CHALMERS

Real-Time Scheduling Analysis of System Tolerating Multiple Transient Faults

RISAT MAHMUD PATHAN

Master's Thesis

International Master's Program in Dependable Computer Systems

CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Division of Computer Engineering
Göteborg 2005

Real-Time Scheduling Analysis of System Tolerating Multiple Transient Faults

Risat Mahmud Pathan
Department of Computer Engineering
Chalmers University of Technology
421 96 Gothenburg, Sweden
risatmp@yahoo.com

ABSTRACT

The influence of computer systems in human life is increasing and thereby increases the need for having reliable, robust and real-time services of computer systems. Avoidance of any catastrophic consequences due to faults in such systems is the main objective now-a-days. In this paper, a probabilistic measure of schedulability of real-time systems tasks is addressed in the presence of multiple transient faults. The main approach is employing Temporal Error Masking (TEM) technique to achieve Node Level Fault Tolerance (NLFT) within the least common multiple of periods of a set of pre-emptive period tasks with at most f transient faults. The Rate Monotonic (RM) scheduling is used to observe how the probability of system success denoted by $P_{success}$, that is the probability of meeting deadlines for all tasks, is affected with worst-case fault distribution. In addition, a recovery algorithm is proposed along with a probabilistic estimate for such recovery when transient faults are early detected by hardware or software Error Detection Mechanism (EDM).

Keywords: *Real-Time Systems, Fault Tolerance, Task Schedulability, Probabilistic Guarantee, Fixed-Priority Scheduling, NLFT, TEM.*

Acknowledgement

I would especially like to thank my supervisor, Jan Jonsson, Associate Professor in Computer Science and Engineering Department, Chalmers University of Technology, for his continuous support, invaluable ideas, excellent comments, feedback, and most importantly his encouragement to carry out this work. His knowledge, guidance and cooperation have provided me the basis and inspiration to work in such an interesting topic on Real-Time Fault-Tolerant computer systems.

Special thanks and gratitude goes to Johan Karlsson, Professor in Computer Science and Engineering Department, Chalmers University of Technology, for his excellent ideas regarding the fault-tolerant areas covered in my thesis work. I would also like to thank Joakim Aidemark, Ph. D., Chalmers University of Technology, for his work in the area of fault tolerant computer systems, which have provided me excellent ideas in fault-tolerant area and necessary experimental data I have used in my thesis work.

I would also like to thank my friends here in Sweden for their support during my thesis work and for the happy moments I have had with them.

Last but not the least, I want to express my deepest gratitude and thanks to my parents, for their continuous support, both financially and mentally, during the years of my study in Sweden.

TABLE OF CONTENTS

1. Introduction:

"Non-faulty systems hardly exists, there are only systems which may have not yet failed."

–J. C. Laprie [1]

Real time systems are being increasingly used in several applications which are time critical in nature. This includes systems where a computer failure can pose a threat to human lives or the environment, and systems where a failure may cause significant economic loss. Recent advantages in technologies have brought much attention to migrating previously mechanical or manual system to embedded computing systems. Among these, systems with timing constraints have been widely used in several time critical applications ranging from fly-by-wire, brake-by wire, autopilot system and space shuttles, to industrial process control, robots and smart automobiles. The tasks in such application are *hard real-time* tasks, which have stringent timing requirement. The consequences of missing the deadline of a hard real time task may be catastrophic. Moreover, the ability to tolerate faults in hard real-time systems is crucial, since a task can potentially miss its deadline when faults occur. In case of a fault, a deadline can be missed if the time taken for recovery from faults is not taken into account during the phase when tasks are submitted or accepted to the system. Hence, fault tolerance is an essential requirement for such systems, due to the catastrophic consequences of not tolerating faults.

Fault tolerant computers have been used in satellites, launchers and other space vehicles since the beginning of the space era to protect the lives of the crews in manned space missions and economic investments in unmanned missions [2]. One of the first widespread uses of fault tolerant computers where public safety was at stake was in commercial airplanes in the early 1990's. At that time systems allowing automatic landing in all visibility conditions were introduced. The system must be extremely dependable, as a failure could cause the plane to crash. The use of fault tolerant computers in avionics was further advanced in 1980s and the early 1990s, when so called fly-by-wire systems were introduced in military aircrafts and commercial airlines, such as the Airbus-320 [3] and Boeing 777 [4]. In fly by wire systems, the mechanical links between the pilot controls and the plane's control surfaces are replaced with computer networks and electronically controlled actuators. These systems must meet very stringent dependability conditions, since the pilot is unable to control the aircraft if the fly-by-wire system fails.

The importance of dependability in embedded system will increase dramatically as future computers take a more active role in everyday control applications such as drive-by-wire and brake-by-wire systems in vehicles. By-wire systems are also currently being considered for safety-critical control systems in cars and other road vehicles. The introduction of brake-by-wire systems and steer-by wire systems will allow future vehicles to be equipped with intelligent safety systems that can help the driver to brake or steer the vehicle to avoid accidents or to mitigate the impact of collisions.

Real-time systems with high dependability requirements are traditionally built with massive replication and redundancy. The main objective is to maintain the properties of correctness and timeliness even in the event of faults. In certain classes of applications, due to space, weight and cost considerations it may not be feasible to provide space redundancy. Such systems need to exploit time redundancy techniques. Due to real-time nature of such systems, it is essential that the exploitation of time redundancy for correctness does not jeopardize the timeliness guarantee. There is a need for the feasibility analysis which provides guarantees for fault-tolerant real-time task sets under the assumption for a defined failure hypothesis. Thus, safety critical systems have to guarantee functional and timing constraints even in the presence of hardware and software failures. One way of guaranteeing that all timing and resource constraints are met is to statically schedule all tasks to meet their deadlines.

2. Basic Concepts: Fault Tolerant Systems

In [1] the term 'dependability' is defined as, "that the property of computer system such that reliance can justifiably be placed on the service it delivers."

2.1 Failure, Error, and Faults:

A system *failure* occurs when the service provided by the system deviates from the specified service. An *error* is a perturbation of internal state of the system that may lead to failure, that is, a failure occurs when the erroneous state causes an incorrect service to be delivered. The cause of the error is called a *fault*. An *active* fault leads to an error; otherwise the fault is dormant.

Task deadlines in hard real-time system must be met even in presence of faults due to their critical nature. Faults to be tolerated can be of three different types: permanent, intermittent, and transient [5, 6]. Permanent faults are caused by total failure of computing unit, and are typically tolerated by hardware redundancy such as spare processors. Transient faults are temporary malfunctioning of the computing unit or any other associated components which causes incorrect results to be computed. Intermittent faults are repeated occurrences of transient faults.

2.1.1 Transient Faults:

Although the ongoing reduction of device geometry and supply voltage increases the risk for not only transient faults but also for permanent faults, our focus in this paper is on transient faults due to their higher frequency of occurrence. The main source of transient faults is environmental disturbances. Environmental disturbances include power fluctuations, electromagnetic interference and ionizing radiation by alpha particles and high energy neutrons [5, 7]. For example, neutron radiation has primarily been a concern in aerospace applications, as high energy neutrons are much more common at higher altitudes than at ground level.

Several studies have shown that transient faults occur at much a higher rate than permanent faults [7]. In [8], measurements showed that transient faults are 30 times more frequent than permanent faults, while in the work in [9] revealed that 83% of all faults were determined to be transient or intermittent. In [7], tables were provided to show that rates of transient faults are about 20 times that of permanent faults. In some real time systems such as satellites and space shuttles, transient faults occur at a much higher rate than in general purpose systems [10]. An orbiting satellite containing a microelectronics test system has been used to measure 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 a 15-minute intervals, and was measured to be as high as 35 in such intervals. Because of high occurrence of transient faults and because permanent faults can be tolerated using physical redundancy, we focus on transient faults in this thesis.

2.1.2 Failure Modes:

A system failure mode describes the ways in which a system can fail. One way of categorizing failure is by their domain [5]. The failure domains can be divided into the *value* domain and the *timing* domain. The value domain may include value failures and fail bounded failures. A *value failure* occurs when the produced output value deviates from an expected value. A *fail bounded failure* occurs when the system produces an incorrect output value, but the output is within a tolerable bound. In the time domain, a *timing failure* occurs when the time of the output value is produced too early or too late. An *omission failure*

occurs if the system does not respond to an input and other part of the system is not aware of the failure. A *crash failure* occurs if the system permanently stops operating and fails to respond to any input. A *fail-stop failure* is a crash failure that is made known to rest of the system.

2.2 Fault Tolerance:

Fault tolerance in distributed systems can be achieved through a hierarchy of system level, node level and circuit level mechanisms. Error handling at the circuit level refers to a mechanism provided by hardware components such as microprocessors and I/O circuits, while error handling at the node level refers to when additional hardware components and/or software components are used. Error handling activities involving more than one computer node in the distributed systems are performed at the system level. Our focus in this thesis is to achieve node level fault tolerance. Fault tolerance relates to providing additional mechanisms that allow faults to be detected and handled in an appropriate way. A fault tolerant system tolerates faults while still maintaining full system operation. In this thesis, we focus on achieving cost effective fault tolerance for handling faults at the computer nodes.

2.2.1 Node Level Error Detection Techniques:

An active fault leads to an error. By providing effective error handling at the node level and circuit level, it is possible to restrict the failure modes a node can exhibit, which allows for simpler protocols and fewer redundant nodes at system level. Error detection at node level can be implemented in hardware or software. Hardware implemented error detection can be achieved by executing the same functions on two processors and compare their outputs for discrepancies. A less costly hardware approach is to use a watchdog processor. A watchdog processor is a simple co-processor that can monitor the control flow or conduct reasonable checks on the output of the main processor. Software implemented error detection may be achieved using redundancy based checks, executable assertions, structural checks, timing checks and control flow checks [5].

Redundancy based checks include for example time redundancy. In *time redundancy*, an instruction, a function or a task is executed twice and the results are compared to allow errors to be detected. In *information redundancy*, errors are detected by duplicating and comparing the contents of variables. Redundancy based techniques can be implemented systematically, which reduces the complexity for application designers. For example, time redundancy may be used without the knowledge of the application, and duplicating variables can be automated by a transformation tool [11].

Executable assertions are based on detailed information about the specification of the application. These checks are implemented using a small software routine that checks the reasonableness of variables. *Structural checks* allow the integrity of data structures such as lists and queues to be checked by including, for example, status information and redundant variables/pointers. *Timing checks* are used to detect erroneous programs execution by checking the time a program runs. Such checks may be provided by the processor in the form of a watchdog timer. *Control flow checks* detects if an incorrect sequence of instructions is executed.

Many modern processors provide on-chip error detection mechanisms such as error detection and correction in memories, caches and registers, illegal op-code detection, address range checking. However the effects of certain faults occurring, for example, in the arithmetic units of a microprocessor, such as adders and multipliers, may pass undetected which justifies the use of software implemented error detection techniques such as executable assertions or time redundant execution [6, 12, 13].

2.2.2 Node Level Fault Tolerance Techniques:

Fault tolerance within the node may be realized in hardware using techniques such as static or dynamic redundancy same as fault tolerance at the system level. Due to associated cost with using hardware, software based approach is more cost effective. Software implemented fault tolerance may be implemented using error-masking or error-recovery mechanisms. Error masking can be achieved in software by triplicating variables or executing a module three times, and then taking a majority vote on the outputs. Triplicated time redundant execution is evaluated in [14] where each software module and voting mechanism executed three times to mask errors. Error recovery is achieved by restoring the system to a fault-free state from which the system can continue whenever an error is detected. In this thesis, time redundant execution is used to mask errors at node level, known as Temporal Error Masking (TEM).

3. Basic Concepts: Real-Time Systems

As mentioned in [15], "Any system where a timely response by the computer to external stimuli is vital is a real-time system. This includes embedded systems that control things like aircraft, nuclear reactors, chemical power plants, jet engines, and other objects where Something Very Bad will happen if the computer does not deliver its output in time. These are called hard real time systems. There is another category called soft real-time systems, which are systems such as multimedia, where nothing catastrophic happens if some deadlines are missed, but where the performance will be degraded below what is generally considered acceptable. In general, a real time system is one in which a substantial fraction of the design effort goes into making sure that task deadlines are met."

Thus, a real time computer must be much more reliable than its individual hardware and software components. It must be capable of working in harsh environments, rich of electromagnetic noise and elementary particle radiation, and in the presence of rapidly changing computation loads. The field of real-time computing is especially rich of research problems because all problems in computer architecture, fault tolerant computing, and operating systems are also problems in real-time computing, with the added complexity that real-time constraints must be met. It is important that the task execution time is predictable to allow the designer to figure out if all critical tasks will meet their deadlines.

Real time computer systems differ from their general purpose counterparts in two important ways. First, they are much more specific in their applications and second, the consequences of their failure are more drastic. The real time computer is typically designed for a specific application, for example to control a particular aircraft. The advantage of this is that the characteristic of the application and its operating environment are more precisely known that for general purpose machines. As a result, it is possible to fine tune real time systems more precisely for optimum performance.

A task in real-time systems is classified in one of two ways: by the predictability of their arrival and by the consequences of their not being executed in time.

Periodic and aperiodic task: Tasks that are executed repeatedly are called periodic tasks. The periodicity of these tasks is known to the designer, and so such tasks can be pre-scheduled. An aperiodic task occurs occasionally. By their very nature, the arrival and workload of aperiodic task cannot be predicted and sufficient computing power must be held in reserve to execute them in a timely fashion. Aperiodic tasks with a bounded inter-arrival time are called sporadic tasks.

Critical and non critical tasks: Critical tasks are those where the violation of a deadline is catastrophic and while non-critical tasks are those where the violation of a deadline is tolerable to some acceptable degree.

Scheduling work in hard real time systems is traditionally dominated by the notion of absolute guarantee that is task deadline must met if the system has to avoid catastrophic consequences. Static analysis is used to determine that all deadlines are met even under the worst case load conditions. With fault-tolerant hard real-time systems this deterministic view is usually preserved even though faults may occur. No fault-tolerant system can however, cope with an arbitrary number of errors in a bounded time. The scheduling guarantee is thus given under the assumption of a certain fault model. If the occurrences of faults are no worse than that assumed in the fault model then all deadlines are guaranteed. The disadvantages of this separation of scheduling guarantee and fault model is that it leads to simplistic analysis; either the system is schedulable or it is not. In this thesis, scheduling issues and errors to justify the notion of probabilistic guarantee for a hard real time system are addressed. By 'probabilistic guarantee' we mean a scheduling guarantee with an associated probability. Hence, a guarantee of 99.95 % does not mean that 99.95 % of the deadlines are met. Rather it implies that the probability of all deadlines being met during given period of operation is 99.95%.

4. Related work:

One of the first scheduling mechanisms for fault tolerance purposes was described by Liestman and Campbell [16]. Their proposed mechanism only deals with periodic tasks whose periods must be multiples of one another and the execution time of recovery tasks must be shorter than that of the original execution time.

The work in [17] presented a scheme which can be used to tolerate faults during the execution of pre-emptive real time tasks. It describes 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.

In [18], a framework for light-weight node-level fault tolerance is presented in addition to a set of mechanisms that should be used in order to achieve fault tolerance. The approach for achieving fault tolerance is based on a real-time kernel that supports light-weight node-level fault tolerance through time-redundant execution of tasks and the use of several software-implemented error-detection techniques. The result in [18] show that NLFT-nodes may provide 55% higher reliability after one year and 60% higher mean time to failure (MTTF) compared to systems with fail-silent nodes.

The work in [19] evaluates a real time kernel that employs TEM for brake-by-wire applications using fault injection. Transient bit flips were injected into the internal registers and flip flops of the CPU that executed the kernel, The experiments show that the percentage of correct results increased from 81% to 89% using temporal error masking, while fail-silent failures decreased from 17% to 10% and value failures decreased from 1.7% to 1.1%. In [20], the focus is on the execution of kernel code rather on the application task which was the focus in [19]. The experimental results showed that, the percentage of detected errors when injecting faults during the execution of each kernel function increased from 93.9% to 97.2 %.

In [21] Pandya and Malek analyze the schedulability of a set of periodic tasks that are scheduled using Rate Monotonic Scheduling and tolerate a single fault. In the presence 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%.

In [22], Ramos-Thuel presented static and dynamic allocation strategies to provide fault tolerance. Two algorithms were proposed to reserve time for the recovery of periodic real-time tasks on a uniprocessor.

The reservation algorithm for scheduling fault-recovery modules and dynamic on-line redundant time allocation strategies was based on the concept of slack stealing.

In [23], the notion of probabilistic guarantee for fault-tolerant hard real-time systems is introduced. Schedulability and sensitivity analysis is used together to establish a maximum fault frequency that a system can tolerate. The fault model is then used to derive the probability that, during the lifetime of the system, faults will not arrive faster than this maximum rate.

Another study presented, in [24], provides an exact schedulability test for fault-tolerant task sets. Time redundancy is used to provide predictable performance in the presence of faults. In [25], a temporal-redundancy-based recovery technique is proposed that tolerates transient task failures in statically-scheduled distributed embedded systems where tasks have timing, resource, and precedence constraints. It is based on taking advantage of task set spare capacity. The amount of spare capacity is distributed over a given period so that task faults can be handled.

In [26], a scheme is presented to guarantee that the execution of real time tasks can tolerate transient and intermittent faults assuming a queue-based scheduling technique. The scheme is based on reserving sufficient slack in the schedule such that a task can be re-executed before its deadline without compromising the guarantees given to other tasks. Only enough slack is reserved in the schedule to guarantee fault tolerance if at most one fault occurs within a given time interval.

In [27], an appropriate schedulability analysis based on response time analysis, for supporting fault-tolerant hard real time systems is proposed. This paper proposed a technique to make use of error recovery technique to carry out fault tolerance. The proposed schedulability takes into account the fact that the recoveries of task may be executed at higher priority levels, since after errors, tasks certainly have a shorter period of time to meet their deadlines.

Time redundant execution of application tasks is addressed in [28]. Whenever transient faults are detected, the affected component is turned off and reintegrated immediately by retrieving the uncorrupted state of the actively redundant partner component.

The work in [30], a scheme is proposed that guarantees the timely recovery from multiple faults within hard real-time constraints in uniprocessor system. Assuming earliest-deadline-first scheduling (EDF) for aperiodic pre-emptive tasks, a necessary and sufficient feasibility check algorithm for fault tolerant scheduling is proposed. In this thesis work, similar approach as in [30] is made for rate monotonic scheduling (RM) for periodic tasks.

In [31], an approach is made to tolerate a single transient-fault within the least common multiple of periods of a set of pre-emptive periodic tasks where the task deadline is equal to its period. The scheduling algorithm used in [31] is rate monotonic (RM). It also finds the probabilistic measure of the system schedulability in case of a single fault in uniprocessor system. In this thesis, the probabilistic estimate in [31] is extended to take into account for multiple transient faults.

The ability to tolerate faults is an extremely desirable characteristic for hard real-time systems. It involves both suitable schedulability analysis, which takes effect of possible faults into account, and fault-tolerance mechanisms, which keeps the system computation complying with the specification even in the presence of faults. Due to the required fault-tolerance /schedulability-analysis synergy and the uncertain nature of faults, most papers published up to date have proposed restrictive and/or ad-hoc solutions to the schedulability analysis. As a result, those solutions impair the applicability and flexibility of the analysis. By contrast, recent approaches [23,24] based on the well known response time analysis has eliminated

these drawbacks by not restricting the way that fault tolerance is carried out and by assuming realistic fault and task modules.

5. Problem Statement:

Real-time systems with high dependability requirements are traditionally built with massive replication and redundancy. Research within the field of the fixed priority scheduling theory has mainly focused on the provision of feasibility tests which determines if a given task set is schedulable that are susceptible to a single fault [21,29, 31].

In this thesis, a schedulability analysis for tolerating multiple transient faults is proposed in conjunction with a probability of having a feasible schedule in case of at most *f* faults with one least common multiple of periods of periodic tasks. Many papers have addressed the problem of tolerating transient faults with some restrictions as follows:

- Only one fault is possible within some operational time.
- Tasks periods are multiple of one another.
- Recovery task has smaller execution time than that of the original task.
- Recovery task has higher priority than that of the original task.

In contrast, the only limitation in this thesis is on the number of maximum faults *f* within one least common multiple of periods. This assumption is reasonable since no system can tolerate an unbounded number of faults.

The work started with the following question:

- How can multiple transient faults be tolerated in distributed real-time systems using Temporal Error Masking (TEM) technique to achieve Node-Level Fault Tolerance (NLFT)?

After establishing the approach for tolerating multiple transient faults, the next question arose:

- What is the worst-case response time in a fault free environment of each copy of each individual task?

Then when faults were considered, the following questions arose:

- How does the distribution of a maximum of *f* faults affect the worst case response time of any task copy?

- How can the existing scheduling be extended to account for any possible distribution of fault pattern?

Since, by providing more slack in the schedule, the task set becomes more likely to be schedulable, the logical question to be addressed in this thesis then became:

- How can more slack be provided in the schedule so that a task set, once un-schedulable because of their maximum execution time, becomes schedulable with some recovery mechanism and reduced execution times?

6. Proposed Solution:

Temporary faults are usually more frequent than permanent faults, and the tolerance of permanent faults requires hardware redundancy which increases dramatically the cost and complexity of its implementation. A system that employs mechanisms to tolerate temporary faults is usually more cost effective. These transient faults can impair individual computations and corrupt the internal state of a computational unit, thus giving rise to permanent malfunction. Because of their very transient nature and short duration, it is difficult to detect and locate transient faults.

In this thesis, to tolerate multiple transient faults using time redundancy, two task copies are run and then, if an error is detected, *f* more recovery copies to tolerate a maximum *f* transient faults are run. If there are at least two matching results, the output is accepted, and if all the outcomes of *f+2* copies are different, it leads to an omission failure.

To find the worst-case response time of all task copies, the algorithm in [30] was used. That algorithm was developed for EDF scheduling for aperiodic tasks, but has been adapted to RM scheduling for periodic task where task deadline is equal to its period. The algorithm finds the start time and finishing time of all tasks copies in one least common multiple of periods. This algorithm is then extended to account for any possible distribution of a maximum of *f* faults within the task set. To recover an infeasible schedule where tasks copies are considered to run up to its worst-case execution time, a probabilistic estimate is made to make the schedule feasible by considering the fact that error detection is possible before a task copy finishes its execution.

The probability of system success, denoted by $P_{success}$, is determined using resultant data from fault injection experiment in 68340[5, 18-20, 31].

The rest of the thesis is organized as follows: First, the necessary task model, background and motivation for this thesis work are given in Section 7 and 8. Then, strategy and analysis for handling multiple transient faults is presented in Section 9 and 10. In Section 11 and 12, necessary function definitions for the algorithm to find the RM schedule in fault free environment is given. Then, properties and theorems for a fault-tolerant algorithm for RM schedulability are presented in Section 13 and 14. In Section 15, a recovery algorithm is proposed to find a schedule, if possible, by reducing the original execution time of tasks. The probabilistic analysis of schedulability is given in conjunction with examples in Section 16. Next, in Section 17, the result of this thesis work is presented. Section 18 concludes the thesis with some pointers to future work.

7. Task Model:

We will consider a uniprocessor system. The task set consists of n tasks, $\Gamma = \{\tau_1, \tau_{2,\ldots} \tau_n\}$. Each task τ_i has a period T_i, and a relative deadline D_i which is equal to the period. Each task copy of τ_i has a worst case execution time C_i and has a priority P_i. The highest priority task has the lowest period T_i. The length of the planning cycle within which the tasks repeat themselves iteratively is the LCM of all task periods.

LCM=Least Common Multiple $\{T_1, T_{2,\ldots\ldots,} T_n\}$

Within one planning cycle, one or more copies of task τ_i will execute. We will denote each task copy by τ_{ij} where *j* is the j^{th} copy of task τ_i.

8. Background and Motivation:

Achieving NLFT to tolerate only one fault using TEM is addressed in [18-20]. In TEM all critical tasks are executed twice and the results are compared in order to detect errors. A third execution is started if an error is detected by the comparison, timer monitor or other Error detection mechanism (EDM). This allows the kernel to mask errors by conducting a majority vote on three results. To ensure that recovery of the erroneous task does not lead to any deadline violation, sufficient slack must be reserved in the schedule. Such schedulability analysis, in case of faults is addressed in [31] where the rate monotonic scheduling algorithm is considered and the response time analysis is used. The schedulability analysis presented in [31] is based on the assumption that there is one processor with critical tasks assigned on it, and the tasks are scheduled under fixed priority assignment. In that paper, each task τ_i is independent, periodic with period T_i, worst case computation time C_i with priority Pi and the relative deadline D_i of task τ_i is equal to its period. The Task with the longest period is given the lowest priority. For τ_i, the worst case response time it experiences with double execution and a third copy of re-execution is summed up in the following equation (1):

$$R_{i,RE} = 2 \cdot C_i + \max_{e \in hpe(i)} C_e + \sum_{k \in hp(i)} \left\lceil \frac{R_{i,RE}}{T_k} \right\rceil \cdot 2 \cdot C_k \qquad (1)$$

The parameters of equation(1) are as follows:

i) $2 \cdot C_i$, τ_i's primary execution time.
ii) C_e, the time to run the third copy of τ_e, and τ_e 's priority is greater than or equals to τ_i, where $hpe(i)=\{e \mid P_e \geq P_i\}$
iii) The execution time of all task that may pre-empt τ_i, where $hp(i)=\{j \mid P_e \geq P_i\}$

The solution to (1) is obtained iteratively by forming a recurrence relation with $R_i^0 = 2C_i$. This iterative procedures finishes either when $R_i^{m+1} = R_i^m$, which gives the worst case response time of τ_i, or when $R_i^{m+1} > D_i$, that is τ_i is considered un-schedulable.

As mentioned in [7-10, 18-20, 17], transient faults are much more common and tolerating such faults at node level rather than at system level has some advantages, it is tempting to device strategy for schedulability analysis of real time system in the presence of faults at the node level. In [31], the schedulability and reliability of a class of real-time systems is designed for achieving NLFT by employing TEM. In that work, the rate monotonic (RM) scheduling algorithm for uniprocessor systems is used with functionality of the independent periodic tasks are assumed where the timing behaviour of the tasks is considered within a specific period called the planning cycle that will repeat for the entire lifetime of the system. The length of the planning cycle is defined as the Least Common Multiple (LCM) of task periods. The assumed fault model for the task in that work is one where only one fault which could lead to possible error is considered. Our intention with this thesis is to extend the RM algorithm to tolerate multiple faults at node level.

In [4], the following three different cases were considered depending on fault occurrence and the detection mechanism for the associated error:

(i) A fault free execution.
(ii) An error detected by double execution or timer.
(iii) An error detected by Error Detection Mechanism (EDM).

The following Figure 1 shows three different scenarios using TEM: in fault free operation (i) a critical task, T is executed twice (denoted by T^1 and T^2) and a comparison of the results is made to detect errors. As the results match, a third copy does not have to be executed and the time may be used by other tasks. In (ii) an error is detected by the comparison and a third copy of task T^3 is then executed. The results of the three copies are checked by a majority vote. If the majority voter detects two matching result of the task, these are accepted as valid result of the task. Otherwise, no result is delivered, which leads to an omission failure.

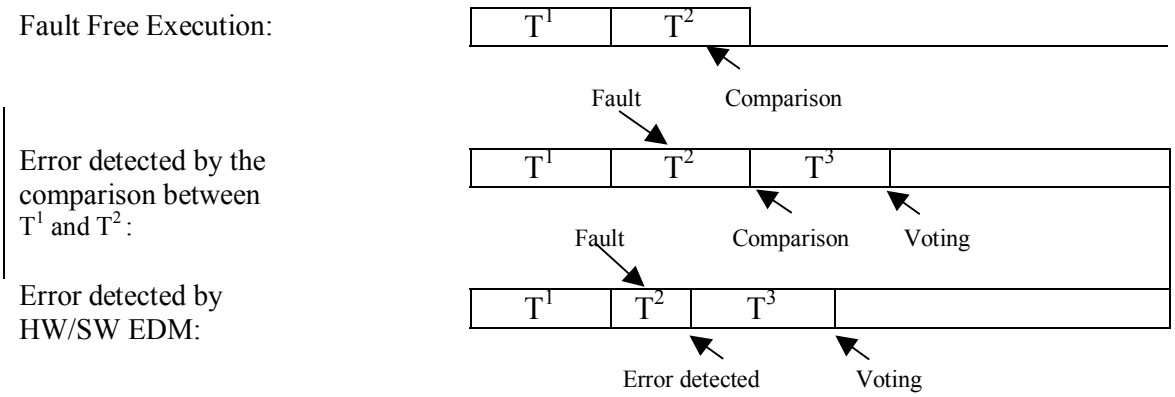


Figure1: Error detection and error recovery using temporal error masking.

In (iii), where an error is detected by a circuit level mechanism or node level mechanism, the affected copy, T^2, is terminated as soon as the error is detected and a new copy, T^3, is started immediately. Note that, when error is detected by EDM, the executing copy may run less than C_i. The new copy will be able to use time reclaimed from the terminated copy as well as time from the available slack. A comparison is made to confirm that the results match before the result is delivered. There is one more scenario similar to scenario (iii), but the fault occurs in copy T^1.

In [31], these three cases are addressed separately, independent of the corresponding error detection mechanism. However, when considering occurrence of multiple faults, addressing the three cases separately is overly optimistic since the underlying assumption must be then only one type of error detection mechanism is active in any planning cycle, that is, multiple errors within in one planning cycle are detected by the same kind of error detection mechanism. But real time systems employing fault tolerance may use different error detection mechanism as mentioned in many papers. So, within one planning cycle, multiple faults may be detected by different error detection mechanism. For example, the first error may be detected by double execution and the second error may be detected by EDM or timer. So, equation (1) should be modified to account for multiple faults. In the next section, we will find the modified equation for response time analysis and show that the schedulability analysis requires n^k combinations of faults to be considered, which is quite impractical. We then propose an algorithm as in [30] to perform the schedulability analysis. In [30], EDF scheduling is considered to account multiple faults for aperiodic tasks.

9. Strategy for handling multiple faults:

9.1 Temporal Error Masking (TEM) mechanism:

In [31], the approach to tolerate only one fault employs Temporal Error Masking (TEM), where two primary copies of each task are run first. If the results of these two run match, the output of any of the

tasks is accepted. Otherwise, when the error is detected by comparison, timer monitor, or by EDM, a third copy is run and majority voting is taken. Either the result is accepted or it leads to omission failure. The underlying assumption is that only one fault could occur while the primary copies are executing in one planning cycle.

		Outcome of third copy	
		Correct	**Faulty**
Outcome of two primary copies	Both copies are correct	Accepted without running the third copy	Accepted without running the third copy
	One of the copies is faulty lead to error	Accepted by majority voting	Omission failure

Error masking by running third copy

Table1: Error masking according to Qian's paper [31] (at most one fault).

In Table1, when both of the primary copies are correct, without running the third copy the result is accepted. And when one of the copies is faulty, by running the third copy, the result is accepted or denied based on majority voting.

When considering multiple faults that might occur in one planning cycle, this approach does not work. For example, consider two faults occurring in one planning cycle. In case of two faults, by running a third copy, we can't mask the error as shown in Table 2. In this case whenever two primary copies are faulty, running the third copy always leads to omission failure.

		Third copy	
		Correct	**Faulty**
Outcome of two primary copies	Both copies are correct	Accepted without running the third copy	Accepted without running the third copy
	One of the copy is faulty	Accepted by majority voting	Omission failure
	Both copies are faulty	Omission Failure	Omission failure

Two faults leads to omission failure and can't be masked.

Table2: Error masking is not possible in case of double faults in primary run executing.

When the number of omission failure is higher the faults are masked at system level by sacrificing the advantages that can be achieved by tolerating the faults at node level. In this paper our intention is to increase the fault tolerance at node level.

9.2 Solution: TEM to mask a maximum of *f* Number of Error:

Any schedulability analysis must rely on some fault model that is the maximum frequency or the maximum number of fault occurrence for achieving fault tolerance. If there is a bound on the number of

transient fault occurrence in one planning cycle, we can generalize the error masking mechanism. Here, we will provide a mechanism to mask a maximum of *f* faults in one planning cycle. Our underlying assumptions are the following:

- Multiple faults caused by the transient faults will not lead to same error. That is two transient faults will not lead to same type of error.
- Based on the previous assumption, it is guaranteed that, whenever we have two or more matching/same results, these results are correct and can be accepted.
- Error propagation is not possible. So, one transient fault could cause at most one error.

The approach to mask ***f*** errors is as follows: we will run two primary copies of the same task. When an error is detected by comparison, timer monitor, or by EDM we will run ***f*** more extra copies of the task to mask at most ***f*** errors. For example, in case ***f***=2, we need to run two more extra copies (third and fourth) when error is detected in any one of the primary copies. Then the four results are compared. If we have **at least** two matching results, the result is accepted. If we have all four different results, it will lead to omission failure as shown in the following table:

		Outcome of Third and Fourth copy		
		Both (3^{rd} and 4^{th}) Correct	**One is correct and one is faulty**	**Both (3^{rd} and 4^{th}) Faulty**
Outcome of two primary copies	Both copies are correct	Accepted without running the third and fourth copy	Accepted without running the third and fourth copy	Accepted without running the third and fourth copy
	One of the copies is faulty	Accepted (3 same results)	Accepted (2 same results)	Omission failure(4 different results)
	Both copies are faulty	Accepted(2 same result)	Omission Failure(all different results)	Omission failure (all different results)

Error masking by running third and fourth copy

Table 3: Masking two errors by running two more extra copies.

In [31], majority voting is taken to decide the outcome whenever a third copy is run. But in this new approach, majority voting will not work since we may have even number of task execution. To mask ***f*** errors, a total ***f+2*** of executions of a task (2 primary copies and *f* extra copies) is run in case of at most ***f*** errors. Result is delivered if there are two or more matching results out of ***f+2*** executions, otherwise, it leads to omission failure.

9.3 Fault Localities:

In our fault model, at most ***f*** faults within a planning cycle are considered which may lead to at most ***f*** errors. The presence of faults within one planning has two localities as follows:

- Case1: All ***f*** faults could occur in a single task (within the two primary executions and within the extra ***f*** copies)

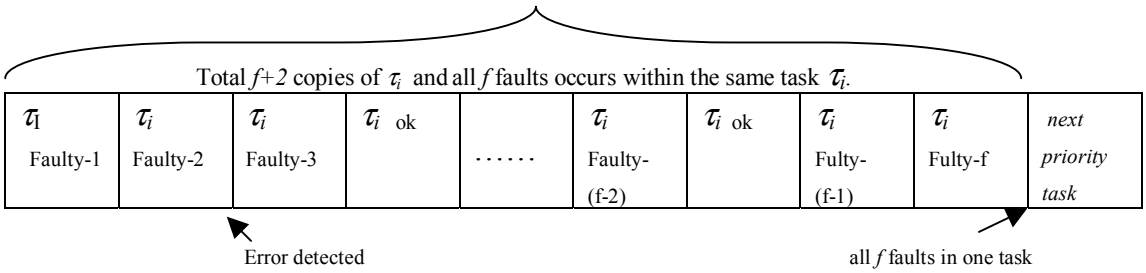


Figure 2: All *f* errors occurs in task τ_i

Faulty-i represents the i[th] faulty copy.

- Case2: All ***f*** faults could occur in different tasks.

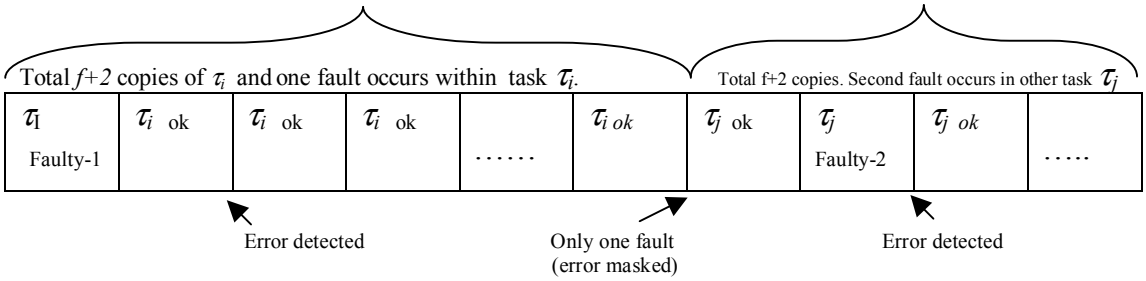


Figure 3: Errors occur in task τ_i and τ_j

9.4 Worst case Scenario (Maximum number of task run):

In fault free environment all tasks run twice hence total ***2N*** task copies are executed. When all ***f*** faults are localized within a single task (case1), the total number of different task execution is ***2N+f***, since each of the ***N*** primary tasks run twice and the only task affected by transient fault causes ***f*** more copies of the task to run. When all ***f*** faults occur in different tasks (case2), a total of **2N +*f*×N** to ***2N+f²*** copies of task are run depending on whether f< N or f ≥ N. Considering this worst case, we need to provide more slack time in schedule when masking at most *f* faults. This means that our strategy to mask multiple faults will be more effective with lower utilization.

When tolerating a total of *f* faults, the worst case response time occurs when all the *f* faults occur in different primary copies of the tasks. But fault may occur when the extra copies of any task due to an error is executing. In that case, no further extra copy is scheduled. So, when faults occur in those additional copies, number of maximum faults that maximizes the response time of the task set is reduced hence this scenario does not represents the worst case. The worst case is when all faults affect individual primary copies of task. In this paper we address all possible fault combinations for scheduling a set of task and thereby consider the worst case in our analysis.

10. Scheduling Analysis:

This section discusses how the analysis in [31] could be extended to tolerate multiple faults. For schedulability analysis the following are considered:

- Rate Monotonic priority assignment algorithm is used.
- Response time analysis is considered.
- Possible re-execution is considered in our analysis.
- Critical instance in which all tasks arrive at time 0 is considered to get the worst case response time.

The response time for each task τ_i has the following three components:

Component1: Time to execute the primary two copies of τ_i.
Component2: Time to execute the recovery copies of τ_i and higher priority tasks of τ_i.
Component3: Time to execute the primary copies of higher priority tasks of τ_i due to pre-emption.

Three techniques for error detection are considered: comparison, timer monitor and EDM.

Calculation of Component 1:

In case of comparison and timer monitor, the two primary copies of $\boldsymbol{\tau_i}$ run up to $\mathbf{2C_i}$ time. In case of EDM, any task copy may run less than $\mathbf{C_i}$ time depending on when the error is detected by software or hardware EDM.

Let, $\mathbf{C_{k\text{-}det\text{-}e}}$ denotes the maximum time the task $\boldsymbol{\tau_e}$ could run in case of $\mathbf{k^{th}}$ error detected by EDM. Any copy of a particular task $\boldsymbol{\tau_e}$ could run up to $\mathbf{C_e}$ or $\mathbf{C_{k\text{-}det\text{-}e}}$ depending on whether $\mathbf{k^{th}}$ error is detected by EDM or not.

The execution time of $\mathbf{j^{th}}$ copy of task $\boldsymbol{\tau_i}$, denoted by $\mathbf{C_{j\text{-}RM\text{-}i}}$, is given as follows:

$$C_{j\text{-}RM\text{-}i} = \begin{cases} C_i & \text{if the task copy is correct} \\ C_i & \text{if error is detected by timer or comparison} \\ C_{k\text{-}det\text{-}i} & \text{if the } k^{th} \text{ error in } j^{th} \text{ copy of task } \tau_i \text{ is detected by EDM} \end{cases}$$

Calculation of Component 2:

The maximum time the recovery copies in τ_i and in all higher priority task of τ_i could run is $C_{j\text{-}RM\text{-}e}$ for each $\mathbf{e \in hpe(i)}$ and j>2, where *e* is a task in the set $\boldsymbol{hpe(i)=\{e \mid P_e \geq P_i\}}$. This factor is the sum of all execution time of the extra copies (j>2) of task τ_i and any higher priority task of τ_i that run due to error detection. So, the factor is the following:

$$\sum_{\substack{e \in hpe(i) \\ \text{and } j>2}} C_{j\text{-}RM\text{-}e}$$

The term $C_{j\text{-}RM\text{-}e}$ is the component 1 for j^{th} copy of task τ_e.

Calculation of Componet3:

The response time for task τ_i is given by the following formula:

$$R_{i,RE} = (\text{Component1} + \text{Component2}) + \sum_{k \in hp(i)} \left\lceil \frac{R_{i,RE}}{T_k} \right\rceil \cdot (C_{1\text{-}RM\text{-}k} + C_{2\text{-}RM\text{-}k})$$

Each higher priority task τ_k will run a total of $\lceil \frac{R_{i,RE}}{T_k} \rceil$ times within the response time ($R_{i,RE}$) of task τ_i. Each time two primary copies will run which are denoted by $C_{1\text{-RM-k}}$ and $C_{2\text{-RM-k}}$. Time to execute the higher priority task of τ_i due to pre-emption is thus:

$\sum_{k \in hp(i)} \lceil \frac{R_{i,RE}}{T_k} \rceil \cdot (C_{1\text{-RM-k}} + C_{2\text{-RM-k}})$ where ***hp*** **(i)={e_l P_e>P_i}**.

10.1 Scheduling of different cases:

10.1.1 Case1: Fault free execution.

As we assumed in the previous section, there are multiple faults occur within a LCM, the faults may either occur during the primary executions of the tasks, or at any other point in time, which does not affect the tasks' executions. Figure 4 shows these two possible situations of fault occurrence.

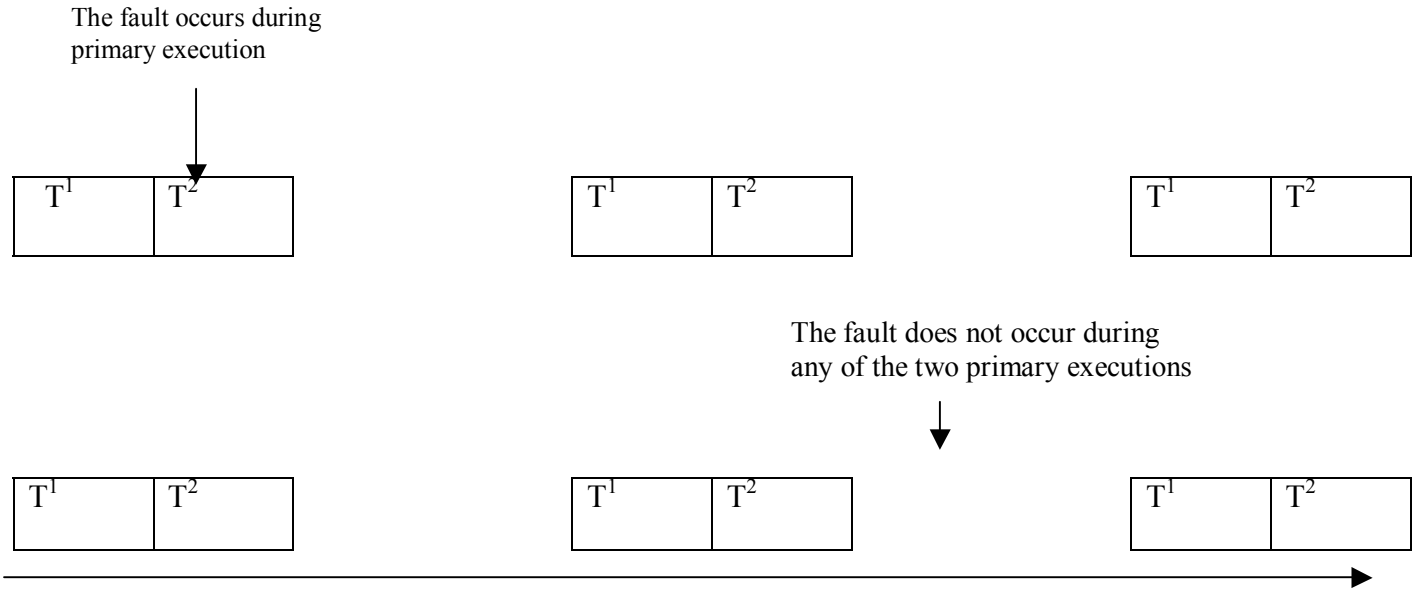


Figure 4: Possible occurrence of faults

Thus, this case of fault-free operation could be considered as three different sub-cases:

1a) Faults don't occur during any of the two primary executions of the tasks
1b) Faults occur during a primary execution, but no error is generated by the transient fault.
1c) Faults occur during a primary execution, an error is generated, but the error is not detected.

The worst case response time for τ_e is derived in the following equation. Since no error is actually detected by comparison or EDM, no task executes any extra copy, hence there is no C_e added in the following equation:

$$R_{i,FF} = 2 \cdot C_i + \sum_{k \in hp(i)} \lceil \frac{R_{i,FF}}{T_k} \rceil \cdot 2 \cdot C_k$$

10.1.2 Case 2: Error detected by EDM/DoubleExecution/Timer

This scenario will occur, when an error is detected by EDM, double execution or by a timer monitor of task τ_e and is masked by TEM by running f extra recovery copies of τ_e. In double execution, error is detected by comparison after the primary two copies has been executed and then f more copies of the task are scheduled. When using timer monitor or EDM, there is the possibility that after or during the first primary execution, the error could be detected. At this point we need to schedule $f+1$ copies of the task. When the second execution of the primary copy violates the timer requirement or error in this second copy is detected by EDM, we need to schedule f more copies of the task provided, that the first primary execution was not error detected by timer or EDM. When any error within the two primary executions is detected by double execution or by a timer, then f extra recovery copies are scheduled. Fault may occur within these extra copies and may cause error that could be detected by timer or by EDM. When the f extra recovery copies of any task is running, no more copies will be scheduled even if error is detected by timer or EDM. Such erroneous execution of any extra copy will be considered as incorrect results when trying to match two or more correct result at the end of $f+2$ copies execution.

In case of an error detected by EDM, copies of a task may not run up to C_i time as depicted by the following figures:

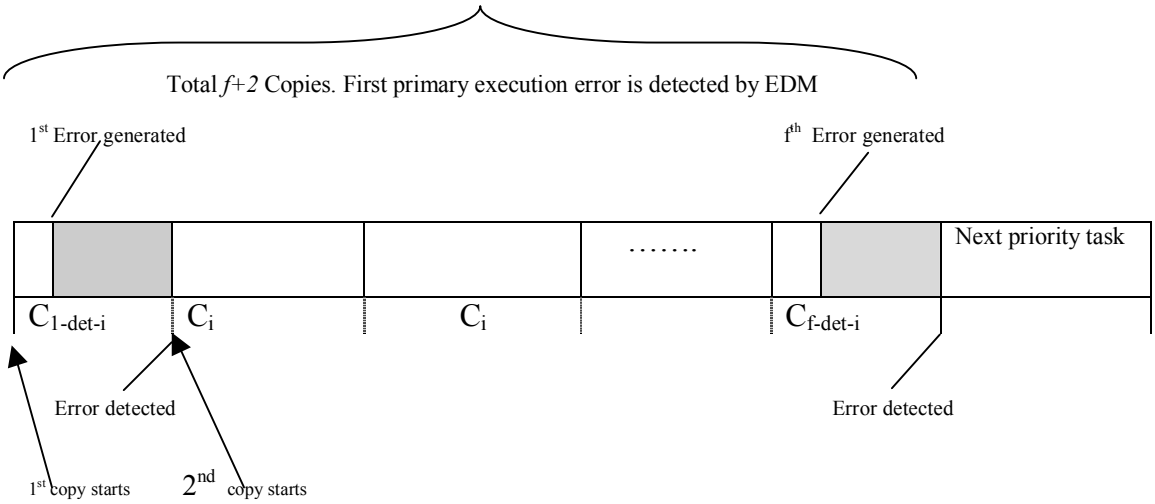


Figure 5: Error detected by EDM only in the first primary copy

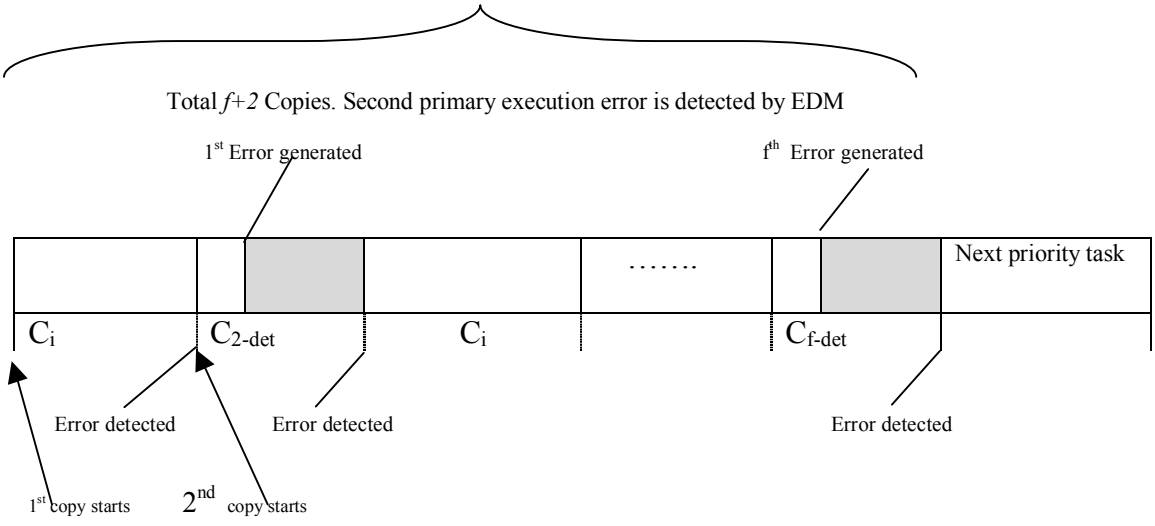


Figure 6: Error detected by EDM only in the second primary copy

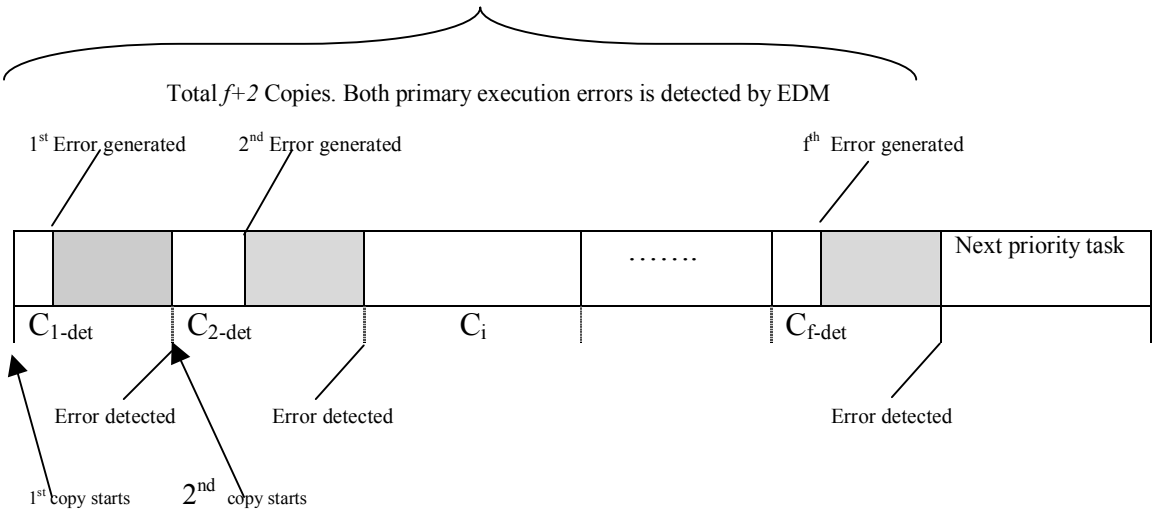


Figure7: Error detected by EDM in the first two primary copies

10.2 Response time analysis:

The formula for response time analysis given in equation (1) is not effective for analysis of multiple faults. When considering multiple faults, equation (1) is not sufficient to find the response time of a task. Since response time analysis in (1) takes into consideration only of a single fault which can occur when the task is executing or in any higher priority tasks. But for multiple faults, we have to take into consideration of any fault that may occur in any copy of a particular task in one LCM. Since two different faults may affect two different copies of particular task.

For multiple faults, the worst case response time for task τ_i is given by:

$$R_{i,RE} = (C_{1\text{-RM-i}} + C_{2\text{-RM-i}}) + C_{j\text{-RM-e}} + \sum_{\substack{k \in hp(i) \\ e \in hpe(i) \\ \text{and } j>2}} \left\lceil \frac{R_{i,RE}}{T_k} \right\rceil \cdot (C_{1\text{-RM-k}} + C_{2\text{-RM-k}}) \quad \ldots \quad \ldots (2)$$

From now on for task τ_i, its j^{th} copy within the LCM will be denoted by τ_{ij}.

10.3 EXAMPLE (Response time analysis):

Using equation (2) the schedulability analysis can be performed as follows for the following task sets:

	Period (T_i)	Execution Time (C_i)
τ_1	6	1
τ_2	9	2

Table 4: Task set

The planning cycle is 18.
τ_1 will execute three copies with in the planning cycle τ_{11}, τ_{12} and τ_{13}.
τ_2 will execute two copies with in the planning cycle τ_{21} and τ_{22}.

Case1: Fault Free Execution

Under fault free execution equation (2) reduces to:

$$R_{i,RE} = 2 \cdot C_i + \sum_{k \in hp(i)} \lceil \frac{R_{i,RE}}{T_k} \rceil \cdot 2 \cdot C_k \qquad (3)$$

For τ_1: Since τ_1 is the highest priority task, it runs without pre-emption by other tasks.

$R_{1,FF} = 2 \cdot 1 = 2$

For τ_2: There are preemptions caused by τ_1 when τ_2 executes.

$2 \cdot 2 + \lceil \frac{1}{9} \rceil \cdot 2 \cdot 1 = 6$

$2 \cdot 2 + \lceil \frac{6}{9} \rceil \cdot 2 \cdot 1 = 6$

Thus, $R_{2,FF}=6$.

Case 2: Faulty Execution

Now, when recovery copies are considered assume f=2, we consider two different scenarios.

Scenario one: One fault occurs in τ_1 and the error is detected by comparison after two of the primary copies of τ_{12} have executed.

Scenario two: Two faults occur in τ_1 and the error is detected by comparison after two of the primary copies of τ_{12} have executed and by timer monitor after τ_{13} *executes*.

Scenario one:

Execution time of τ_{11}: $C_{1\text{-RM-}1}=1$ $C_{2\text{-RM-}1}=1$	Execution time of τ_{13}: $C_{7\text{-RM-}1}=1$ $C_{8\text{-RM-}1}=1$	Execution time of τ_{22}: $C_{3\text{-RM-}2}=2$ $C_{4\text{-RM-}2}=2$
Execution time of τ_{12}: $C_{3\text{-RM-}1}=1$ $C_{4\text{-RM-}1}=1$ (Error is detected here by comparison) $C_{5\text{-RM-}1}=1$ (The third copy of the task) $C_{6\text{-RM-}1}=1$ (The third copy of the task)	Execution time of τ_{21}: $C_{1\text{-RM-}2}=2$ $C_{2\text{-RM-}2}=2$	

Figure 8: Execution time of different copies of tasks (Scenario one).

For τ_1: Since τ_1 is the highest priority task, it runs without pre-emption by other tasks, but the time for re-execution needs to be considered.

$R_{1,RE}$= 1· 2+2=4 (Response time of τ_{11})

For τ_2: There are pre-emption caused by τ_1 when τ_2 executes.

$2.2+ 2 +\lceil \frac{1}{9} \rceil .2.1=8$

$2.2+ 2 +\lceil \frac{8}{9} \rceil .2.1=8$

So, $R_{2,RE}$=8. Hence τ_2 is schedulable.

Scenario two:

Execution time of τ_{11}: $C_{1\text{-RM-}1}$=1 $C_{2\text{-RM-}1}$=1	Execution time of τ_{12}: $C_{3\text{-RM-}1}$=1 $C_{4\text{-RM-}1}$=1 (Error is detected here by comparison) $C_{5\text{-RM-}1}$=1 (The third copy of the task) $C_{6\text{-RM-}1}$=1 (The third copy of the task)	Execution time of τ_{21}: $C_{1\text{-RM-}2}$=2 $C_{2\text{-RM-}2}$=2
	Execution time of τ_{13}: $C_{7\text{-RM-}1}$=1 $C_{8\text{-RM-}1}$=1 (Error is detected here by timer) $C_{9\text{-RM-}1}$=1 (The third copy of the task) $C_{10\text{-RM-}1}$=1 (The third copy of the task)	Execution time of τ_{22}: $C_{3\text{-RM-}2}$=2 $C_{4\text{-RM-}2}$=2

Figure 9: Execution time of different copies of tasks.

For τ_1: Since τ_1 is the highest priority task, it runs without pre-emption by other tasks, but the time for re-execution needs to be considered.

$R_{1,RE}$= 1· 2+2+2=6 (Response time of τ_{11})

For τ_2: There are pre-emption caused by τ_1 when τ_2 executes.

$2.2+ 2+2 +\lceil \frac{1}{9} \rceil .2.1=10$

Here after the first iteration the response time of task τ_2 is greater than its deadline D_2=9. Hence the iteration terminates.

Since, $R^1_{2,RE}$=10>D_2=9, the task τ_2 is not schedulable.

Consider another task set:

	Period (T_i)	Execution Time (C_i)
τ_1	10	1
τ_2	20	4
τ_3	40	6

Table 5: Task set

The planning cycle is 40.
τ_1 will execute three copies with in the planning cycle: τ_{11}, τ_{12}, τ_{13} *and* τ_{14}.
τ_2 will execute two copies with in the cycle: τ_{21} *and* τ_{22}.
τ_3 will execute one copy with in the cycle: τ_{31}.

Case1: Fault Free Execution

For τ_1: Since τ_1 is the highest priority task, it runs without pre-emption by other tasks.

$R_{1,FF}= 2\cdot 1=2$

For τ_2: There are pre-emption caused by τ_1 when τ_2 executes.

$$2\cdot 4+\lceil\frac{1}{20}\rceil\cdot 2\cdot 1=10$$

$$2\cdot 4+\lceil\frac{10}{20}\rceil\cdot 2\cdot 1=10$$

Thus, $R_{2,FF}=10$.

For τ_3: There are pre-emption caused by τ_2 when τ_3 executes.

$$2\cdot 6+\lceil\frac{1}{40}\rceil\cdot 2\cdot 1+\lceil\frac{1}{40}\rceil\cdot 2\cdot 4=22$$

$$2\cdot 6+\lceil\frac{22}{40}\rceil\cdot 2\cdot 1+\lceil\frac{22}{40}\rceil\cdot 2\cdot 4=22$$

Thus, $R_{3,FF}=22$.

Case2: Faulty Execution

Now, when recovery copies are considered, assume f=2, we consider two different scenarios:

Scenario one: One fault occurs in τ_1 and the error is detected by comparison after two of the primary copies of τ_{12} execute and another fault occurs in τ_2 and the error is detected by comparison after two of the primary copies of τ_{21} execute.

Execution time of τ_{12}:
$C_{3\text{-}RM\text{-}1}=1$
$C_{4\text{-}RM\text{-}1}=1$ (Error is detected here by comparison)
$C_{5\text{-}RM\text{-}1}=1$ (The third copy of the task)
$C_{6\text{-}RM\text{-}1}=1$ (The third copy of the task)

Execution time of τ_{21}:
$C_{1\text{-RM-}2}=4$
$C_{2\text{-RM-}2}=4$ (Error is detected here by comparison)
$C_{3\text{-RM-}2}=4$ (The third copy of the task)
$C_{4\text{-RM-}2}=4$ (The third copy of the task)

For τ_1: Since τ_1 is the highest priority task, it runs without pre-emption by other tasks, but the time for re-execution needs to be considered.

$R_{1,\,RE}= 1\cdot 2+2=4$ (Response time of τ_{11})

For τ_2: There are preemptions caused by τ_1 when τ_2 executes.

$$2\cdot 4+ 2+8 +\lceil \frac{1}{20} \rceil \cdot 2\cdot 1=20$$
$$2\cdot 4+ 2+8 +\lceil \frac{20}{20} \rceil \cdot 2\cdot 1=20$$

So, $R_{2,\,RE}=20$. Hence τ_2 is schedulable.

For τ_3: There are pre-emption caused by τ_1 and τ_2 when τ_3 executes.

$$2\cdot 6+ 2+8 +\lceil \frac{1}{40} \rceil \cdot 2\cdot 1+\lceil \frac{1}{40} \rceil \cdot 2\cdot 2=28$$
$$2\cdot 6+ 2+8 +\lceil \frac{28}{40} \rceil \cdot 2\cdot 1+\lceil \frac{28}{40} \rceil \cdot 2\cdot 2=28$$

So, $R_{3,\,RE}=28$. Hence τ_3 is schedulable.

Scenario two: one fault occurs in τ_2 and the error is detected by comparison after two of the primary copies of τ_{21} execute and another is in τ_3 and the error is detected by comparison after two of the primary copies of τ_{31} execute.

For τ_1: Since τ_1 is the highest priority task, it runs without pre-emption by other tasks, but the time for re-execution needs to be considered.

$R_{1,\,RE}= 1\cdot 2=2$ (Response time of τ_{11})

For τ_2: There are preemptions caused by τ_1 when τ_2 executes.

$$2\cdot 4+8+\lceil \frac{1}{20} \rceil \cdot 2\cdot 1=18$$
$$2\cdot 4+8+\lceil \frac{18}{20} \rceil \cdot 2\cdot 1=18$$

So, $R_{2,\,RE}=18$. Hence τ_2 is schedulable.

For τ_3: There are preemptions caused by τ_1 and τ_2 when τ_3 executes.

$2\cdot 6+ 8+12 +\lceil \frac{1}{40} \rceil \cdot 2\cdot 1+\lceil \frac{1}{40} \rceil \cdot 2\cdot 4=42$

Since after the first iteration the response time is greater than the deadline of task $D_3=40$, the iterations terminate.

Since, $R^1_{3,RE}=42>D_3=40$. Hence τ_3 is not schedulable.

10.4 Observation: Traditional Response Time Analysis

In the above schedulability analysis, it is clear from the response time analysis that for some fault patterns, the task set is schedulable while for another fault patterns the task set is not schedulable. For at most *f* faults with total *m* different copies of different tasks within one planning cycle, one has to consider all possible m^f fault patterns for schedulability analysis before a system is put into mission. Table 6 shows two examples of how the number of two possible fault pattern increases with *f*. In the first example there are 5 copies of different tasks in one planning cycle. In the second example there are 7 tasks within one planning cycle. Moreover, each primary copy runs twice.

Maximum *f* Faults	Example 1 (10 primary copies)	Example 2 (14 primary copies)
f=1	$10^1=10$	$14^1=14$
f=2	$10^2=100$	$14^2=196$
f=3	$10^3=1000$	$14^3=2744$
f=4	$10^4=10000$	$14^4=38416$
F=5	$10^5=100000$	$14^5=537824$

Table 6: Number of possible fault patterns increases exponentially as f increases.

If the number of possible task copies is higher so as the number of possible fault combination that grows exponentially. Our objective in this study is to find a suitable analysis that can determine whether a task set is schedulable or not for any fault pattern. To that end, we can not consider traditional response time analysis but must take into consideration of any possible fault patterns. In order to find the efficient way to calculate the response time to see whether a task set is feasible or not, the response time of individual task copy will be taken into consideration. We mentioned in section 9.4, the worst case response time for the task set occur when each of the maximum *f* faults affect different task's one primary copy but not the same task's other primary copy or any of the recovery copies. In this thesis, we extend the fault free response time analysis of individual task copy to account for the worst case fault pattern for a maximum of *f* faults.

In the above examples, the error detection mechanism assumed is either comparison or timer in which cases the primary copies run total $2C_i$ time units. When EDM is used, the error may be detected earlier and this early detection of error provides more slack time in the schedule which could mean that task sets that were previously unschedulable become schedulable. In this study, an algorithm is proposed to find the maximum time that the erroneous copy of a task can run when the error is detected by EDM mechanism.

11. Some Functions Definitions:

Next we define a number of functions for our analysis.

The set of tasks that gets ready at time t is denoted by RD(Γ,t). That is,

RD(Γ,t)={ τ_{ij} | $\tau_i \in \Gamma$ and t=(j-1)· T_i for some j=1,2,…. $\lceil \frac{LCM}{T_i} \rceil$.}

RD(Γ,t) represents a set at time t such that τ_i is in set Γ and the j^{th} copy of task τ_i is released at time t which is multiple of the task's period T_i, that is, t=(j-1)· T_i.

Each task τ_i will run total $\lceil \frac{LCM}{T_i} \rceil$ copies within one planning cycle. Each copy of task is denoted by τ_{i1}, $\tau_{i2}, \tau_{i3}, \ldots \tau_{i\lceil \frac{LCM}{T_i} \rceil}$.

For example for the task set in Table 7:

	Period (T_i)	Execution Time (C_i)
τ_1	10	1
τ_2	20	4
τ_3	40	6

Table 7: Task set

RD(Γ,0)={ $\tau_{11}, \tau_{21}, \tau_{31}$ }
RD(Γ,10)={ $\tau_{12},$}
RD(Γ,20)={ τ_{13}, τ_{22}}
RD(Γ,30)={ τ_{14}}
RD(Γ,other time)={}
……

The RM schedule of task set Γ is denoted by the following function

$$\textbf{RM}(\Gamma,t)=\begin{cases}\tau_{ij} & \text{if RM schedules } \tau_{ij} \text{ between t and t+1}\\ \uparrow & \text{if RM does not schedule any task within t and t+1}\end{cases}$$

where t=0,1,2,…….. represents time. We will use **RM(Γ)** to refer to the RM schedule of Γ.

……

We define ***fin***(τ_{ij}) to be the time when task τ_{ij} completes execution in RM(Γ), and we define the function slack(t_1,t_2) to be the number of free slots between $t=t_1$ and $t=t_2$. That is the number of free slots for which RM(Γ,t)= $\uparrow$(excluding the slot that starts at t_2). RM(Γ) is said to be feasible if ***fin***(τ_{ij})≤ j · T_i for all i=1,2,….n and j=1,2,….. $\lceil \frac{LCM}{T_i} \rceil$.

We define the **REL_LD(t)** be the sum of execution time of all tasks that are released at time t(including double execution).

That is, **REL_LD(t)**= $\sum_{\tau_{ij} \in RD(\Gamma,t)} 2C_i$

…….

We define Γ_{all} as the sequence of all task copies within the planning cycle. That is,

$\Gamma_{all}=\{\tau_{ij} \mid i=1,2,\ldots n \text{ and } j=1,2,\ldots.. \lceil \frac{LCM}{T_i} \rceil\}$

For example, for the above task set in Table 7, $\Gamma_{all}=<\tau_{11}, \tau_{12},\tau_{13},\tau_{14},\tau_{21}, \tau_{22}, \tau_{31}>$. Note that Γ_{all} is a sequence, that is, elements are ordered (τ_{11} is the first element, τ_{12} is the second element and τ_{31} is the last element).

……

Indexof(e,S) returns the position of the element e in sequence S.

For example, Indexof(τ_{21}, Γ_{all})=5

……

We define a function

$$\textbf{Sub(a,b)}=\begin{cases} a-b & \text{if } a>b \\ 0 & \text{otherwise} \end{cases}$$

……

Inspired by the work in [30], We define a function $\Psi(\Gamma,t)$ that finds the amount of work(accumulated execution time) that remains to be completed at time t in RM(Γ). This work is generated by the tasks that become ready at or before time t, that is by the tasks in $\{\tau_{ij} \in \Gamma_{all} \mid T_i \cdot (j-1) \le t\}$.

$$\textbf{Ψ}(\Gamma,t)=\begin{cases} \sum_{\tau_{ij} \in RD(\Gamma,t)} 2C_i & \text{if } t=0 \\ \text{sub}(\ \textbf{Ψ}(\Gamma,t-1)\ ,\ 1\ ) & \text{if } t=LCM \\ \text{sub}(\ \textbf{Ψ}(\Gamma,t-1)\ ,\ 1\ ) + \sum_{\tau_{ij} \in RD(\Gamma,t)} 2C_i & \text{if } t<LCM \end{cases}$$

……

N defines the total number of tasks copies in one LCM that is $N=|\Gamma_{all}|$.

In the next section we develop an algorithm **RM-IND** to find the response time of individual task copy.

12. RM-IND: An algorithm for RM Schedulability

The algorithm in Figure 10 calculates the response time of individual task copy. In this algorithm, which iterates total LCM time, we determine at each time *t* which task τ_{ij} has finished execution and record the task indices (i and j) and the finish time of the primary two copies. If any task copy failed to finish before its deadline, the algorithm stops. If all task copy finish execution without violating the deadline, the algorithm stops. To maintain the record we define the following data structure:

```
record TaskInfo{
        int i;                  // Indices of the task τij
        int j;                  // the copy of task τij
        int First;              //Finish time of first primary copy
        int Second;             //Finish time of second primary copy
              }
```

// Γ_{all} is the set of all task copy, and DONE in an array of type TaskInfo.

ALGORITHM: RM-IND(Γ_{all}, DONE)

```
1  N=| Γall |                          //Total number of task copies in Γall
2  S=∅                                 //Maintains the current set of tasks not finished yet
3  t=0                                 //Loop starts from t=0 and ends at t=LCM
4  DONE[N] of type TaskInfo            //Record all task information
5  First[N]                            //Temporary storage for the first primary copy finish time

6  do while(t≤LCM)
7          if(S≠∅)then
                Find the τlk is the highest priority task in set S
8          End if
9          If(t≠ 0 and ( Ψ(Γ,t-1)- ( Ψ(Γ,t)-REL_LD(t) ))>0) then
10             τlk executes 1 time unit in t-1 to t
11             if τlk has executed Cl time units then
                      First[indexof(τlk)]=t
13             End if
14             if τlk has executed 2Cl time unit then
                      Update DONE                          //Record all information
16                    Remove τlk from set S
18             End if
19             If (t ≥ k· Tl and τlk has not executed 2Ci ) then
20                    τlk is NOT SCHEDULABLE
21                    Task set Γ is not schedulable
22                    Stop
23             End if
24       End if

25       S=S ∪ RD(Γ,t)
26       Find the τlk is the highest priority task in set S

27       If (t ≥ k· Tl and τlk has not executed 2Ci ) then
28             τlk is NOT SCHEDULABLE
29             Task set Γ is not schedulable
30             Stop
31       End if

32       Increment time t by one
33 End while
34 If S=∅ then
       Task set Γ is schedulable
       Stop
35 End if
36 Task set Γ is not schedulable
37 Stop
```

Figure 10: Pseudocode for algorithm RM-IND.

The algorithm **RM-IND** in Figure 10 finds the response time of each copy of each task.

In line 1, the number N represents the number of total task copies in Γ_{all}. In line 2, the set S is set to null to represent the set of ready task that has not yet finished execution at any time. Then, time *t* is set to zero in line 3 to repeat the loop for each *t*. In line 4, an array **TaskInfo** of size N is declared to keep track of an individual task copy and its two primary copies finishing time. In line 5, an array **First** of size N is declared to keep track of when the first primary copy of a task finishes execution.

Line 6-33 is in a while loop which runs a total of LCM times. In line 7-8, the highest priority task in set S is found out if S is not null. The line from 9-24 inside the while loop executes when any work is done in between time t-1 to t. This is determined by the condition in line 9 which is [$\Psi(\Gamma,t-1)$- ($\Psi(\Gamma,t)$-REL_LD(t))]. $\Psi(\Gamma,t-1)$ represents total work to be done at time t-1 and ($\Psi(\Gamma,t)$-RelLd(t)) represents total work to be done at time t excluding any new task load which is released at time t. $\Psi(\Gamma,t-1)$- ($\Psi(\Gamma,t)$-RelLd(t))>0 represents that in time t-1 to t the highest priority task executes for one time unit. In line 11-13, the code checks if the highest priority task has finished executing its first copy and if so, stores the finishing time in array First. In line 14-18, the code checks whether the highest priority task finish executing its second copy and if so, stores the information in array DONE and removes the highest priority task from set S.

In line 19-23, the condition checks whether the highest priority task has violated its deadline or not. In case of violation, the algorithm reports NOT SCHEDULABLE and stops. In line 25, any newly released task in added to the set S. In line 26 the highest priority task is identified. In line 27-31, the code checks whether the highest priority task has violated its deadline or not. In case of violation, the algorithm reports NOT SCHEDULABLE and stops. In line 32, time t is increased by one and iteration begins again.

After LCM time the iteration stops. Then checks whether set S is empty or not in line 34. If the set is empty then all task has scheduled without violating the deadline. If the set is not empty, task set is not schedulable.

EXAMPLE (RM-IND Simulation): Let us simulate the algorithm for the following task set:

	Period (T_i)	Execution Time (C_i)
τ_1	6	1
τ_2	9	2

Table 8: Task Set

$\Gamma_{all}=\{\tau_{11}, \tau_{12}, \tau_{13}, \tau_{21}, \tau_{22}\}$

The simulation of the algorithm RM-ANY for the above example in Table 8 of all steps are given in Figure 11 and Figure 12.

Time t	Set S	Highest priority task	$\Psi(\Gamma,t)$	REL_LD(t)	Modification to DONE	Modification to First	Comments
0	$\{\tau_{11}, \tau_{21}\}$	τ_{11}	6	6			At time zero new tasks are released
1	$\{\tau_{11}, \tau_{21}\}$	τ_{11}	5	0		First[0]=1	First copy of highest priority task finishes
2	$\{\tau_{11}, \tau_{21}\}$	τ_{11}	4	0	DONE[0].i=1 DONE[0].j=1 DONE[0].First=1 DONE[0].Second=2		Second copy of the highest priority task finishes and τ_{11} is removed from S
3	$\{\tau_{21}\}$	τ_{21}	3	0			First copy of highest priority task executes one time unit
4	$\{\tau_{21}\}$	τ_{21}	2	0		First[1]=4	First copy of highest priority task finishes
5	$\{\tau_{21}\}$	τ_{21}	1	0			Second copy of highest priority task executes one time unit
6	$\{\tau_{21}\}$	τ_{21}	2	2	DONE[1].i=2 DONE[1].j=1 DONE[1].First=4 DONE[1].Second=6		Second copy of the highest priority task finishes and τ_{21} is removed from S and τ_{12} is inserted in S
7	$\{\tau_{12}\}$	τ_{12}	1	0		First[2]=7	First copy of highest priority task finishes
8	$\{\tau_{12}\}$	τ_{12}	0	0	DONE[2].i=1 DONE[2].j=2 DONE[2].First=7 DONE[2].Second=8		Second copy of the highest priority task finishes and τ_{12} is removed from S
9	$\{\tau_{22}\}$	τ_{22}	4	4			New copy of task τ_2 is released

Figure 11: Simulation of RM-IND

Time t	Set S	Highest priority task	Ψ(Γ,t)	REL_LD(t)	Modification to DONE	Modification to First	Comments
10	$\{\tau_{22}\}$	τ_{22}	3	0			First copy of highest priority task executes one time unit
11	$\{\tau_{22}\}$	τ_{22}	2	0		Frst[4]=11	First copy of highest priority task finishes
12	$\{\tau_{22}\}$	τ_{22}	3	2			Second copy of highest priority task executes one time unit and τ_{13} is inserted in S
13	$\{\tau_{22}, \tau_{13}\}$	τ_{13}	2	0		First[5]=13	First copy of highest priority task finishes
14	$\{\tau_{22}, \tau_{13}\}$	τ_{13}	1	0	DONE[3].i=1 DONE[3].j=3 DONE[3].First=13 DONE[3].Second=14		Second copy of the highest priority task finishes and τ_{13} is removed from S
15	$\{\tau_{22}\}$	τ_{22}	0	0	DONE[4].i=2 DONE[4].j=2 DONE[4].First=11 DONE[4].Second=15		Second copy of highest priority task executes one time unit and τ_{22} is removed from S
16	{}		0	0			No more task
17	{}		0	0			No more task
18	{}		0	0			No more task. S is empty. So, the task set is RM schedulable.

Figure12: Simulation of RM-IND

The array DONE represents a sequence of finished task according to algorithm in Figure 10 in order of time as follows:

DONE[0].i=1	DONE[1].i=2	DONE[2].i=1	DONE[3].i=1	DONE[4].i=2
DONE[0].j=1	DONE[1].j=1	DONE[2].j=2	DONE[3].j=3	DONE[4].j=2
DONE[0].First=1	DONE[1].First=4	DONE[2].First=7	DONE[3].First=13	DONE[4].First=11
DONE[0].Second=2	DONE[1].Second=6	DONE[2].Second=8	DONE[3].Second=14	DONE[4].Second=15

Table9: Information stored by algorithm RM-TND for task set in Table 8

13. Properties and Theorem: Fault Tolerant Algorithm

Our objective in this paper is not to work for a particular fault pattern but to cover all possible fault patterns in general. The reason is that, by considering all possible fault patter, we are guaranteed to cover the worst case scenario.

In [30] the amount of extra work still to be done at the finishing time of any task and the amount of extra work at any other time is defined for EDF scheduling for aperiodic task. Using the similar approach as in [30], next we define these two quantities for RM scheduling with periodic task.

$\boldsymbol{\delta^f_{ij}(\ \Gamma)}$ is defined as follows as the maximum extra work at time t=fin(τ_{ij}) induced by any fault pattern with *f* faults. That is it defines the extra work at the finishing time of task τ_{ij}. Let **pre(ij)**=lk such that τ_{lk} is the task which finishes immediately before **fin(τ_{ij})**.

This maximum value can be obtained by considering the worst case scenario in each of the following two cases:

- All *f* faults have already occurred before task τ_{ij} begins. Hence the maximum extra work at *fin*(τ_{ij}) is the maximum extra work at *fin*(τ_{lk}), where τ_{lk} is the task that finishes just before τ_{ij} decremented by the slack available between *fin*(τ_{lk}) and *fin*(τ_{ij}). The quantity is :

$$Q1=sub(\ \delta^f_{pre(ij)}(\ \Gamma),\ \ slack(\ fin(pre(ij))\ ,fin(\tau_{ij})\ )$$

- All *f*-1 faults have already occurred before tasks τ_{ij} begins and a new fault occurs at τ_{ij}. Hence, the maximum extra work at *fin*(τ_{ij}) is the maximum extra work due to *f*-1 faults at *fin*(τ_{lk}) where τ_{lk} is the task that finishes just before τ_{ij} and extra work due to the fault at τ_{ij} decremented by the slack available between *fin*(τ_{lk}) and *fin*(τ_{ij}). The quantity is :

$$Q2=\sum_{k=3}^{f+2} C_{k\ -\ RM\ -\ ij}\ +\ sub(\ \delta^{f-1}{}_{pre(ij)}(\ \Gamma)\ \ ,\ slack\ (fin(pre(ij)),\ fin(\tau_{ij}))\ )$$

$\delta^f_{ij}(\ \Gamma)$ is the maximum of any of the above two quantities, defined as follows:

$$\delta^f_{ij}(\ \Gamma)=\begin{cases} 0 & \text{if f=0} \\ \sum_{k=3}^{f+2} C_{k\ -\ RM\ -\ ij} & \text{if ij=11, that is first copy of first task} \\ \max(Q1,\ Q2) & \text{Otherwise} \end{cases}$$

Next, we find the maximum extra work that need to be done for any fault pattern at any time *t*. Remember that we are considering a maximum of *f* faults. Lets define, $\delta^f(\Gamma, t)$ be the maximum amount of extra work needed in case of maximum *f* faults in one LCM. Now we need a way to calculate $\delta^f(\Gamma, t)$, which is defined as the maximum extra work at time t induced by exactly *f* faults.

$$\delta^f(\Gamma, t) = \begin{cases} 0 & \text{if } t=0 \\ sub_b(\delta^f_{ij}(\Gamma), slack(fin(\tau_{ij}), t)) & \text{if } fin(\tau_{ij}) \le t < fin(\tau_{lk}) \text{ where } \tau_{lk} \text{ finishes just after } \tau_{ij} \\ sub(\delta^f(\Gamma, t-1), 1) & \text{if } RM(\Gamma, t-1) = \uparrow \\ \delta^f(\Gamma, t-1) & \text{Otherwise} \end{cases}$$

EXAMPLE (Calculation of $\delta^f_{ij}(\Gamma)$):

	Period (T_i)	Execution Time (C_i)
τ_1	6	1
τ_2	9	2

Table 10: Task Set

Lets calculate $\delta^f_{ij}(\Gamma)$
set $\Gamma_{all} = <\tau_{11}, \tau_{12}, \tau_{13}, \tau_{21}, \tau_{22}>$

The finishing time can be obtained from array DONE of RM-IND:
$fin(\tau_{11})=2$, $fin(\tau_{21})=6$, $fin(\tau_{12})=8$, $fin(\tau_{13})=14$ and $fin(\tau_{22})=15$.

For f=1,
$\delta^f_{11}(\Gamma)=1$, $\delta^f_{12}(\Gamma)=2$, $\delta^f_{13}(\Gamma)=2$, $\delta^f_{21}(\Gamma)=1$, and $\delta^f_{22}(\Gamma)=2$.

For example, $\delta^f_{22}(\Gamma)=2$ is calculated in the following way:

$\delta^1_{22}(\Gamma) = \max(sub(\delta^1_{13}(\Gamma), slack(14,15)), 2+sub(\delta^0_{13}(\Gamma), slack(14,15)))$
$= \max(sub(2,0), 2+sub(0,0)) = \max(2,2) = 2$

For f=2,
$\delta^f_{11}(\Gamma)=2$, $\delta^f_{12}(\Gamma)=4$, $\delta^f_{13}(\Gamma)=2$, $\delta^f_{21}(\Gamma)=3$, and $\delta^f_{22}(\Gamma)=3$.

Now, $\delta^f(\Gamma, t)$ can be easily calculated for any t.

The fault tolerant schedule is denoted by $RM^f(\Gamma)$. We define a function $\Psi^f(\Gamma,t)$ to denote the amount of task needed to be done at time t for any fault pattern with exactly *f* faults as follows:

Work to be done at time t with fault=Extra work at time t with fault+ Work to be done at t without fault

In equation,

$\Psi^f(\Gamma,t) = \delta^f(\Gamma, t) + \Psi(\Gamma,t)$

Note that task τ_{ij} may complete at a time different than $fin(\tau_{ij})$ in $RM^f(\Gamma)$, the function Ψ^f has the important property that it is equal to zero only at the beginning of an idle time slot in $RM^f(\Gamma)$.

There are other properties as in [5] but modified here for RM scheduling:

Property1: $\Psi(\Gamma,t)=0$ if and only if $RM(\Gamma,t)=\uparrow$. That is $\Psi(\Gamma,t)=0$ if and only if there is no work to be done at time t in $RM(\Gamma)$, which means that any task released before t finishes at or before time t in the fault free case.

Property2: $\Psi^f(\Gamma,t)=0$ if and only if there is no work to be done at time t in $RM^f(\Gamma)$, which means that any task released before t finishes at or before time t when subjected to fault pattern *f*.

Property3: $\Psi(\Gamma,t)-\sum_{\tau_{ij}\in RD(\Gamma,t)} 2C_i=0$ if and only if all tasks released before time t is finish execution.

Property4: $\Psi^f(\Gamma,t)-\sum_{\tau_{ij}\in RD(\Gamma,t)} 2C_i=0$ if and only if all tasks released before time t is finish execution for exactly f faults.

Property5: $\Psi^f(\Gamma,t)\geq \Psi(\Gamma,t)$. That is the amount of work incurred when fault occurs is never smaller than the amount of work in fault free case.

Property6: $t=fin(\tau_{ij})$ implies that $\Psi(\Gamma,t-1)>0$. That is the slot before the end of a task is never idle.

The following Theorem 1 shows the necessary condition when a task finishes its execution in the fault tolerant schedule. The δ^f function is an abstraction that represents the extra work to be performed for recovery. This extra work reduces to zero when all ready task completes execution and recovery, as demonstrated by the following theorem which is adapted from [5] where the theorem is defined for a particular fault pattern. We modified it to take into account of any fault pattern.

Theorem1: If $\delta^f(\Gamma,t-1)>0$ and $\delta^f(\Gamma,t)=0$, then, in both $RM(\Gamma)$ and $RM^f(\Gamma)$, any task with release time less than t finishes before time *t*.

Proof: See In Appendix

The following Theorem 2 shows the necessary condition when recovery of task is progressing. The next theorem shows that, if at any time *t* the amount of extra work due to fault is positive and if there is an idle slot at that time *t* in fault free schedule $RM(\Gamma)$, then in the fault tolerant schedule $RM^f(\Gamma)$ at time t+1, the amount of extra work due to fault is reduced by one time unit, that is, recovery of the task is in progress between time t and t+1.

Theorem2: If $\delta^f(\Gamma,t-1)>0$ and $RM(\Gamma,t-1)=\uparrow$ then $\delta^f(\Gamma,t)=\delta^f(\Gamma,t-1)-1$.

Proof: See In Appendix.

Now we will demonstrate one example to show that, for a particular fault pattern, the lowest priority task may finish before deadline where as some higher priority task may miss the deadline.

	Period (T_i)	Execution Time (C_i)
τ_1	6	1
τ_2	9	2

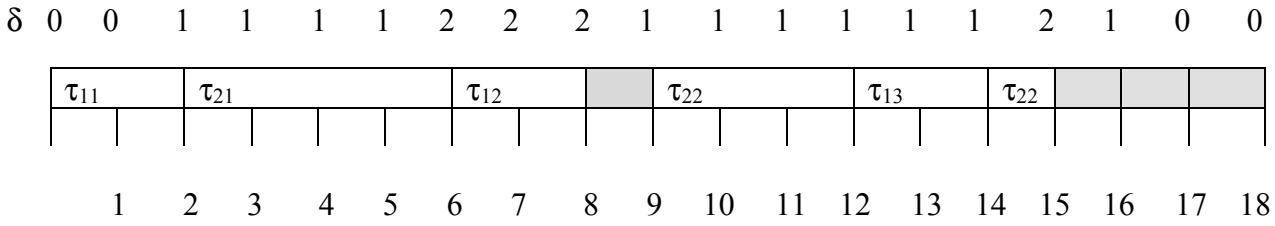
Table 11: Task Set

set $\Gamma_{all}=<\tau_{11}, \tau_{12},\tau_{13},\tau_{21}, \tau_{22}>$

$fin(\tau_{11})=2$, $fin(\tau_{12})=8$, $fin(\tau_{13})=14$, $fin(\tau_{21})=6$ and $fin(\tau_{22})=15$

Consider at most one fault. In this case, we need to schedule one extra copy in case of error.

Under Fault free execution the schedule is as follows:



Observe that, τ_{22} which is the lowest priority task, finishes its execution by time 15 and there is enough slack left to recover from an error. However, one of the higher priority tasks, $\tau_{21,}$ misses its deadline since at the end of the primary execution the value of δ is 2 at time 6. But τ_{12} will now pre-empt task τ_{21} and execute up to time unit 8. After that the recovery for τ_{21} is not possible before the deadline t=9. Hence, if one error occurs in τ_{21}, it will miss its deadline even if the lowest priority task, $\tau_{22,}$ finishes by its deadline.

The next Theorem 3 shows the necessary and sufficient condition when the lowest priority task meets its deadline.

Theorem 3: Given task set Γ and, the lowest priority task τ_{ij} in Γ completes by $T_i \cdot j$ in $RM^f(\Gamma)$, if and only if, $\delta^f(\Gamma,t)=0$ for some t, $fin(\tau_{ij}) \leq t \leq T_i \cdot j$.

Proof: See the Appendix

Since meeting the deadline of the lowest priority task doesn't guarantee that all higher priority tasks also meet the deadline, we have to repeatedly apply Theorem 3 to all task sets Γ_j, j=1,..,N to obtain a sufficient condition for the feasibility of the entire task set. So, it is not sufficient to apply Theorem 3 only to $\Gamma_{all.}$ The next corollary establishes the sufficient condition to check when an entire task set is schedulable.

Corollary 1: A necessary and sufficient condition for the feasibility of $RM^{\alpha}(\Gamma)$ for a given Γ and for given fault pattern with f or less faults can be obtained by applying Theorem 3 to N task sets Γ_j, j=1,..,N where Γ_j contains the j highest priority task in $\Gamma_{all.}$

Proof: Can be proved using induction on N. See the Appendix.

14. Fault Tolerant Algorithm: RM-FT-ANY

To apply the Corollary 1, we will now develop an algorithm RM-FT-ANY to determine schedulability under all fault patterns. The algorithm is given in Figure 13.

ALOGORITHM: RM-FT-ANY(Γ_{all})

```
1    N=| Γall |
2    Q={τ11 }
3    For j=1 to N
4        Array of TaskInfo B of size j
5        Simulate RM-IND (Q,B)
6        τlk=Lowest priority task in Q
7        for t=0 to LCM
8            if(t=fin(τij) as in B  and fin(τij)≥Tl·  (k-1))
9                if  δf(Q,t)=0 for  some fin(τij)≤t≤Ti ·  j then
                         continue with next iteration on t
                 else
                         NOT SCHEDULABLE and STOP

10               End if
11           End if
12       end for
13       if (j<N) then Q=Q∪ {τij} Where τij is the next highest priority task in Γall - Q

14   End for
15   RM SCHEDULABLE
16   STOP
```

Figure 13: Pseudocode for algorithm RM-FT-ANY.

In line 2 of this algorithm, set Q is initialized with the highest priority task, which is currently also the lowest priority task in set Q. Then in the loop 3-14, the set Q is scheduled using RM, where upon Theorem 3 is applied to the lowest priority task in set Q to check if the task is schedulable. In line 13, the next highest priority task is selected, which becomes the lowest priority task in Q.

Now we will apply this algorithm on some example task set:

EXAMPLE 1 (Simulation of RM-FT-ANY):

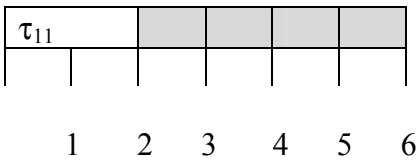
	Period (T_i)	Execution Time (C_i)
τ_1	6	1
τ_2	9	2

Lets calculate $\delta^f_{ij}(\Gamma)$
set $\Gamma_{all}=<\tau_{11}, \tau_{12},\tau_{13},\tau_{21}, \tau_{22}>$
$fin(\tau_{11})=2$, $n(\tau_{21})=6$, $fin(\tau_{12})=8$, $fin(\tau_{13})=14$, and $fin(\tau_{22})=15$

For f=1
Simulate RM-FT-ANY

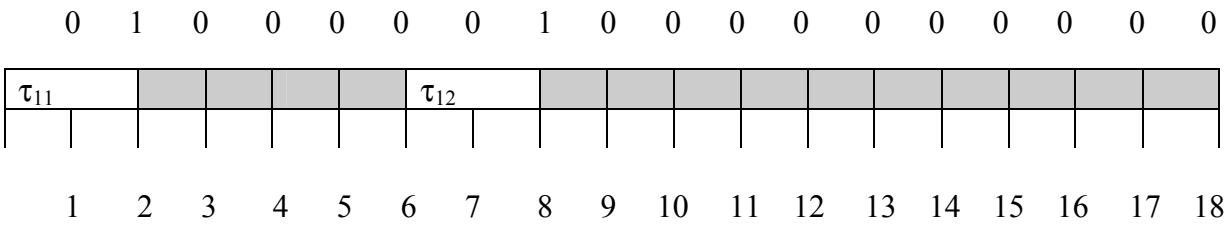
Iteration j=1
$Q=\{\tau_{11}\}$ and $\delta^f_{11}(\Gamma)=1$

δ 0 0 1 0 0 0 0



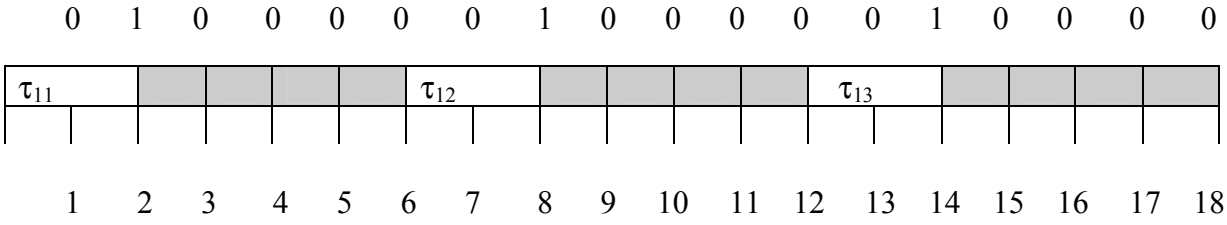
τ_{11} is RM schdedule

Iteration j=2 and Q={τ_{11} , τ_{12}}



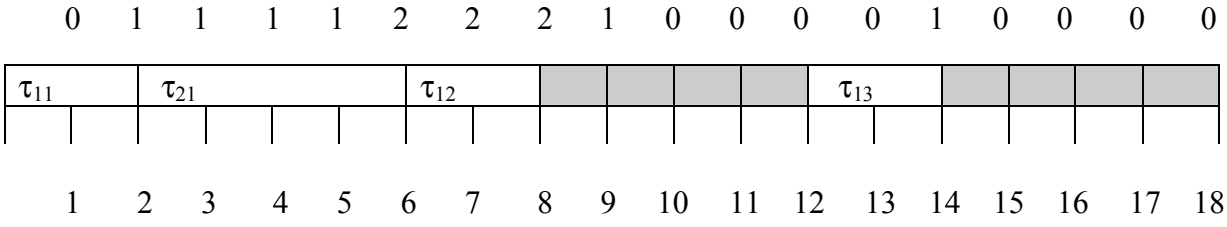
τ_{12} is RM schedulable.

Iteration j=3 and Q={τ_{11} , τ_{12}, τ_{13}}



τ_{13} is not RM schedulable.

Iteration j=4 and Q={τ_{11} , τ_{12}, τ_{13}, τ_{21}}



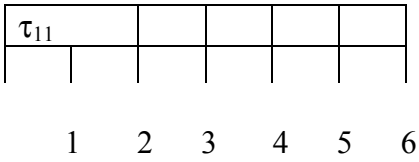
τ_{21} is not RM schedulable.

EXAMPLE 2(Simulation of RM-FT-ANY):

	Period (T_i)	Execution Time (C_i)
τ_1	6	1
τ_2	9	1

j=1
Q={τ_{11}} and $\delta^{f}_{11}(\Gamma)$=1

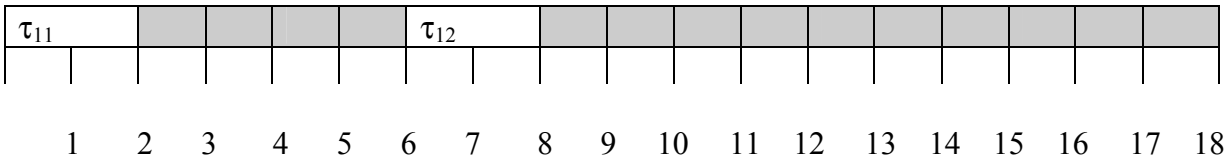
δ 0 0 1 0 0 0 0



τ_{11} is RM schdedule

j=2
Q={τ_{11} , τ_{12}}

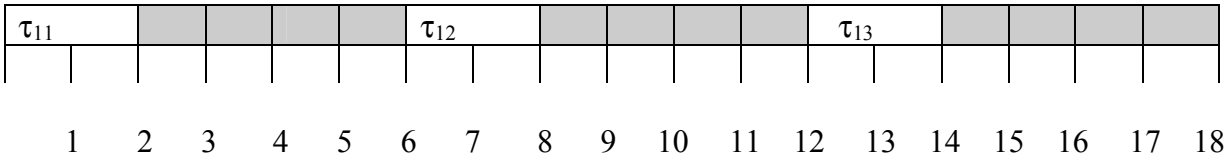
0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0



τ_{12} is RM schedulable.

j=3
Q={τ_{11} , τ_{12}, τ_{13}}

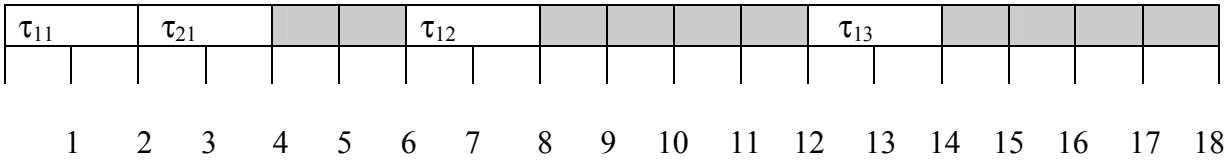
0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0



τ_{13} is RM schedulable.

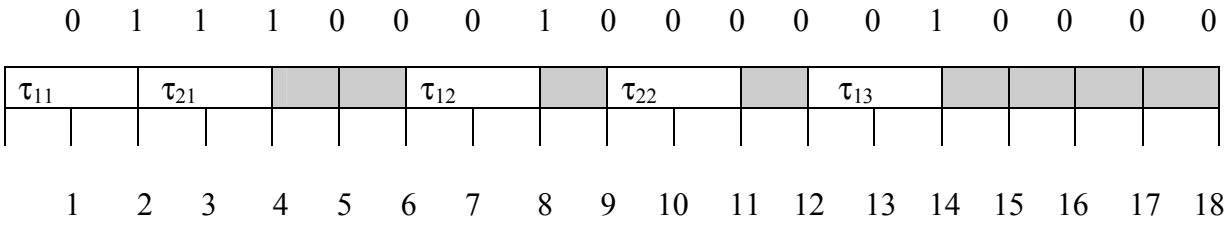
j=4
Q={τ_{11} , τ_{12}, τ_{13}, τ_{21}}

0 1 1 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0



τ_{21} is RM schedulable.

j=5
Q={τ_{11} , τ_{12}, τ_{13}, τ_{21}, τ_{22}}

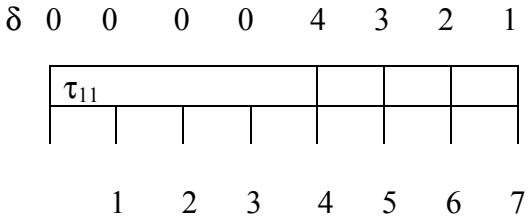


τ_{22} is RM schedulable.

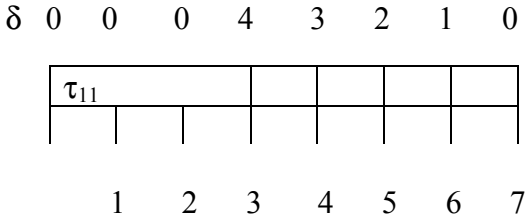
From the above examples we have seen that, if the execution time of the task decreases, a task set that was previously not schedulable may become schedulable. Consequently, a new question arises, is there any circumstance in which the execution time of a task may be reduced? The answer is yes. When an error is detected by EDM, the task copy immediately gets stopped and new copy of a task gets scheduled.

Our next objective in this work is to find the maximum time that a task copy can execute before the error in the task is detected by EDM. The next example shows how this is possible:

Consider only one task τ with C=2 and T=7 and f=2.



Here, the task is not schedulable in case of two faults. But if one error occurs in one of the primary two copies (2^{nd} copy), and the error is EDM detected after 1 time unit, then the two primary copies of the task will run a total of 3 time unit, in which case the task is schedulable in case of a maximum of two faults.



Now we will develop a strategy (MIN-EDM algorithm) to determine when a fault should occur and when the error should be detected by EDM mechanism so that task set that was previously not schedulable (when a task runs a full C time unit) becomes schedulable.

15. Fault-Tolerant Recovery Algorithm: RECOVERY-MIN-EDM

Algorithm RM-FT-ANY will fail when the lowest priority task τ_{ij} fails in set Q for which the following condition is satisfied.

if $\delta^f(Q,t)>0$ for all $fin(\tau_{ij})\le t\le T_i \cdot j$

We will now find other tasks for which a reduction in execution time either decreases δ or provides more slack in the schedule so that the extra amount of work δ can be done before the task deadline.

From RM(Γ) we can determine other consecutive tasks τ_{lk} which are executed later than τ_{ij} and also satisfy the condition $\delta^f(Q,t)>0$ for all $fin(\tau_{lk})\le t\le T_l \cdot k$ and $fin(\tau_{lk})\le T_i \cdot j$. Notice that we are concerned only with consecutive tasks of τ_{ij} that finishes after τ_{ij} but before the deadline of τ_{ij}. Denote the set of such tasks SET1. The extra work needed in these consecutive copies also includes the extra task needed at copy τ_{ij}. Hence, we are looking for tasks whose finish time is later than the release time of τ_{ij}. Consequently, if these tasks execution time can be reduced, then more slack can be found in the schedule and the task τ_{ij} can finish doing its extra work δ and able to finish its execution before its deadline in the fault tolerant schedule.

Again, the extra work needed to be done in τ_{ij} also includes the extra work of the task that finishes before τ_{ij} in RM(Γ). If we can reduce the execution time of these tasks then the factor δ for task τ_{ij} also decreases and task τ_{ij} can finish execution before its deadline. We are concerned here about the tasks those increases the δ of the task τ_{ij}. Denote this set SET2.

Now define the set

AFFECTED $(\tau_{ij}) = \{\tau_{ij}\}\cup$SET1$\cup$ SET2

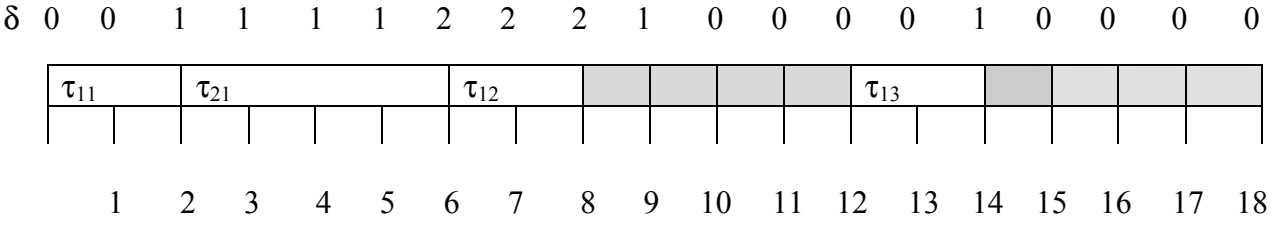
Our aim is to decrease the execution time of all tasks in AFFECTED set so that task τ_{ij} becomes schedulable. We can decrease the execution time of a task by assuming that, the task will generate one error and the error will be detected by EDM at or before the decreased execution time. Such recovery mechanism will be probabilistic since we are relying on the probability of a task copy to be error detected by EDM.

EXAMPLE: (Finding the affected set)

	Period (T_i)	Execution Time (C_i)
τ_1	6	1
τ_2	9	2

set $\Gamma_{all}=<\tau_{11}, \tau_{12},\tau_{13},\tau_{21}, \tau_{22}>$

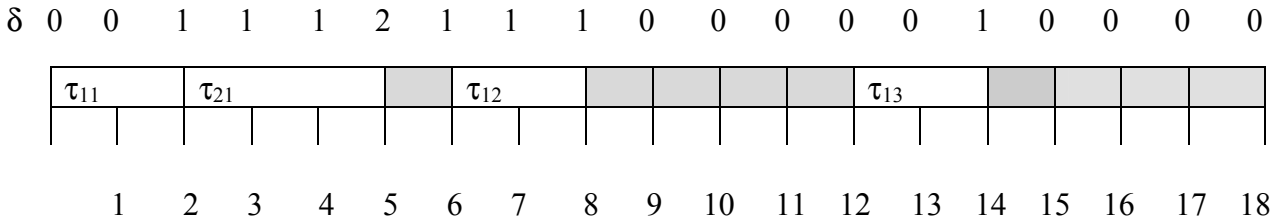
For f=1, when scheduling using algorithm **RM-FT-ANY,** the task τ_{21} cannot be scheduled at iteration j=4.



Task τ_{21} (with deadline t=9) can not be scheduled because δ=1 at deadline t=9 and hence needs recovery.

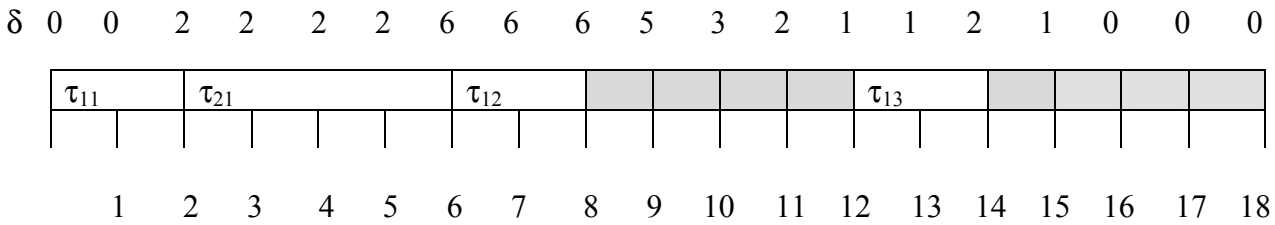
AFFECTED$(\tau_{21})=\{\tau_{21}\}$

Now if the task τ_{21} is in error and the error is EDM detected after the primary copy runs for 1 timeunit, then the task τ_{21} is schedulable.



Task τ_{21} (with deadline t=9) can now be scheduled because δ=0 at t=9 and is hence recovered.

For f=2, when scheduled using algorithm **RM-FT-ANY,** the task τ_{21} cannot be scheduled at iteration j=4.



Task τ_{21} (with deadline t=9) can not be scheduled because δ=5 at deadline t=9 and hence needs recovery.

AFFECTED(τ_{21})={ τ_{11} ,τ_{21}, τ_{12}}
τ_{11} is included in AFFECTED because it contributes to the δ=6 at t=6.
τ_{12} is included because it finishes before τ_{21}'s deadline and δ>0 for all t before its deadline, which also accounts δ at t=6.

We assume that a task with high execution time contributes to a high value of δ. Therefore, we choose to decrease the execution time of τ_{21}. If both primary copies of τ_{21} are affected by errors and if both of them are EDM detected (which is probabilistic), we can reduce the execution time of the two primary copies by 2 time units (further reduction will lead to zero execution time of the primary copies which is not possible). This reduction will lead to 2 time unit slack in the schedule before its deadline. However, we need at least 5 time units slack in the schedule before its deadline. So, the task set is not schedulable.

Next we will develop an algorithm RECOVERY-MIN-EDM(τ_{ij}) to determine whether a failed task in RM-FT-ANY can be scheduled with some modified execution time in another. We maintain a list EDM-LIST of modified task execution times to determine the maximum time within which the error should occur in order to be detected by EDM mechanism. We will select the task from the set AFFECTED(τ_{ij}).

In the algorithm in Figure 14, the first thing that is done is that the number of task copies in set AFFECTED-LIST that can be error detected by EDM is determined and stored in variable *TotalEDMError*. Next the amount of extra work for which the task τ_{ij} misses the deadline is calculated and stored in variable RED_TIME. Our objective is to decrease this extra work (RED_TIME) to zero. Then loop runs, for each iteration attempting to decrease the execution time of some task until RED_TIME becomes zero. Within the while loop, different task copies (a total of *f+2* copies are considered for each task) in AFFECTED-LIST with maximum execution time are selected first. This is because the extra work for which the lowest priority task in AFFECTED-LIST misses its deadline due to the execution of the tasks with larger execution time. We can select such tasks up to the minimum number

of errors (TotalEDMError) that could be detected by EDM. Each such selected task copy is stored in EDM-LIST. When no task from AFFECTED–LIST can be selected, the task from EDM_LIST with minimum execution time is selected, and its execution time is decremented by 1 time unit in each loop-iteration. If the execution time becomes below 1 time unit while decrementing the task's execution time cannot be further reduced, and the task copy is eliminated from EDM-LIST. If RED_TIME reaches to zero, then the task is recoverable. When the task is recoverable, MIN-EDM calculates the time up to which any task copy can run and can still meet the schedule.

ALGORITHM: RECOVERY-MIN-EDM(τ_{ij})

AFFECTED-LIST=AFFECTED(τ_{ij})
EDM-LIST={}
TotalEDMError=f * probability of error detected by EDM in task of AFFECTED-LIST.

If(TotalEDMError=0) NOT RECOVERABLE, STOP

RED_TIME= sub($\delta^{f}_{ij}(\Gamma)$, slack(fin(τ_{ij}), $T_i \cdot j$))

While(RED_TIME>0)

If(AFFECTED-LIST≠∅ and TotalEDMError>0){
EDM-LIST=EDM-LIST∪{τ_{mn}}, where τ_{mn} has the maximum running time in AFFECTED-LIST
Decrease TotalEDMError by 1.
If same task is selected (2+f) times, remove it from AFFECTED-LIST
If execution time of selected τ_{mn} is less than 2, then not RECOVERABLE
Decrease execution time of selected τ_{mn} by 1
MINEDM=Task with minimum execution time in EDM-LIST
Decrease RED_TIME by 1
If (RED_TIME==0) RECOVERABLE and STOP

}

Else If(EDM-LIST ≠∅)
{
Select τ_{mn} in EDM-LIST with minimum execution time
If execution time of selected τ_{mn} is less than 2 then remove the copy of τ_{mn} from EDM-LIST
And Continue
Decrease execution time of selected τ_{mn} by 1
MINEDM=Task with minimum execution time in EDM-LIST
Decrease RED_TIME by one
If (RED_TIME==0) RECOVERABLE and STOP
}

If(AFFECTED-LIST=∅ and EDM-LIST=∅)
NOT RECOVERABLE and STOP

End loop

Figure 14: Pseudocode for the recovery algorithm.

15.1 Correctness of the algorithm RECOVERY-MIN-EDM:

To recover a task τ_i, we are maintaining a list of affected task in AFFECTED-LIST where each task in AFFECTED set also contributes to the extra work that need to be done before the deadline. So, if the extra work to be done to finish a task in AFFECTED-LIST could be reduced, then the extra work to be done to finish τ_i before its deadline is also reduced. For example, if τ_j is the task that is in the AFFECTED-LIST whose execution time is reduced, then if τ_j has higher priority than τ_i, the reduction in execution time will leave slack after the τ_j execution but before the τ_i deadline. Thus τ_i can finish its execution before its deadline in the fault tolerant schedule. If the priority of τ_j is less than the priority of τ_i, the reduction of execution time in τ_j, will leave more slack before the τ_j execution time, since τ_j can be shifted right in the schedule and hence τ_i can finish execution before its deadline. So, the above algorithm provides a mechanism to recover task τ_i in case of faults. Remember the recovery is probabilistic because we are assuming that an error within the task in AFFECTED-LIST will be detected by the EDM mechanism.

16. Probability analysis of schedulability:

16.1: Parameters of probabilities:

The result of injecting faults into applications provides us the parameters of certain probabilities. Those probabilities are denoted by the following table. The values of these parameters are shown in APPENDIX [Result from injecting faults in a microprocessor 68340].

P_x	Given that a fault occurs, an error is generated.
P_{DE}	Given that an error is generated, the error is detected by double execution(DE)
P_T	Given that an error is generated, the error is detected by timer monitor.
P_{EDM}	Given that an error is generated, the error is detected by a hardware error detection mechanism(EDM)
P_{ND}	Given that an error is generated, the error is not detected.
$P_{DE,M}$	Given that an error is detected by DE, the error is masked by TEM
$P_{T,M}$	Given that an error is detected by the timer monitor, the error is masked by TEM
$P_{EDM,M}$	Given that an error is detected by an EDM, the error is masked by TEM

Table 12: Parameters of probabilities

16.2 Probability of Recovery (P_R):

Let S is the set of different tasks from AFFECTED-LIST whose execution times are modified by the **RECOVERY-MIN-EDM** algorithm.

The probability of an error detected and masked by EDM = $\sum_{\tau_{lk} \in S} \frac{C_l}{T_l} \cdot P_x \cdot P_{EDM}$

The probability of recovery of task τ_{ij}, denoted by $P_R(\tau_{ij})$, is:

$$P_R(\tau_{ij}) = \begin{cases} 1 & \text{if no recovery is needed} \\ \sum_{\tau_{lk} \in S} \frac{C_l}{T_l} \cdot P_x \cdot P_{EDM} & \text{otherwise} \end{cases}$$

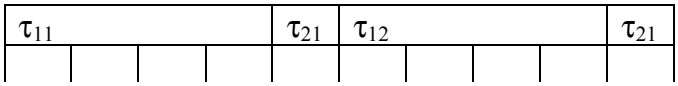
EXAMPLE:

	Period (T_i)	Execution Time (C_i)
τ_1	5	2
τ_2	10	1

set $\Gamma_{all}=<\tau_{11}, \tau_{12},\tau_{21}>$

For f=1:

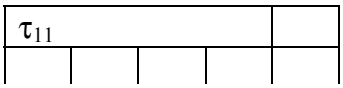
δ 0 0 0 0 2 2 2 2 2 2 2



1 2 3 4 5 6 7 8 9 10

j=1:

δ 0 0 0 0 2 1



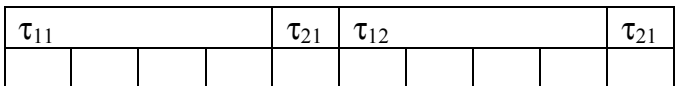
1 2 3 4 5

τ_{11} needs recovery, AFFECTED(τ_{11})={ τ_{11}}
τ_{11} can be recovered by reducing its execution time by 1 time unit as shown in Figure 15.

j=2: The task τ_{12} can be recovered in the same way.

j=3:

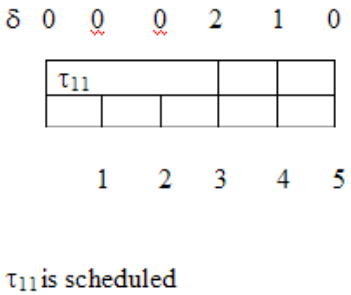
δ 0 0 0 0 2 2 2 2 2 2 2



1 2 3 4 5 6 7 8 9 10

τ_{21} needs recovery, AFFECTED(τ_{21})={ $\tau_{11},\tau_{12}, \tau_{21}$ }
τ_{21} can be recovered by reducing the execution times of task τ_{11} and τ_{12} by 1 time unit as shown in Figure16.

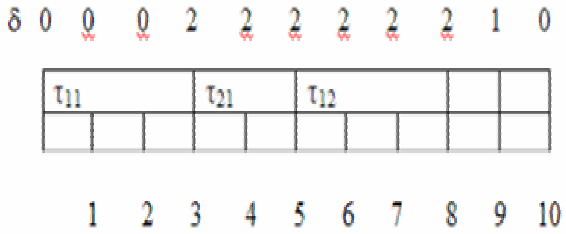
Iteration	AFFECTED-LIST	EDM-LIST	EDMError	RedTime	MinEDM	Comments
0	$\{\tau_{11}\}$	{}	1	1		Initialization before the loop starts
1	$\{\tau_{11}\}$	$\{\tau_{11}\}$	0	0	C_1-1	Among the *2+f* copies the execution time of the first copy is reduced and the task is recovered.



τ_{11} is scheduled

Figure15: Recovery of task τ_{11}

Iteration	AFFECTED-LIST	EDM-LIST	EDMError	RedTime	MinEDM	Comments
0	$\{\tau_{11}, \tau_{12}, \tau_{21}\}$	{}	2	2		Initialization before the loop starts
1	$\{\tau_{11}, \tau_{12}, \tau_{21}\}$	$\{\tau_{11}\}$	1	1	C_1-1	Among the 2+f copies, the execution time of the first copy of τ_{11} is reduced.
2	$\{\tau_{11}, \tau_{12}, \tau_{21}\}$	$\{\tau_{11}, \tau_{12}\}$	0	1	C_1-1	Among the 2+f copies, the execution time of the first copy of τ_{11} is reduced and task is reduced.



τ_{21} is scheduled

Figure16: Recovery of task τ_{21}

16.3 Probability of fault occurrence in task τ_{ij}:

For a maximum of *f* faults, the probability of a fault occurring in τ_{ij} is:

$P(U_{ij}) = [(f+2)*C_i/LCM] \cdot [LCM\text{-}Rel(\tau_{ij})/LCM]$

Here, *(f+2)*C_i/LCM* represents the percentage of work in case of faults occurring in τ_{ij} within one LCM. Again, the probability of a fault occurring is higher early in one LCM. Therefore, we scale the probability by [LCM-Rel(τ_{ij})/LCM].

16.4 Probability of schedulability for a task τ_{ij}:

$$Y_{ij} = \begin{cases} 0 & \text{if task } \tau_{ij} \text{ is not schedulable by RM-FT-ANY} \\ 1 & \text{schedulable} \end{cases}$$

16.5 Probability of error masking:

In this analysis, the probability of error detection and masking by any one of the techniques is $P_{DE} \cdot P_{DE,M} + P_T \cdot P_{T,M} + P_{EDM} \cdot P_{EDM,M}$

Denote the probability of fault occurrence, error detection and masking of all tasks by P_{Error}.

$$P_{Error} = \prod_{\tau_{ij} \in \Gamma_{all}} Y_{ij} \cdot \sum_{\tau_{ij} \in \Gamma_{all}} \left[P(U_{ij}) \cdot P_x \cdot (P_{DE} \cdot P_{DE,M} + P_T \cdot P_{T,M} + P_{EDM} \cdot P_{EDM,M}) \cdot P_R(\tau_{ij}) \right]$$

Using the number given in Appendix, we get:
$P_{DE} \cdot P_{DE,M} + P_T \cdot P_{T,M} + P_{EDM} \cdot P_{EDM,M} = 0.18 \cdot 1.0 + 0.05 \cdot 0.06 + 0.77 \cdot 0.68 = 0.7066$
$P_x \cdot P_{DE} \cdot P_{DE,M} + P_T \cdot P_{T,M} + P_{EDM} \cdot P_{EDM,M} = 0.17 \cdot 0.7066 = 0.120122$

16. 6 Probability of no error masking:

Let's denote the probability of fault free execution by $P_{noerror}$. This occurs when

- No faults occur
- Fault occurs but no error is generated.
- Fault occurs and error generated but error is not detected.

$$P_{noerror} = P_{NF} + \sum_{\tau_{ij} \in \Gamma_{all}} P(U_{ij}) \cdot (1\text{-}P_x) + \sum_{\tau_{ij} \in \Gamma_{all}} P(U_{ij}) \cdot P_x \cdot P_{ND}$$

$$P_{noerror} = 1 - P_x \cdot \left(1\text{-}P_{ND} \right) \cdot \sum_{\tau_{ij} \in \Gamma_{all}} P(U_{ij})$$

16.7 Probability of system success:

Let's denote the probability of the system success by $P_{success}$.

$P_{success}=P_{NoError}+P_{Error}$

EXAMPLE 1:

	Period (T_i)	Execution Time (C_i)
τ_1	9	1
τ_2	18	1
τ_3	36	1

set $\Gamma_{all}=<\tau_{11}, \tau_{12}, \tau_{13}, \tau_{14}, \tau_{21}, \tau_{22}, \tau_{31}>$

For f=0,

By running algorithm RM-FT-ANY, we can see that all the tasks are schedulable. No recovery is needed.

Hence, $\prod_{\tau_{ij} \in \Gamma_{all}} Y_{ij} = 1$

The probability of fault occurrence in all tasks:

$P(U_{11})= 2/36 =.055555$
$P(U_{12})=2/36* (27/36)=0.04166$
$P(U_{13})=2/36*(18/36)=0.027777$
$P(U_{14})=2/36*(9/36)=0.013888$
$P(U_{21})=2/36=0.05555$
$P(U_{22}) =2/36*18/36=0.027777$
$P(U_{31})=2/36=0.055555$

$$\sum_{\tau_{ij} \in \Gamma_{all}} P(U_{ij}) =0.055555*3+0.027777*2+0.041666+0.013888=0.277773$$

Since no recovery is needed for any τ_{ij} we have: $P_R(\tau_{ij})=1$

$P_{Error} =1 \cdot 0.120122 \cdot 0.277773=0.033366$
$P_{NoError}=1-0.17 \cdot (1-0.0) \cdot 0.277773=0.952778$
$P_{success}=0.033366+0.952778=0.986145$

For f=1:

By running algorithm RM-FT-ANY, we can see that all the tasks are schedulable. No recovery is needed.

Hence, $\prod_{\tau_{ij} \in \Gamma_{all}} Y_{ij} = 1$

The probability of fault occurrence in all tasks:

$P(U_{11})$= 3/36 =.08333
$P(U_{12})$=3/36* (27/36)=0.0625
$P(U_{13})$=3/36*(18/36)=0.04166
$P(U_{14})$=3/36*(9/36)=0.02083
$P(U_{21})$=3/36=0.08333
$P(U_{22})$ =3/36*18/36=0.04166
$P(U_{31})$=3/36=0.08333

$$\sum_{\tau_{ij} \in \Gamma_{all}} P(U_{ij}) = 0.083333*3+0.04166*2+0.02083+0.0625=0.41665$$

Since no recovery is needed for any τ_{ij} we have: $P_R(\tau_{ij})=1$

$P_{Error} = 1 \cdot 0.120122 \cdot .41665=0.05005$
$P_{NoError}$=1-0.17$\cdot$ (1-0.0) $\cdot$ 0.41665=0.92916
$P_{success}$=0.05005+0.92916=0.979219

For f=2:

By running algorithm RM-FT-ANY, we can see that all the tasks are schedulable. No recovery is needed.

$\prod_{\tau_{ij} \in \Gamma_{all}} Y_{ij} = 1$

The probability of fault occurrence in all tasks:

$P(U_{11})$= [4/36]=0.111111
$P(U_{12})$=4/36* (27/36)=0.083333
$P(U_{13})$=4/36*(18/36)=0.05555
$P(U_{14})$=4/36*(9/36)=0.02777
$P(U_{21})$=4/36=0.11111
$P(U_{22})$=4/36*18/36=0.083333
$P(U_{14})$=4/36=0.11111

$$\sum_{\tau_{ij} \in \Gamma_{all}} P(U_{ij}) = 0.11111*3+0.083333*2+0.02777+0.05555=0.58331$$

Since no recovery is needed for any τ_{ij} we have $P_R(\tau_{ij})=1$

$P_{Error} = 1 \cdot 0.120122 \cdot 0.58331 = 0.07006$
$P_{NoError} = 1\text{-}0.17 \cdot (1\text{-}0.0) \cdot 0.58331 = 0.90083$
$P_{success} = 0.90083 + 0.07006 = 0.97088$

For f=3:

All the tasks are schedulable using algorithm RM-FT-ANY. No recovery is needed.

$$\prod_{\tau_{ij} \in \Gamma_{all}} Y_{ij} = 1$$

$P(U_{11}) = = 5/36 = .138888$
$P(U_{12}) = 5/36 * (27/36) = 0.104166$
$P(U_{13}) = 5/36 * (18/36) = 0.069444$
$P(U_{14}) = 5/36 * (9/36) = 0.034722$
$P(U_{21}) = 5/36 = 0.138888$
$P(U_{21}) = 5/36 * 18/36 = 0.069444$
$P(U_{14}) = 5/36 = 0.138888$

$$\sum_{\tau_{ij} \in \Gamma_{all}} P(U_{ij}) = 0.13888*3 + 0.06944*2 + 0.034722 + 0.\ 0.104166 = 0.69444$$

Since no recovery is needed for any τ_{ij} we have $P_R(\tau_{ij}) = 1$

$P_{Error} = 1 \cdot 0.120122 \cdot 0.69444 = 0.083417$
$P_{NoError} = 1\text{-}0.17 \cdot (1\text{-}0.0) \cdot 0.694408 = 0.88195$
$P_{success} = 0.88195 + 0.08341 = 0.96536$

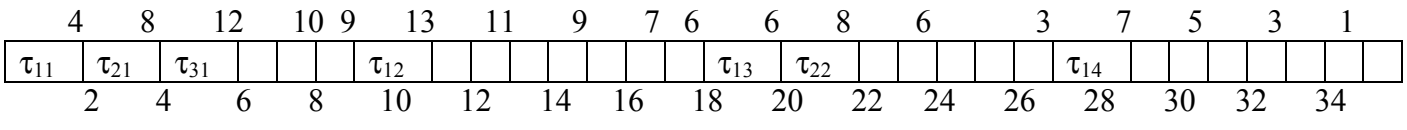
For f=4:
The task set is not schedulable using algorithm RM-FT-ANY.

To see why, we now simulate algorithm RM-FT-ANY.

At j=7:

$Q = \{\tau_{11}, \tau_{12}, \tau_{13}, \tau_{14}, \tau_{21}, \tau_{22}, \tau_{31}\}$

δ



In the schedule, the lowest priority task is $\tau_{31,}$ which is schedulable. However, task τ_{12} finishes at time t=11 and the value of δ=13 (amount of extra work due to faults). It is not possible to complete the extra task δ before the deadline (t=18), since $\delta(Q,t)\neq 0$ for any t such that $fin(\tau_{12})\leq t\leq T_1 \cdot 2=18$. Now if we try to recover by reducing the execution time of tasks in the set AFFECTED(τ_{12}), such recovery is not possible.

If we try to recover τ_{12}, then AFFECTED(τ_{12})={ $\tau_{11,}$ τ_{21}, τ_{13}, τ_{12}}

Since AFFECTED-LIST has all tasks whose execution time is already 1, no further reduction is possible according to the algorithm RECOVERY-MIN-EDM in Figure 14.

So, task set can not be scheduled. Hence, Y_{12} =0.
And

$$\prod_{\tau_{ij}\in \Gamma_{all}} Y_{ij} = 0$$

Hence, P_{error}=0

$P(U_{11})$= =6/36=.16666
$P(U_{12})$=6/36* (27/36)=0.125
$P(U_{13})$=6/36*(18/36)=0.083333
$P(U_{14})$=6/36*(9/36)=0.041666
$P(U_{21})$=6/36=0.16666
$P(U_{21})$=6/36*18/36=0.083333
$P(U_{14})$=6/36=0.16666

$$\sum_{\tau_{ij}\in \Gamma_{all}} P(U_{ij}) = 0.16666*3+0.083333*2+0.041666+0.\ 0.125=0.833312$$

$P_{noerror}$=1-0.17· (1-0.0) · 0.833312=0.858336
$P_{success}$=0+0.858336=0.858336

Intuitively, for this task set if f>3, $P_{success}$ =$P_{noerror}$.

16.7.1 Discussion (Example1):

Before providing more examples, we concentrate on the results from EXAMPLE 1. In the following four graphs, different parameters from EXAMPLE 1 is presented for various number of maximum faults *f*=1, 2, 3, and 4.

The graph in Figure 17 shows that, as the number of maximum fault occurrence increases, the probability of fault occurrence in the all tasks also increases.

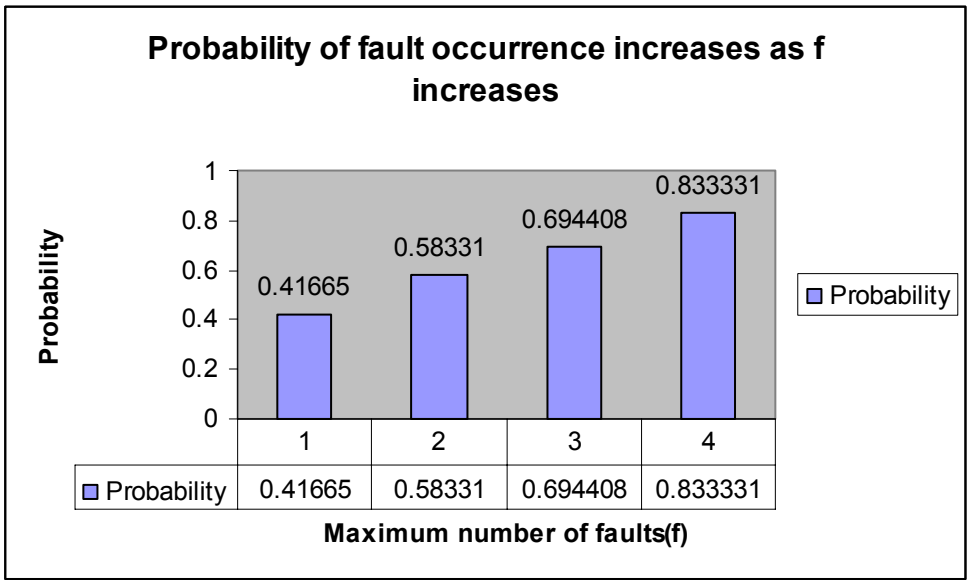


Figure 17: The probability of a fault increases as *f* increases.

In the graph in Figure18, it can be seen that the probability of masking faults increases as we increase the maximum number of fault occurrence. This is not surprising as the system employs error masking capabilities to mask errors. So, if the system is able to tolerate any number of faults, the probability of fault masking will increase as more faults will be masked since the probability of fault occurrence increases as the number of maximum fault occurrence also increases. However, in practice, no system is capable of tolerating an infinite number of faults. So, after a certain value of *f*, the fault masking capability will diminish to zero rather than decreasing slowly (see the column for f=4). Because, when the system is not capable of tolerating a particular number of maximum faults, the system becomes unschedulable and error masking capability is assumed to be zero from that point. From then on, for any higher value of *f*, the fault masking capability remains zero and the system schedulability depends only on the probability of system schedulability during fault free execution.

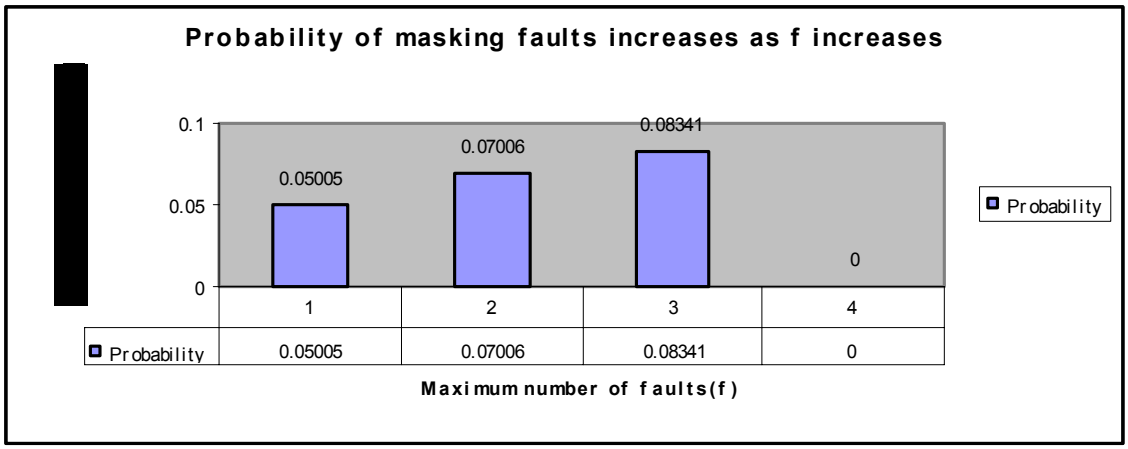


Figure 18: Probability of fault masking increases up to a certain value of f and then becomes zero.

Hence, the probability of error masking increases as the number of maximum faults increases to a certain point and then no error masking is possible since the task set is not schedulable.

In the graph in Figure 19, the probability of system success without fault masking capability is depicted. It can be seen that, the probability of success without fault masking ability decreases as the number of maximum fault occurrences, *f*, increases. This is because, as the errors occur, the probability of fault occurrence increases and the probability of system success without fault masking capability decreases since, without such capability, the system becomes unschedulable.

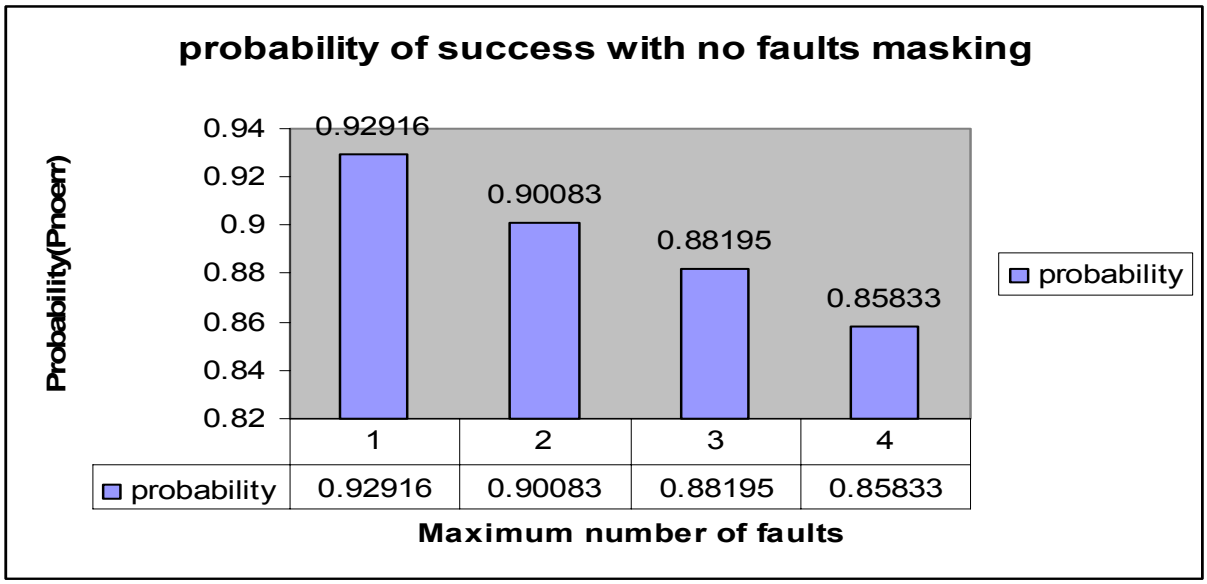


Figure 19: Probability of system success without fault masking decreases as *f* increases

As *f* increases, the probability decreases because the system becomes more vulnerable to faults and the probability of schedulability decreases so there is a decrease in system success without fault masking capability.

The graph in Figure 20 shows the probability of overall system success as the number of maximum faults increases. As the probability of maximum number of fault occurrence increases, there is a decrease in the probability of overall system success. As more errors are occurring, there is an increase in probability of fault masking (P_{error}) and there is also a decrease in the probability of success without fault masking ($P_{noerror}$). However, the sum of these two probabilities has a downward trend. This is because, as the number of maximum fault f occurrence increases, the probability of overall system success is more dependent on the fault masking capability of the system, which is limited for any practical system. So, for higher number of f, the probability $P_{success}$ is low.

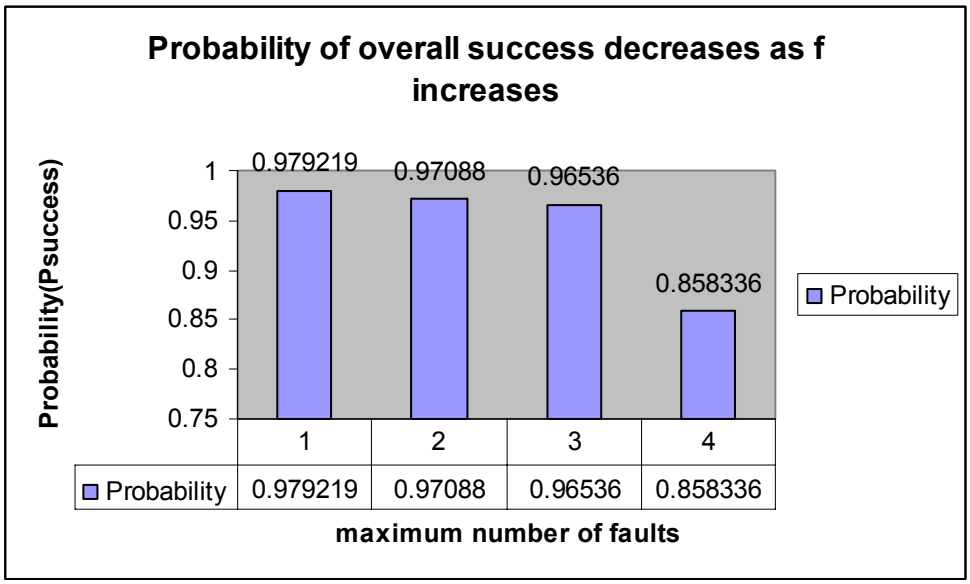


Figure 20: Probability of system success decreases as f increases.

For a higher number of errors, the probability of system success decreases. Observe that, for high value of f, the decrease in system success for two consecutive values of f (from f=1 to f=2) is higher than decrease for two consecutive low values of *f* (from *f*=3 to *f*=4). This is because, for a higher number of errors, the masking capability diminishes and the system success only depends on fault free execution of task which will further decrease as f increases as shown in Figure 20.

So, as the probability of fault occurrence increases, probability of fault masking increases and probability of success without fault masking decreases and probability of overall success decreases.

EXAMPLE 2: The execution time of task τ_1 is increased by one time unit that is given in Example 1.

	Period (T_i)	Execution Time (C_i)
τ_1	9	2
τ_2	18	1
τ_3	36	1

set $\Gamma_{all}=<\tau_{11}, \tau_{12}, \tau_{13}, \tau_{14}, \tau_{21}, \tau_{22}, \tau_{31}>$

For f=1:

All the tasks are schedulable. No recovery is needed.

$$\prod_{\tau_{ij} \in \Gamma_{all}} Y_{ij} = 1$$

The probability of error occurrence in all tasks:

$P(U_{11})=6/36=0.16666$
$P(U_{12})=6/36*27/36=.125$

$P(U_{13})=6/36*18/36=0.08333$
$P(U_{14})=6/36*9/36=.041666$
$P(U_{21})=3/36=0.08333$
$P(U_{22})=3/36*18/36=0.041666$
$P(U_{31})=3/36=0.08333$

$$\sum_{\tau_{ij}\in \Gamma_{all}} P(U_{ij}) = .16666+.125+3*0.08333+2*0.041666=0.624982$$

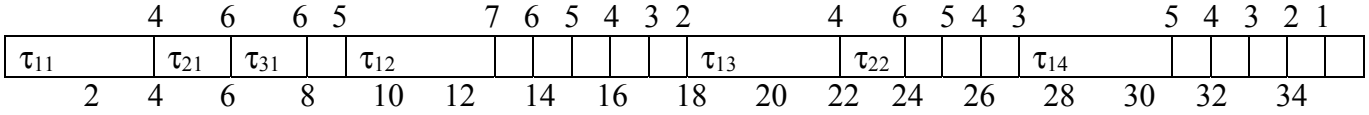
Since no recovery is needed for any τ_{ij} we have $P_R(\tau_{ij})=1$

$P_{Error} = 1 \cdot 0.120122 \cdot 0.624982=0.07507$
$P_{NoError}=1-0.17 \cdot (1-0.0) \cdot 0.624982=0.89375$
$P_{success}=0.89375+0.07507=0.96882$

For f=2:

All the tasks are schedulable with recovery for τ_{12}, τ_{13} which results in a minimum error detection time of 1 time unit.

When task τ_{12} is scheduled, the schedule with δ value is as follows:



AFFECTED(τ_{12})={ $\tau_{11},\tau_{21},\tau_{31},\tau_{12}$}

The recovery process is shown is Figure 21.

Here recovery is not possible.

$P(U_{11})=8/36=0.22222$
$P(U_{12})=8/36*27/36=.16666$
$P(U_{13})=8/36*18/36=0.11111$
$P(U_{14})=8/36*9/36=0.055555$
$P(U_{21})=4/36=0.11111$
$P(U_{22})=4/36*18/36=0.05555$
$P(U_{31})=4/36=0.11111$

$$\sum_{\tau_{ij}\in \Gamma_{all}} P(U_{ij}) = 0.22222+0.16666+3*0.11111+2*0.05555=0.83332$$

$P_{Error}=0$
$P_{noeeror}=1-0.17 \cdot (1-0.0) \cdot 0.83332=0.858335$
$P_{success}=0+0.858336=0.858335$

Intuitively, for this task set if f>3, $P_{success} = P_{noerror}$.

Iteration	AFFECTED-LIST	EDM-LIST	EDMError	RedTime	MinEDM	Comments
0	$\{\tau_{11}, \tau_{21}, \tau_{31}, \tau_{12}\}$	{}	2	2		Initialization before the loop starts
1	$\{\tau_{11}, \tau_{21}, \tau_{31}, \tau_{12}\}$	$\{\tau_{11}\}$	1	1	C_1-1 Of the first copy	Among the 2+f copies the execution time of the first copy is reduced
2	$\{\tau_{11}, \tau_{21}, \tau_{31}, \tau_{12}\}$	$\{\tau_{11}\}$	0	1	C_1 -1	No more reduction is possible for task τ_{11}.

τ_{12} can not be scheduled

MINEDM=C_1-1-1=1

Figure 21: recovery is not possible

16.7.2 Discussion (EXAMPLE 2):

Observe that, Example 2 has larger execution time for task τ_1 that the execution time given in Example 1. For *f*=1, the probability of $P_{success}$ in Example 2 ($P_{success}$=0.96882) is lower than the probability of $P_{success}$ in Example 1 ($P_{success}$=0.979219). This is because with increases execution time there is less possibility to have more slack in the schedule and task with higher execution time contributes to the increased value of δ. Similarly, for *f*=2, the probability of $P_{success}$ in Example 2 (Psuccess=0.858335) is lower than the probability of P success in Example 1 (Psuccess=0.97088).

As the execution times of the primary copies of any task increase, there is more slack required to execute the extra copies when error occurs. But increased execution time in primary copies decreases the amount of available slack in the schedule, hence reducing the probability of system success (that is the probability of schedulability).

Moreover, the recovery mechanism is activated for Example 2 when *f*=2, but for Example 1 the recovery mechanism is activated when *f*=4.

$P_{success}$=0.858335 can be achieved in Example 2 for *f*=2, where the same $P_{success}$=0.858336 can be obtained in Example 1 with *f*=4.

Consequently, it can be said that with higher task utilization, the less number of faults can be masked and thereby having a lower probability of system success.

EXAMPLE3:

	Period (T_i)	Execution Time (C_i)
τ_1	9	1
τ_2	18	1
τ_3	36	5

All the tasks are schedulable with recovery for $\tau_{13,}$ which results in minimum error detection at time 4 for task τ_{31}.

$P(U_{11})$=3/36=0.083333
$P(U_{12})$=3/36*27/36=0.0625
$P(U_{13})$=3/36*18/36=0.041666
$P(U_{14})$=3/36*9/36=0.020833
$P(U_{21})$=3/36=0.083333
$P(U_{22})$ =3/36*18/36=0.041666
$P(U_{31})$=15/36=0.416666

$$P_R(\tau_{13}) = \sum_{\tau_{lk} \in S} \frac{C_l}{T_l} \cdot P_x \cdot P_{EDM} \quad = 5/36 * 0.17 * 0.77 = 0.018$$

$$P_{Error} = \prod_{\tau_{ij} \in \Gamma_{all}} Y_{ij} \cdot \sum_{\tau_{ij} \in \Gamma_{all}} \left[P(U_{ij}) \cdot P_x \cdot (P_{DE} \cdot P_{DE,M} + P_T \cdot P_{T,M} + P_{EDM} \cdot P_{EDM,M}) \cdot P_R(\tau_{ij}) \right]$$

=1*0.120122* [0.083333*2+0.0625+0.04166*.018+0.020833+0.041666+0.41666]
=.120122*.709074=0.085175

$P_{noeeror}$=1-0.17· (1-0.0) · 0.709074=0.879457
$P_{success}$=0.085175+0.879457=0.964632

16.7.3 Discussion (EXAMPLE 3):

Observe that, Example 3 has larger execution time for task τ_3 that the execution time given in Example 2. For *f*=1, the probability of $P_{success}$ in Example 3 ($P_{success}$=0.964632) is lower than the probability of $P_{success}$ in Example 2 ($P_{success}$=0.96882). This is because, even if both have same utilization, tasks with large execution time runs for a long time in a fault tolerant schedule when recovery copies need to be run. Hence, less slack is provided in the schedule.

With the same utilization as in Example 2, Example 3 requires recovery when *f*=1. And the probability of success is $P_{success}$=0.964632.

This recovery mechanism in case of *f*=1 is needed, because the execution time of task τ_{13} is increased 5 time units which requires more slack in the schedule in case of any fault patterns. So, not only utilization but also the execution time of individual task is a major success factor.

17. Results:

It is natural that, in an environment where faults are more likely, the maximum number of faults *f* will have a higher value. As the maximum number of faults *f* increases, the analysis show that the probability of fault occurrence also increases. When error masking capability is introduced in system, the system is more robust and can tolerate more transient faults as the probability of fault occurrence of the system increases. However, no system can tolerate an arbitrary number of faults. It is shown in this thesis that, increased probability of fault occurrence results in increased probability of error masking but with a threshold value of *f*. When error masking is not introduced the probability of system success decreases since in that case system success is attributed only by the fault-free environment system success, not by the system success in environment where transient faults are likely. As the number of maximum faults *f* increases, the probability of system success in fault free environment decreases. Consequently, increased probability of fault occurrence results in decreased probability of system success without fault masking. Decrease in such probability is very sharp with high value of *f*.

As the number of maximum faults *f* increases, the probability of system success without fault masking capability decreases and the probability of system success with fault masking capability increases. This is because, as the number of maximum fault *f* occurrence increases, the probability of overall system success is more dependent on the fault masking capability of the system, which is limited for any practical system. So, for higher number of f, the probability $P_{success}$ is low. Since there is a threshold value for *f* up to which fault masking is effective, beyond that value of *f* overall system success decreases. This is because, for a higher number of errors, the masking capability diminishes and the system success only depends on fault free execution of task which will further decrease as f increases.

To run the recovery copies in case of faults more slack needs to be provided in the schedule. If the amount of slack is not sufficient to run the recovery copies in case of worst-case fault pattern, then the system may not be schedulable. If the task utilization is high, then providing appropriate amount of slack in the schedule may not be possible and the system may not be schedulable. So, probability of overall system success decreases with higher utilization of tasks.

It is interesting to observe that, task sets with same utilization may have different probability of success for the same maximum number of transient faults *f*. This is because, a task set with some task with very high execution time may need to run the recovery copies for a long time which may cause violation of deadline of some tasks with lower priority. Hence, task with higher primary execution time may cause violation of task deadline. So, the probability of overall system success decreases with higher primary execution time of tasks.

18. Conclusion:

Meeting task deadlines strictly is the main objective of hard real-time system. If faults are likely, mechanisms must be employed to tolerate the faults if the system has to avoid catastrophic consequences. Use of redundancy is the solution for achieving fault tolerance. In this thesis, due to space, weight and cost consideration building real-time fault-tolerant embedded system using time redundancy rather than hardware or software redundancy is addressed. Moreover, instead of system level fault tolerance, node-level fault tolerance is more cost effective since it avoids using redundant nodes to achieve fault tolerance and provides opportunity to use simpler protocols.

In this thesis, temporal error masking technique is used at node level to tolerate at most *f* transient faults. The number of recovery copies run in case of an error is *f*. Running *f* recovery copies requires more slack

in the schedule which may not be available for many task sets in hard real-time systems. If we decrease the number of recovery copies, that is *f*, all errors may not be possible to mask at node level. If error could not be masked at node level, system level fault tolerance has to be employed. However, by running less number of recovery copies more slack would be available in the schedule and more task sets could be schedulable which may not be possible if *f* recovery copies are run. Future work could be to find a trade-off between system level and node level fault tolerance, so that, some errors are tolerated at node level by running less than *f* number of recovery copies, thereby, providing more slack in the schedule and hence allowing more task sets to become schedulable, and errors that could not be masked at node level could be tolerated system level.

REFERENCES

[1] J. –C. Laprie, Dependability: Basic Concepts and Terminology, Vol. 5: Springer, 1992.

[2] A. Avizienis, "Design of Fault Tolerant Computers", AFIPS conference proceedings, 1967 Fall Joint Computer Conference, vol. 31, pp.733-743, 1967.

[3] D. Briere and P. Traverse, " AIRBUS A320/A330/A340 electrical flight controls- A family of fault-tolerant systems", FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing, 22-24 June 1993, Toulouse, France, 1993.

[4] L. Andrade and C. Tenning, "Design of Boeing 777 electric system", IEEE Aerospace and Electronics Systems Magazine, vol. 7, pp 4-11, 1992.

[5] Joakim Aidemark. Node-Level Fault Tolerance for Embedded Real-Time Systems. Ph. D. Thesis, Chalmers University of Technology, 2004.

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

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

[8] 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.

[9] R. K. Iyer, D. J. Rossetti and M. C. Hsueh. Measurement and Modelling of Computer Reliability as Affected by System Activity. ACM Trans. On Computer Systems, 4(3): 214-237, Aug. 1986.

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

[11] A. Damm, "The Effectiveness of Software Error-Detection Mechanisms in Real-time Operating Systems", FTCS Digest of Papers. 16th Annual International Symposium on Fault-Tolerant Computing Systems, Washington, DC, USA, 1986.

[12] D. M. Andrews. "Using Executable Assertions for Testing Fault Tolerance", 9th Annual International Symposium on Fault Tolerant Computing, 1979, New-York, USA.

[13] T. Lovric, "Dynamic Double Virtual Duplex System: A Cost Efficient Approach to Fault Tolerance", Dependable Computing for Critical Applications 5, IEEE Computer Society, 1998, pp. 57-74.

[14] M. A. Schuette, J. P. Shen, D. P. Siewiorek, and Y. X. Zhu, "Experimental evaluation of two concurrent error detection schemes", FTCS Digest of Papers. 16th Annual International Symposium on Fault-Tolerant Computing Systems, Washington, DC, USA, 1986.

[15] C. M. Krishna and Kang G. Shin, "Real-Time Systems", McGraw-Hill, 1997.

[16] L. Liestman, R. H. Campbell, "A fault-Tolerant Scheduling Problem," IEEE Trans. Software Eng., vol.12, no.11, pp.1089-1095, 1986.

[17] S. Ghosh, R. Mehlem, D. Mosse and J. S. Sarma. “Fault tolerant rate monotonic scheduling”. Journal of Real-Time Systems, 15(2), September 1998.

[18] Joakim Aidemark, Peter Folkesson, and Johan Karlsson. “ A framework for node level fault tolerance in distributed real time systems,” in *Proc. International Conference on Dependable Systems and Networks (DSN-2005)*, Yokohama, Japan, June 2005.

[19] Joakim Aidemark, J. Vinter, Peter Folkesson, and Johan Karlsson. “experimental evaluation of time redundant execution for a brake-by-wire application,” in *Proc. International Conference on Dependable Systems and Networks (DSN-2002)*, IEEE Computer Society Press, Washington DC, USA, June 2002, pp. 210-215.

[20] Joakim Aidemark, J. Vinter, Peter Folkesson, and Johan Karlsson. “Experimental dependability evaluation of the Artk68-FT real-time kernel,” in *Proc. of the International Conference on Real-Time and Embedded Computer Systems and Applications*, Göteborg, Sweden, August 2004, pp. 625-645.

[21] M. Pandya and M. Malek. Minimum Achievable Utilization for Fault-Tolerant Processing of Periodic tasks. Technical Report TR 94-07, University of Texas at Austin, Dept. of Computer Science, 1994.

[22] S. Ramos-Thuel. Enhancing Fault tolerance of Real-time systems through Time redundancy. Ph. D. Thesis, Carnegie Mellon University, May 1993.

[23] A. Burns, S. Punnekkat, L. Strigini, and D. R. Wright. “Probabilistic scheduling guarantees for fault tolerant real time systems”. Technical report. Department of Computer Science, University of York, 1998.

[24] 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.

[25] N. Kandasamy, J. P. Hayes, and B.T. Murray, “Tolerating Transient Faults in Statically Scheduled Safety Critical Embedded Systems”. Proc. 18th TEEE symposium Reliable Distributed System(SRDS), pp. 212-221, 1999.

[26] Sunondo Ghosh, Rami Melhem and Daniel Mosse, “Enhancing Real-time Schedules to Tolerate Transient Faults”. In Proc. of the 16th IEEE Real Time Systems Symposium, Pisa-Italy, 1995.

[27] G. M. de A. Lima, A. Burns, “An optimal fixed-priority assignment algorithm for supporting fault-tolerant hard real-time systems”, IEEE trans. On Computers, 52(10):1332-1346, Oct, 2003.

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

[29] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in Hard Real-Time Environment. Journal of ACM, 20(1): 40-61, 1973.

[30] Frank Liberato, Rami Melhem, and Daniel Mosse. “Tolerance to multiple faults for aperiodic tasks in hard real time system”. IEEE Trans. Computers 49(9):906-914, 2000.

[31]Lou Qian. “Real-time scheduling analysis of system employing TEM”. Master’s Thesis. Chalmers University of Technology, Sweden, May 2005.

APPENDIX-A

Results from injecting fault in 68340 microprocessor

P_x	Given that a fault occurs, an error is generated.	373 (of 2076)	17%
P_{DE}	Given that an error is generated, the error is detected by double execution(DE)	68 (of 373)	18%
P_T	Given that an error is generated, the error is detected by timer monitor.	19 (of 373)	5%
P_{EDM}	Given that an error is generated, the error is detected by a hardware error detection mechanism(EDM)	286 (of 373)	77%
P_{ND}	Given that an error is generated, the error is not detected.	0 (of 373)	0%
$P_{DE,M}$	Given that an error is detected by DE, the error is masked by TEM	68 (of 68)	100%
$P_{T,M}$	Given that an error is detected by the timer monitor, the error is masked by TEM*[1]	18 (of 286)	6%
$P_{EDM,M}$	Given that an error is detected by an EDM, the error is masked by TEM	194 (of 286)	68%

*[1] Assuming that an extra timer mechanism exists that detects if the execution time of tasks are violated.

APPENDIX-B

Theorem1: If $\delta^f(\Gamma,t-1)>0$ and $\delta^f(\Gamma,t)=0$, then, in both RM(Γ) and $RM^f(\Gamma)$, any task with release time less than t finishes before time *t*.

Proof:

Since $\delta^f(\Gamma,t-1)>0$ and $\delta^f(\Gamma,t)=0$ implies that $\delta^f(\Gamma,t-1)=1$.

$$\Psi^f(\Gamma,t)= \delta^f(\Gamma,t)+ \Psi(\Gamma,t)$$
Or, $\delta^f(\Gamma,t)= \Psi^f(\Gamma,t)- \Psi(\Gamma,t)$

Similarly, $\delta^f(\Gamma,t-1)= \Psi^f(\Gamma,t-1)- \Psi(\Gamma,t-1)=1$.
Since, by property 5, $\Psi^f(\Gamma,t-1)\geq \Psi(\Gamma,t-1)$, we have $\Psi(\Gamma,t-1)=0$ and $\Psi^f(\Gamma,t-1)=1$.
$\Psi(\Gamma,t-1)=0$ implies that all tasks released before t is finished by t in RM(Γ).

$\delta^f(\Gamma,t)= \Psi^f(\Gamma,t)- \Psi(\Gamma,t)=0$
$\Psi^f(\Gamma,t)= \Psi(\Gamma,t)$

Subtracting $\sum_{\tau_{ij}\in RD(\Gamma,t)} 2C_i$ from both sides and using the definition of Ψ, we have,

$\Psi^f(\Gamma,t) - \sum_{\tau_{ij}\in RD(\Gamma,t)} 2C_i = \Psi(\Gamma,t) - \sum_{\tau_{ij}\in RD(\Gamma,t)} 2C_i = sub(\Psi(\Gamma,t-1),1)$

Since $\Psi(\Gamma,t-1)=0$, we have $\Psi^f(\Gamma,t) - \sum_{\tau_{ij}\in RD(\Gamma,t)} 2C_i=0$. So, all task released before t is completed by t in $RM^f(\Gamma)$ by property 4.

Theorem2:

If $\delta^f(\Gamma,t-1)>0$ and RM(Γ ,t-1)=↑ then $\delta^f(\Gamma,t)=\delta^f(\Gamma,t-1)-1$.

Proof:

$\Psi(\Gamma,t-1)=0$ for RM(Γ ,t-1)=↑.
$\delta^f(\Gamma,t-1)= \mathbf{\Psi^f}(\Gamma,t-1)\mathbf{- \Psi}(\Gamma,t-1)= \mathbf{\Psi^f}(\Gamma,t-1)\mathbf{>0}$
$\delta^f(\Gamma,t)$ is the amount of extra work to be done at t. Since RM(Γ ,t-1)=↑, the extra work is equal to the amount of work at t-1 due to *f* faults decreased by one time slot.
$\delta^f(\Gamma,t)= \mathbf{\Psi^f}(\Gamma,t-1)\mathbf{-1}=\delta^f(\Gamma,t-1)-1$ since $\delta^f(\Gamma,t-1)= \mathbf{\Psi^f}(\Gamma,t-1)$**.**

Theorem3(adapted from [5]): Given task set Γ and, the lowest priority task τ_{ij} in Γ completes by $T_i \cdot j$ in $RM^f(\Gamma)$, if and only if, $\delta^f(\Gamma,t)=0$ for some t, $fin(\tau_{ij})\leq t\leq T_i \cdot j$.

Proof: To prove the if part, assume that t0 is the smallest value such that $fin(\tau_{ij})\leq t\leq T_i \cdot j$ and $\delta^f(\Gamma,t)=0$. If $\delta^f(\Gamma,t)=0$ for every t=0,….t0, then $RM^f(\Gamma)$ and $RM^f(\Gamma)$ are identical from t=0 to t=t0, which implies that τ_{ij} completes by $fin(\tau_{ij})\leq T_i \cdot j$ in both schedules. If, however, $\delta^f(\Gamma,t)>0$ for some $0\leq t<t0$, then t_1 be the least time before t0 such that $\delta^f(\Gamma,t)>0$. Note that $t_1< T_i \cdot j$(since t0 is the first value after $T_i \cdot j$ at which $\delta=0$) and that $\delta^f(\Gamma,t_1+1)=0$(by definition of t_1). Hence by Theorem 1, all tasks that are ready before t_1 finish

execution by t_1 in both RM(Γ) and $RM^f(\Gamma)$. Moreover, $\delta^f(\Gamma,t)=0$ for $t=t_1+1,\ldots,t_0$, which means that **$\Psi(\Gamma,t)=\Psi^f(\Gamma,t)$, and thus** RM($\Gamma$) is identical to $RM^f(\Gamma)$ in that period. But τ_{ij} completes in RM(Γ) at $fin(\tau_{ij})$, which means that it also completes by $fin(\tau_{ij})$ in $RM^f(\Gamma)$.

We prove the only if part by contradiction: assume that $\delta^f(\Gamma,t)>0$ for all $fin(\tau_{ij})\leq t\leq T_i\cdot j$ and yet τ_{ij} finishes in $RM^f(\Gamma)$ at t_1 for some $fin(\tau_{ij})\leq t_1\leq T_i\cdot j$. The fact that the lowest priority task, τ_{ij}, executes between time t_1-1 and t_1 means that no other task is available for execution at t_1-1, and thus **$\Psi^f(\Gamma,t_1-1)=1$. Given the assumption that** $\delta^f(\Gamma,t_1-1)>0$, by Property 5(in Section 13) implies that **$\Psi(\Gamma,t_1-1)=0$, which by Property 1 (in Section 13) implies that** $RM(\Gamma,t_1-1)=\uparrow$, and by the definition(in Section 13) of $\delta^f(\Gamma,t)$ leads to $\delta^f(\Gamma,t_1)=0$, which is a contradiction.

Corollary 1: A necessary and sufficient condition for the feasibility of $RM^f(\Gamma)$ for a given Γ and for given fault pattern with f or less faults can be obtained by applying Theorem 3 to N task sets Γ_j, j=1,..,N where Γ_j contains the j highest priority task in Γ_{all}.

Proof: Can be proved using induction on N.

The base case is trivial, when j=1, since there is only a single task. For the induction step, assume that $RM^f(\Gamma_j)$ is feasible and consider $\Gamma_{j+1}=\Gamma_j\cup\{\tau_{lk}\}$, where τ_{lk} has a lower priority than any task in Γ_j. IN $RM^f(\Gamma_{j+1})$, all tasks in Γ_j will finish at exactly the same time as in $RM^f(\Gamma_j)$, since τ_{lk} has the lowest priority. Hence, the necessary and sufficient condition for the feasibility of $RM^f(\Gamma_{j+1})$ is equivalent to the necessary and sufficient condition for the completion of τ_{lk} by D_l (the deadline of τ_{lk}).