

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Problem Statement</b>	<b>3</b>
<b>3</b>	<b>Real-Time Terminology</b>	<b>4</b>
3.1	Tasks . . . . .	4
3.2	Scheduling . . . . .	4
3.3	Processor-Demand Analysis . . . . .	5
<b>4</b>	<b>Fault-Tolerance Terminology</b>	<b>5</b>
4.1	Faults . . . . .	5
4.2	Fault Injection . . . . .	5
4.3	Error Detection . . . . .	6
4.4	Temporal Error Masking . . . . .	6
4.5	Node-Level Fault Tolerance . . . . .	7
<b>5</b>	<b>Related Work</b>	<b>7</b>
5.1	Strongly Related Work . . . . .	8
5.2	Less Related Work . . . . .	9
5.3	Directly Related Work . . . . .	9
5.4	Summary and Motivation . . . . .	10
<b>6</b>	<b>System Model</b>	<b>10</b>
6.1	Task Model . . . . .	10
6.2	Scheduling Model . . . . .	11
6.3	Fault Model . . . . .	11
<b>7</b>	<b>Schedulability Analysis</b>	<b>12</b>
7.1	Fault-Tolerant Processor Demand Analysis . . . . .	12
7.2	Deadline Assignment . . . . .	13
7.2.1	Assigning Minimum Feasible Deadline to an Aperiodic Task . . . . .	14
7.2.2	Assigning Minimum Feasible Deadline to a Recovery Task . . . . .	15
7.2.3	Example: Minimum Feasible Deadline Assignment . . . . .	17
7.3	Calculating the Probability of System Success . . . . .	19
7.3.1	Error-free Execution . . . . .	21
7.3.2	Error Detection by Double Execution or Timer Monitor . . . . .	22
7.3.3	Error Detection by Error Detection Mechanism . . . . .	23
7.3.4	Example: Maximum Execution Time for $C_{e,det}$ . . . . .	24
7.3.5	Probability of Overall System Success . . . . .	25
<b>8</b>	<b>Schedulability Analysis Examples</b>	<b>25</b>
8.1	Example 1: $\lambda = 0$ . . . . .	26
8.1.1	Case 1: Error-free Execution . . . . .	26
8.1.2	Case 2: Error Detection after Double Execution . . . . .	27
8.1.3	Case 3: Error Detection by Error Detection Mechanism . . . . .	28
8.1.4	Probability of Overall System Success . . . . .	30
8.2	Example 2: $\lambda = 0.5$ . . . . .	30
8.2.1	Case 1: Error-free Execution . . . . .	31
8.2.2	Case 2: Error Detection after Double Execution . . . . .	31

8.2.3	Case 3: Error Detection by Error Detection Mechanism . . . .	32
8.2.4	Probability of Overall System Success . . . . .	32
8.3	Example 3: $\lambda = 1$ . . . . .	33
8.4	Example 4: Comparison to RM scheduling . . . . .	33
8.4.1	Case 1: Error-free Execution . . . . .	33
8.4.2	Case 2: Error Detection after Double Execution . . . . .	34
8.4.3	Case 3: Error Detection by Error Detection Mechanism . . . .	35
8.4.4	Probability of Overall System Success . . . . .	35
8.5	Examples Summary and Discussion . . . . .	36
<b>9</b>	<b>Extended Schedulability Analysis</b>	<b>37</b>
<b>10</b>	<b>Conclusions</b>	<b>38</b>
<b>11</b>	<b>Future Work</b>	<b>38</b>
	<b>References</b>	<b>39</b>
<b>A</b>	<b>APPENDIX - Fault Injection Results</b>	<b>42</b>

# 1 Introduction

The use of computers in safety-critical real-time systems has increased rapidly over the last few years [10]. For well over a decade the technology has been used in commercial airplanes, and even longer in military aircrafts. In the automotive industry similar systems are will become increasingly used, for example to control the brakes in a car. These systems are called fly-by-wire respectively brake-by-wire systems [3]. If the brakes of a car would fail or the steering wheel of an airplane stop working, it is obvious that fatal situations could arise, possibly with fatal outcome. We therefore refer to the mentioned systems as *hard real-time systems*, in which at least some of the tasks are required to meet their deadline, or the system will fail.

A classical and revolutionary paper in the real-time field is Liu and Layland's article on scheduling algorithms [28] from 1973, in which they show that rate-monotonic (RM) scheduling is *optimal* among all static-priority uniprocessor schedulers, under the assumption that the deadline of tasks equals their period. In the same article it is also proved that, given the same assumption as above, earliest-deadline-first (EDF) scheduling is optimal among dynamic-priority uniprocessor algorithms. RM and EDF are two very frequently used scheduling algorithms in real-time systems of today. The two different approaches are explained more in detail later in this thesis.

Writing a program totally free from bugs is in practice impossible when it comes to large systems [18]. Further, components of a real-time system can be exposed to electromagnetic radiation in case it is operating in a harsh environment [26]. Because of this, tasks might miss their deadlines, making it obvious that introducing fault-tolerance to the scheduling algorithms is necessary. *Redundancy* is the base upon which fault-tolerance is built, and it can appropriately be divided into three classes: hardware redundancy, software redundancy, and time redundancy [13]. Time redundancy means that the scheduler in one way or another tries to manage enough free time, or slack, in the schedule to be able to re-execute a task in case of fault.

# 2 Problem Statement

The main problem in this thesis is as follows: **How can EDF scheduling be extended to tolerate transient faults by employing temporal error masking (TEM) to achieve node level fault tolerance?**

Previous research in this area has been done before for both RM [24, 34, 29] and EDF [9, 13, 15, 27] scheduling. However, many papers assume recovery tasks of higher priority or shorter execution times (see Section 5), whereas this thesis will not make those assumptions. Further, the benefit with employment of TEM is that tasks may be aborted before they have actually finished their execution, in case of a detected fault. This thesis will use this property in a **probability analysis** of system success. If a task finishes its execution before its deadline, more slack will be available, and thus probability of system success increases.

To answer the question stated above we need to find a way to efficiently analyze a given schedule. For RM scheduling this is appropriately done by using response time analysis to find out the response time of a task. Worst-case response time is then derived by calculating the response time at a critical instance, i.e. when all tasks are released at time 0. For EDF scheduling, however, analyzing the feasibility of a schedule is harder. A technique called *processor-demand analysis* can be used, but approaches using *response time analysis* for EDF also exist. [37] This work will use

processor-demand analysis. Further, we will need to consider occurrence of faults in our analysis which will make the problem even more complicated.

The rest of the thesis is organized as follows: Section 3 and 4 introduce terminology and terms used in the thesis. In Section 5, the reader finds a summary of recent work in the research area. Further, in Section 6, we formally specify the system model we will use to perform our analysis. Section 7 is the core of this thesis in which we present means to derive a probability of system success for a task set, given the specified model. This analysis is exemplified in Section 8. We briefly discuss methods to extend our analysis in Section 9, and finally, in Sections 10 and 11, we make conclusions based on the produced results, and discuss future possible extensions to the thesis.

## 3 Real-Time Terminology

This section briefly describes the real-time terminology and methods that are of importance to this thesis.

### 3.1 Tasks

A reader of this work should have a general knowledge of what a task is in a real-time system. We will however clarify different classifications of tasks.

A *critical* task is a task such that, if it misses a deadline, there will be catastrophic consequences [26]. Hard real-time systems deal with critical tasks. A *noncritical* task is not critical to the application; however, its result will be useless if the task does not finish execution before its deadline. *Periodic* tasks are very common in real-time systems. A periodic task is executed repetitively, for example a sensor checking the temperature every 10 ms. An *aperiodic* task is, in contrast, a task that occur occasionally.

### 3.2 Scheduling

Scheduling is the foundation of any real-time system. By using scheduling we provide means for tasks to share execution time on processors, and access to other resources. As explained in the introduction, there exist two commonly-used priority-driven schedulers nowadays, rate-monotonic (RM) and earliest-deadline-first scheduling (EDF), which were both addressed in [28] by Liu and Layland. RM is an on-line *static priority scheduler* where tasks with shorter periods gain higher priorities, and no modifications of the priorities are made on-line. EDF, on the other hand, is a *dynamic priority scheduler* where tasks are not assigned any priorities before run-time. Instead, the absolute deadline decides the next task to execute. Put in other words; that task with its deadline closest in time is chosen for execution.

An important part of the scheduling process of a hard real-time system is to be able to predict the schedulability of the given task set and scheduler. A schedule is said to be schedulable, or *feasible*, if it satisfies all the constraints imposed on it, i.e. all critical tasks produce correct output before their respective deadline. There are several ways to check feasibility for a schedule. A simple utilization-based approach can be applied for both RM and EDF given the assumption that task deadlines equal their periods. For more complex systems, *response-time analysis* [23] and *processor-demand analysis* [8] can be used. Response-time analysis is applied on static-priority schedulers, by calculating the worst-case completion time for every task. Processor-demand analysis

can be used on EDF schedulers, and as the name implies, the technique analyzes the demand of processing capacity made by different tasks.

The *planning cycle* of any static or dynamic schedule is defined as the *least common multiple* (LCM) of all scheduled tasks periods. This cycle is iteratively repeated for the duration of the lifetime of the system, and hence we need only to consider the planning cycle when analyzing a system, without loss of generality.

### 3.3 Processor-Demand Analysis

Processor-demand analysis was presented by Baruah et al. in [8] in order to study the feasibility of a schedule created by EDF. The technique calculates, at certain control points (all absolute deadlines) in a planning cycle, the amount of processor time demanded by the task set up to that point. If the demanded processor time at every control point is less than the available processor time at the point, the schedule is feasible.

A given task set  $\mathcal{T} = \{\tau_1, \dots, \tau_n\}$  is schedulable if the following equation holds:

$$\forall L : \sum_{i=1}^n \left( \left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) \times C_i \leq L$$

where  $L$  is a given control point and  $C_i$ ,  $D_i$ , and  $T_i$  are the execution time, deadline, and period, respectively, of each task.

## 4 Fault-Tolerance Terminology

In the following section we will briefly describe fault-tolerant terminology and methods that are of importance to this thesis.

### 4.1 Faults

First of all, when talking about fault-tolerance we distinguish between *faults*, *errors*, and *failures* [26]. A fault is either a software bug or some kind of flaw in the hardware, such as a broken wire. An error is a manifestation of a fault; for example, an error will occur when a system tries to propagate a signal through a broken wire. Finally, a failure has occurred if the system fails to provide what is specified. In this work we will mostly address faults.

We can further divide faults into three types: *permanent*, *intermittent*, and *transient*. A permanent fault does not disappear with time, but remains until repaired. An intermittent fault is pending between being harmful to the system and not, which could be the case of a loose wire that sometimes work, and sometimes does not work. Transient faults die away with time and could for example be a software bug which assigns an illegal value to a variable, which is later overwritten. The transient fault might either harm the system, or it might pass unnoticed.

### 4.2 Fault Injection

Fault injection is of interest when it comes to testing fault-tolerant mechanisms. In this thesis in particular, results from fault injection will be used to derive probabilities of system success, and is therefore worth mentioning.

Fault-injection techniques can be divided into three classes: *simulation-based fault injection*, *physical fault injection*, and *software-implemented fault injection* (SWIFI) [2]. In simulation-based fault injection, faults are injected into a simulated system, while in the latter two the faults are injected into the actual physical system. In physical fault injection faults are introduced by, for example, irradiating circuits with heavy ions. On the other hand SWIFI introduces faults by using software, like modifying the contents of CPU registers.

In [5], Aidemark et al. used SWIFI to inject faults into a 32-bit Motorola 68340 micro-controller, where they modeled transient faults as single bit-flips. The experimental results from that paper will be used in this thesis in order to derive probabilities of system success.

### 4.3 Error Detection

Detection of errors can be performed either in hardware or software [2], or a combination of both.

Hardware-implemented error detection can be implemented by having two processors executing the same task. The processors might be strictly synchronized, or loosely coupled, that is, exchanging output via a channel. In this way, output can be compared and errors detected when the outputs differ. A less costly variant to this is a *watchdog* processor that monitors the flow or checks if the outputs are reasonable.

Software-implemented error detection might be implemented in many ways. In *information redundancy* variables are duplicated and their contents compared to detect errors. *Executable assertions* is similar to the watchdog technique, but implemented in the software. Given the specifications of the program, reasonability checks of variables are performed. *Structural checks* are checking the validity of, for example, lists and arrays while *timing checks* control that tasks do not execute past their deadlines. Finally, *control-flow checks* detect errors by analyzing the control flow of a program. If an illegal sequence of actions are detected, an error has occurred.

### 4.4 Temporal Error Masking

One of many proposed methods of recovering from faults is temporal error masking (TEM) proposed by Aidemark et. al [4]. TEM executes all critical tasks twice, in order increase probability of system success. If the two copies produce the same output, the execution of the task is assumed to be correct and the scheduler can schedule another task. On the other hand, if the two copies do not produce the same output, due to a fault, a third copy (hereby referred to as *recovery copy*) is executed and a majority voter is applied on all three of the tasks. The majority voter simply compares the outputs and delivers a result if two of the tasks outputs match, otherwise an omission failure occurs. TEM also applies a timer monitor that detects if a task violates its deadline, and in that case execution is aborted and considered faulty. Further, an error can be detected by one of the error detection mechanisms from the previous section, during the execution of either the first or second copy of the task. In this case execution stopped right away, and a new copy of the task is started. The two tasks that finished their executions are then compared and a result is delivered if the outputs match. Otherwise, like above, an omission failure occurs.

Summed up, we can divide scheduling under TEM into three cases (see Figure 1). No fault leads to fault-free execution. In case of a fault, it can be detected either by

comparison after double execution, by a timer monitor if violating a deadline, or by another error detection mechanism during the execution.

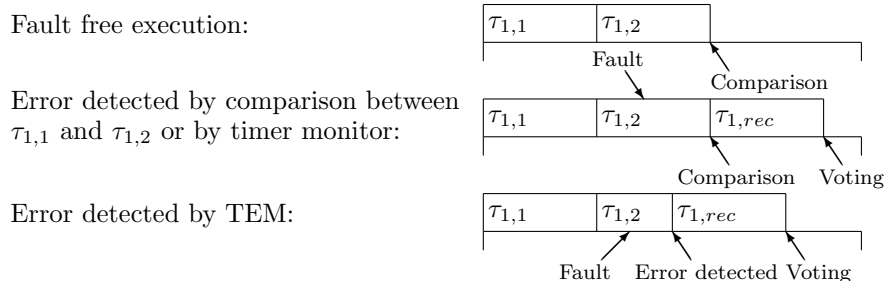


Figure 1: Temporal Error Masking

Pathan extended in [33] TEM to allow multiple faults. In this thesis we will, however, only consider the single fault case as described above.

## 4.5 Node-Level Fault Tolerance

In [3] Aidemark et al. introduced the term node-level fault tolerance (NLFT). More precisely, a framework for *masking errors* in the nodes of a distributed system was proposed, simulated and analyzed.

Masking errors at node-level is proposed as a complement to system-level fault-tolerance in order to improve the dependability of the system. If we can achieve NLFT we do not need to consider the faults at the system-level, and thus we have achieved a transparency level where, to a certain extent, a programmer does not need to consider faults occurring in the nodes. To tolerate all types of faults by NLFT would however be very costly, therefore the authors propose an approach called light-weight NLFT. In the light-weighted approach only transient faults are masked, which is appropriate since they are much more frequent than permanent ones [3, 6, 9, 12].

In order to achieve NLFT, some kind of mechanism has to be used, and in [3] the previously discussed TEM technique is applied. TEM is explained above as a method to tolerate faults, and thus it can be directly applied to tolerate transient faults in light-weight NLFT.

## 5 Related Work

As previously mentioned, redundancy is the base upon which fault-tolerance is built. This thesis focus on the *time redundancy*, and thus most of the related articles and papers are based on this class of redundancy. However, other types of redundancy is of relevance, and will be taken into account, for example hardware redundancy through task replication in multi-processor systems.

By providing time redundancy we can allow tasks to re-execute while still keeping a feasible schedule. Different approaches for task *re-execution* has been proposed, such as the *recovery block scheme* [6, 7, 11, 12, 25, 27, 31], the *deadline mechanism*

[14, 19], and *imprecise computation* [13, 31]. These three concepts are heavily inspired by each other and the main idea for all three of them is to provide less complex back-up versions (*alternates* and *primaries*) for all critical tasks [14, 19, 25]. Due to the reduced complexity of the back-up tasks they will have shorter execution times, still producing acceptable results, making it more probable for the system to sustain feasibility in case of failures. Further, many fault-tolerant theories and algorithms apply straightforward re-execution of tasks, that is re-execution without decreased complexity.

In the following sections, we will describe those methods more in detail.

## 5.1 Strongly Related Work

Liberato et al. studied a recovery block based scheme that guarantees the timely recovery from multiple transient faults in uniprocessor systems [27]. Assuming aperiodic tasks, earliest deadline first scheduling, and  $k$  faults, a necessary and sufficient feasibility-check algorithm was proposed. Under the same assumptions Aydin further improved the analysis by efficiently solving the case where the recovery blocks do not necessarily have the same execution time as the corresponding primary tasks [6]. Moreover he proposed an on-line version of the algorithm, where no a priori knowledge of the release times are known, as well as how to extend the framework to periodic tasks. Burns et al. [11] presented another exact feasibility test making no assumptions on the scheduler used, contrary to the two recently discussed feasibility tests. A discussion on how to use *check-pointing*, that is rolling back execution to a certain point in a task, is also performed. However, the fault model is not as refined as in [6].

Numerous more fault-tolerant scheduling algorithms have been proposed, implemented and evaluated for both dynamic and static priorities. [9, 13, 15] all extend the EDF scheduler to include fault-tolerance in uniprocessor systems, and evaluate the respective algorithm. [9] and [13] assume periodic tasks and transient faults. In [9], Beitollahi et al. presented a scheme to determine an appropriate and efficient time redundancy to tolerate transient faults under EDF scheduling through re-execution. In contrast, Chen et al. [13] use imprecise computation and *resource reclaim* to provide fault-tolerant EDF scheduling. They exploit the fact that the *worst-case execution time* in practice is larger than the actual execution time, meaning that another task can reclaim the resource and start execution directly when the recourse, i.e. the processor, is released. They also present a way to give up some task instances to provide time redundancy. Both algorithms study the percentage of lost critical tasks to evaluate the performance of their respective algorithm. In the first one, no comparison between other fault-tolerant EDF schedulers are made; instead, potentially good values for the *average task utilization* and *mean time to failure* are determined. In the latter several variables, such as the utilization and mean time to failure, are adjusted in order to be able to study the lost ratio. Contreras and Sourrouille [15] discuss basic ideas on how to schedule tasks under re-execution in soft real-time systems, while Kandasamy et al. [24] evaluated a way to tolerate transient faults in embedded multiprocessor systems.

Another interesting approach using resource reclaiming as mentioned above is made in [30] by Manimaran and Murthy. They propose a dynamic multiprocessor fault-tolerant algorithm using *task replication* on different processors. They also introduce methods for the scheduler to look ahead in the schedule. Hong and Goo [22] presented another distributed scheme for scheduling of *hybrid task sets*, that is, task sets containing both periodic and aperiodic tasks. They use imprecise computation and task

replication to achieve fault-tolerance and perform experiments to show the effectiveness of the scheme. Further, Ghosh et al. [16] discussed how to guarantee toleration of transient faults by carefully providing and manipulating idle slots in dynamic or static schedules. The work was further refined in [17] where an extension to the rate monotonic scheduler was proposed. This proposal was however later proved to be incorrect in [36] (there exist test cases under which the scheduler actually failed) and *formal methods* were then used to explain how to extend RM scheduling to include fault-tolerance.

Han et al. [19] introduced an algorithm that was later called CAT/EIT (Checking Available Time/Eliminating Idle Time) [25]. The algorithm is based on the deadline mechanism by using the *last chance philosophy* to schedule the alternates as late as possible. This maximizes the chance for primaries to finish execution on time. Further, EIT is eliminating idle time by choosing an alternative to execute in case the processor is about to become idle. Khan and Sydholm based their Alternate-Primary Recovery (APR) algorithm [25] directly on the CAT/EIT algorithm. The innovating idea in APR is the *backup-primary* that will replace the primary if it fails permanently. The authors then compare APR to CAT/EIT by counting the number of successfully primaries in the system.

[7, 10, 12, 32] are all very theoretical papers. [7] and [10] concentrate on multiprocessor systems where [7] is providing methods for off-line determination of how much a task can increase its execution time with sustained feasibility. The work in [10] on the other hand, proposes means to determine how many processors that are needed to keep a schedule feasible under a certain task set. In [12], Burns et al. consider a uniprocessor system and derive probabilities that faults will not occur at a quicker rate than a certain maximum bound. Pandya and Malek [32] assumed rate monotonic uniprocessor scheduling and proved that if the utilization of a task set is restricted to 0.5 the scheme is feasible, even in case of a single fault.

## 5.2 Less Related Work

Less important to this thesis, but still worth mentioning, is the work in [1, 20, 21, 31, 35]. In [1], a framework is presented that couples dynamic job scheduling with scheduling of active and passive replications. [20] extends the response time analysis techniques to analyze applications scheduled under EDF running on top of a fixed priority scheduler. Hoang et al. [21] proposed a way to calculate the shortest deadline of an arbitrary task, or a new incoming task, while still preserving feasibility. Finally, [31] outlined how fault-tolerant techniques like the recovery block and imprecise communications can be implemented, and in [35] an efficient checkpoint based recovery scheme is presented.

## 5.3 Directly Related Work

Lou addressed in [29] analysis of real-time systems employing TEM to tolerate one single transient fault. The term TEM was described in a previous section as a way to allow node level fault-tolerance (NLFT). Lou assumed RM static scheduling of periodic tasks and presented a way to calculate the *probability of system success*, that is all critical tasks produce correct output before their deadlines. By applying experimental fault-injection results to the proposed formulas, Lou was able to calculate the probability of success for any given task set.

Pathan [33] extended the work of Lou to tolerate multiple transient faults. In

addition, Pathan proposed an algorithm (RM-FT-ANY) for determining schedulability under all fault patterns; that is, since multiple faults are considered we must take all possible pattern of faults into account. Pathan also presented an algorithm (RECOVERY-MIN-EDM) that checks if a tasks execution time can be decreased to allow another task that misses its deadline to finish before its deadline.

In [34] Pathan slightly clarified RM-FT-ANY and changed the name of the algorithm to RM-FT. The algorithm itself was unmodified, rather the pseudo-code and the description of the algorithm was rewritten.

## 5.4 Summary and Motivation

As this related work section reveals, there exist several papers analyzing fault-tolerant real-time systems from many different angles of approach. Fault-tolerant algorithms extending both RM and EDF scheduling have been proposed and evaluated. As previously stated, [9, 13, 15] describe techniques on how to extend a uniprocessor EDF scheduler to provide fault-tolerance, which is also the main goal with this thesis. However, where these papers measure the performance of the algorithm by, for example, counting the percentage of faulty tasks [9, 13] or simply the number of occurred faults [15], this work will focus on **deriving a probability** that a given task set is feasible under certain assumptions. To the best of this author’s knowledge, this approach have not been taken before. [10] tackles EDF scheduling problem under a probabilistic approach, though the paper focus on multiprocessor EDF scheduling and concentrate more on finding an appropriate number of processors to use.

Other performance metrics in the mentioned papers include summing up the time spent on executing unsuccessful tasks [14, 19, 25], the percentage of successfully executed primaries [14, 25] and acceptance of aperiodic tasks [22].

Moreover, many articles [14, 25, 22] simply state that faults occur and provide means to cope with them. No specifics on how to detect the faults are done, even on a very theoretical level, whereas we will propose a relatively **detailed description** of how to achieve NLFT by using TEM. Further, we will also use results from experimental fault injections [5] to derive exact probabilities of system success.

The main contribution in this thesis is thus to **convert the work in [29]** from static scheduling to dynamic EDF scheduling, and to perform an extensive probability analysis similar to what Pathan did in [33]. **More precisely, the aim of this thesis is to derive a probability of system success for a system employing TEM to mask transient faults, given a restricted system model.**

## 6 System Model

In the following section we describe the task and fault models to be used in our analysis, as well as important restrictions of the system.

### 6.1 Task Model

We consider a *uniprocessor system* with a set of periodic tasks  $\mathcal{T} = \{\tau_1, \dots, \tau_n\}$ . The period and relative deadline are denoted  $T_i$  and  $D_i$ , respectively, and they are considered to be equal,  $T_i = D_i$  (this does not apply to the recovery task, see next section). The worst-case execution time is denoted  $C_i$ , while  $t_i$  is the absolute deadline. Tasks are assumed to be independent and the first instance of all tasks are released at the same time. Without loss of generality, we can assume that this occurs at time 0.

## 6.2 Scheduling Model

*Preemptive EDF scheduling* is assumed to be used to schedule the tasks. The scheduler employs TEM, thus all critical tasks are executed twice. The two primary executions of a task have a common deadline, and if one of them misses the deadline the schedule is not feasible. In case of a detected error, a third copy (recovery copy) of the task is executed, which may be assigned an arbitrary deadline. The recovery task is released immediately when the error is detected, at time  $ed_i$  if an error occurred in  $\tau_i$ . We define the set  $\mathcal{D}$  as all absolute deadlines for task set  $\mathcal{T}$  when scheduled under EDF. We also assume  $U_{FF} = \sum_{i=1}^n \frac{2 \times C_i}{T_i} < 1$ , that is the utilization of the processor, excluding the recovery task, is less than 100%.

The length of the planning cycle, as previously described, is the least common multiple of the all the tasks periods:

$$PC = LCM\{T_1, \dots, T_n\}$$

We can analyze any planning cycle without loss of generality, since they are all identical. We choose to analyze the first cycle. We introduce the denotation  $\tau_{ij}$ , where  $j$  is the  $j^{th}$  invocation of task  $\tau_i$ . Let  $\mathcal{T}_{i,PC}$  be all instances of  $i$  in the interval  $0 < t \leq PC$ ,  $\{\tau_{i1}, \dots, \tau_{in}\}$ . Further, let  $\tau_{i,rec}$  be the recovery task for  $\tau_i$ .  $\tau_{ij,rec}$  is denoting the recovery task for the  $j^{th}$  invocation of task  $\tau_i$ .

## 6.3 Fault Model

We consider only transient faults, and that **exactly one transient fault occurs in each planning cycle** that may lead to **exactly one error**. The location of the fault is uniformly distributed. Under these assumptions we can derive the probability that a fault will occur during execution of  $\tau_i$  [29]:

$$P(U_i) = \frac{2 \times C_i}{T_i} = \frac{2 \times C_i}{D_i}$$

Explained in words, the probability of fault occurrence in one of the primary executions of task  $\tau_i$  is its utilization. This is obvious since there is exactly one fault occurrence per planning cycle with the same probability to appear anywhere in the time span. The more time a task spends executing, the bigger the chance is that it will be exposed to an error. For example, if a task executes 75% of the time in every LCM, the probability that the error will occur in this task is 75%.

Similarly, the probability that a fault will occur in a given instance of a task,  $\tau_{ij}$ , is:

$$P(U_{ij}) = \frac{2 \times C_i}{PC}$$

Given the fact that we will use experimental results from fault injection into a real-time kernel [5], we need to assume the same error detection mechanisms in this thesis. The kernel is implemented in the programming language Ada95, and thus software-implemented error detection mechanisms is provided by the language, while the 68340 microcontroller provides us with hardware-implemented error detection mechanisms. See Table 1 for a detailed description of these mechanisms.

Given the error detection mechanisms provided, Aidemark et al. performed fault injection into the 68340 microcontroller in order to evaluate TEM [5]. From their results we derive a set of probability parameters, which we later on will use to derive probabilities of system success. These parameters are presented in Table 2.

Selection of Ada run-time constraint checks	
Ada access check	Attempt to follow a null pointer
Ada range check	Attempt to violate a range constraint of scalar value
Ada index check	Attempt to access an index that is not in the range of the array
Selection of motorola 68340 microcontroller hardware checks	
Bus error	Attempt to access non-existent memory
Address error	Attempt to access a word or a long-word on an odd memory address
Illegal instruction	Attempt to execute a non-existing instruction
Line1010	Attempt to execute an unimplemented instruction
Division by zero	Raised if a division instruction is given a divisor value of 0

Table 1: Error detection mechanisms

$P_x$	Given that a fault occurs, an error is generated
$P_{DE}$	Given that an error is generated, the error is detected by comparison after double execution (DE)
$P_T$	Given that an error is generated, the error is detected by a timer monitor (TM)
$P_{EDM}$	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 TM, the error is masked by TEM
$P_{EDM,M}$	Given that an error is detected by EDM, the error is masked by TEM

Table 2: Probability parameters

## 7 Schedulability Analysis

The following section is the core of this thesis. We derive a formula for estimation of system success for a task set, given our system model. We also investigate the possibility to **delay the deadline** of a recovery task to increase this probability even further.

### 7.1 Fault-Tolerant Processor Demand Analysis

As previously mentioned, we will in this thesis apply processor-demand analysis in order to analyze a given schedule. According to the processor-demand analysis, originally presented in [8], a given task set  $\mathcal{T} = \{\tau_1, \dots, \tau_n\}$  is feasible if

$$\forall L : \sum_{i=1}^n \left( \left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) \times C_i \leq L$$

To allow for fault-tolerance, we need to alter the formula slightly. First of all, since we have assumed that the deadlines of task are equal to their periods,  $D_i = T_i$ , we can replace all occurrences of  $T_i$  with  $D_i$ . Moreover, since we use TEM, every task executes twice; hence we need to replace  $C_i$  with  $2 \times C_i$ . Finally, we need to introduce the possibility of execution of a recovery copy of a task. This will happen if an error is detected or there is a task time-out, as explained in the section about TEM. In conclusion, we get:

$$\forall L : \sum_{i=1}^n \left( \left\lfloor \frac{L - D_i}{D_i} \right\rfloor + 1 \right) \times 2C_i + \delta(L - d_{e,rec}) \times C_e \leq L \quad (1)$$

where  $L$  is a given control point,  $\tau_e$  is the task in which the fault has occurred,  $d_{e,rec}$  is the absolute deadline of the recovery task, and

$$\delta(L - d_{e,rec}) = \begin{cases} 0 & \text{if } L < d_{e,rec} \\ 1 & \text{if } L \geq d_{e,rec} \end{cases}$$

Explained in words, we have modified the processor demand analysis to use TEM to mask one error per planning cycle. The introduction of  $\delta(L - d_{e,rec}) \times C_e$  means that we only take the execution time of the recovery copy of a task in account if its deadline is in the current window, that is, its deadline occurs earlier than or at time  $L$ .

## 7.2 Deadline Assignment

A key element in the modified processor-demand analysis presented in Equation (1) is the possibility to assign different deadlines to the recovery copy of a faulty task. Consider a hard real-time system, in which all critical tasks must meet their deadlines in order to sustain feasibility. In such a system, the absolute deadline of a recovery copy must be equal to that of the two primary executions of the task. However, suppose we have a more relaxed system, where recovery copies can have delayed deadlines while still sustaining feasibility. In this case, delaying the deadline may increase probability of system success.

For example, consider the following task set:

Task	$C_i$	$D_i$
$\tau_1$	1	9
$\tau_2$	3	10

Table 3: Task set

Scheduled under EDF employing TEM it generates the schedule in Figure 2.

Because of a detected error  $\tau_2$  finishes execution at  $t=11$ , and consequently misses its deadline at  $t=10$ . However, if we could delay the deadline of the recovery copy,  $\tau_{2,rec}$ , with one time unit, we would make sure the deadline is kept. Further, it is not hard to see that subsequent invocations of both tasks will all meet their deadlines, given the fact that our model only assumes one fault per planning cycle. Thus, we have demonstrated that **by altering the deadline of a recovery copy, we can increase probability of system success.**

Determining an appropriate deadline value to assign to a recovery copy can be done in several ways. The deadline could be extended with some pre-determined

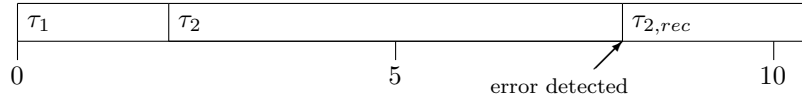


Figure 2:  $\tau_2$  misses its deadline at  $t=10$  if unmodified

value, for example the execution time of the task to be recovered. It could also be relative to the time left in the planning cycle. These two solutions increase probability of system success, as explained above. They are, however, not optimal. If a deadline is extended with a too small value, the recovery copy will miss its deadline, and hence the schedule is unfeasible. On the other hand, extending the deadline with a too large value can be risky as well. The system may be strictly time critical, for example an aircraft, in which delaying deadlines of recovery tasks may have the same effect as a deadline miss. Even if the task meets its deadline it still produces a worthless result, and consequently, occupy execution time which could have been used for other computations.

A better approach would be to define an algorithm that **calculates a deadline that would preserve feasibility**, which is the approach we will take. The minimum feasible deadline is calculated and assigned to the recovery task, leading to feasibility of the task set.

### 7.2.1 Assigning Minimum Feasible Deadline to an Aperiodic Task

Hoang et al. presented in [21] the algorithm `minD_aperiodic()` which calculates the minimum deadline for a soft aperiodic task that preserves a feasible EDF schedule. This is applicable to our model since a recovery copy can simply be seen as an aperiodic task executing once.

The algorithm has to be slightly modified in order to implement TEM. For a task set  $\mathcal{T}$  and an aperiodic task  $\tau_a$  we get the pseudo-code in Algorithm 1. The algorithm initially sets the absolute deadline of the aperiodic tasks to the minimum possible value, which is the release time of the task added by its deadline, as seen in line 1.  $re_a$  is the release time of the aperiodic task. Processor-demand analysis is then performed on the task set, including the aperiodic task, to determine at which point the deadline can be set without causing the task set to become non-feasible. The deadline is delayed every time the processor-demand analysis failed, which corresponds to a deadline miss.

---

**Algorithm 1** minD\_aperiodic( $\mathcal{T}, \tau_a$ )

---

**Require:** EDF-feasible task set  $\mathcal{T}$  with  $U_{FF}(\mathcal{T}) < 1$ , aperiodic task  $\tau_a$

**Ensure:** Minimum feasibility sustainable deadline for  $\tau_a$

```
1:  $d_a := re_a + C_a$ 
2:  $\mathcal{K} := \{d_k \in \mathcal{D} : d_k \leq PC\}$ 
3:  $\mathcal{K} := \mathcal{K} \cup d_a$ 

4: for all  $L \in \mathcal{K}$  do
5:    $pd(L) := \sum_{i=1}^n \left( \left\lfloor \frac{L-D_i}{D_i} \right\rfloor + 1 \right) \times 2C_i$ 
6:   if  $L \geq d_a$  then
7:      $pd(L) := pd(L) + C_a$ 
8:   end if

9:   if  $pd(L) > L$  then
10:     $d_a := pd(L)$ 
11:    update_S() {with new value of  $d_a$ }
12:   end if
13: end for

14: return  $d_a$ 
```

---

Algorithm 1 can be used to **find the deadline of a recovery task**. As long as this new deadline is within the faulty task's original planning cycle, we can guarantee a feasible schedule, since we know that no more faults will occur. However, consider the case where the deadline of a recovery task is delayed in a way that it occurs within the next planning cycle. Parts of, or the whole, execution of the recovery task will then take place in another planning cycle than the original, meaning that we need to extend the scope we are analyzing. In turn, this means that we have one more fault to take in account, which might in turn get its deadline delayed into the next planning cycle. This is exemplified in Section 7.2.3.

### 7.2.2 Assigning Minimum Feasible Deadline to a Recovery Task

The solution to the iterative problem described above is presented in Algorithm 2, which, given an EDF-feasible task set employing TEM with one fault occurrence per planning cycle, returns a set of recovery tasks generating a feasible schedule. A **delay bound**, that is, a maximum number of time units a deadline might be extended, is also implemented. The algorithm iterates over the planning cycles, calculating the minimum deadline for the recovery task in each iteration. The iteration continues for as long as the calculated deadline is outside the current planning cycle. When the deadline of a recovery task is found to remain in its original planning cycle, the iteration is finished and the modified deadlines for the recovery tasks are returned.

The main work of the algorithm is done in minD\_recovery(), see Algorithm 3, which is called every iteration of algorithm 2 to find the minimum deadline for a given recovery task. minD\_recovery is a slightly modified version of Algorithm 1, the difference being that we need to take the previous scheduled recovery tasks in account when performing the processor-demand analysis. The control points are also calculated to be inside the current planning cycle, see line 3. The algorithm then performs processor-demand analysis at the control points. If, at a given point, the processor demand analysis fails, the deadline of the recovery task is set to the value

of the processor demand at that point. This new value is added to the set of control points and the iteration finishes until the set is empty, meaning the minimum deadline has been found.

Also worth mentioning is the variable  $ed_e$  in line 2 of algorithm 3 which is the time of the error detection, and hence, the release time of the recovery task, as specified in section 6.2.

---

**Algorithm 2** minD\_recovery\_n( $\mathcal{T}$ ,  $\tau_{e,rec}$ , delay\_bound)

---

**Require:** EDF-feasible task set  $\mathcal{T}$  with  $U_{FF}(\mathcal{T}) < 1$  and assumed recovery task  $\tau_{e,rec}$  for the first planning cycle. delay\_bound is the amount of time units a recovery task can delay its deadline. A negative delay\_bound means infinity.

**Ensure:** A set  $\mathcal{T}_{rec}$  of recovery tasks with assigned minimum deadlines or false if at least one assigned deadline is outside the delay\_bound.

```

1: curr_pc := 0
2:  $\mathcal{T}_{rec} := \emptyset$ 
3:  $d_{e,rec} := 0$ 

4: while curr_pc := 0 or  $d_{e,rec} > PC \times curr\_pc$  do
5:   curr_pc := curr_pc + 1
6:   if curr_pc  $\neq 0$  then
7:      $\tau_{e,rec} :=$  assume new recovery task
8:   end if
9:    $d_{e,rec} :=$  minD_recovery( $\mathcal{T}$ ,  $\tau_{e,rec}$ ,  $\mathcal{T}_{rec}$ )
10:  if  $d_{e,rec} - d_e > delay\_bound$  and  $delay\_bound > 0$  then
11:    return false
12:  end if
13:   $\mathcal{T}_{rec} := \mathcal{T}_{rec} \cup \tau_{e,rec}$ 
14: end while

15: return  $\mathcal{T}_{rec}$ 

```

---

---

**Algorithm 3** minD\_recovery( $\mathcal{T}, \tau_{e,rec}, \mathcal{T}_{rec}$ )

---

**Require:** EDF-feasible task set  $\mathcal{T}$  with  $U_{FF}(\mathcal{T}) < 1$ , recovery task  $\tau_{e,rec}$ , and previously scheduled recovery tasks  $\mathcal{T}_{rec}$

**Ensure:** Minimum feasibility sustainable deadline for  $\tau_{e,rec}$

```
1:  $d_{max,rec} := \text{biggest\_abs\_deadline}(\mathcal{T}_{rec})$  or 0 if  $\mathcal{T}_{rec} = \emptyset$ 
2:  $d_{e,rec} := \max(ed_e, d_{max,rec}) + C_e$ 
3:  $\mathcal{K} := \{d_k \in \mathcal{D} : d_{max,rec} < d_k \leq \lceil \frac{ed_e}{PC} \rceil \times PC\}$ 
4:  $\mathcal{K} := \mathcal{K} \cup d_{e,rec}$ 

5: for all  $L \in \mathcal{K}$  do
6:    $pd(L) := \sum_{i=1}^n \left( \left\lfloor \frac{L-D_i}{D_i} \right\rfloor + 1 \right) \times 2C_i$ 
7:   for all  $\tau_i \in \mathcal{T}_{rec} \cup \tau_{e,rec}$  do
8:     if  $L \geq d_i$  then
9:        $pd(L) := pd(L) + C_i$ 
10:    end if
11:  end for

12:  if  $pd(L) > L$  then
13:     $d_e := pd(L)$ 
14:    update_S() {with new value of  $d_e$ }
15:  end if
16: end for

17: return  $d_e$ 
```

---

### 7.2.3 Example: Minimum Feasible Deadline Assignment

Consider the following task set:

Task	$C_i$	$D_i$
$\tau_1$	1	6
$\tau_2$	1	9
$\tau_3$	3	18

Table 4: Task set

Given an error-free execution, the schedule in Figure 3 is generated. The planning cycle is 18 time units.

All tasks meet their deadlines, hence, the schedule is feasible. Also observe that there is slack at the end of the schedule. This means that  $U_{FF}(\mathcal{T}) < 1$ , that is, the utilization of the task set, given error-free execution, is less than 100%.

Now, let us assume that a fault in  $\tau_3$  is detected at  $t = 16$ . The schedule in Figure 4 is generated in which we see that  $\tau_{3,rec}$  misses its deadline at  $t = 18$ . Also observe that  $\tau_{3,rec}$  finishes its execution in the next planning cycle. We apply minD\_recovery\_n() to calculate a set of minimum deadlines for recovery tasks. Here, we assume an infinite delay\_bound.

The algorithm starts by deriving a deadline for  $\tau_{3,rec}$  by calling minD\_recovery(). Since no deadlines for previous recovery tasks have been assigned yet we get  $d_{max,rec} := 0$ . The deadline of  $\tau_{3,rec}$  is then set to the least possible value:

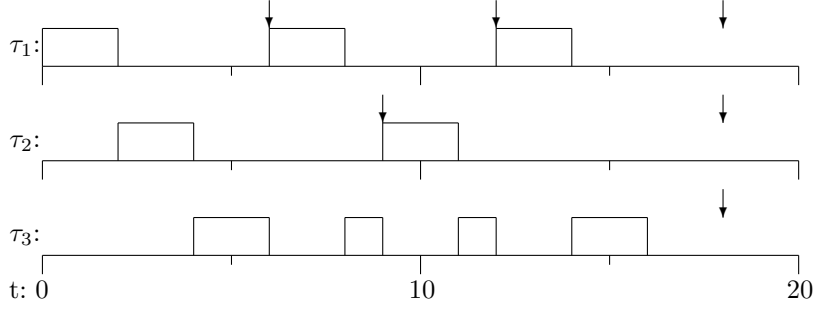


Figure 3: error-free execution

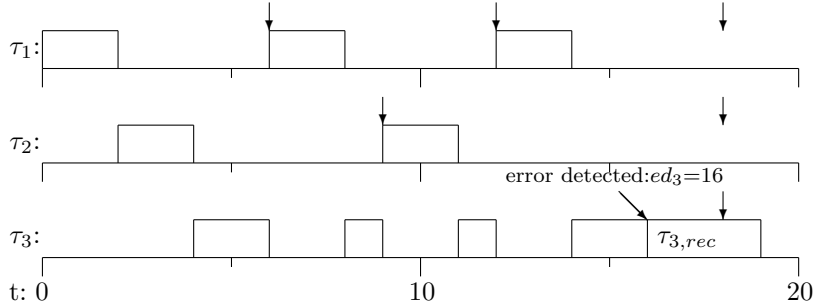


Figure 4: Error detected in  $\tau_3$  at  $t = 16$

$$d_{3,rec} = \max(ed_3, d_{max,rec}) + C_3 = \max(16, 0) + 3 = 19$$

Following this, the control points, that is, all absolute deadlines including the recovery task, for the task set is calculated:

$$\mathcal{K} = \{6, 12, 9, 18, 19\}$$

The processor-demand analysis with all the recovery tasks taken in account (so far only  $\tau_{3,rec}$ ) is then performed, see Table 5. In the table we see that the analysis is successful for every control point, which means that we do not need to delay the deadline any further,  $d_{3,rec} = 19$ , and the modified recovery task is saved in a set,  $\mathcal{T}_{rec}$ .

Since  $d_{3,rec} = 19$  is outside the current planning cycle we have to keep iterating in order to assign a deadline for the recovery task in the next planning cycle. We also need to assume a new fault occurrence. Let us assume that an error is detected in  $\tau_1$  at  $t = 21$  (note here that the execution of  $\tau_1$  has been delayed by 1 time unit compared to the error-free schedule since  $\tau_{3,rec}$  executed 1 time unit in the next planning cycle). The reason for assuming an error detection in  $\tau_1$  is that it will make the algorithm

terminate in this iteration. The algorithm derives a deadline for  $\tau_{1,rec}$  by once again calling `minD_recovery()`.

$d_{max,rec}$ , the deadline of the latest recovery task, is set to 19. The initial deadline for the new recovery task is then calculated as:

$$d_{1,rec} = \max(ed_1, d_{max,rec}) + C_1 = \max(19, 21) + 1 = 22$$

Now, we have to check the control points from  $d_{max,rec}$  to the end of the planning cycle:

$$\mathcal{K} = \{22, 24, 27, 30, 36\}$$

This analysis is performed in Table 6, and again the analysis succeeds for all control points, so  $d_{1,rec} = 22$  is returned. This time, however, the deadline is inside the original planning cycle, which ends at  $t = 36$ . This terminates the iteration and the following task set is returned from the main algorithm:

$$\mathcal{T}_{rec} = \{\tau_{3,rec} : d_{3,rec} = 19, \tau_{1,rec} : d_{1,rec} = 22\}$$

$\mathcal{T}$  together with  $\mathcal{T}_{rec}$  then gives us a feasible schedule that stretches over two planning cycles, see Figure 5. The longer arrows in the figure denote the deadlines of respective recovery task.

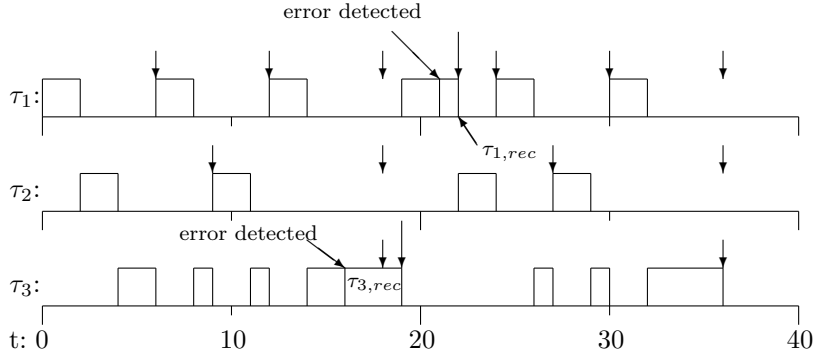


Figure 5: Recovery from errors at  $t = 16$  and  $t = 21$  by delaying a deadline

### 7.3 Calculating the Probability of System Success

In the following section we will derive formulas for **calculating the probability of system success**. As previously stated, we will use results from fault-injection experiments to derive this probability formula, see Table 2. The parameters in the table, for example  $P_X$  (Given that a fault occurs, an error is generated), will be heavily used in our formulas in order to derive a satisfactory model for overall system success. Also, we propose formulas in such a way that  $d_{e,rec}$ , the deadline of the recovery task, can never be larger than the planning cycle. Hence, we limit the analysis to one planning cycle. For a discussion on how to extend this analysis, see section 9.

$L$	$\tau_1$	$\tau_2$	$\tau_3$	$\mathcal{T}_{rec} \cup \tau_{3,rec}$	$\text{pd}(L)$	$\text{pd}(L) \leq L$
6	$(\lfloor \frac{6-6}{6} \rfloor + 1) \times 2 = 2$	$(\lfloor \frac{6-9}{9} \rfloor + 1) \times 2 = 0$	$(\lfloor \frac{6-18}{18} \rfloor + 1) \times 6 = 0$	0	2	YES
9	$(\lfloor \frac{9-6}{6} \rfloor + 1) \times 2 = 2$	$(\lfloor \frac{9-9}{9} \rfloor + 1) \times 2 = 2$	$(\lfloor \frac{9-18}{18} \rfloor + 1) \times 6 = 0$	0	4	YES
12	$(\lfloor \frac{12-6}{6} \rfloor + 1) \times 2 = 4$	$(\lfloor \frac{12-9}{9} \rfloor + 1) \times 2 = 2$	$(\lfloor \frac{12-18}{18} \rfloor + 1) \times 6 = 0$	0	6	YES
18	$(\lfloor \frac{18-6}{6} \rfloor + 1) \times 2 = 6$	$(\lfloor \frac{18-9}{9} \rfloor + 1) \times 2 = 4$	$(\lfloor \frac{18-18}{18} \rfloor + 1) \times 6 = 6$	0	16	YES
19	$(\lfloor \frac{19-6}{6} \rfloor + 1) \times 2 = 6$	$(\lfloor \frac{19-9}{9} \rfloor + 1) \times 2 = 4$	$(\lfloor \frac{19-18}{18} \rfloor + 1) \times 6 = 6$	3	19	YES

Table 5: Processor-demand analysis with  $\mathcal{T}_{rec} = \emptyset$

$L$	$\tau_1$	$\tau_2$	$\tau_3$	$\mathcal{T}_{rec} \cup \tau_{1,rec}$	$\text{pd}(L)$	$\text{pd}(L) \leq L$
22	$(\lfloor \frac{22-6}{6} \rfloor + 1) \times 2 = 6$	$(\lfloor \frac{22-9}{9} \rfloor + 1) \times 2 = 4$	$(\lfloor \frac{22-18}{18} \rfloor + 1) \times 6 = 6$	4	20	YES
24	$(\lfloor \frac{24-6}{6} \rfloor + 1) \times 2 = 8$	$(\lfloor \frac{24-9}{9} \rfloor + 1) \times 2 = 4$	$(\lfloor \frac{24-18}{18} \rfloor + 1) \times 6 = 6$	4	22	YES
27	$(\lfloor \frac{27-6}{6} \rfloor + 1) \times 2 = 8$	$(\lfloor \frac{27-9}{9} \rfloor + 1) \times 2 = 6$	$(\lfloor \frac{27-18}{18} \rfloor + 1) \times 6 = 6$	4	24	YES
30	$(\lfloor \frac{30-6}{6} \rfloor + 1) \times 2 = 10$	$(\lfloor \frac{30-9}{9} \rfloor + 1) \times 2 = 6$	$(\lfloor \frac{30-18}{18} \rfloor + 1) \times 6 = 6$	4	26	YES
36	$(\lfloor \frac{36-6}{6} \rfloor + 1) \times 2 = 12$	$(\lfloor \frac{36-9}{9} \rfloor + 1) \times 2 = 8$	$(\lfloor \frac{36-18}{18} \rfloor + 1) \times 6 = 12$	4	36	YES

Table 6: Processor-demand analysis with  $\mathcal{T}_{rec} = \{\tau_{3,rec} : d_{3,rec} = 19\}$

Given our model, assuming **exactly one fault per planning cycle**, it is appropriate to divide the analysis into three cases:

- error-free operation, no error is detected.
- Error detection by comparison of the two primary tasks or by a timer monitor in case of deadline violation.
- Error detection by an error detection mechanism before a task has finished its execution.

By deriving adequate probability-formulas for each of the three scenarios we can derive the probability of **overall system success**,  $P_{success}$ , by summing them together, as they are independent events.

### 7.3.1 Error-free Execution

Given error-free execution of a planning cycle, there are two possibilities of fault occurrence. Either the fault occurs during one of the primary executions of a task, or the fault occurs when a primary task is not executing, see Figure 6.

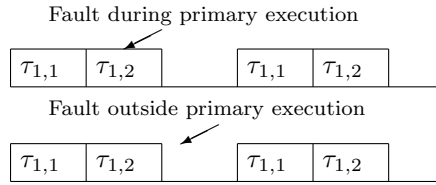


Figure 6: Different possibilities of fault occurrence

The probability of fault occurrence when a primary task is not executing,  $P_{NF}$ , is defined as follows:

$$P_{NF} = 1 - \sum_{i=1}^n P(U_i)$$

Fault occurrence during a primary execution can be further divided into two sub-cases. The first sub-case is that an error is generated by the fault without being detected, which is expressed, given the parameters in Table 2, as:

$$P_{EG,ND} = \sum_{i=1}^n P(U_i) \times P_X \times P_{ND}$$

In the other sub-case, the fault does not generate an error:

$$P_{NEG} = \sum_{i=1}^n P(U_i) \times (1 - P_X)$$

Now, in order to express the probability of success for the whole scenario we need to take the processor demand analysis in account. Since we are considering a scenario

without a detected fault, we can remove that aspect from Equation (1). Further, to be able to use this mathematically we also need to embed this into a formula, so we get for a task set  $\mathcal{T}$ :

$$\zeta(\mathcal{T}) = \begin{cases} 1 & \text{if } \forall L : \sum_{i=1}^n \left( \left\lfloor \frac{L - D_i}{D_i} \right\rfloor + 1 \right) \times 2C_i \leq L \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

The function  $\zeta(\mathcal{T})$  simply returns 1 if the task set is feasible according to the processor-demand analysis, otherwise it returns 0.

We can now derive a formula for system success for the case of no detected error,  $P_{FF}$ . We sum up the three different sub cases of fault occurrence as specified above and multiply it with the function  $\zeta(\mathcal{T})$  in order to make sure that the probability is calculated as 0 in case of a non-feasible EDF-schedule.

$$P_{FF} = \zeta(\mathcal{T}) \times (P_{NF} + P_{EG,ND} + P_{NEG}) \quad (3)$$

### 7.3.2 Error Detection by Double Execution or Timer Monitor

In this scenario, we assume that an error is detected in  $\tau_{ej}$  by comparison after double execution, or by a timer monitor detecting the violation of a deadline. The error is then masked by running a recovery copy. Consequently, we get the following probabilities:

$$P_{EG,DE} = P(U_{ej}) \times P_X \times P_{DE} \times P_{DE,M}$$

which corresponds to the error being masked after being detected by comparison after double execution.

$$P_{EG,TM} = P(U_{ej}) \times P_X \times P_T \times P_{T,M}$$

corresponds to the error being masked after detection by a timer monitor.

Similar to the case of no detected error, we also have to take the processor-demand analysis in account. Recall that Algorithm 2, given a task set, `delay_bound` and an initial recovery task, derived a set of recovery tasks assigned minimum feasibility sustainable deadlines. This is done by performing processor-demand analysis, and hence we can use the function in our analysis. `delay_bound` is, as previously explained a number specifying how much the deadline for a recovery task can be delayed, where `delay_bound = 0` means that the deadline is the same as the original tasks deadline.  $PC - d_{ej,rec}$  is the interval between the deadline of the recovery task and the end of the planning cycle, and hence `delay_bound = PC - dej,rec` would result in a recovery task which could be assigned at most a deadline equal to the end of the planning cycle. See Figure 7.

$d_{ej,rec}$  is calculated as  $j \times D_e$  for task  $\tau_{ej}$ . We also introduce a **scale factor**,  $0 \leq \lambda \leq 1$ , that scales the delay bound. If  $\lambda = 0$  the deadline can not be delayed at all, whereas it can be delayed to the end of the planning cycle if  $\lambda = 1$ . If  $\lambda$  is assigned a value between the extremes, consequently we get a maximum deadline assignment somewhere between them. For task set  $\mathcal{T}$  we define the following function:

$$\gamma(\mathcal{T}, \lambda) = \begin{cases} 1 & \text{if } \text{minD\_recovery\_n}(\mathcal{T}, \tau_{ej,rec}, [\lambda \times (PC - j \times D_e)]) \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

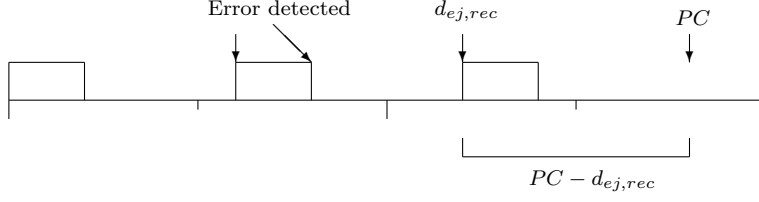


Figure 7:  $PC - d_{ej,rec}$  is the maximum time  $\tau_{ej}$  can be delayed

$\gamma(\mathcal{T}, \lambda)$  simply returns 1 if  $\tau_{ej}$  sustain feasibility of the task set given that its deadline is delayed at most  $\lfloor \lambda \times (PC - j \times D_e) \rfloor$ , as discussed above. The probability that  $\tau_{ej}$  keep its deadline is thus:

$$P_{ej,DET}(\lambda) = \gamma(\mathcal{T}, \lambda) \times (P_{EG,DE} + P_{EG,TM})$$

which can be rewritten as:

$$P_{ej,DET}(\lambda) = \gamma(\mathcal{T}, \lambda) \times P(U_{ej}) \times P_X \times (P_{DE} \times P_{DE,M} + P_T \times P_{T,M}) \quad (5)$$

### 7.3.3 Error Detection by Error Detection Mechanism

If an error is detected by an error detection mechanism, the execution of the faulty task is aborted immediately and a recovery task is executed. Depending on when the error is detected, this may result in increased probability of system success, since more slack will be available in the schedule. Let us denote the execution time of the faulty task  $C_{ej,det}$  given that the error is detected in  $\tau_{ej}$ . Further, we assume a certain latency between the error occurrence and the error detection,  $t_{lat}$ . This value can be seen as a constant, derived by the results in [5]. In order to later on compare our results with a corresponding analysis for RM scheduling, we use the value calculated by Lou [29],  $t_{lat} = 0.45s = \frac{9}{20}$ .

The probability of an error being masked after detection by an error detection mechanism is similar to the previous case, with the difference that we have to weight the probability of fault occurrence because of the shorter execution time in  $\tau_{ej}$ . We also need to take the error detection latency in account. We get:

$$P_{ej,EDM} = \frac{C_{ej,det} - t_{lat}}{C_e} \times P(U_{ej}) \times P_X \times P_{EDM} \times P_{EDM,M} \quad (6)$$

What we need to do in order to be able to analyse this case, however, is to derive a value for  $C_{ej,det}$ . The most straightforward way to do this is to set  $C_{ej,det}$  to the maximum value it can execute while still keeping a feasible schedule, and that is the approach we will take. Note that this value can not be larger than the original execution,  $C_e$ . Deriving a value for  $C_{ej,det}$  is very straightforward. We simply need to check how much the deadline for the recovery task of  $\tau_{ej}$  needs to be delayed, in order to maintain a feasible schedule. Instead of delaying the deadline we derive a value for  $C_{ej,det}$  by subtracting  $C_e$  with that value. This will give us the maximum value for  $C_{ej,det}$ . Algorithm 4 implements this, and it is exemplified in Section 7.3.4.

Since we want to find the maximum value for  $C_{ej, det}$  we assign  $d_{ej, rec}$  the maximum allowed value, that is  $d_{ej, rec} = d_{ej, rec} + \lfloor \lambda \times (PC - j \times D_e) \rfloor$ , similar to the previous case. As discussed, this corresponds to delaying the deadline with as much as the delay bound allows.

---

**Algorithm 4**  $\max C_{ej, det}(\mathcal{T}, \tau_{ej, rec})$

---

**Require:** EDF-feasible task set  $\mathcal{T}$ , and recovery task  $\tau_{ej, rec}$  with an assigned deadline

**Ensure:** Maximum feasibility sustainable execution time for task where an error detection mechanism detected an error,  $\tau_{ej, det}$

```

1:  $d_{tmp} = d_{ej, rec}$  {Save original deadline}
2:  $\mathcal{T}_{rec} = \text{minD\_recovery.n}(\mathcal{T}, \tau_{ej, rec}, \lfloor \lambda \times (PC - j \times D_e) \rfloor)$ 
3: if  $\mathcal{T}_{rec} = \emptyset$  then
4:   return false
5: else
6:    $\text{needed\_slack} = \min(C_e, d_{ej, rec} - d_e)$ 
7:    $d_{ej, rec} = d_{tmp}$  {Set back to original deadline}
8:   return  $C_e - \text{needed\_slack}$ 
9: end if

```

---

### 7.3.4 Example: Maximum Execution Time for $C_{e, det}$

Consider the task set in Table 7, where a fault is detected by EDM in the first invocation of  $\tau_1$ . Assume  $\lambda = 0$ , that is the deadline of the recovery task is not allowed to be delayed. The deadline of the recovery task is thus assigned the value  $d_{1, rec} = 8$ .

Task	$C_i$	$D_i$	$d_{i, rec}$
$\tau_1$	3	8	
$\tau_2$	1	24	
$\tau_{1, rec}$	3		8

Table 7: Task set

If we assume all tasks execute the full execution time, we get the schedule in Figure 8.

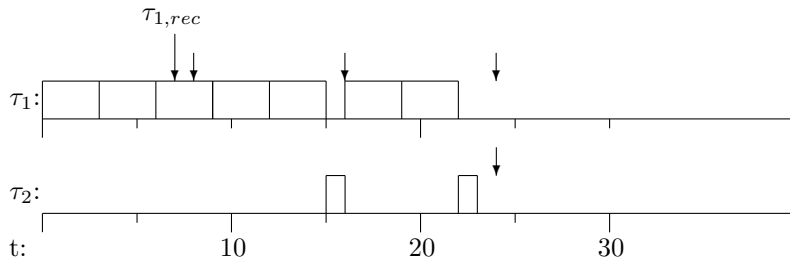


Figure 8:  $\tau_{1, rec}$  misses its deadline

We easily see that  $t_{1,rec}$  will miss its deadline unless the faulty task,  $\tau_{1,det}$ , shorten its execution time. So let us apply Algorithm 4 on the task set to determine how long  $\tau_{1,det}$  can run at most while still keeping a feasible schedule.

In line 2 of the algorithm, our previously defined algorithm,  $\text{minD\_recovery\_n}()$ , is applied in order to derive a minimum deadline for the recovery task. It is easy to see from Figure 8 that the deadline is calculated to be  $d_{1,rec} = 9$ , for an extensive example of that algorithm see Section 7.2.3. At row 6 the algorithm calculates the amount of time units that the deadline needs to be delayed in order to keep a feasible schedule,  $d_{ej,rec} - d_e$ . This value can also be seen as the amount of time units by which we need to decrease the execution of  $\tau_{1,det}$ , in order to keep a feasible schedule. The algorithm returns this new execution time, in this case  $C_{1,det} = 3 - \text{min}(3, 9 - 8) = 2$ .

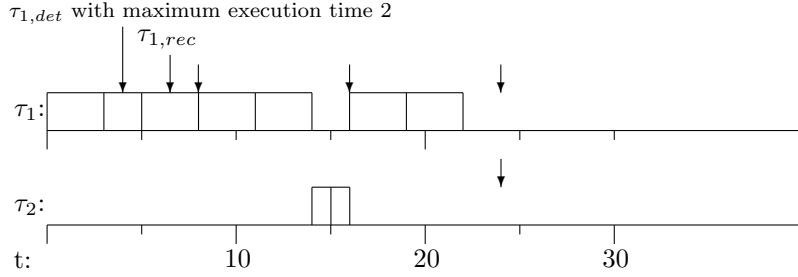


Figure 9:  $\tau_{1,det}$  scheduled with maximum execution time,  $\tau_{1,rec}$  keeps its deadline.

### 7.3.5 Probability of Overall System Success

We have so far derived probabilities for three different scenarios, as explained in Sections 7.3.1 - 7.3.3. To derive the probability of overall system success we only have to sum the probabilities, where we apply every case to every task in a task set  $\mathcal{T} = \{\tau_1, \dots, \tau_n\}$ . To that end, we get:

$$P_{SUCCESS}(\lambda) = P_{FF} + \sum_{e=1}^n \sum_{j=1}^m P_{ej,DET}(\lambda) + \sum_{e=1}^n \sum_{j=1}^m P_{ej,EDM} \quad (7)$$

where  $m$  is the number of invocations of each task in each planning cycle.

Notice that in the second and third term we iterate over all invocations of each task, while in the first term this iteration takes place inside the calculation of  $P_{FF}$ .  $\lambda$  is assigned a value such as  $0 \leq \lambda \leq 1$ , which specifies how much the deadline of the recovery task is allowed to be delayed, as explained in Section 7.3.2.

## 8 Schedulability Analysis Examples

In this section we will exemplify our proposed methods by analysing two different task sets. Example 1-3 examine the effect of delaying the deadlines of recovery tasks by

varying  $\lambda$  for a given task set, while example 4 compares our proposed methods to a similar technique for RM scheduling.

### 8.1 Example 1: $\lambda = 0$

In this example,  $\lambda$  is set to 0 causing the deadline of a recovery task to be hard, that is, we are not allowed to delay it. We use the following task set:

Task	$C_i$	$D_i$
$\tau_1$	3	8
$\tau_2$	1	24

Table 8: Task set

Under error-free execution the schedule in Figure 10 is generated, where the planning cycle is 24.

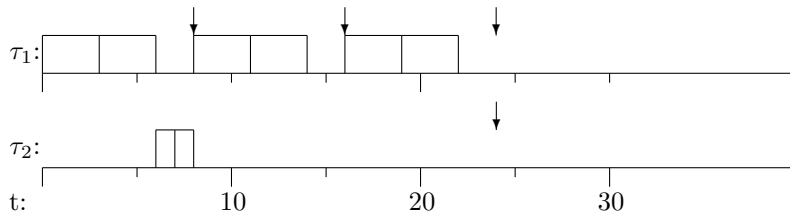


Figure 10: error-free execution

The probability of fault occurrence in the tasks are  $P(U_1) = \frac{3 \times 2}{8} = \frac{3}{4}$  and  $P(U_2) = \frac{1 \times 2}{24} = \frac{1}{12}$ , hence  $\sum_{i=1}^n P(U_i) = \frac{3}{4} + \frac{1}{12} = \frac{5}{6}$ . Probabilities of fault occurrence in the individual invocations of the tasks are  $P(U_{1i}) = \frac{3 \times 2}{24} = \frac{1}{4}$  and  $P(U_{2i}) = \frac{1 \times 2}{24} = \frac{1}{12}$ .

#### 8.1.1 Case 1: Error-free Execution

We start by performing processor-demand analysis of the task set, without assuming any errors. The analysis is found in Table 9.

As we see in the table the analysis is successful for all  $L$ , meaning that function (2) returns the value 1. We get the probability  $P_{FF}$  for this case, by equation (3).

$L$	$\tau_1$	$\tau_2$	$pd(L)$	$pd(L) \leq L$
8	$(\lfloor \frac{8-8}{8} \rfloor + 1) \times 6 = 6$	$(\lfloor \frac{8-24}{24} \rfloor + 1) \times 2 = 0$	6	YES
16	$(\lfloor \frac{16-8}{8} \rfloor + 1) \times 6 = 12$	$(\lfloor \frac{16-24}{24} \rfloor + 1) \times 2 = 0$	12	YES
24	$(\lfloor \frac{24-8}{8} \rfloor + 1) \times 6 = 18$	$(\lfloor \frac{24-24}{24} \rfloor + 1) \times 2 = 2$	20	YES

Table 9: Processor-demand analysis given no fault

$$\begin{aligned}
P_{FF} &= \zeta(\mathcal{T}) \times (P_{NF} + P_{EG,ND} + P_{NEG}) \\
&= 1 \times \left( 1 - \sum_{i=1}^n P(U_i) + \sum_{i=1}^n P(U_i) \times P_X \times P_{ND} + \sum_{i=1}^n P(U_i) \times (1 - P_X) \right) \\
&= 1 \times \left( 1 - \frac{5}{6} + \frac{5}{6} \times \frac{373}{2076} \times \frac{0}{373} + \frac{5}{6} \times \left( 1 - \frac{373}{2076} \right) \right) \\
&= \frac{1}{6} + \frac{5}{6} \times \frac{1703}{2076} \approx 0.85027
\end{aligned}$$

### 8.1.2 Case 2: Error Detection after Double Execution

In this case we need to iterate over all invocations of the tasks. From Figure 10 we see that  $\tau_1$  is invoked three times in the meta-period, while  $\tau_2$  is invoked only once. Equation (4) is applied on each task invocation, which basically calculates the smallest feasibility sustainable deadline of the recovery task. If the calculated deadline is delayed more than  $\lambda$  (here 0) time units the equation returns 0. For this example the deadline assignment is quite trivial, so we only need to look at the schedule in Figure 10 to realise the minimum feasibility sustainable deadline for each recovery task. See Section 7.2.3 for a detailed example of the algorithm.

For all invocations of  $\tau_1$  ( $\tau_{11}$ ,  $\tau_{12}$ , and  $\tau_{13}$ ), we see that there is no room for a recovery task without delaying it, see Figure 11. Hence Equation (4) returns 0 given fault occurrences in these task invocations, causing the probability to turn to 0:

$$P_{11,DET}(0) = P_{12,DET}(0) = P_{13,DET}(0) = 0$$

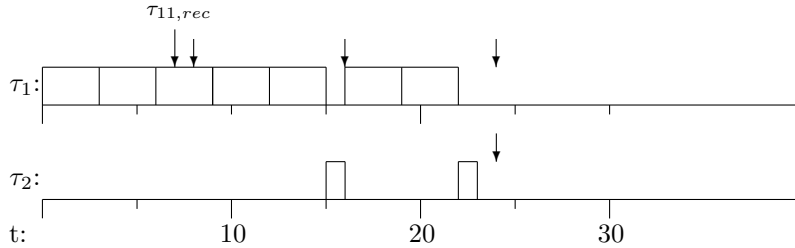


Figure 11:  $\tau_{11,rec}$  misses its deadline

If the fault occurs in  $\tau_{21}$ , however, there is enough room in the schedule to execute a recovery copy within its deadline. We show this in the analysis performed in Table 10, and the resulting schedule in Figure 12. The analysis succeeds, hence Equation (4) evaluates to 1. We get the following probability for  $\tau_{21}$ :

$$\begin{aligned}
P_{21,DET}(0) &= \gamma(\mathcal{T}, \lambda) \times P(U_{21}) \times P_X \times (P_{DE} \times P_{DE,M} + P_T \times P_{T,M}) \\
&= 1 \times \frac{1}{12} \times \frac{373}{2076} \times \left( \frac{68}{373} \times \frac{68}{68} + \frac{19}{373} \times \frac{18}{286} \right) \\
&\approx 0.00278
\end{aligned}$$

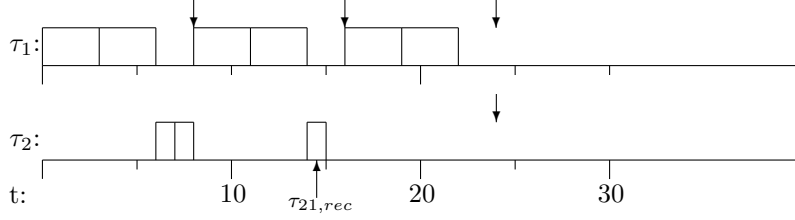


Figure 12: There is enough slack to execute a recovery copy of  $\tau_{21}$

We sum up the probabilities for the different task invocations to get the probability for this case to happen:

$$\sum_{e=1}^n \sum_{j=1}^m P_{ej,DET}(\lambda) = P_{11,DET}(0) + P_{12,DET}(0) + P_{13,DET}(0) + P_{21,DET}(0) \approx 0.00278$$

### 8.1.3 Case 3: Error Detection by Error Detection Mechanism

If an error is detected by an error detection mechanism we iterate over all task invocations, just as in the previous case. For each task invocation, Algorithm 4 is applied, which simply calculates the maximum time the task invocation can execute while still keeping a feasible schedule. Like the previous scenario we will not execute the algorithm step by step; for a detailed example see Section 7.3.4. The value returned by the algorithm is then used in Equation (6) to derive a probability of system success.

Given a fault occurrence in  $\tau_{11}$  that is detected by an error detection mechanism, Equation (4) returns  $C_{11,det} = 2$ . This is shown in Figure 13, where the recovery task for  $\tau_{11}$  just keeps its deadline, and can not have a larger execution time. The same goes for  $\tau_{12}$  and  $\tau_{13}$ , so we get the following probabilities:

$$\begin{aligned}
P_{11,EDM} = P_{12,EDM} = P_{13,EDM} &= \frac{C_{ej,det} - t_{lat}}{C_e} \times P(U_{ej}) \\
&\quad \times P_X \times P_{EDM} \times P_{EDM,M} \\
&= \frac{2 - \frac{9}{20}}{3} \times \frac{3 \times 2}{24} \times \frac{373}{2076} \times \frac{286}{373} \times \frac{194}{286} \\
&\approx 0.01201
\end{aligned}$$

$L$	$\tau_1$	$\tau_2$	$\tau_{21,rec}$	$pd(L)$	$pd(L) \leq L$
8	$(\lfloor \frac{8-8}{8} \rfloor + 1) \times 6 = 6$	$(\lfloor \frac{8-24}{24} \rfloor + 1) \times 2 = 0$	0	6	YES
9	$(\lfloor \frac{8-8}{8} \rfloor + 1) \times 6 = 6$	$(\lfloor \frac{8-24}{24} \rfloor + 1) \times 2 = 0$	1	7	YES
16	$(\lfloor \frac{16-8}{8} \rfloor + 1) \times 6 = 12$	$(\lfloor \frac{16-24}{24} \rfloor + 1) \times 2 = 0$	1	13	YES
24	$(\lfloor \frac{24-8}{8} \rfloor + 1) \times 6 = 18$	$(\lfloor \frac{24-24}{24} \rfloor + 1) \times 2 = 2$	1	21	YES

Table 10: Processor-demand analysis including recovery task  $\tau_{21,rec}$

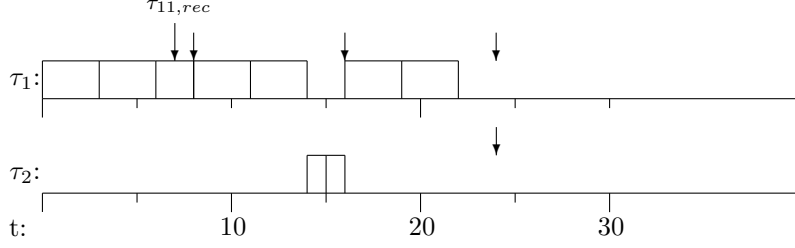


Figure 13:  $\tau_{11,rec}$  can execute at most 2 time units to keep its deadline at  $t = 8$

If the fault occurs in  $\tau_{21}$  it is easy to see that  $C_{21,det} = 1$ , the task can execute fully while still keeping enough slack in the schedule. This is again shown in Figure 12. We get:

$$\begin{aligned}
 P_{21,EDM} &= \frac{C_{21,det} - t_{lat}}{C_e} \times P(U_{21}) \times P_X \times P_{EDM} \times P_{EDM,M} \\
 &= \frac{1 - \frac{9}{20}}{1} \times \frac{1 \times 2}{24} \times \frac{373}{2076} \times \frac{286}{373} \times \frac{194}{286} \\
 &\approx 0.00428
 \end{aligned}$$

The probability for the whole scenario is summed up as:

$$\sum_{e=1}^n \sum_{j=1}^m P_{ej,EDM} = P_{11,EDM} + P_{12,EDM} + P_{13,EDM} + P_{21,EDM} \approx 0.04031$$

#### 8.1.4 Probability of Overall System Success

The probability of overall system success given is calculated by adding the three scenarios together:

$$\begin{aligned}
 P_{SUCCESS}(\lambda) &= P_{FF} + \sum_{e=1}^n \sum_{j=1}^m P_{ej,DET}(\lambda) + \sum_{e=1}^n \sum_{j=1}^m P_{ej,EDM} \\
 &\approx 0.85027 + 0.00278 + 0.04031 \\
 &= 0.89336
 \end{aligned}$$

## 8.2 Example 2: $\lambda = 0.5$

For this example we use the same task set as the previous example in order to study the effect of an increased delay bound,  $\lambda = 0.5$ .

Task	$C_i$	$D_i$
$\tau_1$	3	8
$\tau_2$	1	24

Table 11: Task set

### 8.2.1 Case 1: Error-free Execution

This case is clearly identical to the previous example:

$$\begin{aligned}
P_{FF} &= \zeta(\mathcal{T}) \times (P_{NF} + P_{EG,ND} + P_{NEG}) \\
&= 1 \times \left( 1 - \sum_{i=1}^n P(U_i) + \sum_{i=1}^n P(U_i) \times P_X \times P_{ND} + \sum_{i=1}^n P(U_i) \times (1 - P_X) \right) \\
&= 1 \times \left( 1 - \frac{5}{6} + \frac{5}{6} \times \frac{373}{2076} \times \frac{0}{373} + \frac{5}{6} \times \left( 1 - \frac{373}{2076} \right) \right) \\
&= \frac{1}{6} + \frac{5}{6} \times \frac{1703}{2076} \approx 0.85027
\end{aligned}$$

### 8.2.2 Case 2: Error Detection after Double Execution

Recall that in the previous example, with  $\lambda = 0$ , all invocations of  $\tau_1$  failed if the fault occurred in them. This was because there was not enough slack before the deadline. With  $\lambda = 0.5$ , we delay the deadline for the recovery task half way to the end of the planning cycle, increasing the probability of success. The amount of time units each task invocation can be delayed is, as used in Equation (4):

$$\begin{aligned}
\tau_{11,rec} &= \lfloor 0.5 \times (24 - 1 \times 8) \rfloor = 8 \\
\tau_{12,rec} &= \lfloor 0.5 \times (24 - 2 \times 8) \rfloor = 4 \\
\tau_{13,rec} &= \lfloor 0.5 \times (24 - 3 \times 8) \rfloor = 0 \\
\tau_{21,rec} &= \lfloor 0.5 \times (24 - 1 \times 24) \rfloor = 0
\end{aligned}$$

It is trivial to see in Figure 10 that  $\tau_{11}$  and  $\tau_{12}$  only need to have their deadlines delayed one time unit to make room for a recovery task, while  $\tau_{13}$  has its original deadline at the end of the planning cycle, hence it cannot be delayed.  $\tau_{21}$  do not need to have its deadline delayed, since it succeeded even with  $\lambda = 0$ , and consequently the probability for  $\tau_{21}$  is the same as the previous example. We get:

$$\begin{aligned}
P_{11,DET}(0.5) = P_{12,DET}(0.5) &= 1 \times \frac{3 \times 2}{24} \times \frac{373}{2076} \times \left( \frac{68}{373} \times \frac{68}{68} + \frac{19}{373} \times \frac{18}{286} \right) \\
&\approx 0.00833 \\
P_{13,DET}(0.5) &= 0 \\
P_{21,DET}(0.5) &\approx 0.00278
\end{aligned}$$

We sum up the probabilities:

$$\begin{aligned} \sum_{e=1}^n \sum_{j=1}^m P_{ej,DET}(\lambda) &= P_{11,DET}(0.5) + P_{12,DET}(0.5) + P_{13,DET}(0.5) + P_{21,DET}(0.5) \\ &\approx 0.01944 \end{aligned}$$

By comparing this result to the corresponding analysis where  $\lambda = 0$  (section 8.1.2), we see that the derived probability for this case to happen is  $\frac{0.01944}{0.00278} \approx 7$  times bigger for  $\lambda = 0.5$ .

### 8.2.3 Case 3: Error Detection by Error Detection Mechanism

In the previous example, with  $\lambda = 0$ ,  $\tau_{11}$  and  $\tau_{12}$  needed to decrease their execution one time unit to provide enough slack for their recovery tasks. Here, with  $\lambda = 0.5$  it is trivial to see that they can execute their full execution time, since the recovery tasks have their deadlines delayed.  $\tau_{21}$  can also execute fully. However  $\tau_{13}$  can not delay its deadline, hence  $\tau_{13,det} = 2$ , which provides enough slack for the recovery task. We get the following probabilities:

$$\begin{aligned} P_{11,EDM} = P_{12,EDM} &= \frac{3 - \frac{9}{20}}{3} \times \frac{3 \times 2}{24} \times \frac{373}{2076} \times \frac{286}{373} \times \frac{194}{286} \approx 0.01986 \\ P_{13,EDM} &= \frac{2 - \frac{9}{20}}{3} \times \frac{3 \times 2}{24} \times \frac{373}{2076} \times \frac{286}{373} \times \frac{194}{286} \approx 0.01207 \\ P_{21,EDM} &= \frac{1 - \frac{9}{20}}{1} \times \frac{1 \times 2}{24} \times \frac{373}{2076} \times \frac{286}{373} \times \frac{194}{286} \approx 0.00428 \end{aligned}$$

Hence, the probability of the whole scenario is:

$$\sum_{e=1}^n \sum_{j=1}^m P_{ej,EDM} = P_{11,EDM} + P_{12,EDM} + P_{13,EDM} + P_{21,EDM} \approx 0.05607$$

If we compare this result to the analysis with  $\lambda = 0$ , we see that we have **increased the probability** for this case to happen from 0.04031 to 0.05607. It is appropriate to note that the difference between these two values would be larger if our assumed error detection latency would have been smaller.

### 8.2.4 Probability of Overall System Success

We get the following probability for success of the system:

$$\begin{aligned} P_{SUCCESS}(\lambda) &= P_{FF} + \sum_{e=1}^n \sum_{j=1}^m P_{ej,DET}(\lambda) + \sum_{e=1}^n \sum_{j=1}^m P_{ej,EDM} \\ &\approx 0.85027 + 0.01944 + 0.05607 \\ &= 0.92578 \end{aligned}$$

The corresponding probability for  $\lambda = 0$  is 0.89336, so we see that by increasing  $\lambda$ , we have increased the probability of system success.

### 8.3 Example 3: $\lambda = 1$

This case happens to be identical to the case where  $\lambda = 0.5$ . The deadlines for the recovery tasks are delayed enough with  $\lambda = 0.5$ , delaying them further will not affect the schedule at all. Hence, we get:

$$\begin{aligned} P_{SUCCESS}(\lambda) &= P_{FF} + \sum_{e=1}^n \sum_{j=1}^m P_{e_j, DET}(\lambda) + \sum_{e=1}^n \sum_{j=1}^m P_{e_j, EDM} \\ &\approx 0.85027 + 0.01944 + 0.05607 \\ &= 0.92578 \end{aligned}$$

### 8.4 Example 4: Comparison to RM scheduling

In this example we will use the same task set as in [29], which allows us to compare our work with the corresponding work using RM scheduling. In order to do this we also assume  $\lambda = 0$ , that is, we do not allow delayed deadlines for the recovery tasks. The following schedule is generated, where the planning cycle is 400:

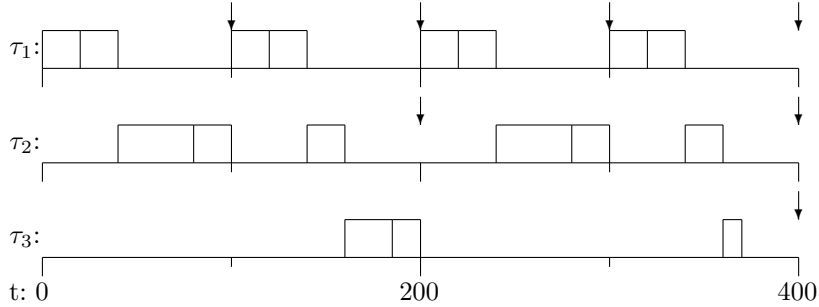


Figure 14: error-free execution

We have  $P(U_1) = \frac{20 \times 2}{100} = \frac{2}{5}$ ,  $P(U_2) = \frac{40 \times 2}{200} = \frac{2}{5}$ , and  $P(U_3) = \frac{25 \times 2}{400} = \frac{1}{8}$ , hence  $\sum_{i=1}^n P(U_i) = \frac{2}{5} + \frac{2}{5} + \frac{1}{8} = \frac{37}{40}$ . Probabilities of fault occurrence in the individual invocations of the tasks are calculated as  $P(U_{1i}) = \frac{20 \times 2}{400} = \frac{1}{10}$ ,  $P(U_{2i}) = \frac{40 \times 2}{400} = \frac{1}{5}$ , and  $P(U_{3i}) = \frac{25 \times 2}{400} = \frac{1}{8}$ .

#### 8.4.1 Case 1: Error-free Execution

As we see in Figure 14, the schedule is feasible. The probability for a successful error-free scenario is thus:

$$\begin{aligned}
P_{FF} &= \zeta(\mathcal{T}) \times (P_{NF} + P_{EG,ND} + P_{NEG}) \\
&= 1 \times \left( 1 - \sum_{i=1}^n P(U_i) + \sum_{i=1}^n P(U_i) \times P_X \times P_{ND} + \sum_{i=1}^n P(U_i) \times (1 - P_X) \right) \\
&= 1 \times \left( 1 - \frac{37}{40} + \frac{37}{40} \times \frac{373}{2076} \times \frac{0}{373} + \frac{37}{40} \times \left( 1 - \frac{373}{2076} \right) \right) \\
&= \frac{3}{40} + \frac{37}{40} \times \frac{1703}{2076} \approx 0.83380
\end{aligned}$$

#### 8.4.2 Case 2: Error Detection after Double Execution

It is trivial to see that there is enough slack available to run recovery copies of any of the invocations of  $\tau_1$  or  $\tau_3$ . We show this in Figure 15. However, a fault occurrence in  $\tau_2$  would cause the schedule to become infeasible, since only 30 time units of slack are available, and  $C_2$  is 40, see Figure 16. We get:

$$\begin{aligned}
P_{1i,DET}(0) &= \gamma(\mathcal{T}, \lambda) \times P(U_{1i}) \times P_X \times (P_{DE} \times P_{DE,M} + P_T \times P_{T,M}) \\
&= 1 \times \frac{1}{10} \times \frac{373}{2076} \times \left( \frac{68}{373} \times \frac{68}{68} + \frac{19}{373} \times \frac{18}{286} \right) \\
&\approx 0.00333 \\
P_{2i,DET}(0) &= 0 \\
P_{3i,DET}(0) &= \gamma(\mathcal{T}, \lambda) \times P(U_{1i}) \times P_X \times (P_{DE} \times P_{DE,M} + P_T \times P_{T,M}) \\
&= 1 \times \frac{1}{8} \times \frac{373}{2076} \times \left( \frac{68}{373} \times \frac{68}{68} + \frac{19}{373} \times \frac{18}{286} \right) \\
&\approx 0.00417
\end{aligned}$$

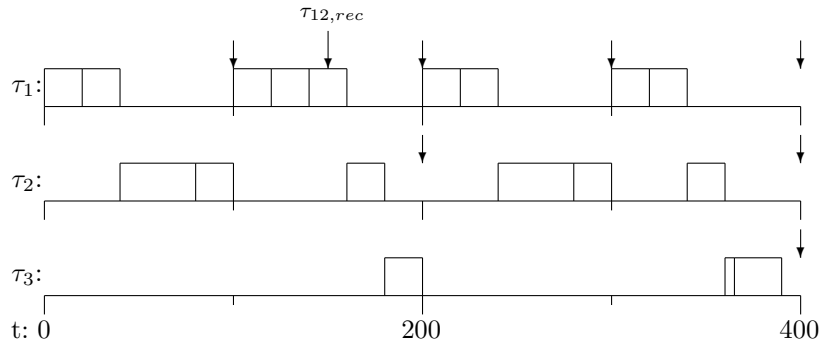


Figure 15:  $\tau_{12,rec}$  can execute fully and keep its deadline

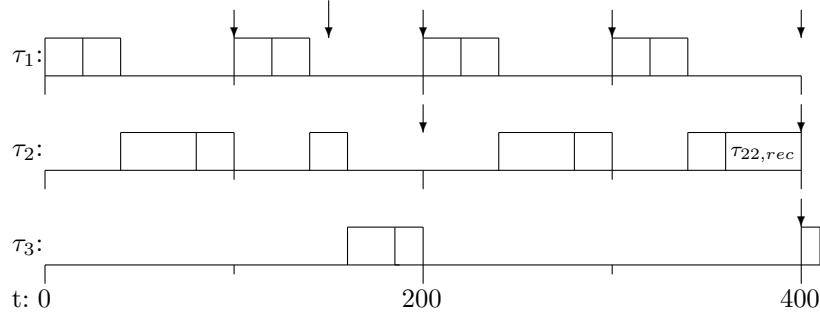


Figure 16:  $\tau_{22,rec}$  causes  $\tau_3$  to miss its deadline

Since we have the same probabilities for each invocation of  $\tau_1$ ,  $\tau_2$ , and  $\tau_3$  we can multiply the probabilities by the number of invocations. Thus, for the whole scenario we get the following probabilities:

$$\sum_{e=1}^n \sum_{j=1}^m P_{ej,DET}(\lambda) = 4 \times P_{1i,DET}(0) + 2 \times P_{2i,DET}(0) + 1 \times P_{3i,DET}(0) \approx 0.01749$$

#### 8.4.3 Case 3: Error Detection by Error Detection Mechanism

If a fault would occur in any of the invocations of  $\tau_1$  or  $\tau_3$  the schedule maintains feasibility, even if the error is detected at the very end of the task. Hence  $C_{1i,det} = 20$  and  $C_{3i,det} = 25$ . However, both  $\tau_{21}$  and  $\tau_{22}$  can only execute 30 time units in order to spare enough space for a recovery task, so  $C_{2i,det} = 30$ . This is trivial to see in Figure 14, since  $\tau_3$  will miss its deadline at 400 if a recovery task for  $\tau_2$  would execute more than 30 time units. We get the following probabilities:

$$\begin{aligned} P_{1i,EDM} &= \frac{20 - \frac{9}{20}}{20} \times \frac{1}{10} \times \frac{373}{2076} \times \frac{286}{373} \times \frac{194}{286} \approx 0.00913 \\ P_{2i,EDM} &= \frac{30 - \frac{9}{20}}{40} \times \frac{1}{5} \times \frac{373}{2076} \times \frac{286}{373} \times \frac{194}{286} \approx 0.01381 \\ P_{3i,EDM} &= \frac{25 - \frac{9}{20}}{25} \times \frac{1}{8} \times \frac{373}{2076} \times \frac{286}{373} \times \frac{194}{286} \approx 0.01147 \end{aligned}$$

Similar to the previous case, we sum up the probabilities as:

$$\sum_{e=1}^n \sum_{j=1}^m P_{ej,DET}(\lambda) = 4 \times P_{1i,EDM}(0) + 2 \times P_{2i,EDM}(0) + 1 \times P_{3i,EDM}(0) \approx 0.07561$$

#### 8.4.4 Probability of Overall System Success

We can now sum up the probability of overall system success:

$$\begin{aligned}
P_{SUCCESS}(\lambda) &= P_{FF} + \sum_{e=1}^n \sum_{j=1}^m P_{ej,DET}(\lambda) + \sum_{e=1}^n \sum_{j=1}^m P_{ej,EDM} \\
&\approx 0.83380 + 0.01749 + 0.07561 \\
&= 0.92690
\end{aligned}$$

The corresponding analysis performed by Lou in [29] resulted in a probability of 93.11% that the system will succeed, which is slightly higher than the result in our analysis. However, Lou made an error in approximating the probability of error generation given a fault occurrence to 17% when it is actually  $\approx 18\%$ , which naturally had a positive effect on the RM analysis. In fact, **the two analyses are identical** except for the difference stated above. The probabilities would have been exactly the same if Lou would have used more accurate approximations. This could be easily shown by using the same erroneous approximation as above in this analysis, which would lead to the two different approaches producing identical formulas, given this task set.

## 8.5 Examples Summary and Discussion

The effect of adjusting  $\lambda$  is found in Figure 17. As we can see, **the probability of system success in our case increases with a larger value for  $\lambda$** . This is not surprising since increasing  $\lambda$  relaxes deadlines for the recovery tasks. However, despite that the result might be obvious it could still be very useful.  $\lambda$  can be set to a pre-defined value, depending on the type of system. For example, an airplane might have very hard deadlines, and thus a low value for  $\lambda$ , while streaming video might not be as critical, thus  $\lambda$  can be given a higher value. An appropriate value can be found by running the system several times, in order to evaluate the performance of the system employing different delay bounds. Our presented methods can even be used to **find the least feasibility sustainable value for  $\lambda$** , given our model of one fault per planning cycle.

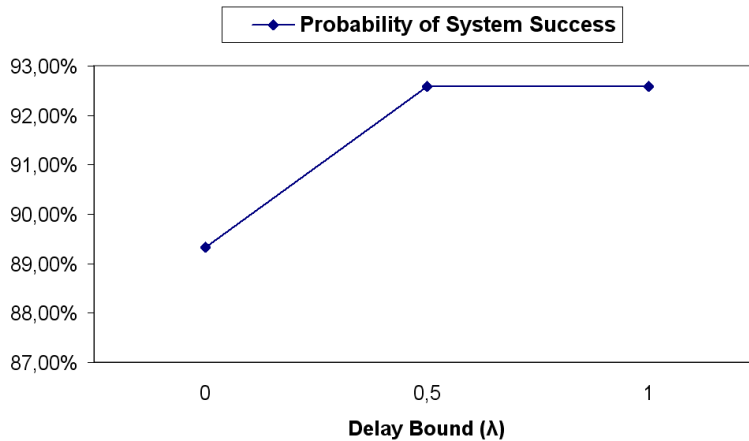


Figure 17: The probability of system success increases with a larger  $\lambda$

If we look at example 4 we have showed that our proposed fault-tolerant EDF scheduling can be as good as the fault-tolerant RM scheduling presented by Lou in [29].

## 9 Extended Schedulability Analysis

In Section 7.2 we investigated the possibility of increasing probability of system success by delaying the deadline of the recovery task. However, when performing our analysis as specified in Section 7.3.2, we assumed  $d_{e,rec} \leq PC$ , that is, the deadline of the recovery task is not allowed to be placed outside the planning cycle. This simplifies our analysis since, if we would place the deadline in the next planning cycle, we would need to take another fault occurrence in account. This section briefly discusses some possible problems and advantages by delaying the deadlines into the next planning cycle.

It is easy to see that there are cases when a deadline delayed into the next planning cycle could be positive. Imagine a task set with low utilization, but somehow a recovery task would need to be assigned a deadline one time unit into the next planning cycle. Not allowing this to happen would make the schedule infeasible. On the other hand, allowing this to happen would not necessarily decrease probability of system success, since there might be enough slack in each planning cycle to allow a delay of one time unit. However, in the new planning cycle, consider a fault in the same invocation of the same task. This would now force the deadline to be delayed two time units into the following planning cycle. See Figure 18, where we consider fault occurrence in the same invocations of  $\tau_{ej,rec}$  twice, in some simplified task set. We see that the task is delayed more into  $PC3$  than into  $PC2$ .

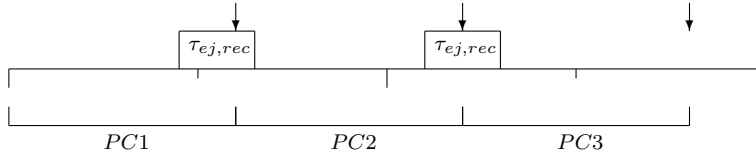


Figure 18:  $\tau_{ej,rec}$  gets delayed more into  $PC3$  than into  $PC2$

The task in a task set that delays the deadline of its recovery task the most could in this sense be considered the most critical. A solution to the problem above, with deadlines being iteratively assigned to occur in the next planning cycle, could for example be to **bound the number of times a task is allowed to be delayed into the next planning cycle**. This would limit the problem, while it could increase the probability of system success for a task set.

Another problem that would have to be taken in account when allowing for deadlines delayed into the next planning cycle is the possibility of fault occurrence in a recovery task. This fault would occur in the part of the task that is executing in the following planning cycle. The schedule could become infeasible, or we could choose to delay the deadline further.

## 10 Conclusions

In this thesis we have addressed the problem of allowing faults in real-time systems. Given a strictly limited model scheduled under EDF, we suggested ways to **determine the probability of system success**. Our analysis is employing TEM in order to achieve NLFT, and thus results from such experiments are used to derive probabilities. However, given another model, other experimental results could be used in a similar manner.

We also proposed two algorithms.  $\text{minD\_Recovery}_n()$  to **derive a set of minimum deadlines providing a feasible schedule**, and  $\text{max}C_{ej, det}()$  to **calculate the maximum time a task can execute while still maintaining feasibility**. Both of these were used in the analysis. Algorithm  $\text{minD\_Recovery}()$  is a part of the  $\text{minD\_Recovery}_n()$ -algorithm, but can also be used on its own to **derive the minimum feasibility sustainable deadline of a recovery task**.

Further, we investigated the effect of **relaxing strict deadlines** of re-executing tasks, recovery tasks, by introducing a **scale factor**,  $\lambda$ . This scale factor can be assigned a value between 0 and 1, where 0 means hard deadlines for recovery tasks, and 1 means they can be delayed extensively. We showed that **by increasing  $\lambda$ , it is possible to increase the probability of overall system success**. We limited our analysis to one planning cycle, which in turn limited how much a recovery task can be delayed; the deadline of the recovery task has to occur inside the planning cycle. However, we investigated the effects of relaxing the analysis even further, by removing this restriction, and letting the deadline be assigned a value inside another planning cycle.

Finally, we showed that our proposed methods for fault tolerant EDF scheduling can be as good as a similar approach taken for RM scheduling.

## 11 Future Work

Future work related to this thesis could be to extend the analysis to include multiple faults. This would increase the complexity of the problem, but also make it more realistic. Further, our current analysis does not allow recovery tasks to be assigned deadlines outside the planning cycle we are analysing, mainly due to complexity reasons. However, it would be far more realistic to allow for deadlines to be delayed into the next planning cycle, which we discussed in Section 9. This would also increase the probability of success for our system.

Another interesting extension would be to try to find examples where our methods actually produce better results than the ones produced by fault tolerant RM scheduling. So far we have only shown that the two methods can produce equal results. Moreover, it would be interesting to see how other experimental results, performed under other fault models than the ones provided in our thesis, would affect the probabilities.

## References

- [1] J. Abawajy. Fault-tolerant dynamic job scheduling policy. In *6th international conference on algorithms and architectures for parallel processing*, pages 165–173, October 2005.
- [2] J. Aidemark. *Node-Level Fault Tolerance for Embedded Real-Time Systems*. PhD thesis, Chalmers University of Technology, 2004.
- [3] J. Aidemark, P. Folkesson, and J. Karlsson. A framework for node-level fault tolerance in distributed real-time systems. Technical Report 04-06, Department of Computer Engineering, Chalmers University of Technology, 2004.
- [4] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. Experimental evaluation of time-redundant execution for a brake-by-wire application. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 210–215, June 2002.
- [5] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. Experimental dependability evaluation of the artk68-ft real-time kernel. In *Proceedings of the International Conference on Real-Time and Embedded Computing Systems and Applications*, August 2004.
- [6] H. Aydin. Exact fault-sensitive feasibility analysis of real-time tasks. *IEEE Transactions on Computers*, 56(10):1372–1386, October 2007.
- [7] P. Balbastre, I. Ripoll, and A. Crespo. Analysis of window-constrained execution time systems. *Real-Time Systems*, 35(2):109–134, February 2007.
- [8] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptively scheduling of periodic, real-time tasks on one processor. *Real-Time Systems Journal*, 2(4):301–324, November 1990.
- [9] H. Beitollahi, S. G. Miremadi, and G. Deconinck. Fault-tolerant earliest-deadline-first scheduling algorithm. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1–6, March 2007.
- [10] V. Berten, J. Goosens, and E. Jeannot. A probabilistic approach for fault tolerant multiprocessor real-time scheduling. In *20th International Parallel and Distributed Processing Symposium*, April 2006.
- [11] A. Burns, R. Davis, and S. Punnekkat. Feasibility analysis of fault-tolerant real-time task sets. In *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems*, pages 29–33, June 1996.
- [12] A. Burns, S. Punnekkat, L. Strigini, and D. R. Wright. Probabilistic scheduling guarantees for fault-tolerant real-time systems. In *Dependable Computing for Critical Applications 7*, pages 361–378, January 1999.
- [13] Y. Chen, X. Yu, and G. Xiong. Fault-tolerant earliest deadline first scheduling with resource reclaim. In *Proceedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing*, pages 278–285, October 2002.

- [14] H. Chetto and M. Silly. Scheduling strategies for periodic tasks to avoid timing faults in critical control systems. In *Automatic Control. World Congress 1993. Proceedings of the 12th Triennial World Congress of the International Federation of Automatic Control*, volume 2, pages 725–728, July 1994.
- [15] J. Contreras and J.-L. Sourrouille. Edf improvements for faults reduction. In *Proceedings of the IEEE International Symposium on Industrial Electronics*, volume 1, pages 75–78, July 1999.
- [16] S. Ghosh, R. Melhem, and D. Mossé. Enhancing real-time schedules to tolerate transient faults. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 120–129, December 1995.
- [17] S. Ghosh, R. Melhem, D. Mossé, and J. S. Sarma. Fault-tolerant rate-monotonic scheduling. *Real-Time Systems*, 15(2):149–181, September 1998.
- [18] M. Hamilton. Towards ultra reliable medical systems. In *Symposium Record of Policy Issues in Information and Communication Technologies in Medical Applications*, number 71-85 in 1, September 1988.
- [19] C.-C. Han, K. G. Shin, and J. Wu. A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults. *IEEE Transactions on Computers*, 52(3):362–372, March 2003.
- [20] M. G. Harbour and J. Palencia. Response time analysis for tasks scheduled under edf within fixed priorities. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, pages 200–209, December 2003.
- [21] H. Hoang, G. Buttazzo, M. Jonsson, and S. Karlsson. Computing the minimum edf feasible deadline in periodic systems. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 125–134, August 2006.
- [22] Y. Hong and H. Goo. A fault-tolerant scheduling scheme for hybrid tasks in distributed real-time systems. In *Proceedings of the Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, pages 3–6, May 2005.
- [23] M. Joseph and P. Pandya. Finding response times in a real-time system. *Computer Journal*, 29(5):390–395, 1986.
- [24] N. Kandasamy, J. P. Hayes, and B. T. Murray. Tolerating transient faults in statically scheduled safety-critical embedded systems. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, pages 212–221, October 1999.
- [25] G. N. Khan and A. Sydhom. Fault-tolerant scheduling of real-time tasks having software faults. In *Canadian Conference on Electrical and Computer Engineering*, pages 731–734, May 2005.
- [26] C. M. Krishna and K. G. Shin. *Real-Time Systems*. McGraw-Hill International Editions, 1997.
- [27] F. Liberato, R. Melhem, and D. Mossé. Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems. *IEEE Transactions on Computers*, 49(9):906–914, September 2000.

- [28] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.
- [29] Q. Lou. Real-time scheduling analysis of system employing tem. Master’s thesis, Chalmers University of Technology, 2005.
- [30] G. Manimaran and C. S. R. Murthy. A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis. *IEEE Transactions on Parallel and Distributed Systems*, 9(11):1137–1152, November 1998.
- [31] E. Nett, H. Streich, P. Bizzarri, A. Bondavalli, and F. Tarini. Adaptive software fault tolerance policies with dynamic real-time guarantees. In *Proceedings on the Second Workshop on Object-Oriented Real-Time Dependable Systems (WORDS ’96)*, pages 78–85, February 1996.
- [32] M. Pandya and M. Malek. Minimum achievable utilization for fault-tolerant processing of periodic tasks. *IEEE Transactions on Computers*, 47(10):1102–1112, October 1998.
- [33] R. M. Pathan. Real-time scheduling analysis of systems tolerating multiple transient faults. Master’s thesis, Chalmers University of Technology, 2005.
- [34] R. M. Pathan. Fault-tolerant real-time scheduling algorithm for tolerating multiple transient faults. In *4th International Conference on Electrical and Computer Engineering*, pages 577–580, December 2006.
- [35] S.-M. Ryu and D.-J. Park. Checkpointing for the reliability of real-time systems with on-line fault detection. In *Lecture Notes in Computer Science*, pages 194–202. Springer, November 2005.
- [36] P. Sinha and N. Suri. On the use of formal techniques for analyzing dependable real-time protocols. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 126–135, December 1999.
- [37] H. X. Zhao, L. George, and S. Midonnet. Worst case response time analysis of sporadic graph tasks with edf scheduling on a uniprocessor. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 23–29, August 2005.

## A APPENDIX - Fault Injection Results

$P_X$	Given that a fault occurs, an error is generated	373 (of 2076)	18%
$P_{DE}$	Given that an error is generated, the error is detected by comparison after double execution (DE)	68 (of 373)	18%
$P_T$	Given that an error is generated, the error is detected by a timer monitor (TM)	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 TM, the error is masked by TEM	18 (of 286)	6%
$P_{EDM,M}$	Given that an error is detected by EDM, the error is masked by TEM	194 (of 286)	68%

Table 12: Results from fault injection into a 68340 microprocessor