LEBANESE UNIVERSITY
Faculty of Technology
Master of Information System Engineering
Laboratory of Embedded and Networked Systems

Energy-Efficient Fault-Tolerant Scheduling for Hard Real-Time Systems

Master Thesis

LAMIS ABOU DARWISH

Laboratory of Embedded and Networked Systems

Advisor: **Dr. Hussein EL GHOR**
Co-advisor: **Dr. Haissam HAJJAR**

Saida, 2018

Laboratory of Embedded and Networked Systems

LEBANESE UNIVERSITY
Faculty of Technology
Master of Information System Engineering
Laboratory of Embedded and Networked Systems

Energy-Efficient Fault-Tolerant Scheduling for Hard Real-Time Systems

Master Thesis

LAMIS ABOU DARWISH

Advisor: **Dr. Hussein EL GHOR**
Co-advisor: **Dr. Haissam HAJJAR**

Approved by Date:.

(Signature)	*(Signature)*	*(Signature)*
Dr. Hussein EL GHOR	Dr. Haissam HAJJAR	Dr. Mohamed HAJJAR

Saida, 2018

Acknowledgements

I would like to thanks all the people that in some way have helped me with the elaboration of this thesis.

First, I would like to thanks to my advisor, Dr. Hussein EL GHOR, for all the help, support, and amiability. He has been an excellent advisor. Sometimes he has shown me the way in those dark days of the academy work, discovering when I have been lost and pulling me when I have needed.

I would like also to thanks to my family. They are all that I have! My mom had always seen me as a superstar, my nephews always spoiling my homework but in a good way and my brother and sister breaking their heads thinking why I spend a lot of time doing these strange things. All of them had encouraged me to follows this way without stopping in the difficult moments that always are there no matter where you are.

I would like to thanks to the master committee and ordinator and especially to the dean of the Faculty of Technology professor Mohamed Hajjar for all the help, support, and amiability.

Although it is difficult to list them, I would like to thanks to all my friends. They always have been there with a good hand extended for me without taking care of my bad mood. Especially, I would like to thanks to the master group. They have welcomed me and I have learned something from all of them.

Finally, I would like to thanks to the Lebanese University and the Faculty of Technology, the crazy place where I have spent a lot of time with the propose of complete this puzzle.

Saida, 2018

Lamis Abou Darwish

Abstract

For the past several decades, we have experienced tremendous growth of real-time systems and applications largely due to the remarkable advancements of IC technology. However, as transistor scaling and massive integration continue, the dramatically increased power/energy consumption and degraded reliability of IC chips have posed significant challenges to the design of real-time systems.

Fault tolerance in underlying hardware is of paramount importance on the reliability of real-time safety-critical systems especially due to the continuously decreasing feature size. In such systems, fault tolerant techniques must respect the timing constraints of the task set in a way that faults have to be detected and appropriate recovery operations must be completed before the deadlines. Conceivably, guaranteeing the reliability of computing systems has also been raised to a first-class design concern.

In addition, the power consumption of battery-operated portable embedded systems have been dramatically increased which further degraded the system reliability and hence posed additional significant challenges to the design of real-time systems. Recent research on real-time autonomous systems has focused on developing advanced scheduling methods on hard real-time systems that are subject to multiple design constraints, in particular, timing, energy reduction, and reliability constraints. Extensive power management techniques have been developed on energy minimization so as to maximize the lifetime achieved as in classical battery operated devices. Among these techniques, the most commonly popular and widely used and traditional approach is dynamic voltage and frequency scaling (DVFS). However, using energy management techniques such as DVFS has made reliability issues to be exacerbated. Hence, energy management and fault-tolerance are two conflicting key objectives in the design of efficient real-time embedded systems.

This thesis targets the problem of designing energy efficient fault-tolerant real-time scheduling algorithms for independent aperiodic tasks on a uniprocessor platform. To this end, we must first investigate the energy management problem with fault-tolerance requirements for hard real-time tasks on a single-core processor that is powered by an energy storage unit and uses DVFS technique to reduce the energy consumption and hence prolong the systems lifetime.

Keywords

Real-time systems, fault tolerance, energy manangement, scheduling, renewable energy.

To my family and friends.

Contents

List of Figures

List of Tables

Chapter 1

General Introduction

Nowadays, embedded systems are becoming increasingly important in our lives. In these embedded devices, the management of energy is a crucial issue. They are more and more varied and appear in extremely diverse sectors such as transport (avionics, cars, buses, ..), multimedia, mobile phones, game consoles, etc. A large part of embedded systems have needs for autonomy and limitations of space (small size) and energy (limited consumption). As a result, the major technological and scientific challenge is to build systems of trust from the point of view of the functionalities provided and the rendered quality of service. It's more about designing these systems at an acceptable cost.

Having drawn heavily on fossil fuel reserves (oil, coal, ..), the use of renewable energies (solar, wind, ..) is proving to be an alternative of choice to feed many systems. For example, for a cell phone whose battery stores a limited amount of energy and can be recharged periodically by its user, the problem here is to minimize the consumed energy so as to maximize its autonomy. Such a problem is usually treated by Dynamic Voltage and Frequency Scaling (DVFS) methods that affect the speed of the processor, which directly affects the energy consumption of the system. However, this assumes that the used processors in this kind of platform support various operating frequencies. Moreover, many new generation portable embedded systems limit or even prohibit human interventions, particularly because they are difficult to access because of the environment in which they operate (example surveillance application of a forest or motorway area), or because they are deployed in very large numbers (example dense wireless sensor network for an accurate topographic survey). These systems then operate more and more often with batteries and / or supercapacitors that recharge continuously with a renewable energy source such as solar energy. Designing such fully autonomous, embedded systems, however, requires the resolution of a number of problems related to the harvesting of ambient energy, its storage and its use, so as to ensure sustainable autonomy, while maintaining a respect for the temporal constraints of the treatment system.

However, the use of renewable energy to power a portable embedded system induces a number of problems related to the characteristics of embedded computing and electronics. Indeed, the singularity of an embedded system lies in its operation in real time: it is subject to temporal constraints attached to the realization of the various activities that it must implement and that consume energy. In so-called critical applications, not only the non-respect of these time constraints can affect the rendered quality of service, but it can in certain critical applications be unacceptable because generating the final shutdown of the system will further degradate the

used hardware or even loss of human lives. When an embedded system has no energy constraint, the major problem to be solved during its design will consist in verifying the feasibility of the application with regard to the temporal specifications, the processing capacities of the hardware architecture used and the needs in processing time of the application software. This problem is well known, studied for forty years. The underlying problem is a classic scheduling problem where only the *"time"* dimension is studied and can be reduced to optimizing a quantity called Quality of Service (QoS). Until very recently, it was assumed that energy was not a constraint. This was supposed to be available in sufficient quantity to ensure the functioning of the system over the entire lifetime of the application. Previous scheduling algorithms can no longer be suitable for an embedded system powered by a renewable energy source and using a small battery whose level fluctuates over time depending on the recovered amount of energy.

At the same time, it is observed that as embedded real-time systems become more and more complex, the required level of reliability for such systems appears to be another open problem. Many of these systems tend to be situated at harsh, remote or inaccessible locations. Consequently, it is often difficult and sometimes even impossible to repair and to perform maintenance. This necessitates the use of fault-tolerant techniques. Fault-tolerant computing refers to the correct execution of user programs and system software in the presence of faults [1]. A system is more reliable if it can cope with fault cases and can be fault-tolerant in the presence of faults. The timing constraints of the real-time applications can be satisfied using appropriate *task scheduling* and the required level of reliability can be achieved by means of *fault-tolerance*. In the case of an energy-autonomous system, reliability also means ensuring that the system will never be short of energy to ensure its treatment. Anticipation of possible cases of energy can, again, be implemented on the basis of the flexibility offered by the system at the level of the execution of the tasks.

As part of this thesis, we are interested in the problem of real-time scheduling under reliability and energy constraints. It's about considering real-time tasks that have needs that are expressed on the one hand in terms of processing time and energy consumed by the processor and on the other hand in terms of the number of tolerated faults. A task configuration is energy overloaded, this means that the amount of energy consumed is greater than the amount of energy available. In additon, the amount of execution time requested is smaller than the available capacity, the system will therefore typically be able to meet all its deadlines or else catastrophic consequences will occur. A major question that needs to be answered is: how to schedule real-time tasks in case of energy where the system keeps reliable and able to tolerate up to k faults.

To answer this question, a uniprocessor Earliest Deadline First (EDF) scheduling algorithm is first analyzed to derive an efficient and exact feasibility condition by considering energy management and fault-tolerance. Second, a scheduling algorithm is designed to achieve energy-autonomous utilization of the processor while meeting the task deadlines and while considering that the system is fed by a renewable energy source. The goal of the former algorithm is to achieve reliability while the goal of the latter algorithm is to achieve a high performance. In this thesis, it is also discussed how to blend these two metrics into the same scheduling framework.

To this end, our target is to investigate the energy managemt problem with fault-tolerance requirements for dynamic-priority based hard real-time tasks on a single-core processor. We develop scheduling algorithms to judiciously make tradeoffs between fault tolerance and energy management since both design objectives usually conflict with each other.

Our thesis is organized as follows:
Chapter 2 introduces concepts and algorithms for scheduling in real-time systems at first. Then, in a second step, we present the concepts and factors interfering in the schedulability test. We focus on on-line scheduling for uniprocessor systems. After that, we direct our work to the state of art related to scheduling of periodic tasks.
Chapter 3 proposes the concept of real-time energy harvesting systems. First, we describe the embedded systems and more precisely the wireless sensor networks. At present, these constitute the majority of embedded applications built around autonomous systems from the energy point of view. Then, we focus on the storage and extraction of renewable energy to power these autonomous systems. We describe a state of the art about technologies associated with energy recovery. After that, we describe the problem of energy autonomy in a system often described as energetically neutral. Finally, we describe different scheduling techniques with the aim of minimizing energy consumption and then aiming to perform an energetically autonomous system.
Chapter 4 introduces the problem of fault-tolerance in real-time systems, scheduling tasks characterized not only by an execution duration constrained by time but also by needs in energy to realize this execution. First, we describe the embedded systems and more precisely the wireless sensor networks. At present, these constitute the majority of embedded applications built around autonomous systems from the energy point of view. Then, we focus on the storage and extraction of renewable energy to power these autonomous systems. We describe a state of the art about technologies associated with energy recovery. After that, we describe the problem of energy autonomy in a system often described as energetically neutral. Finally, we describe different scheduling techniques with the aim of minimizing energy consumption and then aiming to perform an energetically autonomous system.
Chapter 5 targets the problem of designing advanced real-time scheduling algorithms that are subject to timing, energy consumption and fault-tolerant design constraints. To this end, we first investigated the problem of developing scheduling techniques for uniprocessor real-time systems that minimizes energy consumption while still tolerating up to k transient faults to preserve the system's reliability. Two scheduling algorithms are proposed: The first algorithm is an extension of an optimal fault-free low-power scheduling algorithm, named ES-DVFS. The second algorithm aims to enhance the energy saving by reserving adequate slack time for recovery when faults strike. We derive a necessary and sufficient condition that can be checked efficiently for the time and energ feasibility of aperiodic jobs in the presence of faults. Later, we formally prove that the proposed algorithm is optimal for a k-fault-tolerant model. Our simulation results show that, the proposed approach can achieve more energy savings over previous works under reliability constraint.

Chapter 2

State of Art

This chapter is an introduction to real-time computing systems with different types of constraints such as time and energy constraints and fault-tolerant constraints. We first present the main concepts of real-time systems and introduce the problem of real-time scheduling. Then, we focus on three points: (i) monoprocessor real-time systems, (ii) real-time fault tolerant systems and (iii) real time systems subject to energy constraints.

2.1 Real-time Systems

2.1.1 The Concept of Real-Time

The meaning of the concept of real-time is very broad and far from what one can imagine. A real-time system is not a fast-moving system but a system capable of reacting to external stimuli in specified times (temporal constraints). We focus therefore on the definition of time as measurable physical data. An real-time application is therefore a set of activities with associated temporal constraints. The definition of real time widely adopted in the field is that of Stankovic [2]:

Definition 2.1. *The correction of a real-time system does not only depend on the logical result calculations but also the time at which the results are produced.*

In computing systems, we consider the system to be a real-time one when it is able to control a physical process at a speed adapted to the evolution of the control process. Real-time computing systems differentiate themselves from other computing systems by taking into account time constraints whose respect is as important as the accuracy of the delivered results. In other words, the different possible sequences of treatments of the system guarantee that each one of them does not exceed their temporal limits.

2.1.2 Specificities of Real-Time Systems

2.1.2.1 Definitions and Major Characteristics

The definition of the widely adopted real-time system is as follows [3]

Definition 2.2. *We call a real-time system any system whose functioning is subject to the dynamic evolution over time of an external process with which it interacts and whose behavior must be controlled by exploiting limited resources.*

The behavior of a real-time system has different characteristics. We quote six major characteristics that define it:

- *Logical and temporal accuracy:* the system must be able to provide outputs that are consistent with the inputs of the system and this in respect of the temporal constraints.
- *Predictability:* The purpose behind using a real-time system is to ensure that all the tasks (with all their execution configurations) will meet their deadlines. These must therefore be expected and executed within the specified time constraints. To guarantee this, we always use the worst case execution scenario.
- *Determinism:* real-time system always responds in the same way to an incoming event, that is, the system produces the same outgoing event.
- *Reliability:* the system responds to the availabile constraints. Hardware and software components in the system must be reliable, that is, they must be able to provide correct processing of the received as well as the defined information.
- *Complexity:* Every real-time system has a certain complexity that comes from not only from the non-determinism of the external environment with which it interacts (events in the external world often occur asynchronously and appear in an unpredictable order) but also from the features that it achieves.
- *Fault-Tolerant:* in order to respect the constraint of reliability, some real-time systems must be designed to be tolerant to certain faults that may occur or else the system will shut down.

2.1.2.2 Taxonomy of Real-Time Systems

Real-time systems are classified according to the level of criticism of their temporal constraints [4]. We then speak of systems:

- *Hard Real-Time Systems:* In such systems, the non respect of the temporal constraint leads to system faults that can generate catastrophic consequences (in terms of human lives, impact on the environment, or even on the economy) on the system itself or its environment. This means that all system processes must necessarily respect all their temporal constraints. These systems include air traffic control, missile control systems, supervision of nuclear power plants, etc.
- *Soft Real-Time Systems:* This constraint is less demanding than the hard constraint which needs an absolute respect of all the temporal constraints. In such systems, it is acceptable to miss some of the deadlines occasionally with additional value for the system to finish the task, even if it is late. The task's overflow can cause acceptable degradation without serious consequences to the system. An example of this is the difference between sound and image in a video projection.
- *Firm Real-Time Systems:* It is a subclass of soft real time flexible for which the occasional failure of tasks is authorized. Unlike soft-constrained systems, firm real-time system tolerates up to an associated deadline misses, but eventually the performance will degrade if too many

are missed. Therefore, every firm real-time task is associated with some predefined deadline before which it is required to produce its results. However, unlike a hard real-time task, even when a firm real-time task does not complete within its deadline, the system does not fail. The late results are merely discarded. In other words, the utility of the results computed by a firm real-time task becomes zero after the deadline. In firm real-time systems, the quality of such systems is quantified: the measurement of the temporal constraints violations takes the form of a probabilistic data which one will call *Quality of Service (QoS)*. The QoS is related to a particular service such as the number of treated bits of the sound system in your computer. If you miss a few bits, no big deal, but miss too many and you're going to eventually degrade the system. Similar would be seismic sensors. If you miss a few datapoints, no big deal, but you have to catch most of them to make sense of the data. More importantly, nobody is going to die if they don't work correctly.

2.1.3 Characterization and Modeling of Real-Time Tasks

2.1.3.1 Definitions

From the processor's point of view, a *task* is an activity that consumes resources of the computer machine (memory and CPU time). A real-time application consists of a set of tasks. The term *task* refers to the portion of computer code resulting from the compilation of a high level language that will be executed by the processor. A task can be executed a multitude of times during the system life time. We can cite as an example a task that regularly raises the temperature of a sensor. Real-time tasks can be modeled by thinking each periodic task as consisting of a sequential stream of *job* or *task instance* [5]. So, a job is an instance of a real-time task. In a

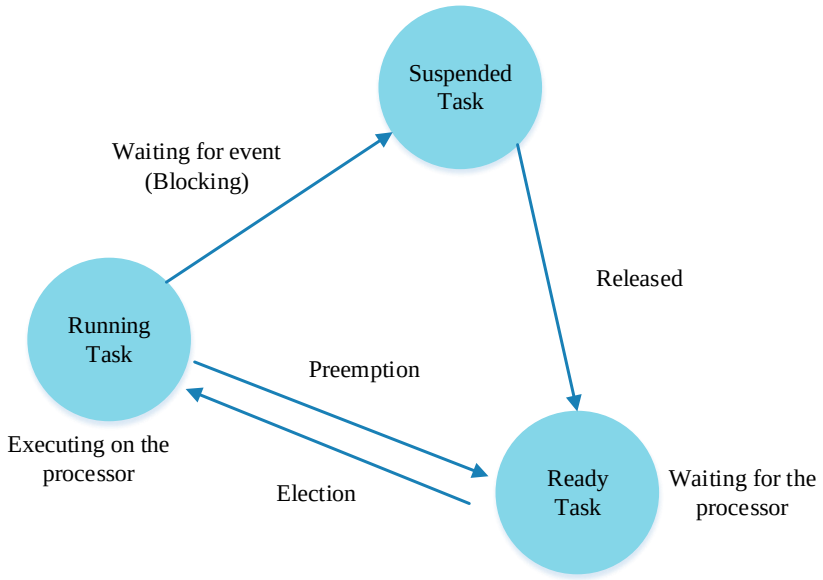

Figure 2.1: *Different possible states for a real-time task*

multitasking environment, at any time, each task can be in one of the following states:

– *Running:* This is the case of the task being executed. When there is only one processor, only one task is running at a given time (this one is chosen according to the considered

scheduling policy and the mode of execution: preemptive or not).

- *Ready:* This is the case of a task waiting for the availability of the processing resources.
- *Suspended:* This is the case of tasks that are waiting for events that will cause them to be released.

Figure 2.1 shows the possible transfers from one state to another. A task (supposed already created) is initially in sleeping state. When it is released, it switches to the ready state where it is waiting to be selected by the scheduler to run. When the scheduler decides to run it according to a given scheduling policy, the task will be allocated to the processor and be executed (active state or running state).
A running task may: (i) be interrupted by another higher priority task and goes into ready state (ii) be waiting for a message, a date, an event or good access to a resource, (iii) be suspended and switch to the blocked state. A task in waiting state for a condition (an incoming message, a date, an event or a release of resource) will be ready to run when the condition will be verified.

Apart from the time periods associated with each task, there are other constraints including:

- *Precedence constraints:* that define a partial order on the execution of tasks. A task with no precedence constraint will be qualified as an independent task.
- *Execution constraints:* that is based on two execution modes: preemptive or non-preemptive. A preemptible task can be interrupted at any time (by a higher priority task) and can be resumed later or immediately on another processor. On the contrary, a non-preemptible task runs from the moment it is elected and retains access to the processor until the end of its execution.
- *Resource constraints:* that result in access to critical resources in mutual exclusion for tasks that want to run.
- *Placement constraints:* that require a task to run on one or more given processors.

Before defining periodic and aperiodic task models, it is necessary to introduce a more general model from which these derive. This basic model is the model of jobs (or task instances).

2.1.3.2 Job Model

A job is characterized by three parameters, as specified in the definition below :

Definition 2.3. *A job is characterized by the triplet* (a, c, d) *where:*

- a *is the release time.*
- c *is the coputation time.*
- d *is the absolute deadline.*

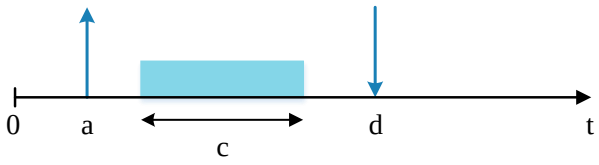

Figure 2.2: *Model of a job*

In other words, a job that arrives at the moment t has required units of execution time that must be assigned to it in the interval $[a, d]$ to respect its temporal deadline (see Figure 2.2).
A job scheduling policy must be implemented. It is in charge of selecting the different jobs to be executed on the system processor. Only active jobs can be scheduled, namely:

Definition 2.4. *A job is said to be active at time t when:*

- *The job has arrived at a time before t* $(a \leq t)$
- *Its deadline is after the moment t* $(t < d)$*,*
- *The job has not finished its execution (less than c units of time have already been executed).*

2.1.3.3 Modeling Real-Time Tasks

As previously stated, the execution of a task gives rise to a set of jobs. There are mainly three types of tasks, depending on how the jobs are enabled:

- *Periodic tasks* are activated regularly according to a fixed period;
- *Sporadic tasks* are irregularly activated but with a minimum duration between the arrival of two consecutive jobs;
- *Aperiodic tasks* are activated in an irregular way without any property that can link the jobs between them.

In this work, we consider the *periodic* case. The cases of aperiodic and sporadic tasks will not be implemented.

2.1.3.3.1 Model of Periodic Tasks Periodic real-time tasks represent tasks whose jobs are recurring with an interval of constant recurrence called period. Several models of periodic tasks have been described and studied in the literature. The simplest and most studied is the one commonly known as Liu and Layland's model [6]. This model consists in characterizing a task by two parameters: C its maximum execution time also called the worst case execution time (WCET), and T its period. Other models are proposed in which we just add parameters to this basic model. A model of periodic tasks a little more sophisticated than the previous one is proposed by [7].
A periodic task $\tau_i(r_i, C_i, D_i, T_i)$ is defined by:

- r_i, the release time of the first job of the task τ_i. In our work, we consider that all tasks are released at time $t = 0$.

– C_i, the worst case execution time (WCET) of each job of the task τ_i.

– T_i, the period of the task τ_i or the duration that separates the arrival of two successive jobs of τ_i.

– D_i, the relative deadlineof the task τ_i also called critical time.

In this characterization, task τ_i makes its initial release at time r_i and its subsequent requests at times $r_i + kT_i$; $k = 1, 2, \cdots$ called release times. The least common multiple of $T_1, T_2, \cdots, T_n$ (called the hyperperiod) is denoted by T_{LCM}. Each request of τ_i requires a worst case execution time of C_i time units. A deadline for τ_i occurs D_i units after each request by which task τ_i must have completed its execution. It is always assumed that $0 < C_i \leq D_i \leq T_i$ for each $1 \leq i \leq n$.
The processor utilization factor or processor load U_{p_i} of the task τ_i corresponds to the activity rate of the processor following the execution of the successive jobs of the task: $U_{p_i} = \frac{C_i}{T_i}$;
The set of parameters is illustrated in figure 2.3:

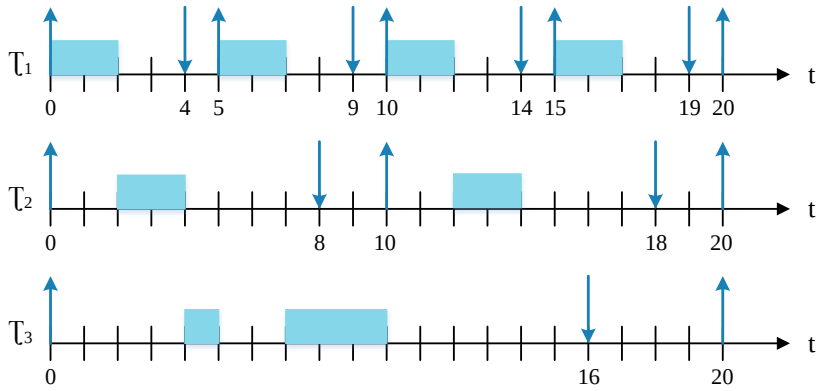

Figure 2.3: *Model of a periodic task*

2.1.3.3.2 Model of Aperiodic Tasks An aperiodic task (or non-periodic task) demands running once. It is produced during a non-recurring event, for example an alarm triggered. As a result, the release time of an aperiodic task is not known beforehand. Aperiodic tasks can be critical or non-critical. A non-critical aperiodic task is not temporally constrained, this means with no deadline and whose execution must be carried out as soon as possible. On the contrary, critical tasks is provided by a deadline and must be completely executed before that deadline.
Critical aperiodic tasks are denoted as follows: A task set $\Gamma = \{\tau_i \mid i = 1, \cdots, n\}$. A four-tuple (r_i, r_i, C_i, D_i) is associated with each task τ_i. Where r_i is the arrival or release time of the task, i.e. the time where τ_i is known by the system. In addition, each request of τ_i requires a worst case execution time of C_i time units. A deadline for τ_i occurs D_i units after each request by which task τ_i must have completed its execution. It is always assumed that $r_i + C_i \leq D_i$ for each $1 \leq i \leq n$.
Figure 2.4 illustrates the model of critical aperiodic task:

2.2 Formulation of the Real-Time Scheduling Problem

A real-time multitasking operating system allows the execution of several programs at once. When several tasks require to run simultaneously on the same processor, several problems arise

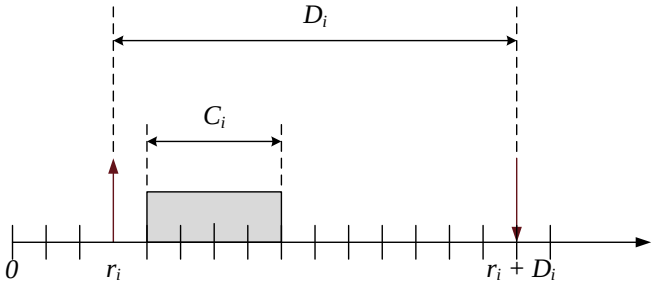

Figure 2.4: *Model of an aperiodic task*

including access to the processor, concurrent access to memory, access to the peripherals, etc. Consequently, we must choose, at every moment, the most appropriate task to run on the processor. To allow this choice, it is necessary to provide a *scheduler* (for the implementation of a scheduling algorithm) allowing the proper functioning of the system, i.e. the respect of the deadlines of the tasks. The scheduler is then the main component of an operating system that chooses the order of execution of the tasks on the processor.

2.2.1 Categories of Real-Time Scheduling

In general, schedulers are classified according to characteristics of the system on which they are located. Some types of schedulers are indicated below:

- *Uniprocessor or Multiprocessor:* Scheduling is performed on a single-processor if all tasks can only run on one and same processor. If multiple processors are available in the system, the scheduling is of type multiprocessor.

- *Offline or online:* The scheduling algorithms can be classified into two categories, namely offline and online scheduling algorithms: *Offline* scheduling is established prior to launching the application to determine a fixed sequence of task execution from all the different characteristics and constraints. Then this sequence is stored in a table and executed online by the processor. An *online* scheduler constitutes of building a dynamic sequence based on the events that occur. However, it relies on data collected by a preliminary analysis of the system carried out off-line to ensure compliance with the time constraints of the tasks.

- *Driven by priority:* A large majority of online scheduling algorithms arrange ready tasks by associating them with a value called *priority*. This is called algorithms driven by priority. Higher is the priority of a task at a given time, shorter will be its processor standby time. We distinguish between schedulers with fixed priorities of those with dynamic priorities according to whether this priority is constant or variable over time.

- *Preemptive or non-preemptive:* A scheduler is *preemptive* if the execution of any task can be interrupted to execute another task that is deemed to have higher priority. On the contrary, if once the task started in progress execution, it can not be interrupted before the end of its execution despite the release of a higher priority task, this scheduling is said to be *non-preemptive*.

– *Idle or non-idle:* A *non-idling* scheduler works without insertion of slack time (non-idling or work-conservative). Therefore, in the case of a non-idling scheduler, if there is at least one task ready and the processor is free, then the scheduler elects the highest priority task and executes it immediately. The idle scheduler works by inserting idle times. Even if the processor is free, there may be a task that waits for a while before being executed based on decision of the scheduler. We will show in the following of this work that the idleness of a scheduler is a necessary property to obtain the best quality of service in a context of energy constraines.

– *Centralized or distributed:* Coulouris [8] proposes the following definition: "A distributed system is a set of autonomous machines connected by a network, and equipped with software dedicated to the coordination of the system activities as well as sharing its resources". On the other side, a system is centralized when the scheduling algorithm for the whole system, distributed or not, is produced on a privileged site of the distributed architecture which contains all the parameters of the tasks.

This thesis focuses on centralized systems restricted to a single processor architecture where preemption is allowed.

2.2.2 Properties of Scheduling Algorithms

The choice of a scheduling algorithm essentially depends on the context defined by the different characteristics of the real-time system on which it is implemented. We quote in this part, some properties and definitions used in the scheduling and feasibility analysis of any real-time application [4].

Definition 2.5. *A scheduling algorithm of a task configuration is valid if and only if all deadlines of tasks are respected.*

Definition 2.6. *A task configuration is said to be feasible if there is at least one scheduling algorithm in which all tasks respect their deadline.*

Definition 2.7. *A task configuration is said to be schedulable if and only if there exists a valid scheduling algorithm for an infinite length.*

Definition 2.8. *The scheduling algorithm ω is optimal in a class of a given scheduling problem, if it can successfully schedule a set of tasks whenever this set is schedulable.*

In other words, if an optimal scheduler fails to build a valid sequence for a given task configuration, then no other scheduler will be able to find a valid sequence for the same configuration. An *schedulability test* makes it possible to determine, before execution, whether a given scheduler will be able to provide a valid scheduling sequence for a given task configuration.
Designing an application of hard real-time type therefore requires that before it starts, that the application is achievable. This is done in the development phase by executing a schedulability test after selecting the scheduler that will be implemented in the operating system.

Definition 2.9. *A scheduling algorithm is said to be clairvoyant when it knows all the features of the tasks to be executed and whose release has not yet taken place. Otherwise, it is said to be non-clairvoyant.*

Definition 2.10. *An algorithm is said to be deterministic if it does not involve any random component in the scheduling decisions. Therefore, a periodic task configuration scheduled several imes by the same deterministic algorithm will present the same task scheduling.*

Definition 2.11. *Consider a configuration of n periodic tasks. The sequence of scheduling for this task configuration, produced by any preemptive scheduling algorithm is always periodic and of period equal to H. H, called hyperperiod, represents the smallest common multiple of periods of the configuration tasks [9].*

2.3 Scheduling of Periodic Tasks

Depending on the specificities of the application, the designer must define a scheduling method in order to respect the temporal constraints of the tasks. In this section, we detail the main classical algorithms for the scheduling of periodic tasks. The algorithms presented are based on the worst case consideration, that is, they evaluate the schedulability of a periodic tasks configuration to a critical instant. The Critical Time of a Task [6] is the time when the request to execute this task arrives simultaneously with the request to execute all higher priority tasks. Therefore, this time corresponds to the longest response time of the scheduler following the execution request of all the tasks.

2.3.1 Scheduling with Fixed Priorities

2.3.1.1 Rate Monotonic Algorithm

The Rate Monotonic (RM) algorithm was introduced by Liu and Layland to schedule a task configuration to implicit deadlines [6]. According to RM, a task has a fixed priority that is inversely proportional to its period. Therefore, RM always gives priority to the job belonging to the smaller task activation period.

RM is optimal in the class of preemptive algorithms with fixed priorities for the scheduling of periodic, independent and synchronous tasks [6]. If any of these conditions is not satisfied, then RM is no longer optimal.

A sufficient condition of schedulability has been proposed by Liu and Layland [6]. Let $U_p = \sum_{i=1}^{n} \frac{C_i}{T_i}$ be the processor utilization factor of a task set Γ of n periodic tasks with deadlines equal to periods. Γ is schedulable by the RM algorithm if U_p verifies the following condition:

$$U_p \leq n(2^{1/n} - 1) \tag{2.1}$$

When analyzing this result, we find that for a large number of tasks, the upper limit of the processor utilization that guarantees a feasible scheduling with RM is about $ln(2) \approx 0.69$. On the

other hand, if tasks are harmonic (all periods are multiple or sub-multiples of others), this limit tends to to 1. Moreover in his experimental study on tasks with periods and random execution costs, Lehoczky et al. [10] show that the RM algorithm is able to schedule configurations of tasks with a CPU utilization limit of around 0.88.

2.3.1.2 Deadline Monotonic Algorithm

The Deadline Monotonic (DM) algorithm was introduced by Leung and Whitehead [11]. DM ranks tasks in ascending order of relative deadlines. DM and RM are some how confused. In DM, the task τ_i with arelative deadline D_i has higher priority than another task τ_j with arelative deadline D_j if $D_i < D_j$. DM is better than RM for periodic tasks whose deadlines are smaller than their periods.

The DM algorithm is optimal in the class of preemptive algorithms, with fixed priorities for scheduling periodic, independent, synchronous and time-bound tasks.

2.3.2 Scheduling with Dynamic Priorities

2.3.2.1 Earliest Deadline First Algorithm

Earliest Deadline First (EDF) is a fixed priority job scheduler with dynamic priorities at the task level. It was presented by Jackson [12]. With the EDF algorithm, the highest priority task at time t is given to the task whose deadline is closest to that time. EDF is mainly used with preemptions: if a very urgent job is released, the job is pre-empted in favor of more urgent. However, EDF can also be used without preemption; This version will not be discussed in this thesis knowing the poor performance of non-preemptive schedulers.

Illustration:
Consider a periodic task set $\Gamma = \{\tau_i(C_i, D_i, T_i) \quad | \quad 1 \leq i \leq 3\}$. $\tau_1(2, 4, 5)$, $\tau_2(2, 8, 10)$ and $\tau_3(4, 16, 20)$. Tasks are synchronous and released at time $t = 0$. The scheduling of the task set Γ according to EDF is illustrated by Figure 2.5:

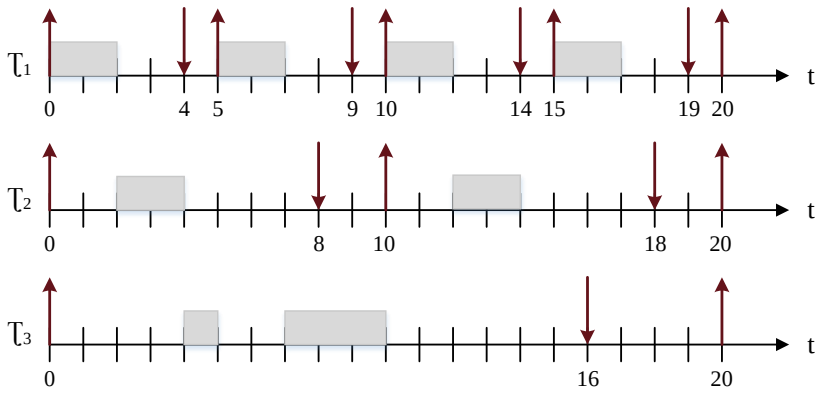

Figure 2.5: *Scheduling by EDF*

EDF is optimal for independent task configurations with deadline constraints ($D_i \leq T_i$) [13].

An EDF *schedulability test* for periodic or sporadic tasks with implicit deadlines is given by the following theorem:

Theorem 2.1. *A configuration of n synchronous periodic tasks with deadlines equal to periods is schedulable with EDF if and only if its processor utilization factor U_p verifies*

$$U_p = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{2.2}$$

EDF is certainly the most popular algorithm. Indeed, EDF has a very strong performance that achieves a CPU utilization factor of 100% under the condition In Theorem 2.1. Moreover this result is also valid for task sets of delayed arrive times, which illustrates the superiority of EDF over RM and DM.
For synchronous periodic tasks with deadline constraints, we will use another test which is proved to be necessary and sufficient. This test incorporates a new criterion, the processor demand $h(t)$ proposed by Baruah et al. in [14].

Theorem 2.2. *A set of n periodic tasks is schedulable by EDF if and only if:*

$$\forall\, t > 0\ h(t) = max\left(0, \sum_{i=1}^{n}\left(1 + \lfloor\frac{t - D_i}{T_i}\rfloor\right) C_i\right) \leq t \tag{2.3}$$

2.4 Conclusion

The purpose of this chapter was to present some basic notions necessary for understanding the context of this thesis. At first, we recalled the general characteristics of real-time systems. Then we described the terminology and concepts related to real time domain and more specifically concerning scheduling. In a second step, we presented the main scheduling algorithms which make it possible to schedule periodic tasks.
In the next chapter, we will introduce real-time systems called autonomous because powered by renewable energy. And we will present the scheduling algorithms specially adapted to the configurations of periodic tasks in these systems.

Chapter 3

Real-Time Scheduling Under Energy Constraints

In this chapter, we are interested in the problem of scheduling tasks characterized not only by an execution duration constrained by time but also by needs in energy to realize this execution. First, we describe the embedded systems and more precisely the wireless sensor networks. At present, these constitute the majority of embedded applications built around autonomous systems from the energy point of view. Then, we focus on the storage and extraction of renewable energy to power these autonomous systems. We describe a state of the art about technologies associated with energy recovery. After that, we describe the problem of energy autonomy in a system often described as energetically neutral. Finally, we describe different scheduling techniques with the aim of minimizing energy consumption and then aiming to perform an energetically autonomous system.

3.1 Embedded Systems

3.1.1 Definition

Wayne Wolf [15] defines an embedded system as any programmable device, without being a computer, even though computers are very often used to build embedded computer systems.
An embedded system is an autonomous electronic / computer system. Autonomous means that it is not attached to the power grid. It is usually composed one or more microprocessors for executing a set of programs that are defined previously during its design and are stored in memory. An embedded system is characterized by its very strong connection with the environment in which it is installed and with which it interacts. This interaction is described as follows:

- The input information of the embedded system comes from sensors.
- The embedded system performs calculations whose results are sent to actuators or output devices such as display screens.

Whatever its nature and complexity, an embedded application integrates a control system and a controlled system. The controlled system is the process or environment that interacts with the control system. The control system is the set of software and hardware elements (microprocessors ...) where the software has a specific delivery function to the application.

3.1.2 Wireless Sensor Network (WSN)

Wireless sensor networks (WSNs) is a research theme in full expansion. They are ubiquitous in many areas, especially intrusion detection, environmental monitoring, medical monitoring, etc. A WSN is made up of nodes that communicate with each other and with a base station. A sensor node is a significant example ofreal-time embedded system, which receives data from its environment via one or more sensors attached to it.

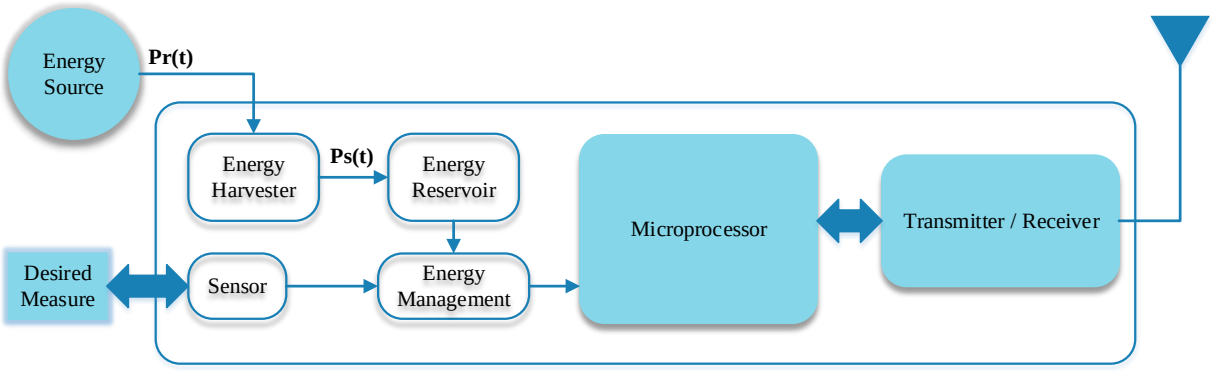

Figure 3.1: *Example of a self-contained wireless sensor system*

Figure 3.1 shows a diagram of a wireless sensor network system. A WSN system is constituted of three main elements: the energy receptor, the energy reservoir and the microprocessor. The energy receptor (for example a photovoltaic cell) has the role of extracting the energy delivered by a source (for example the sun) and then convert it into electrical energy. This electrical energy will recharge an energy reservoir that can be a battery or a super-capacitor and is characterized by its capacity, noted C. We assume that it is ideal in the sense that it can always be charged to its maximum capacity. At any instant t, the energy stored in the reservoir is denoted $E(t)$ and its value is bounded by two constants such as $0 \leq E(t) \leq C$.
The energy received by the energy reservoir during an interval $[t_1, t_2]$ is denoted $E_s(t_1, t_2)$ and is calculated as follows:

$$E_s(t_1, t_2) = \int_{t_1}^{t_2} P_s(t)dt \tag{3.1}$$

Where $P_s(t)$ is the instantaneous power received at instant t including all the energy losses caused by the hardware implementation of the energy recovery process.

The supply of a sensor node (simply called sensor) is a crucial point. Its choice depends not only on the material characteristics of the sensor but also on the different treatments it will have to perform. Most WSNs are used in hostile areas where humans can not easily access them (satellite in space, nuclear environments, etc.). Therefore, recharging or replacing a battery is proved to be difficult and extremely expensive for most applications if not impossible. Hence, the importance of estimating the energy needed to allow a high system performance and ensure its autonomy over a very long period without external intervention.
Thus, we will be able to be satisfied from six months to one year as duration of autonomy for

certain classes of systems such as non-intrusive medical sensors. However, such a period will be considered insufficient for others because of the costs and risks associated with replacing the batteries. This is the reason for which in recent years, we are moving towards a process that consists in never changing the battery knowing that it can be recharged continuously from the environment.
The harvesting of ambient energy is more and more widespread in WSNs where the environment offers multiple sources of energy. This technique involves associating with an energy consumer system, a reservoir of energy for ensuring the temporary storage of energy harvested by the environment.
The introduction of such technology leads us to describe an embedded system in the form of three components: energy harvester (energy receptor), energy reservoir for storing energy (battery and / or super-capacitor) and energy consumer (the computer system).
One of the most important key issues when designing an embedded system is the dimension of its components. We are concerned here in the minimum size of the battery that will allow a perpetual operation of the system taking into account the consumed energy and the energy produced by the environmental source.

3.2 Energy Storage

3.2.1 Energy Storage Elements

The harvested energy is stored in a reservoir which can be a *battery* and / or a *supercapacitor*.

3.2.1.1 Battery

A battery is an electrochemical device that converts chemical energy into energy thanks to a chemical reaction of oxidation-reduction. The electrical energy provided by these electrochemical reactions is expressed in watts (Wh). Batteries are the most widely used energy storage technology for all electronic devices. This is due to the generation of energy-hungry portable devices like digital cameras, camera phones, PDAs, etc. There are 2 types of relatively long lasting batteries: Primary (non rechargeable) batteries compared to secondary (rechargeable) batteries. However, a large-scale adoption would result in important environmental issues. Rechargeable batteries require that the user can access to the electrical grid to recharge them which is not always available even in urban areas.
A battery is characterized by:

- Its *voltage*, expressed in volts (V), which represents the potential of oxidation-reduction between the two electrodes of battery.
- Its *electric charge* in ampere-hour (Ah), which corresponds to the amount of electrons that the battery can hold.
- Its electrical *charging capacity* which represents the maximum charge provided by the battery, corresponding to a complete discharge cycle (between the moment when it is loaded to its full capacity and the moment it is completely discharged).

- Its *cyclability*, expressed in number of cycles, which characterizes the life of the battery, this means the number of times where it can restore a level of energy higher than 80% of its nominal energy.

- Its *mass* (or volume) *energy density*, expressed in watts per kilogram, (Wh/kg) (or in watts hour per liter, Wh/L), which defines the battery life and represents the amount of energy stored per unit mass (or volume) of the battery.

- Its *mass power density*, in watts per kilogram (W/kg), represents the power (electrical energy supplied per unit time) that the battery can deliver.

The use of rechargeable batteries instead of conventional batteries, will thus make it possible to lengthen the life of the embedded systems in which they are integrated. When a classic battery is exhausted, the system is no longer functional and it is said that the sensor is dead [16].
To extend its life time, we would then have to increase the capacity of the conventional battery.. On the contrary, in the case of a rechargeable battery, it is recharged by a natural and inexhaustible energy source such as that provided by a photovoltaic panel. So the application will only use the available energy in the battery when the system is overloaded (i.e. the consumed power of the system is greater than the power transmitted by the source). Therefore the surplus of photovoltaic energy produced during the day will be used to power the system during the night when no energy can be drained from the environment.

3.2.1.2 Supercapacitor

Supercapacitors turn out to be competitive energy reservoirs compared to batteries in the world of small autonomous objects. A super-capacitor is an electrochemical capacitor that has an exceptional energy storage capacity compared to traditional capacitors.
Compared to a rechargeable battery, supercapacitors are characterized by

- Very high power during the charging / discharging cycle.

- Significant cyclability (thousands of charging / discharging cycles)

- A tolerance at low temperatures of up to -40 °C knowing that batteries do not work properly with a temperature below -10 °C.

- The speed of their recharging. A battery may be damaged due to a too fast charge.

Advantages of supercapacitors are: unlimited cycle life (not subject to the wear and aging experienced by the electrochemical battery), low impedance (enhances pulse current handling by paralleling with an electrochemical battery), rapid charging (low impedance supercapacitors charge in seconds), simple charge methods (voltage-limiting circuit compensates for self-discharge; no fullcharge detection circuit needed) and cost-effective energy storage (lower energy density is compensated by a very high cycle count).
On the other side, supercapacitors are unable to deliver the full energy stored since the voltage discharge curve is not at. Other drawbacks are: (i) Supercapacitor cells have low voltages and (ii) They have low energy density and high self-discharge.

3.3 Problem of Autonomy of Embedded Systems

In this paragraph, we introduce a new terminology.

3.3.1 Need for New Terminology

3.3.1.1 Scheduling with energy clairvoyance:

A scheduler is energetically clairvoyant when a priori knowledge of the amount of energy is produced by the source that feeds the system. In the contrary, the scheduler is said not clairvoyant from the energy point of view.

3.3.1.2 Scheduling with time clairvoyance:

A scheduler is temporally clairvoyant when it knows a priori all the characteristics of the future tasks to be performed.

3.3.1.3 Total clairvoyant scheduling:

A scheduler is totally clairvoyant if it has both a time clairvoyance and energy clairvoyance.

3.3.1.4 Temporally feasible scheduling:

We say that scheduling is temporally feasible for a given task set Γ if there is at least one scheduler capable of producing a temporally valid sequence to satisfy all the timing constraints without taking into account its energy constraints.

3.3.1.5 Schedulable task configuration:

A task configuration is schedulable if it exists at least a scheduler able to create a scheduling sequence that satisfies all the temporal and energy constraints. On the contrary, it is said to be non-schedulable when it fails to respect either its temporal constraints or its energy constraints.

3.3.2 Need for a Task Model Adapted to Energy Constraints

Consider a real-time periodic task set Γ. Every task τ_i is characterized by (C_i, D_i, T_i, E_i), where C_i is the worst case execution time (WCET), D_i is the relative deadline, T_i is the period and E_i is the worst case energy consumption (WCEC). Periodic tasks are assumed to be preemptible, independent and initially synchronized (all tasks are released at time $t = 0$).

3.3.3 Need for Specific Scheduling Policies

Consider a task set Γ consisting of three independent periodic and preemptable tasks. $\Gamma = \tau_i(C_i, D_i, T_i, E_i)$. $\tau_1(2, 4, 5, 5)$, $\tau_2(2, 8, 10, 10)$ and $\tau_3(4, 16, 20, 10)$. Tasks are synchronous and released at time $t = 0$. Γ is scheduled according to EDF on a processor powered by a battery with nominal capacity $C = 10$. This battery is recharged by an environmental source. It receives constant power $P_s(t) = P_s = 2$. Figure 3.2 illustrates the scheduling of this configuration according to EDF. We notice that at time $t = 5$, the energy in the battery is exhausted. Ready tasks are

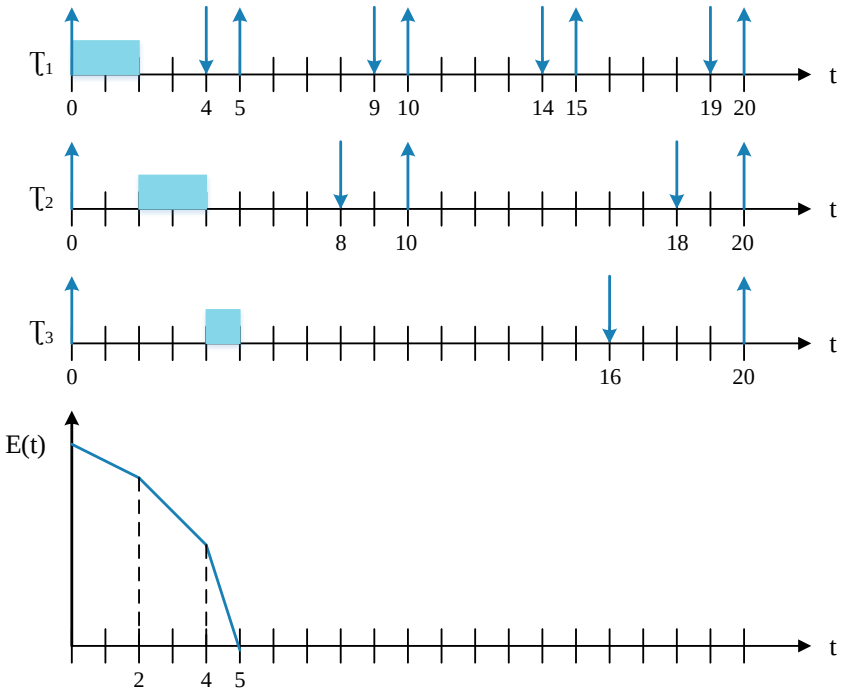

Figure 3.2: *Scheduling by EDF under energy constraints*

then interrupted by missing energy. It is therefore necessary to provide a time for recharging the battery and then being able to continue the execution of the tasks. However, this time lost during recharging of the battery must be calculated very carefully to prevent the violation of the deadlines of the tasks and the collapse of the system. In summary, EDF is not an energetically clairvoyant algorithm and is not adapted to this type of real-time system.
The energy constraint adds a new dimension to scheduling problems. We must find an algorithm capable of performing all tasks in the required time and without running out of energy. Thus, in the framework of this thesis, we must consider two constraints together: the temporal constraint and the energy constraint. Tasks can be performed if there is enough energy in the reservoir that is recharged by a source of energy knowing that it is necessary to guarantee their executions while still respecting their temporal constraints. New types of schedulers are proposed in the literature for Monoprocessor real-time systems subject to time and energy constraints. These scheduling techniques must create a valid sequence of the task configuration that constitutes the application. When the scheduling sequence is valid, the it is valid from both time and energy point of view. We will first explore various existing techniques aimed at to schedule a task configuration while satisfying their temporal and energy constraints.

3.4 Existing Scheduling Policies

3.4.1 Approaches to Minimize Energy Consumption

To reduce the energy consumption of an embedded system, two types of methods exist.

3.4.1.1 Dynamic Power Management (DPM)

DPM can dynamically manage the activity of the system by making switches from a sleep state to an active state and vice versa [19]. DPM method can reduce power consumption of the system without significantly degrading the performance by switching to standby mode when there is no running task and then switch to the active mode when the processor is requested. These methods use processors that have a sleep function. Therefore, the processor is temporarily turned off when necessary. DPM method will consume a little or no energy (called static energy) in this sleep state. For example, the Intel 80200 processor has three modes of operation, including two energy-efficient modes that differ in the number of components.

3.4.1.2 Dynamic Voltage and Frequency Scaling (DVFS)

The method known as Dynamic Voltage and Frequency Scaling (DVFS) allows the processor to change the frequency when necessary. DVFS method uses processors designed to reduce the used energy by varying the supply voltage and therefore the frequency of operation [19]. Thus, if the processor frequency is reduced, the job in execution will increase its execution time. For example, if the frequency is halved, the job will take twice as long to complete its execution. Since energy consumption is a quadratic function of frequency, the fact of reducing it significantly impacts energy consumption. In this context, saving energy is achieved by stretching the execution times, which must be controlled so that we continue to respect the temporal constraints. Therefore, using this category of processors, there will be possibility to slow down the frequency of operation and thus slow down the current task so as to reduce its energy consumption while satisfying its temporal constraints.
Authors have been interested in reducing the energy consumption of the system. Techniques for the reduction of energy consumption using the DPM method are studied by Benini and al. in [20]. In addition, using the DVFS technique, Yao et al. [21] proposed an algorithm that calculates the optimum energy operating frequencies for a set of periodic tasks. The underlying scheduling policy being EDF, this strategy is also optimal from a scheduling point of view. Then, Aydin et al. [22] proposed an approach based on DVFS technique and the necessary and sufficient condition proposed in [6]. They assume that power received by the source is constant. Their algorithm minimizes the energy consumption of periodic tasks by guaranteeing their execution before their deadline. Techniques have also been developed for periodic and aperiodic tasks to reduce the energy consumption of the system [23], [24] .

3.4.2 Approach to Energy Autonomy

In this part we quote some existing algorithms in the literature that make it possible to schedule a set of periodic tasks on a constant speed processor so as to make good use of the energy available in the system while satisfying their temporal and energy constraints. The methods that we describe here are therefore adapted to the model described previously.

3.4.2.1 Optimal LSA Scheduling Algorithm

The Lazy Scheduling Algorithm (LSA) [25] is an on-line scheduling algorithm. It allows to schedule any set of periodic or aperiodic critical tasks according to EDF. These tasks are performed by a monoprocessor powered by a energy reservoir that is fed by a renewable energy source. This energy reservoir receives a instantaneous power $P_s(t)$ that can vary over time.
Each time a task is released, the scheduler calculates a start date specific to this task. This time represents the moment from which the task can begin to run on the processor using the maximum power consumption P_{max} during its execution. Between the arrival time and the start time, the processor is intentionally left in standby to allow the reservoir to recharge. This recharge time interval is calculated by ensuring that the processor has sufficient energy to complete the task in accordance with its deadline. LSA is a clear-sighted point algorithm energy perspective; so it assumes to have a knowledge, at least for a near future, of the quantity of the harvested energy. However, as it does not require to know a priori the time of future arrival of tasks. LSA is not clairvoyant from a temporal point of view.
Consider a single-processor architecture characterized by power consumption P_{max}, in charge of executing a task set. A task $\tau_i(r_i, C_i, D_i, T_i, E_i)$ is characterized by its execution time C_i, its deadline D_i and its energy demand E_i. If LSA can not reliably schedule this task set, then no other algorithm can do it, even if it is totally energetically clairvoyant.
Even if LSA is optimal, it has some disadvantages. Indeed, the assumptions made about the configuration of tasks are very restrictive. The energy consumed by a task is assumed to be proportional to its execution time ($E_i = k.C_i$). However, the total energy that a task consumes during its execution is not proportional to the duration of its execution. This energy depends in particular on the different electronic circuits which the task needs during its execution.

3.4.2.2 EDeg Scheduling Algorithm

3.4.2.2.1 Principle of EDeg: The algorithm EDeg (Earliest Deadline with Energy Guarantee) [26] is a variant of the EDF and the EDL scheduling algorithms for scheduling periodi. We consider a system composed of a single processor that is powered by a reservoir of energy. This energy reservoir is recharged by a renewable energy source. The processor must be able to perform a strict real-time periodic task configuration while considering their energy demands, their deadlines and the energy available in the reservoir.
Thus, at every moment, the task with the closest deadline among the ready tasks is executed only if the reservoir is not empty and there is enough energy in the system so that the execution of the task compromises the execution of future periodic tasks. When there is not enough energy in the system or the reservoir is empty, the processor must be in idle mode (or standby mode) for a period of time (t_{idle}) to recharge the battery. t_{idle} is calculated by using the EDL algorithm.

Feasibility Test: Consider a configuration of n strict real-time periodic tasks where each task τ_i is characterized by its worst-case execution time C_i, its relative deadline D_i, its period T_i and its energy demand E_i. The energy reservoir has a capacity C and the instantaneous power received by the reservoir is given by $P_s(t)$ which takes into account all the losses due to the conversion.

Theorem 3.3. *A task set* Γ *of n periodic real-time tasks is schedulable with EDeg only if :*

$$U_p = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{3.2}$$

and

$$U_e = \sum_{i=1}^{n} \frac{E_i}{T_i} \leq \overline{P_s} \tag{3.3}$$

U_p is the processor utilization factor, U_e is defined in [26] as the energy utilization and $\overline{P_s}$ is the average of the power received over the lifetime of the application.

3.4.2.2.2 Description of the Algorithm: Before stating the algorithm of EDeg, let us define the following notations:

– $SlackEnergy(t)$ is called the energy laxity (slack energy) of the system calculated at the current time t. This variable represents the maximum amount of energy that can be consumed from time t while satisfying all the temporal constraints of the tasks. If a task τ_i is ready and elected at a time t, if the battery is non-empty and if the energetic laxity of the system is positive, the task can be executed. The energy laxity of the system at time t corresponds to the minimum energy laxity among those tasks with their release time greater than the time t and their deadlines less than d (i.e. the absolute deadline of the highest priority task according to EDF).
The energy laxity of the k^{th} job of the task τ_i ($\tau_{i,k}$) at time t is calculated as follows:

$$SlackEnergy(\tau_{i,k}, t) = E(t) + \int_{t}^{d} P_s(x)dx - A_{i,k} \tag{3.4}$$

Where $A_{i,k}$ is the sum of the energy demands required by the jobs released after $r_{i,k}$ and deadline less than or equal to $d_{i,k}$ such that

$$A_{i,k} = \sum_{r_j \geq t, d_j \leq t} E_j \tag{3.5}$$

– $SlackTime(t)$ is the time laxity (slack time) that is the maximum idle time available from the time t. It is calculated with the EDL algorithm which allows to delay at the latest periodic jobs while guaranteeing the respect of their deadlines.

– PENDING is a Boolean variable that is true when there is at least one task ready in the ready queue or false otherwise.

The EDeg algorithm is stated as follows (1):

3.4.2.2.3 Performance of the EDeg Scheduler: The EDeg scheduler has been the subject of a comprehensive performance study published in [26] and [27]. This study made it possible to highlight the superiority EDeg on the conventional EDF scheduler. Notably, it is shown that the EDeg heuristic allows scheduling a hard real-time task configuration given a renewable energy

```
1: procedure Earliest Deadline with Energy Guarantee (EDeg) Algorithm
2:     while (1) do
3:         while PENDING=true do
4:             while (E(t) > E_min and SlackEnergy(t) > 0) do
5:                 execute()
6:             while (E(t) < E_max and SlackTime(t) > 0) do
7:                 wait()
8:         while PENDING=false do
9:             wait()
```

Algorithm 1: Earliest Deadline with Energy Guarantee (EDeg) Algorithm

source with variable energy production and with a limited capacity energy storage unit. Through simulations in [26], EDeg shows real performances compared to EDF variants in terms of the number of configurable task sets according to the size of the battery, the rate of wasted energy, the energy consumption profile and the average idle time of the processor. EDeg allows a reduction of the capacity of the battery from 1.84 to 3 times from that required with EDF according to the system that is weakly, moderately, or heavily loaded.

3.5 Energy Saving-Dynamic Voltage and Frequency (ES-DVFS) Algorithm

The goal from the ES-DVFS [83] is to schedule aperiodic jobs as soon as possible according to earliest deadline first (EDF) in the presence of variability in dynamic execution behavior. We use a modified EDF strategy to reduce the CPU energy consumption by using the dynamic voltage and frequency selection. ES-DVFS uses an on-line speed reduction mechanism to minimize the system-wide energy consumption by adapting to the actual workload. ES-DVFS still guarantees that all deadlines are met.

3.5.1 Computing the Minimum Constant Speed for Each Job

The ES-DVFS scheduler maintains a priority job queue in which jobs are ordered by the EDF basis. In the beginning, the job queue is empty. Scheduling decisions are only applied when any of the following events really arrive: 1) Event 1: a new aperiodic job is ready and is added to the job queue. 2) Event 2: the current job completes its execution.

We use a dynamic-priority assignment approach where jobs are executed by a variable speed processor. Hence, the worst case execution time (WCET) of each job varies depending on the processor slowdown factor under different speed levels. The challenge is how to essentially build an optimal speed schedule which leads to the maximum energy saving. Our technique is based on the assumption that the parameters of each job are only known when it is released. ES-DVFS attempts to allocate the maximum possible amount of slack time based on jobs presented in the ready job list. When Event 1 occurs, the ES-DVFS updates the optimal speed schedule of the processor for all the jobs including the new one in the job queue and consequently the slack time is updated. Further, when an Event 2 occurs, the completed job is removed from the job queue

and we execute the job with the highest priority following a new calculated slowdown factor.

When preemption occurs, we implicitly calculate the new processor speed to the favor of the newly dispatched job. It is formally proved that the jobs will still meet their deadlines if the speed is changed according to the jobs found in the job queue. This is due to the fact that we keep track of the remaining execution times of the preempted job in the queue, and consequently the worst-case workload is still the same.

In aperiodic real-time system, we are not able to reveal how jobs actually arrive. Thus, we cannot know the maximum deadline or the worst case execution time of the jobs before beginning the schedule. Instead, we would like to apply on-line DVS scheduling algorithm to make scheduling decisions only when jobs really arrive, i.e., only for jobs in the job queue.

Based on ES-DVFS, scheduling decisions at time t are as follows: ES-DVFS selects the job with the earliest deadline J_i and then adopt the speed such that the job has to be finished exactly at the release time of the next job. This means that J_i is executed at speed $S_i = C_i/d_i - t$ where a job J_i is associated with worst-case execution time (at S_{max}) C_i and absolute deadline d_i.

However, if we do not consider the maximum intensity of all the ready jobs, it is possible that the required execution speed, at some moment t, of the resulting schedule might miss future deadlines. Therefore, in order to guarantee that we still meet all the jobs deadlines in the ready queue $\mathcal{Q}$, the workload h_k and the intensity I_j must be verified in advance by considering the highest speed between h_k and I_j .

The intensity at time t is calculated thanks to the following equation:

$$I_j = \max_{J_j \epsilon \mathcal{Q}} \left(\frac{\sum d_i \leq d_j C_i}{d_j - t} \right) \tag{3.6}$$

In other words, J_i is executed at speed $S = \max(I_j, h_k)$. Where h_k is the workload of jobs in the ready queue $\mathcal{Q}$. This means

$$h_k = \sum_{J_i \epsilon \mathcal{Q}} \frac{C_i}{d_{max}} \tag{3.7}$$

where d_{max} is the maximum deadline in $\mathcal{Q}$.

The ES-DVFS algorithm provides sound dynamic speed reduction mechanisms. We integrate DVFS techniques with EDF scheduler for aperiodic real-time applications that potentially use uniprocessor devices during execution. ES-DVFS provides an exact energy management technique as function of the CPU frequency in such a way that time constraints are still met. Using this framework, the speed of the jobs ready to be executed is dynamically adjusted on the fly.

As mentioned above, when a job arrives, it is added to so called job ready queue $\mathcal{Q}$. At any time t, there is a single job J_i eligible for execution. Thus, before executing this job, we should use the minimum CPU speed available to stretch out the WCET as much as possible without violating deadlines. Therefore, job J_i must be executed with a speed S equal to the total workload h_k in $\mathcal{Q}$.

Note that stretching out a job J_i at a time t with speed h_k may lead to deadline violations. In this case, using the speed $S = \max(I_j, h_k)$ will result in a total effective workload which is equal to 1. Hence, ES-DVFS can achieve up to 100 percent CPU utilization where all the jobs are completed before their deadlines.

3.6 Conclusion

In this chapter, we first stated the main principles of embedded systems with an attention on wireless sensor networks. We then recalled the principle of the EDeg algorithm that allows the scheduling of a configuration of periodic tasks without losses subject to time and energy constraints. Later, we introduced an optimal energy efficient scheduling algorithm, named ES-DVFS, to reduce the CPU energy consumption. Unlike prior studies, ES-DVFS algorithm provides sound dynamic speed reduction mechanisms. In the next chapter, we will depend on ES-DVFS to produce a new framework for energy efficient real-time systems with fault recovery mechanisms.

Chapter 4

Fault-Tolerant Real-Time Systems

In this chapter, we are interested in studying the problem of fault-tolerance in real-time systems. Scheduling tasks characterized not only by an execution duration constrained by time but also by needs in energy to realize this execution. First, we describe the embedded systems and more precisely the wireless sensor networks. At present, these constitute the majority of embedded applications built around autonomous systems from the fault occurence point of view. We describe a state of the art about technologies associated with fault recovery. After that, we describe the problem of fault-tolerance in a system often described as energetically neutral.

4.1 Introduction

Fault tolerance is a method of accomplishing a continuous system service within the presence of active faults [28]. Numerous fault tolerance techniques were proposed and implemented within the last 30 years. Some techniques are primarily based on single model software, and might only be powerful with hardware faults and transient software faults. One example is Rollback/Retry, additionally called "checkpoint and restart" [29]. In this approach the detection of errors triggers a system rollback to a previously stored state and a re-execution of the same processing. This method is primarily based on backward errors restoration and desires an efficient error detection mechanism. Different techniques apply hardware redundancy to detect and mask errors, as Triple Modular Redundancy (TMR) [30], in which error detection is executed via contrast of the consequences of multiple hardware/software devices.

Also, fault tolerance is the opportunity to extend operating despite the failure of a definitive subset of their hardware or software. There can be either hardware fault or software fault, which distracts the real time systems to approach their deadlines. They can be categorized as hard real-time systems, in which the results of missing a deadline may be disastrous, and soft real-time systems, in which the consequences are nearly smaller. Some examples of hard real-time systems are a space station, a radar for tracking missiles, a system for monitoring a patient in critical condition, etc. In these real-time systems, it is important that tasks finish before their deadline even in the existence of processor failures. This makes fault tolerance a basic concern of hard real-time systems.

The demand for complex and developed real time computing systems continues to grow where fault tolerance and energy are essential requirements that are playing a critical role in the design of new real-time systems. The fault tolerance is known as the ability of the system to react with its

specification regarding the presence of faults in any of its elements . Fault tolerance for transient type can be approached through task re-execution (time redundancy) whereas time as well as space redundancy are required to tackle permanent or intermittent type faults. Transient faults are more known as compared to permanent ones whereas frequency of appearance of intermittent faults is between these two . Thus, tolerance to transient kind of faults is the ultimate requirement for portable devices which can travel broadly around the globe and suffer several environmental effects leading to more chance of failure.

Moreover, fault tolerance is the real estate that enables a system to appropriate with its correct operation even in the existence of faults (errors), and it is broadly achieved by fault detection and consecutive system recovery [31], [32], [59], [35]. Fault tolerance has been a subject of research for a long time, and meaningful amount of work has been composed over the years [32], [35], [36], [37], [38], [39], and [40]. To achieve fault tolerance, systems are commonly constructed such that some repetition is included. The common types of redundancy used are information, hardware, and time redundancy.

Fault tolerance is provided by error-detecting and error-correcting codes while adopting information redundancy, i.e. the data contains extra information (check bits) that can confirm the correctness of the data before it is used (error detection), or even correct inaccurate data bits (error-correction). Various error-detecting and error-correcting codes have been suggested including parity codes, cyclic codes, arithmetic codes etc. [41], [42], [43]. The main disadvantage of error-detecting and error-correcting codes is that they are restricted to errors that take place concurrently with transfer of data (system bus) or errors in memory.

Furthermore, Fault tolerance is managed by time redundancy. However, fault tolerance methods that consume time redundancy are only effective if the faults are of transient type, i.e. errors that take place, but die out after a short period of time. These transient faults frequently result in soft faults. The easiest technique that adopt time redundancy deals with soft errors by performing the same program twice, and it get the appropriate result if the outputs of the two executions are equivalent. Roll-back Recovery with Checkpointing (RRC) is a famous fault tolerance technique that correctly manage with soft errors. RRC has been the attraction of researchers for years [44], [45], [46], [47], [48], [49].

A different approach to recovery block execution is checkpointing, where essential state information is saved periodically during task execution while error checking procedures are run simultaneously. If an error is detected, the system state is rolled back to the last checkpoint and the computation is repeated. Hence it may minimize the amount of recovery overhead compared to the recovery block approach, checkpointing has two disadvantages: expanded runtime overhead due to the regular checkpointing during the fault-free execution and the failure to manage with cases where the error stems from the unique software employment of the task. With recovery blocks, the designer can provide alternate implementations of the same task in the form of different recovery blocks and these can be activated if the error continues.

Unlike classical re-execution schemes where the task (job) is re-executed if an error is detected, RRC deal with soft errors by benefiting of previously saved error-free states of the task, assigned to as checkpoints. During the execution of a job, the task is interrupted and a checkpoint is captured and stored in a memory. The checkpoint involves enough information such that a task can efficiently restart its execution from that interrupted point. For RRC it is critical that each

checkpoint is error-free, and this can be happen by, for example executing acceptance tests to authenticate the exactness of the checkpoint. Once the checkpoint is reserved in memory, the task carries on with its execution. As soft errors may occur at any time during the execution of a task, a fault detection process is used to detect the presence of soft errors. There are different error detection mechanisms that can be used, e.g. watchdogs, duplication schemes etc. [50], [51], [52]. In case that the error detection mechanism detects an error, it forces the task to roll-back to the latest checkpoint that has been stored.

Relying at the implementation, one-of-a-kind RRC schemes exist, and that they vary among every different primarily based on the following key factors. The first component is how plenty data is saved at every checkpoint. With appreciate to this, there are distinctive RRC schemes, i.e. full checkpointing [53], [54], [55], and incremental checkpointing. In a full checkpointing scheme, at each checkpoint the whole state of the task is saved, whilst in an incremental checkpointing scheme most effective the modifications with respect to the most latest saved state are stored.

Another key factor pertains to when checkpoints are taken. With admire to this, there are different RRC schemes, i.e. equidistant checkpointing [54], [55], and non-equidistant checkpointing scheme [56], [57], [58]. In equidistant checkpointing, the checkpoints are arranged equally during the execution of the task (which means that the distance between two successive checkpoints is always the same), while in non-equidistant checkpointing, the checkpoints are not calmly allotted throughout the execution of the task (the gap among two successive checkpoints isn't always continually the same). As shown on this phase, using fault tolerance is frequently associated with adding an overhead that can result in: higher hardware value, higher energy intake, and even affect (degrade) system's overall performance. Consequently, there need to be a clean aim to what extent fault tolerance is needed for a selected system. Minimizing the disadvantage due to using fault tolerance normally requires optimization of the fault tolerance method which is used. The optimization desires for a given fault tolerance method can also vary some of the specific classes of computer systems in which fault tolerance is employed. In standard, computer systems are categorized into non-real-time and real-time systems relying on the requirement to fulfill a given time constraint.

From the fault tolerance opinion, transient faults and intermittent faults demonstrate themselves in a similar manner: they happen for a short time and then disappear without causing permanent damage. Hence, fault tolerance techniques against transient faults are also usable for tolerating intermittent faults and vice versa.

4.2 Background on Fault Tolerance

Faults are organized as: development, physical and interaction faults [59]. Based on persistence faults can further be categorized as permanent, intermittent, and transient [60]. Faults can occur in hardware or/and software. Researchers of the real-time and fault tolerance community have described the crucial need for fault tolerance in real-time systems. In [61], authors stated that "a real-time system can be viewed as one that must deliver the expected service in a timely manner even in the presence of faults". In [62], authors declared the need for fault tolerance in real-time systems by expressing that real-time systems must be sufficiently fault-tolerant to withstand losing large portions of the hardware or the software and still perform critical functions.

There are several different ways in which a program can be developed using formal rules which guarantee that it will satisfy a specification when executed on an fault-free system. As mentioned lately, tasks in real-time systems are, by definition, critical in nature. In applications a well-known as generation shuttles and nuclear power plant controllers, it is vitally important that all tasks approach their deadlines under all circumstances. However, when a component of a computer system fails, it will usually produce some undesirable effects and it can be said to no longer behave according to its specification. Such a breakdown of a component is called a fault and its consequence is called a failure.

- **Failure:** A system failure takes place when the service supported by the system differs from the stated service. For instance, when a user cannot read his stored file from computer memory, then the suggested service is not supported by the system.
- **Error:** An error is a disruption of internal state of the system that may lead to failure. A failure happens when the erroneous state causes an incorrect service to be delivered, for instance, when certain portion of the computer memory is broken and stored files therefore cannot be read by the user.
- **Fault:** The cause of the error is called a fault. An active fault leads to an error; in another way the fault is inactive. For instance, impurities in the semiconductor devices may lead computer memory in the long run to behave erratically.

A system fault takes place when a conveyed service deviates from the desired service. In other words, a system fails when it cannot support the required service [67]. Even a completely designed computer system can be subject to dissimilar faults and therefore fail erratically. As shown in [68], processor faults can be broadly classified into two groups: transient and permanent faults. Transient faults, also named soft errors, are often caused by electromagnetic interference and cosmic ray radiations. They may cause errors in calculation and dishonesty in data, but are not continuous. On the other side, permanent faults, also called hard errors can cause hardware damages to processors and bring them to stop permanently. According to [69] and [70], transient faults happen more frequently than permanent faults. As real-time computing systems persist to grow quickly in both scale and complexity, maintaining high reliability becomes an increasingly challenging issue. Faults in real-time systems that are not subscribed properly in a timely fashion will cause to violations of timing constraints, which can cause disastrous results if the systems are safety-critical, e.g. aircraft, nuclear power plant. Furthermore, supporting fault-tolerance features (the property that enables a system to continue operating properly in the event of failure(s)) to achieve high reliability is particularly attempt after in such systems. Traditional fault-tolerance techniques to treat with faults composed of two parts, i.e. fault detection followed by fault recovery . Examples of techniques that can reveal the processor faults timely and effectively are listed below:

1. A fail-signal processor to send notifications to other processors when faults occur.
2. Watchdog processors for concurrent control flow checking.
3. Signatures that can be used for detection of hardware and software faults.

4. Sanity or consistence checks.

Instead of duplicating the execution of the whole program, checkpointing in concurrence with backward error recovery is also a well-known fault-tolerant approach. Checkpointing denotes to the scheme that determines system states after a period of time and stores a snapshot if no fault is adjusted. In case of a fault detection, the system rolls back to its pervious correct state. Hence, that checkpointing is a special passive replication scheme. Active replication schemes usually need extra system resources, e.g., processing cores, and absorb more energy even under the fault-free events, but they can sustain run-time faults timely and immediately. On the other side, passive replications are only involved in the event of run-time failure(s), and so, does not consume system resources when no faults take place. However, passive replications hold longer to recover from faults and put the system at risk when timing constraints are very strict. The choice of the desired replication schemes for different hard real-time systems is a design decision problem and requires accurate examinations.

Although, it is imperative to explore advanced techniques to provide the timeliness in the presence of faults for real-time systems. Moreover, both fault tolerance and energy reduction are basically achieved by investing system slack time, therefore they are two contradictory goals in nature.

Nowadays, no general technique can be suggested to add fault tolerance in a system. It rely on the requirements of the application . Fault tolerance can be implemented using two approaches, i.e. hardware and software fault tolerance. A fault may occur sporadically, or it may be stable and cause the component to fail permanently. Even when it occurs instantaneously, a fault such as a memory fault may have consequences that manifest themselves after a considerable time. Suggestions for real-time systems that can tolerate faults are the MAFT system [63] , MARS system [64], Maruti operating system [65], and HARTS [66].

A real-time system may be unsuccessful to function accurately either because of faults in the hardware and/or software (physical faults) or because of not behaving in time (timing faults) due to overwhelm conditions. Therefore, to prevent the catastrophic results of missing deadlines, it is fundamental that real-time tasks meet their deadlines even in the presence of faults and/or overwhelm circumstances.

4.3 Fault Tolerant Techniques

The error detection and fault-tolerance techniques are part of the software architecture. The software architecture, including the real-time kernel, error detection and fault-tolerance mechanisms are considered fault-tolerant. We use several mechanisms for sustaining faults: equidistant checkpointing with rollback recovery and active replication. Rollback recovery utilizes time redundancy to tolerate fault appearances. Replication supports space redundancy that permits to spread the timing overhead among several processors. On one hand, replication is exposed to correlated faults, whereas checkpointing can detect them. On the other side, an error might be present undetected in a checkpoint, which might necessitate rollback . Once a fault is detected, a fault tolerance technique has to be mentioned to handle this fault. The elementary fault tolerance technique to retrieve from fault occurrences is re-execution . Re-execution is a technique that

exploits the slack time on the processor to have recovery tasks, which are used to enhance the reliability of original tasks. Also in re-execution, a process is executed again if affected by faults. The time needed for the detection of faults is considered for by the error-detection overhead. When a process is re-executed after a fault detection, the system restores all initial inputs of that process. The process re-execution operation requires some time for this that is captured by the recovery overhead. In order to be restored, the initial inputs to a process have to be stored before the process is executed first time.

Scheduling is also one of the methods to sustain fault in time critical applications. It is consumed to overcome the drawback of check-pointing in distributed environment. It is classified as time-sharing scheduling, space-sharing scheduling, and hybrid scheduling (combination of both time as well as space). Scheduling is used for load balancing as well as fault tolerance in a system on the basis of space or time sharing.

There are many fault-tolerant techniques such as dual/triple modular redundancy and check-pointing with rollback mostly used in treating the occurrence of faults. Dual/triple modular redundancy is frequently considered to achieve reliability against transient faults in multicore platforms, where multiple processing units perform identical copies for each task and their results are voted on to produce a single output. However, fault tolerance methods that consume time redundancy are only effective if the faults are of transient type, i.e. errors that take place, but die out after a short period of time. These transient faults frequently result in soft faults. The easiest technique that adopt time redundancy deals with soft errors by performing the same program twice, and it get the appropriate result if the outputs of the two executions are equivalent. Roll-back Recovery with Checkpointing (RRC) is a famous fault tolerance technique that correctly manage with soft errors. Checkpointing with rollback recovery is a well-effective technique to endure transient faults. Hence, it acquire meaningful time and energy overheads, which go emaciated in fault-free execution states and may not even be feasible in hard real-time systems. Transient faults in fundamental hardware. Checkpointing with rollback-recovery is a comparatively, cost-powerful technique against transient faults. Checkpointing was first suggested for database systems where availability is the primary measure , yet in real-time systems, other essentials such as predictability and timeliness as well as energy consumption should be considered. Although checkpointing enlarge execution time in the lack of faults, it decreases recovery time when faults take place. This is because there is no need to re-execute the whole task, but only the part of the task beginning from the last checkpoint is needed to be re-executed.

Checkpointing with rollback-recovery is a relatively, cost-powerful technique against transient faults. Checkpointing was first suggested for database systems where availability is the primary measure, yet in real-time systems, other essentials such as predictability and timeliness as well as energy consumption should be treated. Although checkpointing increase execution time in the lack of faults, it decreases recovery time when faults take place. This is because there is no need to re-execute the whole task, but only the part of the task beginning from the last checkpoint is needed to be re-executed. At each checkpoint, the system saves its state in a secure device. When a fault is detected, the system rolls back to the most recent checkpoint and resumes normal execution. Checkpointing increases the task execution time, and in the absence of faults, it might cause a missed deadline for a task that completes on time without checkpointing. In the presence of faults, however, checkpointing prohibits the need for task restarts and increases the probability

of a task completing on time with the correct result. Common checkpointing reduces re-execution time due to faults but increases task execution time. On the other side, checkpointing has less effect on task execution in the absence of faults but increases the amount of re-execution that must be achieved after a fault is detected. Therefore, the checkpointing interval, i.e., the duration between two consecutive checkpoints, must be attentively chosen to balance checkpointing cost (the time needed to perform a single checkpoint) with the re-execution time.

The time overhead for re-execution can be decreased with more complex fault tolerance approaches such as rollback recovery with checkpointing. The main principle of this approach is to restore the last non-faulty state of the failing process, i.e., to recover from faults. The last non-faulty checkpoint, has to be stored in advance in the static memory and will be restored if the process fails. Checkpointing provides an essential technique to reduce re-execution time in the presence of faults. DVS can also be used to enhance fault tolerance in a real-time system. If faults occur commonly, the processor speed can be scaled up dynamically.

4.4 Previous Work

Fault-tolerant computing indicates to the correct execution of user programs and system software in the existence of faults . It is commonly accomplished through task re-execution or component redundancy. In real-time embedded systems, it is important to ensure that task re-execution does not jeopardize the timely deadline of tasks. Fault tolerance is frequently achieved in real-time systems through online fault detection, checkpointing, and rollback recovery.

The primary attract of real-time scheduling is to submit deterministic guarantees to timing constraints in hard real-time systems through schedulability analysis. One efficient behavior is to design the utilization bound of a program in such a way that the system is deemed to be schedulable if this bound is never exceeded. Scientists in both scholarly world and industry have depended on different systems to limit vitality utilization in registering systems. A large portion of the current embedded systems work in unsecure and fault inclined conditions. Such systems collect their required energy from the earth with numerous sources of unpredictability. Then again, more than frequently, these systems have timing requirements that ought to be fulfilled despite the current issues and constraints said above.

Dynamic Voltage and Frequency Scaling [96] has risen as one of the best framework level methods for energy consumption. DVFS planning lessens the supply voltage and frequency when conceivable. Its consequences for preserving energy consumption are obvious where supply voltage and frequency specifically influence the system energy consumption. In any case, one result of applying DVFS is the broadened circuit postpone which may undermine the schedulablity of real-time system. Subsequently, an extraordinary number of procedures considering the issue of limiting the vitality utilization without jeopardizing the timing limitations on single-core platforms are proposed for different task models.

For the current DVFS-based research efforts, most of the research either focused on tolerating fixed number of faults or doubtful constant fault rate. However, it was known that there is advantages and disadvantages of voltage scaling on the rate of transient faults. Pop et al. [107] overworked the obstruction of energy and reliability trade-offs for independent heterogeneous embedded systems. The main subject is to recognize transient faults by switching to pre-determined

eventuality schedules and re-executing processes. A latter, constrained programming-based algorithm is expected to demonstrate the voltage levels, process start time and data transmission time to recognize transient faults and trim energy consumption meanwhile meeting the timing constraints of the application. Comparable issues for fixed-priority aperiodic/periodic real-time tasks were previously explored. Broad research has been performed to explore the energy proficiency of real-time systems from both disconnected and online viewpoint [98], [99] , [97], [89]. The power proficient variant of fixed-priority preemptive scheduling for example, rate monotonic scheduling, was investigated in [98]. Power reduction is accomplished by exploiting the slack time both characteristic in system schedule and due to runtime varieties in undertaking execution time. Moreover, authors in [99] exhibited a scheduling approach for hard real-time tasks according to fixed priorities needs appointed in a rate monotonic way. The disconnected scheduling utilizes correct timing examination to infer various voltage scaling factors for each task in light of stochastic characteristics of task execution time.

The online scheduling approach circulates accessible slack time on priority bases. In light of the voltage scaling algorithms, four voltage scaling calculations including Sys-Clock, PM-Clock, and DPM-Clock were proposed in [101] for various equipment which may have high or low voltage scaling overhead and diverse task set characteristics. Of these algorithms, Sys-Clock relegates a single frequency to all tasks in a task set, PM-Clock dole out different frequencies to tasks in a task set, and DPM-Clock powerfully adjusts disconnected task schedule to runtime practices of tasks execution times.

The heuristic has a disconnected part registering a voltage schedule in light of most pessimistic scenario execution time, and an online segment using slack time because of varieties in task execution time for additionally round of energy savings. Both offline and online scheduling schemes were proposed in [102] to deal with the change time and energy overhead of DVS processors. The offline schema produces task schedule configuration time in view of an earlier known undertaking execution time while the online scheme viably suits runtime varieties of task execution time to accomplish energy savings. Online algorithms were exhibited in [97] to viably diminish system energy utilization to handle event streams with hard constant guarantees.

The flexible scheduling scheme controls the power method of the processor to suspend the processing of entry events as late as it could be reasonably be expected. Although energy efficiency in real-time systems were examined from both offline and online perspectives in the above works, fault resistance which is an essential outline requirements were not considered. Fault-tolerance is another essential outline limitation in energy proficient real-time systems. Joint optimization of energy and fault-tolerance in real-time embedded systems has pulled in impressive consideration in the previous decade. In [103] and [87], a fixed priority offline scheduling scheme was proposed based on the rate monotonic scheduling to tolerate faults in hard real-time systems. Authors proposed DVS methods to achieve slacks in a task schedule to reduce energy utilization while tolerating faults during task execution. A task in the task schedule is thought to be defenseless to at most one fault event and the processor can scale its frequency in a persistent range. Fault-tolerance scheduling strategies were also created in [104] to limit the framework level energy utilization while as yet safeguarding the systems original dependability. Fault tolerance failure is accomplished by saving recovery blocks that can be utilized by any task at the runtime.

In addition, authors in [105] exhibited a soft error mindful vitality productive scheduling

procedure for soft real-time systems with stochastic task execution times. The task execution time estimation is displayed as a joint state-space show, the arrangement of which is found by an online Monte Carlo examining based recursive system. An offline reliability power administration conspire is displayed in [106] for real-time tasks with probabilistic execution times. The scheme sets aside simply enough slack to ensure the required reliability while leaving more slack for energy management to accomplish better energy savings. In [107], authors tended to the scheduling and voltage scaling for hard real-time applications that have been statically mapped on heterogeneous scattered embedded systems. Tasks in a given task set are accepted to share a typical deadline and the impact of voltage scaling on system reliability is considered.

In [108], authors inspected the effect of using an application task mapping on the reliability quality of MPSoC. The quantity of transient faults is limited without trading off the timeliness of the system. All these works are energy-aware fault-tolerance schemes. Furthermore, it may statically determine offline task schedules to guarantee hard timing constraints, subsequently preservation cannot use the dynamic slack due to varieties in task execution times and vulnerabilities in fault events for further energy savings. Authors in [109]built up an online scheduling algorithm that joins checkpointing with DVS to tolerate faults in real-time uni-processor systems with periodic tasks. While, this plan cannot deal with hard real-time task scheduling.

In [110], the authors show a way to deal with the combination of fault-tolerant schedules for embedded applications with soft and hard real-time constraints. An arrangement of tasks schedules is combined offline and, at run time, the scheduler chooses the right schedule in light of the fault occurrence, and the real assignment of execution times with the hard timing constraints are ensured. In any case, the exhibited approach does not consider energy into account. In this paper proficient scheduling schemes are proposed to consolidate offline feasibility analysis and online voltage scaling for hard real-time systems in view of the correct planning examination of the rate monotonic algorithm (RMA). Two offline scheduling algorithms that empower the dynamic adjustment are proposed. One is the application level voltage scaling (A-DVS) calculation where every assignment keep running at a similar processor speed. The other one is the task level voltage scaling (T-DVS) algorithm where the tasks keep running at their individual speeds. Rather than continual determining the response time of each task for feasibility examination, the correct planning analysis approach [111] is utilized as a part of the proposed algorithms for feasibility examination. This technique strikingly improves the adjustment of the proposed offline A-DVS and T-DVS algorithms to the runtime behavior of fault events. The adjustment of the offline task schedules to the runtime behavior of fault occurrences is actualized by first pre-computing and saving in a query table the most extreme slack necessities for the processor to dynamically back off, and second recovering and looking at the reserved slack time necessities with the created total slack in the runtime, also, third dynamically scaling down processor speed when the created slack time is equivalent to one or more stored slack than requirements. Further, a simple Scalar-based Intel XScale processor test system, was utilized in [112] to assess the runtime overhead of the proposed scheduling schemes in addition to extensive simulation experiments. A hard real-time test ground has been outlined and the proposed algorithms were likewise confirmed on the test bed.

4.5 Summary

In this chapter, we present a brief summary about fault-tolerant in real-time embedded systems. A state of the art about technologies associated with fault recovery is presented. The next chapter considers the fault model that has reasonable representativity and very general to tolerate a variety of faults in hardware/software in presence of time and energy constraints.

Chapter 5

Energy-Aware Fault-Tolerant Real-Time Scheduling for Embedded Systems

For the past decades, we have experienced an aggressive technology scaling due to the tremendous advancements of IC technology. As massive integration continues, the power consumption of the IC chips exponentially increases which further degraded the system reliability. This in turn poses significant challenges to the design of real-time embedded systems. This chapter targets the problem of designing advanced real-time scheduling algorithms that are subject to timing, energy consumption and fault-tolerant design constraints. To this end, we first investigated the problem of developing scheduling techniques for uniprocessor real-time systems that minimizes energy consumption while still tolerating up to k transient faults to preserve the system's reliability. Two scheduling algorithms are proposed: The first algorithm is an extension of an optimal fault-free low-power scheduling algorithm, named ES-DVFS. The second algorithm aims to enhance the energy saving by reserving adequate slack time for recovery when faults strike. We derive a necessary and sufficient condition that can be checked efficiently for the time and energy feasibility of aperiodic jobs in the presence of faults. Later, we formally prove that the proposed algorithm is optimal for a k-fault-tolerant model. Our simulation results show that, the proposed approach can achieve more energy savings over previous works under reliability constraint.

5.1 Introduction

Embedded systems are becoming increasingly important in our lives. In these embedded devices, the management of energy is a crucial issue. They are more and more varied and appear in extremely diverse sectors such as transport (avionics, cars, buses, ..), multimedia, mobile phones, game consoles, etc. A large part of embedded systems have needs for autonomy and limitations of space (small size) and energy (limited consumption). As a result, the major technological and scientific challenge is to build systems of trust from the point of view of the functionalities provided and the rendered quality of service. It's more about designing these systems at an acceptable cost.

For the past several decades, we have experienced tremendous growth of real-time systems and applications largely due to the remarkable advancements of IC technology. However, as transistor scaling and massive integration continue, the dramatically increased power/energy consumption and degraded reliability of IC chips have posed signficant challenges to the design of real-time embedded systems [71]. Therefore, it is imperative to develop efficient and effective power/energy

management techniques for real-time systems while satisfying the timing constraints. For the past two decades, extensive power management techniques have been developed on energy minimization for real-time systems [72], [73].

Such a problem is usually treated by Dynamic Voltage and Frequency Scaling (DVFS) methods that affect the speed of the processor, which directly affects the energy consumption of the system. The energy-efficient scheduling of real-time tasks in the presence of DVFS has been extensively studied in the last decade [74], [75], [76].

At the same time, it is observed that as embedded real-time systems become more and more complex, the required level of reliability for such systems appears to be another open problem. Many of these systems tend to be situated at harsh, remote or inaccessible locations. Consequently, it is often difficult and sometimes even impossible to repair and to perform maintenance. This necessitates the use of fault-tolerant techniques. Fault-tolerant computing refers to the correct execution of user programs and system software in the presence of faults [77]. Nowadays, the impacts of system failures become more and more substantial, ranging from personal inconvenience, disruption of our daily lives, to some catastrophic consequences such as huge financial loss. Conceivably, guaranteeing the reliability of computing systems has also been raised to a first-class design concern. Recent studies indicate that the emerging low-power design techniques [74] further increase the susceptibility of VLSI circuits to transient faults. Left unchecked, the high power/energy consumption and deteriorating reliability of IC chips will handicap the availability of future generations of real-time computing systems. Hence, faults must be detected and convenient recovery operations must be performed within the timing constraints.

Processor faults can be largely classified into two categories: transient and permanent faults [78]. Transient faults are temporary malfunctioning of the computing unit or any other associated components caused by factors such as electromagnetic interference and cosmic ray radiations, which causes incorrect results to be computed. On the contrary, a permanent or hard fault in hardware is an erroneous state that is continuous and stable. Permanent faults in hardware are caused by the failure of the computing unit. We focus in this chapter on the transient fault since, in most computing systems, the majority of errors are due to transient faults [79]. In the case of an energy-efficient system, reliability also means ensuring that the system will never be short of energy to ensure its treatment. Anticipation of possible cases of energy can, again, be implemented on the basis of the flexibility offered by the system at the level of the execution of the tasks.

In this chapter, we are interested in the problem of real-time scheduling under reliability and energy constraints. It's about considering real-time tasks that have needs which are expressed on the one hand in terms of processing time and energy consumed by the processor and on the other hand in terms of the number of tolerated faults. A task configuration is energy overloaded, this means that the amount of energy consumed is greater than the amount of energy available. In addition, the amount of execution time requested is smaller than the available capacity, the system will therefore typically be able to meet all its deadlines or else catastrophic consequences will occur. A major question that needs to be answered is: how to schedule real-time tasks in case of energy where the system keeps reliable and able to tolerate up to k faults.

To answer this question, a uniprocessor Earliest Deadline First (EDF) scheduling algorithm is first analyzed to derive an efficient and exact feasibility condition by considering energy man-

agement and fault-tolerance. Second, the proposed algorithm is designed to achieve energy autonomous utilization of the processor while meeting the task deadlines.

5.2 Related Work

Researchers in both academia and industry have resorted to various techniques to minimize energy consumption in computing systems. Among these, Dynamic Voltage and Frequency Scaling has risen as one of the best framework level methods for energy consumption. DVFS scheduling reduces the supply voltage and frequency when conceivable for preserving energy consumption. Subsequently, a great number of procedures considering the issue of limiting the energy consumption without jeopardizing the timing constraints on uniprocessor platforms are widely proposed in literature for different task models. Many of the previous work that studied the problem of energy efficient frameworks for real-time embedded systems employ the Dynamic Voltage and Frequency Scaling (DVFS) technique [74], [80], [109], [82], [83].

Yao et al. [80] developed a DVFS scheme for a set of aperiodic real-time tasks scheduled under EDF policy with a focus of minimizing dynamic power consumption for real-time systems. In [82], authors considered the temperate and leakage dependencies and proposed an efficient DVFS scheme to minimize the overall energy consumption while guaranteeing the timing constraints of a real-time system. Later in [83], we settle the hypothesis for energy consumption in real-time systems, we proposed an energy efficient scheduler of aperiodic jobs for real-time embedded systems. Specifically, we applied the concept of real-time process scheduling to a dynamic voltage and frequency scaling (DVFS) technique. Further, we proposed in [84] an energy guarantee scheduling and voltage/frequency selection algorithm targeting at real-time systems with energy harvesting capability. We show that our scheduler achieves capacity savings when compared to other schedulers.

On the other side, fault tolerance, and in general reliability, objectives are of paramount importance for embedded systems [85]: faults and failures can occur in real-time computing systems and can cause hardware errors and/or deadline violations. Since soft errors are more common in computing systems, most researches related to fault tolerance focus on soft errors. Such research efforts was done on scheduling techniques with the joint consideration of energy efficiency and fault tolerance.

Zhu et al. [86] targeted the reliability problem of a real-time system as the probability to execute all tasks, with or without fault occurrences and proposed a linear and an exponential model to capture the effects of dynamic voltage frequency scaling (DVFS) on transient fault rate. They showed that energy management through DVFS is able to reduce the system reliability. Based on this model, they proposed a recovery scheme to schedule a recovery for each scaled job to compensate the reliability loss caused by DVFS.

Melhem et al. [87] investigated the reliability problem for periodic task sets scheduled under EDF on a monoprocessor with the restriction that there is at most one failure (i.e. $k = 1$). Authors presented a checkpointing scheme that can reduce the fault-recovery overhead significantly at the cost of runtime overhead, this means by inserting checkpoints, which may potentially improve the system schedulability and leave more space for energy management. Zhang et al. [88] investigated the same problem but on fixed-priority real-time tasks. For this sake, authors introduced a

combination of checkpointing and DVFS scheme for tolerating faults for periodic task sets while minimizing energy consumption.

More recently, Zhao et al. [89] proposed the Generalized Shared Recovery (GSHR) technique to reserve computing resources that can be shared by different tasks to improve the energy-saving performance. Later, this work was extended for a more general real-time periodic task model [85]. The proposed algorithms aim to determine the processor speed and resource reservation for each task to achieve the goal of energy minimization under the task-level reliability requirement. The advantage of this approach is that the reliability can be quantified and the impacts of DVFS to reliability can also be taken into consideration.

Recently, Han et al. [90] developed effective scheduling methods that can save energy and, at the same time, tolerate up to k transient faults when scheduling a set of aperiodic real-time tasks on a single processor under the EDF policy. For this sake, authors proposed three algorithms: The first two algorithms are based on the previous work performed in [80]. The third algorithm extends the first two by sharing the reserved computing resources and hence better energy saving performance can be achieved. The main drawback of this work is that the problem of improving the system reliability in presence of fault tolerance cannot be simply solved by modifying the work done in [80] .

5.3 Model and Terminology

In this section, we first introduce the system models and related notations. We then formulate our problem formally.

5.3.1 Task Model

We consider a set of n independent aperiodic real-time jobs $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$, where J_i denotes the i^{th} job in a job set $\mathcal{J}$ and is characterized by a tuple (a_i, c_i, d_i). The definition of these parameters are as follows:

- a_i is referred to the arrival time, this means that the time when job J_i is ready for execution.
- c_i is referred to the worst case execution time (WCET) under the maximum available speed S_{max} of the processor.
- d_i is considered as the absolute deadline of job J_i.

We denote the laxity of the job J_i by $d_i - (a_i - c_i)$. We consider that the job set $\mathcal{J}$ is feasible in the real-time sense and under fault-free scenario. That means that when the energy constraints are not taken into consideration, there exists a feasible schedule where all deadlines in $\mathcal{J}$ are met.

5.3.2 Power and Energy Model

We assume the speed / frequency of the processor is equipped with a DVFS-enabled with N discrete frequencies f ranging from $f_{min} = f_1 \leq f_2 \leq \cdots \leq f_N = f_{max}$. We use the term slowdown factor or processor speed S_N as the ratio of the scheduled speed to the maximum processor speed, this means that $S_N = f_N / f_{max}$. The CPU speed can be changed continuously in $[S_{min}, S_{max}]$.

Consequently, when a job J_i is executed under speed S_i, the worst case execution time of J_i becomes equal to c_i/S_i.

For embedded systems, the power is consumed mainly by the processor, and off-chip devices such as memory, I/O interfaces and underlying circuits [91]. Howeever, it has been observed that the power consumption is dominated by dynamic power dissipation, which is quadratically related to supply voltage and linearly related to frequency. In this part, we distinguish between frequency-dependent and frequency-independent power components. Specifically, we adopt the overall power consumption (P) at a slowdown factor S as follows:

$$P = P_{ind} + P_{dep} = P_{ind} + C_{ef} S^{\alpha} \tag{5.1}$$

Where P_{ind} stands for the frequency-independent power including the power consumed by off-chip devices and constant leakage power, which is independent of the system supply voltage and frequency. C_{ef} is denoted as the effective switching capacitance. α is the dynamic power exponent, which is a constant usually larger than or equal to 2.
P_{dep} is considered to be the frequency-dependent active power, including not only the processor power, but also any power that depends on the processing speed S. Consequently, the energy consumption of a job J_i running at the speed S_i, denoted as $E_i(S_i)$, can be expressed as:

$$E_i(S_i) = (P_{ind} + C_{ef} S_i^{\alpha}).\frac{c_i}{S_i} \tag{5.2}$$

We consider that preemption overheads are negligible. Otherwise, they can be incorporated into the job's worst-case execution times [92].

5.3.3 Energy Storage Model

Our system relies on an ideal energy storage unit (battery or supercapacitor), that has a nominal capacity, namely C, corresponding to a maximum energy (expressed in Joules or Watts-hour). The energy level has to remain between two boundaries C_{min} and C_{max} where $C = C_{max} - C_{min}$. We consider that $C(t)$ stands for the energy stored in the energy storage unit at time t. At any time, the stored energy is no more than the storage capacity, that is

$$C(t) \leq C \quad \forall\, t \tag{5.3}$$

5.3.4 Fault Model

During the execution of an operation computing system, both permanent and transient faults may occur due to various reasons, like hardware defects or system errors. In this work, we focus on transient faults since it has been shown to be dominant over permanent faults especially with scaled technology sizes [93].

We consider that the proposed system can afford a maximum of k transient faults. The used system is usually able to detect faults at the end of each job J_i's execution using *acceptance* or *sanity tests* [94]. We assume that the timing and energy overhead for fault detection, denoted as TO_i and EO_i respectively, are not negligible and are not subject to frequency variations.

Faults can occur anywhere at any time during the execution of jobs and multiple faults may hit a single job. The fault recovery scheme in this chapter is based on re-executing the affected job. Consequently, R_i stands for the maximum recovery overhead for executing a job J_i under the maximum speed S_{max}, which is equal to c_i, or $R_i = c_i$. When a fault occurs during the execution of a job, say J_i , a recovery job of the same deadline d_i is released, which is subject to preemption as well.

5.3.5 Terminology

We now give some definitions we will be needing throughout the remainder of this chapter.

Definition 5.12. *A schedule* Γ *for a job set* $\mathcal{J}$ *is said to be valid if the deadlines of all jobs of* $\mathcal{J}$ *are met in* Γ*, starting with a storage fully charged.*

Definition 5.13. *A system is said to be feasible if there exists at least one valid schedule* Γ *for* $\mathcal{J}$ *with a given energy source. Otherwise, it is infeasible.*

In this chapter, we consider that the limiting factors are not only time but are either, both time and energy, only time or only energy. We focus here on feasible systems only.
Formally, we introduce a novel terminology which is peculiar to energy constrained computing systems.

Definition 5.14. *A schedule* Γ *for a job set* $\mathcal{J}$ *is said to be time-valid if the deadlines of all jobs of* $\mathcal{J}$ *are met in* Γ*, considering that* $\forall\ 1 \leq i \leq n$*,* $E_i(S_i) = 0$*.*

Definition 5.15. *A system is said to be time-feasible if there exists at least one time-valid schedule* Γ *for* $\mathcal{J}$*. Otherwise, it is infeasible.*

Definition 5.16. *A schedule* Γ *for a job set* $\mathcal{J}$ *is said to be energy-valid if the deadlines of all jobs of* $\mathcal{J}$ *are met in* Γ*, considering that* $\forall\ 1 \leq i \leq n$*,* $c_i = 0$*.*

Definition 5.17. *A system is said to be energy-feasible if there exists at least one time-valid schedule* Γ *for* $\mathcal{J}$*. Otherwise, it is infeasible.*

5.3.6 Problem Formulation

We formulate our problem formally as follows:
Given a set real-time job $\mathcal{J}$ of n independent aperiodic jobs $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$ with a release time, worst-case execution time and deadline, running on a DVS-enabled platform, and using a monoprocessor. Is it possible to minimize the overall energy consumption for all tasks and potential recovery operations without deadline violations under any fault scenario with at most k transient faults? In our analysis, we use the ES-DVFS scheduling policy [83], which was proved to be optimal in minimizing the total energy consumption for uniprocessor systems under conventional (non-fault-tolerant) analysis techniques. To answer this question, We have to find the CPU speed decisions (including the recoveries) so as to minimize the overall energy consumption within predefined timing constraints when no more than k faults occur.
We refer to a set of speed values during the whole time interval where $\mathcal{J}$ is executed as a speed schedule.

5.4 Fault Tolerant Speed Schedule

5.4.1 Overview of the Scheduling Scheme

In this section, we present an approach to the development of a fault-tolerant DVFS scheduling for dynamic-priority real-time job set on uniprocessor systems to reduce the energy consumption while still guaranteeing the timing constraints. The proposed algorithm is based on the Energy Saving - Dynamic Voltage and Frequency Scaling (ES-DVFS) algorithm that we previously proposed in [83].

Definition 5.18. *A job set $\mathcal{J}$ is said to be k-fault tolerant if all jobs and potential recovery operations can be completed before their corresponding deadlines under any fault scenario with at most k transient faults.*

To ease the presentation of our approach and before proceeding, we first state some basic definitions and then reiterate briefly the general idea of ES-DVFS.

Definition 5.19. *Given a real-time job set $\mathcal{J}$ of n independent aperiodic jobs such that $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$.*

- *$\mathcal{J}(t_s, t_f)$ denotes the set of jobs contained in the time interval $\phi = [t_s, t_f]$, i.e jobs that are ready to be processed at time t_s and with deadlines at or earlier than t_f. $\mathcal{J}(\phi) = \{J_i \mid t_s \leq a_i < d_i \leq t_f\}$.*

- *$W(\phi)$ denotes the total amount of workload of jobs in $\mathcal{J}(\phi)$ in the time interval $[t_s, t_f]$, that means that the total worst case execution time of jobs completely contained in the interval,*

$$W(\phi) = \sum_{J_i \epsilon \mathcal{J}(\phi)} c_i \tag{5.4}$$

- *The processor load $h(\phi)$ over an interval $\phi = [t_s, t_f]$ is defined as*

$$h(\phi) = \frac{W(\phi)}{t_f - t_s} \tag{5.5}$$

- *The intensity of jobs in the time interval $\phi = [t_s, t_f]$, denoted as $I(\phi)$, is defined as*

$$I(\phi) = \max_{J_j \epsilon \mathcal{J}(\phi)} \left(\frac{\sum\limits_{d_i \leq d_j} c_i}{d_j - (t_f - t_s)} \right) \tag{5.6}$$

- *We consider that the* fault-related overhead *of a time interval $\phi = [t_s, t_f]$, denoted as $W_k(\phi)$ is*

$$W_k(\phi) = W_r(\phi) + W_{TO}(\phi) \tag{5.7}$$

  Where $W_r(\phi)$ stands for the worst-case reserved workload to be used in case of recovery, i.e. $W_r(\phi) = k \times (R_l + TO_l)$ and l represents the index of the job with the longest recovery time

in $\mathcal{J}(\phi)$. $J_l = \{J_i \mid max(R_i + TO_i),\ J_i \epsilon \mathcal{J}(t\phi)\}$ *and* $W_{TO}(\phi)$ *denotes the overhead imposed by fault detection from regular jobs, i.e.*

$$W_{TO}(\phi) = \sum_{J_i \epsilon \mathcal{J}(\phi)} TO_i \tag{5.8}$$

Further, $W_k(\phi) \geq W_{k-1}(\phi)$ *for* $k \geq 1$*, since all recovery of jobs have non-negative execution times. For this sake, we restrict our analysis to the k-fault tolerance where we have exactly k faults when investigating the worst-case reserved recovery of fault scenarios with at most k faults.*

– *The energy demand of a job set* $\mathcal{J}$ *on the time interval* $\phi = [t_s, t_f]$ *is*

$$g(\phi) = \sum_{t_s \leq r_k, d_k \leq t_f} E_k(S_k) \tag{5.9}$$

Given a real-time job set $\mathcal{J}$, ES-DVFS was provably optimal in minimizing energy consumption in on-line energy-constrained setting by providing sound dynamic speed reduction mechanisms [83]. ES-DVFS provides an exact energy management technique as function of the processor frequency in such a way that time constraints are still respected. Using this framework, the speed of the jobs ready to be executed is not fixed in the used interval as the previous work in [80] but is dynamically adjusted on the fly. ES-DVFS is employed as follows:

– *Step 1:* Identify an interval $\phi = [t_s, t_f]$, add the ready jobs to the job queue $\mathcal{Q}$ and select the job J_i with the highest priority.

– *Step 2:* Calculate the effective processor load $h(\phi)$ and intensity $I(\phi)$ using equations 5.5 and 5.6 respectively.

– *Step 3:* Set the speed S_i of job J_i to the maximum between $h(\phi)$ and $I(\phi)$.

– *Step 4:* In case of preemption, update S_i.

– *Step 5:* Remove job J_i from the queue $\mathcal{Q}$.

– *Step 6:* Repeat step (1) - (6) until $\mathcal{Q}$ is empty.

Moreover, we proved that ES-DVFS provides an optimal speed schedule for a given job set $\mathcal{J}$.

Lemma 5.1. *[83] An optimal speed schedule for a job set* $\mathcal{J}$ *is defined on a set of time intervals* $\phi = [t_s, t_f]$ *in which the processor maintains a constant speed* $S_i = \max\Big(I(\phi), h(\phi)\Big)$ *where* $h(\phi)$ *and* $I(\phi)$ *are respectively the workload and intensity of jobs in* $\phi = [t_s, t_f]$ *and each of these intervals* $[t_s, t_f]$ *must start at* t_s *and with deadlines at or earlier than* t_f .

5.4.2 Concepts for the EMES-DVFS Model

However, ES-DVFS is optimal in case of fault-free conditions. Hence, To make the above ES-DVFS fault-tolerant, we adopt an approach (we call it MES-DVFS) is to take the fault recovery

into consideration when calculating the effective processor load and intensity in any interval $\phi = [t_s, t_f]$, i.e. to replace $h(\phi)$ and $I(\phi)$ with $h_m(\phi)$ and $I_m(\phi)$ respectively, such that

$$h_m(\phi) = \frac{\sum\limits_{J_i \epsilon \mathcal{J}(\phi)} c_i + k \times R_l}{d_{max} - W_{TO}(\phi) - k \times TO_l} \tag{5.10}$$

Where d_{max} is the maximum deadline in the job set $\mathcal{J}(\phi)$, l is the index of the job with the longest recovery in $\mathcal{J}(t\phi)$ and $W_{TO}(\phi)$ stands for the total fault-detection overheads for regular jobs as defined in Definition 5.19.

In addition, the intensity of the jobs in $\mathcal{J}(\phi)$ at current time t is

$$I_m(t) = \max_{J_j \epsilon \mathcal{J}(\phi)} \left(\frac{\sum\limits_{d_i \leq d_j} c_i + k \times R_l}{d_j - t - W_{TO}(\phi) - k \times TO_l)} \right) \tag{5.11}$$

Aydin, in [95], showed that the feasibility condition of scheduling a job set $\mathcal{J}$ by using EDF scheduler on a single processor that can tolerate a maximum number of k transient faults can be summarized as

Theorem 5.4. *[95] Given a real-time job set $\mathcal{J}$ with k faults to be tolerated and $S_{max} = 1$, if for each interval $[t_s, t_f]$, we have*

$$\frac{\sum\limits_{J_i \epsilon \mathcal{J}(t_s, t_f)} c_i + W_{ft}(t_s, t_f)}{t_f - t_s} \leq 1 \tag{5.12}$$

When a fault is detected, and for the sake of reduce the total energy consumption for both the original jobs and their recovery copies, MES-DVFS executes the copy of the recovered job using a scaled processor speed ($S_i \leq S_{max}$). However, this may not be energy efficient since the fault rate is usually very low in practice.

An extended approach for MES-DVFS (we call it EMES-DVFS), is to execute the recovery copies under the maximum possible processor speed, usually at S_{max}.

Hence, the intensity calculation of the jobs in $\mathcal{J}(\phi)$ can be modified correspondingly, as equation 5.13

$$I_e(t) = \max_{J_j \epsilon \mathcal{J}(\phi)} \left(\frac{\sum\limits_{d_i \leq d_j} c_i}{d_j - t - W_k(\phi)} \right) \tag{5.13}$$

Further, the effective processor load of the jobs in $\mathcal{J}(\phi)$ can also be modified correspondingly, as equation 5.14

$$h_e(\phi) = \frac{\sum\limits_{J_i \epsilon \mathcal{J}(\phi)} c_i}{d_{max} - W_k(\phi)} \tag{5.14}$$

5.4.3 Description of the EMES-DVFS Scheduler

In what follows, we consider a given set of n jobs $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$ that can tolerate up to k faults. Let $\mathcal{Q}(\phi)$ be the list of uncompleted jobs ready for execution at in the time interval $\phi = [t_s, t_f]$. We can formulate our EMLPEDF algorithm to obey the following rules:

Rule 1: The EDF priority order is used to select the future running jobs in $\mathcal{Q}(\phi)$.
Rule 2: The processor is imperatively idle in $[t_s, t_s + 1)$ if $\mathcal{Q}(\phi)$ is empty.
Rule 3: The processor is imperatively busy in $[t_s, t_s + 1)$ if $\mathcal{Q}(\phi)$ is not empty and $0 < C(t_s) \leq C$.
Hence, the following steps must be performed:

1. Select the job, say J_i with the highest priority.
2. Calculate the effective processor load $h_e(\phi)$ and intensity $I_e(\phi)$ using equations 5.14 and 5.13 respectively.
3. Set the speed S_{ei} of job J_i to the maximum between $h_e(\phi)$ and $I_e(\phi)$.

Rule 4: If $S_{ei} < S_{min}$, then $S_{ei} = S_{min}$ $\forall$ $J_i \epsilon \mathcal{J}(\phi)$.
Rule 5: If job, say J_j is released with $d_j < d_i$, then update S_{ei} by **Rule 3**.
Rule 6: If job, say J_k is released with $d_k > d_i$, then complete the execution of J_i.
Rule 7: If job, say J_k is released with $d_k > d_i$, and $c_k > d_k - d_i$ then update S_{ei} by **Rule 3**.
Rule 8: Calculate the energy consumption $E_i(S_{ei})$ according to eq. (5.2).
Rule 9: Calculate the remaining energy in the battery at the end of the execution.
Rule 10: Remove job J_i from the queue $\mathcal{Q}(\phi)$.
Rule 11: Repeat step (1) - (8) until the queue $\mathcal{Q}$ is empty.

5.4.4 Feasibility Analysis

When the job set $\mathcal{J}$ is feasible, it is not difficult to verify that $S_e(\phi) \leq S_m(\phi)$ for a given interval $\phi = [t_s, t_f]$ since $\sum_{J_i \epsilon \mathcal{J}(\phi)} c_i + W_k(\phi) \leq t_f - t_s$, where $S_m(\phi)$ and $S_e(\phi)$ are equal to $\max\left(I_m(\phi), h_m(\phi)\right)$ and $\max\left(I_e(\phi), h_e(\phi)\right)$ respectively.

More importantly, since EMES-DVFS can be successfully applied for a given job set $\mathcal{J}$, then we can guarantee the feasibility of the resulted schedule. This is summarized in the following theorem.

Theorem 5.5. *EMES-DVFS can guarantee that the deadlines of all jobs can be met as long as the following two constraints are satisfied : (1) no more than k faults occur; (2)* $\forall$ i ϵ $[1, n]$, *where n is the number of jobs in the job set* $\mathcal{J}$, *we have* $S_{ei} \leq 1$ *and* $C(t) > 0$ $\forall$ t.

Proof. In EMES-DVFS, a time interval $\phi = [t_s, t_f]$ is only reserved for executing jobs and their recovery copies in the interval. For any job with higher priority, say J_h, and ready at time t, with possible execution overlapping with ϕ , the EMES-DVFS scheduler must preempt the running job and begin executing J_h and hence we will have a new slowdown factor that forces J_h to finish within the interval ϕ. Similarly, for any lower priority job (e.g. J_l) with possible execution overlapping with ϕ, the EMES-DVFS scheduler will update the processor speed and continue

execution. J_l is excluded from execution in the time interval ϕ and will be postponed to the next interval. Hence, scheduling decisions are only applied online when any of the following events really arrive: 1) Event 1: a new aperiodic job is ready and is added to the job queue. 2) Event 2: the current job completes its execution.

Therefore, to prove the theorem, it is sufficient to prove that when we set the processor speed to be S_{ei}, then the schedulability of all jobs in ϕ is guaranteed in the worst case scenario (i.e. against k faults), as long as $S_{ei} \leq 1$ and the energy reservoir is not fully depleted ($C(\phi) > 0$). We prove this by contradiction. Consider that $J_b = (r_b, c_b, d_b)\ \epsilon\ \mathcal{J}(\phi)$ misses its deadline when the processor speed is set to S_{ei}. Here, we have 2 cases:

Case 1: J_b misses its deadline because of time starvation. Then we must be able to find a time $t \leq r_b$, such that for interval $\phi' = [t, d_b]$, we have $\frac{W(\phi')}{S_{ei}} + W_k(\phi') \geq d_b - t$. Here we have 2 cases:

Case 1a: J_b is not contained in the time interval ϕ (i.e. $d_b \geq t_f$). In this case, event 2 occurs, which means that the current running job, say J_i, will complete its execution without being preempted and job J_b will be executed in the next time interval. This contradicts that J_b misses its deadline.

Case 1b: J_b is contained in the time interval ϕ (i.e. $d_b \leq t_f$). In this case, event 1 occurs, which means that the EMES-DVFS scheduler updates the speed schedule of the processor for all the jobs including the new one in the job queue and consequently S_{ei} is updated to another slowdown value S'_{ei} such that $S'_{ei} \geq S_{ei}$ and $\phi' \subseteq \phi$. Here we have 2 cases:

Case 1b1: Effective processor load $h_e(\phi')$ is greater than the intensity $I_e(\phi')$. This means that $S'_{ei}(\phi') = h_e(\phi')$. But, $h_e(\phi')$ is equal to $\frac{W(\phi')}{d_{max} - W_k(\phi')}$ which is greater than or equal to S_{ei}. Since $S'_{ei}(\phi') \geq S_{ei}$, then we have $\frac{W(\phi')}{S'_{ei}} \leq \frac{W(\phi')}{S_{ei}}$. Take Eq. 5.14 into the right-hand side of the above inequality and add $W_k(\phi')$ to both sides. We have $\frac{W(\phi')}{S'_{ei}} + W_k(\phi') \leq d_b - t \leq d_{max} - t$. This violates the assumption that J_b misses its deadline.

Case 1b2: Effective processor load $h_e(\phi')$ is smaller than the intensity $I_e(\phi')$. This means that $S'_{ei}(\phi') = I_e(\phi')$. But, $I_e(\phi')$ is equal to $\max\limits_{J_j \epsilon \mathcal{J}(\phi')} \left(\frac{\sum\limits_{d_i \leq d_j} c_i}{d_j - t - W_k(\phi')} \right)$ which is greater than or equal to S_{ei}. Since $S'_{ei}(\phi') \geq S_{ei}$, then we have $\frac{\sum\limits_{d_i \leq d_j} c_i}{S'_{ei}} \leq \frac{\sum\limits_{d_i \leq d_j} c_i}{S_{ei}}$. Take Eq. 5.13 into the right-hand side of the above inequality and add $W_k(\phi')$ to both sides. We have $\frac{W(\phi')}{S'_{ei}} + W_k(\phi') \leq d_b - t$. This violates the assumption that J_b misses its deadline.

Case 2: J_b misses its deadline because of energy starvation. This means that $.d_b$ is missed with energy demand $g(d_b) = 0$. Then we must be able to find a time $t_0 \leq d_b$ where a job with deadline after d_b is released and no other job is ready just before t_0 and the energy storage unit is fully replenished i.e. $C(t_0) = C$. The processor is busy at least in the time interval $\phi' = [t_0, d_b]$. Here we also have 2 cases:

Case 2a: No job with deadline greater than d_b executes within the time interval ϕ'. This means that all the jobs that execute within ϕ' have release time at or after t_0 and deadline at or before d_b. The amount of energy needed to fully execute these tasks is $g(\phi')$. But since the processor is always busy in the time interval ϕ', then jobs are executed with the minimum possible speed. Further, the energy reservoir is fully charged at t_0. Consequently, $g(\phi') < C(t_0) < C$. We

conclude that all jobs ready within ϕ' can be fully executed with no energy starvation which contradicts the deadline violation at d_b with $C(d_b) = 0$.

Case 2b: At least one job, say J_m is released within time interval ϕ' and with with $r_m > r_b$. Here we have 2 cases:

Case 2b1: J_m is released with $d_m < d_b$, therefore we have to update S_{ei} by Rule 3. Let t_2 be the latest time where J_m is executed. As d_m is lower than d_b and jobs are executed according to preemptive EDF, we have $r_m \geq r_b$ and J_b is preempted by the higher priority job J_m. Thus, the processsor speed must be updated, otherwise d_m will be violated. Since the processor is busy all the times in $[t_2, d_b]$ and the job set $\mathcal{J}$ is time-feasible, then S_{em} will be the minimum speed for the execution of J_m and consequently $g(t_2, d_b) < C(d_b)$. Consequently, the amount of energy that J_m require is at most $g(t_2, d_b)$. That contradicts deadline violation and $C(d_b) = 0$.

Case 2b2: J_m is released with $d_m > d_b$. We consider two cases: (i) $c_m < d_k - d_b$, hence J_b will complete its execution (Rule 6) and the proof is therefore similar to *case 2a.* (ii) $c_m > d_k - d_b$, hence S_{eb} must be updated (Rule 7) and the proof is therefore similar to *case 2b1.*

Since all jobs in $\mathcal{J}$ are executed within time intervals in EMES-DVFS and all jobs within these time intervals are still schedulable when the corresponding speed is applied, we prove the theorem.

□

We state the optimality of EMES-DVFS by proving that a job set $\mathcal{J}$ is feasible in a k-fault-tolerant if and only if all the jobs in $\mathcal{J}$ are executed without violating time and energy constraints. This violation is due to one of the two following reasons: either job, say J_i lacks time (Lemma 5.2) or job J_i lacks energy (Lemma 5.3) to complete its execution before or at deadline d_i. The time starvation occurs when deadline d_i is missed with the energy reservoir not exhausted at d_i. On the other side, the energy starvation case is when the energy reservoir is fully depleted at d_i and J_i is not completed.
Further, the feasibility of the EMES-DVFS scheduler is guaranteed, which is formulated in Lemma 5.2 and Lemma 5.3.

Lemma 5.2. *A real-time job set $\mathcal{J}$ can be time-feasible in a k-fault-tolerant manner by EMES-DVFS if and only if all the jobs in $\mathcal{J}$ can meet their deadlines when they are executed based on the processor speeds determined by EMES-DVFS for every time interval $[t_s, t_f]$.*

Proof. Only if part. Directly follows Theorem 5.5.
If part. Suppose the contrary. Let us consider $\mathcal{J}(\phi)$ as the set of jobs contained in the time interval $\phi = [t_s, t_f]$, this means jobs that are ready to be processed at or after time t_s and with deadlines at or earlier than t_f. We denote a fault pattern $f = \{f_1, f_2, \cdots, f_n\}$, where f_i refers to the number of faults affecting job J_i and its recovery. Hence, we say that f is a k-fault pattern if the total number of faults is exactly k. Formally $h_e(\phi) \leq 1$ for all intervals $[t_s, t_f]$. However, there is a j-fault pattern $j \leq k$ (say f^j) resulting in deadline miss(es). Let us assume that the first deadline violation occurs at $t = d_i$ and that t_0 is the latest time preceding d_i such that either the processor is idle or a job (recovery) of deadline $> d_i$ is executing.
We note that the time t_0 is well-defined in a way that it corresponds to a job release time. In addition, the processor is continuously busy executing jobs (recovery) in the time interval

$\phi^0 = [t_0, d_i)$. Now, let us denote $f^0 \subset f^j$ be the subset of faults affecting jobs in the time interval ϕ^0. Note that the number of faults in f^0 is obviously smaller than k. Since EDF is a work-conserving scheduling algorithm, this means that the processor is never kept idle unless there are no ready jobs, the deadline violation at d_i and the above definition of t_0 imply that the available processor time in the interval time interval ϕ^0 was not sufficient to accommodate the increase in the processor demand even there is no energy starvation in the interval ϕ^0 (the battery is not fully replenished at time d_i). Consequently, we obtain $\sum_{J_i \epsilon \mathcal{J}(\phi^0)} c_i + j \times R_l > d_{max} - W_{TO}(\phi^0) - j \times TO_l$, where j is the number of faults in ϕ^0 and l stands for the index of the job with the longest recovery time in $\mathcal{J}(\phi^0)$. But since $\sum_{J_i \epsilon \mathcal{J}(\phi)} c_i + W_k(\phi) \leq d_i - t_0$ ($h_e(\phi) \leq 1$) and $W_k(\phi) > W_k(\phi^0)$ and $\sum_{J_i \epsilon \mathcal{J}(\phi)} c_i > \sum_{J_i \epsilon \mathcal{J}(\phi^0)} c_i$, we get $\sum_{J_i \epsilon \mathcal{J}(\phi^0)} c_i + W_k(\phi^0) \leq d_i - t_0$ contradicting our assumption that a deadline violation occurs at d_i.

□

Lemma 5.3. *A real-time job set $\mathcal{J}$ can be energy-feasible in a k-fault-tolerant manner by EMES-DVFS if and only if all the jobs in $\mathcal{J}$ can meet their deadlines when they are executed based on the processor speeds determined by EMES-DVFS considering that for every time interval $[t_s, t_f]$, $g(t_s, t_f) > 0$.*

Proof. Only if part. Directly follows Theorem 5.5.
If part. Suppose the contrary. Let us consider $\mathcal{J}(\phi)$ as the set of jobs contained in the time interval $\phi = [t_s, t_f]$. We also denote a fault pattern $f = \{f_1, f_2, \cdots, f_n\}$, where f_i refers to the number of faults affecting job J_i and its recovery. Hence, we say that f is a k-fault pattern if the total number of faults is exactly k and the energy in the reservoir is sufficient to execute all jobs in ϕ. Formally $g(\phi) > 0$ for all intervals $[t_s, t_f]$. However, there is a j-fault pattern $j \leq k$ (say f^j) resulting in deadline miss(es) due to energy starvation. Let us assume that the energy reservoir becomes empty at $t = d_i$ ($C(d_i) = 0$) and that t_0 is the latest time preceding d_i such that the processor is still executing a job (recovery) of deadline $< d_i$.
We note that the time t_0 is well-defined in a way that it corresponds to a job release time. In addition, the processor is continuously busy executing jobs (recovery) in the time interval $\phi^0 = [t_0, d_i)$. Now, let us denote $f^0 \subset f^j$ be the subset of faults affecting jobs in the time interval ϕ^0. Note that the number of faults in f^0 is obviously smaller than k. Since EDF is a work-conserving scheduling algorithm, this means that the processor is never kept idle unless there are no ready jobs, the deadline violation at d_i and the above definition of t_0 imply that the available energy in the reservoir in the time interval ϕ^0 was not sufficient to accommodate the increase in the energy demand even there is no time starvation in the interval ϕ^0, this means $\sum_{J_i \epsilon \mathcal{J}(\phi^0)} c_i + j \times R_l \leq d_{max} - W_{TO}(\phi^0) - j \times TO_l$, where j is the number of faults in ϕ^0 and l stands for the index of the job with the longest recovery time in $\mathcal{J}(\phi^0)$. But since the job set $\mathcal{J}(\phi)$ is feasible, then $g(\phi) > 0$. In addition, the energy demand in ϕ is greater than the energy demand in ϕ^0, since the number of faults in ϕ (k) is more than that in ϕ^0 (j), i.e. $g(\phi) > g(\phi^0)$. Hence, we get $g(\phi) > g(\phi^0) > 0$ contradicting our assumption that a deadline violation occurs at d_i because of energy starvation. □

Now, we may draw Theorem 5.6, a major result of optimality for uniprocessor scheduling in a k-fault-tolerant manner by EMES-DVFS with time and energy constraints.

Theorem 5.6. *The EMES-DVFS scheduling algorithm is optimal for a k-fault-tolerant model.*

Proof. According to Lemma 5.2, EMES-DVFS can schedule a given set of jobs $\mathcal{J}$ in a k-fault-tolerant manner, without violating timing constraints when the energy demand is lower than the maximum energy that is available in the reservoir. According to Lemma 5.3, EMES-DVFS can schedule a given set of jobs Γ in a k-fault-tolerant manner, without violating energy constraints when the processor demand is cannot exceed the maximum available processor time that could be available in any given time interval. As a conclusion, if EMES-DVFS can schedule a given set of jobs $\mathcal{J}$ for a given time or/and energy constraints without time starvation and energy starvation, are the only two reasons for deadline violations, then we conclude that EMES-DVFS is optimal. □

5.5 Simulation Results

In this section, we compare the performance of four scheduling algorithms: EMES-DVFS, MES-DVFS, NPM and LPSSR proposed in [90]. NPM scheme executes jobs with maximum frequency and does not scale down the voltage/frequency.

We developed a discrete event-driven simulator in C that generates a job set $\mathcal{J}$ where the number of jobs varies from 10 to 50. The simulation is repeated 100 times for the same number of jobs.

For the sake of clarity, we use NPM as a reference schedule that represents the schedule of given set of jobs $\mathcal{J}$ without incorporating DVFS. This means that all jobs or recoveries in $\mathcal{J}$ are executed under the maximum processor speed S_{max}. We consider that all the plotted energy consumptions are normalized to NPM. However, to give LPSSR a fair chance, we consider the same parameters as used in [90]. Hence, we consider the following parameters: We assumed that $\alpha = 2$, $C_{ef} = 1$, $P_{ind} = 0.05$, and S_{min} is set to 0.25.
The proposed algorithms are tested with job sets randomly generated as follows: the choice of the arrival time a_i and the relative deadline of each job J_i is uniformly distributed in the interval $[0s, 100s]$ and $[50s, 100s]$ respectively. Moreover, the worst case execution time c_i is randomly generated such that $c_i < d_i$. The timing and energy overhead of detecting faults is considered as 10% of the worst case execution time and its energy consumption respectively. As for the fault arrival rate, we consider 2 cases: safety-critical real-time system with range of 10^{-10} to 10^{-5} /hour or in harsh environment with a range between 10^{-2} and 10^{2} /hour.

All simulation results are computed on a discrete DVFS processor that operates on 8 frequency levels $\{1.00, 0.86, 0.76, 0.67, 0.57, 0.47, 0.38, 0.28\}$ as in the PentiumM processor.

We report here two sets of experiments. The first set is designed to show the energy consumption of the 4 approaches by varying the number of jobs. In the second experiment set-up, we compare the energy consumption by varying the number of faults.

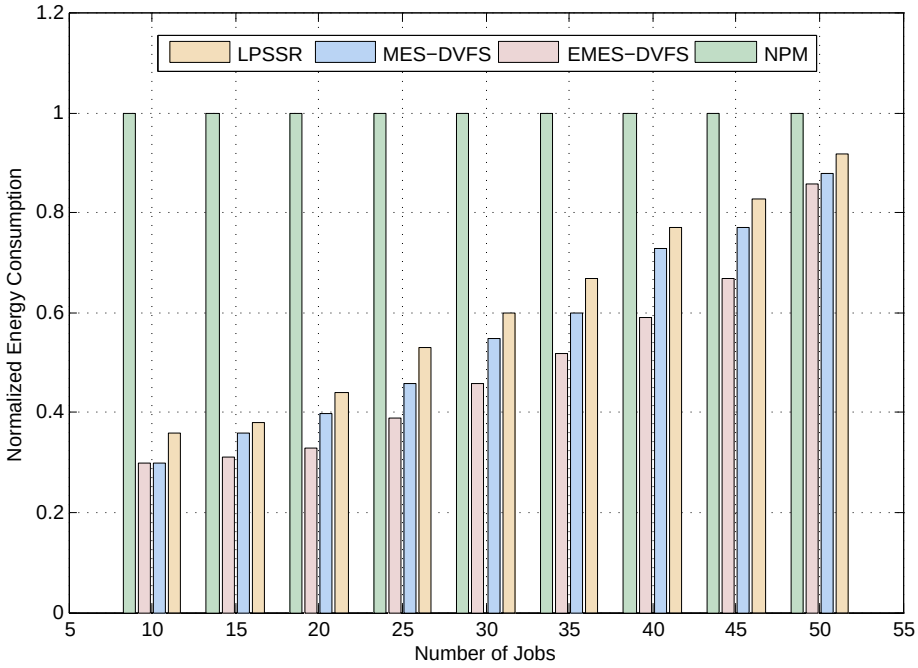

Figure 5.1: *Energy savings by varying the numbers of jobs,* $k = 1$.

5.5.1 Experiment 1: Energy Consumption by Varying the Number of Jobs

First, we take interest in how energy saving performance changes with the number of jobs. We report here the results of four simulation studies where the fault rate is set to 10^{-5} and the number of jobs varies from 10 to 50. Further, we consider that the number of faults in our job set is no more than 1 ($k = 1$). For each job set, we compute the normalized energy consumption metric of the speed schedule by each of the four schedulers. The energy consumption metric is normalized with respect to the NPM scheme, which executes all jobs at S_{max}. Figure 5.1 shows the expected energy consumption of EMES-DVFS and MES-DVFS versus previous schedulers like NPM and LPSSR.

From figure 5.1, we find that the energy consumption of the four schedulers increases as the number of jobs becomes larger. This is reasonable since the likelihood of having large slack time that can be used for DVFS is diminishing as the number of jobs increases. Further, the gain in energy saving provided by EMES-DVFS and MES-DVFS schemes is significant since it can benefit from the optimal amount of slack time that minimizes the expected energy consumption. In other words, EMES-DVFS and MES-DVFS can effectively assign the speeds to each job in such way that the job set becomes feasible at a speed closest to the critical speed.

When the number of jobs is low, we find that the energy savings achieved by all three algorithms are almost the same. This is because most jobs are executed at the lowest speed. With the increasing the number of jobs, our approach starts to show its advantage and achieve high energy saving. In average, additional 51% and 20% energy saving can be achieved by EMES-DVFS when compared with NPM and LPSSR, respectively. Further, the energy consumption difference is around 12% between EMES-DVFS can MES-DVFS, since in the EMES-DVFS scenario, the re-execution of one faulty job is performed at maximum frequency and subsequent slack is left for DVFS.

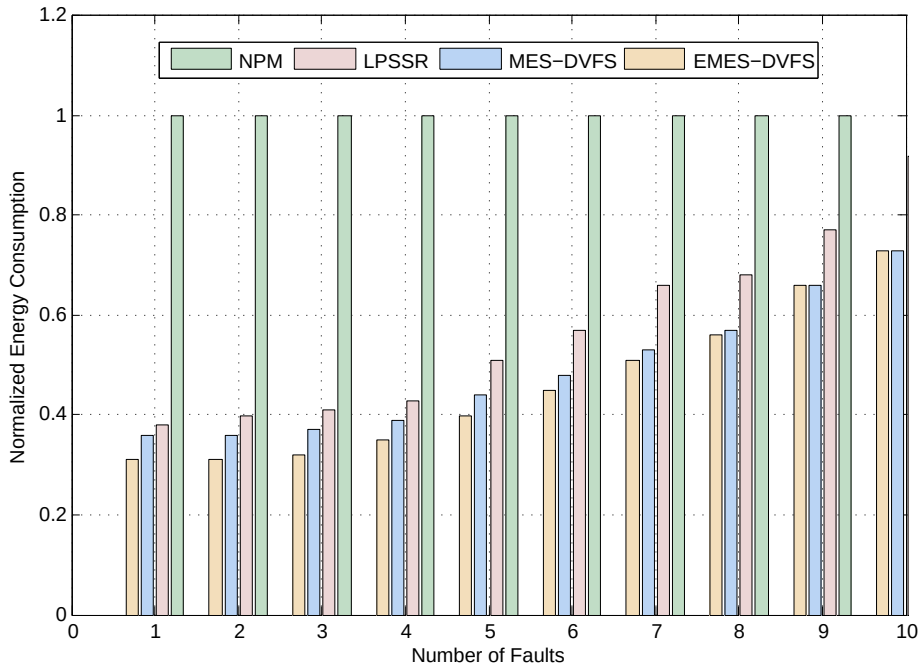

Figure 5.2: *Energy savings by varying the numbers of faults.*

We conclude that our approach gains more energy savings in a sense that it can explore the slacks generated during run-time and hence it can use all the available slack time. This results in more opportunities for backups reclaim and DVFS.

5.5.2 Experiment 2: Energy Consumption by Varying the Number of Faults

In this set of simulations, we evaluated the impact of the number of faults on energy savings. In this simulation, we fix the number of jobs to 15 and the number of faults to be tolerated varies between 1 and 10. Again, 100 different test cases were generated for simulations with the same number of fault. The average results are shown in figure 5.2.

From the figure 5.2, we can find that EMES-DVFS and MES-DVFS can achieve energy savings compared to LPSSR and NPM. Clearly, we find that the energy consumptions by the four schedulers increase rapidly as the number of faults increases, since more expected energy may be consumed due to the increased number of recovery jobs being executed, which in turn limits the maximum amount of dynamic slack used. However, as the number of faults increases, the energy consumption in EMES-DVFS and EMES-DVFS grows but less dramatically.

As illustrated in figure 5.2, the EMES-DVFS and MES-DVFS approaches attain respectively around 19% and14% more energy saving than LPSSR. The reason is that the optimal dynamic slack time to minimize the expected energy consumption is used to the maximum extent by employing speed assignment on the fly. On the contrary, LPSSR is significantly affected by the increasing number of faults in the system and more than 22% additional energy is consumed when fault occurrences increase from 1 to 10. On the other hand, EMES-DVFS could tolerate up to 5 times more faults with same energy as consumed by LPSSR.

As a conclusion, the advantage of our approaches (EMES-DVFS and MES-DVFS) over the other two (LPSSR and NPM) in terms of energy savings is evident in this experiment where

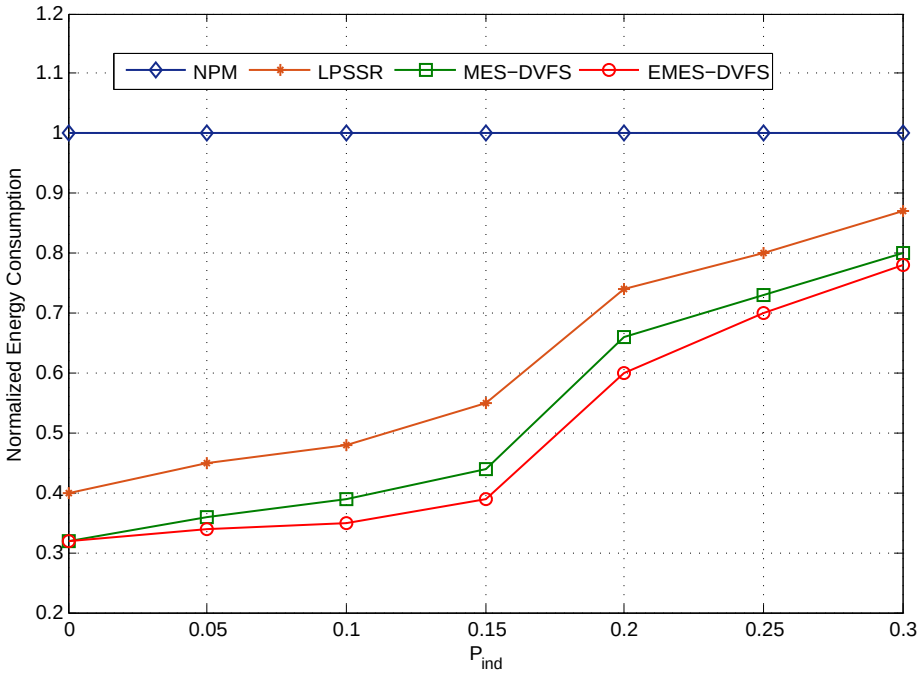

Figure 5.3: *Energy savings by varying P_{ind}.*

EMES-DVFS and MES-DVFS can still guarantee tolerance even under 10 faults and with more energy saving than LPSSR and the energy savings drops around 19% under LPSSR when we compare it with EMES-DVFS.

5.5.3 Experiment 3: Energy Consumption by Varying P_{ind}

In this experiment, we study the impact of frequency-independent power P_{ind} on energy savings. P_{ind} varies between $[0, 0.3]$ for each job and the number of jobs is fixed at 15. According to figure 5.3 , the larger the P_{ind}, the higher the energy consumption. This is due to the fact that as the P_{ind} increases, the contribution of frequency independent energy consumption becomes more dominant, the energy-efficient frequency increases and consequently DVFS has fewer opportunities to be applied. Even under this situation, EMES-DVFS still has the best performance in terms of energy consumption (EMES-DVFS attains approximately 18% more energy saving than LPSSR).

5.5.4 Experiment 4: Percentage of feasible Job Set

In this experiment, we take interest in the percentage of feasible job set that respect their deadlines with the four scheduling algorithms by varying the energy storage capacity. From this experiment, we can deduce two measures. The first one gives us an indication about the percentage of time during which all deadlines are still respected. The second one gives, for each approach and for a given processor load, the minimum size of the storage that ensures time and energy feasibility. We report here the results of two simulation studies where the processor load is set to 0.4 and 0.8, respectively.

Figure 5.4 depicts the percentage of feasible job sets that meet their deadlines over the energy storage capacity C. For each job set, we compute the minimum storage capacity C_{min} which

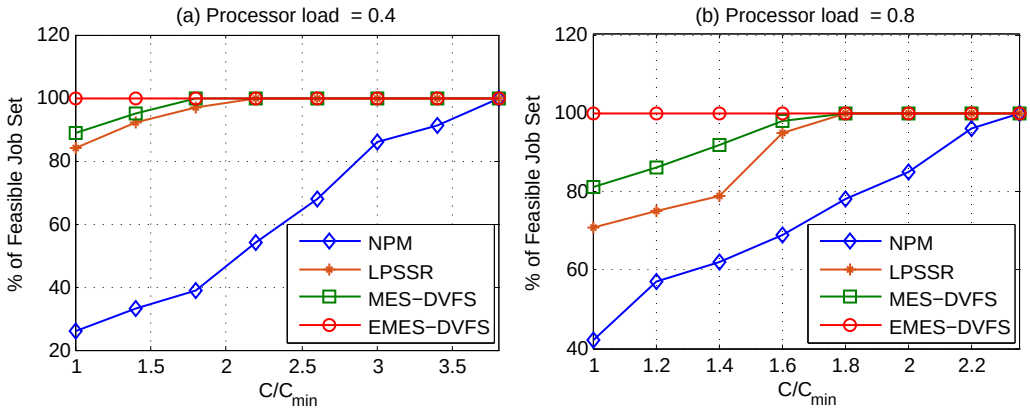

Figure 5.4: *Percentage of feasible job set. (a) Low processor load. (b) High processor load.*

permits achieving time and energy feasibility under EMES-DVFS. We then begin to increase the energy storage capacity till the 4 approaches achieve neutral operation.

Under low processor load (figure 5.4a), it is observed that 100% of job sets meet their deadlines under EMES-DVFS when the energy storage capacity is 4510 energy units, i.e. $C = C_{min} = 4510$ energy units. We start then to increase C till it reaches 8118 where MES-DVFS becomes feasible. this means that means that EMES-DVFS can provide the same level of performance with a storage unit which is about 1.8 times less. The increase in the storage capacity will continue to increase till LPSSR and NPM becomes feasible where the energy storage unit must be respectively more than 2.2 and 3.8 times bigger with LPSSR and NPM to maintain zero deadline miss, compared with EMES-DVFS.

The results for high processor load (figure 5.4b) follow the same trend. Unlike the previous experiment, the relative performance gain of EMES-DVFS in terms of capacity savings is decreasing when the processor load is increasing. EMES-DVFS obtains respectively capacity savings of about 37%, 44% and 57% compared with MES-DVFS, LPSSR and NPM.

It is important here to note that the four approaches require exactly the same storage size when the processor load is equal to 1 since the processor is continuously busy and there is no chance to apply DVFS.

In summary, this experiment points out that the proposed EMES-DVFS approach is very effective in reducing deadline miss rate and storage size even under high processor load. And lower is the processor load rate, higher is the capacity saving and our approach will then outperform the others by a high amount of energy savings.

5.6 Conclusions

In this chapter, we presented and evaluated a novel approach, which aims to minimize energy consumption when scheduling a set of real-time jobs that can tolerate up to k transient faults while still respecting time and energy constraints. We explore the reserved slacks generated during run-time to the maximum extent in such a way that all the available slack time is used for energy reduction, which is carried out using dynamic voltage and frequency scaling (DVFS). Under this notion, we propose an algorithm that estimates an optimal speed reduction mechanism which

maintains feasibility within predefined timing constraints when no more than k faults occur.

Our scheduler dynamically adjusts the jobs' slowdown factors by utilizing run-time slacks which may be increased for recovery demands of the system. It differs from the existing approach where job frequencies assignments are predetermined, and hence it is more flexible and adaptive in minimizing energy consumption while still keeping the system's reliability at a desired level. In addition, we presented two feasibility tests for recovery schemes under variable processor speed which decouples the time and energy constraints. The experimental results demonstrate that the proposed algorithm can significantly improve the energy savings compared with the previous works.

Conclusions

The research presented in this thesis deals with designing scheduling algorithms with the objective of minimizing energy consumption when scheduling a set of real-time jobs that can tolerate up to k transient faults while still respecting time and energy constraints on uniprocessor systems. For this sake, wo proposed two scheduling algorithms: The first algorithm is an extension of an optimal fault-free low-power scheduling algorithm, named ES-DVFS. The second algorithm aims to enhance the energy saving by reserving adequate slack time for recovery when faults strike. The feasibility of the two proposed scheduling schedulers are analyzed for uniprocessor platforms with the main goal at achieving fault-tolerance and energy autonomy, respectively. Both algorithms are designed for a dynamic-priority system, more specifically, the EDF priority policy for a set of implicit-deadline aperiodic jobs.

The presented novel approach, which aims to minimize energy consumption when scheduling a set of real-time jobs that can tolerate up to k transient faults while still respecting time and energy constraints, has an optimal speed reduction mechanism that is carried out using dynamic voltage and frequency scaling (DVFS) and which maintains feasibility within predefined timing constraints when no more than k faults occur.

The experimental results demonstrate that the proposed algorithm can significantly improve the energy savings compared with the previous works. In details, we proved that additional 51% and 20% energy saving can be achieved by EMES-DVFS when compared with NPM and LPSSR, respectively. Further, the energy consumption difference is around 12% between EMES-DVFS can MES-DVFS, since in the EMES-DVFS scenario, the re-execution of one faulty job is performed at maximum frequency and subsequent slack is left for DVFS.

For future work, we will explore the adaptation of the proposed approaches to fixed priority environments in real-time energy harvesting systems.

Bibliography

[1] D. Siewiorek and R. Swarz. *Reliable Computer Systems: Design and Evaluation*. Natick, MA: A. K. Peters, Ltd., 1998.

[2] J. Stankovic. *Misconceptions about real-time computing*. IEEE Computer, 1988.

[3] A. Queudet. *Ordonnancement temps reel avec contraintes de qualite de service*. These de Doctorat de l'Universite de Nantes, 2006.

[4] J. W. S. W. Liu. Real-time systems. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[5] G. -C. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo, Soft Real-Time Systems: Predictability vs. Efficiency. Springer, January 2005.

[6] C. L. Liu and J. W. Layland. *Scheduling algorithms for multiprogramming in a hard-real-time environment*. J. ACM, 20(1), pages 46-61, 1973.

[7] F. Cottet, J. Delacroix, C. Kaiser, and Z.Mammeri. Ordonnancement temps réel. Ed. Hermes, 2000.

[8] G. Coulouris, J. Dollimore, and T. Kindberg. Distributed systems-concepts and design. 2nd Ed, Addison-Wesley Publishers Ltd, 1994.

[9] J. Leung and M. Merril. *A note on preemprive scheduling of periodic real-time tasks*. Information Processing Letters, pages 11(3) : 115-118, 1980.

[10] J.-P. Lehozcky, L. Sha, and Y. Ding. *The rate-monotonic scheduling algorithm :exact characterization and average case behaviour*. in proceedings of the IEEE Real-Time Systems Symposium, pages 166-171, 1989.

[11] J.-Y.-T. Leung and J. Whitehead. *On the complexity of fixed-priority scheduling of periodic, real-time tasks*. Performance evaluation, pages 2 :237-250, 1982.

[12] J.-R. Jackson. *Scheduling a production line to minimize maximum tardiness*. Research Report 43, Management Science Research Project, University of California, Los Angeles, 1955.

[13] M.-L. Dertouzos. *Control robotics : the procedural control of physical processes*. Information Processing, pages 2 :237-250, 1974.

[14] S.-K. Baruah, L.-E. Rosier, and R.-R. Howell. *Algorithms and complexity : Concerning the preemptive scheduling of periodic real-time tasks on one processor*. Real-Time Systems Journal, 2(4), pages 301-324, 1990.

[15] W. Wolf. Computers and components : Principles of embedded computing system design. Morgan Kaufman Publishers, 2000.

[16] R. Casas and O. Casas. *Battery sensing for energy-aware system design.* In Computer, vol.38, no. 11, pages 48-54, 2005.

[17] A. Reinders. *Options for photovoltaic solar energy systems in portable products.* TCME Fourth International symposium, 2002.

[18] N.S. Shenck and J.A. Paradiso. *Energy scavenging with shoe-mounted piezoelectrics.* In IEEE Micro, Vol. 21. No. 3, pages 30-42, 2001.

[19] B. Gaujal and N. Navet. Ordonnancement temps réel et minimisation de la consommation d'énergie, volume 2 of Systèmes temps réel. Hermès, 2006.

[20] L. Benini, A.Bogliolo, and G. D. Micheli. *A survey of design techniques for system-level dynamic power management.* In IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 8, no. 3, pages 299-316, 2000.

[21] F. Yao, A. J. Demers, , and S. Shenker. *A scheduling model for reduced cpu energy.* In IEEE Symposium on Foundations of Computer Science, pages 374-382. IEEE, 1995.

[22] H. Aydin, R. Melhem, D. Mossé, and P. Mejia-Alvarez. *Determining optimal processor speeds for periodic real-time tasks with different power characteristics.* In Euromicro Conference on Real-Time Systems, pages 225-232, 2001.

[23] D. Shin and J. Kim. *Dynamic voltage scaling of periodic and aperiodic tasks in priority-driven systems.* In ASPDAC'03, pages 653-658, 2004.

[24] N. Min-allah, Y. Wang, J. Xing, W. Nisar, and A. Kazmi. *Towards dynamic voltage scaling in real-time systems - a survey.* In IJCSES International journal of Computer Sciences and Engineering Systems, Vol.1, No.2, CSES International, 2007.

[25] C. Moser, D. Brunelli, L. Thiele, , and L. Benini. *Real-time scheduling with regenerative energy.* In 18th Euromicro Conference Real-Time Systems, 2006.

[26] H. El Ghor, M. Chetto, and R. Hage Chehade. *A real-time scheduling framework for embedded systems with environmental energy harvesting.* Computers & Electrical Engineering, 37(4) :498-510, 2011.

[27] H. El Ghor, M. Chetto, and R. Hage Chehade. *A nonclairvoyant real-time scheduler for ambient energy harvesting sensors.* International Journal of Distributed Sensor Networks, 2013.

[28] A. Avizienis, J.-C. Laprie and B. Randell, *Fundamental Concepts of Dependability*, in Technical Report 739, Department of Computing Science, University of Newcastle upon Tyne, 2001.

[29] D.K. Pradhan, Fault-Tolerant Computer System Design, Prentice-Hall, Inc., 1996.

[30] B. Randell, P. Lee and P.C. Treleaven, *Reliability Issues in Computing System Design*, in ACM computing Surveys, vol. 10, issue 2, pp. 123-165, 1978.

[31] I. Koren and C. Krishna, Fault-tolerant systems. Morgan Kaufmann, 2007.

[32] A. Avizienis, Fault-tolerance: *The survival attribute of digital systems*, Proceedings of the IEEE, vol. 66, no. 10, pp. 1109-1125, 1978.

[33] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, *Basic concepts and taxonomy of dependable and secure computing*, Dependable and Secure Computing, IEEE Transactions on, vol. 1, pp. 11–33, Jan 2004.

[34] J.-C. Laprie, *Dependable computing and fault tolerance : Concepts and terminology*, in Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on, Jun 1995.

[35] J. Laprie, *Dependable computing and fault-tolerance*, Digest of Papers FTCS-15, pp. 2–11, 1985.

[36] A. Avizienis and J. Kelly, *Fault tolerance by design diversity- concepts and experiments*, Computer, vol. 17, pp. 67–80, 1984.

[37] J. Sosnowski, *Transient fault tolerance in digital systems*, Micro, IEEE, vol. 14, no. 1, pp. 24–35, 1994.

[38] R. Al-Omari, A. Somani, and G. Manimaran, *A new fault-tolerant technique for improving schedulability in multiprocessor real-time systems*, in Parallel and Distributed Processing Symposium., Proceedings 15th International, IEEE, 2001.

[39] D. Pradhan and N. Vaidya, *Roll-forward checkpointing scheme: A novel fault-tolerant architecture*, Computers, IEEE Transactions on, vol. 43, no. 10, pp. 1163-1174, 1994.

[40] J. Bruno and E. Coffman Jr, *Optimal fault-tolerant computing on multiprocessor systems*, Acta Informatica, vol. 34, no. 12, pp. 881–904, 1997.

[41] A. Avizienis, *Arithmetic error codes: Cost and effectiveness studies for application in digital system design*, Computers, IEEE Transactions on, vol. C-20, pp. 1322-1331, Nov 1971.

[42] T. R. Rao, *Biresidue error-correcting codes for computer arithmetic*, Computers, IEEE Transactions on, vol. C-19, pp. 398-402, May 1970.

[43] W. C. Huffman and V. Pless, *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2003.

[44] P. L'Ecuyer and J. Malenfant, *Computing optimal checkpointing strategies for rollback and recovery systems*, Computers, IEEE Transactions on, vol. 37, no. 4, pp. 491-496, 1988.

[45] E. Gelenbe and M. Hernández, *Optimum checkpoints with age dependent failures*, Acta Informatica, vol. 27, no. 6, pp. 519-531, 1990.

[46] C. Krishna, Y. Lee, and K. Shin, *Optimization criteria for checkpoint placement*, Communications of the ACM, vol. 27, no. 10, pp. 1008–1012, 1984.

[47] V. Nicola, Checkpointing and the modeling of program execution time. University of Twente, Department of Computer Science and Department of Electrical Engineering, 1994.

[48] E. Coffman Jr and E. Gilbert, *Optimal strategies for scheduling checkpoints and preventive maintenance*, Reliability, IEEE Transactions on, vol. 39, no. 1, pp. 9-18, 1990.

[49] Y. Ling, J. Mi, and X. Lin, *A variational calculus approach to optimal checkpoint placement*, Computers, IEEE Transactions on, vol. 50, no. 7, pp. 699-708, 2001.

[50] A. Mahmood and E. McCluskey, *Concurrent error detection using watchdog processors-a survey*, Computers, IEEE Transactions on, vol. 37, no. 2, pp. 160-174, 1988.

[51] A. Ziv and J. Bruck, *Analysis of checkpointing schemes with task duplication*, Computers, IEEE Transactions on, vol. 47, no. 2, pp. 222-227, 1998.

[52] J. Smolens, B. Gold, J. Kim, B. Falsafi, J. Hoe, and A. Nowatryk, *Fingerprinting: bounding soft-error-detection latency and bandwidth*, Micro, IEEE, vol. 24, pp. 22-29, Nov 2004.

[53] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and S. Scott, *A reliability-aware approach for an optimal checkpoint/restart model in hpc environments*, in Cluster Computing, 2007 IEEE International Conference on, pp. 452–457, 2007.

[54] K. Shin, T. Lin, and Y. Lee, *Optimal checkpointing of real-time tasks*, Computers, IEEE Transactions on, vol. 100, no. 11, pp. 1328-1341, 1987.

[55] S. Kwak, B. Choi, and B. Kim, *An optimal checkpointing-strategy for real-time control systems under transient faults*, Reliability, IEEE Transactions on, vol. 50, no. 3, pp. 293-301, 2001.

[56] S. Hiroyama, T. Dohi, and H. Okamura, *Comparison of aperiodic checkpoint placement algorithms*, in Advanced Computer Science and Information Technology, vol. 74 of Communications in Computer and Information Science, pp. 145-156, Springer Berlin Heidelberg, 2010.

[57] S. Hiroyama, T. Dohi, and H. Okamura, *Aperiodic checkpoint placement algorithms - survey and comparison*, Journal of Software Engineering and Applications, vol. 6, pp. 41–53, 2013.

[58] T. Ozaki, T. Dohi, and N. Kaio, *Numerical computation algorithms for sequential checkpoint placement*, Perform. Eval., vol. 66, pp. 311-326, June 2009.

[59] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. *Basic Concepts and Taxonomy of Dependable and Secure Computing.* IEEE Transactions on Dependable and Secure Computing, 1(1):11-33, 2004.

[60] B. W. Johnson. Design & analysis of fault tolerant digital systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[61] K. G. Shin, and P. Ramanathan, *Real-time computing: A new discipline of computer Science and engineering*, Proc. of IEEE Computer, 82(1), 6-24, Jan. 1994.

[62] C. M. Krishna, and Y.H. Lee, *Guest-editors' introduction: Real-time systems*, IEEE Computer, 24(5) 10-11, May 1991.

[63] R. M. Kieckhafer, Fault-tolerant real-time task scheduling in the M AFT distributed system, 22nd Hawaii Int'l. Conference on System Sciences, 145-151, Jan. 1989.

[64] H. Kopetz, A. Damm, C. Koza, and Mulozzani, *Distributed fault-tolerant real-time systems: The MARS approach*, IEEE Micro, 5(1), 25-40, Feb. 1989.

[65] M. Saksena, J. da Silva, and A. K. Agrawala, *Design and implementation of Maruti-II*, sang son ed., Principles of Real-Time Systems, Prentice Hall, 1994.

[66] K. G. Shin, *HARTS: A distributed real-time architecture*, IEEE Computer, 24(5), 25-35, May 1991.

[67] P. A. Lee and T. Anderson. Fault Tolerance: Principles and Practice. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition, 1990.

[68] J. Srinivasan, A. S.V., B. P., R. J., and C.-K. Hu. Ramp: A model for reliability aware microprocessor design. IBM Research Report, RC23048, 2003.

[69] X. Castillo, S. R. McConnel, and D. P. Siewiorek. *Derivation and calibration of a transient error reliability model.* IEEE Trans. Comput., 31:658-671, July 1982.

[70] R. K. Iyer, D. J. Rossetti, and M. C. Hsueh. *Measurement and modeling of computer reliability as affected by system activity.* ACM Trans. Comput. Syst., 4:214-237, August 1986.

[71] E. Bini and G. C. Buttazzo. *Measuring the performance of schedulability tests.* Real-Time Systems, 30(1-2):129-154, May 2005.

[72] D. Brooks, P. Bose, S. Schuster, H. Jacobson, P. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P. Cook. *Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors.* Micro, IEEE, 20(6):26-44, Nov 2000.

[73] R. Gupta. *Dynamic voltage scaling for system-wide energy minimization in real-time embedded systems.* Proceedings of the International Symposium on Low Power Electronics and Design ISLPED '04, pages 78-81, Aug 2004.

[74] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. *Power-aware scheduling for periodic real-time tasks.* IEEE Transactions on Computers, 53(5):584-600, 2004.

[75] H. Aydin, V. Devadas, and D. Zhu. *System-level energy management for periodic real-time tasks.* In Proc. of IEEE Real-Time Systems Symposium (RTSS), pages 313–322, Dec. 2006.

[76] V. Devadas and H. Aydin. *On the interplay of dynamic voltage scaling and dynamic power management in real-time embedded applications.* In Proc. ACM Conference on Embedded Systems Software (EMSOFT'08), 2008.

[77] D. Siewiorek and R. Swarz. *Reliable Computer Systems: Design and Evaluation.* Natick, MA: A. K. Peters, Ltd., 1998.

[78] Srinivasan J, Adve SV, Bose P, Rivers J, Hu CK. *Ramp: A model for reliability aware microprocessor design.* IBM Research Report, RC23048, 2003.

[79] Castillo X, McConnel SR, Siewiorek DP. *Derivation and calibration of a transient error reliability model.* IEEE Trans Comput 31:658–671, 1982.

[80] F. Yao, A. Demers, and S. Shenker. *A scheduling model for reduced cpu energy.* In Foundations of Computer Science, Proceedings., 36th Annual Symposium on, pages 374-382, oct 1995.

[81] Y. Zhang, K. Chakrabarty, and V. Swaminathan. *Energy-aware fault tolerance in fixed-priority real-time embedded systems.* In Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design, ICCAD'03, 2003.

[82] H. Huang and G. Quan. *Leakage aware energy minimization for real-time systems under the maximum temperature constraint.* In Design, Automation Test in Europe Conference Exhibition (DATE), 2011, pages 1-6, March, 2011.

[83] Hussein El Ghor, E. M. Aggoune. *Energy efficient scheduler of aperiodic jobs for real-time embedded systems*, International Journal of Automation and Computing, pages 1-11, 2016.

[84] Hussein EL GHOR, Maryline CHETTO. *Energy Guarantee Scheme for Real-time Systems with Energy Harvesting Constraints.* International Journal of Automation and Computing, to appear.

[85] Baoxian Zhao, Hakan Aydin and Dakai Zhu. Energy Management under General Task-Level Reliability Constraints. IEEE 18th Real Time and Embedded Technology and Applications Symposium, 2012.

[86] Zhu D, Melhem R, Mosse D. *The effects of energy management on reliability in real-time embedded systems.* In: ICCAD '04, Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design, IEEE Computer Society, Washington, DC, pp 35-40, 2004.

[87] R. Melhem, D. Mosse, and E. Elnozahy. The interplay of power management and fault recovery in real-time systems, IEEE Transactions on Computers, 53(2):217-231, Feb 2004.

[88] Y. Zhang and K. Chakrabarty. *A unified approach for fault tolerance and dynamic power management in fixed-priority real-time embedded systems.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 25(1):111-125, jan. 2006.

[89] B. Zhao, H. Aydin, and D. Zhu. *Generalized reliability-oriented energy management for real-time embedded applications.* In 48th ACM/EDAC/IEEE Design Automation Conference (DAC), pages 381-386, june 2011.

[90] Qiushi Han, Linwei Niu, Gang Quan, Shaolei Ren and Shangping Ren. *Energy efficient fault-tolerant earliest deadline first scheduling for hard real-time systems.* Real-Time Systems 50:592-619, 2014.

[91] T. D. Burd and R. W. Brodersen. *Energy efficient cmos microprocessor design.* In Proc. of The HICSS Conference, Jan. 1995.

[92] J. W. S. W. Liu. Real-time Systems, NJ, USA: Prentice Hall, 2000.

[93] P. Hazucha and C. Svensson. *Impact of cmos technology scaling on the atmospheric neutron soft error rate.* IEEE Trans. on Nuclear Science, 47(6) 2586-2594, 2000.

[94] Pradhan DK. *Fault-tolerant computer system design.* Prentice-Hall Inc, Upper Saddle River, 1996.

[95] Aydin H. Exact fault-sensitive feasibility analysis of real-time tasks. IEEE Trans Comput 56(10):1372-1386, 2007.

[96] Han, Qiushi, Energy-aware Fault-tolerant scheduling for Hard Real-time Systems. FIU Electronic Theses and Dissertations, 2015.

[97] Huang, K., Santinelli, L., Chen, J., Thiele, L., Buttazzo, G. *Adaptive dynamic power management for hard real-time systems.* In: Proceedings of the IEEE Real-Time Systems Symposium, 2009.

[98] Shin, Y., Choi, K. *Power conscious fixed priority scheduling for hard real-time systems.* In: Proceedings of the DAC, 1999.

[99] Gruian, F. *Hard real-time scheduling for low-energy using stochastic data and dvs processors.* In: Proceedings of the International Symposium on Low Power Electronics and Design, pp. 46–51. , 2001.

[100] Pop, P., Poulsen, K., Izosimov, V., Eles, P. *Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems.* In: Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis, 2007.

[101] Saewong, S., Rajkumar, R. *Practical voltage-scaling for fixed-priority rts ystems.* In: Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, 2003.

[102] Krishna, C., Lee, Y. *Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time systems.* IEEE Transactions on Computers 52 (12), 1586-1593, 2003.

[103] Perathoner, S., Chen, J., Lampka, K., Stoimenov, N., Thiele, L. *Combining optimistic and pessimistic dvs scheduling: an adaptive scheme and analysis.* In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2010.

[104] Zhao, B., Aydin, H., Zhu, D. *Enhanced reliability-aware power management through shared recovery technique.* In: Proceedings of the ICCAD, pp. 63–70, 2009.

[105] Iqbal, N., Siddique, M., Henkel, J. *Seal: Soft error aware low power scheduling by monte carlo state space under the influence of stochastic spatial and temporal dependencies.* In: Proceedings of the ICCAD, pp. 134-139, 2011.

[106] Zhu, D., Aydin, H., Chen, J. *Optimistic reliability aware energy management for real-time tasks with probabilistic execution times.* In: Proceedings of the Real-Time Systems Symposium, 2008.

[107] Pop, P., Poulsen, K., Izosimov, V., Eles, P. *Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems.* In: Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis, 2007.

[108] Shafik, R., Al-Hashimi, B., Chakrabarty, K. *Soft error-aware design optimization of low power and time-constrained embedded systems.* In: The Proceedingsof the DATE, 2010.

[109] Zhang, Y., Chakrabarty, K., Swaminathan, V. *Energy-aware fault tolerance in fixed-priority real-time embedded systems.* In: Proceedings of the ICCAD, 2003.

[110] V., Pop, P., Eles, P., Peng, Z. *Scheduling of fault-tolerant embedded systems with soft and hard timing constraints.* In: Proceedings of the DATE, 2008.

[111] Lehoczky, J., Sha, L., Ding, Y. *The rate monotonic scheduling algorithm: exact characterization and average case behavior.* In: IEEE Real-Time Systems Symposium, 1989.

[112] Contreras, G., Martonosi, M., Peng, J., Ju, R., Lueh, G., Xtrem. *A power simulator for the intel xscale core.* In: ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, 2004.

5.6.0.0.0.1