People’s Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

University of Batna 2

Faculty of Mathematics and Computer Science

Department of Computer Science

Thesis

For the Degree of Doctor of Science
In
COMPUTER SCIENCE

By

Barkahoum KADA

Fault-Tolerance and Scheduling in Embedded Real-Time Systems

Under the Supervision of: Prof. Hamoudi KALLA

Committee members:

Dr. Hocine Riadh	President	University of Batna2
Dr. Houassi Hichem	Examiner	University of Khenchela
Dr. Maarouk Toufik Messaoud	Examiner	University of Khenchela
Prof. Kalla Hamoudi	Supervisor	University of Batna2

Acknowledgement

First and foremost, I would like to thank Allah almighty for giving me the strength, knowledge, ability, and opportunity to accomplish this thesis. Praise be to Allah.

I am deeply grateful to Prof. Hamoudi Kalla, my supervisor, for his constant support, guidance and kindness during this research. This work would not have been possible without his guidance and involvement, his support, and encouragement on daily basis from the beginning till this moment. I am thankful to him for having long discussions with me and giving me invaluable suggestions which helped me to grow my understanding. I deeply express my gratitude and thank him for his support and concern.

I am happy to acknowledge my deepest sincere gratitude to Dr. Riadh Hocine for the honor that he makes to preside this jury. I also thank the members of the thesis committee: Dr. Hichem Houassi and Dr. Toufik Messaoud Maarouk for having accepted to assess my thesis.

A special thanks to my family. I deeply owe it to my parents, who motivated and helped me at every stage of my life.

I owe thanks to a special person, my husband, for his continued and unfailing love, support, and understanding during my pursuit of a Ph.D. degree that made the completion of this thesis possible. He was always there with me in the moments when there was no one to solve my difficulties. I greatly value his contribution and deeply appreciate his belief in me. Last but not the least, I appreciate my daughter and my son, for the long lasting patience and understanding they're shown during the entire Ph.D. work and thesis writing.

Abstract

Recently, fault tolerance and energy consumption have attracted a lot of interest in the design of modern embedded real-time systems. Fault tolerance is fundamental for these systems to satisfy their real-time constraints even in the presence of faults and is needed because it is practically impossible to build a perfect system. Transient faults are the most common, and their number is dramatically increasing due to the high complexity, smaller transistors sizes, higher operational frequency, and lowering voltages. Dynamic voltage and frequency scaling (DVFS) is an energy saving technology enabled on most current processors.

This work addresses the issue of fault-tolerant scheduling with energy minimization for hard real-time embedded systems. Our first proposition is an efficient fault tolerance approach that combines two well-known methods: active replication and checkpointing with rollback. Based on this approach we have proposed two algorithms. Static Fault-Tolerant Scheduling algorithm SFTS that explores hardware resources and timing constraints to tolerate multiple transient fault occurrences with respect to hard real-time constraints of precedence-constrained applications. Dynamic Voltage and Frequency Scaling Fault-tolerant Scheduling algorithm DVFS-FTS is proposed to satisfy real-time constraints and to achieve more energy saving even in the presence of faults by adapting the DVFS technique. According to the simulation results, the proposed algorithms have been shown to be very promising for emerging systems and applications where timeliness, fault tolerance, and energy reduction need to be simultaneously addressed.

Keywords: Fault Tolerance, Transient Faults, Checkpointing, Active Replication, Dynamic Voltage Frequency Scaling (DVFS), Energy Minimization.

Résumé

Récemment, la tolérance aux fautes et la consommation d'énergie ont attiré beaucoup d'intérêt dans la conception des systèmes temps-réel embarqués modernes.

La tolérance aux fautes est fondamentale pour ces systèmes pour satisfaire leurs contraintes temps-réel même en présence de fautes et elle est nécessaire car il est pratiquement impossible de construire un système parfait. Les fautes transitoires sont les plus courants et leur nombre augmente considérablement en raison de la complexité élevée, des tailles de transistors plus petites, fréquence de fonctionnement plus élevée et des tensions abaissées. DVFS est une technologie de minimisation d'énergie activée sur la plupart des processeurs actuels.

Ce travail traite le problème d'ordonnancement tolérant aux fautes avec minimisation d'énergie pour les systèmes temps-réel embarqués critiques. Notre première proposition est une approche efficace de tolérance aux fautes qui combine les deux techniques : réplication active et checkpointing. En se basant sur cette approche, nous avons proposé deux algorithmes. L'algorithme d'ordonnancement tolérant aux fautes SFTS qui explore l'architecture matérielle et les contraintes temporelles pour tolérer multiples fautes transitoires en respectant les contraintes temps-réel critiques des applications avec contraintes de précédences. L'algorithme d'ordonnancement tolérant aux fautes DVFS-FTS est proposé pour satisfaire les contraintes temporelles et minimiser la consommation d'énergie même en présence de fautes en adaptant la technique DVFS. Selon les résultats de simulation, les algorithmes proposés se sont révélés très prometteurs pour les applications où le respect des contraintes temporelles, la tolérance aux fautes et la minimisation d'énergie doivent être traitées simultanément.

Mots Clés: Tolérance aux Fautes, Fautes Transitoires, Checkpointing, Réplication Active, Stratégie d'adaptation dynamique de la tension (DVFS), Minimisation d'énergie.

مـــلخـــــص

في الآونة الأخيرة اجتذب التسامح مع الخطأ واستهلاك الطاقة الكثير من الاهتمام في تصميم أنظمة الوقت الحقيقي المدمجة الحديثة. يعد التسامح مع الخطأ أمرًا أساسيًا لهذه الأنظمة لتلبية قيود الوقت الفعلي حتى في حالة وجود الأخطاء وهي ضرورية لأنه من المستحيل عمليا بناء نظام مثالي. تعتبر الأخطاء العابرة هي الأكثر شيوعًا ويزداد عددها بشكل كبير بسبب التعقيد العالي، أحجام الترانزستورات الأصغر، تردد التشغيل العالي والجهود المنخفضة. مقياس الجهد والتردد الديناميكي (DVFS)هو تقنية موفرة للطاقة يتم تمكينها في معظم المعالجات الحالية.

يعالج هذا العمل مسألة الجدولة المتسامحة مع الأخطاء مع تقليل استهلاك الطاقة لأنظمة الوقت الحقيقي المدمجة. اقتراحنا الأول هو نهج فعال للتسامح مع الخطأ يجمع بين طريقتين معروفتين : النسخ النشط ونقاط التفتيش مع التراجع. بناءً على هذا النهج اقترحنا خوارزميتين. خوارزمية جدولة متسامحة للخطأ SFTS تستكشف هيكل المعدات والقيود الزمنية اللازمة للتسامح مع الأخطاء الانتقالية المتعددة مع احترام قيود الوقت الحقيقي الحرجة للتطبيقات المقيدة الأسبقية. و خوارزمية الجدولة المتسامحة للأخطاء DVFS-FTS لتلبية قيود الوقت وتقليل استهلاك الطاقة حتى في حالة وجود أخطاء من خلال تكييف تقنية DVFS. وفقًا لنتائج المحاكاة فقد ثبت أن الخوارزميات المقترحة واعدة جدًا للتطبيقات حيث يتم احترام ضيق الوقت ، التسامح مع الأخطاء وتقليل الطاقة في وقت واحد.

الكلمـــــات المفتاحيـــــة : التسامح مع الخطأ ، أخطاء عابرة ، نقاط التفتيش Checkpointing، النسخ المتماثل النشط Replication Active ، مقياس تردد الجهد الديناميكي (DVFS) ، تقليل استهلاك الطاقة.

Contents

List of Figures

List of tables

Abbreviations

EDF Earliest Deadline First

NFT Non Fault Tolerant

SFTS Static Fault-Tolerant Scheduling Algorithm

DVFS Dynamic Voltage and Frequency Scaling

DVFS-FTS DVFS Fault-Tolerant Scheduling Algorithm

CH Checkpointing

DAG Directed Acyclic Graph

EXH_FTS Exhaustion Fault-Tolerant Scheduling Algorithm

DVFS_CH DVFS Fault-Tolerant Scheduling Algorithm with Checkpointing

ES Energy Saving

DPM Dynamic Power Management

ABS Anti-lock Braking System

RM Rate Monotic

DMDeadline Monotic

RT Real-Time

LLF least laxity First

WCRT Worst Case Execution Time

WCCT Worst Case Communication Time

TReady Ready List

JFTT Job-Oriented Fault Tolerance with Task-level Speed Scaling

List of symbols

Notation	Definition
τ_i	The *i*th task in the task set $\Gamma = \{\tau_i \,/\, i = 1 \ldots n\}$
C_i	The worst execution time of τ_i in a fault free condition
D_i	The deadline of the task τ_i
U_i	The utilization of the task τ_i
e_{ij}	Data dependency between τ_i and τ_j
P_j	The *j*th processor in the set of processors $P = \{Pj \,/\, j=1 \ldots M\}$.
K	Number of transient faults
λ	Failure rate
θ	The criticality threshold
m_i	Number of checkpoints for the task τ_i
$C_i(m_i)$	The fault free execution time of the task τ_i using checkpointing
O_i	The time overhead for the task τ_i for saving one checkpoint
$R_i(m_i)$	The recovery time of the task τ_i with m_i checkpoints under one fault
r_i	The time overhead for the task τ_i to rollback to the latest checkpoint
$WCRT_i$	The worst case response time of the task τ_i in the presence of K faults
$\tau_i(j)$	The *j*th segment of the task τ_i
τ_i^j	The *j*th replica of the task τ_i
$P_\#$	Virtual processor.
ST_i	The start time of the task τ_i
FT_i	The finish time of the task τ_i
α	Checkpoint saving
μ	Recovery cost
D_i^{ef}	Effective deadline of the task τ_i
f_i^{opt}	Optimal Frequency

f_{max}	Maximum Frequency
C_{ef}	Effective Loading Capacitance
V	Supply Voltage
P	Power Consumption
P_S	Static Power
P_{ind}	Constant independent of operating frequency
P_d	Dynamic Power
E_i	Energy Consumption
E_{total}	Total Energy

CHAPTER 1

General Introduction

1.1. Context

With the rapid growth in technology, the contemporary computing systems available in today's era are shrinking in size and weight, exhibiting high performance and are capable of communicating with each other over the network. This has made embedded systems common place in everyday life. Unlike general purpose systems, embedded systems receive input from different sources through sensors and provide output to different devices through actuators without human intervention. These systems are used in many diverse application areas namely, automated industry applications, automotive applications, avionics, defense applications, consumer electronics etc. Many of the embedded systems are specially made for performing real-time tasks where the timing constraints are important. Such systems are known as real-time embedded systems. For example, in a missile guided system, the highly critical hard real-time tasks like target sensing and track correction require an independent system mounted on the missile to sense the target and correct the path of the missile. If these tasks are not completed in time, the missile may home onto unwanted area and cause disaster [Digalwar 2016].

Based on the cost of failure associated with not meeting timing constraints, real-time systems can be classified broadly as either hard or soft. A hard real-time constraint is one whose violation can lead to disastrous consequences such as loss of life or a significant loss to property as a missile guided system. In contrast, a soft real-time constraint is less critical; hence, soft real-time constraints can be violated. However, such violations are not desirable, either, as they may lead to degraded quality of service, and it is often the case that the extent of violation be bounded. Multimedia system is an example of system with soft real-time constraints.

Fault tolerance is fundamental for real-time systems to satisfy their real-time constraints even in the presence of faults. As shown in [Srinivasan et al. 2003], processor faults can be broadly classified into two categories: transient and permanent faults. Transient faults are the most common, and their number is dramatically increasing due to the high complexity, smaller transistors sizes, higher operational frequency, and lowering voltages [Djosic and Jevtic 2013, Salehi et al. 2016, Li et al. 2015, Krishna 2014]. They may cause errors in computation and corruption in data, but are not persistent. On the other hand, permanent faults, also called hard errors can cause hardware damages to processors and bring them to halt permanently.

Fault tolerance is essentially based on redundancy. In literature [Dubrova 2013, Motaghi and Zarandi 2014, Zhang and Chakrabarty 2006, Izosim et al, 2008], two families of redundancy are used in fault tolerant scheduling of real-time systems: **spatial redundancy** and **time-based redundancy**. Spatial redundancy is effective to tolerate multiple spatial faults (permanent or transient) and is more preferable for safety-critical systems. However, it is very costly and can be used only if the amount of resources is virtually unlimited [Pop et al. 2009].

In order to reduce cost, other techniques are required such as recovery with checkpointing and re-execution which are classified by Motaghi and Zarandi (2014) time-based redundancy. However these techniques introduce significant time overheads, where the non respect of time-constraint can lead to unschedulable solution. Therefore, the design of an efficient fault-tolerant approach is required to meet time and cost constraints of embedded systems.

Dynamic power/energy management is an active area of research in the design of embedded real-time systems. Extensive power management techniques [Tavana et al. 2014, Li et al. 2011, Gupta 2004, Hu et al. 2016, Han et al. 2015] have been developed on energy minimization for real-time systems under a large diversity of system and task models [Mahmood et al. 2017, Wei et al. 2012]. Among these techniques, Dynamic Voltage and Frequency Scaling (DVFS) is one of the most popular and widely deployed schemes. Most modern processors, if not all, are equipped with DVFS capabilities. DVFS dynamically adjusts the supply voltage and working frequency of a process to reduce power consumption at the cost of extended circuit delay.

The real-time scheduling on multiprocessor system with only the timing constraints has been identified as a NP-hard problem [Shin and Ramanathan 1994]. In addition of the two criteria: ***reliability*** *and* ***energy consumption*** *makes the real-time scheduling problem even hard to study.*

1.2. Contributions

In this thesis, we will be interested in fault-tolerant scheduling with energy minimization of hard real-time tasks with precedence constraints in multiprocessor platform. Thus, the main contributions of this thesis are:

- ✓ We design an efficient fault-tolerant scheduling approach that explores hardware resources and timing constraints. This approach combines two well-known policies: checkpointing with rollback and active replication. Replicas collaboration is introduced to tolerate spatially or temporally faults and satisfy critical task constraint. To the best of our knowledge, this is the first work introducing the idea of collaboration between replicas in active replication technique with checkpointing. The proposed approach classifies the real-time tasks into critical and noncritical ones, according to the utilization of the task. For the non critical task, we adopt checkpointing with rollback technique to tolerate multiple transient faults. Whereas for the critical task, we adopt active replication as it is the fault-tolerant method that explores hardware resources to meet timing constraints and provide high reliability even when deadlines are tight.
- ✓ Based on the explained fault-tolerance approach, we have proposed a fault-tolerant scheduling algorithm **SFTS** which can tolerate *K* transient faults.
- ✓ We investigate the energy minimization problem for fault-tolerant scheduling of hard real-time systems. We extend the proposed fault-tolerance approach to incorporate it with DVFS to exploit the released slack time for energy saving. DVFS is used during uniform checkpointing with rollback technique. However, with active replication, task replicas must be performed at the maximum frequency given the probability of failure is low.
- ✓ An efficient fault-tolerant scheduling heuristic **DVFS_FTS** based on the Earliest-Deadline-First (EDF) algorithm is presented to minimize energy consumption while tolerating *K* transient Faults.

1.3. Thesis organization

The rest of this dissertation is organized as follows:

Chapter 2 introduces the basic concepts of real-time system and embedded system. It presents their characteristics, architectures and the classification of real-time scheduling.

Chapter 3 provides in the first part an overview of dependability characteristics (attributes, impairments and means) and the different classes of faults. The second part is dedicated to our principle aim: fault tolerance. We present their different techniques and the principle classes of redundancy in real-time systems (spatial redundancy and time-based redundancy).

Chapter 4 provides an overview of related work on multiprocessor scheduling, fault tolerance, and energy consumption in embedded real-time systems.

Chapters 5 and 6 are devoted to the main contributions of this dissertation

Chapter 5 focuses on the choice of fault-tolerant mechanisms that ensure our system reliability. It starts with a general description of our system model (Application, Architecture, and fault model). Then, we concentrate on describing our fault tolerance approach based on active replication and uniform checkpointing with rollback. After, we exploit this approach in the first proposed fault-tolerant scheduling algorithm **SFTS**. Finally, simulation results are given to prove the performance of the proposed algorithm.

Chapter 6 is dedicated to another challenge of real-time embedded systems: **energy minimization**. We extend the proposed fault-tolerance approach in chapter 5 to incorporate it with DVFS to achieve more energy saving. Then, we present the fault tolerant scheduling algorithm **DVFS_FTS** developed for reducing dynamic energy. Finally, Experiment results have shown that the proposed algorithm achieves a considerable amount of energy saving compared to others algorithms.

Chapter 7 concludes the thesis by discussing the overall contribution of the research. In addition, it discusses limitations of the work and points to future research directions.

CHAPTER 2

Real-Time Systems

2.1. Introduction

The distinguishing characteristic of a real-time system in comparison to a non-real-time system is the inclusion of timing requirements in its specification. That is, the correctness of a real-time system depends not only on logically correct segments of code that produce logically correct results, but also on executing the code segments and producing correct results within specific time frames. Thus, a real-time system is often said to possess dual notions of correctness, logical and temporal.

In this chapter, we present first the basic concepts of real-time systems and their classification. Then, we provide classes of real-time scheduling. We focus in this thesis on real-time embedded systems. Finally, we describe some basic concepts pertain to embedded systems.

2.2. Definition

The Oxford dictionary defines real-time as "*the actual time during which a process or event occurs*". In computer science, real-time systems are defined by Burns and Wellings (2001) "*those systems in which the correctness of the system depends not only on the correctness of logical result of computation, but also on the time on which results are produced*". The validity of a real-time system depends not only on the results of the processing performed but also on the temporal aspect.

Recently, the term real-time is widely used to describe many applications and computing systems that are somehow related to time, such as real-time trackers, gaming systems and information services. The following list contains certain examples of practical real-time applications:

- Mobile and communication systems.

- Multimedia and entertainment systems: multimedia information is in the form of streaming audio and video.
- Data distribution systems which notify users of important information in a short delay (few minutes or less). Such systems are found mainly in transport systems to inform passengers of accidents and schedule delays or changes.
- General purpose computing such as in financial and banking systems.
- Medical systems such as peacemakers and medical monitors of treatments or surgical procedures.
- Industrial automation systems such as the ones found in factories to control and monitor production process. For example, sensors collect parameters periodically and send them to real-time controllers, which evaluate the parameters and modify processes when necessary. These systems can handle non-critical activities as in logging and surveillance.
- General control management systems such as the ones found in avionic systems. Real-time engine controllers are responsible of automatic navigation and detection of hardware malfunctions or damages through reading sensors and processing their parameters and react within an acceptable delay.

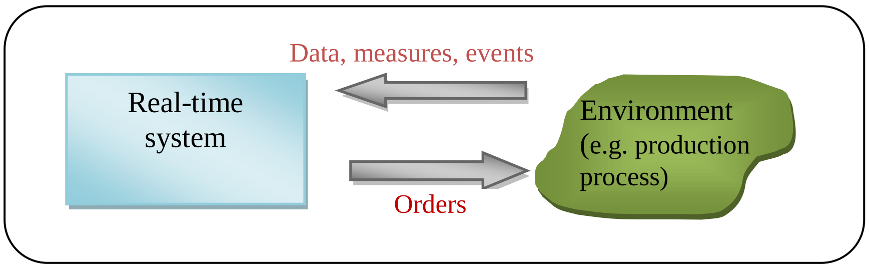


Figure 2-1 Real-time system

2.3. Classfication of real-time systems:

Depending on the criticality of the timing constraints, three categories of real-time system can be distinguished:

2.3.1. Hard real-time system:

The correctness of their outputs depends on respecting given timing constraints or catastrophic results occur. If such systems fail in performing their tasks within acceptable deadline margins, their results become useless and might lead to catastrophic consequences.

It’s a system subject to strict timing constraints, that is to say for which the slightest temporal error can have catastrophic human or economic consequences. Air traffic control systems and nuclear station control systems are real-time strict.

2.3.2. Soft real-time system:

Soft real-time systems have flexible timing constraints and they perform less critical activities and tasks. The quality of services provided by soft real-time systems depends on providing results within a minimum delay. If such delay is not respected, the quality degrades but not the correctness of the execution or results.

2.3.3. Mixed critical system:

They are defined by [Saraswat et al. 2010] and [Izosimov 2008] the systems with tasks of different levels of time-criticality, for example running hard real-time and soft real-time tasks in the same system.

2.4. Real-time task:

A real-time task is a sequence of instructions that is the basic unit of a real-time system. The tasks perform inputs / outputs and calculations to control processes via a set of sensors and actuators, possibly all or nothing, for example set of tasks performing the speed controller of a car or the automatic control of a plane.

2.4.1. Real-time task characteristics:

A real-time application is composed of a set of n tasks denoted by Γ, where $\Gamma = \{\tau_1, \tau_2, \dots, \tau_n\}$. Generally, a real-time task is described by the following parameters (all these parameters are illustrated in Figure 2-2)

- ***R***

- **R_i (Ready time or Release time):** it is the time on which the task τ_i can begin its execution;
- **ST_i (Start time), FT_i (End time):** are respectively the time on which the task τ_i is executed on the processor also called the start time of execution and the time on which the task τ_i finishes its execution also called the end time of execution;
- **RT_i (Response time):** it represents FT_i -R_i;
- ***WCRT (Worst Case Response time) C_i*:** which is an estimation of the longest possible execution time of any task τ_i, i.e., the actual execution time of a task should never exceed its WCRT in any scenario. The evaluation of WCRT of tasks is very important for the reliability of real-time systems to be valid, the value of this parameter must not be overestimated too much, must be safe (never overestimated) and the pessimism of their estimations increases relatively to the criticality of the application [Qamhieh 2015];
- ***D_i (Deadline):*** which is the time interval in which each task executes with respect to its release time. In hard real-time systems, any task must always meet their deadline, whereas execution tardiness of task is accepted in soft real-time systems. Two types of deadlines exist:
    - ***Relative D_i*:** the time interval between the start of the task and the completion of the real-time task is known as relative deadline. It is basically the time interval between arrival and corresponding deadline of the real-time task.
    - ***Absolute D_i +R_i*:** the time within which execution of a task should be completed.
- ***L_i (Laxity):*** this is the largest time for which the scheduler can safely delay the task execution before running it without any interruption.
- ***T_i (Period):*** which is the minimum inter-arrival time between two releases of the same task.

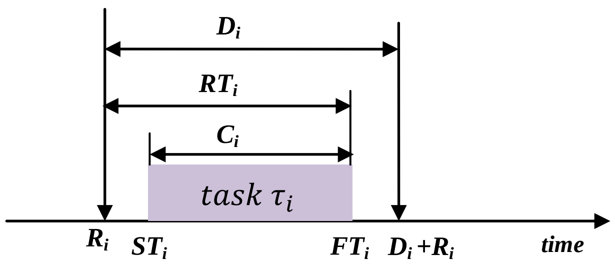


Figure 2-2 Real-time task parameters

The processor utilization of task τ_i is defined as the task's processor usage and it is denoted by $U_i = \frac{C_i}{D_i}$. The utilization U of a task set Γ is the sum of utilization of its tasks, where $U = \sum_{i=1}^{n} U_i$

2.4.2. States of real-time task

The main objective of a real-time scheduler is to guarantee the correctness of the results while respecting the timing constraints of the tasks (no deadline miss).

Based on the decisions of the scheduler, a real-time task can be in one of the following states:

- **Ready state:** The task is activated and it is available for execution, but it is not currently selected by the scheduler to execute on a processor.
- **Running state:** The task is assigned to a processor and it is actually executing.
- **Blocked state:** if the task is waiting for an event to happen such as an I/O event, it remains blocked and cannot be scheduled until the event happens. Then the task moves to the ready state.

The different states of tasks are shown in Figure 2-3. Moreover, a real-time scheduler controls the transitions between the ready and running states of tasks, but it has no control over the external events that block the execution of tasks.

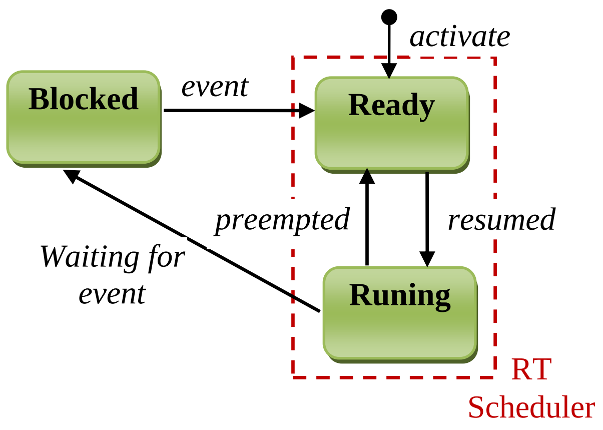


Figure 2-3 Different states of a real-time task [Qamhieh 2015]

2.4.3. Types of real-time tasks

There exist three types of real-time tasks:

2.4.3.1. Period task:

A task τ_i is called periodic if the event that conditions its activation occurs at regular intervals of time (period) Ti and each activation is called instance.

2.4.3.2. Aperiodic task:

The activation time is random and can not be anticipated, since its execution is determined by the occurrence of an internal event (for example the arrival of a message) or an external event (e.g. the requests of the operator). Anti-lock Braking System (ABS) in modern cars is a typical system that employs aperiodic real-time tasks.

2.4.3.3. Sporadic task:

It's a special case of aperiodic tasks where a minimum duration of time separates two successive activations. To take them into account, these tasks are often considered as periodic tasks [Kermia 2009] to apply the existing results of the periodic tasks.

2.4.4. Precedence Constraints and Dependencies

A dependency between two tasks τ_i and τ_j can be of two types: a precedence dependency and / or a data dependency. A precedence dependence between (τ_i , τ_j) means that the task τ_j cannot begin its execution until the task τ_i has been completed. Precedence constraints are indirectly real-time constraints and we say that the task τ_i is a predecessor of the task τ_j and τ_j is a successor of τ_i [Forget 2011].

A data dependency between (τ_i , τ_j) indicates that the task τ_i produces a data that is consumed by τ_j . This dependence necessarily leads to precedence between the tasks. The tasks are said to be independent when they are defined only by their temporal parameters [Ndoye 2014].

The set of dependencies between the tasks can be modeled by a Directed Acyclic Graph DAG where the nodes represent the tasks and the arcs the dependencies between the tasks. An example of DAG is shown in the Figure 2-4.

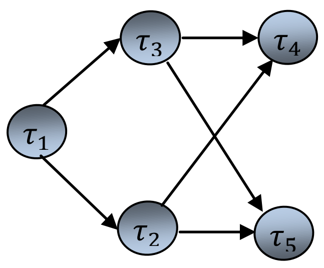


Figure 2-4 Precedence constraints example

In this thesis, we are interested in hard real-time systems with aperiodic dependant tasks.

2.4.5. Makespan

Reflects the time that elapses between the start date of the first executed task and the finish date of the last executed task. The goal is to develop algorithms which in addition to respecting other time constraints, minimize makespan [Lin and Liao 2008].

2.5. Real-time scheduling classification

Real-time scheduling is defined as the process that defines the execution order of tasks on processor platforms. There are several classes of real-time scheduling algorithms, we can cite the following classification [Yahiyaoui 2013]:

2.5.1. Uniprocessor/Multiprocessor:

Real-time scheduling is said uniprocessor scheduling if the architecture has only one processor. If multiple processors are available, the scheduling is multiprocessors.

2.5.2. Off-line/On-line:

In off-line scheduling, the schedules for each task need to be determined in advance, therefore it requires prior knowledge of the characteristics of tasks. It only incurs little runtime overhead. In contrast, on-line scheduling calculates the schedules during runtime, hence it can provide more flexibility to react to uncertainties of task characteristics at the cost of large runtime overhead.

2.5.3. Preemptive/Non-Preemptive:

A scheduling is preemptive if the execution of any task can be interrupted to requisition the processor for another more urgent or higher priority task. The scheduling is said to be non-preemptive if, once started, the task being executed cannot be interrupted before the end of its execution.

2.5.4. Static/Dynamic priorities:

Most scheduling algorithms are priority-based: they assign priorities to the tasks in the system and these priorities are used to select a task for execution whenever scheduling decisions are made. A scheduling algorithm is called static priority algorithm if there is a unique priority associated with each task. e.g. of such algorithms is Rate Monotic (RM).

A scheduling algorithm has dynamic priorities, if the priorities of the tasks are based on dynamic parameters (for example laxity). e.g. of such category is the Least Laxity First (LLF) scheduling algorithm.

These classes of real-time systems are illustrated in Figure 2-5.

2.5.5. Feasibility and optimality:

A task is referred to as **schedulable** according to a given scheduling algorithm if its worst-case response time under that scheduling algorithm is less than or equal to its deadline. Similarly, a task set is referred to as schedulable according to a given scheduling algorithm if all of its tasks are schedulable.

A task set is said to be **feasible**, if there is at least one scheduling algorithm that can schedule the task set while meeting all task deadlines [Legout 2014].

Additionally, a scheduling algorithm is referred as **optimal** if it can schedule all of the task sets that can be scheduled by any other algorithm. In other words, all of the feasible task sets [Zahaf 2016].

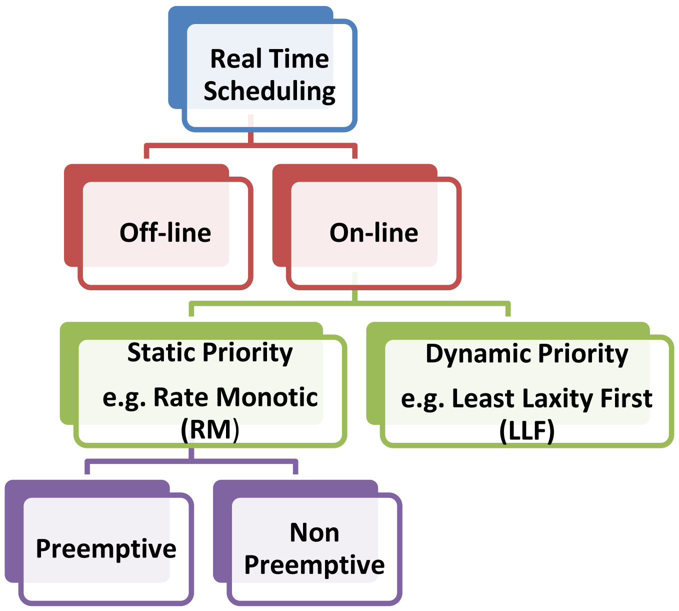

Real Time Scheduling
Off-line
On-line
Static Priority e.g. Rate Monotic (RM)
Dynamic Priority e.g. Least Laxity First (LLF)
Preemptive
Non Preemptive

Figure 2-5 Classification of real-time scheduling

2.6. Embedded Systems

In this thesis the applications that interest us are real-time and also embedded. This forces us to take into account the properties of these systems in the work that we carry out.

2.6.1. Definition1

An embedded system can be broadly defined as a device that contains tightly coupled hardware and software components to perform a single function, forms part of a larger system, is not intended to be independently programmable by the user, and is expected to work with minimal or no human interaction [Jimenez 2014].

2.6.2. Definition2

An embedded system is a combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a specific function.

Most embedded systems interact directly with processes or the environment, making decisions on the fly, based on their inputs. This makes necessary that the system must be reactive, responding in real-time to process inputs to ensure proper operation. Besides, these systems operate in constrained environments where memory, computing power, and power supply are limited. Moreover, production requirements,

2.6.3. Application of embedded systems

Embedded systems are used in different applications like automobiles, telecommunications, smart cards, missiles, satellites, computer networking and digital consumer electronics (see Figure 2-6).

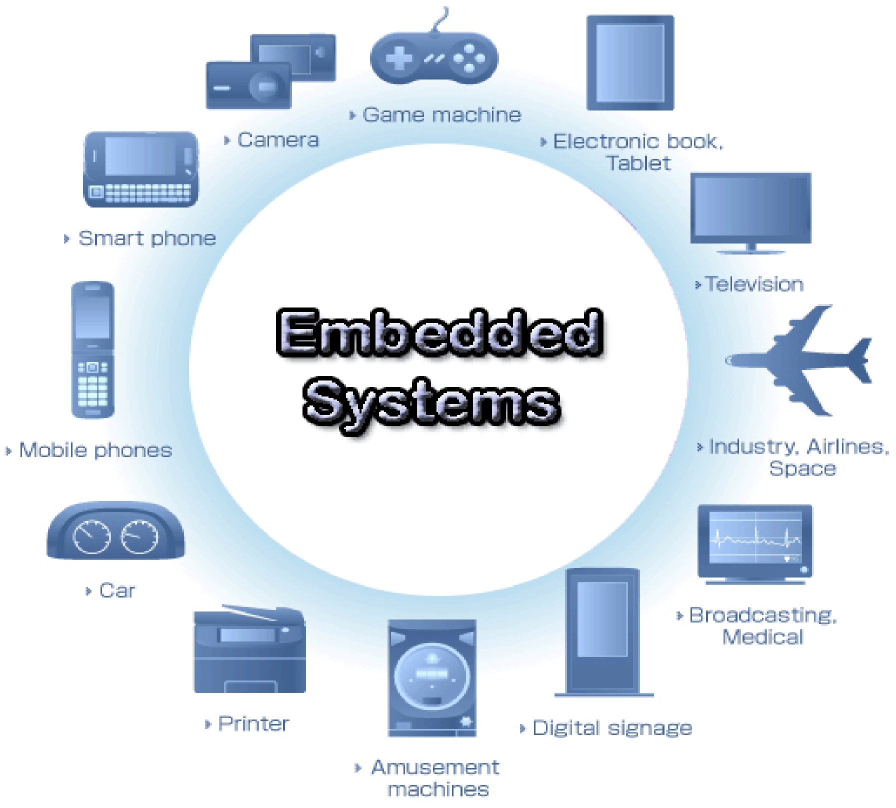


Figure 2-6 Examples of embedded systems

2.6.4. Embedded system architecture

It consists of a hardware part, which interacts with environment and formed by a set of physical elements: processor(s), memory(s) and inputs/outputs. At the same time, a specific software part which consists of programs and a power source.

Embedded systems sometimes require the use of several processors which can be of different types. A first classification of architectures for embedded systems depends on the number of processors: single-processor or multi-processor architecture. There are different classifications for multiprocessor architectures:

Homogeneous / Heterogeneous in relation to the nature of the processors available to the architecture:

- **Homogeneous**: In this case the processors are identical .i.e. they are interchangeable and they have the same computing capacity;
- **Heterogeneous**: The processors are either independent .i.e. processors are not intended to perform the same tasks or uniform .i.e. the processors perform the same tasks but each processor has its own computational capacity.

Homogeneous / Heterogeneous depending on the nature of communications between processors:

- **Homogeneous**: If the communication costs between each pair of processors in the architecture are always the same;
- **Heterogeneous**: If the communication costs between processors vary from one pair of processors to another.

Parallel / Distributed according to the type of memory available to the architecture:

- **Parallel**: This architecture model corresponds to a set of processors communicating by shared memory (see Figure 2-7);
-

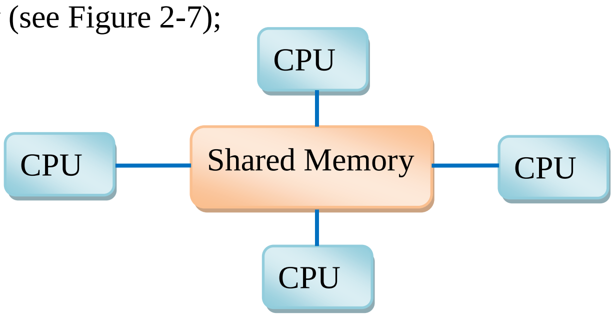


Figure 2-7 Parallel architecture

- **Distributed**: It corresponds to a set of distributed memory processors communicating by messages (see Figure 2-8).

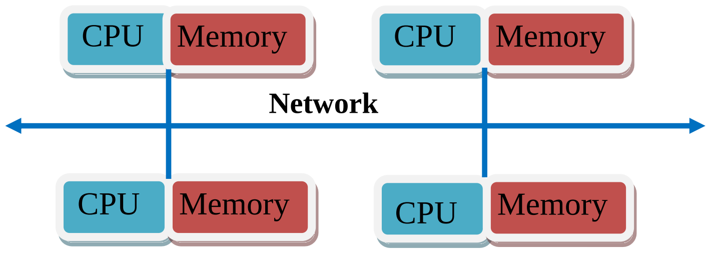


Figure 2-8 Distributed architecture

As represented in Figure 2-9, multiprocessor architectures are often represented by a graph where the vertices are the processors. If an arc connects two vertices, this means that these two vertices can communicate directly through the communication medium (bus, memory ...).

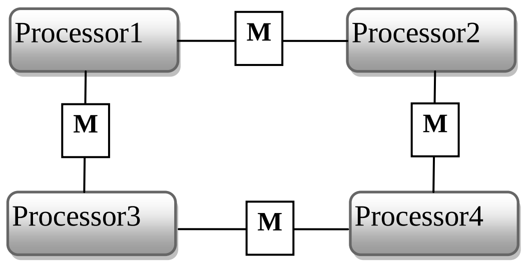


Figure 2-9 Example of multiprocessor architecture

2.6.5. Characteristics of Embedded Systems

The design of an embedded system must respect a certain number of characteristics, we list below the most important:

- **Must be dependable:**
  - **Reliability**: R(t) = Probability of system working correctly provided that is was working at t=0.
  - **Maintainability**: M(d) = Probability of system working correctly d time units after error occurred.
  - **Availability**: Probability of system working at time t
  - **Safety**: No harm to be caused.
  - **Security**: Confidential and authentic communication.
- **Must be efficient:**
  - Energy efficient.
  - Code-size efficient (especially for systems on a chip).
  - Run-time efficient.
  - Weight efficient.
  - Cost efficient.
- **Many Embedded System must meet real-time constraints:**

- A real-time system must react to stimuli from the controlled object (or the operator) within the time interval dictated by the environment.
- For real-time systems, right answers arriving too late (or even too early) are wrong.

2.7. Conclusion

We have presented in this chapter real-time and embedded systems, their characteristics, application, and classification. In this thesis, we have considered critical real-time systems, i.e. those which must satisfy the time constraints to prevent the system from the various possible disasters. As the main characteristic of these systems is to be reliable, we will present in the following chapter the basic concepts of fault tolerance and dependability.

CHAPTER 3

Dependability and Fault Tolerance

3.1. Introduction

Fault tolerance is the ability of a system to continue performing its intended function in spite of faults. In a broad sense, fault tolerance is associated with reliability, with successful operation, and with the absence of breakdowns. A fault-tolerant system should be able to handle faults in individual hardware or software components, power failures or other kinds of unexpected disasters and still meet its specification. The ultimate goal of fault tolerance is the development of a dependable system.

The first part of this chapter begins by defining what dependability is. Then, we describe the basic concepts in the field of dependability (attributes, impairments and means) and identify the different classes of faults.

The second part is dedicated to our principle objective: fault tolerance. We present their different techniques and the principle classes of redundancy in real-time systems (spatial redundancy and time-based redundancy).

3.2. Dependability

Dependability is the ability of a system to deliver its intended level of service to its users [Krakowiak 2004]. As computer systems become relied upon by society more and more, the dependability of these systems becomes a critical issue. In the next, we present three characteristics of dependability as shown in Figure 3-1: attributes, impairments and means.

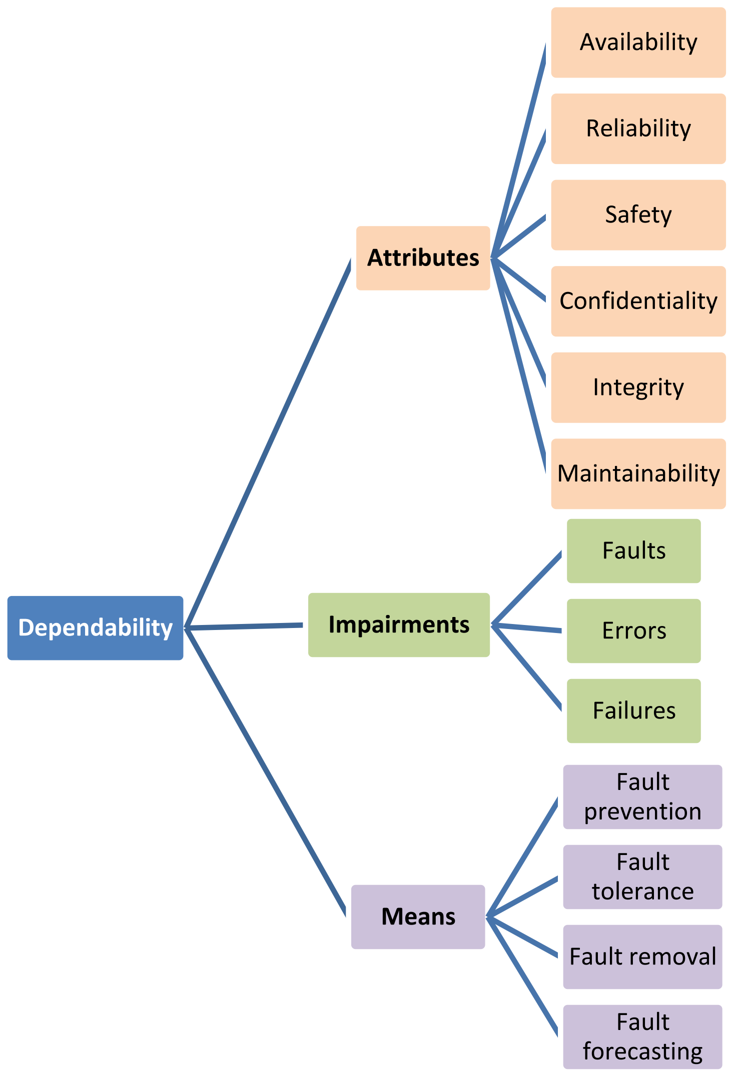


Figure 3-1 Dependability characteristics

3.2.1. Dependability attributes:

The attributes of dependability express the properties which are expected from a system:

- **Reliability**: The ability of the system to deliver its service without interruption.
- **Safety**: The ability of the system to perform its functions correctly or to discontinue its function in a safe manner.
- **Availability:** The proportion of time which system is able to deliver its intended level of service.
- **Confidentiality:** The absence of unauthorized disclosure of information.
- **Integrity:** The absence of inappropriate alterations to information leads to integrity
- **Maintainability:** The ability to undergo modifications and repairs.

3.2.2. Dependability impairments:

Dependability impairments are usually defined in terms of faults, errors, failures which are linked as illustrated in Figure 3-2. A common feature of the three terms is that they give us a message that something went wrong. A difference is that, in case of a **fault**, the problem occurred on the physical level; in case of an **error**, the problem occurred on the computational level; in case of a **failure**, the problem occurred on a system level.

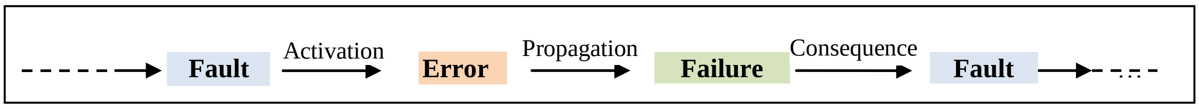


Figure 3-2 Dependability impairments

3.2.2.1. Fault:

A fault is a physical defect, or flaw that occurs in some hardware or software component. Examples are short-circuit between two adjacent interconnects, broken pin, or a software bug.

Classes of faults

As shown in Figure 3-3, the work done by [Avizienis et al. 2004] classifies all faults according to eight basic viewpoints. Each of these eight classes is called an elementary fault class.

Classification According to Phase of Creation or Occurrence

The lifecycle of a system consists of a development phase and a use phase. The development phase involves all activities that lead to the system being ready to deliver its service for the first time, from the conception of an initial idea, to a specification, to a design, to the manufacturing, and to the final deployment. The use phase involves everything that happens after the system has been deployed and consists of alternating periods of service delivery, service outage and service shutdown (an intentional and authorized interruption of the service). Faults can be introduced into a system during either phase. Faults introduced during the development phase are called **development faults**. Faults introduced during the use phase are called **operational faults**.

Classification According to System Boundaries

A system is separated from its environment by a common frontier called the system boundary. Based on this boundary, faults can be classified according to whether they originate within it or outside of it. Faults originating within the system boundary are called internal faults. Faults originating outside of it are called external faults. What exactly is an internal or external fault therefore depends on where we trace the system boundary. The physical deterioration of a component would be an example of an internal fault. The failure of a cooling system that is part of the environment, and whose purpose is to prevent overheating of the system, would be an example of an external fault.

Classification According to the Phenomenological Cause

Another way to classify faults is whether they can be attributed to people or whether they are due to natural phenomena. Using this criterion, we distinguish between **human-made faults**, which are those for which we can blame a person, and **natural faults**, which are those for which we will have to blame natural phenomena.

Classification According to Dimension

The dimension of a fault refers to whether it affects hardware or software. In the first case we talk about hardware faults; in the second, we talk about software faults. Examples of hardware faults include the deterioration of physical parts, loose connectors, broken wires, and manufacturing defects. Examples of software faults include typos in source code and incorrectly implemented functions.

Classification According to Objective

Human-made faults, which we saw a moment ago, can be classified according to their objective. In that case we distinguish between malicious and non-malicious faults. **Malicious faults** are those that are introduced with the objective to cause harm to the system or its environment. **Non-malicious faults**, unsurprisingly, are those introduced without the intent to cause harm. Examples of malicious faults include Trojan horses, trapdoors, viruses, worms, zombies, and wiretapping. Examples of non-malicious faults include any honest mistake when designing, deploying or using a system.

Classification According to Intent

Another way of classifying human-made faults is according to their intent. Here we distinguish between deliberate faults and non-deliberate faults. To decide whether a fault is deliberate or non-deliberate, we basically have to ask the person that just introduced the fault “did you do that on purpose?”. If the answer is yes, we have a deliberate fault; otherwise, we have a non-deliberate fault. An example is when a designer purposely chooses not to add any electromagnetic shielding to reduce the weight or cost of a system. Depending on the electromagnetic harshness of the environment where the system needs to operate, this may be a fault

Classification According to Capability

Avižienis et al. 2004 also classify human-made faults according to the capability (or competence) of the person introducing the fault. Using this criterion we distinguish between accidental faults and incompetence faults. **Accidental faults** are those introduced inadvertently, presumably due to a lack of attention and not due to a lack of skills; whereas **incompetence faults** are those introduced due to a lack of skills.

Classification According to Persistence

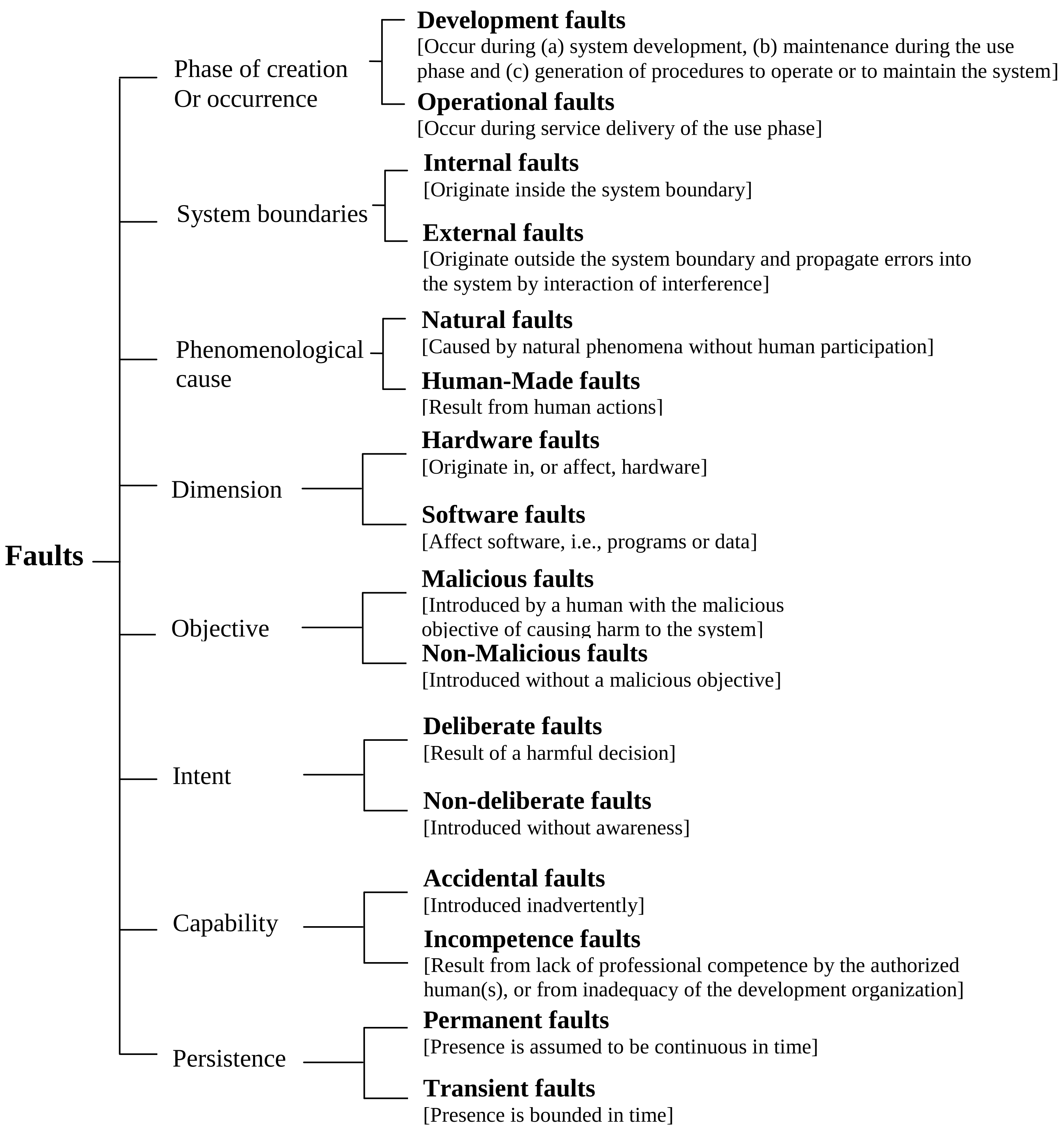
Finally, Avizienis et al. (2004) classify faults according to their persistence: faults can either be **permanent faults**, meaning that once present, they do not disappear again without external interventions such as repairs; or they can be **transient faults**, meaning that they disappear after some time. An example of a permanent fault is a deteriorated component. An example of a transient fault is an electromagnetic interference.

3.2.2.2. Error:

An error is a deviation from correctness or accuracy in computation, which occurs as a result of a fault. Errors are usually associated with incorrect values in the system state. For example, a circuit or a program computed an incorrect value, an incorrect information was received while transmitting data.

3.2.2.3. Failure:

A failure is a non-performance of some action which is due or expected. A system is said to have a failure if the service it delivers to the user deviates from compliance with the system specification for a specified period of time. A system may fail either because it does not act in accordance with the specification, or because the specification did not adequately describe its function.



- **Faults**
  - Phase of creation Or occurrence
    - **Development faults** [Occur during (a) system development, (b) maintenance during the use phase and (c) generation of procedures to operate or to maintain the system]
    - **Operational faults** [Occur during service delivery of the use phase]
  - System boundaries
    - **Internal faults** [Originate inside the system boundary]
    - **External faults** [Originate outside the system boundary and propagate errors into the system by interaction of interference]
  - Phenomenological cause
    - **Natural faults** [Caused by natural phenomena without human participation]
    - **Human-Made faults** [Result from human actions]
  - Dimension
    - **Hardware faults** [Originate in, or affect, hardware]
    - **Software faults** [Affect software, i.e., programs or data]
  - Objective
    - **Malicious faults** [Introduced by a human with the malicious objective of causing harm to the system]
    - **Non-Malicious faults** [Introduced without a malicious objective]
  - Intent
    - **Deliberate faults** [Result of a harmful decision]
    - **Non-deliberate faults** [Introduced without awareness]
  - Capability
    - **Accidental faults** [Introduced inadvertently]
    - **Incompetence faults** [Result from lack of professional competence by the authorized human(s), or from inadequacy of the development organization]
  - Persistence
    - **Permanent faults** [Presence is assumed to be continuous in time]
    - **Transient faults** [Presence is bounded in time]

Figure 3-3 Fault classes [Avizienis et al. 2004]

3.2.3. Dependability means:

Dependability means are the methods and techniques enabling the development of a dependable system. Fault tolerance, which is the subject of this thesis, is one of such methods.

It is normally used in a combination with other methods to attain dependability, such as fault prevention, fault removal and fault forecasting.

3.2.3.1. Fault prevention:

Fault prevention is attained by quality control techniques employed during the design and manufacturing of hardware and software. They include structured programming, information hiding, modularization, etc., for software, and rigorous design rules for hardware.

3.2.3.2. Fault tolerance:

Fault tolerance is intended to preserve the delivery of correct service in the presence of active faults. Fault tolerance is achieved by using some kind of redundancy.

3.2.3.3. Fault removal:

Fault removal is performed both during the development phase, and during the operational life of a system. Fault removal during the development phase of a system life-cycle consists of three steps: **verification**, **diagnosis**, **correction**. Fault removal during the operational life of a system is **corrective or preventive maintenance**. **Corrective maintenance** is aimed at removing faults that have produced one or more errors and have been reported, while **preventive maintenance** is aimed to uncover and remove faults before they might cause errors during normal operation.

3.2.3.4. Fault forecasting:

Fault forecasting is conducted by performing an evaluation of the system behavior with respect to fault occurrence or activation. Evaluation has two aspects:

- **Qualitative or ordinal evaluation**, which aims to identify, classify, rank the failure modes, or the event combinations (component failures or environmental conditions) that would lead to system failures,
- **Quantitative or probabilistic evaluation**, which aims to evaluate in terms of probabilities the extent to which some of the attributes of dependability are satisfied; those attributes are then viewed as measures of dependability.

The main objective of this dissertation is fault tolerance in real-time embedded systems. Therefore, the next section elaborates more on fault tolerance in general, and the specific techniques used in this thesis in particular.

3.3. Fault tolerance:

Whatever precautions are taken, the occurrence of faults is inevitable (human error, malicious intent, aging of equipment, natural disaster, etc.). This does not mean that one should not try to prevent or eliminate faults, but the measures taken can only reduce the likelihood of their occurrence. Several techniques of fault tolerance have been proposed, they are all based on redundancy. In the next, we first present fault tolerance techniques and then we describe the different types of redundancy.

3.3.1. Fault tolerance techniques

The goal of fault tolerance is to provide a correct system service in spite of faults. As shown in Figure 3-4, Fault tolerance is carried out by **error processing** and by **fault treatment** [Derasevic 2018]. Error processing is aimed at removing errors from the computational state, if possible before failure occurrence; fault treatment is aimed at preventing faults from being activated again.

3.3.1.1. Error processing

May be realized by using the following three primitives [Laprie 1995]:

- ***Error detection:*** It is done by identifying the erroneous state before replacing it with an error-free one
- ***Error recovery:*** It is done by restoring an error-free state starting from the erroneous state. This can be achieved using two different approaches:
    - *Backward recovery* is done by restoring the system to a prior error-free state using the pre-saved points in time, called the recovery points (checkpoints) that were established before the error has occurred.

- *Forward recovery* is done by transforming an erroneous state with a new state in which the system may resume to provide its service, but possibly in a degraded mode.

- ***Error compensation:*** It is done by employing enough redundancy to allow the system to provide its service in spite of the erroneous internal state.

3.3.1.2. Fault treatment

Is accomplished by the execution of two subsequent steps. The first step is called *fault diagnosis* and it involves discovering what are the causes of errors covering their location and nature. The next step is called *fault passivation* and its aim is to realize the prime goal of fault treatment which is to prevent faults from causing any further errors, i.e. to passivate them. This step is accomplished by excluding the identified faulty components from the rest of the system execution. If this exclusion causes the system not to be able to preserve the delivery of intended service, then a *reconfiguration* of the system might be realized [Derasevic 2018].

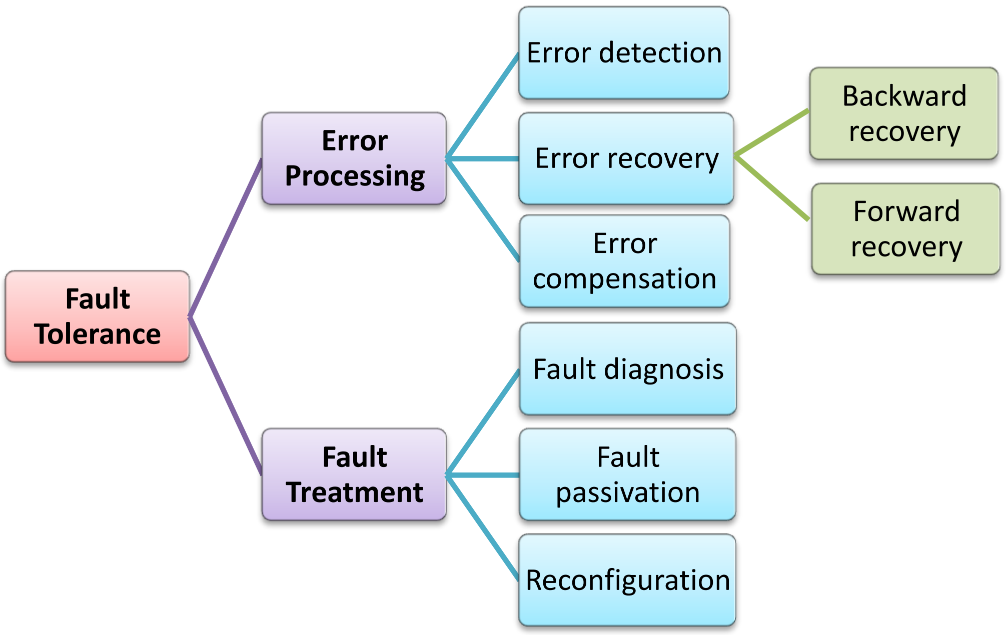


Figure 3-4 Fault tolerance techniques

3.3.2. Error detection techniques

In order to achieve fault tolerance, a first requirement is that faults have to be detected. Researchers have proposed several error detection techniques, including watchdogs, assertions, signatures, duplication, and memory protection codes.

Signatures [Oh et al. 2002a, Nicolescu et al. 2004]: Are among the most powerful error detection techniques. In this technique, a set of logic operations can be assigned with precomputed “check symbols” (or “checksum”) that indicate whether a fault has happened during those logic operations. Signatures can be implemented either in hardware, as a parallel test unit, or in software. Both hardware and software signatures can be systematically applied without knowledge of implementation details.

Watchdogs: In the case of watchdogs [Benso et al. 2003, Kalla 2004, Bachir 2019], program flow or transmitted data is periodically checked for the presence of faults. The simplest watchdog schema, watchdog timer, monitors the execution time of processes, whether it exceeds a certain limit.

Assertions [Peti et al. 2005]: Are an application-level error detection technique, where logical test statements indicate erroneous program behavior (for example, with an “if” statement: if not then). The logical statements can be either directly inserted into the program or can be implemented in an external test mechanism. In contrast to watchdogs, assertions are purely application-specific and require extensive knowledge of the application details. However, assertions are able to provide much higher error coverage than watchdogs.

Duplication: If the results produced by duplicated entities are different, then this indicates the presence of a fault. Examples of duplicated entities are duplicated instructions [Oh et al. 2002b], procedure calls [Oh et al. 2002c], functions and whole processes. Duplication is usually applied on top of other error detection techniques to increase error coverage.

Other error detection techniques: There are several other error detection techniques, for example, Memory protection codes, transistor-level current monitoring, or the widely used parity-bit check. Therefore, several error detection techniques introduce an error detection overhead, which is the time needed for detecting faults. In our work, unless other specified, we account the error-detection overhead in the worst-case execution time of tasks.

3.3.3. Redundancy for fault tolerance

As defined by [Zammali 2016], fault tolerance aims to avoid failures despite the faults present, and is essentially based on redundancy. Redundancy is when you create multiple copies of a component (hardware, software, data, etc.) or run so that the copy performs the same function, service, or role as the original component (or execution).

According to [Dubrova 2013, Motaghi and Zarandi 2014, Zhang and Chakrabarty 2006, Izosim et al, 2008] two families of redundancy are used in fault tolerant real-time systems (Figure 3-5): the family of **spatial redundancy** and the family of **temporal redundancy (time-based redundancy).**

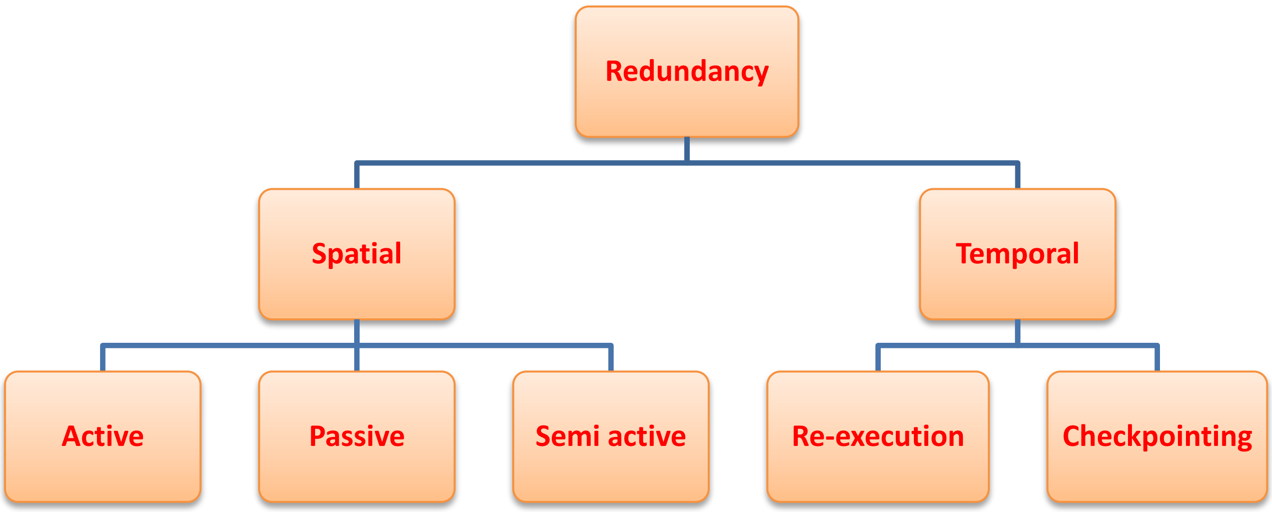


Figure 3-5 Different types of redundancy in real-time systems

3.3.3.1. Time-based redundancy methods:

In time-redundancy method additional time is spent to recompute a failed computation on the same hardware. This scheme works well for the transient faults as they are not likely to repeat during recomputation [Nikolov 2015].

RE-EXECUTION

The simplest fault tolerance technique to recover from fault occurrences is re-execution [Izosim 2009]. With re-execution, a task is executed again if affected by faults. The time needed for the detection of faults is accounted for by error detection overhead. When a task is re-executed after a fault has been detected, the system restores all initial inputs of that task. The task re-execution operation requires some time for this, which is captured by the recovery overhead. In order to be restored, the initial inputs to a task have to be stored before the process is executed for first time.

ROLLBACK RECOVERY WITH CHECKPOINTING

The time needed for re-execution can be reduced with more complex fault tolerance techniques such as rollback recovery with checkpointing [Izosim et al. 2008]. The main principle of this technique is to restore the last non-faulty state of the failing task. The last non-faulty state, or checkpoint, has to be saved in advance in the static memory and will be restored if the task fails.

3.3.3.2. Spatial redundancy methods:

Spatial redundancy is mainly deployed in safety domains like avionics [Runge 2012]. It tolerates both permanent and transient faults and has the advantage of simplified fault detection. However, it comes with high design and production cost [Huang et al. 2011].

Active redundancy: All replicas of the same task are executed simultaneously on different processors with the objective of providing the same result. This strategy avoids the use of costly checkpoints. However, it requires that result execution be deterministic to ensure consistency.

Passive redundancy: Also known as primary-backup. Only one replica, called primary, makes all the decisions, and sends updates to the other replicas, called backups, which then apply the changes.

Semi active redundancy (Hybrid redundancy): Combines the advantages of static and dynamic redundancy where one part is redundant in an active way and the other part is redundant in a passive way. Hybrid redundancy is based on error detection and recovery techniques that allow the system to be reconfigured in the event of a fault. It relies on the fault masking technique to prevent the production of incorrect results. For example, to tolerate a permanent fault of a processor or a communication medium, active redundancy is used for the software components of the algorithm and passive redundancy for the communications [Kalla 2004].

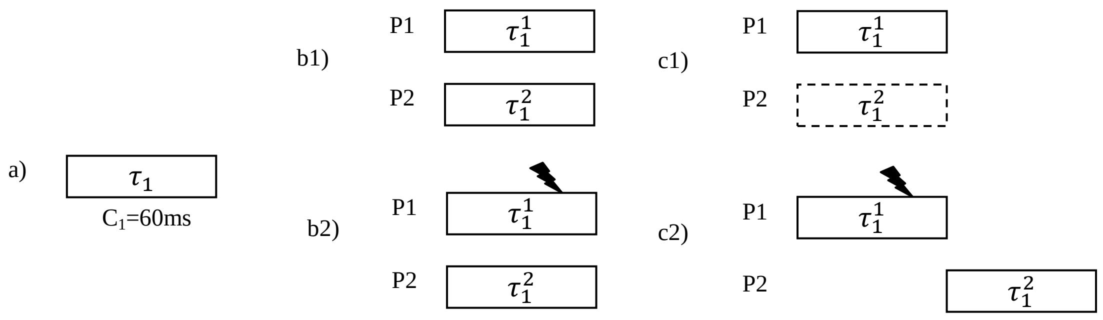


Figure 3-6 Active redundancy b) and Passive redundancy c)

In Figure 3-6 we illustrate active redundancy and passive redundancy. We consider task τ_1 with the worst-case execution time of 60 ms Figure 3-6a. The task $\tau_1$will be replicated on two processors P1 and P2. Which is enough to tolerate a single fault. In the case of active replication, illustrated in Figure 3-6b, replicas τ_1^1 and τ_1^2 are executed in parallel, which, in this case, improves system performance. However, active redundancy occupies more resources compared to passive redundancy because τ_1^1 and τ_1^2 have to run even if there is no fault, as shown in Figure 3-6b1. In the case of primary-backup, illustrated in Figure 3-6c, the "backup" replica τ_1^2 is activated only if a fault occurs in τ_1^1. However, if faults occur, primary-backup takes more time to complete compared to active redundancy as shown in Figure 3-6c2, compared to Figure 3-6b2.

Comparison between the three approaches of redundancy:

The Table 3-1 compares the three approaches of redundancy: active, passive and hybrid according to the criteria of response time, error handling and recovery after failure.

Table 3-1. Comparison between three approaches of redundancy

Comparison Criterion	Active redundancy	Passive redundancy	Hybrid redundancy
Response time	Rapid response time	Better response time in the absence of faults. The failure of the primary replica can significantly increase response time.	Response time depends on the level of active replication versus passive replication.
Error handling	With compensation	With recovery	With compensation and recovery
Recovery after failure	Immediate	Not immediate	Not immediate

3.4. Conclusion

The aim of this chapter was been to introduce the concept of dependability in real-time systems. In accomplishing this goal, we have introduced the main notions about fault tolerance in these systems. We have presented different classes of faults, errors, and failures.

We have also introduced fault tolerance techniques and more specifically the different classes of redundancy deployed in the fault tolerant systems.

In the next chapter, we provide an overview of related work on fault tolerance approaches and real-time scheduling techniques with the joint consideration of energy efficiency and fault tolerance.

CHAPTER 4

Literature Review

4.1. Introduction

As explained in Chapter 2, a real-time system is responsible for delivering logically correct computations within the predefined deadlines. The violations of task deadlines in real-time systems can potentially lead to catastrophical consequences [Han 2015]. To guarantee the timing constraints, real-time scheduling that primarily determines the resource allocation and management has been widely adopted as one of the most effective techniques. In general, real-time scheduling determines when, where, and how to execute a set of real-time tasks such that all deadlines can be met and other design metrics, e.g. power consumption and reliability can be optimized. As presented also in chapter 2, Real-time scheduling can be classified into various categories from different perspectives (architecture, severity of task deadlines, and scheduling mechanisms).

This chapter presents at first a state of the art on works realized on real-time scheduling. Then, we discuss the work done on fault tolerant scheduling with consideration of transient and permanent faults. Finally, we present some related researches that have study the interaction between energy and fault tolerance techniques in real-time systems.

4.2. Real-time uniprocessor scheduling

In this section, we present the uniprocessor scheduling algorithms and their corresponding schedulability tests.

4.2.1. Rate Monotic RM

Rate Monotonic scheduling algorithm is a fixed task priority algorithm. It assigns a priority according to the task's period: the shorter period is the higher priority.

Liu and Layland (1973) have given the necessary and sufficient condition of schedulability: a real-time system composed of n tasks can be scheduled by Rate Monotic if the following condition is satisfied:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1) \tag{4.1}$$

Example 1: Let Γ a set of 2 periodic tasks with implicit deadlines (see Table 4-1).

Table 4-1. Example of Rate Monotic: Task set details

Γ	C_i	T_i
τ_1	3	10
τ_2	2	15

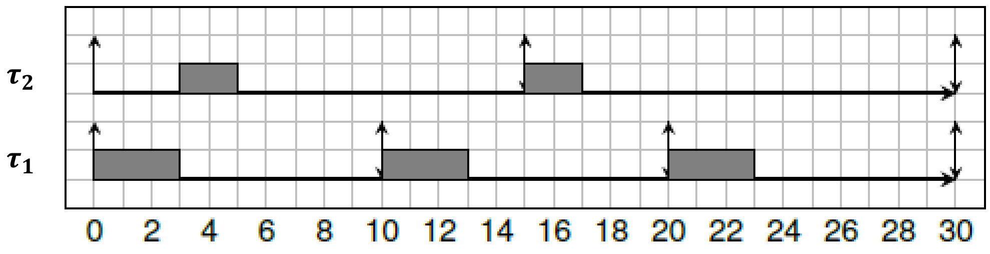


Figure 4-1 Example of scheduling with Rate Monotic

The total task set utilization is:

$$U = \frac{3}{10} + \frac{2}{15} = 0.43$$

The utilization is less than 0.82, thus the task set is schedulable under RM (Figure 4-1).

4.2.2. Deadline Monotic DM

Deadline Monotonic is a scheduling algorithm from a fixed task priority scheduling class. With Deadline Monotonic, task priorities are assigned according to their deadlines; the highest priority is assigned to the task with the shortest deadline. In contrary of Rate Monotonic, DM considers tasks with constrained deadlines.

The sufficient condition of schedulability is inspired by the sufficient condition of Liu and Layland as follows: a real-time system composed of n tasks can be scheduled by DM if the following condition is satisfied:

$U = \sum_{i=1}^{n} \frac{C_i}{D_i} \leq n(2^{\frac{1}{n}} - 1)$ (4.2)

Example 2: Let Γ a set of 3 periodic tasks with timing characteristics given in Table 4-2.

Table 4-2. Example of Deadline Monotic: Task set details

Γ	**C$_i$**	**T$_i$**	**D$_i$**
τ_1	5	10	9
τ_2	4	15	7
τ_3	6	30	1

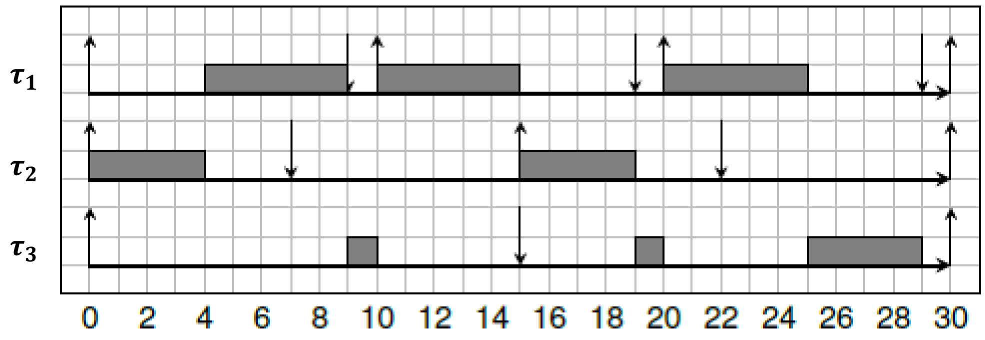


Figure 4-2 Example of scheduling with DM

We can see in Figure 4-2 that the worst case response time of τ_3 is greater than it's deadline. Thus τ_3 is not schedulable.

4.2.3. Earliest Deadline First EDF

Earliest Deadline First (EDF) is a dynamic priority scheduling algorithm. It selects a task according to its deadline such that a task with earliest deadline has higher priority than others. It means that the priority of a task is inversely proportional to its absolute deadline. Since absolute deadline of a task depends on the current instant of time so every instant is a scheduling event in EDF as the deadline of the task changes with time. EDF is an optimal scheduling algorithm on preemptive uniprocessor.

For implicit deadline tasks, EDF has a utilization bound of 100%. Thus, an exact schedulability test can be driven based on utilization as following if all deadlines are implicit:

$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$ (4.3)

For constrained deadline tasks, a lot of sufficient tests were driven. The response time based analysis is hard to perform since that the critical moment does not arrive at time 0 [Zahaf 2016].

Example 3:

Figure 4-3 shows the simulation of scheduling the task set of example 2. As we can see the task set is not schedulable under EDF, and that at moments t = 19 and t = 22 deadlines are missed.

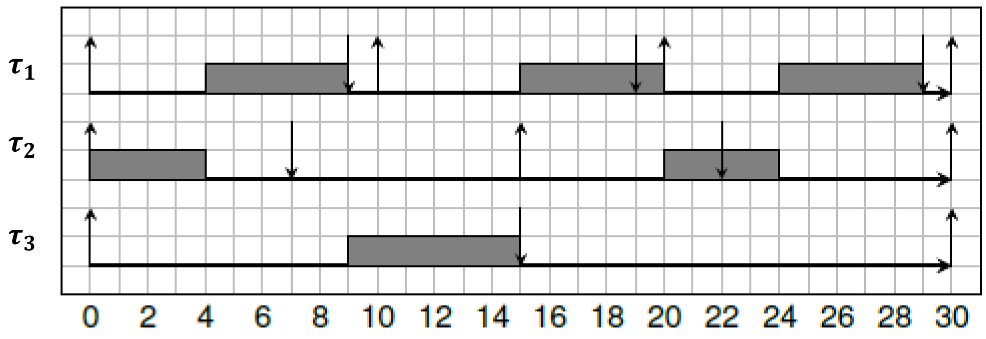


Figure 4-3 Example of scheduling with EDF

4.3. Real-time multiprocessor scheduling

The problem of real-time scheduling applications on multiprocessor systems is more complicated and challenging than real-time scheduling on uniprocessor systems. It is because there are more decisions to be taken in the case of multiprocessor scheduling and more issues to be considered. We can summarize these issues as follows:

i. Allocating tasks to processors;
ii. Assigning tasks priorities to be used by the scheduler;
iii. Tasks migration between processors.

In the next section, we present the approaches solving these problems and the related work carried out with each approach.

4.3.1. Related work on real-time multiprocessor scheduling

A lot of work concerning real-time multiprocessor scheduling has been proposed. Mainly, two approaches had been followed:

4.3.1.1. Partitioned scheduling:

Partitioning tasks among processors is transforming the problem of allocation to *m* processors to *m* uniprocessor problems. Most partitioning algorithms pass through three steps [Zahaf 2016]:

1. Sort tasks in order of some criteria (period, deadline, density, utilization, etc.);
2. Assign tasks in the order of step 1 to a processor where it will always meet all deadlines when assigned to that processor, and it does not cause another previously assigned task to miss a deadline. If a task verifies these conditions, the task fits on this processor. This step is performed using schedulability tests;
3. After each task has been assigned to a processor, we use the well-known uniprocessor scheduling algorithm on each processor to schedule the processor's respective tasks.

The problem of allocation tasks has been shown to be NP-hard [Leung and Whitehead 1982] and requires heuristics. Different allocation schemes such as traditional Bin-packing heuristics: First Fit (FF), Best Fit (BF), and Worst Fit (WF) have been evaluated in [Lieu 2000], and how the ordering of tasks can affect the task allocation results is investigated in [Oh and Son 1995]. Later, the characteristics of real-time tasks were exploited to develop more effective task partitioning schemes in [Fan et al. 2014, Fan and Quan 2011]. For example, as shown in [Fan and Quan 2011], by grouping harmonic tasks into the same core, system schedulability can be greatly enhanced. On the other hand, partitioning of dynamic-priority periodic tasks on multiprocessor is explored in [Baruah 2007a, Baruah 2013]. Simple heuristics such as BF, FF, and WF have been evaluated, and extensions to these approaches are proposed. As shown in [Baruah 2013], ordering tasks in decreasing utilization can significantly improve system schedulability.

4.3.1.2. Global scheduling:

In contrast to partitioned scheduling, the global scheduling allows task migration. All processors have the same ready-queue and the *m* highest priority tasks are run at the same time on *m* processors. Global scheduling has several advantages compared to partitioned scheduling because it allows fewer context switches/preemption. This is because the scheduler will only preempt a task when there are no processors idle. When a task executes for less than its worst-case execution time, the slack time of the task can be utilized by all other tasks, not just those on the same processor. Figure 4-4 shows partitioned scheduling in the face of Global Scheduling

A *semi-partitioned scheduling* is a combination of the two previous approaches, in which most tasks are fixed to specific processors to reduce number of migration, while a few tasks migrate across processors to improve processor utilization [Qamhieh 2015].

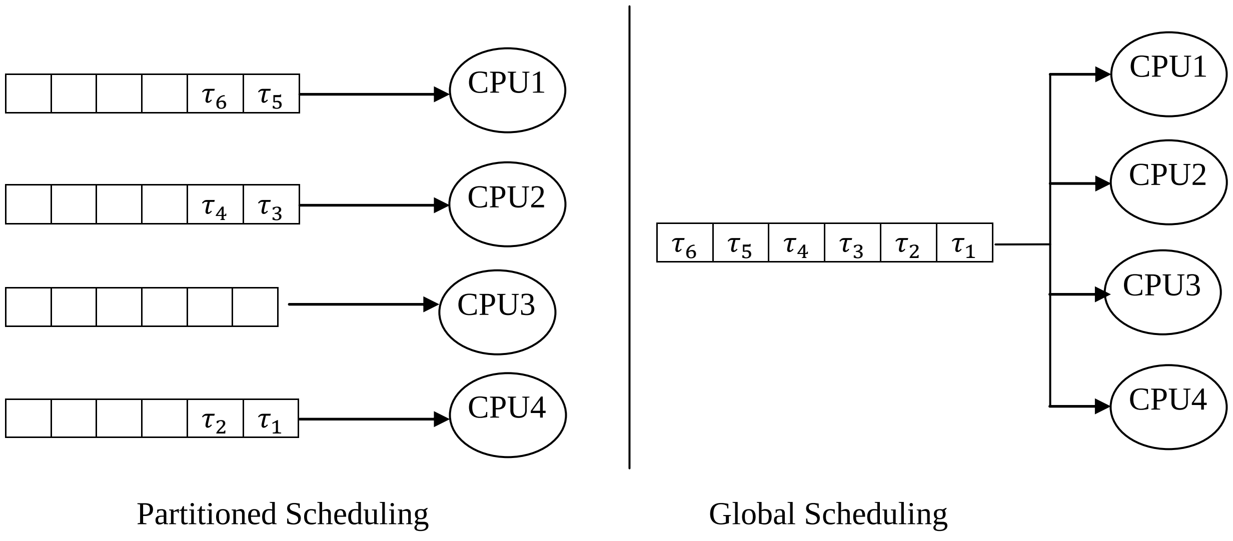


Figure 4-4 Partitioned Scheduling vs Global Scheduling [Zahaf 2016]

There is a great number of works on global scheduling. PFair (Proportionate Fairness) is an optimal multiprocessor scheduling algorithm which was introduced by Baruah et al. (1996) for periodic and sporadic global scheduling of implicit deadline task sets. PFair utilizes the full capacity of processors by scheduling successfully any task set whose utilization is not greater than the number of processors. A new schedulability test for global scheduling of fixed-priority tasks with arbitrary deadlines on identical multi-core processors has been proposed in [Baruah et al. 2007b].

Finally, the effects of semi-partitioning scheduling on improving system schedulability are examined in [Fan and Quan 2012]. By allowing a limited number of tasks to be split and assigned to different cores, the utilization bound of the system is increased, and hence the system schedulability can be improved.

4.4. Related work on fault tolerant real-time scheduling

In the literature, we can identify several advanced fault tolerant techniques that are very effective in guaranteeing the schedulability of hard real-time systems in the presence of faults.

Processor faults can be largely classified as transient or permanent. A transient fault happens for a short period time and then disappears without caused physical damage to the processor. On the contrary, a permanent fault disables a processor permanently. Many different replication methods were explored to make tradeoffs between fault tolerance and system resource usage, e.g. the number of processors required for a feasible schedule.

Chen et al. (2007) introduced several replication schemes to tolerate a fixed number of faults for periodic real-time tasks on homogeneous multiprocessor systems. Two problems are studied in the paper. One is to minimize the maximum utilization in a system with a specified number of processors. The other is to minimize the number of processors required for deriving a feasible schedule. In that work, only active replication is considered. Later on, two heuristics referred to R-BFD (Reliable Best-Fit Decreasing) and R-BATCH (Reliable Bin-packing Algorithm for Tasks with Cold standby and Hot standby) were introduced in [Kim et al. 2010]. They consider a fail-stop failure model. The Cold standby and Hot standby are in fact the active replication and passive replication, respectively. Zhang et al. (2014) proposed two fault-tolerant scheduling methods on multiprocessor systems via both active and passive backup copies to tolerate at most one processor permanent failure. The first method uses the integer linear programming method to obtain the optimal results. The second one is a heuristic algorithm which can achieve close to optimal results within polynomial running time.

Significant research has been presented to deal with transient faults. These faults occur more frequently than permanent faults in modern computing systems. While transient faults can occur in both uniprocessor and multiprocessor platforms, a part of current researches are focused on uniprocessor platforms [Han et al. 2003, Aydin 2007, Zhang et al. 2003] and the other part on multiprocessor platforms. Han et al. (2003) proposed a combined primary and backup scheme to tolerate at least one transient fault. The backup is assumed to be fault-free and of lower quality yield. The timing constraint is guaranteed by scheduling the backups with higher priority at the cost of quality loss. In [Zhang et al. 2003], the schedulability analysis for fixed-priority tasks with checkpoints was investigated, and an effective checkpointing scheme was proposed. Subsequently, in [Aydin 2007], a dynamic programming approach was proposed to evaluate the feasibility of aperiodic task sets under preemptive Earliest Deadline First (EDF) scheduling given a fault-tolerance constraint, i.e maximum *K* fault. For multiprocessor systems, Pop et al. (2009) proposed a more comprehensive approach to the synthesis of fault tolerant schedule for applications on heterogeneous distributed systems. They used the combination of checkpointing and active replication to deal with the fault-tolerance problem. A meta-heuristic (Tabu search) was constructed to decide the fault-tolerance policy, the placement of checkpoints, and the mapping of tasks to processors with the aim of minimizing the overall schedule length. Similar analysis was conducted in [Huang et al. 2011] where hardware replication and software re-execution were employed to take both permanent and transient faults. All these works do not consider energy consumption as a design constraint, which makes them insufficient for energy-constrained real-time systems. Table 4-3 provides the summary of related work on fault tolerant scheduling.

Table 4-3. Summary of fault-tolerant scheduling

Types of faults	Fault tolerance techniques	Platform	References
Permanent and transient	Active replication and re-execution	Homogeneous multiprocessor	Huang et al. 2011
	Passive replication		Samal et al 2014
Permanent faults	Active replication	Homogeneous multiprocessor	Chen et al. (2007), Hashimoto et al. 2002
		Heterogeneous multiprocessor	Girault et al. 2004, Bachir 2019
	Passive replication	Homogeneous multiprocessor	Zarinzad et al. 2008
		Heterogeneous multiprocessor	Qin and Jiang 2006, Oh and Son 1997
	Active and passive	Homogeneous	Kim et al. 2010, Zhang et al.

	replication	multiprocessor	2014
Transient faults	Checkpointing	uniprocessor	Zhang et al. 2003, Aydin 2007
		multiprocessor	Han et al. 2015, Izosimov et al. 2012, Wei et al. 2012
	Active replication	Heterogeneous multiprocessor	Girault and Kalla 2009
	Active replication and checkpointing	Heterogeneous multiprocessor	Pop et al. 2009
		Homgeneous multiprocessor	Motaghi and Zarandi 2014
	Active and passive replication	uniprocessor	Han et al. 2003

4.5. Related work on fault-tolerant scheduling with energy minimization

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 emerged as the most effective technique for energy reduction [Aydin et al. 2004].

DVFS scheduling reduces the supply voltage and frequency when possible, which result on conserving energy consumption. However, one consequence of applying DVFS is the extended circuit delay which may undermine the schedulablity of real-time system [Han 2015]. As a result, a great number of techniques studying the problem of minimizing the energy consumption with respect of timing constraints are proposed in the literature [Aydin et al. 2004, Quan and Niu 2004, Zahaf 2016, Digalwar 2016, Hu et al. 2016].

Also, extensive researches have analyzed the interplay of energy trade-offs and fault-tolerance techniques [Ejlali et al. 2006, Melhem et al. 2004, Zhang and Chackrabarty 2006]. Redundancy-based fault-tolerance techniques (such as re-execution checkpointing and replication) and DVFS-based low-power techniques compete for the available slack. The interplay of power management and fault recovery has been addressed in [Melhem et al. 2004], where checkpointing policies were evaluated with respect to energy. In [Ejlali et al. 2006], time redundancy was used in conjunction with information redundancy, which does not compete with DVFS for slack, to tolerate transient faults. In [Zhang and chackrabaty 2006], fault tolerance and dynamic power management were studied, and rollback recovery with checkpointing was used to tolerate multiple transient faults in distributed systems. Addressing energy and reliability simultaneously is especially challenging because lowering the voltage to reduce energy consumption has been shown to increase the number of transient faults exponentially [Zhu et al. 2004]. The main reason for such an increase is that, with lower voltages, even very low energy particles are likely to create a critical charge that leads to a transient fault. However, this aspect has received very limited attention. Zhu and Aydin (2009) have proposed a reliability-aware DVFS heuristic for uniprocessor systems, and a single-task checkpointing scheme was evaluated in [Zhu et al. 2004]. In [Pop et al. 2007], we consider the energy versus reliability trade-offs in the context of distributed time-triggered systems, where tasks and messages are scheduled based on a static-cyclic scheduling policy, and transient faults are tolerated using task re-execution. Table 4-4 provides the summary of related work on fault-tolerant scheduling with energy minimization. It can be observed from the table that energy aware fault tolerance is sufficiently addressed for independent tasks. For dependant tasks energy aware fault tolerance is not thoroughly addressed. Also the combination of software replication and time redundancy methods with DVFS technique is not addressed in the literature.

Table 4-4. Summary of fault-tolerant scheduling with energy minimization

Task model	**Fault-tolerance techniques**	**techniques to minimize energy**	**References**
Independent tasks	Checkpointing	DVFS	Melhem et al. 2004, Zhang and Chakrabaty 2006, Wei et al. 2012, Zhu and Aydin 2009, Salehi et al. 2016, Zhu et al. 2004

	Re-execution	DVFS	Djosic and Jevtic 2013, Pop et al. 2007
	Time and information redundancy	DVS and ARB	Ejlali et al. 2006
Dependant tasks	Stand by sparing and re-execution	DVFS and DPM	Tavana et al. 2014
	Active replication	DVFS	Assayad et al. 2012
	Checkpointing	DVFS	Li et al. 2015

4.6. Conclusion

In this chapter, we have reviewed some closely related work in the literature. First, existing researches on real-time scheduling for various task and system models are discussed. Then, we have presented the related research in fault tolerant real-time scheduling with consideration of permanent and transient faults. Finally, we have presented the interplay of energy and fault-tolerance techniques on employing DVFS in real-time scheduling in detail. Based on the above discussions, we can see that fault-tolerant scheduling under various constraints still poses a grand challenge for researchers.

In the next chapter, we describe the proposed fault tolerance approach which combines two strategies of the basic families of redundancy.

CHAPTER 5

A Fault-Tolerant Scheduling Algorithm Based on Checkpointing and Redundancy for Distributed Real-Time Systems

5.1. Introduction

Fault tolerance techniques have been proposed for real-time systems to satisfy their constraints even in the presence of faults. Transient faults are the most common, and their number is continuously increasing due to the high complexity, smaller transistor sizes, higher operational frequency, and lower voltage levels [Djosic and Jevtic 2013; Han et al. 2013; Paul et al. 2009; Wei et al. 2012]. These faults happen for a short time and then disappear without causing permanent damage. Transient faults have become the main concern in the design of modern embedded real-time systems.

In this chapter, we present a novel fault tolerance approach based on scheduling heuristic to tolerate a fixed number of transient faults. Our approach combines active replication, which provides space-redundancy, and checkpointing with rollback recovery, which provides time-based redundancy. In addition, we propose a new fault-tolerant scheduling heuristic which generates, from a given hard real-time application and a given multiprocessor distributed architecture, a fault tolerant distributed static schedule which tolerates *K* transient faults.

The rest of this chapter is organized as follows. A brief overview of related work is provided in section 2. Section 3 describes our application model, hardware model, and fault model. Section 4 explains our fault tolerance approach through examples. In section 5, we present our proposed static fault-tolerant scheduling heuristic. Simulation results are discussed in section 6, and finally, section 7 concludes the chapter.

5.2. Literature review

Extensive research has been presented to investigate the software-based fault tolerance techniques against transient faults. In the software replication technique [Girault et al. 2004; Assayad et al. 2012; Samal et al. 2014; Meroufel and Belalem 2014] multiple replicas (active or passive) of each task are executed on different processors.

Bachir (2019) proposed three fault-tolerant scheduling heuristics to tolerate permanent faults of a single processor. The first heuristic AAA-FAULT^{DT} is used to minimize the scheduling length using passive replication. The second FT-TDEP is based on hybrid redundancy and the third AAA-Fault^{IDT} based on active replication. The primary-backup approach (passive replication) is used as a fault-tolerant scheduling technique in [Samal et al. 2014] to guarantee real time tasks constraints in the presence of permanent or transient faults. The authors proposed fault-tolerant scheduling for independent tasks using a hybrid genetic algorithm.

The replication technique is effective to tolerate spatial multiple faults (permanent or transient) and it is more preferable for safety-critical systems [Ejlali et al. 2012]. However, scheduling multiple replicas of each task on different processors may not be affordable due to cost constraints [Ropars et al. 2015].

Checkpointing with rollback recovery [Han et al. 2015, Izosimov et al. 2012, Wei et al. 2012, Zhang and Chakrabarty 2006, Kumar et al. 2015) and re-execution [Izosimov et al. 2008, Gui and Luo 2013] are classified by Motaghi and Zarandi (2014) as time-based redundancy methods. These methods try to deal with transient faults by serial executions in the same processor of faulty task. Izosimov et al. (2008) proposed a quasi static scheduling of fault tolerant embedded systems composed of hard and soft processes. In which re-execution is employed to recover from multiple faults. Han et al. (2015) presented a task allocation scheme for minimizing energy consumption while ensuring the fault tolerance requirement of the system. They develop an efficient method to determine the checkpointing scheme to tolerate at least one transient faults on a single processor. These methods do not impose any hardware cost overhead and are not effective to tolerate transient faults whose durations are very long. Moreover, serial execution may cause the non respect of time constraints.

The combination of software replication and time-based redundancy techniques to tolerate multiple transient faults with low overhead in terms of hardware cost, energy consumption and total execution time have been studied in few works related to our research [Pop et al. 2009, Motaghi and Zarandi 2014].

Pop et al. (2009) have proposed a fault-tolerance policy assignment strategy to decide which fault tolerance technique, for instance checkpointing, active replication or their combination, is the best suited for a particular process in the application. A dynamic fault-tolerant scheduling DFTS has been proposed in [Motaghi and Zarandi 2014]. This algorithm uses task utilization to dynamically select the type of fault recovery method in order to tolerate multiple transient faults.

This chapter attempts to solve the following problem which is an NP-hard problem. Given an homogeneous architecture, how to schedule an application of hard dependant tasks on the architecture under multiple transient faults which may occur spatially or temporally.

The main contributions of this chapter are summarized as follows:

- Tolerating multiple transient fault occurrences with respect of application time constraints.
- Combine two different policies: checkpointing and active replication to propose an efficient fault-tolerant scheduling approach that explores hardware resources and timing constraint.
- Replicas collaboration is introduced to tolerate spatially or temporally faults and satisfy critical task constraint. To the best of our knowledge, this is the first work introducing the idea of collaboration between replicas in active replication technique with checkpointing.

The proposed approach classifies the real-time tasks into critical and noncritical ones, according to the utilization of the task. For the noncritical task, we adopt checkpointing with rollback technique to tolerate multiple transient faults. The main reason for this choice that many studies showed the efficiency of checkpointing technique to deal with these faults. Whereas for the critical task, we adopt active replication as it is the fault tolerant method that explores hardware resources to meet timing constraints and provide high reliability even when deadlines are tight. With a view to tolerate spatially or temporally faults and satisfy critical task constraint, we have also introduced replicas collaboration. This collaboration can be seen as communication in case of fault occurrence. If one replica is faulty, the second replica has to send it the correct state to complete the execution before its deadline.

5.3. System model

5.3.1. Application model

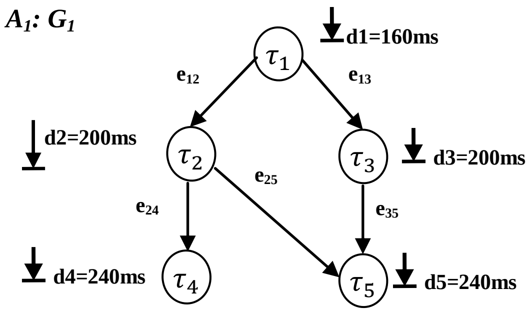
The real-time application considered in this chapter consists of n hard dependant tasks, denoted as: $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$. The dependence $\tau_i \rightarrow \tau_j$ means that τ_i execution precedes τ_j execution. So we say that τ_j is a successor of τ_i and symmetrically that τ_i is a predecessor of τ_j . The timing characteristics of the task τ_i are defined as a tuple (C_i, D_i), where C_i is the worst case execution time of the task in a fault-free condition and D_i is the deadline of the task. The utilization of task τ_i is defined as:

$$U_i = \frac{C_i}{D_i} \qquad 0 \leq U_i \leq 1$$

The system utilization is therefore calculated as:

$$U = \sum_{i=1}^{n} U_i$$

We model an application ***A*** as a directed acyclic graph **DAG**. Each node represents one task. An edge e_{ij} indicates data-dependency between two tasks τ_i and τ_j.



Task	C_i	D_i
τ_1	30	160
τ_2	40	200
τ_3	60	200
τ_4	40	240
τ_5	40	240

WCCT	
e_{12}	10
e_{13}	5
e_{24}	10
e_{25}	10
e_{35}	5

Figure 5 -1 Hard real-time application example

An example of an application A_1 composed of 5 dependant tasks is represented as a **DAG** G_1 shown in Figure 5-1. The two tables give the timing characteristics of a task set and data dependencies.

5.3.2. Hardware model

The architecture is considered as a set of M homogeneous processors denoted as: $P = \{P_1, P_2, \ldots, P_M\}$.

A processor is composed of a computing unit, to execute tasks, and one or more communication unit to send/receive data to/from communication links. A point-to-point communication link is composed of a sequential memory that allows it to transmit data from one processor to another.

Each processor is connected with the others through communication links. So our architecture is homogeneous and fully connected. We can model the architecture by a graph, where each node is a processor and each edge is a communication link.

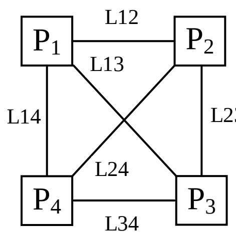


Figure 5-2 Hardware example

An example of an architecture graph with four processors P_1 to P_4, and six communication links is shown in Figure 5-2.

5.3.3. Fault model

In this chapter, we focus on transient faults which are the most common faults in today's distributed real-time systems. In our model, we assume at most k faults to be tolerated on each task. Given a fault arrival rate λ and a task execution interval t, the mean number of faults that arrive during the interval is λt [Zhang and Chakrabarty 2006]. In order to target a system with reasonable real-time performance with fault tolerance, Zhang and Chakrabarty (2006) indicate that the value of k should be taken as multiple of λt, e.g. $2\lambda t \leq k \leq 3\lambda t$. A transient fault affects only the task running on a specific processor as transient faults have short duration.

The error detection and fault-tolerance mechanisms are part of the software architecture. The error detection overhead is considered as part of the task execution time. The software architecture, including the real-time kernel, error detection and fault tolerance mechanisms are themselves fault-tolerant.

5.4. The proposed fault-tolerant approach

We propose a novel fault-tolerant approach which combines software replication and time redundancy methods for tolerating k transient faults. We use these two techniques in order to meet time and cost constraints of hard real-time applications even in the presence of transient faults.

As time-based redundancy we use checkpointing with rollbacks for non critical tasks. Once a fault is detected, the task being affected rolls back to the last saved checkpoint and re-execute the faulty segment [Kumar and Rachit 2011]. Inserting one checkpoint to task τ_i refers to the operation of saving its current state in memory.

As software replication, we use active replication for critical tasks to meet their deadlines even in the presence of faults.

Similar to the related work in [Motaghi and Zarandi 2014], we compute the task utilization U_i of each task τ_i in the ready list of the scheduler to decide if τ_i critical or noncritical.

$$\tau_i = \begin{cases} critical & U_i \geq \theta \\ non\ critical & U_i < \theta \end{cases} \text{ Where } 0 \leq \theta \leq 1$$

The criticality threshold θ will be computed for each task in the ready list when there is an idle processor, therefore θ vary from one task to another. We will present in section 5 the calculation of the criticality θ.

5.4.1. Checkpointing with rollback recovery

The time overhead for re-execution can be reduced with more complex techniques such as rollback recovery with checkpointing [Zhang and Chakrabarty 2006, Eles et al. 2008]. By using this technique, once a fault is detected during the execution of the task τ_i, it needs to rollback to the latest checkpoint [Salehi et al. 2016].

We consider the following assumptions related to task execution and fault arrivals:

- The checkpointing intervals are equal for the same task.
- Faults are detected as soon as they occur.
- No faults occur during checkpointing and rollback recovery.

The fault-free execution time of task τ_i using checkpointing is a function of the number of checkpoints m_i and is formulated in Equation (5.1a)

$$C_i(m_i) = C_i + m_i O_i \tag{5.1a}$$

Where O_i is the time overhead required for saving one checkpoint and is proportional to the worst case execution time C_i of each task.

The recovery time of τ_i with m_i checkpoints under a single failure includes two parts, the time to rollback to the latest checkpoint and the time to re-execute the faulty segment. We denote it $R_i(m_i)$ and formulate it in Equation (5.1b)

$$R_i(m_i) = r_i + \frac{C_i}{m_i} \tag{5.1b}$$

Where r_i is the time overhead to rollback to the latest checkpoint and is proportional to the worst case execution time of each task.

In general, in the presence of k faults, the worst case response time $WCRT_i$ of task τ_i using checkpointing with rollback recovery can be obtained by the Equation (5.1)

$$WCRT_i = C_i(m_i) + k * R_i(m_i) \tag{5.1}$$

As related work, Paul et al. (2009) showed that the optimal number of checkpoints to minimize the worst case response time $WCRT_i$ considering k faults can be calculated as:

$$m_i = \begin{cases} m_i^- = \left\lfloor \sqrt{\dfrac{k * C_i}{O_i}} \right\rfloor, & if\ C_i \leq\ m_i^-(m_i^- + 1)\dfrac{O_i}{k} \\ m_i^+ = \left\lceil \sqrt{\dfrac{k * C_i}{O_i}} \right\rceil, & if\ C_i >\ m_i^-(m_i^- + 1)\dfrac{O_i}{k} \end{cases}$$

Where O_i is the time overhead for saving one checkpoint and C_i is the worst case execution time of task τ_i. As the number of checkpoints is an integer, thus we use m_i^- (the floor) or m_i^+(the ceiling) as a value. If $C_i \leq\ m_i^-(m_i^- + 1)\frac{O_i}{k}$, we use the floor value. Otherwise, the ceiling value is used.

For the sake of easy presentation, m_i is simply denoted by

$$m_i = \left\| \sqrt{\frac{k * C_i}{O_i}} \right\| \quad (5.2)$$

An example of checkpointing with rollback recovery is presented in Figure 5-3.

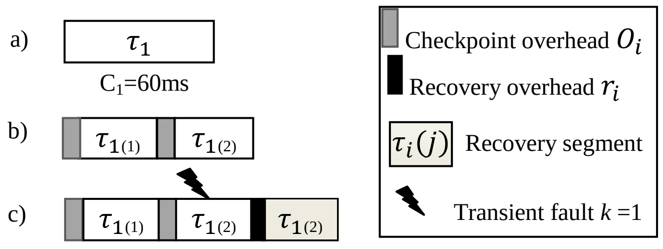


Figure 5-3 Checkpointing with rollback recovery

We consider task τ_1 with worst execution time $C_1 = 60\ ms$ in Figure 5-3a. In Figure 5-3b two checkpoints are inserted at equal intervals. Thus, task τ_1 is composed of two equal execution segments $\tau_{1_{(1)}}$ and $\tau_{1_{(2)}}$. In Figure 5-3c, a fault affects the second execution segment $\tau_{1(2)}$. This faulty segment is executed again starting from the second checkpoint. The recovery segment is represented by a light gray rectangle.

5.4.2. Active replication with checkpointing

The disadvantage of checkpointing with rollback recovery technique is that cannot explore the spare capacity of available processors in the architecture to reduce the schedule length. However software replication techniques (active and passive replication) have the ability to execute task replicas in parallel on different computation nodes. With active replication, all the task replicas are executed independent of fault occurrences [Girault et al. 2009]. However, with passive replication, replicas are executed only if faults occur [Han 2015, Zhang et al. 2014].

In our work, we are interested in active replication. If there is enough time to rollback the task τ_i to the last saved checkpointing in the presence of faults we use active replication to guarantee the respect of deadline. The task τ_i is replicated on two collaborative replicas; τ_i^1 and τ_i^2, both of which to be executed on different processors in the same time. We introduce collaboration between replicas to tolerate k faults and respect τ_i deadline. We consider the following assumptions:

- All checkpoints are assumed to be fault-free, i.e., no faults can occur during checkpointing saving.
- Each task's primary copy and backup copy must not be assigned to the same processor.

- Each task's primary copy and backup copy cannot be faulty at the same time.

Our goal is to tolerate k faults with respect of task τ_i deadline. To achieve this goal, we use active replication technique where each critical task is replicated on several replicas. But it is possible that these replicas can be faulty due to multiple fault occurrences, therefore our goal will be missed and this technique will be infeasible. This is the case in the work presented in [Motaghi and Zarandi 2014].

As solution, we introduce collaboration between replicas of the critical task to tolerate each coming fault in the primary or the backup replicas (τ_i^1,τ_i^2) to achieve the feasibility of our approach. For computation purpose, we add an extra virtual processor to the architecture, noted *P#*.

Once the active replication approach is decided for a task τ_i, first it has to be scheduled on virtual processor *P#* (τ_{iv}) at start time ST_i as illustrate in Figure 5-4.

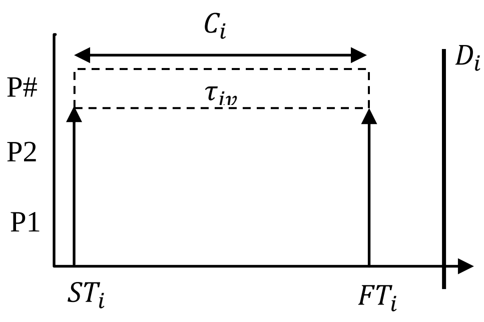


Figure 5-4 Scheduling of τ_{iv} on virtual processor P#

Then we place in this task τ_{iv} the appropriate m_i checkpoints (with equal checkpoint intervals $\frac{C_i}{m_i}$) obtained after calculation of the optimal number of checkpoints with Equation (5.2).

After that τ_i is replicated on two replicas τ_i^1 and τ_i^2 which must be scheduled on two different processors and checkpointed alternatively by projection of the checkpoints of the initial task as presented in Figure 5-5.

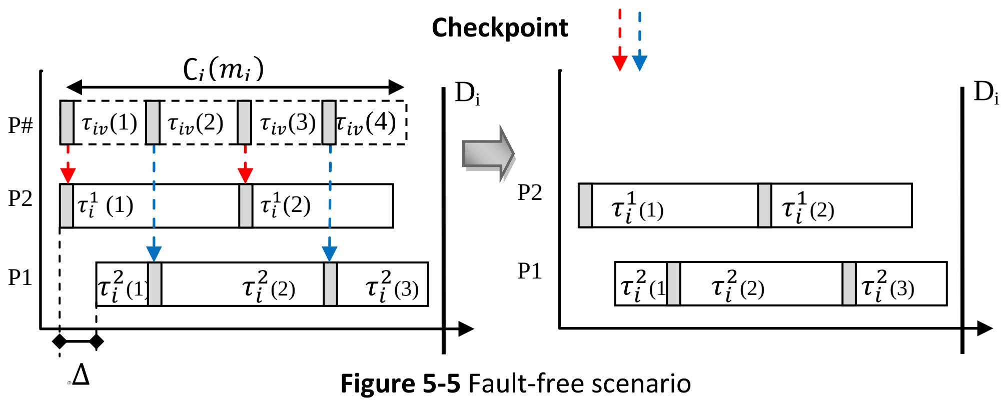


Figure 5-5 Fault-free scenario

The alternative checkpointing idea of the two replicas τ_i^1 and τ_i^2 is to ensure the collaboration between replicas and to minimize the number of checkpoints of the original task τ_i. In this case we can meet the task deadline even in the presence of faults.

In Figure 5-5, Δ represents the difference between the start time of the two replicas τ_i^1 and τ_i^2 (the start time of each replica depends on the availability of processors). It can be written as:

$$\Delta = ST(\tau_i^1) - ST(\tau_i^2) \tag{5.3}$$

For successful of our alternative checkpointing idea, Δ should be less or equal than the checkpointing interval, so we have:

$$0 \le \Delta \le \frac{C_i}{m_i}$$

With this approach the start time ST_i of a task τ_i can be given by:

$$ST_i = \min_{1\le j\le 2}\left(ST\left(\tau_i^j\right)\right) \tag{5.4}$$

Where $ST\left(\tau_i^j\right)$ is the start time of the replica τ_i^j.

Consequently, the actual finish time FT_i of task τ_i is given by:

$$FT_i = ST_i + WCRT_i \tag{5.5}$$

And respectively, the best finish time $FT_{best}(\tau_i)$ of task τ_i can be written as:

$$FT_{best}(\tau_i) = \min_{1\le j\le 2}\left(FT\left(\tau_i^j\right)\right) \tag{5.6}$$

Where $FT\left(\tau_i^j\right)$ is the finish time of the replica τ_i^j.

In the case of fault occurrence in the execution of one of the replicas (τ_i^1 or τ_i^2), the results produced by the no faulty replica must be sent to the other replica to continue the execution as shown in Figure 5-6.

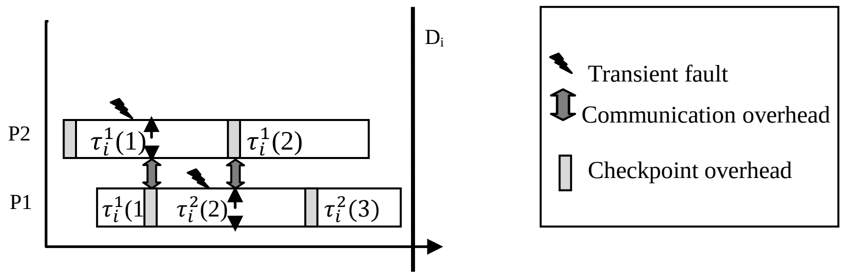


Figure 5-6 Fault occurrence scenario

By using this technique, the worst case response time $WCRT_i$ of the task τ_i in the presence of *k* faults is formulated in Equation (5.7):

$$WCRT_i = \begin{cases} C_i + \left\lfloor \frac{m_i}{2} \right\rfloor \times O_i\,, & if\ k = 0 \\ C_i + \left\lceil \frac{m_i}{2} \right\rceil \times O_i + k \times com(\tau_i^1\,, \tau_i^2)\,, & \text{otherwise or} \\ C_i + \left\lfloor \frac{m_i}{2} \right\rfloor \times O_i + k \times com(\tau_i^1,\, \tau_i^2) & \end{cases} \tag{5.7}$$

Where the term $(C_i + \left\lfloor \frac{m_i}{2} \right\rfloor \times O_i)$ is the maximum execution time of τ_i using the new active replication with checkpointing without any faults, and $Com(\tau_i^1\,,\, \tau_i^2)$ is the cost of communication step between the replicas τ_i^1 and τ_i^2 for single fault, which is multiplied by *k* in the presence of *k* faults.

The best worst case response time $WCRT_i$ of the task τ_i can be written as:

$$WCRT_{best}(\tau_i) = \min_{1 \le j \le 2} WCRT(\tau_i^j) \tag{5.8}$$

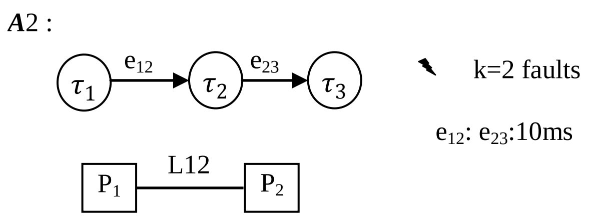
Where $WCRT(\tau_i^j)$ is the worst case response time of the replica τ_i^j and is calculated with Equation (5.7).

5.5. Motivational example

Figure 5-7 represents an application ***A***2 composed of three dependant tasks (τ_1, τ_2 and τ_3) and an architecture composed of two processors *P₁* and *P₂*.

The fault model assumes two faults, thus *k* =2. The checkpoint saving and recovery overhead are considered equal to 5 ms. According to these values, the optimal number of checkpoints is 3, 3 and 4 for τ_1, τ_2 and τ_3 respectively. After computation of the criticality of each task, we get that τ_1 and τ_2 are not critical, so checkpointing with rollback can be applied. But the task τ_3 is critical, consequently τ_3 is replicated on two copies τ_3^1 and τ_3^2.

In the case if the replicas of τ_3 are faulty due to multiple fault occurrence, we introduce collaboration between replicas. As illustrated in Figure 5-8, the faulty replica receives the last correct state from the no faulty replica via communication step between τ_3^1 and τ_3^2 and continue execution to meet its deadline.



Task	***Cᵢ***	***Dᵢ***
τ_1	35	100
τ_2	35	180
τ_3	60	260

Figure 5-7 An application example A2 to be scheduled on P1 and P2 under k=2 faults

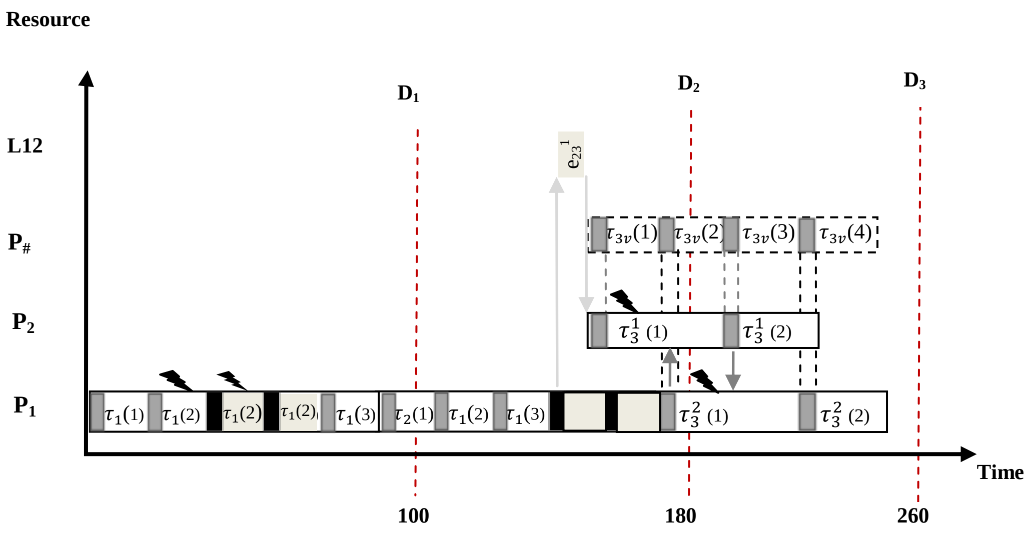


Figure 5-8 Fault tolerant scheduling: combination of checkpointing with rollback for tasks τ_1, τ_2 and active replication with checkpointing for the task τ_3

5.6. The proposed fault tolerant scheduling algorithm

Our fault tolerant scheduling algorithm is presented in Figure 5-9. The algorithm takes as input the application ***A***, the number ***K*** of transient faults that have to be tolerated, the architecture ***P*** and the real time constraints.

```
SFTS Algorithm :
Inputs:
$\Gamma = \{\tau_1, \tau_2, \dots, \tau_n\}$.
P = {P1, P2, …, PM}
K transient faults for each task
Real time constraints
---------------------------------------------------------------
TReady = $\{\tau_i \in \Gamma \mid pred(\tau_i) = \emptyset\}$
While TReady $\neq \emptyset$ do
{  Select $\tau_i$ ∈ TReady having the minimum $D_i$
   Calculate $U_i$
   If $U_i < \theta$ then
   { Apply checkpointing with rollback
     Calculate $WCRT_i$ with Equation (5.1)
     Schedule $\tau_i$ on an idle processor
   }
   Else
   { Apply the proposed active replication with replicas
      collaboration
     Calculate $WCRT_i$ with Equation (5.7)
```

```
        Schedule the replicas of  $\tau_i$ $(\tau_i^1, \tau_i^2)$ on two different
processors
    }
Schedule messages sent by $\tau_i$
$TReady = TReady - \{\tau_i\} \cup \{\tau_j \in succ(\tau_i) | pred(\tau_j) \nexists TReady\}$
} End SFTS
```

Figure 5-9 The proposed static fault tolerant scheduling algorithm SFTS

Our scheduling algorithm is a list scheduling based heuristic, which uses the concept of ready tasks and ready list. By ready task τ_i we mean that all τ_i's predecessors have been scheduled. The heuristic initializes the list *TReady* with tasks without predecessors and is looping while *TReady* isn't empty. At first, the ready task τ_i with minimum deadline is selected for placement in the schedule. Then the scheduler calculates the criticality of task τ_i. If the task τ_i is noncritical, the checkpointing with rollback policy will be applied and the maximum response time of the task τ_i will be calculated with Equation (5.1).

Otherwise, the critical task τ_i will be replicated and the proposed new active replication will be applied. In this case the maximum response time of the task τ_i will be calculated with Equation (5.7). Finally, the task τ_i is removed from the ready list *TReady* and all its successors are added to the list.

The scheduler calculates the criticality θ of the best task τ_i (having the minimum deadline D_i) selected from *TReady*. The start execution time of τ_i is defined as ST_i.

To satisfy the real time constraints of the application to be scheduled even in the presence of faults, for each task τ_i $(ST_i + WCRT_i)$ should be less than D_i. So we have:

$$WCRT_i < D_i - ST_i \tag{5.8}$$

We take WC_i the worst case response time of task τ_i using checkpointing with rollback recovery and we replace it by Equation (5.1) to compute θ.

$$C_i + m_i \times O_i + k \times (r_i + \frac{C_i}{m_i}) < D_i - ST_i \tag{5.9}$$

As described in [Motaghi and Zarandi 2014] we have:

$$O_i = C_i \times \alpha \tag{5.10}$$

$$r_i = C_i \times \mu \tag{5.11}$$

Where $\boldsymbol{\alpha}$ and $\boldsymbol{\mu}$ are constant factors.

By replacing the values of O_i and r_i with Equations (5.10) and (5.11) we have:

$$C_i \times \left(1 + m_i \times \alpha + k \times (\mu + \frac{1}{m_i})\right) < D_i - ST_i \tag{5.12}$$

$$U_i \times \left(1 + m_i \times \alpha + k \times (\mu + \frac{1}{m_i}) \right) < 1 - \frac{ST_i}{D_i} \tag{5.13}$$

Finally, we can get the criticality θ

$$U_i < \frac{1 - \frac{ST_i}{D_i}}{1 + m_i \times \alpha + k \times \mu + \frac{k}{m_i}} = \theta \tag{5.14}$$

5.7. Experimental results

To evaluate the performance of our approach, we have generated a set of 50 different applications with 10, 20, 30, 40, and 50 tasks implemented on architecture consisting of 4 processors. The execution time of each task was randomly assigned in the range of [30 ms, 80 ms] also the deadline was randomly generated. We have varied the number of maximum tolerated faults k considering 2 to 10 faults. Table 5-1 summarizes the configuration parameters used in our experiments.

Table 5-1. Simulation parameters

Parameters	Values
Number of processors	4
Application size (Number of tasks)	(10 , 20 , 30 , 40 , 50)
Execution time (ms)	[30 , 80] – [10 , 100]
Recovery overhead μ	(5%, 10%)
Checkpoint saving α	(5%, 10%)
Number of faults k	(2 , 4 , 6 , 8 , 10)
Fault arrival rate λ	(0.005 , 0.01 , 0.02 , 0.04)

First, we were interested to evaluate the proposed approach with regard to the overheads, in term of schedule length, introduced due to fault tolerance. For this, we have implemented each application without any fault tolerance concerns. This non-fault-tolerant implementation, NFT, has been obtained using EDF algorithm. Then, we have implemented each application on its corresponding architecture using the proposed SFTS algorithm. Let L_{SFTS} and L_{NFT} be the schedule lengths obtained using SFTS and NFT, respectively. The overhead due to introduced fault tolerance is defined as 100 x (L_{SFTS} - L_{NFT})/ L_{NFT}.

Table 5-2 presents the maximum, the minimum and the average time overheads introduced by SFTS compared to NFT in the case of different number of faults. The average fault tolerance overheads introduced by SFTS increase with the number of tolerated faults. In this case the application size and the number of computation nodes were fixed with 40 tasks and four computation nodes where the number k of faults being 2, 4, 6, 8 to 10.

We were also interested to evaluate the proposed algorithm SFTS versus the fault tolerance approach: checkpointing with rollback recovery CH. Figure 5-10 shows that the scheduling length of SFTS is always considerably lower than CH. This is due to the combination of checkpointing and replication which result in decreasing the schedule length of SFTS. We can assume that SFTS performs much better compared to checkpointing.

In Table 5-3, the effect of checkpointing parameters such as checkpoint overhead α and recovery overhead μ on the scheduling time overhead is shown. The effect of increasing the checkpoint overhead α is more than the effect of recovery overhead μ on the scheduling timing overhead of SFTS. However increasing these parameters α and μ increases considerably the scheduling timing overhead of CH. Either way the minimum timing overhead of SFTS is significantly lower than CH timing overhead. We can resume that the scheduling timing overhead of SFTS does not significant change by varying the checkpointing parameters as shown in Figure 5-11.

Table 5-2. Fault tolerance overheads due to SFTS for different number of faults

K	% Maximum	% Minimum	% Average
2	46.90	22.81	32.45
4	62.74	30.18	42.78
6	69.29	42.99	54.85
8	80.12	52.38	64.72
10	91.74	61.44	75.20

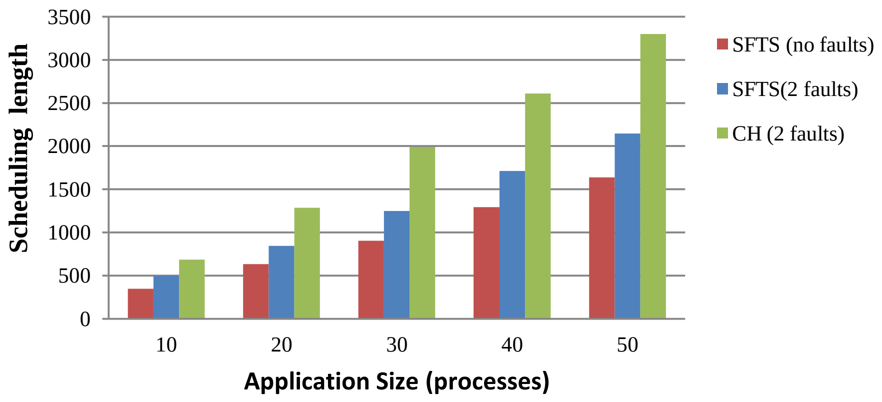


Figure 5-10 Impact on schedule length of application size considering k = 2 faults

Table 5-3. Timing overhead of SFTS compared with checkpoint considering various checkpoint parameters

Checkpoint overhead α	*Recovery overhead μ*	*Fault method*	*Timing overhead%*		
			Max	*Min*	*Average*
0.05	0.05	*SFTS*	39.54	22.39	28.62
		CH	50.00	49.57	49.78
0.05	0.1	*SFTS*	37.43	23.72	28.71
		CH	68.33	54.56	56.09
0.1	0.05	*SFTS*	39.05	20.56	30.96
		CH	68.33	66.78	67.88
0.1	0.1	*SFTS*	39.84	21.07	28.98
		CH	73.34	72.81	73.02

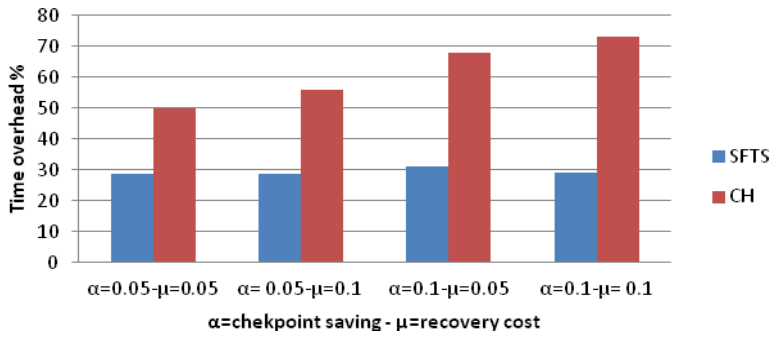


Figure 5-11 The time overhead of STFS compared to checkpointing by varying checkpoint saving α and recovery overhead μ

To evaluate the feasibility analysis of SFTS under the variation of fault arrival rate λ, we conducted another set of experiments. We have generated a set of 50 applications of 10 real-time tasks. The execution time of each task was randomly assigned in the range of [10 ms, 100 ms] also the deadline was randomly generated. According to results presented in Figure 12, the feasibility rate of SFTS decreases from 100% to 32% within varying λ from 0.005 to 0.04. The reason that by increasing λ, the number of expected faults k for each task will increase. This will result in the probability of missing task constraint will increase.

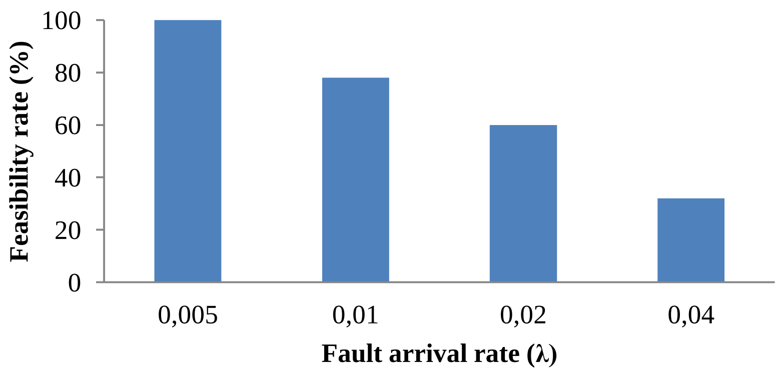


Figure 5-12 Feasibility rate of SFTS by doubling fault arrival rate (λ)

5.8. Conclusion

In this chapter, we have proposed a novel fault tolerance approach for scheduling applications with hard real-time constraints on real-time distributed systems. The approach combines checkpointing with rollback policy and a new strategy of active replication that uses replicas collaboration to guarantee the task deadline even in the case of faults. The task utilization is calculated to decide the type of fault recovery method in order to tolerate K transient faults.

Based on this approach, we have proposed an efficient fault tolerant scheduling algorithm SFTS, which is a list scheduling based heuristic. Our algorithm can be feasible even if the two replicas of the critical task are faulty; in this case, we have introduced the collaboration between replicas when a fault is detected. This idea permits respect of task deadline and ensures the feasibility of our algorithm. Simulation results show the performance and effectiveness of combining checkpointing and redundancy to tolerate transient faults.

CHAPTER 6

An Efficient Fault-Tolerant Scheduling Approach with Energy Minimization for Hard Real-Time Embedded Systems

6.1. Introduction

Energy consumption and fault tolerance have attracted a lot of interest in the design of modern embedded real-time systems. Dynamic power/energy management is an active area of research and many techniques have been proposed to minimize energy consumption under a large diversity of system and task models [Mahmood et al. 2017, Wei et al. 2012]. Dynamic voltage and frequency scaling (DVFS) is an energy saving technology enabled on most current processors. It enables a processor to operate at multiple voltages where each corresponds to a specific frequency. Because the energy consumption of a processor is proportional to voltage squared, the processor's energy consumption can be considerably reduced by lowering CPU voltage and processing speed [Zhu et al. 2013].

Based on the fault tolerance approach presented in chapter 5 and DVFS technique, we propose in this chapter a fault-tolerant DVFS scheduling heuristic which generates, from a given hard real-time application and a given multiprocessor architecture, a task allocation scheme that minimizes energy consumption and tolerates *K* arbitrary transient faults.

The rest of the chapter is organized as follows. An overview of related work is provided in Section 2. The system models considered in this work are introduced in Section 3. The proposed fault-tolerance approach is explained in Section 4. The strategy that utilizes this approach and DVFS technique to minimize energy is explained in Section 5. The proposed

DVFS_FTS algorithm is presented in Section 6. Simulation results are discussed in Section 7, and finally, the conclusion is given in Section 8.

6.2. Related work

Several papers have been published are closely related to our research, these researches differ in many aspects, such as task models (dependent or independent tasks, hard or soft deadlines, periodic or aperiodic tasks), multiprocessor or uniprocessor platforms, online or offline scheduling and the fault-tolerance technique adopted.

Assayad et al. (2012) proposed a scheduling heuristic to minimize the schedule length, the global system failure rate and the power consumption of the generated schedule. Active replication of tasks and data dependencies is used to increase the system reliability and dynamic voltage scaling DVS is used for energy minimization. Gan *et al.* (2011) proposed a synthesis approach to decide the mapping of hard real-time applications on distributed heterogeneous systems, such that multiple transient faults are tolerated, and the energy consumed is minimized. For recovery from faults, they used replication technique.

Djosic and Jevtic (2013) developed a fault-tolerant DVFS algorithm for real-time application of independent tasks. This algorithm combines DVFS for optimizing energy consumption and re-execution recovery for fault tolerance, but their scope is restricted to single processor systems. Han *et al.* (2015) introduced an efficient method to determine the checkpointing scheme that can tolerate *k* transient faults on a single processor. Also, they proposed a task allocation scheme to reduce energy consumption.

The combination of replication and time-based redundancy techniques to tolerate multiple transient faults with low overhead in terms of energy consumption and total execution time has been studied in few works related to our research [Tavana et al.2014].

Tavana *et al.* (2014) have proposed a standby-sparing scheme which addressed simultaneously reliability and energy consumption. The proposed scheme by employing both hardware redundancy (standby-sparing) and time redundancy (re-execution) in some cases, can tolerate many transient faults. To reduce energy consumption, they applied two techniques DPM (Dynamic Power Management) used by the spare unit and DVS (Dynamic Voltage Scaling) used by the primary processor.

6.3. System models

6.3.1. Application model

The real-time application considered in this chapter consists of n hard aperiodic dependent tasks, denoted as $\Gamma = \{\tau_1, \tau_2, \dots, \tau_n\}$. Tasks are non-preemptive and cannot be interrupted by other tasks. Tasks send their output values in messages, when terminated. All required inputs have to arrive before activation of the task. Each task τ_i is characterised by a tuple (C_i, D_i), where C_i is the worst case execution time of the task at the maximum frequency/voltage in a fault free condition and D_i is the deadline of the task.

We model an application A as a Directed Acyclic Graph DAG. Each node represents one task. An edge e_{ij} indicates data-dependency between two tasks τ_i and τ_j.

6.3.2. Scheduling model

List scheduling is the most commonly used scheduling approach for dependent tasks represented by DAGs [Zhao et al. 2013]. Based on the *Earliest Deadline First* (EDF) scheduling policy, we propose an EDF list scheduling based heuristic which uses the concept of ready task and ready list. By ready task τ_i we mean that all τ_i's predecessors have been scheduled. The heuristic initializes the ready list with ready tasks and is looping while there is at least one task in the list. The ready tasks with the earliest deadline have the higher priority for scheduling.

6.3.3. Fault model

During the execution of an application, faults may be hard to avoid due to different reasons, such as hardware failure, software errors, devices exposed to intense temperatures, and external impacts [Zhang et al. 2015]. As a result, transient and intermittent faults are more frequent than permanent ones. In this chapter we consider these types of faults as their number has been dramatically higher.

From the fault tolerance point of view, transient faults and intermittent faults manifest themselves in a similar manner: they happen for a short time and then disappear without causing a permanent damage. Hence, fault tolerance techniques against transient faults are also applicable for tolerating intermittent faults and vice versa. Therefore, from now, we will refer to both types of faults as transient faults and we will talk about fault tolerance against transient faults, meaning tolerating both transient and intermittent faults.

6.3.4. Platform and Energy model

We assume that there are m homogeneous processors, each of them is DVFS enabled with a set of L operating frequencies. We denoted with $F = \{f_1, f_2, \dots, f_L\}$ with $0 \leq f_L \leq f_{L-1} \leq \dots \leq f_1 = f_{max}$. We assume the frequency values are normalized with respect to f_{max}, i.e. $f_{max} = 1$.

The energy model used in this work is the same to the one used in the literature [Djosic et al. 2013, Mahmood et al. 2017, Assayad et al. 2012 and Zhang et al. 2015], where the power consumption P of a system is given by Equation (6.1):

$$P = P_S + h(P_{ind} + P_d) = P_S + h(P_{ind} + C_{ef}\, V^2 f) \tag{6.1}$$

Where P_S is the static power, P_{ind} is the frequency-independent power and P_d is the frequency-dependent power. The parameter $h = 1$ when the system is in the working state. Otherwise, when the system is in the standby state, $h = 0$. C_{ef} is the effective loading capacitance and V is the supply voltage and it is a function of working frequency f. The static power can be removed only by turning off the whole system, P_{ind} is a constant independent of operating frequency. As the energy consumption due to frequency scaling is independent of P_S, we take into account only the frequency-dependent power P_d and we set $P_S = 0$. Hence, the power consumption P can be written as:

$$P = C_{ef}\, V^2 f \tag{6.2}$$

Since $f \propto V$, and according to Equation (6.2), the dynamic power P can be expressed as a polynomial of frequency of degree α, where α has been set to 3 in most of the published papers on energy consumption [Zhang et al. 2015, Zahaf et al. 2016]. Hence, we reformulate P in Equation (6.3) as:

$$P = C_{ef} f^3 \tag{6.3}$$

The energy consumed by task τ_i is given by Equation (6.4):

$$E_i(f_i) = C_{ef}\, C_i\, f_i^2 \tag{6.4}$$

Where C_i is the execution time of task τ_i under frequency f_i. The total energy E_{total} consumed by processors during the execution of a task set is expressed by Equation (6.5):

$$E_{total} = \sum_{i=1}^{n} E_i(f_i) \tag{6.5}$$

In this study, we consider only processor energy consumption.

6.4. The fault-tolerance approach

As explained in chapter 5, we combine uniform checkpointing with rollback recovery and active replication for tolerating k transient faults. We use these two techniques in order to meet time constraints and to increase the reliability of hard real-time applications even in the presence of faults.

6.4.1. Uniform Checkpointing with Rollback Recovery

By using this technique, once a fault is detected during the execution of the task τ_i, it needs to restore the saved state to continue task execution. We consider the following assumptions :

- The checkpointing is uniform (checkpoint intervals are equal for the same task).
- Faults are detected as soon as they occur.
- The checkpoint saving and rollback recovery are themselves fault-tolerant.

6.4.2. Collaborative active replication

In our work, we are interested in active replication. If there is enough time to rollback to the last saved checkpoint in the presence of faults, we use active replication to guarantee and respect task τ_i deadline. The task τ_i is replicated on two collaborative replicas; τ_i^1 and τ_i^2, both of which are be executed on different processors at the same time. We also introduce collaboration between replicas to tolerate multiple faults and respect task τ_i deadline.

For the sake of uniformity and clarity, we will consider the original task τ_i as the primary replica τ_i^1 and its replica as the backup replica τ_i^2. We consider the following assumptions:

- All checkpoints are assumed to be fault-free, i.e., no faults can occur during checkpoint saving.
- Each task's primary copy and backup copy must not be assigned to the same processor.
- Each task's primary copy and backup copy cannot be faulty at the same time.
- Faults are detected as soon as they occur, and the recovery will be with the no faulty replica.

Our goal is to tolerate k faults with respect to task τ_i deadline. To achieve this goal, we use active replication technique. However, it is possible that both primary and backup replicas are faulty due to multiple fault occurrence. Therefore, our goal will be missed, and active replication alone will be infeasible. This is the case in the work presented in [Motaghi and Zarandi 2014].

As a solution, we introduce collaboration between replicas to tolerate each coming fault in the primary or the backup replicas (τ_i^1, τ_i^2) to achieve the feasibility of our approach.

Once the active replication approach is decided for a task τ_i, we execute the following steps:

Step1: τ_i has to be scheduled on virtual processor *P#* ($\boldsymbol{\tau}_i$#) at start time ST_i as illustrated in Figure 6-1(a);

Step2: We insert in ($\boldsymbol{\tau}_i$#) the appropriate m_i^* checkpoints obtained with Equation (5.2);

Step3: τ_i is replicated, which will result in two replicas τ_i^1 and τ_i^2 which must be scheduled on two different processors;

Step4: The initial checkpoints of the task $\boldsymbol{\tau}_i$# are projected onto τ_i^1 and τ_i^2 alternatively, as illustrated in Figure 6-1(b).

The alternative checkpointing idea of the two replicas τ_i^1 and τ_i^2 is to ensure the collaboration between replicas and to minimize the number of checkpoints of the original task τ_i. In this case, we can meet the task deadline even in the presence of faults.

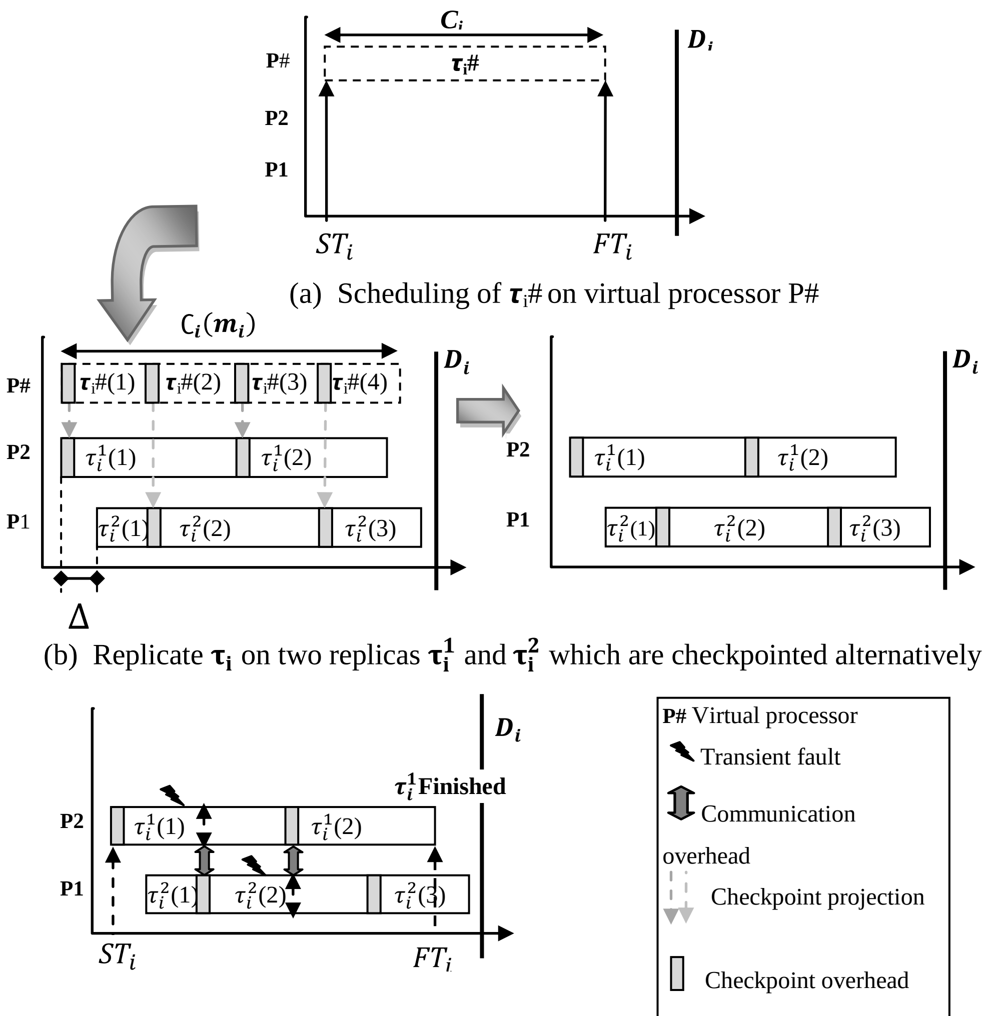


Figure 6-1 Illustration of different steps of collaborative active replication

In case of fault occurrence in the execution of one of the replicas (τ_i^1 or τ_i^2), the results produced by the no faulty replica must be sent to the faulty replica at checkpoint with Send/Receive communication to continue the execution. As shown in Figure 6-1(c), when fault affects the first execution interval $\tau_i^1(1)$, the no faulty replica τ_i^2 sends at checkpoint the correct state to the faulty task via communication step.

6.5. DVFS based fault-tolerance approach

The DVFS technique can assign different frequencies to each task, which gives us a useful way to minimize energy consumption of applications [Hu et al. 2016]. We extend the proposed fault-tolerance approach to incorporate it with DVFS to exploit the released slack time to achieve more energy saving.

According to the proposed fault-tolerance approach, we adopt active replication to meet timing constraints and provide high reliability even when deadlines are tight. However, task replicas must be performed at the maximum frequency given the probability of failure is low. We assume that DVFS is used during uniform checkpointing with rollback technique.

Similar to [Han et al. 2015], we assume that checkpointing is not affected by processor frequency. We focus on the fault-free execution and like [Salehi et al. 2016 and Melhem 2004], we aim to reduce the fault-free energy consumption because recovery executions have a small probability of being performed, and for this reason their energy consumption is a negligible fraction of the total energy consumption. The recovery time of a faulty task is always performed at the maximum frequency to preserve its original reliability.

6.5.1. Optimal frequency assignments

In this section, we search the optimal frequency assignments assuming all tasks their deadlines. In the existence of precedence constraints, a task may have to complete well before its deadline to ensure that all its successor tasks can finish in time. Therefore, as in [Zhao et al. 2013], we can define the effective deadline of a task τ_i as follows:

$$D_i^{ef} = \begin{cases} D_i & , succ(\tau_i) = \emptyset \\ min\left(D_i, D_j^{ef} - C_j\right) & , \tau_j \in succ(\tau_i) \end{cases} \tag{6.6}$$

Where $succ(\tau_i)$ is the set of successor tasks of τ_i .

The frequency f_i^{opt} that allows task τ_i to successfully complete execution before its deadline D_i^{ef} while minimizing energy consumption and tolerating K faults with checkpointing with rollback should satisfy the following:

$$ST_i + \frac{C_i(m_i)}{f_i^{opt}} + K * R_i(m_i) \leq D_i^{ef} \tag{6.7}$$

Where ST_i and $\frac{C_i(m_i)}{f_i^{opt}}$ are respectively the start time and the fault-free execution time of task τ_i with m_i checkpoints performed at frequency f_i^{opt}. $R_i(m_i)$ is the recovery time of τ_i under a single failure performed at the maximum frequency f_{max} ($C_i(m_i)$ and $R(m_i)$ were defined with Equations (6.5) and (5.1a) respectively).

After evaluation of Equation (6.7), we obtain the following solution:

$$f_i^{opt} \geq \frac{C_i(m_i)}{D_i^{ef} - ST_i - K * R_i(m_i)} \qquad (6.8)$$

If $f_i^{opt} \nexists F$,we choose neighboring frequencies $f_L < f_i^{opt} < f_{L-1}$ and $f_{L-1}, f_L \in F$.

Hence, the minimize energy consumed during the execution of task τ_i is given by:

$$E_i\left(f_i^{opt}\right) = C_{ef}\,\frac{C_i(m_i)}{f_i^{opt}}\,{f_i^{opt}}^2 = C_{ef} C_i(m_i)\, f_i^{opt} = C_{ef}\,\frac{C_i(m_i)^2}{D_i^{ef} - ST_i - K * R_i(m_i)} \qquad (6.9)$$

6.6. The proposed DVFS fault-tolerant scheduling algorithm

Our DVFS fault-tolerant schedule is presented in algorithm **DVFS_FTS**. The algorithm takes as input the application ***A***, the number ***K*** of transient faults that have to be tolerated, the architecture ***P***, the set of frequency levels ***F*** and the real-time constraints.

Our scheduling algorithm is a list scheduling based heuristic, which uses the concept of ready task and ready list. By ready task $\boldsymbol{\tau_i}$, we mean that all $\boldsymbol{\tau_i}$'s predecessors have been scheduled. The heuristic initializes the list *TReady* with tasks without predecessors in line 1 and is looping while *TReady* isn't empty (line 4-25). At first, the ready task $\boldsymbol{\tau_i}$ with minimum deadline is selected for placement in the schedule (line 5). Then, the maximum response time of the task $\boldsymbol{\tau_i}$ will be calculated with Equation (5.1) under maximum frequency (line 6). The checkpointing with rollback policy will be applied if the task deadline can be satisfied on the processor P_j at the earliest start time (line 10-13). In this case, the task $\boldsymbol{\tau_i}$ will be performed under the frequency f_i^{opt} calculated based on Equation (6.8) (line 12-13). Otherwise, the task $\boldsymbol{\tau_i}$ will be replicated and the proposed new active replication will be applied. In this case, the maximum response time of the task $\boldsymbol{\tau_i}$ will be calculated with Equation (5.7) under the maximum frequency (line 14-18). After execution of the task $\boldsymbol{\tau_i}$, its energy consumption will be calculated and the total energy will be updated in lines 22-23. Finally, the task $\boldsymbol{\tau_i}$ will be removed from the ready list *TReady* and all its successors are added to the list in line 24.

```
**DVFS_FTS Algorithm**
Inputs:
$\Gamma = \{\tau_1, \tau_2, \dots \tau_n\}$
$\mathcal{P} = \{P_1, P_2, \dots, P_m\}$
$F = \{f_1, f_2, \dots, f_L\}$
K transient faults for each task
Real-time constraints
1. TReady = $\{\tau_i \in \Gamma \mid pred(t_i) = \emptyset\}$
2. Schedulable = True
3. $E_{total} = 0$
4. While $TReady \neq \emptyset$ do
5. {  Select $\tau_i \in TReady$ having the minimum deadline $D_i$
6.    Compute $WCRT_i$ with Equation (5.1)under maximum frequency
7.    Compute the start time $ST_{ij}$ of $\tau_i$ on all processor $P_j$ in $\mathcal{P}$
8.    $ST_i = min_{j=1..m} ST_{ij}$
9.    If $D_i - ST_i \geq WCRT_i$ then
10.     { Schedule $\tau_i$ on $P_j$ at the earliest start time     /* $P_j$
   is the processor with min $ST_i$*/
11.       Apply checkpointing for $\tau_i$
12.       Compute $f_i^{opt}$ based on Equation(6.8)
13.       Perform $\tau_i$ under $f_i^{opt}$ frequency }
14.     Else
15.     { Compute $WCRT_i$ with Equation(5.7)under maximum frequency
16.       If $D_i - ST_i \geq WCRT_i$ then
17.        { Schedule both $\tau_i$ on $P_j$ and its replica on another
   processor $P_k$ at the earliest start time.
18.          Apply collaborative active replication for $\tau_i$ }
19.       Else
20.        { Schedulable = False
21.          Break  }}
22.Compute the energy consumption $E_i(f_i)$
23.$E_{total} = E_{total} + E_i(f_i)$
24.TReady = TReady- $\{\tau_i\} \cup \{\tau_j \in succ(\tau_i) \mid pred(\tau_j) \nexists TReady\}$
    }
**End DVFS_FTS**
```

Figure 6-2 The proposed DVFS_FTS algorithm

6.7. Performance evaluation

In this section, we evaluate the performance of the proposed DVFS_FTS algorithm. For comparison we have implemented our algorithm and the following schemes:

EXH_FTS: Fault tolerant scheduling algorithm with energy minimization using exhaustion method.

DVFS_CH: Fault tolerant scheduling algorithm that uses checkpointing with roll back technique for fault tolerance and DVS for reduce energy. This algorithm is extended from JFTT scheme [Zhang et al. 2006] for tasks with precedence constraints (application DAG).

The performance is measured in term of normalized total energy saving. We formulate the parameter energy saving ES in Equation (6.10).

$$ES = 100 * \frac{E_{FTS} - E}{E_{FTS}} \tag{6.10}$$

Where E_{FTS} is the energy consumption of the proposed algorithm with all tasks are executed at the highest frequency and E is the energy consumption of a compared algorithm with DVFS scheme.

6.7.1. Simulation parameters

Table 6-1. Parameters for simulation

Parameter	Value(fixed-varied)
Number of processors	4
Application size (Number of tasks)	(10 , 20 , 30 , 40 , 50)
Execution time (ms)	[10 , 100]
Normalized frequency	[0.1 – 1] with a step of 0.1
Checkpoint overhead *O*	(1%, 2%, 5%, 10%, 15%, 20%)
Number of faults *K*	(1 , 2 , 3 , 4 , 5)

Before presenting our experimental results, we present the simulation parameters as follows: The method of generating random graphs is the same as [Qamhieh 2015]. We have generated a set of DAG applications with 10, 20, 30, 40 and 50 tasks. Within a task set, the worst case response time on maximum operating frequency C_i for each task is randomly generated with values uniformly distributed in the range of [10ms, 100ms]. We assume $C_{ef} = 1$ and the operating frequencies are set as $F = \{0.1, 0.2, \dots, 1\}$. The parameters and the values used in our simulation are summarized in Table 6-1.

6.7.2. Experiment results

The first set of experiments compares the energy savings of algorithms with respect to number of transient faults (see Figure 6-3). In this experiments, we set application size $\Gamma = 10$ tasks, the checkpoint overhead O = 2% and vary K from 1 to 5. As can be seen clearly from the figure that the performance on energy saving of DVFS_FTS algorithm outperforms both DVFS_CH and EXH_FTS schemes. For instance, when the number of transient faults is 5 faults, the *ES* of DVFS_FTS is greater than DVFS_CH and EXH_FTS by 7.17% and 6.34% respectively. Furthermore, we can observe that the energy savings of the three algorithms decreases with the increase of the number of transient faults.

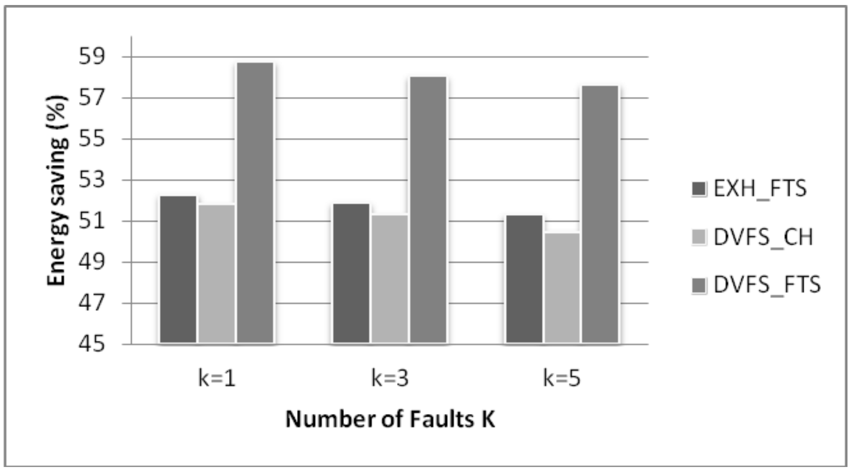


Figure 6-3 The impact of number of faults on energy saving.

The second set of experiments is to investigate the performance of the different approaches with respect to application size (see Figure 6-4). In this set of experiments, we set the checkpoint overhead O = 2% and K =3 and vary the application size Γ from 10 tasks to 50 tasks. We can see that the energy saving increases when the number of tasks increases. The energy saving of DVFS_FTS is greater than DVFS_CH and EXH_FTS schemes by: (6.73%, 6.18%), (6.76%, 5.8%), (7.68%, 6.75%), (8.74%, 8.45%), (8.61%, 8.8%) for number of tasks of 10, 20, 30, 40 and 50, respectively. The results of our proposed algorithm always outperform those of the others, which show the efficiency of the DVFS_FTS algorithm.

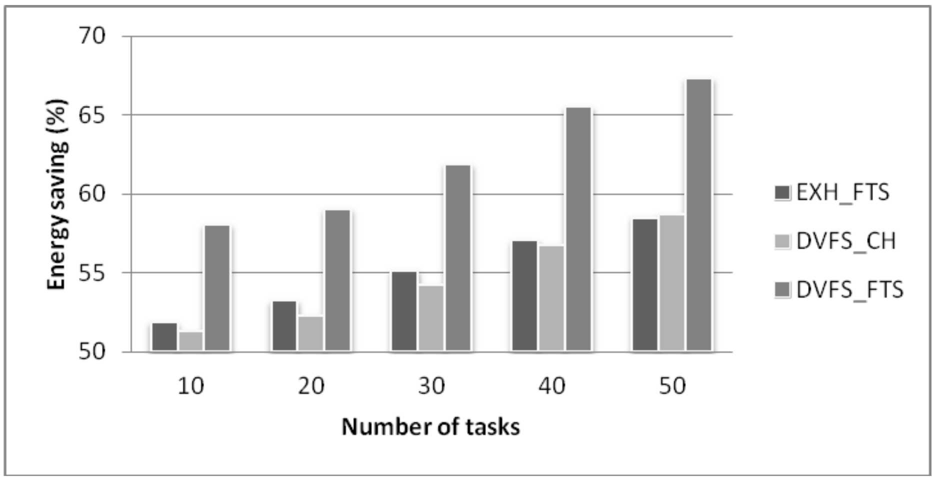


Figure 6-4 The impact of application size on energy saving considering *K*=3 faults.

In the third set of experiments, we show the impact of checkpointing overhead on the performance of algorithms (see Figure 6-5). In this set of experiments, we set application size $\Gamma = 20$ tasks, K =3 faults and vary O from 1% to 20%. As can be seen from the figure, the energy saving of the three schemes decreases when O increases. However, the *ES* of DVFS_FTS decreases about 5.87% when O increases from 1% to 20% and less than the *ES* of DVFS_CH and EXH_FTS decrease about 6.5% and 6.76% respectively.

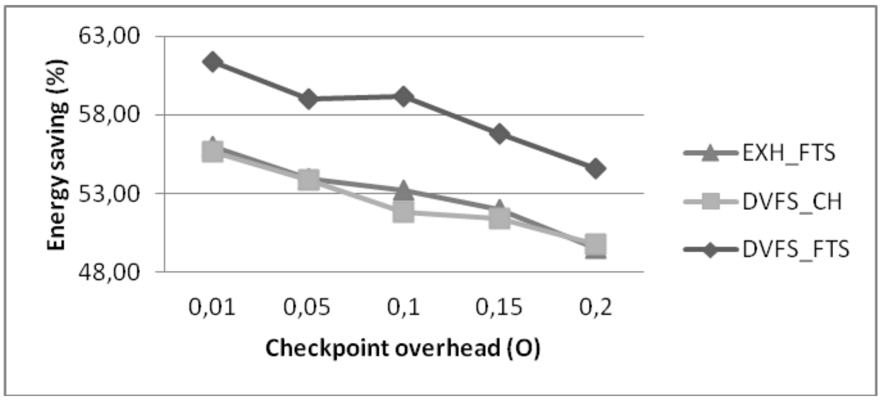


Figure 6-5 The impact of checkpoint overhead on energy saving considering *K*=3 faults.

From these experiments, we can resume that the proposed algorithm DVFS_FTS outperforms the other two algorithms.

6.8. Conclusion

In this chapter, we have studied the trade-off between fault tolerance and energy minimization in hard real-time systems running on multiprocessor platforms. We have extended the fault tolerance approach already presented in chapter 5 with DVFS technique to attain more energy saving. Then, we have presented our fault-tolerant scheduling algorithm DVFS_FTS that exploits DVFS technology to reduce energy consumption and the proposed fault-tolerance approach to tolerating *K* transient faults for applications that can be modeled with a DAG (precedence-constrained applications). Simulation results have shown that the proposed algorithm achieves a considerable amount of energy saving compared to EXH_FTS and DVFS_CH algorithms.

CHAPTER 7

Conclusion and Future Work

Nowadays, embedded real-time systems are prevalent in our daily lives. They are growing rapidly in both scale and complexity. However, these progresses have brought unprecedented challenges for designing of these systems that are subject to a variety of constraints, e.g. timing, reliability and power. The reliability of these systems can be increased by the use of fault tolerance techniques.

Fault tolerance is a well established research topic and it offers a wide variety of techniques to achieve correct operation even in the presence of errors. Checkpointing with rollback Recovery is one technique that efficiently copes with transient faults. The advantage of this technique over other fault tolerance techniques is that it is not as costly as other techniques which require a significant amount of hardware redundancy, and in case of errors, the faulty segment of the task that is being executed instead of restarting the task from the beginning. The main drawback of checkpointing with rollback is that it introduces a time overhead which depends on the number of checkpoints that are used. Active replication technique usually requires extra system resources e.g. processor, and consume more energy even under the fault-free scenarios, but they can tolerate faults timely and promptly.

In this dissertation, we focused our efforts on developing an efficient and effective approach for fault tolerant scheduling of hard real-time systems with the purpose of providing guarantees to timing constraints under transient faults while optimizing power consumption.

We started by studying how we can combine fault tolerance techniques of the two basic classes of redundancy: time-based redundancy and spatial redundancy to tolerate spatially or temporally faults, satisfy critical task constraint and explore hardware resources. For this purpose, we have proposed a new fault-tolerance approach that combines checkpointing with rollback technique and active replication in an original scheme.

We have also introduced replicas collaboration to tolerate spatially or temporally faults and satisfy critical task constraint. To the best of our knowledge, this is the first work introducing the idea of collaboration between replicas in active replication technique with checkpointing.

The proposed approach classifies the real-time tasks into critical and noncritical ones, according to the utilization of the task. For the noncritical task, we adopt checkpointing with rollback technique to tolerate multiple transient faults. Whereas for the critical task, we adopt active replication as it is the fault tolerant method that explores hardware resources to meet timing constraints and provide high reliability even when deadlines are tight.

Based on this approach, we have proposed two fault tolerant scheduling algorithms: SFTS and DVFS-FTS algorithms. SFTS is a static fault tolerant scheduling algorithm. It uses task utilization to decide which fault tolerance method will be applied (checkpointing with rollback or active replication) to tolerate spatial and temporal faults. While DVFS-FTS is an EDF list scheduling based heuristic. It exploits DVFS technology and the proposed fault-tolerance approach to reduce energy consumption and tolerating *K* transient and intermittent faults for applications that can be modeled with a DAG.

Future work

Our work remains opening to future contributions like:

- Evaluate our fault-tolerant scheduling algorithms on real-life applications.
- We are interested in extending our efficient fault-tolerant scheduling algorithms to address the problem of online fault-tolerant scheduling of application with mixed-critical tasks in heterogeneous architecture. In addition of tolerating transient faults, we will take into consideration permanent faults.
- We are interested in studying a new state of checkpointing scheme to minimize as possible the number of checkpoints as a result minimize a time overhead.
- Another potential point for further development would be to extend the fault model and consider more faults e.g. permanent faults. Then, this would open possibility for implementing further fault tolerant mechanism to deal with newly considered faults and their effects.

References

Arlat, J., Crouzet, Y., Deswarte, Y., Fabre, J.C., Laprie, J and Powell, D. (2006). Encyclopédie de l'informatique et des systemes d'information. Chapitre Tolérance aux fautes, pages 241–270. Vuibert, 2006.

Assayad, I., Girault, A. and Kalla, H. (2012). Scheduling of real-time embedded systems under reliability and power constraints. International Conference on Complex Systems (ICCS), Agadir, Morocco, November 2012, IEEE.

Avizienis, A., Laprie, JC., Randell, B. (2004). Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable And Secure Computing, Vol. 1, No. 1, January-March 2004

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

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

Bachir, M. (2019). Ordonnancement tolérant aux fautes pour les systèmes distribues temps réel embarques. PhD thesis 2019.

Baruah, S. (2013). Partitioned EDF scheduling: a closer look. Real-Time Systems, 49(6):715–729, 2013.

Baruah, S. K., Cohen, N. K., Plaxton,C. G. and Varvel, D. A. (1996). Proportionate progress: a notion of fairness in resource allocation. Algorithmica, 15(6):600–625, June 1996

Baruah, S. and Fisher, N. (2007a). The partitioned dynamic-priority scheduling of sporadic task systems. Real-Time Systems, 36(3):199–226, 2007.

Baruah, S. and Fisher, N. (2007b). Global deadline-monotonic scheduling of arbitrary deadline sporadic task systems. In E. Tovar, P. Tsigas, and H. Fouchal, editors, 148 Principles of Distributed Systems, volume 4878 of Lecture Notes in Computer Science, pages 204–216. Springer Berlin Heidelberg, 2007

Benoit, A., Hakem. M. and Robert, Y. (2008). Contention awareness and fault tolerant scheduling for precedence constrained tasks in heterogeneous system. Parallel Computing, Elseiver 2008.

Benso, A., Di Carlo, S., Di Natale, G. and Prinetto, P. (2003). A watchdog processor to detect data and control flow errors. Proc. 9th IEEE On-Line Testing Symp., 144-148, 2003.

Besseron , X. (2010). Tolérance aux fautes et reconfiguration dynamique pour les applications distribuées a grande échelle. PhD thesis, National Institut Polytechnique -Grenoble - inpg, 2010.

Bizot, G. (2012). Gestion de l'activite et de la consommation dans les architectures multi-coeurs massivement paralleles. PhD thesis, Université de Grenoble, 2012.

Burns, A. and Wellings, A. J. (2001). Real-time systems and programming languages: ADA 95. Real-Time Java, And Real-Time Posix. Pearson Education, 2001.

Chen, J.-J., Yang, C.-Y., Kuo,T.-W. and Tseng, S.-Y. (2007). Real-time task replication for fault tolerance in identical multiprocessor systems. Real-Time and Embedded Technology and Applications Symposium, IEEE, 0:249–258, 2007.

Derasevic, S. (2018). Node fault tolerance for distributed embedded systems based on Ftt-Ethernet. PhD thesis, 2018.

Digalwar, M. A. (2016). Energy efficient multicore scheduling algorithms for real time systems. PhD thesis, 2016.

Dima, C., Girault, A. and Sorel,.Y. (2004). Static fault-tolerant real-time scheduling with pseudo-topological orders. Proceeding of Joint Conference, Grenoble, France, 2004. Volume 3253 of LNCS, Springer-verlag.

Dima, C., Girault, A. Lvarenne, C. and Sorel, Y. (2001). Off-line real-time fault-tolerant scheduling. Euromicro Workshop on parallel and distributed processing, Italy, 2001.

Djosic, S. and Jevtic, M. (2013). Dynamic voltage and frequency scaling algorithm for fault tolerant real-time systems. Microelectronics Reliability journal of Elsevier, Vol. 53, July 2013, pp. 1036-1042.

Dubrova, E. (2013). Fault-tolerant design. Springer, no ISBN 978-1-4614-2113- 9, 15 Mars 2013,185 p.

Ejlali, A., Al-Hashimi, B. and Eles, P. (2012). A standby-sparing technique with low energy-overhead for fault-tolerant hard real-time systems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 31, March 2012, No. 3, pp. 329-342.

Ejlali, A. B., Al-Hashimi, M., Schmitz, M., Rosinger, P. and Miremadi, S. G. (2006). Combined time and information redundancy for SEUtolerance in energy-efficient real-time Systems. Very Large Scale Integration (VLSI) Systems, 14(4), pp. 323-335, 2006.

Eles, P., Izosimov,V., Pop, P.and Peng, Z. (2008). Synthesis of fault-tolerant embedded systems. In Proceedings of 2008 Design Automation and Test in Europe Conference (DATE), Munich, Germany, March 2008, pp. 1117-1122.

Fan, M. and Quan, G. (2011). Harmonic-Fit partitioned scheduling for fixed-priority real-time tasks on the multiprocessor platform. In Embedded and Ubiquitous Computing (EUC), 2011 IFIP 9th International Conference on, pages 27–32, Oct 2011.

Fan, M. and Quan, G. (2012). Harmonic semi-partitioned scheduling for fixed-priority real-time tasks on multi-core platform. In Proceedings of the Conference on Design, Automation and Test in Europe, DATE '12, pages 503–508, San Jose, CA, USA, 2012. EDA Consortium.

Fan, M., Han, Q., Quan, G. and Ren, S. (2014). Multi-core partitioned scheduling for fixed-priority periodic real-time tasks with enhanced rbound. In Quality Electronic Design (ISQED), 2014 15th International Symposium on, pages 284–291, March 2014.

Forget, J., Grolleau, E., Pagetti, C. and Richard, P. (2014). Dynamic priority scheduling of periodic tasks with extended precedences. tel-00978366, version 1 - 14 Apr 2014 160 In IEEE

16th Conference on Emerging Technologies Factory Automation (ETFA), Toulouse, France, September 2011.

Gan; J., Gruian; F., Paul Pop, P. and Madse. J. (2011). Energy/Reliability trade-offs in fault-tolerant event-triggered distributed embedded systems. In Proc. of the 16th Asia South Pacific Design Automation Conference ASP-DAC, 2011, pp. 731-736. https://doi.org/10.1109/ASPDAC.2011.5722283.

Girault, A. and Kalla, H. (2009). A novel bicriteria scheduling heuristics providing a guaranteed global system failure rate. IEEE Transactions on Dependable and Secure Computing, 2009, Vol. 6, Oct.-Dec. 2009, pp. 241-254

Girault, A. and Kalla, H. (2009). A novel bicriteria scheduling heuristics providing a guaranteed global system failure rate. IEEE Trans. on Dependable and Secure Computing, 2009, 6: 241-254. https://doi.org/10.1109/TDSC.2008.50

Girault, A., Kalla, H. and Sorel, Y. (2004). A scheduling heuristics for distributed real-time embedded systems tolerant to processor and communication media failures. International Journal of Production Research, 2004, Vol. 42, pp. 2877-2898.

Gui, S. and Luo, L. (2013). Reliability analysis of real-time fault-tolerant task models. Design Automation for Embedded Systems, Vol. 17, March 2013, pp. 87-107.

Gupta, R. (2004). Dynamic voltage scaling for system wide energy minimization in real-time embedded systems. In Low Power Electronics and Design, 2004. ISLPED ’04. Proceedings of the 2004 International Symposium on, pages 78– 81, Aug 2004.

Han, Q. (2015). Energy-aware fault-tolerant scheduling for hard real-time systems. PhD thesis, Florida International University, 2015

Han, Q., Fan, M., Nin, L. and Quan, G. (2015). Energy minimization for fault tolerant scheduling of periodic fixed-priority applications on multiprocessor platforms. In Proceedings of 2015 Design, Automation and Test in Europe Conference and Exhibition (DATE), Grenoble, France, March 2015, pp. 830-835.

Han; Q., Fan, M. and Quan, G. (2013). Energy minimization for fault tolerant real-time applications on multiprocessor platforms scheduling using checkpointing. In Low Power Electronics and Design (ISLPED), IEEE International Symposium on, Beijing, China, September 2013, pp. 76-81. https://doi.org/10.1109/ISLPED.2013.6629270

Han, C.-C., Shin, K. and Wu, J. (2003). A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults. Computers, IEEE Transactions on, 52(3):362 – 372, march 2003.

Hashimoto, K, Tsuchiya,. T. and Kikuno, T. (2002). Effective scheduling of duplicated tasks for fault-tolerance in multiprocessor systems. IEICE Transactions on Information and Systems, E85-D(3):525–534, march 2002.

Hu, Y., Liu, C., Li, K., Chen, X. and Li, K. (2016). Slack allocation algorithm for energy minimization in cluster systems. Future Generation Computer Systems, vol. 74, pp. 119-131, 2016. https://doi.org/10.1016/j.future.2016.08.022

Huang, J., Blech, J., Raabe, A., Buckl, C. and Knoll, A. (2011). Analysis and optimization of fault-tolerant task scheduling on multiprocessor embedded systems. In Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS '11, pages 247–256, New York, NY, USA, 2011. ACM.

Quan, G. and Niu, L. (2004). Fixed priority scheduling for reducing overall energy on variable voltage processors. In In 25th IEEE Real-Time System Symposium, pages 309–318. IEEE Computer Society, 2004.

Qin, X. and Jiang, H. (2006). A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems. Science Direct, Parallel Computing 32 (2006) 331–356

Izosimov, V. (2009). Scheduling and optimization of fault-tolerant distributed embedded systems. PhD thesis, Linköping University, 2009.

Izosimov, V., Pop, P., Eles, P. and Peng, Z. (2008). Scheduling of fault tolerant embedded systems with soft and hard timing constraints. In Proceedings of 2008 Design, Automation and Test in Europe Conference (DATE), Munich, Germany, March 2008, pp. 915-920.

Izosimov, V., Pop, P., Eles, P. and Peng, Z. (2012). Scheduling and optimization of fault-tolerant embedded systems with transparency/ performance trade-offs. ACM Trans. Embedded computing systems (TECS), Vol. 11, September 2012, Article No. 61.

Jimenez, M., Palomera, R. and Couvertier. I. (2014). Introduction to embedded systems using microcontrollers and the MSP430. Springer 2014. DOI: 10.1007/978-1-4614-3143-5_1

Kada B. and Kalla, H. (2019a). An efficient fault-tolerant scheduling approach with energy minimization for hard real-time embedded systems. CIT , 2019.

Kada B. and Kalla, H. (2019b). A Fault-Tolerant Scheduling Algorithm Based on Checkpointing and Redundancy for Distributed Real Time Systems. IJDST, Vol 10, issue 3, article 4, July-Sept 2019. DOI https://doi.org/10.4018/IJDST.2019070104.

Kalla, H. (2004). Génération automatique de distributions/ordonnancements temps réel fiables et tolérant les fautes. PhD thesis, Institut national polytechnique de Grenoble, spécialité systèmes et logiciel, 2004.

Kermia, O., Ordonnancement temps-réel multiprocesseur de taches non-préemptives avec contraintes de précédences, de périodicité stricte et de latence. PhD thesis, Université Paris XI, 2009.

Kim, J., Lakshmanan, K. and Rajkumar, R. R.. (2010). R-batch: Task partitioning for fault-tolerant multiprocessor real-time systems. In Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology, CIT '10, pages 1872–1879, Washington, DC, USA, 2010. IEEE Computer Society.

Krakowiak, S. (2004). Tolerance aux fautes – 1 introduction, techniques de base. Ecole doctorale de Grenoble Master 2 Recherche "Systèmes et Logiciel", Université Joseph Fourier Projet Sardes (INRIA et IMAGLSR), 2003-2004.

Krishna, C. M. (2014). Fault-tolerant scheduling in homogeneous real-time systems. ACM Computing Surveys, Vol. 46, April 2014, Article No. 48.

Kumar, A. and Alam, B. (2015). Improved EDF algorithm for fault tolerance with energy minimization. IEEE International Conference on Computational Intelligence & Communication Technology (CICT), Ghaziabad, India, February 2015. doi: 10.1109/CICT.2015.84

Kumar, P. and Rachit, G. (2011). Soft-checkpointing based hybrid synchronous checkpointing protocol for mobile distributed systems. International Journal of Distributed Systems and Technologies, 2(1), 1-13. doi:10.4018/jdst.2011010101

Laprie, JC. (1995) Dependability - ITS ATTRIBUTES, IMPAIRMENTS AND MEANS. In: Randell B., Laprie JC., Kopetz H., Littlewood B. (eds) Predictably Dependable Computing Systems. ESPRIT Basic Research Series. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-79789-7_1

Legout, V. (2014). Ordonnancement temps réel multiprocesseur pour la réduction de la consommation énergétique des systèmes embarqués. PhD thesis, Institut Des Sciences Et Technologies Paris. 2014.

Leung, J. Y.-T. and Whitehead, J. (1982). On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks. Performance Evaluation, Vol 2, Issue 4, December 1982, Pages 237-250. https://doi.org/10.1016/0166-5316(82)90024-4.

Li, L., Choi, K., & Nan, H. (2011). Effective algorithm for integrating clock gating and power gating to reduce dynamic and active leakage power simultaneously. Proceedings of 12th International Symposium On Quality Electronic Design, pp. 1-6.

Li, Z., Ren, S. and Quan, G. (2015). Energy minimization for reliability-guaranteed real-time applications using DVFS and checkpointing techniques. Journal of Systems Architecture, vol. 61, pp. 71-81, 2015. http://dx.doi.org/10.1016/j.sysarc.2014.12.002

Li, Z.; Wang;L., Ren, S. and Quan, G. (2013). Energy minimization for checkpointing-based approach to guaranteeing real-time systems reliability. In Proc. IEEE 16th Int. Symp. Object/Compon./Service-Oriented Real-Time Distrib. Comput. (ISORC), 2013, pp. 1-8. http://dx.doi.org/10.1109/ISORC.2013.6913209

Lin, C-H. and Liao C.-J. (2008). Makespan minimization for multiple uniform machines. Comput. Ind. Eng., 54(4):983–992, 2008

Liu, J. (2000). Real-Time Systems. Prentice Hall, NJ, 2000.

Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. Journal of ACM, 20(1), pp. 46-61, January 1973.

Mahmood, A., Khan, S., Albalooshi, F. and Awwad, N. (2017). Energy-aware real-time task scheduling in multiprocessor systems using a hybrid genetic algorithm. Electronics, 2017, 6(2), 40. http://dx.doi.org/10.3390/electronics6020040

Marouf, M. (2012). Ordonnancement temps réel dur multiprocesseur tolérant aux fautes appliqué à la robotique mobile. PhD thesis, Ecole nationale supérieure des mines de paris, Spécialité Informatique T. R, Robotique et Automatique, 2012.

Megel, T. (2012). Placement, ordonnancement et mécanismes de migration de tâches tempsréel pour des architectures distribuées multicœurs. PhD thesis, Institut National Polytechnique de Toulouse (INP Toulouse), 2012.

Melhem, R., Mosse, D. and Elnozahy, E. (2004). The interplay of power management and fault recovery in real-time systems. IEEE Trans. Comput., 2004, 53: 217-231. https://doi.org/10.1109/TC.2004.1261830

Meroufel, B.and Belalem, G. (2014). Collaborative services for fault tolerance in hierarchical data grid. International Journal of Distributed Systems and Technologies, 5(1), 1-21, January-March 2014. doi: 10.4018/ijdst.2014010101

Motaghi, M. H.and Zarandi, H. R. (2014). DFTS : Dynamic fault-tolerant scheduling for real-time tasks in multicore processors. Microprocessors and Microsystems Journal of Elsevier, Vol. 38, February 2014, pp. 88-97.

Ndoye, F. (2014). Ordonnancement temps réel préemptif multiprocesseur avec prise en compte du coût du système d'exploitation. PhD thesis. Université Paris-Sud, 2014

Nicolescu, B., Savaria, Y. and Velazco, R. (2004). Software detection mechanisms providing full coverage against single bit-flip faults. IEEE Trans. on Nuclear Science, 51(6), 3510-3518, 2004.

Nikolov, D. (2015). Fault tolerance for real-time systems analysis and optimization of roll-back recovery with checkpointing. Doctoral Dissertation Fault-tolerant computing Lund, January 2015

Oh, Y. and Son, S. H. (1995). Allocating fixed-priority periodic tasks on multiprocessor systems. Real-Time Syst., 9(3):207–239, Nov. 1995.

Oh, Y. and Son, .S. H. (1997). Scheduling real-time tasks for dependability. Journal of Operational Research Society, 48(6):629–639, June 1997

Oh, N., Shirvani, P. P., and McCluskey, E. J. (2002a). Control-flow checking by software signatures. IEEE Trans. on Reliability, 51(2), 111-122, 2002

Oh, N., Shirvani, P. P., and McCluskey, E. J. (2002b). Error detection by duplicated instructions in super-scalar processors. IEEE Trans. on Reliability, 51(1), 63-75, 2002.

Oh, N., Shirvani, P. P., and McCluskey, E. J. (2002c). Error detection by selective procedure call duplication for low energy consumption. IEEE Trans. on Reliability, 51(4), 392-402, 2002.

Pop, P., Izosimov, V., Eles, P. and Peng, Z. (2009). Design optimization of time-and- cost-constrained fault-tolerant embedded systems with checkpointing and replication. IEEE Trans. Very Large Scale Integration (VLSI) Systems, Vol. 17, March 2009, pp. 389-340.

Pop, P., Poulsen, K. H., Izosimov, V. and Eles, P. (2007). Scheduling and voltage scaling for energy/reliability trade-offs in fault tolerant time-triggered embedded systems. Hardware/software code-sign and system synthesis, pp. 233-238, 2007.

Peti, P., Obermaisser, R. and Kopetz, H. (2005). Out-of-Norm Assertions. Proc. 11th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS), 209- 223, 2005.

Qamhieh, M. (2015). Scheduling of parallel real-time dag tasks on multiprocessor systems. Ph.D thesis, Paris-Est University, 2015.

Qin, X. and Jiang, H. (2006). A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems. SceinceDirect, Parallel computing 2006.

Ropars, T., Lefray, A., Kim, D. and Schiper, A. (2015). Efficient process replication for MPI applications: sharing work between replicas. 29th IEEE International Parallel & Distributed Processing Symposium (IPDPS2015), Hyderabad, India, 2015. <hal-01121959>

Runge, A. (2012). Reliability enhancement of fault-prone many-core systems combining spatial and temporal redundancy. 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012.

Salehi, M., Tavana, M., K., Rehman, S., Shafique, M., Ejlali, A. and Henkel, J. (2016). Two-state checkpointing for energy-efficient fault tolerance in hard real-time systems. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 24, July 2016, pp. 2426 – 2437.

Samal, A. K., Mall, R. and Tripathy, C. (2014). Fault tolerant scheduling of hard real-time tasks on multiprocessor system using a hybrid genetic algorithm. Swarm and Evolutionary Computation journal of Elsevier, Swarm and Evolutionary Computation Journal of Elsevier, Vol. 14, February 2014, pp. 92-105.

Saraswat, P. K., Pop, P. and Madsen, J. (2010). Task mapping and bandwidth reservation for mixed hard/soft fault-tolerant embedded systems. In Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE (pp. 89-98). IEEE.

Shin, K. and Ramanathan, P. (1994). Real-Time Computing: A New Discipline of Computer Science and Engineering. Proc. IEEE, 82(1):6–24, Jan. 1994.

Srinivasan, J. Adve, S. V., Bose, P., Rivers, J. A., Hu, C. K., Emma, P., Linder, B. and Wu, E. Y. (2003). Ramp: A model for reliability aware microprocessor design. IBM Research Report, RC23048, 2003.

Tavana, M. K., Teimouri, N., Abdollahi, M. and Goudarzi, M. (2014). Simultaneous hardware and time redundancy with online task scheduling for low energy highly reliable standby-sparing system. ACM Trans. Embedded Computing Systems, Vol. 13, Issue 4, November 2014, Article No. 86.

Wei, T., Mishra, P., Wu, K. and Zhou, J. (2012). Quasi-static fault tolerant schemes for energy-efficient hard real-time systems. Systems and Software Journal of Elsevier, Vol. 85, June 2012, pp. 1386-1399.

Yahiaoui, K. L'apport des outils de l'intelligence artificielle dans les systemes temps-reel : ordonnancement des taches. PhD thesis, Université d'Oran, 2013

Zahaf, H. E. (2016). Energy efficient scheduling of parallel real-time tasks on heterogeneous multicore systems. Ph.D thesis, Université de Lille 1, Sciences et Technologies, 2016.

Zammali, A. (2016). Approche d'intégrité bout en bout pour les communications dans les systèmes embarqués critiques : application aux systèmes de commande de vol d'hélicoptères. PhD thesis, Université Paul Sabatier - Toulouse III, 2016.

Zarinzad,] G., Rahmani, A.M. and Dayhim, N. (2008). A novel intelligent algorithm for fault tolerant task scheduling in real-time multiprocessor systems. In: Proceedings of the Third International Conference on Convergence and Hybrid Information Technology; 2008 Nov 11–13; p. 816–21.

Zhang, J., Sha, E. H-M., Zhuge, Q., Yi, J. and Wu, K. (2014). Efficient fault-tolerant scheduling on multiprocessor systems via replication and deallocation. International Journal Embedded Systems, Vol. 6, Nos. 2/3, 2014, pp. 216–224.

Zhang, L., Li, k., Xu, Y., Ying, M., Zhang, F. and Li, K. (2015). Maximizing reliability with energy conservation for parallel task scheduling in a heterogeneous cluster. Information Sciences, vol. 319, pp. 113-131, 2015. https://doi.org/10.1016/j.ins.2015.02.023.

Zhang, Y. and Chakrabarty, K. (2006). A unified approach for fault tolerance and dynamic power management in fixed-priority real-time embedded systems. IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, Vol. 25, Issue 1, January 2006, pp. 111-125.

Zhang, Y. Chakrabarty, K. and Swaminathan ,V. (2003). 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, pages 209–, Washington, DC, USA, 2003. IEEE Computer Society.

Zhao, B., Aydin, H. and Zhu, D. (2013). Shared recovery for energy efficiency and reliability enhancements in real-time applications with precedence constraints. ACM Trans. Des. Autom. Electron. Syst., 18, 2, Article 23 (March 2013), 21 pages. http://dx.doi.org/10.1145/2442087.2442094

Zhu, X., Ge, R., JinguangSun, J. and He, C. (2013). Energy-efficient elastic scheduling for independent tasks in heterogeneous computing systems. Systems and Software Journal of Elsevier, Vol. 86, pp. 302-314, 2013. https://doi.org/10.1016/j.jss.2012.08.017

Zhu, D., Melhem, R. and Mosse, D. (2004). The effects of energy management on reliability in real-time embedded systems. Proc. of the International Conference on Computer Aided Design, pp. 35-40, 2004.

Zhu, D. and Aydin, H. (2009). Reliability-aware energy management for periodic real-time tasks. IEEE Transactions on Computers, 58(10), pp. 1382 – 1397, 2009.