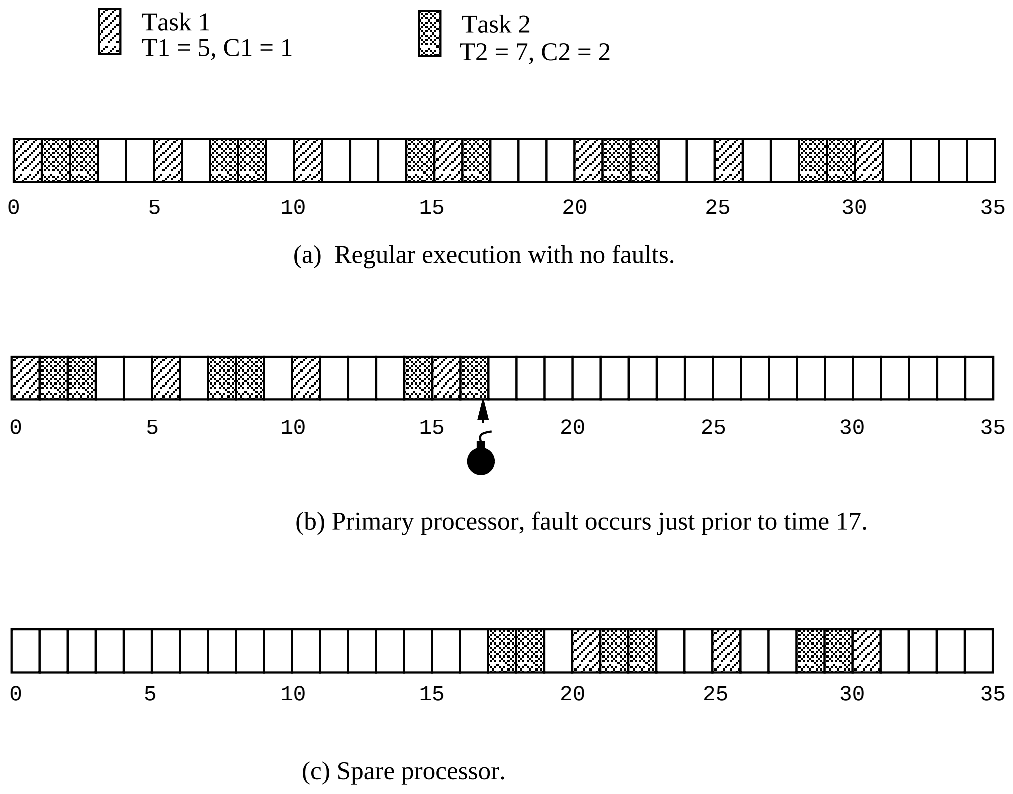
When one considers introducing fault tolerance into the computation, a host of issues need to be considered in addition to those already existing. The only means of providing fault tolerance is by introducing redundancy in the system. The selection of the appropriate level of time and/or space redundancy is driven by the requirements of the application. Redundancy is provided by creating replicas at some level of computation, usually at the task level in real-time systems. Time redundancy is provided by re-executing the task multiple number of times. The original execution and re-executions can all be performed on a single processor or on different processors. The choice is dependent on the fault model assumption. For real-time systems, time redundancy is the most desirable choice, provided that there is sufficient laxity in the deadlines and there is enough spare capacity that other tasks do not miss their deadlines. This will allow maximum utilization of the available resources. However, if the deadlines are stringent and very little laxity is available, space redundancy is the only choice. Thus an ideal design is one which effectively resolves a tradeoff between these two choices such that minimum cost overhead is incurred and all tasks are guaranteed to meet their deadlines under the fault assumptions. This space-time tradeoff is fundamental to the design of responsive computer systems. The result presented here optimizes the tradeoff to provide scheduling guarantees for a single fault in an environment for periodic tasks.

2.3 Single fault with re-execution of task for recovery

We analyze the following scenario:

- A set of tasks is executing on a single processor and the tasks are scheduled by the RMS policy.
- All the tasks are independent.
- A fault may occur at any instant.
- The interval between successive faults is greater than the largest period in the task set.
- The fault is detected before the next occurrence of a departure of a task from the processor. For example, if a lower priority tasks is executing during the occurrence of a fault and some time later another higher priority task is supposed to preempt the first task, the fault should be detected before the higher priority task is expected to depart under normal execution.
- The recovery action is to re-execute all the partially executed tasks at the instant of the fault detection. This includes the currently executing task and all the preempted tasks.
- The tasks are required to meet their deadlines even if they have to be re-executed due to the occurrence of a fault.
- The priorities of the RMS policy are maintained even during recovery. Maintaining the priorities of tasks is very important since RMS is a fixed priority scheduling policy and the priorities are assigned at system design time. This approach simplifies the design process because the designer does not have to worry about assigning separate priorities for recovery and analyze the effect of the change in priorities on the schedulability of the task set.

One should note that at this stage that we do not place any restrictions on the kind of faults that can be tolerated or the architecture of the system. As long as these conditions are satisfied by the design, the results of this paper are valid. For example, if one were to consider a hardware permanent crash fault, the recovery and subsequent computation would have to be performed on



Task 1
T1 = 5, C1 = 1
Task 2
T2 = 7, C2 = 2
0
5
10
15
20
25
30
35
(a) Regular execution with no faults.
(b) Primary processor, fault occurs just prior to time 17.
(c) Spare processor.

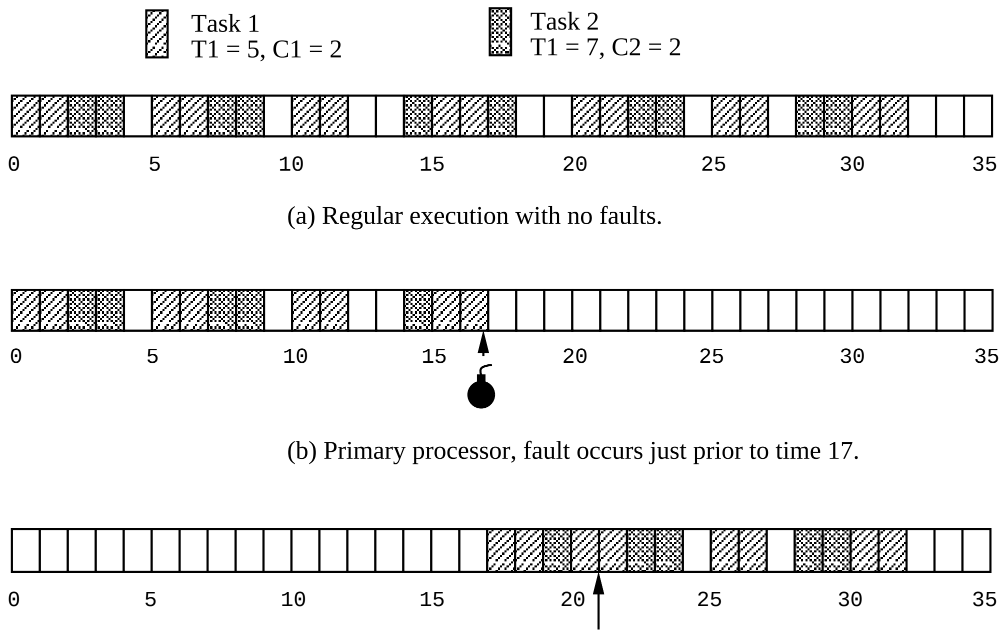
Figure 1: Feasible schedule in presence of a fault.

a spare processor. On the other hand, if a software fault occurs, the recovery is possible on the primary processor itself. An incorrect computation fault can be handled if the fault is detected, perhaps by consistency checks, before the task is expected to depart. In addition, the recovery program for a task need not be the same as the one that is normally executed as long as its computation time is less than or equal to the computation time of the primary code.

Two examples are shown in Figures 1 and 2. Both of them consider a task set consisting of two tasks with periods 5 and 7. In these examples, we assume crash faults of processors. In Figure 1, $C_1 = 1$ and $C_2 = 2$ and the processor state as a function of the time is shown under regular execution in Figure 1(a). We observe that the schedule is feasible when no fault occurs. Figures 1(b) and 1(c) show the state of the processor and the spare respectively when a fault occurs just prior to the time instant 17. The fault occurs before Task 2 could complete and so it is re-started on the spare and it meets the deadline of time 21.

Figure 2(a) shows the execution profile of the two tasks whose periods are again 5 and 7 respectively. However, in this example $C_1 = 2$ and $C_2 = 2$. Though the schedule is feasible when no fault occurs, the same is not true when a fault causes the recovery action to be taken. The arrival of Task 1 at time 15 preempts Task 2 and a fault occurs just prior to its completion at time 17. So the spare restarts the execution of both tasks, starting with Task 1 as it is a higher priority task. Task 1 completes at time 19 and manages to meet its deadline of time 20. The re-execution of the Task 2 starts at time 19 but is preempted at time 20 by the arrival of the next instance of Task 1 and so Task 2 misses its deadline of time 21.

It seems obvious from these examples that certain amount of time redundancy should be provided for recovery and that the RMS scheduling criteria ($U < 0.69$) is not sufficient. A trivial


Task 1
T1 = 5, C1 = 2
Task 2
T1 = 7, C2 = 2
0
5
10
15
20
25
30
35
(a) Regular execution with no faults.
(b) Primary processor, fault occurs just prior to time 17.
Task 2 misses deadline

(c) Spare processor.

Figure 2: Infeasible schedule in presence of a fault.

solution is to "reserve" enough space for all tasks so that in the event of a fault, there is enough spare capacity in terms of time such that the task can be re-executed and still meet its deadline. Since the worst possible time for a fault to occur is just prior to the completion of the task, the amount of extra time to be devoted to task i for recovery is an additional C_i. Thus in the Rate Monotonic Analysis of the schedulability of the entire task set, the computation time for all tasks have to be assumed to be $2C_i$. This means that, in a general case, the effective minimum achievable utilization on each processor is just 0.345, i.e., half of 0.69. However, the situation is not as pessimistic as it appears. We will prove in the following section that a minimum achievable utilization of 0.5 guarantees enough time redundancy to complete recovery before the deadlines. Thus as long as the utilization factor of a task set on a processor is less than or equal to 0.5, the task set is guaranteed a feasible schedule in presence of a single fault.

2.4 Motivation

One of the popular traditional approaches to the design of fault-tolerant system is the use of N-modular-redundancy (NMR) [11]. In this technique, every processor is provided with extra spares. The spares may be hot, warm or cold. For real-time systems, hot spares is the preferred choice as no time is wasted to perform recovery. A spare is said to be hot if it synchronously performs all the computations with the primary processor and takes over if the primary processor fails. For fault models such as incorrect computation and Byzantine faults, there may not be any distinction between the primary and the spares as they all perform the same computation and vote on the result to mask faulty results. If we assume crash or fail-stop model, NMR requires that each processor be duplicated to tolerate a single fault and so the number of processors in a fault-tolerant system is $2m$ where m is the number of processors in the original system. Such a system, called a duplex system, can tolerate up to one fault between the primary and the spare and up to m faults as long as no more than one fault affects a particular primary and its spare. This is achieved by having the space overhead equal to the size of the original system, i.e., by doubling the space resources.

The space overhead of duplex system is very high for many applications and it is usually desirable to have a single spare for the group of m processors so that if any processor fails, the spare can be substituted in its place. Whereas providing a single spare is a simple feat in non-real-time systems, ensuring that the recovery will be performed within the deadlines is not easy. The contributions of this paper makes it easy to guarantee recovery by limiting the utilization factor on a processor at 0.5. If U_S is the total utilization factor of a large set of tasks, the number of processors needed in a system with a single spare is $\lceil U_S/0.5 \rceil + 1$. This assumes crash faults and even distribution of the utilization factor. This is likely to be significantly less than $2 * \lceil U_S/0.69 \rceil$ in the duplex system. Interestingly, the trivial solution to ensure recovery by doubling the computation time requirements will require $\lceil U_S/0.35 \rceil + 1$ processors, which is nearly the same as that required by the duplex system.

In addition to tolerating hardware crash faults, a major application of the result is towards tolerating software faults. We will deal with this in greater detail in Section 4.

3 Determination of minimum achievable utilization

Before we prove that the minimum achievable utilization is 0.5, we present the definitions of some terms used in the proof. The *recovery* is defined as re-execution of *all* the partially executed tasks where the priority of the RMS is maintained. Thus during the recovery of a lower priority task, if a higher priority task arrives, the higher priority task will preempt the recovery of the lower priority task. In addition, if the fault affects multiple tasks, higher priority tasks will perform recovery action first. A schedule is said to be *feasible* for a set of tasks if the task set can be guaranteed a schedule under Rate Monotonic Algorithm (i.e., all tasks will meet their deadlines) *even if recovery has to be performed* due to a single fault that can occur at any arbitrary instant of time. A set of tasks is said to *fully utilize a processor* if the task set has a feasible schedule and increasing the computation time of any task in the set causes the schedule to become infeasible. The *minimum achievable utilization* is the minimum of the utilization factor of every possible sets of tasks that fully utilize the processor. We define a *critical instant* for a task to be that instant at which an arrival of the task will have the largest response time in the presence of some fault. The schedule of a set of tasks that fully utilizes the processor will have at least one critical instant for some task i where the response time is the period of that task. We shall call that time interval between the arrival and the deadline of the task i as the *critical period.*

A fault that occurs just prior to the completion of a task creates the maximum delay for that task and any lower priority tasks that have been interrupted by it. Hence we only need to examine the effects of a fault at the instants when the tasks are about to be completed.

We will consider a number of cases that will lead to the proof of theorem that the minimum achievable utilization is 0.5.

3.1 Case 1: Task set with one task

Consider a task set comprising of a single task (C_1, T_1). In this case, the release time does not matter.

Observation 1 *The minimum achievable utilization for a task set with one task is 0.5.*

Proof: This is obvious since C_1 cannot exceed $T_1/2$. If C_1 equals some value x such that $T_1/2 < x \leq T_1$ and if a fault occurs at some instant t such that $kT_1 + T_1/2 < t < kT_1 + x$,

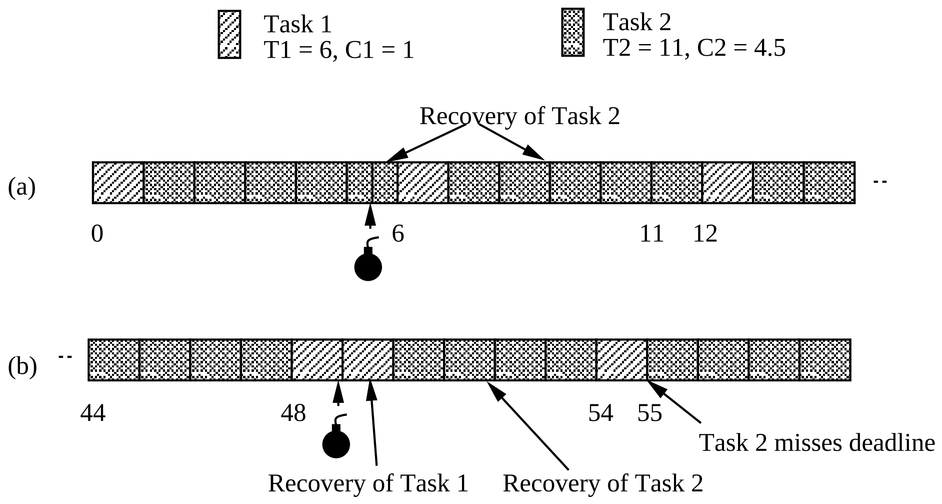


Figure 3: Schedule of two tasks with periods 6 and 11.

$k = 0, 1, \ldots$, then there is not sufficient time to re-execute the task and still meet the deadline at time $(k+1)T_1$. The processor is fully utilized when $C_1 = T_1/2$ and hence $U = 0.5$. $\square$

It is important to note that even if the task set has more than one task, the computation time of each of the tasks cannot exceed half the value of its period, i.e., $C_i \leq T_i/2$, $i = 0, 1, \ldots, n$ where n is the number of tasks in the set.

3.2 Case 2: Task set with two tasks

Before we begin the analysis of the minimum achievable utilization for this case, let us consider the issue of release times. In the traditional RMS analysis the worst delay for the Task 2 is observed when it arrives simultaneously with the Task 1. If the first arrival of the Task 2 can then be feasibly scheduled, any subsequent arrivals will also meet their deadlines and so one has only to consider the feasibility conditions of the simultaneous arrivals of the tasks. This is not necessarily true when one considers the possibility of faults. For example, consider the task set $\{(1,6),(4.5,11)\}$ where release times are zero. By considering just the first arrival, it would appear that the task set has a feasible schedule and the processor is fully utilized. This is shown in Figure 3(a). Tasks 1 and 2 are released simultaneously and since Task 1 has higher priority, it starts execution and departs at time 1 when Task 2 begins. A fault occurs just prior to the completion of Task 2 at time instant 5.5 and it is restarted to perform recovery. Task 1 again arrives at time 6 and it preempts recovery. The recovery just completes at time 11 when the next arrival of Task 2 occurs. However, if a fault occurs just before time instant 49, the schedule is infeasible. This is shown in Figure 3(b). Task 2 arrives at time 44 and is preempted by Task 1 which arrives at time 48. A fault occurs just prior to the completion of Task 1 at time 49 and so both tasks have to be re-executed. Task 1 recovers in time at time instant 50 when the recovery of Task 2 begins. However, the next arrival of Task 1 occurs at time 54 and it preempts the recovery of Task 2 and causes it to miss the deadline at time 55. Only 4 units of time are available to Task 2 for recovery in the time interval 50–54 whereas its computation time is 4.5. Thus the correct value of C_2 that fully utilizes the processor is $C_2 = 4$. Hence, in our analysis, we have to consider all possible values of release times.

Consider a set of two tasks, $\{(C_1, T_1, R_1), (C_2, T_2, R_2)\}$ with arbitrary release times. We will first consider the case when $T_2 \geq 1.5T_1$. Next we will consider various subcases when $T_2 < 1.5T_1$.

3.2.1 Case 2a: $T_2 \geq 1.5T_1$

Theorem 1 *The minimum achievable utilization is 0.5 for a set of two tasks satisfying the condition* $T_2 \geq 1.5T_1$.

Proof: We first prove that as long as the utilization factor is less than or equal to 0.5, a feasible schedule is guaranteed for the task set; then we give a particular instance where the processor is fully utilized and the utilization factor is 0.5.

From Observation 1, it is clear that $C_1 \leq T_1/2$. Within any interval $[R_2 + kT_2, R_2 + (k+1)T_2)$, $k = 0, 1, \ldots$, there are at most $\lceil T_2/T_1 \rceil$ arrivals of Task 1. The worst possible scenario is when Task 2 is about to be completed and is preempted by the arrival of Task 1 and the fault occurs just prior to completion of Task 1. In this case both Tasks 1 and 2 need to be executed again. Task 1 will meet its deadline since $C_1 \leq T_1/2$. Task 2 will meet its deadline if the following condition is satisfied:

$$\left(\left\lceil \frac{T_2}{T_1} \right\rceil + 1\right) C_1 + 2C_2 \leq T_2 \tag{1}$$

i.e. if

$$\left(\left\lceil \frac{T_2}{T_1} \right\rceil + 1\right) \frac{C_1}{T_2} + \frac{2C_2}{T_2} \leq 1 \tag{2}$$

Under traditional RMS analysis, the feasibility condition is $(\lceil T_2/T_1 \rceil)C_1 + C_2 \leq T_2$. But in a fault-tolerant system, each task will have to be executed once more under the worst case scenario of the occurrence of a single fault.

Assume that the utilization factor of the task set is less than or equal to 0.5, i.e.,

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} \leq 0.5 \tag{3}$$

Therefore,

$$\begin{aligned} &\frac{2C_1}{T_1} + \frac{2C_2}{T_2} \leq 1 \\ &\Rightarrow \frac{2C_1}{T_1} - \left(\left\lceil \frac{T_2}{T_1} \right\rceil + 1\right) \frac{C_1}{T_2} + \left(\left\lceil \frac{T_2}{T_1} \right\rceil + 1\right) \frac{C_1}{T_2} + \frac{2C_2}{T_2} \leq 1 \end{aligned} \tag{4}$$

Thus the feasibility condition given by Equation 2 is guaranteed if

$$\begin{aligned} &\frac{2C_1}{T_1} - \left(\left\lceil \frac{T_2}{T_1} \right\rceil + 1\right) \frac{C_1}{T_2} \geq 0 \\ &\Rightarrow \frac{2C_1}{T_1} - \left(\left\lceil \frac{T_2}{T_1} \right\rceil + 1\right) \frac{C_1}{T_1}\frac{T_1}{T_2} \geq 0 \\ &\Rightarrow \frac{C_1}{T_1}\left[2 - \left(\left\lceil \frac{T_2}{T_1} \right\rceil + 1\right) \frac{T_1}{T_2}\right] \geq 0 \\ &\Rightarrow 2 - \left(\left\lceil \frac{T_2}{T_1} \right\rceil + 1\right) \frac{T_1}{T_2} \geq 0 \\ &\Rightarrow \left(\left\lceil \frac{T_2}{T_1} \right\rceil + 1\right) \frac{T_1}{T_2} \leq 2 \end{aligned} \tag{5}$$

Equation 5 is satisfied when $T_2/T_1 \geq 1.5$, i.e., when $T_2 \geq 1.5T_1$. Thus any task set satisfying the condition of the Theorem 1 is guaranteed a feasible schedule if the utilization factor is less that or equal to 0.5.

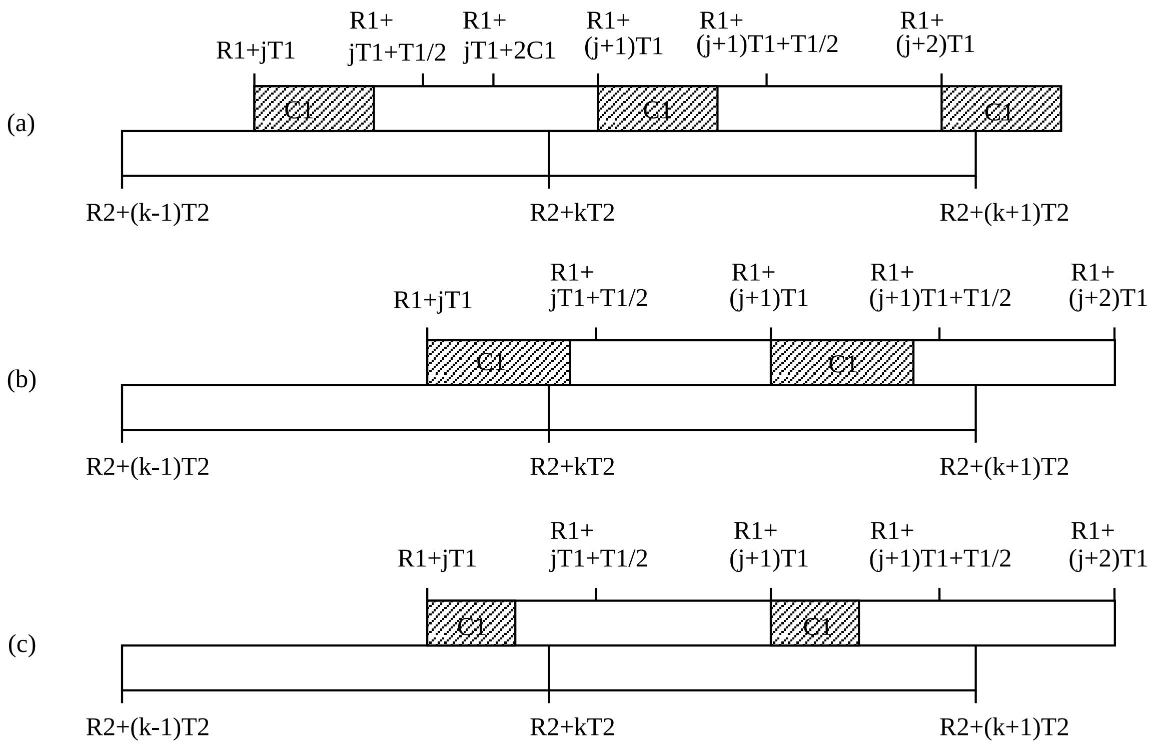


Figure 4: Modeling subsequent arrivals of the tasks.

Now consider the cases when $C_1 = T_1/2$, $C_2 = 0$ and $C_1 = 0$, $C_2 = T_2/2$. In each of these two cases, the processor is fully utilized since increasing C_2 in the first case and C_1 in the second case causes the schedule to become infeasible. In both cases, the utilization factor is 0.5. We have also proved that as long as the utilization factor is less than or equal to 0.5, the tasks can be feasibly scheduled. Hence, when $T_2 \geq 1.5T_1$, the minimum achievable utilization is 0.5. $\square$

3.2.2 Case 2b: $T_2 < 1.5T_1$

We will take the following approach in our proof for this proof: We will first show that each instance of a task can be modeled as the arrival of the first instance with some values of release times R'_1 and R'_2. Then we will prove that the first instances can be feasibly scheduled for all possible values of release times as long as the utilization factor is less than or equal to 0.5, i.e., we will prove that the minimum achievable utilization among all task sets that fully utilize the processor during the first instance is 0.5. Also, without loss of generality, we can assume that one of R_1 or R_2 is zero and $R_1 < T_2$, $R_2 < T_1$.

Consider Figure 4 where we are interested in the feasibility of meeting the deadline at time instant $R_2 + (k+1)T_2$ of the $(k+1)^{th}$ instance of Task 2 that arrives at time instant $R_2 + kT_2$. We consider various cases below where $R_1 + jT_1 \leq R_2 + kT_2 < R_1 + (j+1)T_1$.

- If $R_2 + kT_2 \geq R_1 + jT_1 + 2C_1$ as shown in Figure 4(a), the $(k+1)^{th}$ instance of the Task 2 can be modeled as the first instance of the Task 2 in the task set $\{(C_1, T_1, R'_1 = R_1 + (j+1)T_1 - R_2 - kT_2), (C_2, T_2, R'_2 = 0)\}$. This is possible because any fault during the execution of the $(j+1)^{th}$ instance of Task 1 does not affect the schedulability of the $(k+1)^{th}$ instance of Task 2.

- If $R_2 + kT_2 < R_1 + jT_1 + 2C_1$ as shown in Figures 4(b) and (c), the $(k+1)^{th}$ instance of the Task 2 can be modeled as the first instance of the Task 2 in a task set $\{(C_1, T_1, R_1' = 0), (C_2, T_2, R_2' = R_2 + kT_2 - R_1 - jT_1)\}$.

In the Appendix, we consider all possible cases of the release times and the periods of the tasks. For each of those cases, we present the value of the task computation times that fully utilize the processor during the first instance of the Task 2. For each of those cases, we prove that when the processor is fully utilized during the first instance of Task 2, the utilization factor is greater then 0.5.

Theorem 2 *The minimum achievable utilization for a set of two tasks satisfying condition $T_2 < 1.5T_1$ is 0.5.*

Proof: We have shown that any subsequent instance of two tasks after the first instance can be modeled as the first instance with some release times. Then we have proved in the Lemmas 3–14 in the Appendix that for all possible values of release times, if the processor is fully utilized for the first instance, the utilization factor is greater than or equal to 0.5. Hence, the minimum achievable utilization for a set of two tasks satisfying condition $T_2 < 1.5T_1$ is 0.5. □

3.3 Case 3: Task set with $n > 2$ tasks

Consider a set of n tasks

$$S_n = \{(C_1, T_1, R_1), (C_2, T_2, R_2), \ldots, (C_n, T_n, R_n)\}$$

whose utilization is

$$U_n = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

We will prove by induction that the minimum achievable utilization for a set of n tasks is 0.5. Let us assume that the minimum achievable utilization for a set of $n-1$ tasks is 0.5. We will prove that this is also true for a set of n tasks.

Consider the set of the first $n-1$ tasks

$$S_{n-1} = \{(C_i, T_i, R_i) | i = 1, \ldots, n-1\}$$

whose utilization is

$$U_{n-1} = \sum_{i=1}^{n-1} \frac{C_i}{T_i}$$

If both sets S_n and S_{n-1} have a feasible schedule and $U_{n-1} \geq 0.5$ then $U_n \geq 0.5$ (because $U_n \geq U_{n-1}$). Thus we need to consider only those cases where $U_{n-1} < 0.5$. But since $U_{n-1} < 0.5$, S_{n-1} will have a feasible schedule because of our assumption. Thus we only need to consider the feasibility of scheduling the Task n.

3.3.1 Case 3a: $T_n \geq 1.5T_{n-1}$

Theorem 3 *The minimum achievable utilization is 0.5 for a set of n tasks satisfying the condition $T_n > 1.5T_{n-1}$ and assuming that the minimum achievable utilization of any set of $n-1$ tasks is 0.5.*

As in the case of a set of two tasks, if the following condition representing the worst possible scenario is satisfied, the corresponding task set has a feasible schedule. Note that the reverse is not true, i.e., the task set may not satisfy the following condition and still have a feasible schedule.

$$\sum_{i=1}^{n-1} \left(\left\lceil \frac{T_n}{T_i} \right\rceil + 1 \right) C_i + 2C_n \leq T_n \tag{6}$$

i.e.,

$$\sum_{i=1}^{n-1} \left(\left\lceil \frac{T_n}{T_i} \right\rceil + 1 \right) \frac{C_i}{T_n} + \frac{2C_n}{T_n} \leq 1 \tag{7}$$

Assume that $U_n \leq 0.5$. Therefore

$$\sum_{i=1}^{n-1} \frac{C_i}{T_i} + \frac{C_n}{T_n} \leq 0.5$$

$$\Rightarrow \sum_{i=1}^{n-1} \frac{2C_i}{T_i} + \frac{2C_n}{T_n} \leq 1$$

$$\Rightarrow \sum_{i=1}^{n-1} \left[\frac{2C_i}{T_i} - \left(\left\lceil \frac{T_n}{T_1} \right\rceil + 1 \right) \frac{C_i}{T_n} \right] + \sum_{i=1}^{n-1} \left(\left\lceil \frac{T_n}{T_i} \right\rceil + 1 \right) \frac{C_i}{T_n} + \frac{2C_n}{T_n} \leq 1$$

Thus the condition in Equation 7 is guaranteed if

$$\sum_{i=1}^{n-1} \left[\frac{2C_i}{T_i} - \left(\left\lceil \frac{T_n}{T_1} \right\rceil + 1 \right) \frac{C_i}{T_n} \right] \geq 0 \tag{8}$$

If $T_n \geq 1.5T_{n-1}$, then $T_n > 1.5T_i$, $i = 1, \ldots, n-2$ because $T_n > T_{n-1} > \cdots > T_2 > T_1$. Whenever $T_n > 1.5T_i$, $i = 1, \ldots, n-1$, then $\left[\frac{2C_i}{T_i} - \left(\left\lceil \frac{T_n}{T_1} \right\rceil + 1 \right) \frac{C_i}{T_n} \right] \geq 0$. Thus the sum is also non-negative and Equation 7 is satisfied and the task set is guaranteed a feasible schedule. Thus for all sets of n tasks, the minimum achievable utilization is 0.5 if $T_n \geq 1.5T_{n-1}$. □

3.3.2 Case 3b: $T_n < 1.5T_{n-1}$

When $T_n < 1.5T_{n-1}$, we will consider two subcases in the following lemmas. Assume that the set of tasks S_n fully utilizes the processor. We note that the set S_{n-1} does not fully utilize the processor since $U_{n-1} < 0.5$. Add task n to the set S_{n-1} and increment its computation time till the processor is fully utilized and this value of the computation time is C_n. Hence only the task n has at least one critical period where the occurrence of a fault and subsequent recovery will cause the task to just meet its deadline. There are two possible cases: the worst case instant of the occurrence of a fault is just prior to completion of the task n itself in which case the recovery is solely the re-execution of only the task n, or, the worst case instant of occurrence of a fault is just prior to the completion of some other task i, $1 \leq i \leq n-1$. In the former case,

$$C_n = \frac{T_n - x_1 C_1 - x_2 C_2 - \cdots - x_{n-1} C_{n-1}}{2}$$

where x_i is the fraction of the time that the processor spends executing the task i *in the critical period* of Task n and $x_i \leq \lceil T_n / T_i \rceil$. In the latter case,

$$C_n = T_n - y_1 C_1 - y_2 C_2 - \cdots - y_{n-1} C_{n-1}$$

where y_i is the fraction of the time that the processor spends in the normal execution *and recovery* of the Task i in the critical period of Task n. Here, $y_i \leq \lceil T_n/T_i \rceil + 1$.

Lemma 1 *The minimum achievable utilization is 0.5 for a set of n tasks satisfying the conditions $T_n < 1.5T_{n-1}$ and $C_n = (T_n - x_1C_1 - x_2C_2 - \cdots - x_{n-1}C_{n-1})/2$ where $x_i \leq \lceil T_n/T_i \rceil$, assuming that the minimum achievable utilization for a set of $n-1$ tasks is 0.5.*

Construct a set S'_{n-1} of $n-1$ tasks as follows:

$$S'_{n-1} = \{(C_1, T_1, R_1), (C_2, T_2, R_2), \ldots, (C_{n-2}, T_{n-2}, R_{n-2}), (C_{n-1}\frac{T_n}{T_{n-1}} + C_n, T_n, R_n)\}$$

The utilization factor U'_{n-1} of the set S'_{n-1} is the same as that of S_n, i.e. $U'_{n-1} = U_n$. Now consider a fault just prior to the completion of the task $(C_{n-1}\frac{T_n}{T_{n-1}} + C_n, T_n, R_n)$ during an interval which is a critical interval for the set S_n. The time to completion of the task is

$$\begin{aligned}
t_c &= x_1C_1 + x_2C_2 + \cdots + x_{n-2}C_{n-2} + 2 * (C_{n-1}\frac{T_n}{T_{n-1}} + C_n) \\
&= x_1C_1 + x_2C_2 + \cdots + x_{n-2}C_{n-2} + 2C_{n-1}\frac{T_n}{T_{n-1}} + T_n - x_1C_1 - x_2C_2 - \cdots - x_{n-1}C_{n-1} \\
&= T_n - x_{n-1}C_{n-1} + 2C_{n-1}\frac{T_n}{T_{n-1}} \\
&= T_n + C_{n-1}(2\frac{T_n}{T_{n-1}} - x_{n-1})
\end{aligned}$$

Since $x_{n-1} \leq \lceil T_n/T_{n-1} \rceil$ and since $T_n < 1.5T_{n-1}$, $x_{n-1} \leq 2$ and so

$$t_c > T_n$$

Thus the last task misses the deadline and so the set S'_{n-1} has an infeasible schedule. But since we have assumed that the minimum achievable utilization of a set of $n-1$ tasks 0.5, the utilization factor of S'_{n-1} must exceed 0.5. However, $U_n = U'_{n-1}$ and so $U_n > 0.5$. Thus the minimum achievable utilization of every set of n tasks that satisfy the conditions of this lemma is 0.5. □

Here we have proved that every set of n tasks that fully utilizes the processor and satisfies the conditions of the Lemma 1 can be converted into another set of $n-1$ tasks that has an infeasible schedule. As an example, consider a set of three tasks $S_3 = \{(0.5, 3, 0), (0.5, 4, 0), (1.5, 5, 0)\}$. This task set fully utilizes the processor. From this task set, we construct the set $S'_2 = \{(0.5, 3, 0), (2.125, 5)\}$. The set S'_2 has an infeasible schedule because if a fault occurs at a time just prior to the completion of Task 2 at time instant 2.625, there is not enough spare time to recover.

Lemma 2 *The minimum achievable utilization is 0.5 for a set of n tasks satisfying the conditions $T_n < 1.5T_{n-1}$ and $C_n = T_n - y_1C_1 - y_2C_2 - \cdots - y_{n-1}C_{n-1}$ where $y_i \leq \lceil T_n/T_i \rceil + 1$, assuming that the minimum achievable utilization of a set of $n-1$ tasks is 0.5.*

Assume that the set of tasks S_n fully utilizes the processor. Since the set S_{n-1} does not fully utilize the processor, increment the computation time of the Task $n-1$ in S_{n-1} so that the utilization factor is 0.5. Let this increase be Δ and let the new set be $S'_{n-1} = \{(C_1, T_1, R_1), \ldots, (C'_{n-1}, T_{n-1}, R_{n-1})\}$ with the utilization factor $U'_{n-1} = 0.5$ where $C'_{n-1} = C_n + \Delta$. It is easy to observe that $C_n \geq 2\Delta$. Since the Task $n-1$ is the lowest priority task in the set of $n-1$ tasks, any reduction of the computation time of Δ from C'_{n-1} frees up at least 2Δ amount of time for Task n that will not be interrupted by the other tasks. The amount is 2Δ because reduction of Δ also frees up an extra Δ from recovery. Thus,

$$\begin{aligned} U_n &= \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_{n-1}}{T_{n-1}} + \frac{C_n}{T_n} \\ &\geq 0.5 - \frac{\Delta}{T_{n-1}} + \frac{2\Delta}{T_n} \\ &> 0.5 \end{aligned}$$

□

We now prove the following theorem for the general case.

Theorem 4 *For a set of n tasks, the minimum achievable utilization is 0.5.*

Proof: In Theorem 3 and Lemmas 1 and 2 we have proved that the minimum achievable utilization for a set of n tasks is 0.5 provided that the minimum achievable utilization for a set of $n-1$ tasks is 0.5. In addition, this theorem is true for one task as shown in Observation 1 and has also been proven to be true for a set of two tasks in Theorems 1 and 2. Hence, by induction it is true for all n. □

4 Implementation Issues

4.1 Tolerating hardware crash faults

Consider a distributed system with a common spare. The spare is not idle but it monitors the state of the processors. After completion of each instance of each task, a processor sends a message to the spare indicating that the task is successfully completed. The spare maintains a list of all tasks in the system and the processor on which they are executing. From this information, it can either be provided a look-up table of all completion times of the tasks or these completion times can be easily computed "on-the-fly". Let C_{comm} be the maximum communication latency of the network. If some task was supposed to be completed at time t_c, the spare expects a confirmation by the time $t_c + C_{comm}$. In case this message is not received, the processor is declared faulty and the spare takes over the faulty processor's task set and initiates recovery. In the rate monotonic analysis of the task set on each processor, the communication time and the overhead in reconfiguration is assumed to be included in the computation time of the task. So, if some task i has computation requirement of C_i, then $C'_i = C_i + C_{comm} + C_{overhead}$ is used for analysis. This technique assumes that the communication delays are finite and bounded, which is not an unreasonable assumption for practical applications. It also requires that the executable code of all tasks be accessible to the spare.

As we have discussed in Section 2, the space overhead for guaranteeing deadlines in the presence of a single fault for duplex systems is $2 * \lceil U/0.69 \rceil$ processors. However, the number of processors needed for a system with a single spare with recovery is $\lceil U/0.5 \rceil + 1$. U is the total utilization factor of the task set and we assume that the task set is partitioned so that the utilization factor is evenly distributed. Table 1 shows the number of processors required by each scheme for different values of the utilization factors. We observe that providing a common spare significantly reduces the size of the system and the effect is more pronounced for large values of utilization factors.

Table 1: Number of processors m in systems where computation times are doubled for RMS analysis, duplex systems and in a system with a common spare for recovery for different values of utilization factor

U	Doubling computation time in RMS analysis $m = \left\lceil \frac{U}{0.345} \right\rceil + 1$	Duplex system $m = 2 * \left\lceil \frac{U}{0.69} \right\rceil$	Common spare with recovery $m = \left\lceil \frac{U}{0.5} \right\rceil + 1$
0.5	3	2	2
0.69	3	2	3
1	4	4	3
2	7	6	5
3	10	10	7
4	13	12	9
5	16	14	11
6	19	18	13
7	22	22	15
8	25	24	17
9	28	28	19
10	30	30	21
100	291	290	200

4.2 Tolerating incorrect computation faults caused by hardware fault

Triple Modular Redundancy (TMR) systems are required to tolerate incorrect computation faults. A duplex system can only detect the presence of an incorrect computation fault because the results of the two processors do not agree. A third processor is required so the majority result is assumed to be correct. A similar technique as described above can be used to tolerate a single incorrect computation fault. Rather than having a TMR system, a duplex system with a spare can be used. In case the duplex pair detects an error, the spare is used to perform recovery. The number of processors required for a TMR system is $3 * \lceil U/0.69 \rceil$ whereas the number of processors required for a duplex with a spare for recovery is $2 * \lceil U/0.5 \rceil + 1$. Again, U is the total utilization factor for the entire task set. The number of processors required for both schemes is shown in Table 2. We notice that the duplex with a spare again requires less space overhead as compared to a TMR system. However, the benefit is not as large as that observed for crash faults.

4.3 Tolerating software faults and intermittent and transient hardware faults

We believe that the greatest application of the results of this paper would be towards tolerating software faults and intermittent and transient hardware faults. In space and hostile industrial applications, outside environment conditions such as alpha particles, electrostatic interference, etc., cause transient errors. In addition software faults such as stack overflows in the operating systems, etc., are best handled by re-execution. By limiting the utilization factor to 0.5 on a processor, we can guarantee that recovery can be performed within the deadlines. Even though we consider the re-execution of all partially executed tasks, it is not necessary if a fault affects a single task. That task can be re-executed to meet its deadline and we can be confident that the re-execution will not cause other tasks to miss their deadlines. In addition, the recovery code need not be the same as the

Table 2: Number of processors m in TMR system and in a duplex system with a common spare for recovery for different values of utilization factor

U	TMR system $3 * \left\lceil \frac{U}{0.69} \right\rceil$	Duplex system with common spare $1 + 2 * \left\lceil \frac{U}{0.5} \right\rceil$
0.5	3	3
0.69	3	5
1	6	5
2	9	9
3	15	13
4	18	17
5	21	21
6	27	25
7	33	29
8	36	33
9	42	37
10	45	41
100	435	401

primary code. This is especially true for software faults where an alternate program is desirable. As long as the time to execute the recovery program is less than or equal to the execution time of the primary program, we can be certain that the deadlines will be met.

4.4 Tolerating Multiple Faults

Multiple faults can be tolerated under our analysis as long as the interval between successive faults is larger than the largest period in the task set. Under this assumption, unlimited transient faults can be tolerated and k permanent crash faults can be tolerated by providing k spares and limiting the utilization factor on each processor to 0.5. For certain task sets and k, NMR system will yield lesser space overhead and greater fault coverage and would be easier to implement. This is the case if

$$k + \left\lceil \frac{U}{0.5} \right\rceil > (k+1) \left\lceil \frac{U}{0.69} \right\rceil$$

where U is the utilization factor for the entire task set. Again we assume that the task set is partitioned so that the utilization factor is evenly distributed. For example, if the total utilization factor $U = 0.6$, and $k = 1$, NMR uses only two processors whereas our approach would require three processor. But for most general cases, providing k common spares would result in lesser overheads.

5 Conclusions

We have provided a theoretical foundation for fault-tolerant processing of periodic real-time tasks scheduled by the Rate Monotonic Scheduling policy. Under the scenario that recovery from a fault involves restarting all the partially executed tasks while maintaining the priority levels of RMS policy, we show that the minimum achievable utilization on a processor is 0.5. This result guarantees that all tasks will meet their deadlines even in the presence of a fault if the utilization factor on a processor is restricted to 0.5. This bound is much better than the maximum utilization factor of 0.345 (0.69/2) that would be obtained if the computation times of all tasks were naively doubled in RMS analysis to provide for recovery time. The result provides a framework for tolerating transient and intermittent hardware and software faults where re-execution is the preferred recovery technique. In addition, this result is applicable to tolerating permanent crash and incorrect computation faults where spares must be employed to replace faulty processors. In such a system we show that the space redundancy achieved by maintaining a common pool of spares is, in most cases, less than that of an NMR system.

The contributions of this paper form an important component in the evolution of Responsive Systems. The concept of providing guarantees of meeting the deadlines in the system in spite of the occurrence of faults is integral to the design of fault-tolerant real-time systems for critical applications. Providing a simple criterion to ensure the feasibility of meeting all deadlines in the presence of a single fault considerably reduces the complexity encountered by designers. This will lead to safer and dependable use of real-time systems for critical applications.

References

[1] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *Journal of the Association for Computing Machinery*, vol. 20, no. 1, pp. 46–61, 1973.

[2] J. Lehoczky, L. Sha and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior", *Proceedings of the Real-Time Systems Symposium*, pp. 166–171, December 1989.

[3] S. K. Dhall and C. L. Liu, "On a Real-Time Scheduling Problem", *Operations Research*, vol. 26, no. 1, pp. 127–140, January-February 1978.

[4] E. L. Lawler and C. U. Martel, "Scheduling Periodically Occurring Tasks on Multiple Processors", *Information Processing Letters*, vol. 12, no. 1, pp. 9–12, 13 February 1981.

[5] J. Y.-T. Leung and M. L. Merril, "A Note on Preemptive Scheduling of Periodic, Real-Time Tasks", *Information Processing Letters*, vol. 11, no. 3, pp. 115–118, 18 November 1980.

[6] M. Malek, "Responsive Systems: A Challenge for the Nineties", Keynote Address, *EuroMicro 90, Microprocessing and Microprogramming*, vol. 30, August 1990.

[7] A. Liestman and R. H. Campbell, "A Fault-Tolerant Scheduling Problem", *IEEE Transactions on Software Engineering*, vol. SE-12, no. 11, pp. 1089–1095, November 1986.

[8] C. M. Krishna and K. G. Shin, "On Scheduling Tasks with a Quick Recovery from Failure", *IEEE Transactions on Computers*, vol. C-35, no. 5, pp. 448–455, May 1986.

[9] M. Takegaki, H. Kanamaru and M. Fujita, "The Diffusion Model Based Task Remapping for Distributed Real-Time Systems", *Proceedings of the Real-Time Systems Symposium*, pp. 2–11, December 1991.

[10] M. Barborak, M. Malek and A. Dahbura, "The Consensus Problem in Fault-Tolerant Computing", *ACM Computing Surveys*, vol. 25, no. 2, pp. 171–220, June 1993.

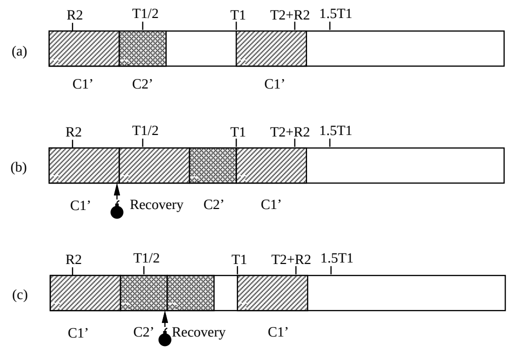


Figure 5: Two tasks considered in Lemma 3.

[11] D. P. Siewiorek and R. S. Swarz, The Theory and Practice of Reliable System Design, Digital Press, 1982.

A Appendix

A.1 Various cases of two tasks where $T_2 < 1.5T_1$

We consider various cases of two tasks satisfying the condition $T_2 < 1.5T_1$. For each case, we present the values of the computation times of the tasks that fully utilize the processor during the first instance of the Task 2 and show that the utilization factor for all these cases is greater than 0.5. Let $C_1 = T_1/2$ and $C_2 = 0$. We observe that the processor is fully utilized and the utilization factor $U = 0.5$. Now decrement the computation time of Task 1 by an amount Δ so that $C_1^{'} = C_1 - \Delta$. Increment the computation time of Task 2 to $C_2^{'}$ so that the processor is again fully utilized. This value of $C_2^{'}$ and the corresponding utilization factor $U^{'}$ are considered for various cases below. We use the notation t_c to denote the time instant of completion (departure) of the Task 2.

Case (i): $R_1 = 0, R_2 \geq 0$

We will first prove a series of lemmas each of which considers different possible values of the parameters of the tasks.

Lemma 3 *The utilization factor is greater than 0.5 for a task set that fully utilizes the processor for the first instance of the Task 2 if the task set is such that $T_2 < 1.5T_1$, $R_1 = 0, R_2 \geq 0$ and satisfies the following conditions*

$$T_1 + C_1^{'} \geq T_2 + R_2 \tag{9}$$

$$C_1^{'} \geq \frac{T_1}{3} \tag{10}$$

In this case, R_2 and $C_1^{'}$ are such that the second instance of the Task 1 completes after the deadline of Task 2 as shown in Figure 5(a), i.e., $T_1 + C_1^{'} \geq T_2 + R_2$ which also implies that $R_2 < C_1^{'}$. In this case

$$C_2^{'} = T_1 - 2C_1^{'} = 2\Delta$$

If a fault occurs just before the completion of Task 1 as shown in Figure 5(b),

$$t_c = 2C_1^{'} + C_2^{'} = T_1 < T_2 + R_2$$

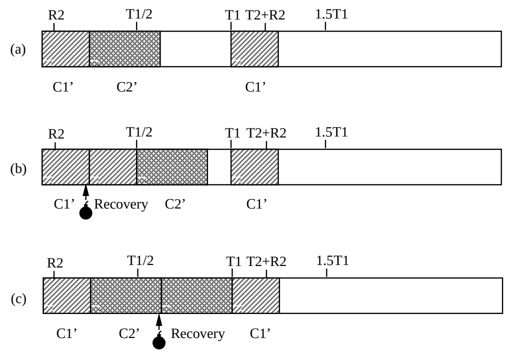


Figure 6: Two tasks considered in Lemma 4.

and so the Task 2 meets its deadline. If the fault affects Task 2 as shown in Figure 5(c),

$$t_c = C_1' + 2C_2' = 2T_1 - 3C_1' < T_1$$

The utilization factor is

$$\begin{aligned} U' &= \frac{C_1'}{T_1} + \frac{C_2'}{T_2} \\ &= \frac{C_1}{T_1} - \frac{\Delta}{T_1} + \frac{2\Delta}{T_2} \\ &= 0.5 + 2\Delta\left(\frac{1}{T_2} - \frac{1}{2T_1}\right) \\ &> 0.5 \end{aligned}$$

since $T_2 < 1.5T_1 < 2T_1$ and so $1/T_2 - 1/(2T_1)$ is a positive number. □

Lemma 4 *The utilization factor is greater than 0.5 for a task set that fully utilizes the processor for the first instance of Task 2 if the task set is such that* $T_2 < 1.5T_1$, $R_1 = 0, R_2 \geq 0$ *and satisfies the following conditions*

$$T_1 + C_1' \geq T_2 + R_2 \tag{11}$$

$$C_1' < \frac{T_1}{3} \tag{12}$$

This is illustrated by Figure 6. If $C_1' < T_1/3$, then C_2' cannot be 2Δ because even though the schedule can tolerate a fault just prior to completion of Task 1, it cannot tolerate a fault if it occurs just prior to completion of Task 2. In this case,

$$C_2' = \frac{T_1 - C_1'}{2}$$

and the largest value of $\Delta = 1.5T_1 - T_2 - R_2$ (otherwise we will violate the condition of Equation 11 that $T_1 + C_1' \geq T_2 + R_2$). If a fault affects Task 1 as shown in Figure 6(b),

$$t_c = 2C_1' + C_2' = T_1 + \frac{T_1}{4} - \frac{3\Delta}{2} \leq T_1$$

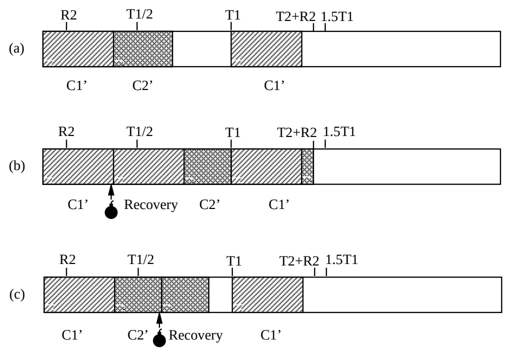


Figure 7: Two tasks considered in Lemma 5.

If the fault affects Task 2 as shown in Figure 6(c),

$$t_c = C_1^{'} + 2C_2^{'} = T_1$$

The utilization factor

$$\begin{aligned} U^{'} &= \frac{C_1^{'}}{T_1} + \frac{C_2^{'}}{T_2} \\ &= \frac{C_1 - \Delta}{T_1} + \frac{T_1 - C_1 + \Delta}{2T_2} \\ &= 0.5 + \Delta\left(\frac{1}{2T_2} - \frac{1}{T_1}\right) + \frac{T_1}{4T_2} \end{aligned}$$

remembering that $C_1 = T_1/2$. Since $1/2T_2 - 1/T_1$ is a negative number, the minimum of right hand side occurs when Δ is maximum and in that case,

$$\begin{aligned} U^{'} &= 0.5 + (1.5T_1 - T_2 - R_2)\left(\frac{1}{2T_2} - \frac{1}{T_1}\right) + \frac{T_1}{4T_2} \\ &= 0.5 + \frac{T_1}{T_2} + \frac{T_2}{T_1} - 2 + R_2\left(\frac{1}{T_1} - \frac{1}{2T_2}\right) \\ &= 0.5 + \frac{(T_2 - T_1)^2}{T_2 T_1} + R_2\left(\frac{1}{T_1} - \frac{1}{2T_2}\right) \\ &> 0.5 \end{aligned}$$

since both $(T_2 - T_1)^2/(T_2T_1)$ and $R_2(1/T_1 - 1/2T_2)$ are non-negative numbers. □

Lemma 5 *The utilization factor is greater than 0.5 for a task set that fully utilizes the processor for the first instance of Task 2 if the task set is such that* $T_2 < 1.5T_1$, $R_1 = 0, R_2 \geq 0$ *and satisfies the following conditions*

$$T_1 + C_1^{'} < T_2 + R_2 \tag{13}$$

$$R_2 < C_1^{'} \tag{14}$$

$$C_1^{'} \geq \frac{T_2 + R_2}{4} \tag{15}$$

Here, the value of $C_2^{'}$ that fully utilizes the processor is given by

$$C_2^{'} = R_2 + T_2 - 3C_1^{'}$$

Figure 7 illustrates this subcase. Let us see what happens when the fault affects Task 1 just prior to its completion as shown in Figure 7(b). We note that $2C_1^{'} + C_2^{'} = R_2 + T_2 - C_1^{'} < T_1$. Hence, the recovery action of Task 1 delays the Task 2 so that it does not complete before the second arrival of Task 1, which then preempts it. Thus the time to completion of the second task includes three executions of the first task and itself and so

$$t_c = 3C_1^{'} + C_2^{'} = R_2 + T_2,$$

i.e., the second task just meets its deadline at time $R_2 + T_2$. If a fault affects Task 2, there are two possibilities: the recovery might complete before the second arrival of Task 1 or it might not, in which case it will be preempted. In the first case, Task 2 obviously meets its deadline. In the latter case,

$$t_c = 2C_1^{'} + 2C_2^{'} = 2R_2 + 2T_2 - 4C_1^{'} \leq T_2 + R_2$$

since $C_1^{'} \geq (T_2 + R_2)/4$.

The utilization factor is given by

$$\begin{aligned}
U^{'} &= \frac{C_1^{'}}{T_1} + \frac{C_2^{'}}{T_2} \\
&= \frac{C_1 - \Delta}{T_1} + \frac{R_2 + T_2 - 3C_1^{'}}{T_2} \\
&= \frac{C_1}{T_1} - \frac{\Delta}{T_1} + \frac{R_2 + T_2 - C_1^{'} - 2C_1^{'}}{T_2} \\
&= \frac{C_1}{T_1} - \frac{\Delta}{T_1} + \frac{(R_2 + T_2 - C_1^{'}) - 2(C_1 - \Delta)}{T_2} \\
&= \frac{C_1}{T_1} + \Delta\left(\frac{2}{T_2} - \frac{1}{T_1}\right) + \frac{R_2 + T_2 - C_1^{'} - T_1}{T_2} \\
&> 0.5
\end{aligned}$$

since $(2/T_2 - 1/T_1) > 0$ and from Equation 13, $R_2 + T_2 > T_1 + C_1^{'}$. □

Lemma 6 *The utilization factor is greater than 0.5 for a task set that fully utilizes the processor for the first instance of Task 2 if the task set is such that* $T_2 < 1.5T_1$, $R_1 = 0, R_2 \geq 0$ *and satisfies the following conditions*

$$T_1 + C_1^{'} < T_2 + R_2 \tag{16}$$

$$R_2 < C_1^{'} \tag{17}$$

$$C_1^{'} < \frac{T_2 + R_2}{4} \tag{18}$$

In this situation,

$$C_2^{'} = \frac{T_2 + R_2 - 2C_1^{'}}{2}$$

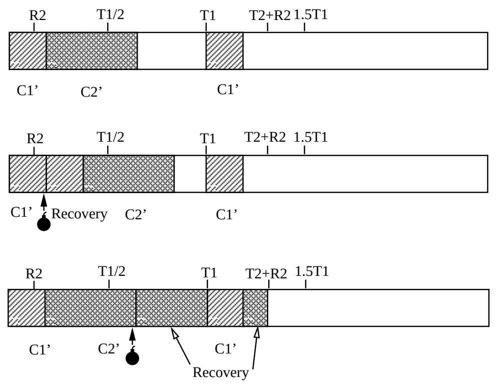


Figure 8: Two tasks considered in Lemma 6.

The situation is shown in Figure 8. We again verify that Task 2 will meet its deadline when a fault occurs. If a fault affects Task 1, the worst case situation is as before, i.e., the task 2 is delayed and might be preempted by the second arrival of Task 1. Thus,

$$t_c = 3C_1^{'} + C_2^{'} = 2C_1^{'} + \frac{R_2 + T_2}{2} < R_2 + T_2$$

because $C_1^{'} < (T_2 + R_2)/4$ (however, in the Figure 8(b), the Task 2 completes before the second arrival of the Task 1). If a fault occurs just before Task 2 was to complete as shown in Figure 8(c), then its completion time is given by

$$t_c = 2C_1^{'} + 2C_2^{'} = T_2 + R_2$$

The utilization factor is given by

$$\begin{aligned} U^{'} &= \frac{C_1^{'}}{T_1} + \frac{C_2^{'}}{T_2} \\ &= \frac{C_1^{'}}{T_1} + \frac{T_2 + R_2 - 2C_1^{'}}{2T_2} \\ &= 0.5 + C_1^{'}\left(\frac{1}{T_1} - \frac{1}{T_2}\right) + \frac{R_2}{2T_2} \\ &\geq 0.5 \end{aligned}$$

□

Note that Lemmas 3– 6 also cover all cases when the two tasks are released simultaneously, i.e., $R_1 = R_2 = 0$.

Lemma 7 *The utilization factor is greater than 0.5 for a task set that fully utilizes the processor for the first instance of Task 2 if the task set is such that* $T_2 < 1.5T_1$, $R_1 = 0, R_2 \geq 0$ *and satisfies the following conditions*

$$T_1 + C_1^{'} < T_2 + R_2 \leq T_1 + 2C_1^{'} \tag{19}$$

$$C_1^{'} \leq R_2 < 2C_1^{'} \tag{20}$$

$$C_1^{'} > \frac{T_2 + 2R_2}{5} \tag{21}$$

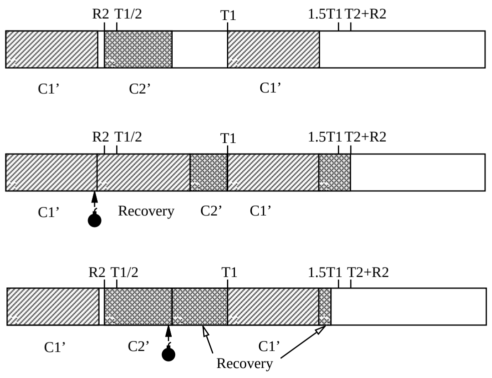


Figure 9: Two tasks considered in Lemma 7.

Note that $R_2 \geq C_1^{'}$ implies $T_1 + C_1^{'} < T_2 + R_2$ because $T_2 > T_1$. This is the situation where the first arrival of Task 2 occurs after Task 1 has completed execution as shown is Figure 9(a). However, if a fault occurs just prior to completion of Task 1, it will affect Task 2 because the recovery action will delay it. Also, since $T_1 + 2C_1^{'} \geq T_2 + R_2$, a fault occurrence just prior to completion of the second instance of Task 1 pushes the recovery action beyond the deadline of Task 1. Hence, Task 2 should complete prior to T_1 in the absence of a fault. Under the conditions of this lemma, the processor is fully utilized if

$$C_2^{'} = min(T_1 - R_2, T_2 + R_2 - 3C_1^{'})$$

Here $min(a, b)$ is a function that returns the minimum of a and b (Note: $min(a, b) \leq a$ and $min(a, b) \leq b$). Let us again check if all deadlines are met in the presence of a fault. If the fault affects the first arrival of Task 1 as shown in Figure 9(b),

$$\begin{aligned} t_c &= 3C_1^{'} + C_2^{'} \\ &= 3C_1^{'} + min(T_1 - R_2, R_2 + T_2 - 3C_1^{'}) \\ &\leq 3C_1^{'} + R_2 + T_2 - 3C_1^{'} \\ &= T_2 + R_2 \end{aligned}$$

If the fault occurs just prior to the completion of the Task 2 as shown in Figure 9(c), then

$$\begin{aligned} t_c &= R_2 + 2C_2^{'} + C_1^{'} \\ &= R_2 + C_1^{'} + 2min(T_1 - R_2, R_2 + T_2 - 3C_1^{'}) \\ &\leq R_2 + C_1^{'} + 2(R_2 + T_2 - 3C_1^{'}) \\ &= T_2 + R_2 + T_2 + 2R_2 - 5C_1^{'} \\ &< T_2 + R_2 \qquad \text{From Equation 21} \end{aligned}$$

A fault during the execution of the second instance of Task 1 does not affect the execution of the first instance of Task 2 since it has already completed execution. The utilization factor

$$U^{'} = \frac{C_1^{'}}{T_1} + \frac{C_2^{'}}{T_2}$$

If $T_1 - R_2 \leq T_2 + R_2 - 3C_1^{'}$ then $C_2^{'} = T_1 - R_2$ and so

$$U^{'} = \frac{C_1^{'}}{T_1} + \frac{T_1 - R_2}{T_2}$$

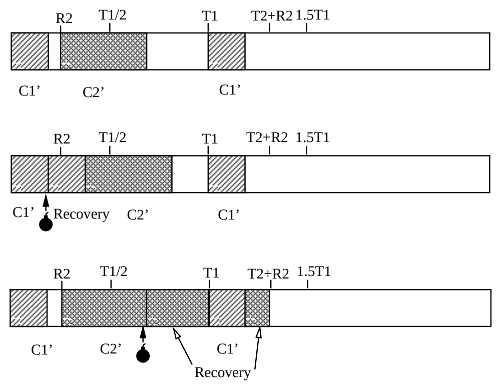


Figure 10: Two tasks considered in Lemma 8.

$$
\begin{aligned}
&> \frac{C_1'}{T_1} + \frac{T_2 - 2C_1'}{T_2} && \text{From Equation 20} \\
&> \frac{C_1'}{T_2} + \frac{T_2 - 2C_1'}{T_2} \\
&= 1 - \frac{C_1'}{T_2} \\
&> 0.5
\end{aligned}
$$

because $C_1'/T_2 < C_1'/T_1 < 0.5$. On the other hand, if $C_2' = T_2 + R_2 - 3C_1'$, then the utilization factor is given by

$$
\begin{aligned}
U' &= \frac{C_1'}{T_1} + \frac{T_2 + R_2 - 3C_1'}{T_2} \\
&= 1 + \frac{C_1'}{T_1} + \frac{R_2}{T_2} - \frac{3C_1'}{T_2} \\
&\geq 1 + \frac{C_1'}{T_1} + \frac{C_1'}{T_2} - \frac{3C_1'}{T_2} && \text{From Equation 20} \\
&> 1 - \frac{C_1'}{T_1} \\
&> 0.5
\end{aligned}
$$

$\square$

Lemma 8 *The utilization factor is greater than 0.5 for a task set that fully utilizes the processor for the first instance of Task 2 if the task set is such that* $T_2 < 1.5T_1$, $R_1 = 0, R_2 \geq 0$ *and satisfies the following conditions*

$$T_1 + C_1' < T_2 + R_2 \leq T_1 + 2C_1' \tag{22}$$

$$C_1' \leq R_2 < 2C_1' \tag{23}$$

$$C_1' \leq \frac{T_2 + 2R_2}{5} \tag{24}$$

Figure 10 illustrates two tasks that satisfy the conditions of Lemma 8. Under the conditions above, the processor is fully utilized if

$$C_2^{'} = min\left(T_1 - R_2, \frac{T_2 - C_1^{'}}{2}\right)$$

If the fault occurs just prior to completion of Task 1, then

$$\begin{aligned}
t_c &= 3C_1^{'} + C_2^{'} \\
&= 3C_1^{'} + min\left(T_1 - R_2, \frac{T_2 - C_1^{'}}{2}\right) \\
&\leq 3C_1^{'} + \frac{T_2 - C_1^{'}}{2} \\
&= \frac{T_2 + 5C_1^{'}}{2} \\
&\leq T_2 + R_2 \qquad \text{From Equation 24}
\end{aligned}$$

If a fault occurs just prior to completion of the Task 2, then

$$\begin{aligned}
t_c &= R_2 + 2C_2^{'} + C_1^{'} \\
&= R_2 + C_1^{'} + 2min\left(T_1 - R_2, \frac{T_2 - C_1^{'}}{2}\right) \\
&\leq R_2 + C_1^{'} + T_2 - C_1^{'} \\
&= R_2 + T_2
\end{aligned}$$

The utilization factor

$$U^{'} = \frac{C_1^{'}}{T_1} + \frac{C_2^{'}}{T_2}$$

If $T_1 - R_2 \leq (T_2 - C_1^{'})/2$ then $C_2^{'} = T_1 - R_2$ and so

$$\begin{aligned}
U^{'} &= \frac{C_1^{'}}{T_1} + \frac{T_1 - R_2}{T_2} \\
&> \frac{C_1^{'}}{T_1} + \frac{T_2 - 2C_1^{'}}{T_2} \qquad \text{From Equation 23} \\
&> \frac{C_1^{'}}{T_2} + \frac{T_2 - 2C_1^{'}}{T_2} \\
&= 1 - \frac{C_1^{'}}{T_2} \\
&> 0.5
\end{aligned}$$

because $C_1^{'}/T_2 < C_1^{'}/T_1 < 0.5$. If $T_1 - R_2 > (T_2 - C_1^{'})/2$ then $C_2^{'} = (T_2 - C_1^{'})/2$ and

$$\begin{aligned}
U^{'} &= \frac{C_1^{'}}{T_1} + \frac{T_2 - C_1^{'}}{2T_2} \\
&= 0.5 + \frac{C_1^{'}}{T_1} - \frac{C_1^{'}}{T_2} \\
&> 0.5
\end{aligned}$$

□

Lemma 9 *The utilization factor is greater than 0.5 for a task set that fully utilizes the processor for the first instance of Task 2 if the task set is such that* $T_2 < 1.5T_1$, $R_1 = 0, R_2 \geq 0$ *and satisfies the following conditions*

$$T_1 + 2C_1^{'} < T_2 + R_2 \leq 2T_1 \tag{25}$$

$$C_1^{'} \leq R_2 < 2C_1^{'} \tag{26}$$

In this case the processor is fully utilized if

$$C_2^{'} = min\left(T_1 - R_2, \frac{T_2 - C_1^{'}}{2}\right)$$

Note that the Task 2 completes before the second arrival of Task 1 and so a fault occurrence during the execution of the second instance of Task 1 will be after its completion. If a fault occurs just prior to completion of the first instance of Task 1, then the completion time of Task 2 is

$$\begin{aligned}
t_c &= 3C_1{}' + C_2^{'} \\
&= 3C_1{}' + min\left(T_1 - R_2, \frac{T_2 - C_1^{'}}{2}\right) \\
&\leq 3C_1{}' + T_1 - R_2 \\
&\leq T_1 + 2C_1{}' + C_1{}' - R_2 \\
&< T_2 + R_2 + C_1{}' - R_2 \qquad \text{From Equation 25} \\
&< T_2 + C_1{}' \\
&< T_2 + R_2 \qquad \text{From Equation 26}
\end{aligned}$$

If the fault occurs just prior to completion of Task 2, then,

$$\begin{aligned}
t_c &= R_2 + 2C_2^{'} + C_1{}' \\
&= R_2 + 2min\left(T_1 - R_2, \frac{T_2 - C_1^{'}}{2}\right) + C_1^{'} \\
&\leq R_2 + T_2 - C_1^{'} + C_1^{'} \\
&\leq T_2 + R_2
\end{aligned}$$

When $C_2^{'} = T_1 - R_2$, the utilization factor is

$$\begin{aligned}
U^{'} &= \frac{C_1^{'}}{T_1} + \frac{T_1 - R_2}{T_2} \\
&> \frac{C_1^{'}}{T_1} + \frac{T_2 - 2C_1^{'}}{T_1} \qquad \text{From Equation 26} \\
&> \frac{C_1^{'}}{T_2} + \frac{T_2 - 2C_1^{'}}{T_1} \\
&> 1 - \frac{C_1^{'}}{T_2} \\
&> 0.5
\end{aligned}$$

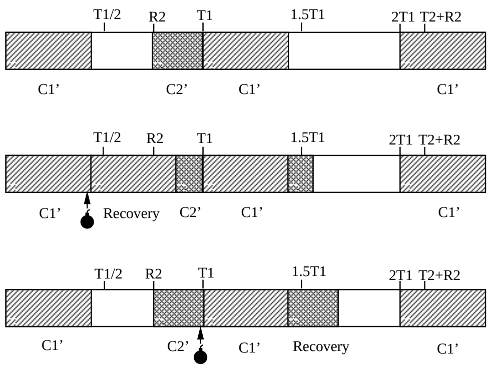


Figure 11: Two tasks considered in Lemma 10.

If $C'_2 = (T_2 - C'_1)/2$, then

$$\begin{aligned} U' &= \frac{C'_1}{T_1} + \frac{1}{2} - \frac{C'_1}{2T_2} \\ &> 0.5 \end{aligned}$$

□

Lemma 10 *The utilization factor is greater than 0.5 for a task set that fully utilizes the processor for the first instance of Task 2 if the task set is such that* $T_2 < 1.5T_1$, $R_1 = 0, R_2 \geq 0$ *and satisfies the following conditions*

$$2T_1 < T_2 + R_2 \tag{27}$$

$$C'_1 \leq R_2 < 2C'_1 \tag{28}$$

Here, $R_2 + T_2$ is greater than $2T_1$, i.e., the deadline of Task 2 is beyond the arrival of the third instance of Task 1 as shown in Figure 11. It is easy to observe that

$$R_2 + T_2 < 2T_1 + C'_1 \tag{29}$$

This means that even when the deadline is beyond $2T_1$, it is effectively prior to the completion time of the third instance of Task 1. Hence, under all conditions when $R_2 + T_2 > 2T_1$, Task 2 should complete before time $2T_1$ and that becomes its effective deadline. The value of C'_2 that fully utilizes the processor is

$$C'_2 = T_1 - R_2$$

If the fault occurs just prior to the completion of the first instance of Task 1, then

$$\begin{aligned} t_c &= 3C'_1 + C'_2 \\ &= 3C'_1 + T_1 - R_2 \\ &\leq 3C'_1 + T_1 - C'_1 \qquad \text{From Equation 28} \\ &= T_1 + 2C'_1 \\ &< T_1 + T_1 \\ &= 2T_1 \end{aligned}$$

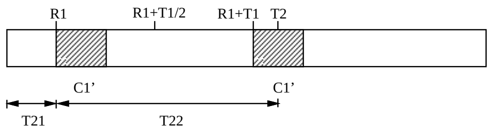


Figure 12: Two tasks considered in Lemma 12.

and if the fault occurs just prior to the completion of Task 2, then $t_c = R_2 + 2C_2^{'} + C_1^{'} = R_2 + 2T_1 - 2R_2 + C_1^{'} \leq 2T_1$. The utilization factor

$$\begin{aligned} U^{'} &= \frac{C_1^{'}}{T_1} + \frac{T_1 - R_2}{T_2} \\ &> \frac{C_1^{'}}{T_1} + \frac{T_1 - 2C_1^{'}}{T_2} \qquad \text{From Equation 28} \\ &> 0.5 \end{aligned}$$

□

Lemma 11 *The utilization factor is greater than 0.5 for a task set that fully utilizes the processor for the first instance of Task 2 if the task set is such that $T_2 < 1.5T_1$, $R_1 = 0, R_2 \geq 0$ and satisfies the following conditions*

$$2C_1^{'} < R_2 \tag{30}$$

The first instance of the Task 2 can be modeled by the first instance of the Task 2 in the task set $\{(C_1^{'}, T_1, R_1 = T_1 - R_2), (C_2^{'}, T_2, R_2 = 0)\}$. As we shall prove in Lemmas 12–14, the above task set that fully utilizes the processor during the first instance of the Task 2 has a utilization factor greater than 0.5. □

Case (ii): $R_1 > 0, R_2 = 0$

Lemma 12 *The utilization factor is greater than 0.5 for a task set that fully utilizes the processor for the first instance of Task 2 if the task set is such that $T_2 < 1.5T_1$, $R_1 > 0, R_2 = 0$ and satisfies the following conditions*

$$R_1 + T_1 \leq T_2 \tag{31}$$

Two tasks that satisfy the conditions of this lemma are illustrated in Figure 12. Consider a split of task 2 such that $T_2 = T_{21} + T_{22}$ such that $T_{22} = T_2 - R_1$ and $T_{21} = R_1$. Consider a task set $S^{'} = \{(C_1^{'}, T_1, 0), (C_{22}, T_{22}, 0)\}$ that fully utilizes the processor. Since we have already proved in Lemmas 3– 6 that when two tasks are released simultaneously, both tasks can meet their first deadlines in the presence of a fault as long as their utilization factor is less than 0.5, i.e., the minimum achievable utilization is 0.5, then

$$\frac{C_{22}}{T_{22}} + \frac{C_1^{'}}{T_1} \geq 0.5$$

and so

$$C_{22} \geq \left(0.5 - \frac{C_1'}{T_1}\right)(T_2 - R_1)$$

Thus the value of C_2' that fully utilizes the processor is such that

$$C_2' \geq \frac{R_1}{2} + C_{22}$$

The utilization factor is given by

$$\begin{aligned}
U' &= \frac{C_1'}{T_1} + \frac{C_2'}{T_2} \\
&\geq \frac{C_1'}{T_1} + \frac{R_1}{2T_2} + \frac{C_{22}}{T_2} \\
&\geq \frac{C_1'}{T_1} + \frac{R_1}{2T_2} + \left(0.5 - \frac{C_1'}{T_1}\right)\frac{(T_2 - R_1)}{T_2} \\
&\geq \frac{C_1'}{T_1} + \frac{R_1}{2T_2} + 0.5 - \frac{R_1}{2T_2} - \frac{C_1'}{T_1} + \frac{C_1'R_1}{T_1T_2} \\
&\geq 0.5 + \frac{C_1'R_1}{T_1T_2} \\
&\geq 0.5
\end{aligned}$$

□

Lemma 13 *The utilization factor is greater than 0.5 for a task set that fully utilizes the processor for the first instance of Task 2 if the task set is such that $T_2 < 1.5T_1$, $R_1 > 0, R_2 = 0$ and satisfies the following conditions*

$$R_1 + T_1 > T_2 \tag{32}$$

$$R_1 + 2C_1' < T_2 \tag{33}$$

We notice that when the processor is fully utilized,

$$C_2' \geq \frac{T_2 - 2C_1'}{2}$$

The worst case scenario occurs when both the tasks have to be executed twice in the presence of a single fault and so $C_2' = (T_2 - 2C_1')/2$. In this case the completion time of task 2 is

$$t_c = 2C_1' + 2C_2' = T_2$$

In all other cases, $C_2' > (T_2 - 2C_1')/2$. Thus the utilization factor is

$$\begin{aligned}
U' &= \frac{C_1'}{T_1} + \frac{C_2'}{T_2} \\
&\geq \frac{C_1'}{T_1} + \frac{T_2 - 2C_1'}{2T_2} \\
&\geq 0.5 + \frac{C_1'}{T_1} - \frac{C_1'}{T_2} \\
&\geq 0.5
\end{aligned}$$

□

Lemma 14 *The utilization factor is greater than 0.5 for a task set that fully utilizes the processor for the first instance of Task 2 if the task set is such that $T_2 < 1.5T_1$, $R_1 > 0, R_2 = 0$ and satisfies the following conditions*

$$R_1 + T_1 > T_2 \tag{34}$$

$$T_2 \leq R_1 + 2C_1^{'} \tag{35}$$

Under the condition of this lemma given by Equation 35, if a fault occurs during the execution of task 1, the completion of recovery will occur after the deadline of the task 2 has passed. So, in the absence of faults, the task 2 should complete execution before task 1 arrives at time instant R_1. Also if $T_2 \leq R_2 + C_1^{'}$, then the task 2 should be completed before R_1 even in the presence of a fault occurrence during task 2. Thus the value of the computation time of task 2 that fully utilizes the processor is

$$C_2^{'} \geq \frac{R_1}{2}$$

The utilization factor

$$\begin{aligned}
U^{'} &= \frac{C_1^{'}}{T_1} + \frac{C_2^{'}}{T_2} \\
&\geq \frac{C_1^{'}}{T_1} + \frac{R_1}{2T_2} \\
&\geq \frac{C_1^{'}}{T_1} + \frac{T_2 - 2C_1^{'}}{2T_2} \qquad \text{From Equation 35} \\
&\geq 0.5 + \frac{C_1^{'}}{T_1} - \frac{C_1^{'}}{T_2} \\
&\geq 0.5
\end{aligned}$$

□