Energy-aware and Fault-tolerant Design Strategies for Safety-critical Systems

Thesis submitted to the
Indian Institute of Technology Guwahati
for the award of the degree

of

Doctor of Philosophy
in
Computer Science and Engineering

Submitted by
Piyoosh P

Under the guidance of
Dr. Arnab Sarkar and Dr. Santosh Biswas

Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
February, 2020

Abstract

Nowadays, safety-critical embedded systems are increasingly being implemented on computing platforms which involve complex microarchitectural designs with multi-million gates per chip and small feature sizes, to meet the computation and performance demands of the applications. Although, such intricate sophistication is inevitable, they come with associated side-effects such as, high energy dissipation and increase in the probability of faults. In addition, a significant class of contemporary embedded systems are driven by limited energy sources like batteries. On the other hand, there is an increasing trend in many futuristic safety-critical systems to be shipped with quantifiable measures of system reliability. Increased reliability/tolerance against faults are typically achieved through additional hardware resources and/or by utilizing residual system capacity. Similarly, execution slowdown and switching parts of the system into inactive low-power states, are two commonly used system level strategies towards enhancing energy efficiency. Therefore, energy-awareness and fault tolerance have emerged as critically important design constraints in the development of modern safety-critical systems. *In this dissertation, we present a few novel scheduling and modeling strategies for safety-critical systems which aim to achieve one of the following objectives: 1) energy minimization, 2) fault tolerance, 3) fault tolerance with energy-awareness, and 4) low-overhead fault detection.*

The entire thesis work is composed of multiple distinct contributions which are categorized into four phases. In the first phase, we have developed a procrastination based scheduling methodology to minimize static energy consumption in a symmetric multiprocessor system. As discussed above, real-time systems running safety-critical applications need to ensure functional correctness in the presence of permanent/transient faults. Hence, the second

phase endeavors towards the development of an efficient scheduling strategy with support for recovery from permanent faults, in a real-time multiprocessor system. The first two phases discuss scheduling mechanisms for homogeneous multiprocessor systems. However, today's platforms are increasingly becoming heterogeneous to cater to the stringent and customized performance demands of different applications. Scheduler design for heterogeneous systems is challenging because the same application may exhibit different timing as well as power characteristics on the various processing elements. Hence, the third phase combines the objectives of the first two phases (minimizing energy consumption and providing fault tolerance) and targets to develop a standby-sparing based combined scheduling strategy for imbibing both energy-awareness and fault tolerance in heterogeneous real-time multi-core systems. The second and third phases assume that faults are always detectable and aim towards the design of efficient procedures that provide functional correctness in the presence of faults. In the fourth and last phase of this dissertation, we have endeavored towards the design of a low-overhead formal fault diagnosis mechanism which actively monitors the system and detects the presence of unobservable faults.

All the presented works have been validated through extensive simulation based experiments using synthetically generated workloads as well as real world benchmarks. The obtained results have demonstrated the versatility and efficacy of the proposed approaches.

Declaration

I certify that:

a. The work contained in this thesis is original and has been done by me under the guidance of my supervisors.

b. The work has not been submitted to any other Institute for any degree or diploma.

c. I have followed the guidelines provided by the Institute in preparing the thesis.

d. I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

e. Whenever I have used materials (data, theoretical analysis, figures, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references. Further, I have taken permission from the copyright owners of the sources, whenever necessary.

Piyoosh P

Copyright

Certificate

This is to certify that this thesis entitled, **"Energy-aware and Fault-tolerant Design Strategies for Safety-critical Systems"**, being submitted by **Piyoosh P**, to the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, for partial fulfillment of the award of the degree of Doctor of Philosophy, is a bonafide work carried out by him under our supervision and guidance. The thesis, in our opinion, is worthy of consideration for award of the degree of Doctor of Philosophy in accordance with the regulation of the institute. To the best of our knowledge, it has not been submitted elsewhere for the award of the degree.

.....

Dr. Arnab Sarkar
Associate Professor
Department of Computer Science and Engineering
IIT Guwahati

.....

Dr. Santosh Biswas
Associate Professor
Department of Computer Science and Engineering
IIT Guwahati

Dedicated to

Achan, Amma, Anju, Kuttettan, Vrinda, my teachers

Whose blessing, love and inspiration paved my path of success

Acknowledgments

I wish to express my deepest gratitude to my supervisors, Dr. Arnab Sarkar and Dr. Santosh Biswas for their valuable guidance, inspiration, and advice. I feel very privileged to have had the opportunity to learn from, and work with them. Their constant guidance and support not only paved the way for my development as a research scientist but also changed my personality, ability, and nature in many ways. I have been fortunate to have such advisors who gave me the freedom to explore on my own and at the same time the guidance to recover when my steps faltered. Besides my advisors, I would like to thank the rest of my thesis committee members: Prof. P. Bhaduri, Prof. J. K. Deka, and Prof. H. K. Kapoor, for their insightful comments and encouragement. Their comments and suggestions helped me to widen my research from various perspectives.

I would like to express my heartful gratitude to the director, the deans and other managements of IIT Guwahati whose collective efforts has made this institute a place for world-class studies and education. I am thankful to all faculty and staff of Dept. of Computer Science and Engineering for extending their co-operation in terms of technical and official support for the successful completion of my research work. I thank TCS (TATA Consultancy Services) for providing Ph.D. research fellowship for the period July 2015 to June 2019.

I am thankful to my friends Rajesh, Sanjit, Satish, Sangeet, Sanjay, Vasudevan, Achyutmani, Rakesh, Basina, Sharmaji, Mayank, Harsha, Hareesh and Rohit Verma (Wg Cmdr) for supporting and motivating to overcome any problems either in work and otherwise. The countless discussions, sharing ideas has improved our research. I am also grateful to all my seniors, friends and juniors especially Jiss bhai, Vivekettan, Haris bhai, Sandeep bhai, Sonu bhai, Sajith bhai, Anoop, Arun Mathew, Thomas, Dileepettan, Vishnu, Vijith, Mathew, Vivek, Sibaji, Anirban, Rana, Satisha, Vidyapu, Nayantara,

Dipika, Aswathy, Sivakumar, Manju miss, Cherinet, Rakesh Tripathi, Biswajit, Shounak, Sukarn and many others for their unconditional help and support. You made my life at IIT Guwahati a memorable experience.

Most importantly, none of this would have been possible without the love and patience of my family. I want to thank parents, Anju, Vrinda, kuttettan, Ettathi, Ambiliamma, Achan, Nandhu and Arju for being a constant source of love, concern, support, and strength all these years.

Contents

List of Figures

List of Algorithms

List of Tables

List of Acronyms

ARM *Advanced RISC Machines*

CMOS *Complementary Metal-Oxide Semiconductor*

DAG *Directed Acyclic Graph*

DES *Discrete Event System*

DPM *Dynamic Power Management*

DTM *Dynamic Thermal Management*

DVS *Dynamic Voltage Scaling*

ECU *Electronic Control Unit*

EDF *Earliest Deadline First*

EFI *Electronic Fuel Injection*

EPSP *Earliest Potential Suspension Point*

EPWP *Earliest Potential Wakeup Point*

ESSM *ERfair Scheduler with Suspension on Multiprocessors*

FT-FS *Fault Tolerant Fair Scheduler*

HVAC *Heating, Ventilation and Air Conditioning*

MLAD *Measurement Limitation based Abstract DES Diagnosis*

PDAs *Personal Digital Assistants*

RM *Rate Monotonic*

TDP *Thermal Design Power*

WCET *Worst Case Execution Time*

List of Symbols

T_i	i^{th} task
T	$T = \{T_1, T_2, ..., T_n\}$; set of n tasks
e_i	Execution time of task T_i
p_i	Period of task T_i
s_i	Start/arrival time of task T_i
u_i	Utilization/weight of task T_i
eu_i	Effective utilization/weight of task T_i
cr_i	Criticality level of task T_i
TI_{be}	Break-even time of a processor
SR_i	Slack generation rate of task T_i
V_j	j^{th} processor/processing core
V	$V = \{V_1, V_2, ..., V_m\}$; set of m processors/processing cores
t_p	Periodic safety checkpoint interval
t_r	Recovery interval
F_{OT}	Fault occurrence time

F_{DT}	Fault detection time
F_{RT}	Fault recovery time
ts_l	l^{th} time slice
tsl_l	Length of l^{th} time slice
sh_i^l	Share of a task T_i in l^{th} time slice
$A1$	Set of *needy* tasks
$A2$	Set of *affluent* tasks
$A3$	Set of *balanced* tasks
O_i	Overallocation rate of an *affluent* task T_i
U_i	Underallocation rate of a *needy* task T_i
e_i^{LP}	Worst case execution time of T_i on LP core
e_i^{HP}	Worst case execution time of T_i on HP core
f_{max}^{LP}	Maximum processing frequency on LP core
f_{max}^{HP}	Maximum processing frequency on HP core
pr_i	Primary copy of T_i
bk_i	Backup copy of T_i
Pr_List	Sorted list of primary copies of tasks
Bk_List	Sorted list of backup copies of tasks
F_i	Failure event of type i

Chapter 1

Introduction

With the growth in technology and larger scales of production, intelligent automation systems have found widespread usage in safety-critical applications across all domains of engineering, ranging from avionics and automobiles to industrial processes, manufacturing, and electronic systems. A system or an application is said to be *safety-critical* if its failure can result in serious injury, loss of life or property, or damage to the environment [4, 10, 61, 63]. Many safety-critical systems also have constraints on time and hence, such class of systems is categorized into *real-time safety-critical* systems. These systems are characterized by their ability to respond events that may happen in their operating environment within stipulated temporal constraints. Thus, the correctness of these systems depends not only on the value of the computation but also on the time at which the results are produced [26, 76]. The time instant by which a valid result should be produced is called *deadline*. Examples of real-time safety-critical systems include fly-by-wire in aircrafts, pacemakers in health-care, anti-lock braking systems in automobiles, reactors in nuclear plants, etc. *This dissertation deals with the design of safety-critical systems in general and real-time safety-critical systems in particular.*

Nowadays, modern computing platforms are being manufactured using complex microarchitectural designs with multi-million gates per chip having small feature sizes. Although such intricate designs allow most of today's embedded systems to meet the computation and performance demands of safety-critical applications (often termed as *tasks*), they come with associated side-effects such as increase in the probability of

faults and high energy dissipation. More and more of these embedded systems are being deployed in harsh environments such as outer space, which also adversely affect fault-rates in these systems. In addition, there is an increasing trend in many futuristic safety-critical systems to be shipped with quantifiable measures of system reliability. Therefore, the ability to maintain functional and temporal correctness in the presence of *faults*[1] (*permanent/transient*) is a key design requirement in such safety-critical systems. Increased reliability/tolerance against faults are typically achieved through time and/or hardware redundancies [37,62,64,102]. Time redundancy based approaches use the slack capacity available in an underloaded system to achieve fault tolerance [64]. Important time redundancy based strategies include re-execution [94, 95] and checkpointing with rollback recovery [40, 57]. In the re-execution scheme, whenever a fault is detected, the faulty task is either re-executed from the beginning or a different version of the task, called recovery block, is executed in order to recover from the fault. Unlike the re-execution based approach, checkpointing involves periodically saving the intermediate states of a task during its execution. On the occurrence of a failure, the latest saved internal state of the task is restored and execution resumes from this saved state. Hardware redundancy based approaches incorporate extra hardware into the design to either detect or override the effects of failed components [62]. Important hardware redundancy based strategies include *N-modular redundancy* and the use of *standby spares* [37, 51]. In N-modular redundancy, multiple units running in parallel execute redundant copies of the same workload and mask errors by voting on their outputs. In the standby spares approach (also called *backup redundancy*), the fault affected primary unit is replaced by an identical secondary unit subsequent to a fault. *In this dissertation, we endeavor to develop efficient fault-tolerant design strategies for safety-critical systems.*

With the exponential increase in computational demands over the years, system designers are forced to continuously increase operating frequencies of the involved computation platform. As a result, the processing elements of the computing platform have to often operate at very high frequencies to meet the computational demands of

[1]For simplicity, and with a slight abuse of terminology, the terms fault and failure have henceforth been used interchangeably.

applications, and this causes a higher power consumption in the system. This power dissipation, which is caused due to dynamic switching activities of the processors when executing applications, is termed as *dynamic power consumption*. At the system level, such dynamic power dissipation may be controlled by lowering the processor's operating frequency while taking care that the performance demands of the workloads in the system can still be satisfied. The operating frequencies of a *Complementary Metal-Oxide Semiconductor* (CMOS) based processor are typically a linear function of the supply voltage (V_{dd}) and hence can be varied by scaling V_{dd}. Further, it has been shown that energy dissipated per cycle in CMOS circuits scale quadratically to the supply voltage [89, 108]. Hence, careful management of processor operating frequencies through *Dynamic Voltage Scaling* (DVS) has the potential to provide large energy savings. Further, most of today's high-end devices are being manufactured using deep sub-micron technologies with multi-million gates per chip. This has caused leakage power dissipation (also termed as *static power dissipation*) through the gates in these chips, which is an equally important source of energy wastage as its dynamic counterpart. This static power consumption is independent on the operating frequencies and switching activities of the processors and is always present whenever the system is powered-on. Figure 1.1 gives a detailed idea about the power dissipation in CMOS circuits over the years with a decrease in the gate length [15]. It has been observed that after the year 2005 onwards, static power consumption has become comparable with its dynamic counterpart and has even become the predominant component. Static power dissipation can be reduced by suspending parts of the system such as one or more processors, cache, RAM, etc., when the utilization of system's full capacity is not needed to meet performance requirements. Energy dissipation of the suspended components are typically negligible compared to their active states [13, 14, 18]. In addition, a significant class of contemporary real-time embedded systems is driven by limited energy sources like batteries. Therefore, it is essential to provide energy efficient computing mechanisms to reduce power consumption in these systems. *In this dissertation, we also endeavor to develop efficient energy-aware design strategies for safety-critical systems.*

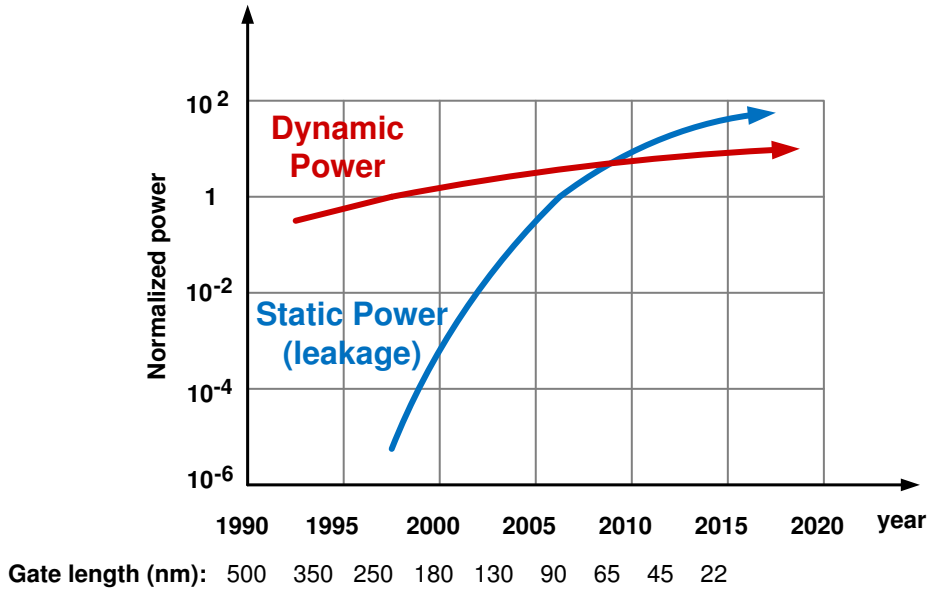


Figure 1.1: *Increase of static and dynamic power in CMOS circuits [15]*

A large class of safety-critical systems such as pacemakers in health-care, satellites in outer space, etc. are often also energy-constrained. Hence, mechanism for both fault tolerance and energy-awareness must be incorporated in their design process. A closer observation reveals that both for mechanisms which aim to provide energy efficiency as well as those targeted to achieve fault tolerance, essentially endeavor to meet their goals by exploiting available redundant system resources in terms of time (temporal redundancy) or hardware (spatial redundancy) or both. Thus, the goals of energy efficiency and that of fault tolerance may be considered to be conflicting in nature. Hence, simultaneously satisfying both these goals will necessitate a careful trade-off which judiciously balances the fraction of the available redundant resources dedicated to achieve energy efficiency with the fraction dedicated to achieve fault tolerance [84, 118]. *This dissertation also deals with the development of combined energy-aware and fault-tolerant design strategies for safety-critical systems.*

The continuous endeavor towards higher performance along with the necessity to meet more and more finely-tuned application specific performance objectives is fast transforming computation platforms from homogeneous to heterogeneous multi-cores.

Homogeneous processing platforms use identical computing elements to satisfy the computational demands of various applications. On the contrary, processing platforms with varying types of computing elements are called heterogeneous (or unrelated) platforms. For example, *Advanced RISC Machines* (ARM) has developed a heterogeneous processing architecture, called ARM big.LITTLE which has been deployed in cutting-edge mobile devices such as Samsung Galaxy Note 4, S10, etc. The big.LITTLE platform contains two types of cores, one of which is high-performance, called the big cores, while the other is of lower performance and power-efficient, referred to as LITTLE cores. Due to the differences in the internal microarchitectures of big and LITTLE cores, the same task may exhibit different execution rates as well as power consumption characteristics on the different cores [100, 101]. *In this dissertation, we endeavor to develop design strategies for heterogeneous computing systems as well, along with our designs for homogeneous platforms.*

The design strategies for fault tolerance discussed above assume faults to be detectable. However, enforcement of fault tolerance can only be achieved through the incorporation of safe design methodologies which enable efficient active monitoring and detection of unobservable faults in the system. Therefore, it is desirable to incorporate efficient strategies for fault diagnosis (detection and isolation) in the construction of safety-critical systems. It may be noted from the literature that formally constructed safe design methodologies are often used to make a system amenable to efficient active monitoring and detection of unsafe execution states whenever the system behavior deviates from its stipulated specification. *Discrete Event System* (DES) is an important formal design technique that is often used for automated failure diagnosis in a wide range of systems primarily because of its systematic modeling approach and the simplicity of its associated algorithms [28, 103, 115]. DES allows a structured, hierarchical modeling procedure to generate composite models of complex systems from individual component models and then allows the incorporation of diagnostics over these composite models. Through this approach, DES methods are able to avoid the often tedious and involved efforts that are required to construct detailed one-shot monolithic models of the

complex system to be diagnosed. Further, complex systems which even include continuous dynamics can also be viewed as DESs at a certain level of abstract discretization. DES provides a fault detection mechanism known as *Diagnoser* which actively monitors behavior of the system and detects the occurrence of unexpected events (faults) in an effective way. It may be noted from the literature that complexity of the diagnosis processes, that is, constructing a diagnoser and testing its diagnosability, is exponential in the number of system states [28, 103, 115]. This may lead to prohibitively huge state-space requirements in the design of diagnosers for large and complex systems. *As a spin-off from our efforts related to energy-aware and fault-tolerant scheduling, we have also endeavored to develop an efficient, low-overhead, DES-based fault diagnosis design strategy for safety-critical systems, as part of this dissertation.*

1.1 Challenges

During the development of efficient design strategies for safety-critical systems, a designer must encounter several challenges. This is because the design may need to satisfy various stringent constraints including those related to timeliness, resource utilization, fault tolerance, power dissipation, cost, etc. We now enumerate a few such important challenges and discuss them.

1. **Timeliness:**

   Real-time systems are characterized by their ability to generate results whose correctness depends not only on the value of the computation but also on the time domain. The time before which a task should complete its execution without causing any damage to the system is called its *deadline*. Therefore, a real-time safety-critical system must be able to guarantee all timing constraints (execution times and deadlines) of various applications/tasks that co-execute in the system.

2. **Resource constraints:**

   Over the years, the nature of computing platforms used in real-time systems has seen a distinct transformation from uni-cores to homogeneous multi-cores to heterogeneous multi-core systems. These computing platforms typically consist of

a limited number of processing elements (i.e., resources). Therefore, the design strategies for safety-critical systems must be able to effectively utilize the processing capacity of underlying platform to satisfy the computational demands of real-time applications.

3. **Fault tolerance:**
   As the real-time systems become more and more complex, the need to actively monitor the system and provide safety guarantee has become critical. In general, safety-critical systems must adhere to strict specifications on the operation of its critical components. However, these specifications may be violated by faults caused by environmental disturbances leading to a failure. Therefore, in addition to satisfying timing constraints, fault tolerance (that is, ensuring functional correctness in the presence of faults) is fast emerging as a design constraint of paramount importance in safety-critical systems.

4. **Energy efficiency:**
   Modern computing platforms involve complex microarchitectural designs with multi-million gates per chip and small feature sizes to meet the computation and performance demands of safety-critical applications. Consequently, these platforms consume high energy during run-time. When the computing capacities of such platforms are not fully utilized, it is possible to employ efficient reconfiguration strategies to control and lower their energy consumption based on the computation demands of the applications running on them.

5. **Simultaneous handling of multiple conflicting challenges:**
   A closer observation reveals that both for mechanisms which aim to provide energy efficiency as well as those targeted to achieve fault tolerance, essentially endeavor to meet their goals by exploiting available redundant system resources in terms of time (temporal redundancy) or hardware (spatial redundancy) or both. Thus, the goals of energy efficiency and that of fault tolerance may be considered to be conflicting in nature. Hence, simultaneously satisfying both these goals will necessitate a

careful trade-off. In safety-critical systems, if all timing and fault tolerance related constraints are satisfied, then the system designer can focus on minimizing the overall energy consumption to prolong the battery lifetime of systems.

1.2 Objectives

The principle aim of this dissertation has been to investigate the theoretical and practical aspects of energy-efficient and fault-tolerant design strategies for safety-critical systems keeping in view the challenges/hurdles discussed in the previous section. In particular, the objectives of this work may be summarized as follows:

1. Development of an efficient *energy-aware* scheduling methodology to minimize static energy consumption in a symmetric real-time multiprocessor system.

2. Design and implementation of an efficient *fault recovery* mechanism to handle the transient overloads caused by the permanent failure of a processor in a real-time multiprocessor system.

3. The third objective combines the first two objectives (minimizing energy consumption and providing fault tolerance) and develops a standby-sparing based *energy-aware fault-tolerant* scheduling strategy for heterogeneous real-time multicore systems.

4. Development of a formally constructed light-weight *fault diagnosis* mechanism to actively monitor the system and detect the presence of unobservable failures in a safety-critical system.

1.3 Contributions

As a part of the research work, we have developed the following energy-efficient and fault-tolerant design strategies for safety-critical systems:

1. **Energy-efficient Fair Scheduling on Real-time Multiprocessor Systems** Proportional fair schedulers with their ability to provide optimal schedulability

along with hard timeliness and QoS guarantees on multiprocessors, form an attractive alternative in real-time embedded systems that concurrently run a mix of independent applications with varying timeliness constraints. In this work, we propose an efficient, optimal proportional fair scheduler called *ERfair Scheduler with Suspension on Multiprocessors* (ESSM) which attempts to reduce system wide energy consumption by locally maximizing the processor suspension intervals while not sacrificing the ERfairness timing constraints of the system. The proposed technique takes advantage of higher execution rates of tasks in underloaded ERfair systems and uses a novel procrastination scheme to search for time points within the schedule where suspension intervals are locally maximal. ESSM not only ensures 100% resource utilization but also guarantees that fairness accuracy of no task will ever be violated due to the procrastination applied. To the best of our knowledge, this is the first work targeted at minimizing static energy dissipation in proportionally fair scheduled real-time multiprocessor systems.

2. **Fault-tolerant Fair Scheduling on Real-time Multiprocessor Systems**
   The ability to maintain functional and temporal correctness in the presence of faults is a key requirement in many safety-critical embedded systems. This work proposes an efficient fault recovery mechanism for real-time multiprocessor systems scheduled using a low overhead and semi-partitioned optimal proportional fair scheduling technique. We assume a system that can handle a single permanent processor fault at any time, using cold back-ups (with pre-specified activation/recovery time subsequent to the detection of a fault). As a result of the fault, the system may suffer transient overloads during such recovery periods, potentially leading to unacceptable fairness deviations and consequent rejections/early terminations of critical jobs. The proposed fault-tolerant scheduler, called *Fault Tolerant Fair Scheduler* (FT-FS), attempts to minimize such job terminations/rejections during recovery, by judiciously redistributing slacks accumulated by a subset of jobs, delivering more sustainable performance in the process.

   FT-FS employs a novel slack donation scheme from overallocated to underallo-

cated tasks with the objective of maximizing resource utilization. In times of uncontrollable overloads during recovery when task rejection becomes inevitable, the proposed framework takes care to reject a minimum number of the least critical tasks. The scheduler also ensures that all tasks which execute through the recovery period have progressed by their prescribed amounts at the end of the recovery period when the spare processor gets activated. This empowers the framework to handle fresh faults immediately after recovery. Hence in this work, two consecutive processor faults only need to be separated by just the recovery interval, and this duration is typically small especially in closely coupled systems. The underlying scheduling structure being based on DP-Fair, FT-FS is able to ensure high resource utilization and fair rate-based execution progress while incurring low scheduling related overheads through controlled migrations and context-switches.

3. **Fault and Energy Aware Scheduling on Real-time Heterogeneous Dual-cores**

   Devising scheduling strategies for modern safety-critical real-time systems implemented on heterogeneous platforms is a challenging as well as a computationally demanding problem. As a consequence, today we face a scarcity of low-overhead scheduling techniques which are applicable to heterogeneous platforms. In this work, we attempt to develop an efficient *energy-aware* heuristic scheme called, *FENA-SCHED*, for the *fault-tolerant* scheduling of real-time tasks on two-core heterogeneous platforms, where one core is high-performance and the other core is power-efficient. In order to provide fault-tolerance against transient processor faults, we consider a standby-sparing approach where the power-efficient core is used to execute primary task versions while the high-performance core is operated as a spare to re-execute fault affected tasks (i.e., backups). Since, the execution of backups scheduled on the spare core are cancelled subsequent to the fault-free execution of their primaries, we employ *Dynamic Power Management* (DPM) on both cores to minimize energy consumption. For a DPM-enabled system, we found that designating power-efficient (modest performance) core as primary and power-

hungry (high-performance) core as spare yields better energy savings as compared to its counterpart. Further, FENA-SCHED utilizes *backup-backup overloading* to minimize energy consumption while guaranteeing tolerance against a given number of transient processor faults.

4. **A Formal Design Strategy for Fault Diagnosis in Safety-critical Systems** Unlike the second and third contributions which assume faults are always detectable and target towards the design of efficient scheduling procedures that provide functional correctness in the presence of faults, in this work, we have developed an efficient, low-overhead, DES-based fault diagnosis design strategy for safety-critical systems.

   In this work, we present a new fault diagnosis approach called *Measurement Limitation based Abstract DES Diagnosis* (MLAD), which attempts to reduce state space complexity of the diagnosis process while simultaneously preserving full diagnosability. The MLAD approach carefully applies a set of distinct measurement limitation operations on the state variables of the original DES model based on fault compartmentalization to obtain separate behaviorally abstracted DES models and corresponding abstract diagnosers with far lower state spaces. The set of measurement limitation operations are so designed that although, any single abstract diagnoser may compromise diagnosability in seclusion, the additive combination of all diagnosers running in parallel always ensures complete diagnosability. Effective measurement limitation also ensures that the combined state space of the abstract diagnosers is much lower than that of the single full diagnoser that may be derived from the original DES model.

1.4 Organization of the Thesis

The thesis is organized into seven chapters. Each contributory chapter (Chapter 3 to Chapter 6) starts with a detailed description of the system under consideration, followed by extensive discussions about the proposed scheme. Then, we discuss the experimental evaluation of the proposed methodology. All chapters conclude with a summary of

contributions. The last chapter of the thesis (Chapter 7) discusses the conclusions and future perspectives of this research work. A summary of the contents in each chapter is as follows:

- **Chapter 2**: *Energy/Fault Aware Real-time and Safety-critical Systems - Background and State-of-the-art*

  This chapter first provides an overview on the structure of real-time systems, followed by a review of various scheduling strategies under four important design considerations: i) real-time resource allocation on multiprocessor platforms, ii) energy awareness, iii) imbibing fault tolerance, and iv) fault tolerance with energy awareness. Most of the fault-tolerant procedures that provide functional correctness in the presence of faults, assume faults to be detectable. However, enforcement of such fault tolerance can only be achieved through the incorporation of safe design methodologies which enable efficient active monitoring and detection of unobservable faults in the system. Therefore, it is desirable to incorporate efficient fault diagnosis (detection and isolation) strategies in the construction of safety-critical systems. In this chapter, we also discuss an important formally constructed safe design technique known as Discrete Event System (DES), and introduce the formal definitions related to DES-based fault diagnosis framework. Then, we analyze a few important state-of-the-art works related to DES-based fault diagnosis.

- **Chapter 3**: *Energy-efficient Fair Scheduling on Real-time Multiprocessor Systems*

  In this chapter, we present a novel energy-efficient design strategy which minimizes static energy consumption in a homogeneous real-time multiprocessor system. Being based on a work-conserving proportional fair scheduling mechanism, in this chapter, we develop an energy-aware design strategy which attempts to locally maximize the total length of suspension intervals while simultaneously reducing the number of such intervals using a novel procrastination scheme, thus lowering energy consumption in the process.

- **Chapter 4**: *Fault-tolerant Fair Scheduling on Real-time Multiprocessor Systems*

This chapter proposes a novel time-cum-hardware redundancy based fault-tolerant design strategy for real-time multiprocessor systems containing cold-standby spares. In this work, we assume a system that can handle at most one permanent processor fault at any given time using a single cold-standby spare that takes a fixed recovery time to attain operational state after the detection of a fault. Thus, during any recovery period subsequent to a failure, the system is forced to work with one less processor resource. The primary objective of the work presented in this chapter is to satisfy all timing constraints of tasks during any recovery interval, given the total workload to be handled.

- **Chapter 5**: *Fault and Energy Aware Scheduling on Real-time Heterogeneous Dual-cores*

  Research conducted in this chapter deals with the development of a combined energy-aware and fault-tolerant design strategy for heterogeneous systems. This work first develops a standby-sparing based fault-tolerant scheme for scheduling real-time applications on heterogeneous dual-core systems consisting of a power-efficient core and a high-performance core. By utilizing dynamic power management (DPM) technique with backup-backup overloading, the proposed fault-tolerant framework is further extended to minimize overall energy consumption in the system.

- **Chapter 6**: *A Formal Design Strategy for Fault Diagnosis in Safety-critical Systems*

  This chapter proposes a formal fault detection and isolation (that is, fault diagnosis) framework for the design of safety-critical systems. The objective of this novel fault diagnosis strategy is to reduce state space complexity involved in the diagnosis process while simultaneously preserving full diagnosability.

- **Chapter 7**: *Conclusions and Future Perspectives*

  The thesis concludes with this chapter. We discuss the possible extensions and future works that can be done in this area.

Chapter 2

Energy/Fault Aware Real-time and Safety-critical Systems - Background and State-of-the-art

This dissertation is oriented towards the development of efficient design strategies for real-time and safety-critical systems which aim to achieve one of the following objectives: 1) energy minimization, 2) fault tolerance, 3) fault tolerance with energy awareness, and 4) low-overhead fault detection. The previous chapter provided an overview of the challenges imposed by the diversity in the nature of computing platforms, performance requirements including those related to timeliness, fault tolerance, power dissipation, etc., towards the development of modern safety-critical systems.

In this chapter, we present a brief background as well as state-of-the-art related to different types of real-time systems, followed by a survey on discrete event system (DES) based fault diagnosis framework. We first provide an overview on the structure of real-time systems. Then, we discuss various scheduling strategies under four important design considerations: i) real-time resource allocation on multiprocessor platforms, ii) energy awareness, iii) imbibing fault tolerance, and iv) fault tolerance with energy awareness. It may be observed from the literature that most of the fault-tolerant procedures that provide functional correctness in the presence of faults, assume faults to be detectable. However, enforcement of such fault tolerance can only be achieved through the incorporation of safe design methodologies which enable efficient active monitoring and detection of unobservable faults in the system. Therefore, it is desirable to incor-

porate efficient fault diagnosis (detection and isolation) strategies in the construction of safety-critical systems. DES is an important formally constructed safe design technique that is often used for automated failure diagnosis in a wide range of systems primarily because of its systematic modeling approach and the simplicity of its associated algorithms [28, 103, 115]. In this chapter, we also introduce the formal definitions related to DES-based fault diagnosis framework. Then, we discuss and analyze a few important state-of-the-art works related to DES-based fault diagnosis.

2.1 An Overview of Real-time Systems

Typically, real-time systems are composed of the following layers [90]:

- **An application layer**, which is composed of a set of all applications that requires execution in the system.
- **A real-time scheduler**, which takes the scheduling decisions and provides services to the application layer.
- **A hardware platform**, which includes the processors (among other things such as memories, communication networks, etc.).

We will now present each of these layers in detail and introduce the theoretical models enabling researchers to analyze these systems and design efficient schedulers for real-time systems to allocate the application tasks on the hardware platform.

2.1.1 The Application Layer

The application layer contains a set of applications that the system needs to execute. In real-time systems, the applications are often composed of a set of *recurrent* tasks. Each such task may represent a piece of code (i.e., program) which is triggered by external events that may happen in their operating environment. Each execution of the task is referred to as a *task instance* or *job*. We now discuss a set of definitions related to real-time tasks.

2.1.1.1 A Real-time Task Model

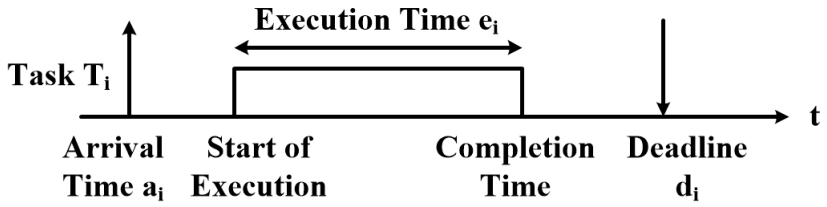


Figure 2.1: *Temporal characteristics of a real-time task T_i*

Formally, a real-time task (denoted by T_i; shown in Figure 2.1) can be characterized by the following parameters:

1. **Arrival time** (a_i) is the time at which a task becomes ready for execution. It is also referred as *release time* or *request time* of the task.

2. **Start time** (s_i) is the time at which a task starts its execution.

3. **Execution time** (e_i) is the time required by the processor to finish the computational demand of a task without interruption.

4. **Finishing time** or *Completion time* is the time at which a task finishes its execution.

5. **Deadline** is the time before which a task is required to meet its execution requirement. If deadline is computed with respect to the system start time (at 0), it is referred to as *absolute deadline* (D_i). If it is computed with respect to its arrival time, it is referred to as *relative deadline* (d_i).

6. **Slack time** or *Laxity* is the maximum amount of time by which execution of a task can be delayed after its activation to complete within its deadline: $d_i - e_i$.

7. **Priority** is the importance given to a task in context of the schedule at hand.

A real-time task T_i can be classified as periodic, aperiodic and sporadic based on regularity of its activation [26].

1. **Periodic** tasks consist of an infinite sequence of identical activities, called *instances* or *jobs*, that are regularly activated at a constant rate. The activation time of the first periodic instance is called *phase* (ϕ_i). The activation time of the k^{th} instance is given by $\phi_i + (k-1)p_i$, where p_i is the activation period (*fixed inter-arrival time*) of the task.

2. **Aperiodic** tasks also consist of an infinite sequence of identical jobs. However, their activations are not regularly interleaved.

3. **Sporadic** tasks consist of an infinite sequence of identical jobs with consecutive jobs separated by a *minimum inter-arrival time*.

Following are the three levels of constraint related to the deadline of a task:

1. *Implicit Deadline*: All task deadlines are equal to their periods ($d_i = p_i$).

2. *Constrained Deadline*: All task deadlines are less than or equal to their periods ($d_i \leq p_i$).

3. *Arbitrary Deadline*: All task deadlines may be less than, equal to, or greater than their periods.

We now provide a few other definitions related to tasks and task set.

- *Utilization*: The utilization of a (implicit deadline) task T_i is given by $u_i = e_i/p_i$. In case of constrained deadline, $u_i = e_i/d_i$.

- *Hyperperiod*: It is the minimum interval of time after which the schedule repeats itself. For a set of periodic tasks (with periods $p_1, p_2, \ldots, p_n$) activated simultaneously at $t = 0$, the hyperperiod is given by the least common multiple of the periods.

- *Static and Dynamic Task System*: In a static task system, the set of tasks that is executed on the platform is completely defined before start running the application. In a dynamic task system, some tasks may experience a modification of their properties while other tasks leave or join the executed task set at run-time.

2.1.2 A Real-time Scheduler

A real-time scheduler acts as an interface between applications and hardware platform. It configures and manages the hardware platform (e.g., manage hardware interrupts, hardware timers, etc.). More importantly, it schedules the tasks using a real-time scheduling algorithm. The set of rules that, at any time, determines the order in which tasks are executed is called a *scheduling algorithm*.

Given a set of tasks, $T = \{T_1, T_2, ..., T_n\}$, a *schedule* is an assignment of tasks to available processors, so that each task is executed until completion. A schedule is said to be *feasible* if all tasks can be completed according to a set of specified constraints. A set of tasks is said to be *schedulable* if there exists at least one algorithm that can produce a feasible schedule. A scheduling algorithm is said to be *optimal* if it is able to find a feasible schedule, if one exits. An algorithm is said to be heuristic if it is guided by a heuristic function in taking its scheduling decisions. A *heuristic* algorithm tends toward the optimal schedule but does not guarantee finding it. In *work-conserving* scheduling algorithm, processor is never kept idle while there exist a task waiting for execution on the processor.

2.1.3 Processing Platform

The term *processor* refers to a hardware element in the platform which is able to process the execution of a task.

1. **Uniprocessor** system can only execute one task at a time and must switch between tasks.

2. **Multiprocessor** system will range from several separate uniprocessors tightly coupled using high speed network to multi-core. It can be classified as follows:

   (a) **Homogeneous:** The processors are *identical*, i.e., they all have the same functional units, instruction set architecture, cache sizes and hardware services. The rate of execution of every task is same on all processors. Hence,

the worst-case execution time of a task is not impacted by the particular processor on which it is being executed.

(b) **Uniform:** The processors are *identical* - but they are running at different frequencies. Hence, all processors can execute all tasks but the speed at which they are executed and their worst-case execution time vary in function of the processor on which they are executing.

(c) **Heterogeneous:** The processors are different, i.e., processors may have different configurations, frequencies, cache sizes or instruction sets. Some tasks may therefore not be able to execute on some processors in the platform, while their execution speeds (and their worst-case execution times) may differ on the other processors.

2.1.4 Multi-criticality Systems

Many real-time systems support the execution of applications with different relative importance values (some times called *criticality levels*) on a common platform [83]. For example, in modern avionics systems, flight control tasks (responsible for the vehicle's safety) are considered to be more safety-critical than military mission tasks (responsible for say, firing a missile at the enemy target). Scheduling decisions in these scenarios must consider the criticality levels of applications. Particularly, in times of overload the objective is usually to allow the execution of the highest critical tasks in the system while rejecting the least critical ones, such that the overload condition can be mitigated [83,93].

The background on the structure of real-time systems and the evolution of scheduling algorithms for these systems can be found in [15, 26, 96]. In the next section, we discuss various classifications of real-time scheduling approaches.

2.2 A Classification of Real-time Scheduling Approaches

Preemptive Vs. Non-preemptive Scheduling: *Preemptive* schedulers are based on the assumption that the execution of a task may be interrupted and the processor directed to run a different piece of code after the interrupt. The unfinished portion of

the interrupted task thus has to be re-allocated to may be, a different processor [42]. On the contrary, scheduling algorithms following a *non-preemptive* approach must allow a task to execute until completion. As a result, the response time to external events may be quite long if some tasks have a large execution time. However, many task systems are inherently atomic in the sense that task invocations must execute to completion without interruption once started. Preemptive schedulers are unusable for these task systems.

Online Vs. Offline Scheduling: In *offline* scheduling, the scheduler has a priori knowledge of the task set and its constraints, such as arrival times, execution times, precedence constraints, etc. The schedule is generated and stored at design time and dispatched later during runtime of the system. Offline scheduling is also referred to as *static* scheduling [34]. On the other hand, *online* scheduling algorithms make their scheduling decisions at runtime based on the information about the tasks that have arrived so far. Although they are often flexible and adaptive, they may incur significant overheads because of runtime processing. However, they are a must in systems which do not have enough information before run-time to execute the scheduler statically. Online scheduling is also referred to as *dynamic* or *runtime* scheduling.

Clock-Driven Vs. Event-Driven Scheduling: In *clock-driven* schedulers, scheduling decisions are made at specific time instants which are chosen a priori before the system begins its execution [77]. Typically, in a system that uses clock-driven scheduling, all parameters of the job set are fixed and known. It is also called a *time-driven* scheduling approach. A *table-driven* scheduler is an example of clock-driven approach. Here, the schedule is generated and stored in a table off-line. The system timer kicks off execution of a segment of code of a task at each scheduling decision time by referring to the table at run time.

In the *event-driven* approach, scheduling points are defined by events such as job release or completion. Generally, these schedulers assign priorities to each task. At each scheduling instant, the currently highest priority task present in the ready queue gets hold of the resource (Hence, they are also called *priority-driven schedulers*). These

algorithms leave a resource idle only when no job requiring the resource is ready for execution.

The *Rate Monotonic (RM)* [75, 77] and *Earliest Deadline First (EDF)* [75, 77] algorithms (discussed in Section 2.3.1) are examples of event-driven approach. Event-driven schedulers are more proficient than clock-driven schedulers because they can feasibly schedule some task-sets that clock-driven schedulers cannot. These are also more flexible because they can feasibly schedule sporadic and aperiodic tasks in addition to periodic tasks whereas clock-driven schedulers can only handle periodic tasks.

Static Priority Vs. Dynamic Priority Scheduling: The distinction between *static priority* and *dynamic priority* scheduling is based on the priority management policy adopted by a priority-driven scheduler. In the static priority scheme, tasks are assigned an integer priority value that remains fixed for the lifetime of the task. Whenever a task is made ready to run, the active task with the highest priority commences or resumes execution, preempting the currently executing task if need be. Priority values may change at run time in case of dynamic priority schedulers. *Rate Monotonic (RM)* [75, 77] and *Deadline Monotonic (DM)* [3, 77] are examples of Static priority scheduling while *Earliest Deadline First (EDF)* [75, 77] and *Least Slack Time First (LST)* [77] are examples of dynamic priority scheduling.

Partitioning Vs. Global Scheduling: In the context of multi-processor scheduling policies, a *global* scheduler is one which puts all the ready tasks in a single queue and selects the highest priority task at each invocation irrespective of which processor is being scheduled. Thus, a task is allowed to execute on any processor, even when resuming after having been preempted. In a purely *partitioned* approach, on the other hand, the set of tasks is partitioned into as many disjoint subsets as there are processors available, and each such subset is associated with a unique processor [27, 79, 107]. Thus, all instances of a task get executed on the same processor. Between these two extremes of no inter-processor migration and full migration, there is an intermediate class of algorithms (known as *semi-partitioned*) that allow *restricted migration*. For example, different jobs

of the same task may be allowed to execute on different processors. However, a single job may be constrained to execute on a particular processor.

2.3 A Brief Survey of Scheduling Algorithms

In this section, we discuss scheduling strategies under four important design considerations: i) timeliness, ii) energy awareness, iii) imbibing fault tolerance, and iv) simultaneous handling of multiple conflicting challenges such as timeliness, energy awareness and fault tolerance.

2.3.1 Traditional Real-time Scheduling Approaches

As representative examples of traditional real-time scheduling approaches, we have chosen the *Rate Monotonic (RM)* and *Earliest Deadline First (EDF)* algorithms since they have proved to be two of the most widely used techniques over the years and forms the foundation upon which most of the real-time scheduling theories have developed.

Rate Monotonic (RM) Algorithm: The *RM* algorithm [75, 77] is a preemptive, static priority scheduler applicable in a hard real-time environment. It assigns priorities to tasks based on their periods; the shorter the period, the higher the priority. Because the rate of job releases of a task is the inverse of its period, the priority is directly proportional to the task's rate, and hence the name *rate monotonic*.

An example should clarify the strategy. Let us consider three tasks T_1, T_2 and T_3 with computational requirements e_1, e_2, e_3 being 1, 2, 5 and period of execution p_1, p_2, p_3 being 4, 5, 20. Figure 2.2 shows the RM schedule of the system. T_1 having the shortest period has the highest priority and is always executed as soon as its job is released. T_2 has the next highest priority and is executed in the background of T_1. Similarly, T_3 executes in the background of both T_1 and T_2.

An important shortcoming of the RM algorithm (as shown by Liu and Layland in [75]) is that even on uniprocessor systems no more than 69% of the processor may be utilized to ensure scheduling feasibility of a set of tasks under rate-monotonic priority assignment in the worst case (when the number of tasks is large (∞)).

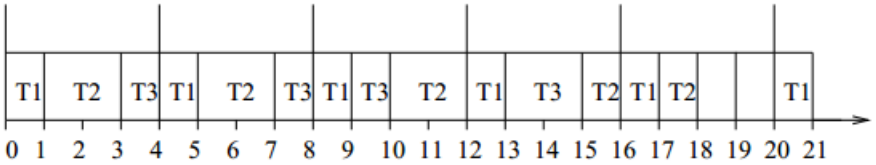


Figure 2.2: *An example RM schedule*

Earliest Deadline First (EDF) Algorithm: *EDF* [75,77] is a preemptive, dynamic priority algorithm. It assigns priorities to individual jobs of the tasks according to their absolute deadlines during run-time. Figure 2.3 shows the *EDF* schedule of two tasks T_1 and T_2 having computational requirements (e_i) 2 and 9 respectively and period (p_i) 5 and 15 respectively. If more than one task have the same absolute deadline, any one of them may be randomly chosen. Figure 2.4 shows the corresponding *RM* schedule.

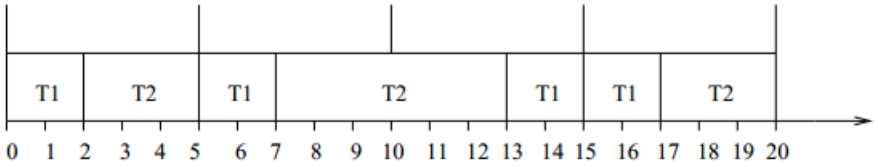


Figure 2.3: *An example EDF schedule*

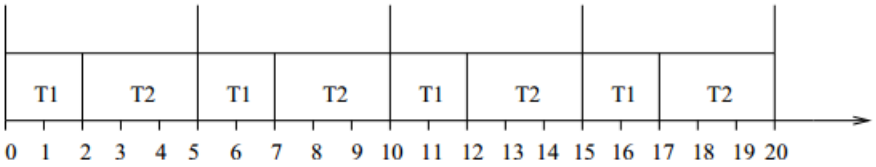


Figure 2.4: *The RM schedule*

A variation of EDF scheduling is the *Least Laxity First (LLF)* [86] scheduler. In LLF, at each scheduling point, a laxity value ($\mathcal{L}$) is computed for each task and the task having the smallest laxity is assigned the highest priority. Here, the laxity value ($\mathcal{L}_i$) for a task T_i is defined as: $\mathcal{L}_i = d_i - (t - re_i)$, where d_i denotes the deadline of T_i, t the current time and re_i the remaining execution requirement of T_i.

Both EDF and LLF are optimal uniprocessor schedulers. That is, if a set of tasks is unschedulable under EDF or LLF, then no other scheduling algorithm will be able to schedule this task-set. The essential difference between EDF and LLF is that by incorporating the remaining execution requirement of a task in its scheduling decision, LLF also takes into consideration the available flexibility for scheduling a task. However, on multi-processor systems both these algorithms (as well as RM) proves to be inefficient in terms of the achievable processor utilization in the worst-case [70].

2.3.2 Real-time Scheduling on Multiprocessor Systems

Traditionally, scheduling of real-time applications (termed as tasks) on multiprocessors make use of either a partitioned or global approach [35] (as discussed in Section 2.2).

2.3.2.1 Partitioned Scheduling Schemes

In a partitioned approach, each task is assigned to a single designated processor on which it executes for its entire lifetime. This approach has the advantage of transforming the multiprocessor scheduling problem to a uniprocessor scheduling one. Hence, well known optimal uniprocessor scheduling approaches such as *Earliest Deadline First* (EDF), *Rate Monotonic* (RM) [26], etc. may be used. In addition, the overheads of inter-processor task migrations and local cache misses is far smaller than global scheduling. Finally, because task-to-processor mapping (which task to schedule on which processor) need not be decided globally at each time-slot, the scheduling overhead associated with a partitioning strategy is lower than that associated with a non-partitioning strategy [8, 9, 27]. However, a major drawback of partitioning is that in the worst case, no more than half the system capacity may be utilized in order to ensure that all timing constraints are met [8].

Optimal assignment of tasks to processors in partitioning is a bin-packing problem which can be stated as follows: given a list L of items of size $\{a_1, a_2, ..., a_n\}$, where $a_i \in (0, 1]$ (a_i represents the weight of task i), the problem of bin packing is to pack these items into a minimal number of unit capacity bins. The problem is known to be NP-hard and several polynomial time heuristics have been proposed to solve it.

The performance of any bin-packing algorithm is evaluated by a measure called *competitive-ratio(R)* which may be defined as follows:

$$R = \lim_{n\to\infty} \sup \frac{A(L)}{OPT(L)},$$

where, L is a list of items $\{a_1, a_2, ..., a_n\}$ of size n, $A(L)$ is the number of bins required by the bin packing algorithm A when list L is used and $OPT(L)$ is the best off-line number of bins required. It is easy to interpret that the use of an infinite sized list in the above measure gives us the worst-case performance ratio. However, there may be many other lists of smaller size that also gives us the worst-case ratio. We consider below some of the well known approaches [79, 107].

Next Fit(NF): This is one of the simplest of the known heuristics. It starts from the first bin and defines it as the active bin. If the next incoming item fits the bin, it places it in that bin. Otherwise, it creates a new bin, makes it the new active bin, and packs the item into this bin. Thus, at any given time, there is only one active bin. The NF algorithm has a competitive-ratio of 2.

First Fit(FF): Given a list of bins, the FF algorithm assigns the next item to the first bin that can accept it.

Best Fit(BF): The BF algorithm assigns the next item to such a processor that can accept the task and will have minimal remaining spare capacity after its addition.

Worst Fit(WF): WF is opposite to BF; it will find a bin which will fit a new item with the largest spare capacity left over. The algorithms FF, BF and WF discussed above have a competitive-ratio of 1.7.

First Fit Decreasing(FFD): *FFD* is the same as FF, but the items are considered in non-increasing order of their sizes. In a similar fashion, *Best Fit Decreasing (BFD)* and *Worst Fit Decreasing (WFD)* can also be defined. All these algorithms FFD, BFD and WFD have a competitive-ratio of 1.22. Although, this competitive-ratio of 1.22 is the

best among all the algorithms, the fundamental requirement of these algorithms, which is non-increasing order of items in list L, may not satisfy the criteria of On-line. Thus, all these algorithms (FFD, BFD and WFD) are generally used as off-line strategies.

2.3.2.2 Global and Semi-partitioned Scheduling Schemes

Unlike partitioning, global and semi-partitioned scheduling schemes allow the migration of tasks from one processor to another during execution. Over the years, a few global optimal schemes such as Pfair, ERfair, etc. and semi-partitioned optimal techniques like DP-Fair, have been proposed. All these scheduling approaches allow the possibility of utilizing the entire capacity of all processors in the system, resulting in high resource utilization. Additionally, they possess many attractive features like flexible resource management, dynamic load distribution, fault resilience, etc. [107].

Most of these global and semi-partitioned scheduling strategies are based on the idea of proportional rate based execution progress for all tasks. Typically, such proportional fairness can be achieved by providing guarantees of the following form for each task: *complete X units of execution for application A out of every Y time units. Proportionate fair* (*Pfair*) scheduling introduced by Baruah et al. [16] is known to be the first optimal global scheduler for real-time repetitive tasks with implicit deadlines, on a multiprocessor system. Later, Anderson et al. [5] presented a work-conserving version of *Pfair*, called *Early-Release fair* (*ERfair*) scheduler, which never allows a processor to be idle in the presence of runnable/ready tasks. Since these global schemes attempt to maintain fair proportional progress for all tasks at all time slots, they may incur unrestricted preemption/migration overheads. More recently DP-Fair [71], an approximate proportional fair scheduler with a more relaxed execution rate constraint, was proposed. DP-Fair is a semi-partitioned scheduling technique which allows restricted preemptions/migrations. Now, we discuss two underlying fair scheduling schemes used in this thesis, in detail.

ERfair Scheduling [5]: Consider a set of periodic tasks $\{T_1, T_2, ..., T_n\}$. A task, say T_i, may arrive at any time within the schedule length, execute for an arbitrary number of instances and then depart. Each instance of T_i has a computation requirement of e_i time

units required to be completed within a period of length p_i time units. ERfair schedulers need to manage their task allocation and preemption in such a way that not only are all task deadlines met, but also each task is executed at a consistent rate proportional to its task weight $\frac{e_i}{p_i}$. Typically, ERfair algorithms consider discrete time lines and divide the tasks into equal-sized subtasks. Subtasks are scheduled appropriately to ensure fairness. The fairness accuracy is generally defined in terms of the *lag* between the amount of time that has been actually allocated to a task and the amount of time that would be allocated to it in an ideal system with a time quantum approaching zero. Formally, the lag of task T_i at time t, denoted $lag(T_i, t)$, is defined as follows:

$$lag(T_i, t) = (e_i/p_i) * t - allocated(T_i, t), \tag{2.1}$$

where $allocated(T_i, t)$ is the amount of processor time allocated to T_i in $[0, t)$. A schedule is $ERfair$ iff:

$$(\forall\ T,\ t\ ::\ lag(T, t) < 1) \tag{2.2}$$

That is, Equation 2.2 infers that the underallocation associated with each task must always be less than one time quantum. A subtask in an ERfair system becomes eligible for execution immediately after its previous subtask completes execution. Obviously, for such a criterion to be guaranteed, we must have

$$\sum_{i=1}^{n} e_i/p_i \leq m \tag{2.3}$$

where, m denotes the number of identical processors in the system. Equation 2.3 infers that the total workload (summation of tasks weights) should be less than or equal to the full system capacity to schedule a set of tasks in the system effectively. Equations 2.1, 2.2 and 2.3 are taken from [5]. In our energy-aware work (Chapter 3), we use ERfair as the underlying scheduling mechanism.

DP-Fair Scheduling [71]: Unlike ERfair, DP-Fair [71] is an approximate proportional fair scheduler with a more relaxed execution rate constraint. It is an optimal algorithm and enables full resource utilization. That is, given n tasks and m processors,

schedulability is ensured provided,

$$\sum_{i=1}^{n} e_i/p_i \leq m \tag{2.4}$$

where, e_i and p_i denote the worst case execution time and period of a task T_i, respectively. In DP-Fair, time is partitioned into slices, demarcated by the deadlines of all jobs in the system. Within a time slice, each task is allocated a workload equal to its proportional fair share and assigned to one or two processors for scheduling. Job subtasks within a slice are typically scheduled using variations of traditional fairness ignorant schemes such as Earliest Deadline First (EDF [76]). Through such a scheduling strategy, DP-Fair is able to deliver optimal resource utilization while enforcing strict proportional fairness (ERfairness) only at period/deadline boundaries. DP-Fair is a semi-partitioned scheduling technique which allows at most $m-1$ task migrations and $n-1$ preemptions within a time slice and thus incurs much lower overheads compared to ERfair. In our fault-tolerant work (Chapter 4), we use a discrete approximation of DP-Fair as the underlying scheduling mechanism.

2.3.3 Energy-aware Real-time Scheduling Strategies

In this section, we first provide an elaborate discussion on the need of energy-aware execution in real-time systems and important techniques for handling energy dissipation in such systems. Subsequently, we discuss a review of various existing energy-aware scheduling strategies on homogeneous multiprocessor systems.

2.3.3.1 Energy-aware Execution: Need and Techniques

Power Consumption of CMOS circuits: Let P denote the total power consumption of a CMOS-based processor/core during active operation while P_{sleep} denote its dissipated power when suspended. The total power P has three major components namely, dynamic power consumption (P_d), static power consumption (P_s) and an inherent power cost to keep the processor on (P_{on}) [55].

$$P = P_d + P_s + P_{on}.$$

The dynamic power consumption (P_d) of CMOS circuits is given by:

$$P_d = C_{eff} \times V_{dd}^2 \times f.$$

where, V_{dd} is the supply voltage, C_{eff} is the average switched capacitance per cycle, and f is the clock frequency. Here, V_{dd} may be considered to be roughly proportional to f. The major components of static current in a standard inverter are reverse bias junction current [111] and subthreshold conduction [111]. Hence, the static power consumption, P_s, is given by:

$$P_s = (V_{dd} \times I_{subn} + |V_{bs}| \times I_j)L_g.$$

where, V_{bs} is the the body bias voltage, I_{subn} is the subthreshold leakage current, I_j is the reverse bias junction current and L_g is the number of devices in the circuit.

Need for Energy-aware Execution: As technology scales to lower feature sizes, leakage or static power consumption/energy dissipation are becoming design parameters of higher criticality. With each technology generation, leakage drain is expected to increase by a factor of more than 5 and has already become the major source of power wastage within a chip [60]. The problem of power wastage in general and leakage drain in particular has been further aggravated by the advent of high-end portable embedded systems such as *Personal Digital Assistants* (PDAs), cell phones, car on-board systems etc., which are powered by limited energy sources like batteries [65]. Hence, techniques for controlling power/energy consumption are being applied at all system levels starting from hardware and firmware to architectural, system and even application level.

Techniques For Reducing Energy Consumption: At the operating system level, two primary mechanisms are generally used to reduce energy consumption: 1) Dynamic Voltage Scaling (DVS) [89, 108] and 2) Dynamic Power Management (DPM) [21, 67]. The first mechanism reduces dynamic energy consumption and involves lowering the processor's operating frequency by appropriately scaling its supply voltage when the full speed is not required. As energy dissipated per cycle in CMOS circuits scale quadrati-

cally to the supply voltage, this strategy is able to provide large energy savings in DVS enabled processors. On the other hand, DPM mechanism tries to minimize static energy dissipation in the system by putting a processor in low-power suspension/sleep mode for as long as possible while still guaranteeing the tasks' timing constraints. However, transition between idle and active states require a fixed amount of time and energy. Hence, a purely greedy policy is often not acceptable because it degrades performance and may not decrease energy consumption. Thus, one of the primary tasks of suspension based algorithms is to predict when the idle period will be long enough so as to compensate the transition cost. There has also been an attempt to maximize the duration of idle intervals by delaying task execution using the procrastination scheduling model [13].

2.3.3.2 A Review of Energy-aware Scheduling on Multiprocessor systems

Energy-aware scheduling algorithms on multiprocessor systems are mainly grouped into two categories: partitioned scheduling [30, 53] and global scheduling [21, 67–69]. While partition oriented scheduling strategies maintain separate local ready queues for tasks in each processor, global scheduling employs only one queue for all tasks assuming a single system-wide priority space. Partitioning is often the favored approach primarily due to its lower overheads and ease of implementation using well known uniprocessor schedulers [76] on individual processors. However, partitioning often suffers from low resource utilization [27]. On the other hand, global scheduling has attractive features such as flexible resource management, dynamic load distribution, fault resilience, high resource utilization etc. [107]. Chen et al. [30] explored the energy-efficient scheduling of periodic real-time tasks on multiprocessor systems with the consideration of leakage current along with DVS. Huang et al. [53] presented a run-time task reallocation scheme that improves the energy efficiency of leakage-aware DVS on multi-core processors. A two phased scheduling heuristic for sporadic tasks on heterogeneous multi-cores was presented by Awan and Petters in [12]. Here, the first phase attempts to minimize dynamic energy dissipation by assigning each task to its favorite processor based on the task's dynamic energy consumption affinity towards different processors. The second phase reduces static energy consumption by trading off higher dynamic energy consumption of

a task to enhance the ability of the processors to use more efficient sleep states. All these approaches follow a partition oriented strategy, and hence, their resource utilizations are often low.

Bhatti et al. [21] presented a DPM strategy for global multiprocessor systems called Assertive Dynamic Power Management (AsDPM). AsDPM first determines the minimum number of active processors needed to fulfill the execution requirement of released jobs at runtime. Then it attempts to cluster the distributed idleness existing on a subset of the active processors into longer continuous idle intervals so that these obtained intervals may be employed to switch some of the processors to deeper low power states for a longer duration of time. This AsDPM strategy is then used along with global schedulers like Global-EDF or Global-LLF to get a better reduction in the energy consumption. As both Global-EDF and Global-LLF are known to be sub-optimal (that is, they cannot fully utilize the complete capacity of the set of processors comprising a multiprocessor system), the global strategy presented in [21] also becomes sub-optimal in nature.

Legout et al. [67] presented an offline power-aware heuristic scheduling algorithm called Linear Programming DPM (LPDPM) which tries to increase the duration of idle periods so that deeper low-power states may be attained. It models processor idle time as an additional task and tries to reduce the number of preemptions (or executions) of this additional task. In [68], Legout et al. improved their previous work by employing an existing online scheduler called Fixed Priority until Zero Laxity (FPZL) to schedule tasks inside intervals delimited by consecutive task releases. The approach uses dynamic slack reclamation in order to activate deeper low-power states online. Then they extended their works in [69] to both hard real-time and mixed-criticality (MC) systems. In all their works, they used a mixed integer linear program to compute a partial schedule that optimizes the length of idle periods and an existing scheduling algorithm (FPZL) to further increase the length of the idle periods online. Since FPZL is not optimal for generic periodic tasks having arbitrary period lengths, these algorithms are also not optimal. Also, all these algorithms use a hybrid offline-online strategy and hence cannot be employed in completely dynamic scenarios where a task may arrive/depart at

any time. A purely online adaptive static power management strategy called Balanced Workload Scheme (BWS) has been presented by Chen et al. in [29] for hard real-time pipelined multiprocessor systems. At each adaptation instant, the BWS heuristic attempts to maximize the number of processors that may be switched to sleep mode and exploits the slacks generated at run-time to effectively extend sleep durations.

However, there has not been a significant effort towards the development of energy-efficient proportional fair scheduling methodologies. In chapter 3, we have chosen *ER-fair* [5], a work-conserving proportional fair scheduler as our underlying scheduling scheme and developed a novel energy-efficient algorithm called *ERfair Scheduler with Suspension on Multiprocessors (ESSM)*. The ESSM algorithm attempts to locally maximize the total length of suspension intervals while simultaneously reducing the number of such intervals using a novel procrastination mechanism, thus lowering energy consumption in the process.

2.3.4 Fault-tolerant Real-time Scheduling Strategies

In this section, we first provide an overview of various fault-tolerant techniques used in the design of real-time systems. Then, we discuss a review of various existing fault-tolerant scheduling strategies on homogeneous multiprocessor systems.

2.3.4.1 Fault-tolerant Techniques

As discussed in the introductory chapter, two major approaches for achieving fault tolerance are *time* and *hardware redundancies* [37,80]. Time redundancy based approaches use the slack capacity available in an underloaded system to achieve fault tolerance [64]. Important time redundancy based strategies include re-execution [94, 95] and checkpointing with rollback recovery [40, 57]. In the re-execution scheme, whenever a fault is detected, the faulty task is either re-executed from the beginning or a different version of the task, called recovery block, is executed in order to recover from the fault. Unlike the re-execution based approach, checkpointing involves periodically saving the intermediate states of a task during its execution. On the occurrence of a failure, the latest saved internal state of the task is restored and execution resumes from this saved state.

Hardware redundancy based approaches incorporate extra hardware into the design to either detect or override the effects of failed components [62]. Important hardware redundant strategies include *N-modular redundancy* and use of *standby spares* [37, 51]. In N-modular redundancy, multiple units running in parallel execute redundant copies of the same workload and mask errors by voting on their outputs. In the standby spares approach (also called *backup redundancy*), the fault affected primary unit is replaced by an identical secondary unit subsequent to a fault.

Two major standby-sparing techniques used to achieve fault-tolerance in real-time systems are *hot-standby* and *cold-standby* [72, 113]. In hot-standby, the backup unit runs concurrently with the primary unit and so, there is no delay in replacing the faulty primary with the backup. However, the resource demands for such a system could be about twice or even more (depending on the degree of fault-tolerance desired) compared to a system without hot-standby sparing. On the other hand in cold-standby, the backup becomes operational only after a fault in the primary is detected. As the primaries and backups do not execute concurrently, additional resource demands that are necessary to achieve fault-tolerance in cold-standby systems are typically far lower compared to systems with hot-standbys. However in this case, recovery to the nominal system state after failure requires a finite amount of time called *recovery time*, to replace the faulty primary [72, 113]. In addition to lower resource demands, another important advantage of cold-standby sparing is that fault-tolerance do not necessitate extra power for running spares, during normal operation. Hence, this scheme may be useful for systems where power consumption is an important design constraint. Moreover, a system equipped with cold-standby spares has lower overall operational costs and higher life times compared to a hot-standby system.

2.3.4.2 A Review of Fault-tolerant Scheduling on Multiprocessor systems

Fault-tolerant scheduling approaches for handling transient as well as permanent processor failures in multiprocessor systems have received a lot of attention in the last two decades [2, 20, 59, 64, 94, 95]. Time and hardware redundancies as well as a combination of both, are the major approaches towards achieving fault-tolerance in real-time sys-

tems. A detailed survey on different fault-tolerant scheduling schemes for homogeneous real-time multiprocessor systems may be found in [64]. In this section, we discuss a few important fault-tolerant scheduling approaches in detail.

Time redundancy based scheduling schemes: Time redundancy based approaches use the slack capacity available in an underloaded system to achieve fault-tolerance [64]. Important time redundancy based scheduling strategies include re-execution [94, 95] and checkpointing with rollback recovery [40, 57]. In the re-execution scheme, whenever a fault is detected, the faulty task is either re-executed from the beginning or a different version of the task, called recovery block, is executed in order to recover from the fault. Pathan and Jonsson [95] presented a re-execution based time redundant fault-tolerance scheme for scheduling a set of fixed-priority sporadic tasks on multiprocessors, to tolerate multiple permanent as well as transient failures. They also presented a feasibility test that can be used to ensure satisfaction of all deadlines even in the presence of processor failures and task errors. Recently, Pathan [94] extended the fault-tolerant framework presented in [95] to incorporate probabilistic schedulability guarantees, resulting in the probabilistic satisfaction of individual task deadlines. It may be noted that the works presented in [94, 95] use a fixed priority scheduling scheme which may possibly result in significantly lower resource utilizations compared to dynamic priority schemes. In addition, the preemptive nature of these global multiprocessor scheduling policies make them potentially susceptible to unrestricted preemptions and migrations. Unlike the re-execution based approach, checkpointing involves periodically saving the intermediate states of a task during its execution. On the occurrence of a failure, the latest saved internal state of the task is restored and execution resumes from this saved state. However, saving checkpoints has an associated cost in terms of both time and space, and hence, checkpoints when taken too frequently may lead to significant overheads. In [40], El-Sayed and Schroeder provided an extensive analysis of the performance, energy and I/O costs associated with a wide array of checkpointing policies. Checkpointing schemes must not only maintain an account of overheads, but also be aware of the available slack

capacity at all times so that the increase in overall execution cost do not lead to deadline violations.

Hardware redundancy based scheduling schemes: Hardware redundancy based approaches incorporate extra hardware into the design to either detect or override the effects of failed components [62]. Important hardware redundancy based scheduling strategies include N-Modular Redundancy (NMR), Primary/Backup (PB), and Standby Sparing (SS) [64]. In NMR, multiple copies of a hardware resource running in parallel execute redundant copies of the same workload and mask errors by voting on their outputs. Since NMR requires multiple hardware units, it is expensive and used only in very critical fault-tolerant systems. In the PB approach, each task is considered to have one primary copy and one or more backup copies. A backup copy may be active or passive. An active backup always executes along with its primary while a passive backup is activated only after the primary fails. In the SS approach, one or more spare processors are maintained as standby and the fault affected primary processor is replaced by an identical secondary unit subsequent to a fault. A majority of the PB based approaches often utilize partition-oriented scheduling schemes to assign the primary and backup copies of tasks onto distinct processors. Whenever a transient or permanent processor failure is detected at run-time, the outputs of the backup copies assigned on non-faulty processors are considered as the correct outputs. As a consequence of partitioning, achievable resource utilizations with these approaches may be considerably lower compared to global schemes [8]. In [2], Al-Omari et al. proposed an adaptive primary-backup (PB) based fault-tolerant scheme to schedule soft real-time aperiodic tasks on multiprocessor systems. By considering the dynamics of faults and task parameters in the system, they provided a mechanism which controls the degree of overlap between the primary and backup versions of tasks within the schedule. Kim et al. [59] presented a fault-tolerant task allocation strategy called R-BFD (Reliable Best-Fit Decreasing) which allocates active backups in such a way that a primary and its active backups are not assigned on the same processor. In their work, they extended R-BFD by proposing another task allocation algorithm called R-BATCH (Reliable Bin-packing Algorithm for Tasks with

Cold standby and Hot standby) which reduces the resource over-provisioning costs using passive backups. Recently, Bhat et al. [20] presented a system model in which each task is characterized by an application-specific constraint called *recovery time requirement* (RTR). RTR specifies the number of consecutive deadlines of the primary task that a backup can afford to miss without the system being considered to have failed. The authors then use this RTR based model and extended the fault-tolerant task allocation problem discussed in [59]. They presented different task allocation strategies which satisfy the recovery time requirements of all tasks while attempting to optimize resource utilization. However, determining the optimal resource over-provisioning required to efficiently implement a PB based scheme is a complex problem and requires careful design.

Time and Hardware redundancy based scheduling schemes: A few fault-tolerant scheduling mechanisms based on a combination of time and hardware redundancies may also be found in the literature. Mottaghi et al. [87] presented a fault-tolerance mechanism which dynamically selects either hardware redundancy or checkpointing with rollback recovery based on the criticalities of tasks. In [57], Kang et al. presented a fault-tolerant scheme which utilizes dual modular redundancy (DMR) and checkpointing to detect and correct transient faults that may corrupt the result for a given application modeled as a task graph. They determined the optimal points in the schedule where checkpoint should be placed while taking into account the associated overheads.

In spite of the additional flexibility and considerably higher resource utilizations that cold-standby sparing can potentially achieve within a global scheduling scenario, currently there does not exist any significant research that attempts at such a design approach. This is primarily because of the following two major challenges related to global scheduling with cold-standby: i) Efficiently managing task executions both during nominal system operation as well as during the recovery interval, so that all timeliness constraints can always be guaranteed over the entire schedule length, ii) Efficiently handling the execution progress of tasks while having an upper bound on both preemptions and migrations so that their overheads can be accounted even in the face of faults. In Chap-

ter 4, we propose a novel combined time-cum-hardware redundancy based fault-tolerant semi-partitioned scheduling strategy called *Fault Tolerant Fair Scheduler* (*FT-FS*) for real-time multiprocessor systems. In order to appropriately handle both the design challenges just mentioned, a work-conserving version of the DP-Fair algorithm has been used as the underlying scheduling methodology. Thus, the proposed FT-FS scheduler is able to effectively combine the benefits of high resource utilization and bounded context switches of DP-Fair along with all the advantages of cold-standby sparing as discussed above.

2.3.5 Energy and Fault Aware Real-time Scheduling Strategies

Recently, the problem of efficient handling of both energy and fault becomes an important research topic in real-time systems. Ejiali et al. proposed a standby-sparing based energy-aware fault-tolerant scheme for the scheduling of aperiodic tasks on dual-cores [39]. In order to reduce energy consumption, they employed DVFS for the main unit and DPM for the spare. Later, this work has been extended by Haque [52] et al. for periodic real-time applications. Recently, Guo et al. generalised the work presented in [52] for real-time systems deployed on platforms consisting of more than two processing cores [49]. An energy-aware partitioning and scheduling algorithm for standby-sparing systems has been presented in [112]. It is essentially a primary-backup approach for dual-core platform where the primaries and back-ups of tasks are always mapped to distinct cores. Given two-cores V_1 and V_2, if the primaries of all tasks which require shared resources are mapped to one of the cores (say, V_1), while the primaries of the remaining tasks which do not require shared resources are mapped to the other core (say, V_2). Another scheduling technique for standby-sparing systems has been proposed in [85]. They presented a DVFS based energy-aware strategy to schedule fixed-priority real-time tasks. However, all these works [39, 49, 52, 85, 112] have been targeted towards multi-core platforms consisting of *homogeneous* cores.

In the context of emerging heterogeneous multiprocessor/multi-core systems, there exists a severe dearth of energy-aware and fault-tolerant scheduling schemes. Recently, Roy et al. explored energy-awareness on heterogeneous multicore platforms consisting of

both high-performance and power-efficient cores, using standby-sparing [100]. However, their strategy is oblivious of the number of faults to be tolerated. Due to this, the offline schedule constructed by their scheme consumes more energy than that required to tolerate a specified number of faults. To alleviate this issue, in Chapter 5, we propose a standby-sparing based energy-aware fault-tolerant scheduling strategy for heterogeneous systems. We develop a low-overhead heuristic scheme called, FENA-SCHED, for the fault-tolerant scheduling of real-time applications on heterogeneous dual-core systems consisting of a power-efficient core and a high-performance core. FENA-SCHED utilizes DPM with *backup-backup overloading* [46, 47] to minimize energy consumption while guaranteeing tolerance against a given number of transient processor faults.

The design strategies for fault tolerance discussed above assume faults to be detectable. However, enforcement of fault tolerance can only be achieved through the incorporation of safe design methodologies which enable efficient active monitoring and detection of unobservable faults in the system. Therefore, it is desirable to incorporate efficient strategies for fault diagnosis (detection and isolation) in the construction of safety-critical systems. With this insight, in the next two sections, we discuss an important formally constructed safe design methodology known as *Discrete Event System* that is often used to make a system amenable to efficient active monitoring and detection of unsafe execution states (that is, faults) whenever the system behavior deviates from its stipulated specification.

2.4 DES Modeling and Fault Diagnosis Framework

In this section, we present in brief the state-based modeling formalism for DES framework where each state represents a distinct system status. For more details, the reader is referred to literature [1, 23, 25, 104, 115].

The DES model G is defined as:

$$G = \langle X, S, \Im, X_0 \rangle \tag{2.5}$$

where X is a finite set of states, $S = \{s_1, s_2, ..., s_n\}$ is a finite set of discrete state variables assuming values from some finite sets, called the domains of the variables, $\Im$

is a finite set of transitions and X_0 is the initial state.

Any state $x \in X$ is a mapping of each state variable to one of the elements of the domain of the variable. A *transition* $\tau \in \Im$ from a state x to another state x^+ defined as $\tau = \langle x, x^+ \rangle$, where, $x = initial(\tau)$ is the initial state of the transition and $x^+ = final(\tau)$ is the final state of the transition. A *trace* of a model G is a sequence of transitions *generated* by G denoted as $q = \langle \tau_1, \tau_2, ..., \tau_f \rangle$, where $initial(\tau_{i+1}) = final(\tau_i)$, for $i = 1$ to $(f-1)$. The language of G, $L(G)$ is a subset of $\Im^w$, where $\Im^w$ is the set of all infinite sequences of $\Im$. Any finite prefixes of $L(G)$ is a subset of $\Im^*$, the Kleene closure of $\Im$. The post language of G after a trace q is denoted as $L(G)/q = \{r \in \Im^* \mid qr \in L(G)\}$. $L_f(G)/q \subset L(G)/q$ comprises of the finite prefixes of the infinite traces of $L(G)/q$. We assume G is live (this means that there is a transition defined at each state x in X) and each model state is reachable from some initial state(s).

2.4.1 Model with Measurement Limitation

This subsection formally introduces the notion of measurement limitation in the DES framework and discusses the non-determinism in the states and transitions of G due to the limitation in the measurability of one or more state variables.

Measurements: In practical systems, it may not be possible to measure all the variables at a given time. Under this limitations in measurement, the state variables in the system can be partitioned into two disjoint subsets: *measurable* set, S_m and *unmeasurable* set, S_u, where

$$S = S_m \cup S_u \;\; and \;\; S_m \cap S_u = \phi. \tag{2.6}$$

Definition 2.4.1 (Measurement Equivalence of G-states)**.** The *measurement equivalence* of two G-states x and y denoted by xEy, is defined as $x|_{S_m} = y|_{S_m}$, where $x|_{S_m}$ is the restriction of the domain of the state variables in the state x to S_m.

Therefore, all the measurable state variables of any two measurement equivalent states have same values.

Measurable and Unmeasurable Transitions: Under the limitations in measurement, the transition set $\Im$ is partitioned into two sets, *measurable* (denoted as $\Im_m$) and *unmeasurable* (denoted as $\Im_u$). A transition $\tau = \langle x, x^+ \rangle$ is said to be unmeasurable if xEx^+ (that is, states x and x^+ are measurement equivalent).

Definition 2.4.2. A *projection operator* $P : \Im^* \rightarrow \Im_m^*$ is defined as: $P(\epsilon) = \epsilon$, the null string; $P(\tau) = \tau$, if $\tau \in \Im_m$; $P(\tau) = \epsilon$, if $\tau \in \Im_u$; $P(q\tau) = P(q)P(\tau)$, where $q \in L_f(G)$, $\tau \in \Im$.

Therefore, the projection P of a trace q erases the unmeasurable transitions from that trace. $P(q)$ is termed as the *measurable trace corresponding to the trace* q.

Definition 2.4.3. The inverse projection operator $P^{-1} : \Im_m^* \rightarrow 2^{\Im^*}$ is defined as

$$P^{-1}(q) = \{q' \in \Im^* \mid qEq'\}. \tag{2.7}$$

Therefore, the inverse projection P^{-1} on a measurable trace q gives the set of sequences of measurement equivalent transitions.

Definition 2.4.4 (Measurement Equivalence of G-Transitions)**.** Two measurable transitions $\tau_1 = \langle x_1, x_1^+ \rangle$ and $\tau_2 = \langle x_2, x_2^+ \rangle$ are *measurement equivalent*, denoted as $\tau_1 E \tau_2$, when $x_1 E x_2 \wedge x_1^+ E x_2^+$. Similarly, two sequences of transitions q and q' are measurement equivalent if $P(q) = \langle \tau_1, \tau_2, ... \rangle$, $P(q') = \langle \tau_1, \tau_2, ... \rangle$ and $\tau_i E \tau_i', i \geq 1$.

2.4.2 Fault Modeling

Each state x is assigned a failure label defined by an unmeasurable status variable $C \in S$ with its domain $= \{N, F_1, F_2, ..., F_k\}$, where $F_i, 1 \leq i \leq k$, stand for permanent failure status and N stands for normal status. Therefore, a state in the model may either denote a normally operating status of the system or a faulty status. The set of all normal G-states is denoted as X_N, where $x_N \in X_N$ represents a normal G-state. We assume k types of fault in the system. A state is denoted as x_{F_i} if it is a faulty state due to the fault of type i. For a normal G-state x_N, $x_N(C) = \{N\}$. Similarly, for a failure state (or synonymously, an F_i G-state) x_{F_i}, $x_{F_i}(C) = \{F_i\}$. The set of all states x such

that $F_i \in x(C)$ is denoted as X_{F_i}. We use the words fault and failure interchangeably because here, the fault or failure of a system mean the same. Simultaneous occurrence of more than one fault is not considered here.

In fault modeling, a G-transition $\tau = \langle x, x^+ \rangle$ is called a *normal G-transition* if $x, x^+ \in X_N$. The set of all normal G-transitions is denoted as $\Im_N$. A G-transition $\tau = \langle x, x^+ \rangle$ is called an F_i *G-transition* if $x, x^+ \in X_{F_i}$. The set of all F_i G-transitions is denoted as $\Im_{F_i}$. Similarly, a G-trace q is called normal G-trace if all transitions in q are normal G-transitions. If all transitions in a G-trace q are F_i G-transitions then q is called F_i G-trace. A *failure* causing transition $\tau_{F_i} = \langle x, x^+ \rangle$, where $x(C) \neq x^+(C)$ indicates the first occurrence of some failure F_i. Since failures are assumed to be *permanent*, there is no transition from any x_{F_i} to any x_N.

A DES model G is said to be diagnosable if it is always possible to determine the faulty status of the states after the occurrence of a fault, using the sequence of measurements. Let $\Psi(X_{F_i}) = \{q | q \in L_f(G)$ and $final(q) \in X_{F_i}$ and q ends in a measurable transition$\}$.

Definition 2.4.5. **F_i-Diagnosability**: A DES model G is said to be F_i-diagnosable under a measurement limitation for fault F_i if the following holds

$$(\exists n \in \mathbb{N})[\forall q \in \Psi(X_{F_i})](\forall r \in L_f(G)/q)[|r| \geq n \Rightarrow D] \tag{2.8}$$

where the condition D is $\forall u \in P^{-1}[P(qr)],\ final(u) \in X_{F_i}$.

This definition is taken from the literature, [103, 104]. It means: let q be any finite prefix of a trace of G that ends in an F_i-state and let r be any sufficiently long continuation of q. Condition D then requires that every sequence of transitions, measurement equivalent with qr, shall end into an F_i-state. This implies that, along every continuation r of q, one can detect the occurrence of fault F_i within a finite delay, specifically in at most n transitions of the system after q.

Example: Consider a benchmark system consisting of a pump, a valve and a controller (Figure 2.5(a)). Assume that the system is equipped with a valve flow sensor and let

its outputs be no flow and flow. The state-based DES model of this pump-valve system, defined as $G = \langle X, S, \Im, X_0 \rangle$ is shown in Figure 2.5(b).

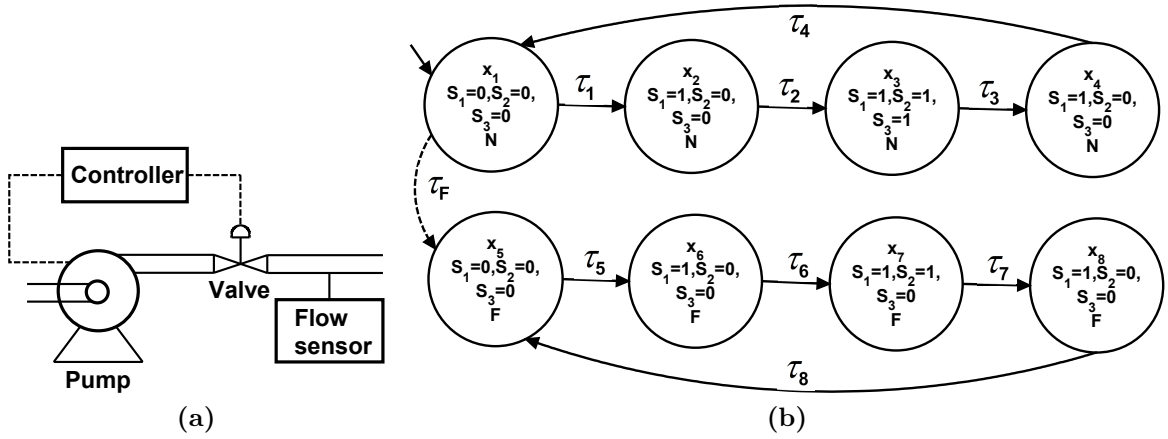


Figure 2.5: *(a) Pump-valve system; (b) its state-based DES model.*

Now, we illustrate various notions corresponding to this state-based DES modeling formalism in detail. In a state-based approach, the state set of the system is partitioned according to the failure status of the state [115]. We assume a "Stuck_Closed" failure of the valve (say, fault F) in the system. The pump and controller are assumed to be fault-free. The occurrence of the fault F in G is represented through an *unmeasurable* transition τ_F. Therefore, the state set X of G is partitioned as $X = X_N \cup X_F$, where $X_N = \{x_1, x_2, x_3, x_4\}$ and $X_F = \{x_5, x_6, x_7, x_8\}$. $X_0 = \{x_1\}$. $S = \{s_1, s_2, s_3, C\}$ is the set of state variables associated with each state x of G. Here, the state variables s_1, s_2, s_3 denote status of the valve (open (1)/closed (0)), status of the pump (on (1)/off (0)) and readings of the flow sensor (flow (1)/no flow (0)), respectively, and C is the failure label with its domain being $\{N, F\}$. $S_m = \{s_1, s_2, s_3\}$ and $S_u = \{C\}$. All transitions except τ_F are measurable. Here, transitions τ_1 and τ_5 are measurement equivalent, i.e., $\tau_1 E \tau_5$ due to $x_1 E x_5 \wedge x_2 E x_6$ (see, Definitions 2.4.1, 2.4.4). □

Note: There has traditionally been two distinct streams of works related to fault diagnosis based on whether the DES modeling methodology is event-based or state-based. The transition τ_1 of model G shown in Figure 2.5(b) can be equivalently defined in an

event-based model as follows: τ_1 is fired from state x_1 to state x_2 due to the occurrence of "valve open" event (say, VE) at state x_1. So, τ_1 can defined as $\tau_1 = \langle x_1, VE, x_2 \rangle$. In an event-based approach, a fault is generally represented using an unobservable event (say, event f). Therefore, its corresponding failure causing transition from a state x to another state x^+, denoted as $\tau_f = \langle x, f, x^+ \rangle$ also becomes unobservable. In event-based diagnosis approaches, fault diagnosis (detection and identification of the occurrence of a fault) is performed based on the observation of event sequences [103, 104]. Here, a system model G is said to be diagnosable for any fault event if its occurrence can be detected within a finite delay using the record of observed events. In a state-based approach, the state set of the system can be partitioned according to the faulty status of the state [115]. Each state x is assigned a failure label defined by an unmeasurable status variable $C \in S$ with its domain being $\{N, F_1, F_2, ..., F_k\}$, where $F_i, 1 \leq i \leq k$, stand for permanent failure status and N stands for normal status. For example, let us consider the two states x_1 and x_5 of model G shown in Figure 2.5(b). Even though the measurable state variables (S_1, S_2, S_3) of x_1 and x_5 have same values, their unmeasurable status variable C has different values, that is, $x_1(C) = N$ and $x_5(C) = F$. Therefore, the transition $\tau_F = \langle x_1, x_5 \rangle$ represents a failure causing transition and is unmeasurable. In state-based approaches, fault diagnosis is performed based on the sequence of output measurements associated with the system states. The assumption on partitioning the state space of the system has two benefits [115]. First, this is particularly useful in cases where the failure might have occurred before the start of diagnosis. In these situations, a failure can be diagnosed by determining the faulty status of the states using the sequence of measurements. Another benefit is that this framework simplifies the transition function of the diagnoser. Specifically, at each step, after receiving a new measurement, this approach only has to update the estimate of the system's state as normal or faulty or uncertain, and thus it avoids label propagation as done in [103]. □

2.4.3 Diagnoser Construction Procedure

This subsection describes the construction procedure for the diagnoser G_{diag} under measurement limitation. As of now, we denote the states, transitions and traces of model

G as G-states, G-transitions and G-traces, respectively. Similarly, we use D-states, D-transitions and D-traces, respectively to represent the diagnoser states, transitions and traces.

The diagnoser is represented as a directed graph

$$G_{diag} = \langle Z, A \rangle \tag{2.9}$$

where Z is the set of *D-states*, and A is the set of *D-transitions*. Each D-state $z \in Z$ is a set of G states representing the uncertainty about the actual state and each D-transition $a \in A$ of the form $\langle z_i, z_f \rangle$ is a set of measurement equivalent transitions, representing the uncertainty about the occurrence of the actual measurable transition. The following definition is introduced to discuss the procedure for constructing the diagnoser from G.

Definition 2.4.6. Unmeasurable reach of a set of G-states: The unmeasurable reach of a set Y of G-states is the transitive closure (Kleene closure) of the unmeasurable successors of Y and is denoted as $\mathcal{U}^*(Y)$, where the unmeasurable successor of a set Y of G-states is defined as $\mathcal{U}(Y) = \bigcup_{x \in Y} \{x^+ | \tau = \langle x, x^+ \rangle \in \Im_u\}$

Construction of the Diagnoser: The initial D-state (z_0) is obtained as $\mathcal{U}^*(X_0)$. Now, consider the construction of the transitions from a D-state z. Let $\Im_{mz}$ denote the set of measurable G-transitions from the states $x \in z$. Let A_z be the set of all measurement equivalence classes of transitions obtained from $\Im_{mz}$. Corresponding to each of these classes, there is a transition a, emanating from z. For a transition a emanating from the D-state z, the successor state z^+ via the transition a is computed in two steps: (i) first, a set z_a^+ is computed as the set $\{final(\tau) | \tau \in a\}$, (ii) the set z^+ is then obtained as $z^+ = \mathcal{U}^*(z_a^+)$. For each $a \in A$, the initial and the final D-states are designated as $initial(a)$ and $final(a)$, respectively. Therefore, from the above discussion, $initial(a) = z$ and $final(a) = z^+$. The set of the diagnoser transitions is augmented as $A \leftarrow A \cup \{a\}$ and the set of states is augmented as $Z \leftarrow Z \cup \{z^+\}$. It may also be noted that each D-state $z \neq z_0$ contains measurement equivalent states; z_0, however, may contain (initial) states, which are not necessarily measurement equivalent.

Diagnosability Analysis: Now, we introduce certain definitions and properties needed for diagnosability analysis on the diagnoser.

Definition 2.4.7. Embedding of G-traces in D-traces:. Given a D-trace $\gamma = \langle a1, a2, ..., ak\rangle$, a G-trace q, where $P(q) = \langle \tau_1, \tau_2 ..., \tau_k\rangle$, is said to be embedded in γ, if $\tau_i \in ai$, $1 \leq i \leq k$. The set of all G-traces embedded in a D-trace γ is represented as $A_D(\gamma)$.

The fault label of any D-state $z = \langle x_1, x_2, ...x_i, ...\rangle$ is defined as $z(C) = \bigcup_{x \in z} x(C)$.

Definition 2.4.8. Normal D-state:. A D-state z is called *normal* and denoted as z_N, if $z(C) = \{N\}$; the set of all normal D-states is denoted as Z_N.

Definition 2.4.9. F_i D-state:. A D-state z is called an F_i D-state and denoted as z_{F_i}, if $F_i \in z(C)$. The set of all F_i D-states is denoted as Z_{F_i}.

Definition 2.4.10. F_i-certain D-state:. An F_i D-state z is called an F_i-certain D-state if $z \subseteq X_{F_i}$.

Definition 2.4.11. F_i-uncertain D-state:. An F_i D-state which is not F_i-certain is called F_i-uncertain.

Property 1. If two traces $q, y \in A_D(\gamma)$, where q is an F_i G-trace and y is a normal G-trace, then the D-states traversed by γ are F_i-uncertain.

Proof. The property also follows from diagnoser construction. As any D-transition $a \in \gamma$ has a normal G-transition and a F_i G-transition (which are equivalent), so source and destination D-states of a are F_i-uncertain. □

Therefore, an F_i-certain D-state contains only F_i G-states whereas an F_i-uncertain D-state contains both F_i G-states and normal G-states. So, a fault is diagnosed if the diagnoser reaches any F_i-certain D-state. Let consider a D-trace γ consisting of a sequence of F_i-uncertain D-states which is actually a composition of a normal as well as faulty G-trace (see Property 1 and Definition 2.4.7). If so, a fault cannot be diagnosed until γ is exited.

Definition 2.4.12. **F_i-indeterminate cycle:**. An F_i-indeterminate cycle is an F_i-uncertain D-cycle such that there are at least two measurement equivalent *syntactic* cycles y and q in G, one comprising only normal G-states and the other comprising F_i G-states, corresponding to the D-cycle.

The equivalence between F_i-diagnosability and the absence of F_i-indeterminate cycles has been formally established for DES models [103, 115]. Consider two measurement equivalent cycles y and q in G, one comprising only normal G-states and the other comprising F_i G-states. If the system is under normal condition, the diagnoser moves in a normal cycle and once a fault F_i occurs it moves in the fault cycle. As both the normal and fault cycles are measurement equivalent, they are indistinguishable from one another. Therefore, in the presence of an F_i-indeterminate cycle, it is not possible to predict whether the diagnoser is moving under normal or fault cycle leading to non-diagnosability.

2.5 A Review of DES-based Fault Diagnosis Schemes

In this section, we discuss and analyze a few important state-of-the-art works related to DES-based fault diagnosis.

A wide variety of methodologies have been proposed to solve the problem of fault diagnosis for systems modeled as DESs [24, 91, 92, 99, 116]. The idea of failure diagnosis in DESs was first proposed by Sampath et al. [103,104]. A new fault detection paradigm called safe diagnosability was introduced and studied by Paoli and Lafortune in [91]. Safe diagnosability requires the detection of a fault prior to the commencement of an unsafe behavior within the failed mode of operation of the system. Then they extended their diagnosability theory to design a diagnosing controller which safely detects faults and takes control actions to switch between the nominal and reconfigured control policies, subsequent to faults [92]. A comprehensive survey for the state-of-the-art in fault diagnosis techniques for DESs has been discussed in detail by Zaytoon and Lafortune [116]. The gist of any DES based fault diagnosis framework is to develop normal and fault models corresponding to nominal and failure scenarios. Subsequently, a diagnoser is built using

the knowledge of the states traversed in the normal and faulty models. The diagnoser determines whether the system is operating under faulty, non-faulty or uncertain conditions. Such diagnosers therefore, significantly enhance reliability and predictability of systems operating under faults. However, the complexity of the diagnosis processes, that is, constructing a diagnoser and testing its diagnosability, is exponential in the number of system states and doubly exponential in the number of state variables [28, 103, 115]. This may lead to prohibitively huge state-space requirements in the design of diagnosers for large and complex systems.

In order to reduce complexity of the diagnosis process, various approaches have been presented in the literature. Among them, the objective of one class of approaches is to prevent the actual construction of a diagnoser, if faults in the system are verified to be non-diagnosable [45, 56, 114]. For this purpose, these schemes aim towards the design of light weight test frameworks which can verify the diagnosability of a DES model in polynomial-time. The other significant class of works essentially aim at reduction in the state space complexity involved in the construction of a diagnoser [33,105,115]. This class can further be considered to consist of strategies which follow one of two distinct design approaches. The first is the top-down approach which derives a reduced diagnoser from a monolithic model of the overall system using model reduction techniques [105, 115]. Model reduction essentially works by eliminating redundant models/variables in the system whose measurements are not required for fault diagnosis. The second approach follows a bottom-up strategy which generates a set of local diagnosers from component models and performs global diagnosis of the system by collectively combining the diagnosis of these local diagnosers [33, 36, 106]. These local diagnosers are typically derived from physically modular subcomponents within the system.

With these insights, discussion on DES-based fault diagnosis has been categorized based on the three major streams of research on formal approaches to fault diagnosis, namely, i) works that attempt to device low overhead diagnosability verification mechanisms, ii) approaches towards top-down diagnoser construction, and iii) strategies that attempt to synthesize the overall diagnoser bottom-up from subcomponent diag-

nosers/models. We now describe the overall themes of works corresponding to these major streams of research, in more detail.

2.5.1 Diagnosability Verification Strategy

The objective of this stream of approaches is to obtain a light weight test framework for verifying diagnosability in polynomial-time, without going through the heavily compute-intensive process of actually constructing a diagnoser [45, 56, 114].

Jiang et al. [56] proposed an algorithm for testing diagnosability whose complexity is of the fourth order of the number of states in the system. Yoo and Lafortune [114] presented a polynomial-time diagnosability verification algorithm by constructing a verifier automata whose complexity is of the second order of the number of states in the system. Then they modified this verifier to test a relaxed notion of diagnosability called *I-diagnosability* in polynomial time. I-diagnosability is applicable to those systems in which a failure event may be detected within a finite delay subsequent to the occurrence of special observable events called *indicator events* associated with that failure. For example, consider the case study with the *Heating, Ventilation and Air Conditioning* (HVAC) system discussed in Section II-B-2 of [103]. Under normal operation, the controller of the system issues the command 'open valve' whenever it senses a demand for heating (termed as heating load). Similarly, the command 'close valve' is issued when the load disappears. Sampath et al. [103] designated 'open valve' and 'close valve' as indicator events corresponding to the valve failure events, 'stuck-closed' and 'stuck-open', respectively. This is because the failure 'stuck-closed' ('stuck-open') can be diagnosed only after the command 'open valve' ('close valve') has been issued.

Gascard et al. [45] proposed a polynomial diagnosability verification strategy for systems whose models can be partitioned according to distinct operation modes. The principal advantage of these approaches is that a diagnoser needs to be constructed only if the light weight verifier deems the system to be diagnosable. However, only the process of diagnosability verification is performed in polynomial-time. If the test is successful, there is still a need to construct the diagnoser, and this may be highly compute intensive.

2.5.2 Diagnoser Design

We now focus our attention towards strategies dealing with the actual construction of a diagnoser. Sampath et al. [103] presented an event-based fault diagnosis approach which constructs a diagnoser from the global model of the system. They assumed the systems to be partially-observable where a subset of events are deemed to be unobservable due to the lack of sensors to detect their occurrence. For example, there are no sensors available to observe the "Stuck_Closed" failure or operating temperature of the engine or temperature of the air entering into the engine, in the EFI system discussed in Section 6.1, and hence, the events associated with them are unobservable. Diagnosis is conducted by extracting the observable part of both the nominal and faulty system models through a projection operation which eliminates all unobservable events. The principle drawback of this approach is the huge state space complexity of the diagnoser, which is exponential with respect to the number of model states and doubly exponential with respect to the number of events in the system, in the worst case. In order to reduce state space complexity involved in the construction of a diagnoser, various strategies have been adopted and these follow either a top-down or a bottom-up design approach.

2.5.2.1 Top-down Diagnoser Design Strategies

This class of strategies derive a reduced diagnoser from a monolithic model of the overall system using model reduction/abstraction techniques [105,115]. Zad et al. [115] proposed a model reduction technique for their state-based diagnosis framework and have shown its effect on a heating system (refer Section-II of [115]). The heating system uses a heater, a temperature sensor and a controller to regulate the temperature of a room about a set point. The effect of disturbances such as the temperature of an adjoining room and the ambient temperature, is represented through a load model. As the model for the heating system is functionally independent of the load model, faults associated with the heating system model also do not depend on the load model. Therefore, with respect to fault diagnosis of the heating system, consideration of the load model becomes redundant and its removal allows reduction in state space of the diagnoser, while not

compromising diagnosability. In order to obtain a diagnoser with reduced state space, Zad et al. [115] applied projection operations to derive an abstracted model of the system by eliminating redundant models and then constructing the diagnoser from this abstract model.

K. Schmidt [105] extended Zad's model reduction approach to a physically modular system in which each module is functionally independent of the behavior of other modules. The overall model for such a system can easily be obtained by hierarchically combining the models of each subcomponent module within it. K. Schmidt [105] applied projections on individual subcomponent models to obtain their reduced models. Although model abstractions are applied to individual subcomponent models, the diagnosers are not constructed at subcomponent levels as their abstract models cannot be guaranteed to be diagnosable. Thus, the abstract subcomponent models are first combined in a bottom-up fashion to obtain a reduced model for the entire system. Then, the diagnoser is constructed from this reduced monolithic model of the system in a top-down manner. They also presented the idea of abstraction-based diagnosability by representing a failure as the violation of a given prefix-closed specification language. However, in the worst case, state space complexity of reduced monolithic diagnosers constructed using either of these model reduction/abstraction schemes become comparable to that of a monolithic diagnoser constructed from the full-blown non-reduced system model [103].

2.5.2.2 Bottom-up Diagnoser Design Strategies

This class of strategies generates a set of local diagnosers from projected component models of a physically modular system and performs global diagnosis through the additive combination of the diagnoses of the local diagnosers [33, 36, 106]. Such additively combined diagnosers for a system have the potential of being exponentially smaller in their state space complexities compared to diagnosers derived from monolithic system models. Since this modular strategy uses a set of local diagnosers similar to MLAD to conduct global diagnosis, we discuss an important modular bottom-up diagnoser synthesis approach proposed by Debouk et al. [36], in detail.

Debouk et al. [36] presented a fault diagnosis framework for a modular architecture in

which each failure in the system can be directly associated to a distinct local component module (also called a subsystem), where it occurs. The diagnoser construction process starts by verifying the diagnosability of each subsystem model where faults may possibly originate and then actually constructing a local diagnoser for the model if the verification is successful. However, if the diagnosability test results in a failure, the non-diagnosable subsystem is incrementally composed with another subsystem, and the bigger composite subsystem thus obtained, is again subjected to a similar phase of diagnosability test and possible diagnoser construction. The composition is carried out with the objective of eliminating diagnosability violating traces from the non-diagnosable subsystem model (see Definition 2.5.3). This mechanism of incremental composition continues until the corresponding diagnosability test results in a success. The above process finally provides the minimal set of local diagnosers which together allows the diagnosability of all faults in the system. The diagnoser for the system is derived through the additive combination of these local diagnosers. Now, we discuss the notion of diagnosability introduced by Debouk et al. [36] in more detail.

Consider a modular system G composed of two subsystems G_1 and G_2. The event set Σ corresponding to G is partitioned into the set of observable events Σ_o and the set of unobservable events Σ_{uo}, such that, $\Sigma = \Sigma_o \cup \Sigma_{uo}$. Let $\Sigma_f \subseteq \Sigma_{uo}$ denote the set of failure events which are to be diagnosed and Π_f denote the failure partition. The prefix-closed language generated by G, denoted as $L(G)$, describes the behavior of the system. $G_i, i = 1, 2$ is defined on $\Sigma_i = \Sigma_{oi} \cup \Sigma_{uoi}$, with $\Sigma = \Sigma_1 \cup \Sigma_2$. Let Π_{fi} on $\Sigma_f^i, i = 1, 2$ denote the failure partition for $L(G_i)$ under the assumption that Π_{f1} and Π_{f2} do not share any failure type. The notion of diagnosability defined by Sampath et al. [103] is now extended to this modular architecture setup and is formally stated as follows:

Definition 2.5.1. (Sampath et al. [103]): A prefix-closed and live language L is said to be diagnosable with respect to a projection $P : \Sigma \rightarrow \Sigma_o$ and with respect to a partition Π_f on Σ_f, if the following holds:

$(\forall i \in \Pi_f)(\exists n_i \in N)(\forall s \in \Psi(\Sigma_{fi}))(\forall t \in L/s)(\parallel t \parallel \geq n_i \Rightarrow (w \in P_L^{-1}(P(st)) \Rightarrow \Sigma_{fi} \in w))$, where $\Psi(\Sigma_{fi})$ denotes the set of traces ending with a failure event in Σ_{fi}, L/s denotes all continuation in $L(G)$ of the trace s, $\parallel t \parallel$ denotes the length of trace t,

and $P_L^{-1}(u)$ denotes all traces v in $L(G)$ such that $P(v) = u$.

Informally, Definition 2.5.1 means that a language is said to be diagnosable with respect to a set of observable events and a failure partition if within a finite delay, the occurrence of any failure can be detected using the record of observable events. A property is formalized in the context of languages in the following definition under the assumption that the failure is diagnosable locally.

Definition 2.5.2. (Debouk et al. [36]): Consider the languages $L(G)$, $L(G_1)$ and $L(G_2)$ defined over the set of events Σ, Σ_1 and Σ_2, respectively. Let $G = G_1 \parallel G_2$. $L(G)$ is said to be live with respect to $L(G_i), i = 1$ or 2 if $\forall s \in L(G), Q_i(s)$, where Q_i is the projection from $\Sigma \rightarrow \Sigma_i$, is arbitrarily long.

The sufficient conditions for a system to be diagnosable under the modular architectural setup is defined as:

Definition 2.5.3. (Debouk et al. [36]): $L(G)$ is diagnosable with respect to the set of observable events Σ_o and the failure partitioning Π_f on Σ_f if $L(G_i), i = 1, 2$, is diagnosable with respect to the set of observable events Σ_{oi} and the failure partitioning Π_{fi} on Σ_f^i, and $L(G)$ is live with respect to $L(G_i)$.

Informally, Definition 2.5.3 means that the modular architecture is able to detect and isolate all failures if for each of these failures the diagnoser of the subsystem which exhibits the failure is capable of detecting and isolating it.

The applicability of modular approaches is limited under two distinct scenarios, (i) when the architecture of the system is inherently monolithic and thus, making it difficult to obtain modular subcomponent DES models; (ii) when one or more fault types are majorly distributed over a significant part of the entire system and hence, it is difficult to localize these faults within one or few subcomponents. In both the above scenarios, the combined state space of the generated localized diagnosers may become comparable or even higher than the diagnoser that is derived from the overall original DES model of the system. Hence for these scenarios, we may not obtain any significant reduction in state space.

With the insights obtained from the analysis of different approaches for reducing complexity of the diagnosis process, we propose here a top-down approach called *Measurement Limitation based Abstract DES Diagnosis* (*MLAD*), towards localized light weight diagnoser construction from reduced DES models. Here, such reduced models have been obtained through the behavioral decomposition of the overall DES model on the basis of the compartmentalization of faults, instead of functional modularization as in [33,36,106]. The basic DES model reduction mechanism is based on measurement limitation of the state variables similar to the work presented by Zad et al. [115]. However, Zad's work allows limitation to only redundant variables such that diagnosability of the resultant reduced model is never compromised. In comparison, MLAD carefully chooses and forcefully limits the measurement of a subset of even the non-redundant variables to obtain reduced models from which controlled partially compromised diagnosers can be constructed. Given the set of all (say k) faults, such a reduced diagnoser can be obtained by choosing a carefully designated subset of variables whose limitation can possibly lead to compromised diagnosability of only a stipulated subset of (say k') faults while not affecting the diagnosability of the remaining $(k-k')$ faults. MLAD performs a stipulated number (say L) of such controlled limitations on the original model to obtain a set of L partially compromised reduced diagnosers whose combination ensures diagnosability of all faults. For example, let us consider a system with k possible faults and modeled using S variables. Let us assume that measurement limitation of a designated subset of S_1 variables in S produces a reduced model whose corresponding diagnoser G^1_{diag} (have state space say, Z_1) compromises a distinct subset of (say) $k - k_1$ faults out of the k possible faults. Thus, MLAD chooses L separate subsets of $S_1, S_2, ..., S_L$ variables $(\forall i, S_i \subseteq S)$ and applies measurement limitation on the original model using each such subset to allow the generation of L reduced diagnosers $G^1_{diag}, G^2_{diag}, ..., G^L_{diag}$ (having corresponding state spaces $Z_1, Z_2, ..., Z_L$) which ensure the diagnosis of $k_1, k_2, ..., k_L$ faults, respectively. The L subsets of variables are so chosen that $k = k_1 \cup k_2 \cup ... \cup k_L$. Thus, the L reduced diagnosers running in parallel ensure the diagnosability of all faults. The design approach discussed above allows significant reduction in state space from the

conventional approach if, $Z_1 + Z_2 + ... + Z_L << Z$, where Z represents the state space of the single full diagnoser derived from the original DES model. Analysis over various test cases show that by effectively selecting the sets of variables to be measurement limited, *MLAD* is able to achieve handsome reductions in state space in most practical scenarios by additively combining a set of streamlined partially compromised diagnosers. Generally, the complexity of the proposed work is additive in nature with respect to the exponential complexity of the existing monolithic approaches.

2.6 Summary

This chapter started with a brief overview about the basic terms and definitions of real-time systems, followed by literature survey of various scheduling algorithms for real-time multiprocessor systems with the consideration of different design parameters such as timeliness, energy awareness, fault tolerance, etc. Then, we have presented the fundamental definitions related to DES-based fault diagnosis, followed by an overview of various DES-based fault diagnosis mechanisms for safety-critical systems. These concepts and definitions may either be referred or reproduced appropriately later in this thesis. In the next chapter, we present an energy-aware fair scheduling strategy namely ESSM, developed by us for real-time homogeneous multiprocessor systems.

Chapter 3

Energy-efficient Fair Scheduling on Real-time Multiprocessor Systems

In the last chapter, we discussed various scheduling algorithms for real-time multiprocessor systems with the consideration of different design parameters. We also discussed and analyzed various DES-based fault-diagnosis mechanisms for safety-critical systems. As mentioned earlier, there has not been a significant effort towards the development of energy-efficient proportional fair scheduling methodologies. In this chapter, we develop an efficient energy-aware fair scheduling strategy for real-time safety-critical systems having homogeneous multiprocessors as the computing platform. The proposed energy-efficient algorithm called *ERfair Scheduler with Suspension on Multiprocessors (ESSM)* makes use of *ERfair* [5], a work-conserving proportional fair scheduler as an underlying scheduling scheme. The ESSM algorithm attempts to locally maximize the total length of suspension intervals while simultaneously reducing the number of such intervals using a novel procrastination mechanism, thus lowering energy consumption in the process.

The chapter first presents the energy consumption model and specification of the system considered in our work. The proposed energy-efficient ESSM scheduling scheme, along with an example and analysis of its complexity is described next. Subsequently, we discuss the important experimental results which highlight the performance of the proposed energy-aware scheduling strategy under various scenarios. Finally, we conclude our work.

3.1 Energy Consumption Model

The basis of our energy management strategy is to suspend one or more processors during idle times, as negligible amount of energy is consumed in suspended mode. However, a fixed amount of time and energy is consumed when the system transits between the idle and active states. Therefore, the processor needs to stay at the lower power state for at least long enough so as to recover the cost of transitioning in and out of the state. This is called the break-even time TI_{be} [19, 66] of a processor. That is, TI_{be} is the minimum time length of an idle period during which shutting down a processor will save power, and is defined as follows:

$$TI_{be} = TI_{tr} + TI_{tr}\frac{P_{tr} - P_{on}}{P_{on} - P_{sleep}} \tag{3.1}$$

Here, P_{on} represents the power consumption in active state, P_{sleep} is the power consumed in the lower power state, while TI_{tr} and P_{tr} is the cost of transition in terms of time and power, respectively. In general, TI_{be} is the sum of two terms: TI_{tr}, the total transition time (i.e., the time required to enter and exit the low-power state), and $TI_{tr}\frac{P_{tr}-P_{on}}{P_{on}-P_{sleep}}$, the minimum time that has to be spent in the low-power state to compensate the excess power consumed during state transition [19, 66]. Hence, whenever a suspension interval is created, Equation 3.1 is used to determine whether the idle time is long enough to compensate for the additional energy spent in transition. If the idle time interval is too small, the sleep is not actually taken.

3.2 System Specification and It's Properties

The system under consideration consists of a set of n dynamic real-time periodic tasks $T = \{T_1, T_2, ..., T_n\}$ to be executed on a set of m homogeneous processors, using the ER-fair scheduling strategy. We consider discrete time lines and denote time by t ($t \in \mathbb{N}$). Each task T_i in the system is defined by a 3-tuple (s_i, e_i, p_i), where, s_i represents the start/arrival time of the current instance or job of T_i, e_i denotes the execution requirement of each of its jobs and p_i which is referred to as its period, denotes the inter-arrival time between consecutive jobs. The utilization u_i (also called *weight*) which represents

the task's minimum required rate of execution, is given by $u_i = e_i/p_i$.

Processor States: The objective of an ESSM scheduled system is to allow maximal processor suspensions such that the temporal constraints of none of the tasks are ever violated. So, any processor in the system may be in one of two states, *Awake* or *Sleeping*. It is *Awake* when it is in high-power mode ready to execute client tasks. The processor is said to be in *Sleeping* state when it is *Suspended/Shutdown* in low-power mode. One or more processors in the system may transit to *Sleeping* state for a certain interval, say, τ, if and only if the combined compute power of the rest of the processors is sufficient to handle the total workload on the system within this interval τ. Figure 3.1(a) illustrates the two processor states along with their transition events.

Task States: A task in an ESSM scheduled system may be in one of four different states: 1) *Running* (R), 2) *Suspended* (S), 3) *Free* (F), and 4) *Engaged* (E). Out of these, the first two states (*Running* and *Suspended*) are jointly referred to as the *Active* state A ($= R \cup S$) while the other two states ($Free$ and *Engaged*) are denoted as the *Completed* state C ($= F \cup E$) when considered jointly. A task which has already arrived is said to be in *Active* state if it has not completed executing all subtasks of its current instance. An *Active* task is in *Running* state if it is either currently executing on a processor, or it is in the ready queue. An *Active* task may sometimes be procrastinated for a certain interval, say, τ_1, to allow a processor to *suspend* itself. Such a procrastinated task is said to be in *Suspended* state during this interval τ_1. When a task has completed executing all subtasks of its current job and is waiting for its next instance to arrive, it is said to be in *Completed* state. A *Completed* task may also be used to allow a processor to sleep for a certain interval, say, τ_2. The *Completed* task is in *Engaged* state during this interval τ_2 when its individual utilization is being used by a processor to suspend itself. Otherwise, if the *Completed* task has not yet been used to shutdown a processor, it is said to be in $Free$ state. Figure 3.1(b) pictorially represents the different states in which a task can be in.

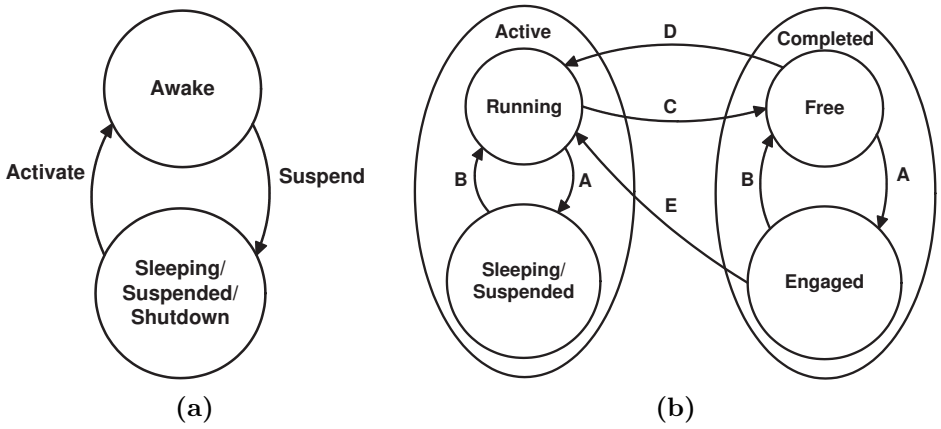


Figure 3.1: *(a) Processor states; (b) Task states: Labels on the arcs denote the state change events; A: Procrastinate, B: End of procrastination duration, C: Finished execution, D and E: End of period/New job begins.*

The *combined utilization* (also referred to as *total utilization* or *workload*) for any given set of tasks, say, $\Lambda = \{T_{\lambda 1}, T_{\lambda 2}, ..., T_{\lambda|\Lambda|}\}$ is defined as the summation of utilizations of the tasks in Λ and is denoted by U_Λ. That is,

$$U_\Lambda \ = \ \sum_{i=1}^{|\Lambda|} u_{\lambda i}. \tag{3.2}$$

The set of periodic tasks (comprising all *Active* and *Completed* tasks) executing at a given time is feasibly schedulable if its total system utilization is at most m, the available number of processors [5], that is,

$$U_{A \cup C} \ \leq \ m. \tag{3.3}$$

The system is said to be underloaded with respect to its total system utilization if $U_{A \cup C} < m$. A new task (say, T_i having utilization u_i) may be accepted by an underloaded ERfair system only if the above feasibility condition (refer Equation 3.3) remains valid even after incorporation of T_i, that is,

$$U_{A \cup C} + u_i \leq m. \tag{3.4}$$

Example 1: Consider a set of six tasks, T_1 $(0, 10, 50)$, T_2 $(0, 12, 60)$, T_3 $(0, 30, 150)$, T_4 $(0, 30, 150)$, T_5 $(0, 20, 50)$ and T_6 $(10, 15, 75)$ to be executed on three unit capacity

processors. So, the total system capacity is (m =) 3. Task T_6 having $s_i = 10$ arrives at time $t = 10$ and all other tasks arrive at $t = 0$. $u_1 = u_2 = u_3 = u_4 = u_6 = 1/5$ and $u_5 = 2/5$. As T_6 has its arrival time at $t = 10$, the total system utilization at $t = 0$ is given by $U_{A \cup C} = U_R = \sum_{i=1}^{5} u_i = 6/5$. The system is feasibly schedulable at $t = 0$ since it is underloaded (that is, $U_{A \cup C}$ (= 6/5) < m (= 3)). A new task T_6 having utilization $u_6 = 1/5$ is accepted at $t = 10$ as it satisfies the feasibility condition in Equation 3.4. Due to the entry of task T_6 at $t = 10$, the updated total utilization $U_{A \cup C}$ (= U_R) becomes 7/5. □

Theorem 3.2.1. *In an underloaded system with total system utilization $U_{A \cup C}$, $\lceil U_{A \cup C} \rceil \leq m$ processors are sufficient to ensure ERfair schedulability.*

Proof. It follows from Equation 3.3 that if $U_{A \cup C} \leq m - i$ ($i \in \mathbb{N}$, $0 \leq i \leq m - 1$), then $m - i$ processors are sufficient to handle the workload $U_{A \cup C}$. Thus, the minimum number of processors that are sufficient to handle a system load $U_{A \cup C}$ is $\lceil U_{A \cup C} \rceil$. □

From Theorem 3.2.1, it may be inferred that as $\lceil U_{A \cup C} \rceil$ processors are sufficient to ensure ERfair schedulability, $m - \lceil U_{A \cup C} \rceil$ processors may safely sleep (remain suspended) perpetually without any possibility of ERfairness violation. However, even when only $\lceil U_{A \cup C} \rceil$ processors are active, the system may still be underloaded if $U_{A \cup C} < \lceil U_{A \cup C} \rceil$. The system is said to be fully loaded when $U_{A \cup C} = \lceil U_{A \cup C} \rceil$.

It may be noted that as there is no spare capacity in a fully loaded system, all tasks will ideally complete at the last time slots of their periods. As the next instances of the tasks arrive at the beginning of their next periods (which immediately follow the end of their previous periods), all tasks continue to remain in their *Running* states forever, with no task ever moving to its *Completed* state. In such a system, $U_{A \cup C}$ remain perpetually same as U_R. However, in an underloaded system, the tasks will actually complete execution of their jobs before the end of their periods. Let a task T_i complete its execution at time, say, t_a (referred to as the finishing time f_i of the current job of T_i) before the end of its period, which occurs at t_b. Hence, T_i will be evicted from the ready queue and will be in the *Completed* state within the interval $[t_a, t_b)$ (the next instance of T_i will arrive at t_b). Now, assuming that all tasks were in the *Running* state at time

t_a and no other tasks changed states within the interval $[t_a, t_b)$, the actual total load of the *Running* tasks (U_R) in the interval $[t_a, t_b)$ is obtained as: $U_R = U_{A \cup C} - u_i$.

Further, it may be observed that if $\lceil U_R \rceil < \lceil U_{A \cup C} \rceil$, a processor may in principle be *suspended* for the interval $[t_a, t_b)$. However, such a processor suspension will not yield any gain in net energy savings (and will rather induce higher net energy dissipation) if the length of the interval $[t_a, t_b)$ is less than TI_{be}, the break-even time (refer Equation 3.1). As we will see later, this is the essential reason why the ESSM algorithm does not suspend processors greedily at all possible opportunities. Instead, it takes a more poised approach by searching for time points within the schedule at which the processor suspension intervals may be locally maximal and actually suspending a processor only if the corresponding sleep is found to be profitable.

In general, at any given instant, an underloaded ERfair system will have some spare capacity beyond U_R, the actual total load, which will be proportionally shared among the currently *Running* tasks in the system. Hence, the effective or actual execution rates of these tasks will be higher than their required execution rates u_i. The effective execution rate (also referred to as *effective weight*) eu_i of each task T_i is then obtained as the up-scaled version of its original utilization as follows:

$$eu_i = min(\frac{u_i}{U_R} \times \lceil U_R \rceil,\ 1). \tag{3.5}$$

Assuming that a system was ERfair at an instant, say ,t_x, in the past and the current time is t, the number of subtasks of execution that a task T_i (whose *lag* value at time t_x is 0 ($lag(T_i, t_x) = 0$; refer Equation 2.1)) should complete in the interval $[t_x, t)$ is given by: $u_i \times (t - t_x)$. However, in an underloaded system, T_i will actually complete $eu_i \times (t - t_x)$ subtasks. So, $oa_i(t) = (eu_i - u_i) \times (t - t_x)$ denotes T_i's overallocation at time t. Therefore, at t, the scheduler can safely suspend T_i for an interval $\lfloor oa_i(t)/u_i \rfloor$ without any possibility of ERfairness violation. We refer to this interval ($\lfloor oa_i(t)/u_i \rfloor$) as the slack of task T_i at t (denoted by $slack_i(t)$; assuming that $slack_i(t_x) = 0$). In general,

$$slack_i(t) = slack_i(t_x) + \lfloor \frac{oa_i(t)}{u_i} \rfloor. \tag{3.6}$$

Example 2: Let us continue with the system and scenario considered in Example 1.

It may be inferred that as $\lceil U_{A \cup C} \rceil = 2$ processors are sufficient to ensure ERfair schedulability, $m - \lceil U_{A \cup C} \rceil = 1$ processor may safely sleep (remain suspended) perpetually without any possibility of ERfairness violation (unless dynamic arrival of tasks at some future time causes $U_{A \cup C}$ to surge above 2; if this happens, then all three processors must be awakened). As system utilization $U_R = 6/5$ and system capacity $\lceil U_{A \cup C} \rceil = 2$ at time $t = 0$, the initial effective execution rates of the *Running* tasks are $eu_1 = eu_2 = eu_3 = eu_4 = 1/3$ and $eu_5 = 2/3$. It may be noted that the five *Running* tasks at $t = 0$ proportionally consume the entire available spare capacity with their combined effective execution rates being $\sum_{i=1}^{5} eu_i = 2$. Executing at these effective rates, the over-allocations acquired by the five *Running* tasks at $t = 10$ are $oa_1(10) = oa_2(10) = oa_3(10) = oa_4(10) = 4/3$ and $oa_5(10) = 8/3$ and the corresponding slacks accumulated by them are $slack_1(10) = slack_2(10) = slack_3(10) = slack_4(10) = slack_5(10) = \lfloor 20/3 \rfloor = 6$. Due to the arrival of task T_6 at $t = 10$, the system utilization U_R increases to 7/5 and the modified effective utilizations of the tasks become $eu_1 = eu_2 = eu_3 = eu_4 = eu_6 = 2/7$ and $eu_5 = 4/7$. □

Theorem 3.2.2. *At a given instant of time, the slack generation rates of each task T_i in the Running set is constant and is given by: $SR_i = \frac{\lceil U_R \rceil - U_R}{U_R}$, where U_R denotes the total load of the Running tasks.*

Proof. (Using step-by-step deduction)

1. Required rate of execution of each task T_i: $u_i = \frac{e_i}{p_i}$
2. Actual or effective rate of execution: $eu_i = \frac{\lceil U_R \rceil}{U_R} \times u_i$ (refer Equation 3.5).
3. So, rate at which T_i gets overallocated: $eu_i - u_i = u_i \times \frac{\lceil U_R \rceil - U_R}{U_R}$
4. Now, as T_i requires to execute at the rate $\frac{e_i}{p_i}$, it may be suspended for $\lfloor \frac{p_i}{e_i} \rfloor$ time slots for each unit of execution overallocation.
5. Hence, for $u_i \times \frac{\lceil U_R \rceil - U_R}{U_R}$ units of over-allocation, T_i may be suspended for $\lfloor u_i \times \frac{\lceil U_R \rceil - U_R}{U_R} \times \frac{p_i}{e_i} \rfloor$ time slots $= \lfloor \frac{\lceil U_R \rceil - U_R}{U_R} \rfloor$ time slots, which is constant for a given system utilization.

6. Thus, slack rates of all *Running* tasks is $SR_i = \frac{\lceil U_R \rceil - U_R}{U_R}$

□

Theorem 3.2.3. *Slack generation rates of tasks T_i in the Completed state is $SR_i = -1$.*

Proof.

1. A task T_i is said to be in the *Completed* state if it has finished the execution of its current instance and is waiting for its next job to arrive at the beginning of its next period.

2. As the slack of a newly arrived job is always zero, the slack of a completed task T_i at any time t is given by $p_i - t$, and it decreases as time progresses, ultimately reducing to 0 at the end of the period p_i.

3. Hence, all tasks in the *Completed* state have a slack generation rate of $SR_i = -1$.

□

From Theorems 3.2.2 and 3.2.3, the general definition of the slack generation rate SR_i of a task T_i may be obtained as:

$$SR_i = \begin{cases} \frac{\lceil U_R \rceil - U_R}{U_R} & \text{if } T_i \in \textit{Running} \\ -1 & \text{if } T_i \in \textit{Suspended} \text{ or } \textit{Completed} \end{cases} \tag{3.7}$$

The definition for the slack of task T_i at time t (refer Equation 3.6) may be rewritten in terms of its slack generation rate SR_i as:

$$slack_i(t) = slack_i(t_x) + \lfloor SR_i \times (t - t_x) \rfloor \tag{3.8}$$

where, t_x denotes a time in the past at which the value of slack is known, and slack generation rate does not change in the interval $[t_x, t)$.

It may be observed from the definition of slack (Equation 3.8) that a task's slack is 0 when it just arrives. As it keeps executing at rate eu_i (Equation 3.5) in underloaded ($U_R < m$) ERfair systems, its slack $slack_i(t)$ also keeps on increasing at a rate $\frac{\lceil U_R \rceil - U_R}{U_R}$ (Theorem 3.2.2) linearly with time until the point (say, f_i) where it finishes execution of its current instance. This is the point where the slack of the task is highest.

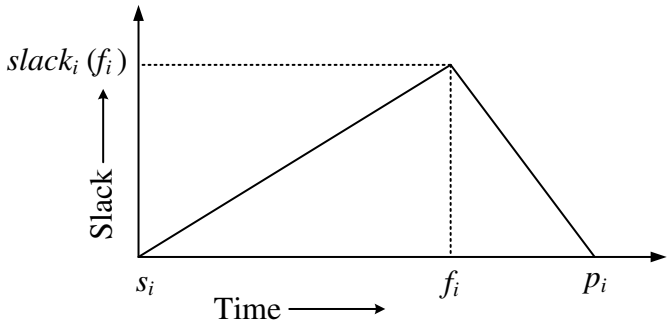


Figure 3.2: *Task T_i's variation of slack over time. s_i, f_i and p_i represent the arrival time, finish time and period of T_i, respectively. $slack_i(f_i)$ denotes the slack generated at time f_i.*

Thereafter, its slack decreases linearly with time at the rate -1 (Theorem 3.2.3) until its deadline/period-end is reached where the slack becomes 0 again. Figure 3.2 gives a pictorial representation of a task's slack variation with time. We call this diagram the *slack-graph* of a task.

It follows from Theorem 3.2.2 that, at any given instant, all rising edges of *Running* tasks have the same slope as the slack generation rate is constant. Similarly, Theorem 3.2.3 says that all falling edges have a slope of $= -1$. So, no two rising edges or two falling edges ever intersect. The slope of the rising edges of all tasks change whenever the number of tasks in *Running* state changes. This happens because whenever such an event occurs, the effective execution rates of all tasks and, hence, all currently running task's slack generation rate changes.

As discussed above, at most $\lceil U_{A \cup C} \rceil$ processors are sufficient to feasibly schedule a set of tasks having total system utilization $U_{A \cup C}$. We also saw that at any time t in an underloaded ERfair system ($U_{A \cup C} < \lceil U_{A \cup C} \rceil$), each task T_i in the task set T acquires a slack, represented as $slack_i(t)$, which denotes the interval of time (starting from t) for which T_i may be safely suspended without any chance of ERfairness violation. Let us assume that we choose a subset of tasks $\Lambda = \{T_{\lambda 1}, T_{\lambda 2}, ..., T_{\lambda |\Lambda|}\}$ from T whose combined utilization ($\sum_{i=1}^{|\Lambda|} u_{\lambda i}$) is φ and minimum slack at time t ($min_{i=1}^{|\Lambda|} slack_{\lambda i}(t)$) is γ. We refer to this minimum slack γ among any given set of tasks Λ as the *Group_Slack* of Λ at t and is denoted by $Group\_Slack(\Lambda, t)$ (thus, here, $Group\_Slack(\Lambda, t) = \gamma$). It

may be noted that the set Λ may be formed by choosing tasks from those in states *Free* or *Running* or both. Now, if $\varphi \geq U_{R\cup F} - \lfloor U_{R\cup F} \rfloor$, then one processor in the system may be safely shut down for an interval γ (after suspending all tasks in Λ for the interval γ) without any possibility of ERfairness violation. (Here, we consider only the combined utilization of the *Running* and *Free* tasks ($U_{R\cup F}$) since the utilization of the *Suspended* and *Engaged* tasks have already been used to suspend processor(s)). The processor transits back to the *Awake* state at the end of this interval as the task having the minimum slack in the *Free* state transits to *Running* state at this time due the arrival of its next instance. A processor that slept at time t may even have to be awakened at any time t_1 before $t+\gamma$ if a new task dynamically arrives at t_1 and the total capacity of the processors awake at t_1 is insufficient to handle the combined workload of the new task along with the existing *Running* tasks.

Without loss of generality, let us again assume that set Λ has been formed by selecting a set $\Lambda 1$ of tasks in *Running* state and $\Lambda 2$ of tasks in *Free* state. That is, $\Lambda = \{\Lambda 1 \cup \Lambda 2\}$ ($\Lambda 1 \cap \Lambda 2 = \varnothing$). As a result of the suspension of the tasks comprising Λ for the interval γ, tasks in $\Lambda 1$ move from *Running* to *Suspended* while those in $\Lambda 2$ move from *Free* to *Engaged* for the time span γ. Due to this modification in the sets of tasks contained in different task states, the value of the system utilization U_R and combined system utilization $U_{R\cup F}$ are updated accordingly.

Another important observation is that a constant overhead referred to as the break-even time TI_{be} is associated with each shutdown. This means that processor suspension for an interval $Group\_Slack(\Lambda, t) = \gamma$ will provide a net gain in terms of energy savings only if $\gamma - TI_{be} \geq 0$. It is clear that in order to minimize the negative impact of TI_{be}, it is better to avoid short sleep durations and instead search for time points within the schedule where the suspension intervals may be locally maximal. We refer to such a time point of maximal sleep as a *viable suspension point.*

Theorem 3.2.4. *Let $\Lambda = \{\Lambda 1 \cup \Lambda 2\}$, ($\Lambda 1 \in$ Running, $\Lambda 2 \in$ Free) denote a subset of tasks from T whose combined utilization is sufficient to allow a processor to suspend itself ($\sum_{i=1}^{|\Lambda|} u_{\lambda i} \geq U_{R\cup F} - \lfloor U_{R\cup F} \rfloor$). Let us further assume a time t at which the condition $Group\_Slack(\Lambda 1, t) < Group\_Slack(\Lambda 2, t)$ is satisfied for Λ. Then, the next viable*

suspension point corresponding to set Λ *will be obtained at the first subsequent time instant* t_p *in the future at which at least one of the following two events occur in the corresponding slack distribution graph:*

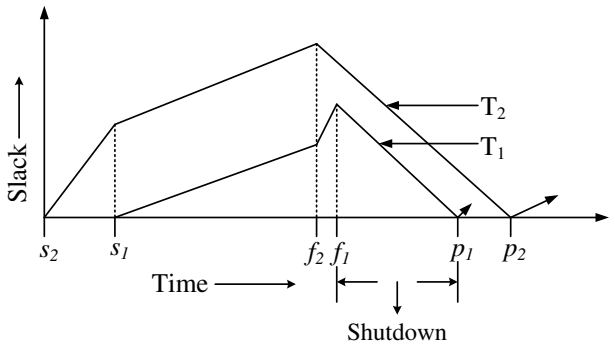
1. *The rising edge of the task having the minimum slack in* $\Lambda 1$ *intersects with the falling edge of the task having the minimum slack in* $\Lambda 2$*.*

2. *The task currently having the minimum slack in* $\Lambda 1$ *completes execution.*

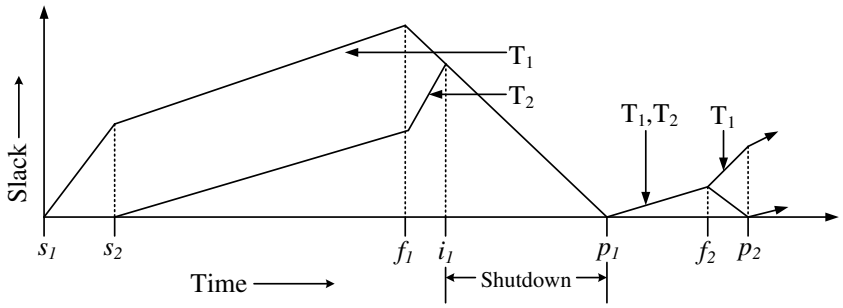
Proof. The set of tasks in $\Lambda 1$ will be represented by parallel rising edges in the slack distribution graph (refer Theorem 3.2.2). Similarly, slacks of tasks in $\Lambda 2$ will be represented by parallel falling edges (Theorem 3.2.3). Here, we assume that there are $Free$ tasks in the system at time t ($\Lambda 2 \neq \varnothing$). At any given time, *Group_Slacks* of $\Lambda 1$ and $\Lambda 2$ will be represented by the minimum ordinate values among the rising and falling edges, respectively. While the *Group_Slack* of $\Lambda 1$ will monotonically increase at a fixed rate (Theorem 3.2.2), the *Group_Slack* of $\Lambda 2$ will monotonically decrease at rate -1 (Theorem 3.2.3). If the lowest rising and falling edges have to intersect at any time, say, t_p (so that condition 1 of the theorem is satisfied), then the minimum ordinate value (and hence the *Group_Slack*) in $\Lambda 1$ must be lower than the minimum ordinate value (and hence the *Group_Slack*) in $\Lambda 2$ before time t_p. Thus, *Group_Slack* of the admissible set Λ ($= min(Group\_Slack(\Lambda 1, t), Group\_Slack(\Lambda 2, t))$) also increases monotonically at rate $\frac{\lceil U_R \rceil - U_R}{U_R}$ before t_p. At t_p, *Group_Slacks* of $\Lambda 1$ and $\Lambda 2$ become equal ($Group\_Slack(\Lambda 1, t_p) = Group\_Slack(\Lambda 2, t_p)$; this may not be exactly true because we consider discrete time lines. On a discrete time line, t_p denotes the first time slot ($> t$) at which $Group\_Slack(\Lambda 1, t_p) \geq Group\_Slack(\Lambda 2, t_p)$). After t_p, *Group_Slack* of $\Lambda 1$ will become higher than $\Lambda 2$'s *Group_Slack*, and hence *Group_Slack* of Λ will decrease monotonically. Hence, a locally maximal sleep interval is obtained at t_p, and so, t_p forms a *viable suspension point.*

If a task in $\Lambda 1$ (say, T_a) completes at some time, say, t_1, before the time t_2 at which the tasks with the minimal slacks in $\Lambda 1$ and $\Lambda 2$ were to intersect, then task T_a is moved from $\Lambda 1$ to $\Lambda 2$ at t_1. *Group_Slacks* of $\Lambda 1$ and $\Lambda 2$ are updated accordingly (if there are no $Free$ tasks in the system at time t ($\Lambda 2 = \varnothing$), then the time t_2 do not exist). If T_a was the task with the minimum ordinate value in $\Lambda 1$ (before completion), then $Group\_Slack(\Lambda 1, t_1)$ will become greater than $Group\_Slack(\Lambda 2, t_2)$ after T_a is moved from $\Lambda 1$ to $\Lambda 2$. Hence,

both condition 1 and condition 2 of the theorem will be simultaneously satisfied at t_1. The next viable suspension point is therefore obtained at $t_p = t_1$. □



(a) *Processor shuts down when task T_1 having lower slack terminates. f_1 denotes the viable suspension point*



(b) *Processor shuts down when the rising edge of task T_2 intersects with the falling edge of task T_1. i_1 denotes the viable suspension point*

Figure 3.3: *Shutdown intervals for a hypothetical admissible task set consisting of two tasks*

Figure 3.3 shows the sleep intervals at viable suspension points for an hypothetical task set consisting of two tasks. While in Figure 3.3(a) shutdown occurs when task T_1 having lower slack terminates, in Figure 3.3(b) shutdown occurs when the rising edge of task T_2 (having lower slack) intersects with the falling edge of task T_1 at i_1. In both cases, the processor transits back to the *Awake* state when the next instance of T_1 starts.

3.3 The ESSM Scheduling Strategy

The ESSM algorithm aims to maximize the sum of the total sleep durations over all processors while simultaneously trying to lower the number of such sleeps because a

constant overhead in terms of both time and energy is associated with each suspension (refer Section 3.1). We now present a brief discussion on the working of ESSM.

3.3.1 ESSM: Algorithm Overview

When a schedule starts, we have a set (say, R) of *Running* tasks having total utilization U_R. All slack values are equal to 0. The *Completed* set (say, C) is empty. Hence, $U_R = U_{R \cup F} = U_{A \cup C}$. As discussed in the previous subsection, ESSM will maintain $m - \lceil U_{A \cup C} \rceil$ processors in the *Sleeping* state. On the rest of the processors, ESSM starts executing tasks in ERfair fashion and all tasks begin to generate slack at the same rate $\frac{\lceil U_R \rceil - U_R}{U_R}$ (Theorem 3.2.2). At any given time t, ESSM searches for the next time point t_p at which either a *Running* task (say, T_a) completes execution or the rising edge of the slack trajectory of a *Running* task (say, T_a) intersects with the falling edge of a $Free$ task (Theorem 3.2.4). This time instant tp at which the possibility of locally maximal suspension intervals is obtained is called the *Earliest Potential Suspension Point* (EPSP). An $EPSP$ becomes a *viable suspension point* if $slack_a(t_p) - TI_{be} \geq 0$. At any *viable suspension point* t_p, ESSM tries to build a set Λ whose weight $\sum_{i=1}^{|\Lambda|} u_{\lambda i}$ is atleast $U_{R \cup F} - \lfloor U_{R \cup F} \rfloor$. If such a set Λ is formed, it is referred to as an *admissible task set*.

ESSM tries to generate an *admissible task set* Λ as follows: First, the subset $\Lambda 2$ of the $Free$ tasks whose slacks are greater than or equal to $Slack_a(t_p)$ is formed (if T_a completed execution at t_p, then it is included in $\Lambda 2$). The sum of utilizations of tasks in $\Lambda 2$ is $U_{\Lambda 2}$ and of those $Free$ tasks not included in $\Lambda 2$ is given by $U_{F \setminus \Lambda 2}$. If $U_{\Lambda 2} \geq U_{R \cup F} - \lfloor U_{R \cup F} \rfloor$ then set $\Lambda 2$ is sufficient to form an *admissible set* Λ. Otherwise, ESSM tries to build a set $\Lambda 1$ using a subset of those tasks from the *Running* set whose slack values are greater than or equal to and closest to $Slack_a(t_p)$ and $U_{\Lambda 1} \geq U_R + U_{F \setminus \Lambda 2} - \lfloor U_R \rfloor$. Although any subset of tasks with slack values greater than or equal to $Slack_a(t_p)$ from the *Running* set could have been used to form the $\Lambda 1$, we choose the tasks with slack values closest to $Slack_a(t_p)$ to allow the slacks of higher slack-valued tasks to grow further and potentially form *admissible sets* with higher *viable slack* values in future. The set of tasks $\Lambda 1 = \{T_{\lambda 1}, T_{\lambda 2}, ..., T_{\lambda |\Lambda 1|}\}$ is characterized as follows:

1. $Group\_Slack(\Lambda 1, t) \geq Slack_a(t_p)$
2. $\sum_{i=1}^{|\Lambda 1|} u_{\lambda i} \geq (U_R + U_{F \setminus \Lambda 2} - \lfloor U_R \rfloor)$
3. $\forall\ i,\ Slack_{\lambda i}(t) \geq Slack_a(t_p)$
4. $\nexists\ i, j$, such that: $T_i \in \Lambda 1$, $T_j \notin \Lambda 1$, $Slack_a(t_p) < Slack_j(t) < Slack_i(t)$ and $\sum_{k=1}^{|\Lambda 1|} u_{\lambda k} - u_i + u_j \geq (U_R + U_{F \setminus \Lambda 2} - \lfloor U_R \rfloor)$
5. $\nexists\ i$, such that: $T_i \in \Lambda 1$ and $\sum_{k=1}^{|\Lambda 1|} u_{\lambda k} - u_i \geq (U_R + U_{F \setminus \Lambda 2} - \lfloor U_R \rfloor)$.

If such an *admissible set* Λ can be formed, then a currently *Awake* processor (say, V_i) is put to sleep. The tasks in $\Lambda 1$ transit to *Suspended* state while those in $\Lambda 2$ transit to the *Engaged* state. At the end of this interval $Slack_a(t_p)$, the slacks of at least one *Engaged* task and zero or more *Suspended* tasks reduces to 0. Hence, Λ ceases to remain admissible, and V_i must transit back to *Awake* state. The instant $t_p + Slack_a(t_p)$ is called the *Earliest Potential Wakeup Point* (EPWP). All *Suspended* and *Engaged* tasks whose slacks reduced to 0 transit to the *Running* state. The remaining *Engaged* tasks transit to the *Free* state.

Extending Continuous Sleep Intervals: The sleep interval for a processor, say, V_i, may be extended if a new admissible set (say, $\Lambda' = \Lambda 1' \cup \Lambda 2'; \Lambda 1' \in R, \Lambda 2' \in F$) may be formed at time $t_{p'}$ $(= t_p + Slack_a(t_p))$ with its *viable suspension point* at $t_{p'}$ itself. This is possible for the admissible set Λ' if $Group\_Slack(\Lambda 1', t_{p'}) \geq Group\_Slack(\Lambda 2', t_{p'})$, where $\Lambda 1'$ and $\Lambda 2'$ denote respectively the *running* and *Free* tasks in Λ'.

ESSM tries to generate such an admissible set Λ' as follows: First, ESSM attempts to build Λ' using only the available *Free* tasks adding sequentially the least slack valued tasks in F (with slack values at least TI_{be}) into the set $\Lambda 2'$. This is only possible if the sum of the utilizations of all the tasks in $\Lambda 2'$ at $t_{p'}$ is at least equal to $U_{R \cup F} - \lfloor U_{R \cup F} \rfloor$ $(U_{\Lambda 2'} \geq U_{R \cup F} - \lfloor U_{R \cup F} \rfloor)$. However, if this condition is not satisfied, ESSM tries to build the subset $\Lambda 1'$ using tasks from the *Running* set whose slack values are greater than or equal to and closest to $Group\_Slack(\Lambda 2', t_{p'})$ and the sum of whose utilizations is greater than or equal to $U_{R \cup F} - \lfloor U_{R \cup F} \rfloor - U_{\Lambda 2'}$.

We now present a theorem (Theorem 3.3.1) to show optimality of the ESSM algorithm.

Theorem 3.3.1. *Tasks in an ESSM scheduled system always satisfy ERfairness.*

Proof. As has been proved in [5], the ERfair scheduling algorithm is optimal and allows 100% resource utilization. Therefore, the ESSM algorithm may also be proved to be optimal if it can be shown to satisfy ERfairness for all tasks at each time instant in the schedule. Like ERfair, ESSM follows the same scheduling strategy of executing the most urgent subtasks (where urgency is related to earliest subtask pseudo-deadlines) at each time slot. ESSM's admission control strategy for accepting/rejecting new tasks (refer Equation 3.4) is also same as ERfair. The only aspect where ESSM deviates from the original ERfair algorithm is the procrastination of task executions to allow processor suspension. However, it may be observed from Theorems 3.2.2 and 3.2.3 that all sleeping tasks are brought back to *Running* state before their slack reduces to zero. Due to this, the *lag* of any task (refer Equation 2.1) can never be higher than zero due to procrastination. Hence, there is no possibility of ERfairness violation due to procrastination of execution. So, ERfairness is always satisfied within the schedule and this guarantees the optimality of ESSM. □

Example 3: Let us continue with the system and scenario considered in Examples 1 and 2. At t=0, the estimated finish times of T_1, T_2, T_3, T_4 and T_5 are $f_1 = 30$, $f_2 = 36$, $f_3 = 90$, $f_4 = 90$, and $f_5 = 30$. Since T_1 and T_5 have the earliest finishing time $f_1 = f_5 = 30$, the next potential suspension point ($EPSP$) is obtained at $t = 30$.

However, a new task T_6 arrives at $t = 10$. System utilization $U_{A \cup C}$ ($= U_R$) increases to 7/5. The new slack generation rate of each task becomes $SR = 3/7$. The effective execution rates change to $eu_1 = eu_2 = eu_3 = eu_4 = eu_6 = 2/7$ and $eu_5 = 4/7$ and finish times become $f_1 = 33$, $f_2 = 40$, $f_3 = 103$, $f_4 = 103$, $f_5 = 33$, and $f_6 = 63$. EPSP also gets updated and scheduled at $t = 33$ (finishing time of T_1 and T_5). Both T_1 and T_5 finish their executions at $t = 33$ and are moved to the set of $Free$ tasks. The slacks of T_1, T_2, T_3, T_4, and T_5 are 17, and that of T_6 is 10. At $t = 33$, $Slack_1(33) = Slack_5(33) = 17$, and $u_1 + u_5 \geq U_{R \cup F} - \lfloor U_{R \cup F} \rfloor$. Therefore, one processor (say, V_1) can be shutdown for the next 17 time slots at $t = 33$ with the next earliest potential wakeup point ($EPWP$)

being $t = 50$. The *Running* tasks at $t = 33$ are T_2, T_3, T_4 and T_6. U_R becomes 4/5 and slack generation rate is $SR = 1/4$. The effective execution rates of T_2, T_3, T_4, and T_6 become 1/4 and their corresponding estimated finish times are $f_2 = 41$, $f_3 = 113$, $f_4 = 113$, and $f_6 = 67$. Therefore, $EPSP = 41$. Figure 3.4 depicts this scenario.

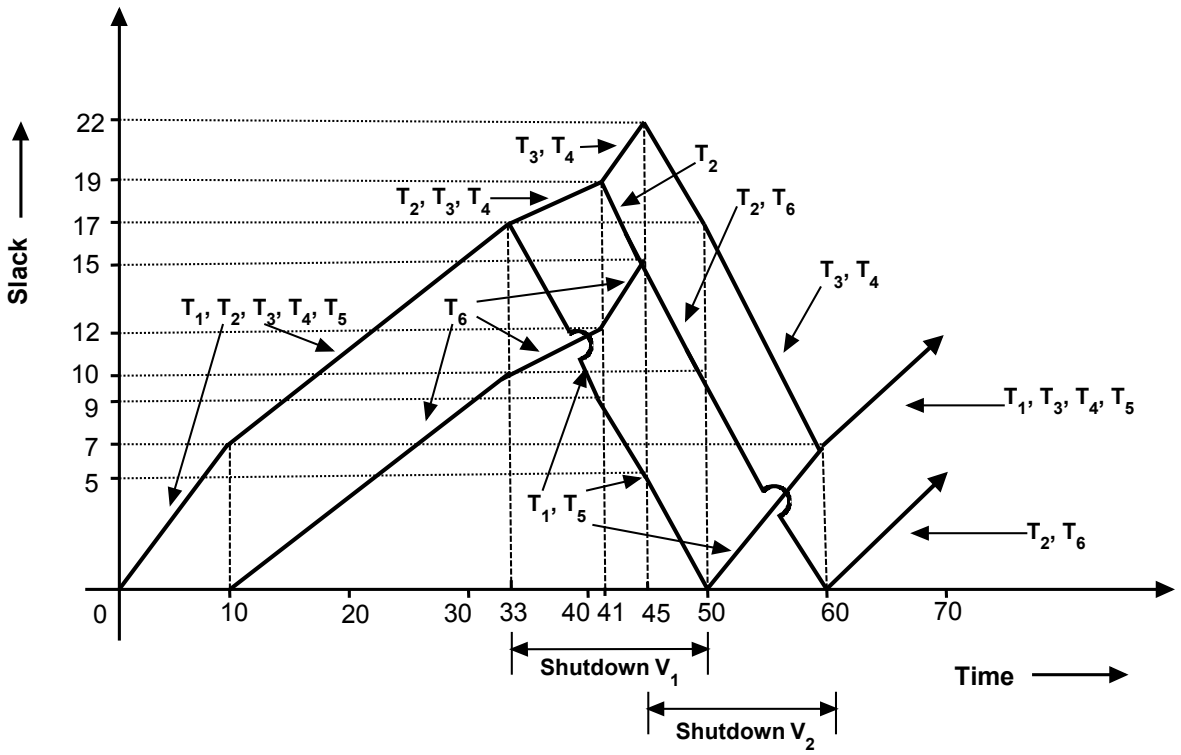


Figure 3.4: *Slack-graph of tasks T_1, T_2, T_3, T_4, T_5, and T_6 in the time interval* $(0, 70]$.

At time $t = 41$, T_2 finishes and is moved to the set of *Free* tasks. The slacks of T_2, T_3, and T_4 are same ($= 19$) and that of T_6 is 12. Since T_2 finishes with $Slack_2(41) = 19$, an admissible set (say, Λ) if formed at $t = 41$ must contain T_2 as one of its elements, and $Group\_Slack(\Lambda, 41)$ must be equal to 19. Hence, Λ can only contain T_2, T_3 and T_4. T_6 cannot be included in Λ since $Group\_Slack(\Lambda, 41) \geq (Slack_6(41) = 12)$. Therefore, $U_\Lambda \ngeq U_{R \cup F} - \lfloor U_{R \cup F} \rfloor$. So, the last remaining processor cannot be suspended at $t = 41$. Due to the completion of task T_2's current instance, U_R decreases to 3/5 and slack generation rate of each *Running* task becomes $SR = 2/3$. The effective execution rates change to $eu_3 = eu_4 = eu_6 = 1/3$ and finish times become $f_3 = 95$, $f_4 = 95$, and $f_6 = 60$.

Since the rising edge of T_6 will intersect with the falling edge of T_2 at $t = 45$, the next $EPSP$ is obtained at time $t = 45$.

At the intersection point $t = 45$, the slacks of T_2, T_3, T_4, and T_6 are 15, 22, 22, and 15. Here, ESSM generates an admissible task set using these tasks with a group slack of 15. Therefore, the last remaining processor V_2 will be suspended between $t = 45$ and $t = 60$. The sleep duration for processor V_1 ends at time $t = 50$ as the next jobs of T_1 and T_5 arrive at this time. At $t = 50$, $U_R = 3/5$ and the slack generation rates of the *Running* tasks T_1 and T_5 become $SR = 2/3$. $eu_1 = 1/3$, $eu_5 = 2/3$, and $f_1 = f_5 = 80$. Therefore, $EPSP = 80$, the finishing time of T_1 and T_5. The next $EPWP$ is obtained at $t = 60$, the wakeup time of V_2. The rest of the schedule continues as shown in Figure 3.4. □

Data Structures: The algorithm primarily uses four data structures, namely, an AVL tree $\mathcal{A}$ of the *Running* tasks ordered in terms of their pseudo-deadlines and three different lists of tasks each ordered in non-decreasing fashion in terms of available slack values. The first list $\mathcal{H}$ consists of the $Free$ tasks. Lists L_i hold the *Suspended* and *Engaged* tasks corresponding to each *Sleeping* processor V_i. The indices of the *Running* tasks at a given time are maintained in another list $\mathcal{LL}$ $\{ll_1, ll_2, ..., ll_z\}$.

3.3.2 Detailed Algorithm

The ESSM algorithm consists of four functions. The main function *ESSM* (Algorithm 1) carries out the overall scheduling. It calls function *Sleep-If-Admissible* (Algorithm 2) to determine if an *admissible set* may be formed at each Earliest Potential Suspension Point $EPSP$ and shut down the processor if such a set exists. *ESSM* calls function *Sleep-If-Extendable* (Algorithm 4) at each Earliest Potential Wakeup Point $EPWP$ to determine if the sleep interval of a given processor may be extended by forming a new admissible task set that has its viable suspension point at $EPWP$ itself. *Sleep-If-Extendable* wakes up the processor if such an extension is not possible. Finally, *ESSM* calls function *Next Scheduled Event* (Algorithm 3) to find out the time slot at which the earlier of the immediately next $EPSP$ or $EPWP$ is expected to occur.

ALGORITHM 1: *Algorithm ESSM*

```
1  {Given: A set of n tasks and m processors. Let n' denote the number of tasks in
   Running state, m' denote the number of Awake processors, U_R denote the sum of
   utilizations of these n' tasks at any instant, and t_e denote the current time slot.};
2  Initialize: m' = ⌈U_R⌉ {Calculate the number of processors sufficient to handle workload
   U_R};
3  for Each time slot t do
4      Execute the most urgent m' tasks from the set of Running tasks in ERfair fashion;
5      if t_e == EPSP {Current time slot is an Earliest Potential Suspension Point} then
6          Call function Sleep-If-Admissible {Algorithm 2};
7          if Admissible set was formed and a processor suspended using Algorithm 2 then
8              Call function Next Scheduled Event {Algorithm 3};
9      else if t_e == EPWP {Current time slot is an Earliest Potential Wakeup Point}
       then
10         Call function Sleep-If-Extendable {Algorithm 4};
11         if Processor must wakeup at the current time then
12             Call function Next Scheduled Event {Algorithm 3};
13     else if A new task or the next instance of an existing task T_a has arrived then
14         Add T_a to the set of Running tasks; Update system weight U_R;
15         if u_a + U_R > m' {System has become overloaded with the inclusion of T_a} then
16             Wake up a sleeping processor;
17         Call function Next Scheduled Event {Algorithm 3};
```

ALGORITHM 2: *Function Sleep-If-Admissible*

{This function is called either when a task T_a finishes, or when T_a's slack curve intersects with that of a $Free$ task at current time slot t_p. t_p forms a *Potential suspension Point.*};
if $Slack_a(t_p) < TI_{be}$ {TI_{be} *denotes break-even time*} **then**
    Exit {An admissible set cannot be formed at t_p};
if T_a *completed execution at* t_p **then**
    Move T_a from *Running* to *Free* state;
Form a subset Λ_{tmp} of $Free$ tasks whose slack values are greater than or equal to $Slack_a(t_p)$;
{The following *While* loop attempts to form a set $\Lambda 2$ of $Free$ tasks from Λ_{tmp}};
while *(*$U_{\Lambda 2} < U_{R \cup F} - \lfloor U_{R \cup F} \rfloor$*)* ***OR*** *(*$\Lambda_{tmp} == \emptyset$*)* **do**
    Move the least slack valued task in Λ_{tmp} to $\Lambda 2$;
if $U_{\Lambda 2} \geq U_{R \cup F} - \lfloor U_{R \cup F} \rfloor$ **then**
    Shutdown processor V_i;
    $Potential - Wakeup - Point_i = t_p + Slack_a(t_p)$ {Potential wakeup time of processor V_i};
else
    Find $U_{F \setminus \Lambda 2}$ {sum of utilizations of those $Free$ tasks for whom *slack value* $< Slack_a(t_p)$};
    Build a subset $\Lambda 1$ of those tasks from the *Running* set whose slack values are greater than or equal to and closest to $Slack_a(t_p)$ and the sum of whose utilizations ($U_{\Lambda 1}$) is greater than or equal to $U_R + U_{F \setminus \Lambda 2} - \lfloor U_R \rfloor$;
    if $U_{\Lambda 1} < (U_R + U_{F \setminus \Lambda 2} - \lfloor U_R \rfloor)$ **then**
        Exit {An admissible set cannot be formed at t_p};
    else
        Shutdown processor V_i; $Potential - Wakeup - Point_i = t_p + Slack_a(t_p)$;

ALGORITHM 3: *Function Next Scheduled Event*

Update U_R, sum of utilizations, along with *slack rates* and *finish times* of all *Running* tasks;
Let eft and f_{eft} denote the index of this earliest finishing *Running* task and its finish time;
if *Free set is not empty* **then**
    Find t_{cut}, the earliest intersection point between slack curves of a *Running* and *Free* task;
else
    $t_{cut} = \infty$;
if *($t_{cut} < EPSP$) **OR** ($f_{eft} < EPSP$)* **then**
    $EPSP = \mathbf{min}(t_{cut}, f_{eft})$;
$EPWP = \forall Processors V_i, \mathbf{min}(Potential - Wakeup - Point_i)$;
$NSE = \mathbf{min}(EPSP, EPWP)$;

ALGORITHM 4: *Function Sleep-If-Extendable*

{If an admissible set Λ' is possible at t_p, the $Potential - wakeup - point$ of processor V_i, then this function continues V_i in suspended mode. Otherwise, it wakes up V_i.};
Update U_R and $U_{R \cup F}$;
Form a subset $\Lambda 2'$ by sequentially adding least slack valued $Free$ tasks until $U_{\Lambda 2'}$ becomes greater than or equal to $U_{R \cup F} - \lfloor U_{R \cup F} \rfloor$;
if $U_{\Lambda 2'} \geq U_{R \cup F} - \lfloor U_{R \cup F} \rfloor$ **then**
    $Potential - Wakeup - Point_i = t_p + Group\_Slack(\Lambda 2', t_p)$; exit {$V_i$'s sleep is extended};
else
    Form the subset $\Lambda 1'$ of the *Running* tasks whose slack values are greater than or equal to and closest to $Group\_Slack(\Lambda 2', t_p)$ and $U_{\Lambda 1'} \geq U_{R \cup F} - \lfloor U_{R \cup F} \rfloor - U_{\Lambda 2'}$;
    if *Such a set $\Lambda 1'$ may be formed* **then**
        $Potential - Wakeup - Point_i = t_p + Group\_Slack(\Lambda 2', t_p)$; exit {$V_i$'s sleep is extended};
    else
        Wake up processor V_i; exit;

3.3.3 Complexity Analysis

Lemma 3.3.2. *Complexity of function Sleep-If-Admissible (Algorithm 2) is* $O(n \cdot lg(n))$.

Proof.

1. Steps 5 in function *Sleep-If-Admissible* requires extractions and insertions on an AVL tree and list H. All are $O(lg(n))$ time operations.
2. Steps 6, 8, and 14 are all $O(n)$ list operations in the worst case.
3. Step 15 involves the transfer of a set of n tasks in the worst case from AVL tree $\mathcal{A}$ to list L_i. This takes $O(n \cdot lg(n))$ time.
4. All other steps take constant time. Therefore, function *Sleep-If-Admissible* has an overall complexity of $O(n \cdot lg(n))$.

□

Lemma 3.3.3. *Complexity of Sleep-If-Extendable (Algorithm 4) is* $O(n \cdot lg(n))$.

Proof.

1. Steps 2 and 3 involves $O(n)$ list operations in the worst case.
2. Step 7 involves the transfer of a set of n tasks in the worst case from AVL tree $\mathcal{A}$ to list L_i. This takes $O(n \cdot lg(n))$ time.
3. The other steps take constant time. Hence, function *Sleep-If-Extendable* has an overall complexity of $O(n \cdot lg(n))$.

□

Lemma 3.3.4. *Complexity of Next Scheduled Event (Algorithm 3) is of* $O(n)$ $(m \leq n)$.

Proof.

1. Updation of the sum of utilizations of the *Running* tasks in step 1 takes $O(n)$ time. Calculation of Slack rate takes constant time (Equation 3.7) while the finish times of all tasks may be calculated in $O(n)$ time.

2. Determining the earliest finishing task (T_{eft}) and its completion time (f_{eft}) also takes $O(n)$ time in the worst case.

3. Step 9 finds the earliest wakeup point among all the sleeping processors and takes time proportional to $O(m)$.

4. All other steps in the function take constant time to execute and the function as a whole runs in $O\{min(m, n)\}$ time. Assuming, $m \leq n$, its complexity is $O(n)$.

□

Theorem 3.3.5. *The amortized scheduling complexity C per time slot of ESSM (Algorithm 1) is as follows:*

$$C = \begin{cases} O(m \cdot lg(n)) & ; \quad E \geq n \\ O(\frac{m \cdot n \cdot lg(n)}{E}) & ; \quad otherwise. \end{cases} \tag{3.9}$$

where, n denotes the number of tasks, m the number of processors and E the average execution requirement of the n tasks.

Proof.

1. Steps 4 of algorithm *ESSM* (algorithm 1) takes $O(m \cdot lg(n))$ time since it involves m extractions from and insertions into the AVL tree of *Running* tasks.

2. Steps 5 to 17 involves the occurrence of a set of events such as arrival of a task, potential suspension Point ($EPSP$), and potential wakeup point ($EPWP$). An $EPSP$ may occur only when either: (i) a *Running* task completes execution or (ii) the rising edge of the slack trajectory of a *Running* task intersects with the falling edge of a $Free$ task. A Wakeup occurs every time a shutdown occurs. Now, clearly, these events can occur only a constant number of times (c_i) during the lifetime of each task instance T_i. Hence, the total number of iterations possible for these events during the lifetime of the n tasks is: $\sum_{i=1}^{n} c_i$.

3. Steps 6 and 10 involve calls to *Sleep-If-Admissible* and *Sleep-If-Extendable*, respectively, each of which takes $O(n \cdot lg(n))$ time (Lemmas 3.3.2 and 3.3.3).

4. Steps 8, 12 and 17 calls *Next Scheduled Event* and takes $O(n)$ time (Lemma 3.3.4).

5. Therefore, the total complexity of these events over all the n task instances become $\sum_{i=1}^{n} c_i \cdot O(n \cdot lg(n)) = O(n^2 \cdot lg(n))$.

6. Each task T_i has an execution requirement e_i. So, the time required to execute n tasks on m processors become $\frac{\sum_{i=1}^{n} e_i}{m}$.

7. If E denotes the average execution time of the n tasks, $\frac{\sum_{i=1}^{n} e_i}{m} = \frac{n \cdot E}{m}$.

8. Hence, the average complexity of executing the above events per time slot is: $\frac{m \cdot O(n^2 \cdot lg(n))}{n \cdot E} = O(\frac{m \cdot n \cdot lg(n)}{E})$

9. Therefore, steps 5 to 17 have a complexity of $O(\frac{m \cdot n \cdot lg(n)}{E})$ per time slot.

10. The overall complexity of the algorithm per time slot is $O(m \cdot lg(n))$ when $E \geq n$ and is equal to $O(\frac{m \cdot n \cdot lg(n)}{E})$ otherwise.

□

The above result shows that the complexity of the ESSM algorithm becomes $O(m \cdot n \cdot lg(n))$ when the number of tasks n is larger compared to the average execution requirement of the tasks ($n > E$). However, when the task set size is smaller than E (as in the case of most embedded systems), the complexity reduces to $O(m \cdot lg(n))$, which is same as the complexity of the typical proportional fair algorithms.

3.4 Experiments and Results

We have experimentally evaluated the performance of our algorithm and compared it against two other optimal algorithms namely, *Basic-ERfair*, a strictly proportional fair algorithm and *Boundary Fair (Bfair)*, a semi-fair algorithm [119]. The evaluation methodology is based on simulation studies using an experimental framework that is described in the next subsection. An overview of both *Basic-ERfair* and *Boundary Fair* algorithms are as follows:

Basic-ERfair: Given a set of m processors $\{V_1, V_2, ..., V_m\}$ and n ($\geq m$) tasks, *Basic-ERfair* chooses the m most urgent tasks from a priority queue at each time slot and allocates processors to these tasks in the order in which they have been extracted from

the priority queue. Basic-ERfair suspends processors only when the number of *Running* tasks $|\mathcal{R}|$ within the system becomes less than the number of processors m. Basic-ERfair maintains $m - |\mathcal{R}|$ processors in the *Sleeping* state whenever $|\mathcal{R}| < m$.

Boundary Fair (Bfair): Bfair partitions time into slices, demarcated by the arrivals and departures of all the jobs in the system. Within a time slice, each task is allocated a workload equal to its proportional fair share and assigned to one or more processors for scheduling. Tasks allocated to a processor within a time slice in an underloaded Bfair scheduled system execute in an EDF-like fashion starting from the beginning of the slice and complete their execution before the end of the time slice is reached. After completion of execution of the allocated task shares, each processor idles up to the end of the current time slice. In our experiments, it has been assumed that the Bfair scheduled system suspends its processors during such idle intervals, provided the duration of these intervals are at least equal to the break-even time TI_{be}.

3.4.1 Experimental Setup

The experimentation framework used is as follows: The datasets consist of randomly generated hypothetical periodic tasks whose utilizations ($\frac{e_i}{p_i}$) and execution periods (p_i) have been taken from normal distributions. Given the number of tasks to be generated (n) and the total utilization of the n tasks (U), the task utilizations have been generated from a distribution with standard deviation (σ) = 0.1 and mean (μ) = U/n. The summation of utilizations of these generated tasks is not constant. However, making the summation of utilizations constant helps in the comparison of the algorithms. Therefore, the utilizations have been scaled uniformly to make the cumulative utilization of each task set constant and equal to U. Different types of datasets have been generated by setting distinct values for the following parameters:

1. *Execution requirements* e_i: Task execution times are generated from normal distributions with different values of mean (μ)-standard deviation (σ) pairs. Four different (μ, σ) combinations $(100, 10)$, $(200, 20)$, $(300, 30)$ and $(400, 40)$ were used.

2. *Task set size n*: The number of tasks have been varied between 20 and 100.

3. *Number of processors m*: Systems consisting of 2 to 20 processors have been used.

4. *Workload*: Seven different workloads were considered; we have considered cases when the processor is 70%, 75%, 80%, 85%, 90%, 95%, or 100% loaded.

5. *Break-even time* TI_{be}: In order to study the effect of break-even time, it is varied between 1 and 10 ms.

The length of a time slot is 1 ms. The total schedule length is $100,000$ time slots.

3.4.2 Results: ESSM vs. Basic-ERfair

We have measured the average of the total length of profitable suspension intervals (suspension intervals greater than break-even time TI_{be}) over the entire schedule length for both *ESSM* and *Basic-ERfair* algorithms running them on 100 different instances of each dataset type. To estimate the actual energy savings obtained, we have deducted TI_{be} from each profitable sleep interval. Using these values, we have found the shutdown ratio SDR such that the total sleep duration by using *ESSM* is SDR times the total sleep duration when *Basic-ERfair* is used. Table 3.1 summarizes the results obtained for ESSM and Basic-ERfair on eight processor systems.

From Table 3.1, the following important observations and inferences may be made: For a given number of tasks and processors, Basic-ERfair always produces the same total length of sleep intervals. Moreover, at 95% workload, it is not able to produce any sleep. This is because Basic-ERfair suspends processors only when the number of *Running* tasks $|\mathcal{R}|$ becomes less than the number of processors m. Basic-ERfair maintains $m-|\mathcal{R}|$ processors in *Sleeping* state whenever $|\mathcal{R}| < m$, which contributes to the sleep durations obtained in Basic-ERfair. Beyond this, it fails to extract any further sleep due to the absence of any procrastination mechanism. Given m processors, when the workload rises beyond a certain percentage, $|\mathcal{R}|$ becomes greater than $m-1$, and almost no sleep is obtained, as all available processors remain active throughout the schedule length. The shutdown ratio SDR increases with increase in system load percentage and mean

Table 3.1: *Sleep Duration and Shutdown Ratio: ESSM vs. Basic-ERfair-8 processor system*

U	E	n=20			n=30			n=40		
		Basic-ERfair	**ESSM**	**SDR**	**Basic-ERfair**	**ESSM**	**SDR**	**Basic-ERfair**	**ESSM**	**SDR**
75%	100	199996	230794	1.15399	199996	215554	1.07779	199996	208018	1.04011
	200	199996	235286	1.17645	199996	222726	1.11365	199996	215287	1.07645
	300	199996	236299	1.18151	199996	225023	1.12513	199996	218005	1.09004
	400	199996	236543	1.18273	199996	226196	1.13100	199996	220949	1.10476
85%	100	99998	133424	1.33426	99998	116476	1.16478	99998	108671	1.08673
	200	99998	139383	1.39385	99998	124543	1.24545	99998	114950	1.14952
	300	99998	140280	1.40282	99998	127313	1.27315	99998	118489	1.18491
	400	99998	140635	1.40637	99998	128642	1.28644	99998	121686	1.21688
95%	100	0	25626	—	0	11241	—	0	5325	—
	200	0	33461	—	0	18729	—	0	10104	—
	300	0	35554	—	0	21746	—	0	12972	—
	400	0	37302	—	0	23288	—	0	15433	—
n: Total number of tasks; U: Total system load percentage; E: Mean execution time; SDR: Shutdown ratio										
Fixed parameters:- Break-even time: 2; #processors: 8										

execution time. This is due to the fact that as average available task slacks reduce with rise in system load, procrastination in ESSM helps to extend total sleep durations by identifying all profitable locally maximal viable suspension points and put the processor to sleep at least for a period equal to its corresponding viable slack. SDR, however, exhibits a slow decrease with an increase in the number of tasks (for a given system load, mean execution time, and number of processors). This is because a higher number of dynamic tasks with arbitrary arrival times imply a drop in the probability of obtaining viable suspension points with high $Group\_Slack$ values.

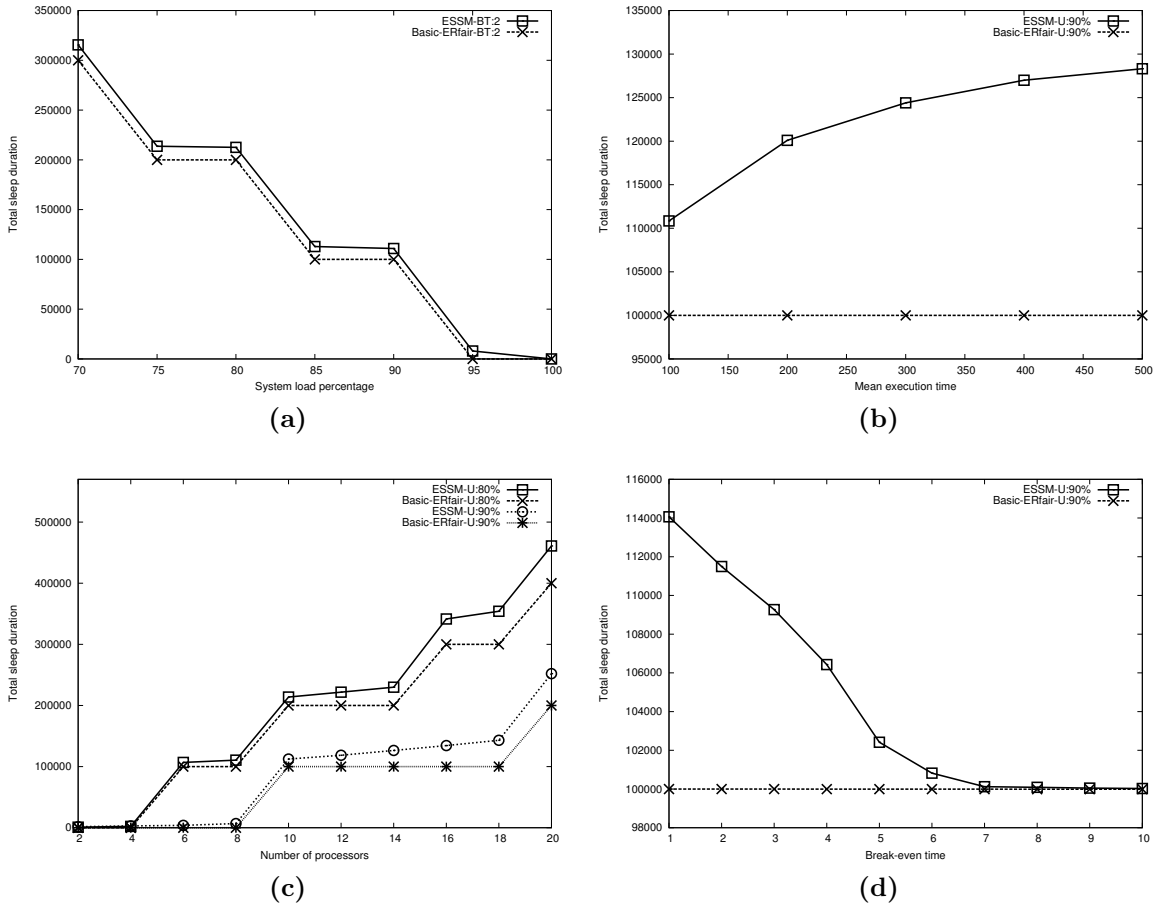


Figure 3.5: *(a) 40 Tasks, 10 Processors, 2 ms break-even time (BT); (b) 40 Tasks, 10 Processors, 90% System Load (U), 2 ms break-even time (BT); (c) 40 Tasks, 80% or 90% System Load (U), 2 ms break-even time (BT); (d) 40 Tasks, 10 Processors, 90% System Load (U).*

Figure 3.5(a) compares the sleep durations achieved by ESSM and Basic-ERfair as the system load varies between 70% and 100% (other parameters remaining constant). As is obvious, the total sleep durations for both approaches progressively decrease with increasing workload due to a dearth in available slacks. However, ESSM is seen to outperform Basic-ERfair by a significant margin almost throughout. An important observation here is the stepwise nature of the plots. The reason for this phenomenon lies in the maximum number of processors required to handle a given workload. For example, with 10 processors between 70% and 80% system load, 2 processors may be perpetually maintained in sleep state. However, under higher loads between 80% and 90%, only 1 processor may be put to sleep.

The variation of sleep time with average execution time is shown in Figure 3.5(b). For a given system load (90%), increase in mean execution time also allows a corresponding increase in the periods of tasks. This in turn enhances the possibility of higher *Group_Slack* for eligible admissible sets at viable suspension points. Therefore, overall sleep times may be seen to increase. On the other hand, variations in execution times do not show any significant changes in sleep times with Basic-ERfair. Figure 3.5(c) shows the effect of variation in the number of processors on average sleep durations, with other parameters remaining constant. It may be observed that total sleep times increase with a higher number of processors. As in Figure 3.5(a), the plots in Figure 3.5(c) also exhibit a stepwise nature due to similar reasons.

In order to study the effect of TI_{be}, the number of processors and tasks are fixed as 10 and 40, respectively. Figure 3.5(d) shows the change in total sleep durations with varying break-even times. It may be observed that for ESSM, the total shutdown length decreases with increasing TI_{be} values. This is because higher the value of TI_{be}, lower becomes the total number of profitable sleep intervals. Variation in TI_{be}, however, do not have any effect on the sleep durations of Basic-ERfair.

3.4.3 Results: ESSM vs. Bfair

We now compare the performance of *ESSM* with a semi-fair optimal algorithm, *Boundary Fair (Bfair)* [119]. The evaluation has been conducted on similar lines (as the

comparison with *Basic-ERfair*) by measuring the average of the total length of profitable suspension intervals (suspension intervals greater than 1 ms; assuming break-even time $TI_{be} = 1$ ms) over the entire schedule length for both *ESSM* and *Bfair*. The shutdown lengths obtained for both ESSM and Bfair incorporates overheads due to context switches. Here, we assume the delay corresponding to a single context switch to be 5.24 μs, which is the actual average context switch overhead on a 24-core Intel Xeon L7455 system under typical workloads [17]. Figure 3.6(a) shows plots for the average context switch overhead (in μs) per processor per time slot incurred by ESSM and Bfair. For a given simulation run, this value is obtained by first finding the average number of context switches per processor per time slot and then multiplying it with the cost of a single context switch (5.24μs). It may be observed that with an EDF-like semi-partitioned scheduling strategy within time slices, the overhead for Bfair varies only slightly from about 1.1 to 1.3μs as the system load increases from 70% to 100% (for 40 tasks and 4 processors). In comparison, with a proportional fair global scheduling policy, ESSM's overhead increases from ~ 1.9 to $\sim 3.0\mu$s.

In Figure 3.6(b), we show the total sleep durations (in ms) achieved by ESSM and Bfair over a schedule length of 100,000 time slots on a 10-processor system running 40 tasks with the average task period size being 400 time slots. With an increase in system load, total sleep durations for both approaches progressively decrease. However, Bfair consistently achieves better total sleep durations throughout. This may be attributed to two major reasons: (i) As discussed above, Bfair incurs lower context switch overheads, allowing it to obtain higher overall slack times compared to ESSM. (ii) As Bfair has a non-fair EDF-like scheduling policy within time slices, the total slack in any processor gets accumulated as a continuous idle interval towards the end of every time slice. In contrast, the average size of a continuous idle interval for ESSM is typically smaller due to its necessity to maintain strict fairness at each time slot. Thus, the possibility of profitable and longer slack durations is higher for Bfair than in ESSM.

However, it may be observed from the figure that ESSM performs fairly and is not drastically outperformed by Bfair, especially on the closely coupled multi-core platform

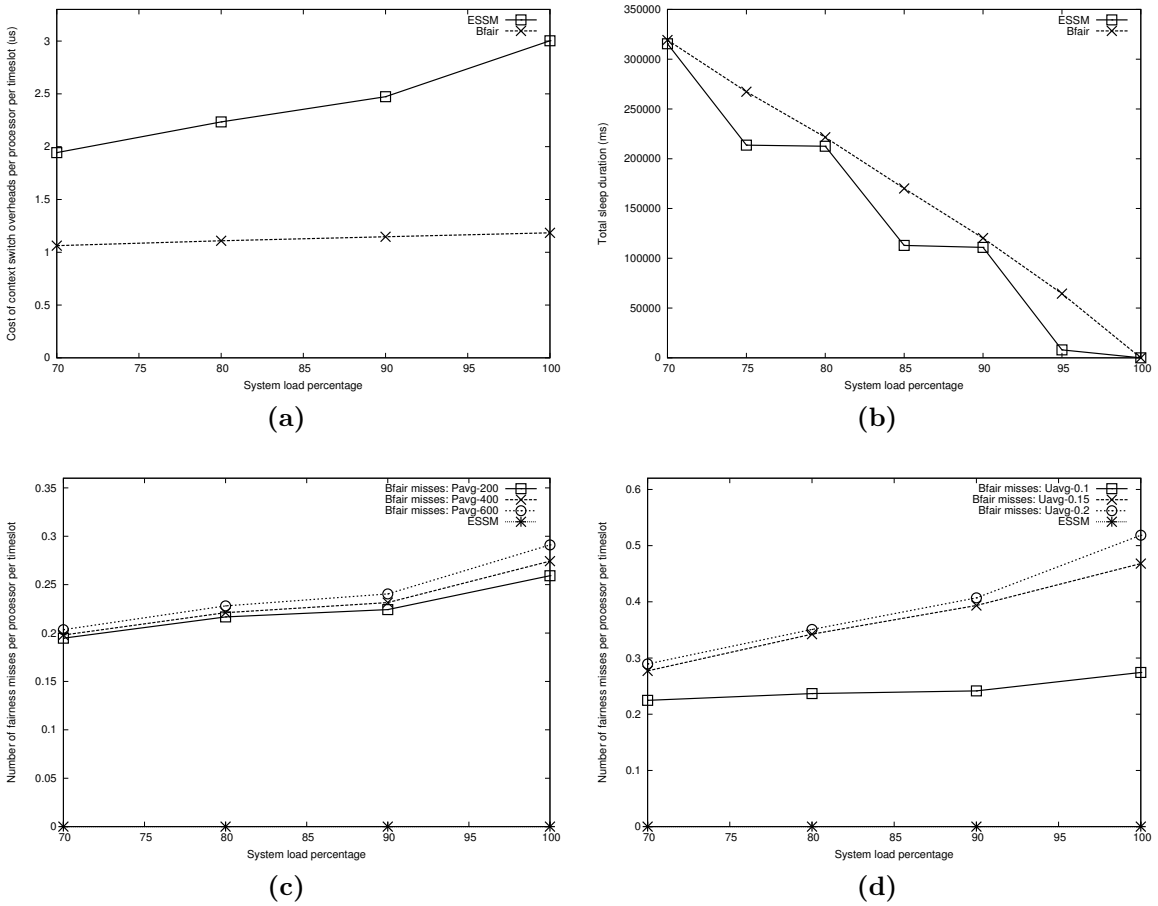


Figure 3.6: *(a) Variation in the cost of context switch overheads obtained by ESSM and Bfair: 40 Tasks, 4 Processors; (b) Total sleep durations achieved by ESSM and Bfair: 40 Tasks, 10 Processors, 1 ms break-even time; (c) & (d) Fairness results for ESSM and Bfair: 4 Processors, 1 ms break-even time. P_{avg} denotes average task period size (in time slots) and U_{avg} denotes average individual task utilization.*

Intel Xeon L7455. Such fair performance may be partially attributed to a simple context switch aware task-to-processor assignment technique incorporated in ESSM. *At each time slot, ESSM assigns the m (m denotes number of processors) most urgent tasks to the processors in two rounds. In the first round, ESSM tries to allocate each of the m tasks to that processor where it executed the last time it was assigned. The unallocated tasks after this round are assigned to the remaining processors in the second round.*

A last observation in Figure 3.6(b) is the stepwise nature of the plot for ESSM. As

discussed in the previous subsection, this is due to variations in the approximate number of processors required to handle a given workload. For example, with 10 processors, and between 70% and 80% system load, two processors may be maintained in the sleep state. However, for loads between 80% and 90%, only one processor may be put to sleep.

However, although Bfair exhibits better energy-awareness properties with respect to ESSM, its fairness properties are far poorer. The results in Figures 3.6(c) and 3.6(d) show that the degree of unfairness (in terms of the number of pseudo-deadline misses per processor per time slot) suffered by Bfair is directly proportional to average task period lengths and average individual task weights. It may be noted that ESSM produces zero fairness deviations in all cases and remains perfectly fair. Figure 3.6(c) shows that even with its lower context switch overheads, Bfair suffers ~ 0.2 ERfairness violations/misses per processor per time slot on a 70% loaded system. For a time slot size of 1 ms, this is equivalent to about 200 ERfairness misses per processor per one second. The fairness miss rate becomes higher as system load increases.

It may be further observed from Figure 3.6(c) that, especially at higher system loads, Bfair's fairness degrades significantly as average task period lengths increase from 200 to 600 time slots. This is due to the fact that the average duration of time slices in Bfair is directly proportional to average task period lengths. As Bfair is guaranteed to be perfectly fair only at time slice boundaries (hence its name Boundary Fair) with unrestricted fairness deviation within time slices, the degree of unfairness increases as the average length of task periods become higher. Plots in Figure 3.6(d) reveal that fairness of Bfair degrades as the average individual task utilization increase from 0.1 to 0.2. This is primarily because, as the average individual task utilization increases, mean task share sizes also increase (task shares are specified in terms of the number of time slots of execution that a task should complete within a time slice to be proportionally fair at time slice boundaries). As Bfair executes these task shares in an EDF-like fashion and completely ignores proportional fairness in the execution progress of tasks within a time slice, higher the average task share sizes, higher becomes the fairness deviation. Thus, although semi-fair algorithms like Boundary Fair, which trade-off fairness in order

to reduce context switches show better energy-awareness properties, they cannot provide strict fairness guarantees as ensured by ESSM.

3.5 Summary

In this chapter, we presented a novel energy-efficient scheduling strategy that attempts to minimize static energy consumption in a symmetric multiprocessor system. The proposed technique takes advantage of higher execution rates of tasks in underloaded ERfair systems and uses a procrastination scheme to search for time points within the schedule where suspension intervals are locally maximal. We have designed, implemented, and evaluated the ESSM algorithm and proved the feasibility of this scheme. The simulation-based experimental results are promising. In the next chapter, we present an efficient fault-tolerant design strategy for real-time safety-critical systems having homogeneous multiprocessors as the computing platform.

Chapter 4

Fault-tolerant Fair Scheduling on Real-time Multiprocessor Systems

In the previous chapter, we have considered the energy-aware scheduling of real-time applications executing on a homogeneous multiprocessor system, and have assumed the underlying hardware computing platform to be fault-free. However, processing platforms are subject to a variety of faults. Such faults are broadly classified into either permanent or transient [6]. Permanent processor faults are irrecoverable and do not go away with time. On the other hand, transient faults are short-lived (momentary) and their effect goes away after some time. Therefore, apart from guaranteeing the timely execution of tasks in a resource-constrained environment, ensuring proper functioning of the system even in the presence of faults (i.e., fault tolerance) has currently become a design constraint of paramount importance.

In this chapter, we present a semi-partitioned fair fault-tolerant scheduling strategy for real-time homogeneous multiprocessor systems containing cold-standby spares. The chapter first describes the system model under consideration. Then, we present our proposed fault-tolerant scheduling strategy to effectively handle permanent processor faults in the system. Later, the chapter discusses important experimental results which highlight the performance of the proposed fault-tolerant scheduling scheme under various scenarios. Finally, the chapter concludes by presenting a case study using an automated flight control system to illustrate the applicability of the proposed fault recovery mechanism in real world scenarios.

4.1 System Model and Problem Formulation

We consider a real-time multiprocessor system consisting of a set of n periodic tasks $T = \{T_1, T_2, ..., T_n\}$, to be executed on a set of m homogeneous processors $V = \{V_1, V_2, ..., V_m\}$. We assume a discrete time line where the interval $[t, t+1)$ is referred to as a time slot t ($t \in \mathbb{N}$). Each task T_i in set T is defined by a 3-tuple (e_i, p_i, cr_i), where, e_i denotes the *Worst Case Execution Time* (WCET) *requirement* of each instance/job of T_i, p_i denotes the fixed inter-arrival time between consecutive instances (referred to as *period*) and cr_i measures the relative importance of T_i with respect to other tasks (denoted as the *criticality level* of T_i; cr_i takes an integer value in the range $[1, 100]$). Each task T_i is associated with a *utilization* (also called *weight*), denoted as u_i, which is defined as the ratio of its execution requirement (e_i) and period (p_i). Each processor V_j ($\in V$) has unit capacity.

In the nominal mode of operation, the system periodically checks for a processor fault, every t_p time slots. There is a cold standby processor which is activated if and when there is a permanent processor fault. The standby processor requires a finite time, called *recovery time* t_r to become operational. Let, t_i be the time instant in the past at which the system last checked for a fault. Let, $F_{OT}(= t_i + t_f)$, be the instant at which the fault actually occurs before the next periodic check at $F_{DT}(= t_i + t_p)$ and so, the system recovers at the instant $F_{RT} = F_{DT} + t_r$. We assume that the system can handle at most one permanent processor fault at any given time; no further faults are assumed to occur during the recovery period $[F_{DT}, F_{RT})$.

Problem Formulation: Given a set of n real-time periodic tasks and m homogeneous processors, design an efficient fault recovery mechanism which attempts to satisfy all DP-Fairness based timeliness constraints during any recovery period subsequent to a permanent processor failure.

4.2 Fault Tolerant Fair Scheduler (FT-FS)

In this section, we describe the overall working of proposed *FT-FS* scheduler.

The pseudocode of the *FT-FS* scheduler is presented in Algorithm 5. Similar to DP-

ALGORITHM 5: Fault Tolerant Fair Scheduler: *FT-FS*

Input: T: Task set; m: number of processors; t_p: periodic safety check point interval; t_r: recovery interval; tsl_l: length of the time slice (ts_l) currently being scheduled
Output: Generate and execute schedules of task instances

```
1  Initialize t = next_slice = 0; check_point = t_p;
2  while TRUE do
3      if t = check_point then
4          if a fault is detected at time t then
5              Interrupt all processors;
6              next_slice = t + t_r;
7              check_point = next_slice + t_p;
8              Call function FT-FS_Faulty() and wait until completion;
9          else
10             check_point = check_point + t_p;
11     if t = next_slice then
12         Call function FT-FS_Normal();
13         Execute the generated schedule on each processor in parallel;
14         next_slice = next_slice + tsl_l ;
15     t = min(check_point, next_slice);
```

Fair, *FT-FS* partitions time into slices, demarcated by the periods/deadlines of all tasks in the system. The duration between any two consecutive deadlines (say, the $(l-1)^{th}$ and l^{th} deadlines) is referred to as a time slice ts_l and tsl_l denotes the length of ts_l. The entire *FT-FS* scheduler essentially executes within a while loop (lines 2-15) which continues until the system stops. Within the loop, *FT-FS* checks whether the value of timer t coincides with the next check pointing instant (variable *check_point*). If it does and a fault is detected, all processors in the system are interrupted and the scheduler enters into *fault mode* (by calling the function *FT-FS_Faulty()*; refer Algorithm 7) for a duration t_r which denotes the recovery interval. In the absence of a fault, it simply updates *check_point* to next periodic check point instant.

If timer t coincides with the time slice boundary (variable *next_slice*), *FT-FS* first invokes *FT-FS_Normal()* (refer Algorithm 6) to generate a *work-conserving schedule* for the ensuing time slice. Then it executes tasks in the time slice, based on the generated

schedule. Finally, before proceeding to the next iteration of the loop, *FT-FS* updates timer t to the earlier between the next check point instant and the next time slice boundary. We now discuss the function *FT-FS_Normal()* in detail. Function *FT-FS_Faulty()* is discussed in Section 4.2.3.

4.2.1 FT-FS: Normal Mode of Operation

The pseudocode of the function *FT-FS_Normal()* is presented in Algorithm 6. At the beginning of time slice ts_l (say, at time t), each task T_i is allocated a share sh_i^l (proportional to its weight) to be executed within ts_l and is calculated as follows:

ALGORITHM 6: Function *FT-FS_Normal()*

Input: R_l: Active task set, $\bar{e_i}$: the remaining execution requirement of each task T_i ($\in R_l$)
Output: Generate schedule for time slice ts_l

```
for each task T_i in R_l do
    sh_i^l = min(eu_i × tsl_l, ē_i) {refer Equation 4.2};
    ē_i = ē_i − sh_i^l;
scap = m × tsl_l − Σ_{i=1}^{|R_l|} sh_i^l {refer Equation 4.5};
if scap > 0 then
    for each task T_i in R_l do
        if ē_i > 0 then
            Determine T_i's urgency factor uf_i, and update its share sh_i^l and ē_i {refer Equation 4.6};
    Recalculate scap using Equations 4.3 and 4.5;
if scap > 0 then
    Create a list lt of tasks for which ē_i > 0, sorted in non-increasing order of their lag(T_i, t + tsl_l) values;
    for each task T_i in lt do
        sh_i^l = sh_i^l + 1, ē_i = ē_i − 1, scap = scap − 1;
        if ē_i = 0 then
            Remove T_i from list lt;
        if scap = 0 then
            exit;
After finalizing the shares of all tasks in R_l generate schedule for time slice ts_l using McNaughton's wrap-around rule [82].
```

Let, R_l denote the set of active tasks at time t. The total workload within time slice ts_l becomes: $L = \sum_{i=1}^{|R_l|} u_i$. Given L, the *effective* execution rate (also referred to as *effective weight*) of each task T_i at time t is calculated as:

$$\forall T_i \in R_l, eu_i = min(\frac{\mathcal{M}}{L} \times u_i,\ 1) \tag{4.1}$$

where, $\mathcal{M} = m$, is the number of available processors. *FT-FS_Normal()* first determines an initial share for each task T_i in R_l as:

$$\forall T_i \in R_l, sh_i^l = \lfloor min(eu_i \times tsl_l,\ \bar{e_i}) \rfloor \tag{4.2}$$

where, $\bar{e_i}$ is the currently remaining execution requirement of T_i's current instance. Let, sum_shr^l denote the sum of shares of all tasks within time slice ts_l. Thus,

$$sum\_shr^l = \sum_{i=1}^{|R_l|} sh_i^l \tag{4.3}$$

Given sum_shr^l, there exists a *feasible* schedule within ts_l only if,

$$sum\_shr^l \leq \mathcal{M} \times tsl_l \tag{4.4}$$

If $sum\_shr^l < \mathcal{M} \times tsl_l$, there exists some spare capacity (denoted as *scap*) within time slice ts_l:

$$scap = \mathcal{M} \times tsl_l - sum\_shr^l \tag{4.5}$$

Now, *FT-FS_Normal()* proportionally distributes this residual capacity *scap* among all tasks in R_l for which $\bar{e_i} > 0$. Thus, the modified task shares become:

$$\forall T_i \in R_l, sh_i^l = sh_i^l + \lfloor min(scap \times uf_i,\ \bar{e_i}) \rfloor \tag{4.6}$$

where, uf_i is termed as the *relative urgency factor* of T_i and is defined as:

$$uf_i = \bar{e_i}/\bar{p_i} \Big/ \sum_{i=1}^{|R_l|} \bar{e_i}/\bar{p_i} \tag{4.7}$$

Here, $\bar{p_i}$ is the currently remaining time before the completion of T_i's period. Equation 4.6 ensures that the fraction of the residual capacity allocated to task T_i is proportional to its relative execution urgency uf_i. If there still remains some residual capacity after updating task shares, *FT-FS_Normal()* creates a list lt of tasks for which $\bar{e_i} > 0$,

sorted in non-increasing order of their $lag(T_i, t + tsl_l)$[1] values. Here, $lag(T_i, t + tsl_l)$ denotes the lag of task T_i at the end of ts_l (at time $t + tsl_l$), assuming T_i to have executed its share sh_i^l in ts_l. Thus, $lag(T_i, t + tsl_l) = \frac{e_i}{p_i} \times (t + tsl_l) - (allocated(T_i, t) + sh_i^l)$.

Now, *FT-FS_Normal()* sequentially chooses the next task in lt and increases its share by 1 until the residual capacity *scap* is exhausted. After this, a schedule for all tasks in R_l is generated corresponding to time slice ts_l, using McNaughton's wrap-around rule [82]. Based on this generated schedule, tasks allocated to each processor are executed in parallel until completion of the time slice.

Example 1: Consider a set of eight periodic tasks, T_1 $(11, 52, 1)$, T_2 $(13, 50, 2)$, T_3 $(13, 54, 3)$, T_4 $(11, 52, 4)$, T_5 $(10, 51, 5)$, T_6 $(11, 54, 6)$, T_7 $(11, 50, 7)$ and T_8 $(10, 52, 8)$ to be executed on two unit capacity processors ($m = 2$) using the *FT-FS* scheduling scheme. Figure 4.1 depicts the *FT-FS* schedule for the first 100 time slots.

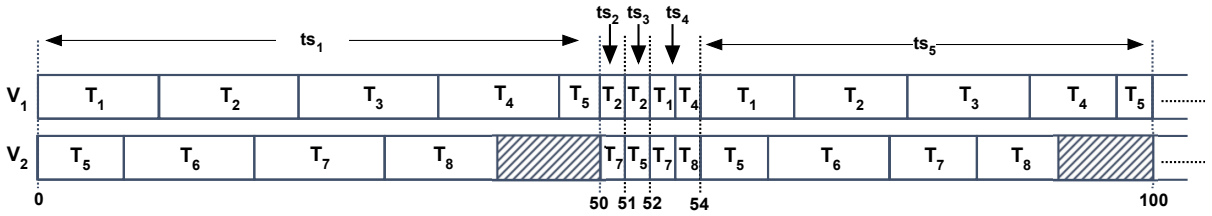


Figure 4.1: *FT-FS schedule for first 100 time slots.*

The time slice boundaries corresponding to the above task set occur at time instants $50, 51, 52, 54$ and 100. Therefore, $tsl_1 = 50, tsl_2 = 1, tsl_3 = 1, tsl_4 = 2$ and $tsl_5 = 46$. So, at the beginning of ts_1, the initial allocated shares of currently active tasks $T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8$ become: $sh_1^1 = sh_4^1 = sh_6^1 = sh_7^1 = 11, sh_2^1 = sh_3^1 = 13$ and $sh_5^1 = sh_8^1 = 10$, (see Equations 4.1 and 4.2). Here, $sum\_shr^1(= 90) < m \times tsl_l(= 2 \times 50 = 100)$ (see Equation 4.4) and all active tasks are able to satisfy their required execution requirements. Therefore, there exists a feasible schedule within ts_1 (refer Algorithm 6) and these tasks are executed in ts_1, based on the generated schedule. At the beginning of the second time slice ts_2, i.e., at time $t = 50$, the currently active tasks

[1] *lag* [7] defines the fairness deviation for each task T_i at time t and is given as: $lag(T_i, t) = (e_i/p_i) \times t - allocated(T_i, t)$. Here, $allocated(T_i, t)$ is the amount of execution actually completed by T_i subsequent to its start at time 0.

in the system are T_2 and T_7. The final allocated shares of these tasks to be executed within ts_2 are given by: $sh_2^2 = sh_7^2 = 1$. At the beginning of time slice ts_3, the currently active tasks are T_2, T_5, T_7, and their allocated shares are 1, 1, 0, respectively. Here, *lag* based priority order of these three tasks is: $T_5 > T_2 > T_7$, and total number of available time slots is: $m \times tsl_3 = 2 \times 1 = 2$. Therefore, tasks T_2 and T_5 acquire a share of 1 time unit each, in time slice ts_3. In the fourth time slice ts_4, the currently active tasks are $T_1, T_2, T_4, T_5, T_7, T_8$, and their allocated shares are 1, 0, 1, 0, 1, 1, respectively. Here, *lag* based priority order of these six tasks is: $T_1 > T_4 > T_8 > T_7 > T_5 > T_2$, and total number of available time slots is 4. Therefore, tasks T_1, T_4, T_7, T_8 acquire a share of 1 time unit each, in time slice ts_4. The rest of the schedule continues as shown in Figure 4.1. □

4.2.2 Fault Model

As discussed, a fault occurring at F_{OT} remains undetected up to F_{DT} (refer Section 4.1: System Model). Hence, the *FT-FS* scheduler continues allocating tasks on all m processors between F_{OT} and F_{DT}, although tasks assigned to the faulty processor during this interval cannot progress. Subsequent to the detection of the fault, *FT-FS* updates its information regarding the partial progress of each running task at F_{DT}, taking into account the schedule of tasks allocated to the faulty processor during $[F_{OT}, F_{DT}]$.

At F_{DT}, the scheduler moves to the *fault mode* of operation and transits back to its *normal operational mode* at the end of the recovery period at F_{RT}. Due to the unavailability of one processor during the recovery period, the tasks may be forced to execute at lower rates than their required execution rates, potentially leading to *transient overloads*. In this context, the objective of *FT-FS* in *fault mode* is to maintain DP-Fairness (that is, ERfairness at all deadline boundaries) of the system during the recovery period even under transient overloads, allowing judicious task rejections, if need be.

4.2.3 FT-FS: Fault Mode of Operation

In order to recover from a fault, Algorithm *FT-FS* calls the function *FT-FS_Faulty()*. This function first determines the schedule of tasks for the entire recovery period, at time F_{DT}, and then, executes tasks in accordance with the generated schedule until the system recovers, at time F_{RT}. The pseudocode of *FT-FS_Faulty()* is presented in Algorithm 7. *FT-FS_Faulty()* proceeds in a time slice by time slice manner within the recovery interval. The first time slice starts at time F_{DT} while the last slice ends at F_{RT}. We use the term ts_l to denote the l^{th} time slice within the recovery period. The term tsl_l is used to denote the length of ts_l.

ALGORITHM 7: Function *FT-FS_Faulty()*

Output: Task schedule for recovery period: $[F_{DT}, F_{RT})$

Initialize: Current time $tt = F_{DT}$;
while $tt < F_{RT}$ **do**
  Determine tsl_l, the length of the ensuing time slice;
  Calculate required rates of execution $\bar{e}_i/\bar{p}_i$ and effective weights eu_i^f, for all tasks in R'_l;
  if *system is in unsafe state* {*refer Equation 4.8*} **then**
    Partition task set R'_l into three disjoint subsets $A1, A2$ and $A3$ based on whether the tasks are *needy*, *affluent* or *balanced*, respectively;
    $G^f = \sum_{i=1}^{|A1|} U_i$; $H^f = \sum_{j=1}^{|A2|} O_j$;
    if $H^f < G^f$ **then**
      Reject T_i, the least critical task instance with the highest U_i value, from $A1$;
      $tt = max(s_i, F_{DT})$ {backtrack schedule};
      Update R'_l back to its contents at time tt;
      Update $\bar{e}_i, \bar{p}_i$ of all tasks in R'_l back to their values at tt;
      Remove T_i from R'_l;
    else
      Call function *Weight_Donation()*;
  if *system is in safe state* **then**
    Call function *FT-FS_Normal()*;
    $tt = tt + tsl_l$;
Execute schedule for the interval $[F_{DT}, F_{RT})$;

Let R'_l denote the set of active tasks at the beginning of ensuing time slice ts_l (say, at

time t) within the recovery period. *FT-FS_Faulty()* first determines tsl_l, which is given by the earliest deadline among the deadlines of all currently active task instances. The *effective* execution rate of each task T_i in R'_l at time t, denoted as eu_i^f, is calculated using Equation 4.1, where $\mathcal{M} = m - 1$ (the number of available processors during the recovery period). The state of the system is considered to be *safe* at time t, if the actual effective rates of execution for all tasks are at least equal to their required execution rates. That is,

$$\forall T_i \in R'_l, eu_i^f \geq \bar{e_i}/\bar{p_i} \tag{4.8}$$

The system is considered to be in an *unsafe* state, otherwise. The tasks T_i for which the condition: $eu_i^f < \bar{e_i}/\bar{p_i}$, is true, are referred to as *needy* tasks. Similarly, tasks T_i for which: $eu_i^f > \bar{e_i}/\bar{p_i}$, is true, are referred to as *affluent*. Lastly, when: $eu_i^f = \bar{e_i}/\bar{p_i}$, we refer the tasks to be in *balanced* condition. Inability to steer the system to *safety* from an *unsafe* state makes the system susceptible to a *failure event* in the future, in which *DP-Fairness constraints* for one or more *needy* tasks may be violated. Here, *DP-Fairness constraint* refers to the necessity to meet ERfairness only at time slice boundaries.

The algorithm checks for the safety of the system at the beginning of each time slice ts_l using Equations 4.1 and 4.8. If the state of the system is *safe*, the schedule generation proceeds as in the *normal* operational mode, and *FT-FS_Faulty()* calls *FT-FS_Normal()* in this case. In an *unsafe* situation, *FT-FS_Faulty()* performs a set of corrective actions in the endeavor to drive the system to *safety*. *FT-FS_Faulty()* first divides the set of tasks R'_l into three disjoint subsets $A1, A2$ and $A3$ based on whether the tasks are *needy*, *affluent* or *balanced* respectively, such that $R'_l = A1 \cup A2 \cup A3$. The *overallocation rate* corresponding to an *affluent* task T_j in $A2$ is given by:

$$O_j = eu_j^f - \bar{e_j}/\bar{p_j} \tag{4.9}$$

Similarly, the *underallocation rate* of a *needy* task T_i in $A1$ is denoted by:

$$U_i = \bar{e_i}/\bar{p_i} - eu_i^f \tag{4.10}$$

Now, if $O_j \geq U_i$, it is clear that both tasks T_i and T_j will be able to complete their execution requirements in a time slice, provided T_j is allowed to *donate* a portion U_i

from its effective weight eu_j^f, to T_i. After T_i *accepts* this donation, the effective execution rates of T_i and T_j become: $\bar{e_i}/\bar{p_i}$ and $eu_j^f - U_i$, respectively.

As an illustration, let us consider two tasks T_1 (*affluent*) and T_2 (*needy*) with effective and required weights ($\{eu_i^f, \bar{e_i}/\bar{p_i}\}$) $\{1/4, 1/5\}$ ($O_1 = 1/20$) and $\{1/10, 1/8\}$ ($U_2 = 1/40$), respectively. Given the parameters, in the absence of weight donation, T_1 and T_2 will complete 10 and 4 time slots of execution, respectively within a time slice ts_l of length $tsl_l = 40$. However, according to required rates, T_1 and T_2 need to complete 8 and 5 time slots of execution, respectively. Subsequent to weight donation, the effective weights of T_1 and T_2 become: 9/40 and 1/8, respectively. T_1 now completes ($9/40 \times 40 =$) 9 time slots, while T_2 is able to complete ($1/8 \times 40 =$) 5 time slots in ts_l, and so, both tasks are able to satisfy at least their required execution requirements.

Now, *FT-FS_Faulty()* calculates the aggregate (denoted by H^f) over the *overallocation rates* of tasks in $A2$: $H^f = \sum_{T_j \in A2} O_j$. Similarly, the aggregate *underallocation rate* G^f over U_i's is: $G^f = \sum_{T_i \in A1} U_i$. When $H^f \geq G^f$, *FT-FS_Faulty()* initiates *weight donation* (refer Algorithm 8), which allows each *needy* task T_i in $A1$ to receive the necessary boost to its effective weight so that it can execute at its required execution rate $\bar{e_i}/\bar{p_i}$ in time slice ts_l. For this, *FT-FS_Faulty()* selects the first *needy* task T_i and the first *affluent* task T_j from sets $A1$ and $A2$, respectively. If $O_j \geq U_i$, T_j donates a weight equivalent to U_i from its effective weight eu_j^f to T_i. As a result of this donation, the effective execution rates of T_i and T_j become: $\bar{e_i}/\bar{p_i}$ and $eu_j^f - U_i$, respectively. Since the requirement of T_i is satisfied, it is removed from $A1$ and added into the *balanced* set $A3$. In the special case, when $O_j = U_i$, T_j is also moved to $A3$. On the other hand, if $O_j < U_i$, T_j can only partially satisfy T_i's need. The effective execution rates of T_i and T_j subsequent to donation become: $eu_i^f + O_j$ and $\bar{e_j}/\bar{p_j}$. T_j is moved to $A3$. This donation of weights continues until $A1$, the set of needy tasks, becomes empty.

If $H^f < G^f$, it means that the aggregate slack weight (H^f) of $A2$ is not sufficient to satisfy the total additional need (G^f) of $A1$, and so, even weight donation cannot guarantee DP-Fairness of all tasks. Therefore, one or more task instances (jobs) in $A1$ must be rejected so that the rest of the system remains *fail-operational*. At any

ALGORITHM 8: Function *Weight_Donation()*

Input: Needy set: $A1$, Affluent set: $A2$, Balanced set: $A3$
Output: Updated $A1, A2, A3$

```
1  while A1 ≠ ∅ do
2      Select first needy task T_i and first affluent task T_j from sets A1 and A2,
       respectively;
3      if O_j ≥ U_i then
4          eu_j^f = eu_j^f − U_i;  eu_i^f ← ē_i/p̄_i;
5          O_j ← O_j − U_i; U_i ← 0;
6          A1 ← A1 \ T_i; A3 ← A3 ∪ T_i;
7          if O_j = 0 then
8              A2 ← A2 \ T_j; A3 ← A3 ∪ T_j;
9      else
10         eu_j^f = eu_j^f − O_j;  U_i ← U_i − O_j;  O_j ← 0;
11         A2 ← A2 \ T_j; A3 ← A3 ∪ T_j;
```

scheduling point if $H^f < G^f$, *FT-FS_Faulty()* rejects the least critical task instance T_i having the highest underallocation value (U_i). This strategy attempts to ensure the rejection of the least number of high criticality jobs. Then, the algorithm backtracks the schedule generation back to the time $tt = max(s_i, F_{DT})$, where s_i is the arrival time of T_i. Now, *FT-FS_Faulty()* attempts to regenerate the schedule from time tt, this time, without task T_i in R'_l. Subsequent to a rejection, the actual effective execution rates of the tasks increase relatively due to reduced workload effected by the task rejection. This increases the possibility of the system becoming *safe* (checked using Equation 4.8) or at least reduces the total additional requirement of *needy* tasks during the weight donation process. Further rejections may be required if the weight donation process fails in satisfying the requirement of all needy tasks. After the entire schedule for the interval $[F_{DT}, F_{RT}]$ is generated, *FT-FS_Faulty()* executes the schedule and returns back to the main function *FT-FS()* at time F_{RT}.

Example 2: Let us continue with the same system scenario discussed in Example 1, with the exception that processor V_2 now suffers a permanent fault at time $F_{OT} = 38$. The fault is detected at $F_{DT} = 40$ and the recovery interval t_r being 60, we get $F_{RT} = 100$.

After detecting the fault at $F_{DT} = 40$, the main function *FT-FS* calls *FT-FS_Faulty()*, which proceeds in a time slice by time slice manner within the interval $[F_{DT}, F_{RT}]$. Therefore, the recovery duration consists of five slices $ts_1, ..., ts_5$ ($tsl_1 = 10, tsl_2 = tsl_3 = 1, tsl_4 = 2, tsl_5 = 46$) based on task deadlines. Figure 4.2 depicts the schedule generated by *FT-FS_Faulty()* during the interval $[F_{DT}, F_{RT}]$.

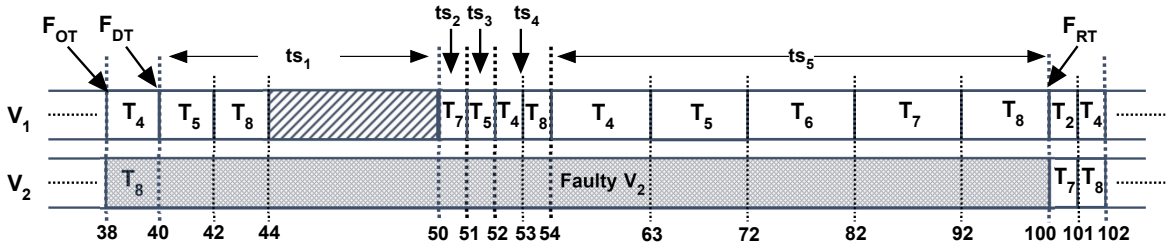


Figure 4.2: *The schedule generated by FT-FS_Faulty() during the interval* $[40, 100]$

At the beginning of time slice ts_1, the active task set is $R'_1 = \{T_4, T_5, T_8\}$. The effective and required weights ($\{eu_i^f, \bar{e}_i/\bar{p}_i\}$) of these active tasks are {0.3526, 0.6666}, {0.3268, 0.1818} and {0.3205, 0.1666}, respectively. Here, $eu_4^f < \bar{e}_4/\bar{p}_4$ (refer Equation 4.8) and the system is in *unsafe* state. Partitioning R'_1 we get, $A1 = \{T_4\}$ (needy), $A2 = \{T_5, T_8\}$ (affluent) and $A3 = \emptyset$ (balanced). Since, $H^f (= 0.2989) < G^f (= 0.31405)$, the system cannot be driven to safety without incurring any rejection. Hence, task T_4 (only needy task in R'_1) is rejected and the system becomes safe subsequent to its removal. *FT-FS_Faulty()* therefore calls *FT-FS_Normal()* to generate the schedule for ts_1 as depicted in Figure 4.2. Similarly, *FT-FS_Faulty()* generates the schedules for time slices ts_2 and ts_3.

At time $tt = 52$, the next instances of T_1, T_4 and T_8 arrive and the active task set R'_4 becomes $\{T_1, T_2, T_4, T_5, T_7, T_8\}$. Here, the system is in an *unsafe* state with $A1 = \{T_1, T_2, T_4, T_5, T_7, T_8\}$ and $A2 = A3 = \emptyset$. Since $H^f < G^f$, the least critical task T_1 in $A1$ is rejected and *FT-FS_Faulty()* reattempts to generate a feasible schedule (now, without T_1) for ts_4 (T_1 has arrived at the beginning of ts_4). However, schedule generation fails again and the system is still observed to be *unsafe* with $H^f < G^f$ ($A1 = \{T_2, T_4, T_5, T_7, T_8\}$ and $A2 = A3 = \emptyset$). *FT-FS_Faulty()* now rejects T_2, the least critical task, and attempts to regenerate a feasible schedule starting from the beginning

of ts_2, the stipulated time for the arrival of T_2. After generating the schedules for ts_2 and ts_3 (now, without T_2) in a similar manner as before, a feasible schedule for ts_4 (without tasks T_1 and T_2) can be constructed with the task shares for T_4, T_5, T_7, T_8 being $sh_4^4 = 1, sh_5^4 = 0, sh_7^4 = 0, sh_8^4 = 1$. Proceeding further, at the beginning of ts_5 at time $tt = 54$, the system again becomes *unsafe* due to the arrival of the next instances of tasks T_3 and T_6 ($A1 = \{T_3, T_4, T_5, T_6, T_7, T_8\}$ and $A2 = A3 = \emptyset$). As $H^f < G^f$, the least critical task T_3 is rejected and the schedule generation for ts_5 is reattempted without T_3. This time, although the system is still unsafe, $H^f > G^f$ and so, system safety may be achieved through weight donation alone with no further task rejection being required. Subsequent to successful weight donations (refer Algorithm 8), the effective weights of the tasks T_4, T_5, T_6, T_7, T_8 in R'_5 are updated from 0.20665 to 0.2, 0.19155 to 0.19103, 0.1990 to 0.2037, 0.21492 to 0.21739, 0.18788 to 0.18788, respectively. At time $t = F_{RT} = 100$, the backup processor becomes operational and the system recovers from the fault. □

Example 3: Let us continue with the same system scenario discussed in Example 1, with the exception that system has three unit capacity processors and processor V_2 suffers a permanent fault at time $F_{OT} = 38$. The fault is detected at $F_{DT} = 40$ and the recovery interval t_r being 60, we get $F_{RT} = 100$. Figure 4.3 depicts the *FT-FS* schedule for this system in the fault mode of operation. It may be observed from Figure 4.3 that

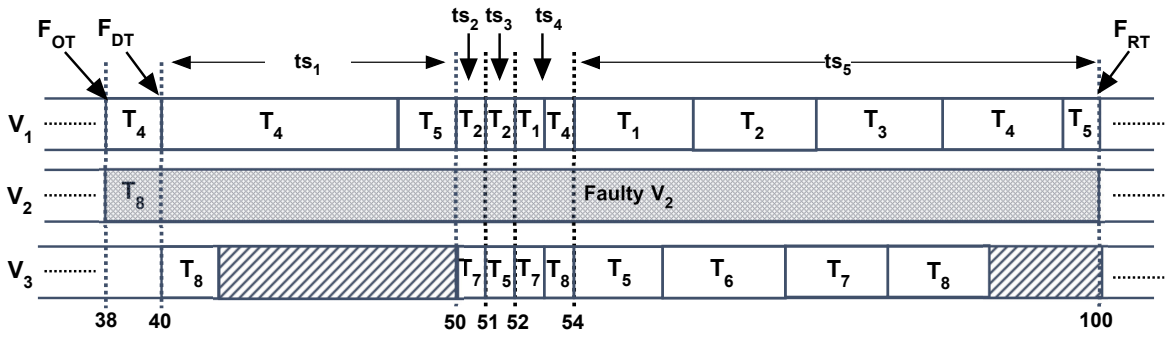


Figure 4.3: *The schedule generated for a 3 processor system under fault mode of operation*

even in the presence of the faulty processor (V_2), the system incurs no task rejections. This is because FT-FS effectively schedules the given workload on the remaining two

non-faulty processors. □

4.2.4 Complexity Analysis

In this section, we present the time complexity analysis of the proposed FT-FS scheduler, with the objective of accounting the scheduling overheads associated with the algorithm.

Lemma 4.2.1. *Complexity of function FT-FS_Normal() (Algorithm 6) is $O(nlgn)$ per time slice.*

Proof. The loops in lines 1-3, 6-8 and 12-17 run once for each task with each step inside the loop taking constant time. So, the complexity of these loops become $O(n)$. Steps 4 and 9 which involves updation of the spare capacity *scap* and takes $O(n)$ time. Step 11 which creates a modified task list, consumes $O(nlgn)$ time. Step 18 involves generation of a schedule within a time slice and also takes $O(n)$ time. Therefore, function *FT-FS_Normal()* has an overall complexity of $O(nlgn)$ per time slice. □

Theorem 4.2.2. *Amortized complexity of function FT-FS_Normal() (Algorithm 6) is $O(1)$ per processor per time slot.*

Proof. From Lemma 4.2.1, the worst case time complexity of function *FT-FS_Normal()* is obtained as $O(nlgn)$ per time slice. The length of a time slice may be considered to be roughly proportional to the average task period length. Further typically, it may be realistically assumed that: *#tasks* $n << $ *average period size* $\times$ *#processors* m. Hence, *#tasks* $n << $ *time slice length* $\times$ *#processors* m. So, the amortized complexity of *FT-FS_Normal()* may be considered to be $O(1)$ per processor per time slot. □

Lemma 4.2.3. *Complexity of function Weight_Donation() (Algorithm 8) is $O(n)$.*

Proof. Algorithm 8 performs weight donation from affluent tasks in set $A2$ to needy tasks in $A1$, until the number of needy tasks in $A1$ reduces to zero. The maximum number of iterations of the while loop (Steps 1 to 11) is upper bounded by the initial cardinality of $A1$ which is $O(n)$. Therefore, this loop takes $O(n)$ time. Steps 6, 8 and 11 involve insertion and deletion operations at the beginning of sets $A1, A2, A3$ and each such operation takes constant time. The other steps in the while loop also consume $O(1)$ time. Therefore, the worst case time complexity of *Weight_Donation()* is $O(n)$. □

Lemma 4.2.4. *Average complexity of function FT-FS_Faulty() (Algorithm 7) is* $O(nlgn)$.

Proof. Function *FT-FS_Faulty()* is used to generate a schedule for the recovery duration $[F_{DT}, F_{RT})$. Step 3 involves determination of the length of the ensuing time slice, at each time slice boundary within $[F_{DT}, F_{RT})$. This length is given by the earliest among the deadlines of all currently active instances and takes $O(n)$ time for its computation. Each of the Steps 4 to 7 involves operations over sets of tasks where the cardinality of each of these sets are upper bounded by the total number of tasks, n. So, each of these steps takes $O(n)$ time. Step 9 which involves finding the least critical *needy* task with the highest utilization value, say T_i (from set $A1$), and consumes $O(n)$ time to complete. Step 10 updates tt to determine the number of time slices by which the schedule should be backtracked and this takes $O(1)$ time. Step 11 involves updation of the active task set R'_l, with respect to the backtracked schedule, and this can be done in $O(n)$ time. Step 12, which updates the $\bar{e}_i$ and $\bar{p}_i$ values of all tasks in the modified set R'_l, takes $O(n)$ time. Step 13 involves deletion of task T_i from R'_l and incurs an overhead of $O(n)$. Steps 15 and 17 call functions *Weight_Donation()* (Lemma 4.2.3) and *FT-FS_Normal()* (Lemma 4.2.1), respectively. These steps take $O(n)$ and $O(nlgn)$ times, respectively. Step 18 increments tt and takes $O(1)$ time. Therefore, the complexity of executing the single iteration of the *while* loop (steps 2 to 18) is $O(nlgn)$ in the worst case. The number of iterations of the while loop is determined by the number of time slices for which the schedule needs to be generated. This number must include backtracked re-executions of certain time slices (the time slice intervals during re-execution may possibly be different due to task rejections). Let us represent k to be the number of time slices generated by the active tasks (excluding the rejected tasks) within the recovery interval and k' to be the total number of time slices by which the schedule was backtracked due to all task rejections. Now given k and k', the complexity of *FT-FS_Faulty()* may be obtained as $O((k + k')nlgn)$. In this work, we have assumed a closely-coupled system in which the cold-standby processor may be activated within a short recovery duration (varying between say, tens of milliseconds to at most a few seconds) subsequent to the detection of a fault. Further, as time slices are demarcated by task period boundaries, average length of a time slice can be considered to be approximately equal to the mean period length of the given task-set. We have also observed that for a large class of real-time systems including automotive and avionic systems, most process control systems, satellites etc.,

task period lengths typically vary between tens to hundreds of milliseconds, or even more. Therefore, the number of time slices k, within the recovery duration may be reasonably considered to be bounded by a small constant. Average number of time slices by which the schedule needs to be backtracked (say, k") at each task rejection is upper bounded by k, and so, k" may also be considered to assume small values. Moreover, the algorithm attempts to choose tasks with high utilization values for rejection. Due to this, transient overloads during recovery are typically mitigated by only suffering a small number of rejections (say, n_r), in most cases. Based on these arguments, k' (=k"×n_r) can also be assumed to be upper bounded by a small constant. Hence, assuming both k and k' to be small constants, the average complexity of *FT-FS_Faulty()* becomes $O(nlgn)$. □

Theorem 4.2.5. *Amortized complexity of function FT-FS_Faulty() (Algorithm 7) is* $O(1)$ *per processor per time slot.*

Proof. At the beginning of the recovery interval, *FT-FS_Faulty()* takes $O(nlgn)$ time to generate a schedule for the entire recovery period (Lemma 4.2.4). Therefore, the amortized complexity of *FT-FS_Faulty()* may be obtained by determining the average overhead incurred by the function per processor per time slice over the recovery interval $[F_{DT}, F_{RT})$. In our work, we have represented $[F_{DT}, F_{RT})$ as an integral number (say, k) of time slices with the duration of a typical time slice being assumed to be equal to average task period length. Therefore, amortized complexity of *FT-FS_Faulty()* becomes, $O(nlgn/($ $k \times$ *average period size* $\times$ *#processors* $m))$. Finally, with the realistic assumption that: *#tasks* $n \times lgn \leq k \times$ *average period size* $\times$ *#processors* m, amortized complexity of *FT-FS_Faulty()* may be considered to be $O(1)$ per processor per time slot. □

Example 4: Let us consider a typical fully loaded system consisting of 8 processors with 40 tasks having an average execution requirement of 50 ms. With average task utilization being 1/5 (=8/40), the average task period length becomes 250 ms. Thus in this case, *#tasks* (40) << *average period length* (250) × #processors (8). These numbers thus validate the amortized time complexity results for *FT-FS_Normal()* and *FT-FS_Faulty()* obtained in Theorem 4.2.2 and Theorem 4.2.5, respectively. □

Theorem 4.2.6. *Amortized complexity of the FT-FS Scheduler (Algorithm 5) is* $O(1)$ *per processor per time slot.*

Proof. It follows from Theorem 4.2.2 and Theorem 4.2.5. □

The complexity analysis conducted above shows that FT-FS is a low-overhead algorithm which incurs $O(1)$ average amortized overhead. This theoretical analysis has also being supported by our obtained experimental results as discussed in Section 4.3.4. Before presenting the detailed results, we now present the experimental framework used in this work.

4.3 Experiments and Results

We have evaluated the performance of the *FT-FS* algorithm through an extensive set of simulation studies (conducted over an experimental framework which is described in the next subsection) and compared its performance against a basic fault-tolerant fair scheduling strategy called *Basic-FS*.

Basic-FS: In its normal mode of operation, *Basic-FS* is same as *FT-FS*. However, it follows a naive and much simpler fault recovery policy. To drive the system to safety during transient overloads in the recovery period subsequent to a fault, *Basic-FS* repeatedly rejects the least critical tasks from set $A1$ until the overload is mitigated. Therefore, neither does this strategy employ *weight donation* nor *backtracking* subsequent to a task rejection.

4.3.1 Experimental Setup

The experimental setup in this work consists of a scenario generation framework which provides input test datasets corresponding to different scenarios over which both the *Basic-FS* and the *FT-FS* algorithms are evaluated. Each result data point is the average obtained by running the algorithms over 100 different datasets corresponding to a given scenario. The schedule length in all simulations have been taken to be 100000 time slots with the length of a time slot being assumed as 1 millisecond. We now discuss the scenario generation framework in more detail.

Given the number of tasks to be generated (n) and the summation of weights of the n tasks (L), the individual task weights (u_i) have been generated from a normal

distribution with standard deviation (σ) = 0.1 and mean (μ) = L/n. The summation of weights of these generated tasks is not constant. However, making the summation of weights constant helps in the comparison of the algorithms. Therefore, the weights have been scaled uniformly to make the cumulative weight of each task set constant and equal to L. Task execution periods (p_i) have been generated from another normal distribution with $\mu = 400$ and $\sigma = 40$. Each task T_i in the system is assigned a criticality level cr_i which is denoted by an integer generated randomly from a uniform distribution with in the range $[1, n]$. The framework also includes a fault injection mechanism which randomly generates fault occurrence instants using Poisson distribution with fault rate $\lambda = 1 \times 10^{-5}$ [87]. The mechanism take cares that two consecutive fault occurrence instants are separated by at least: *recovery interval* t_r + *periodic safety check interval* t_p. The value of t_p has been assumed to be 10 milliseconds in all experiments.

Now, various simulation scenarios have been generated by setting different values for the following parameters:

1. Number of processors m: This parameter is varied between 2 to 10.

2. Task set size n: The number of tasks in the system have been varied from 20 to 100.

3. Workload: Our experiments are conducted on 70%, 75%, 80%, 85%, 90%, 95% or 100% loaded systems.

4. Recovery time t_r: In order to study the effect of t_r, the recovery interval has been varied from 50 to 200 (in milliseconds).

4.3.2 Performance Evaluation Parameters

Fault of a processor may lead to transient overloads in the system, which in turn may necessitate the rejection of one or more jobs in order to keep the system fail-operational. The performance of the proposed framework has been evaluated using four different metrics:

1. **#Job_Rejections**: This denotes the average number of jobs rejected over 100 runs (with distinct datasets) by the algorithms *FT-FS* and *Basic-FS* corresponding to a given scenario.

2. **#Penalty**: This parameter represents the average (over 100 runs) aggregate penalty suffered by the system due to job rejections. Here, penalty (corresponding to the single run of the experiment) is calculated by the summation of the criticality values of rejected jobs over the schedule length.

3. **Normalized context switch overhead**: It is the average preemption/migration overhead (in μs) per processor per time slot, incurred by FT-FS.

4. **Normalized scheduling overhead**: It is the average overhead (in μs) per time slot, incurred by FT-FS towards making scheduling decisions.

4.3.3 Results: Performance

Figures 4.4 - 4.6 show the comparison of *FT-FS* and *Basic-FS* based on the performance parameter #Job_Rejection. The plots shown in Figure 4.4 are obtained for two distinct values of recovery period t_r (100 and 200) and varying system workloads (U) corresponding to systems consisting of two processors and 40 tasks. It may be observed from Figure 4.4 that for all cases, as expected, number of job rejections increases as system workload becomes higher due to progressive reduction in slack capacity within the system. Also, rejection rates are seen to increase very steeply when the system becomes almost fully loaded (for, $U > 95\%$). In addition, as recovery interval represents the time for which the system must deal with a sub-nominal capacity, plots for which $t_r = 200$ exhibit higher rejections compared to plots with $t_r = 100$. Finally, the plots clearly show the superiority of the proposed *FT-FS* algorithm over *Basic-FS* in terms of the ability to control job rejections. Empowered with the weight donation and post rejection backtracking mechanisms embedded with *FT-FS*, it incurs significantly lower rejections with respect to *Basic-FS*.

Figure 4.5 considers scenarios containing 40 tasks, 80% or 90% loaded systems with

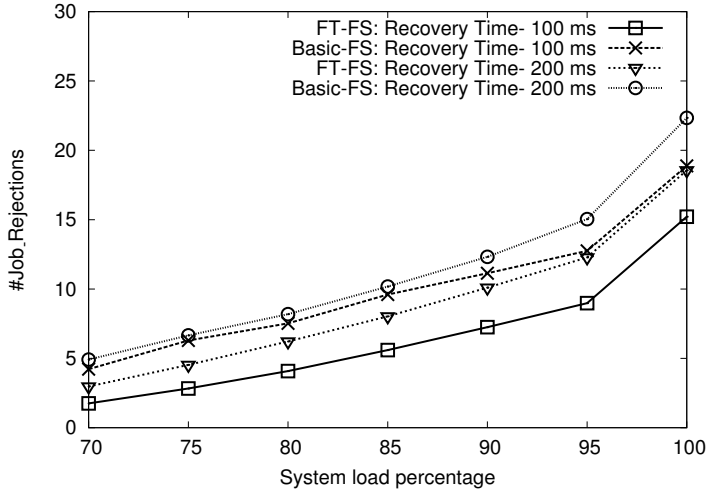


Figure 4.4: *#Job_Rejections vs. System load: 2 processors, 40 tasks*

varying number of processors. As is obvious, for any given number of processors, 90% loaded systems suffer higher rejections compared to systems where load $U = 80\%$. It may also be observed that job rejections decrease with increase in the number of processors when the workload, recovery time and number of tasks remain unchanged. This may be attributed to the fact that fractional loss in system capacity, during recovery, due to the failure of a single processor, decreases with increase in the available number of processors. Also as discussed above, *FT-FS* being more efficient, is seen to always perform better than *Basic-FS*.

Figure 4.6 shows the variation in #Job_Rejections as the number of tasks is varied in a two processor, 80% or 90% loaded system with 100 *ms* as the recovery interval. Here, we observed that #Job_Rejections increases with growth in the number of tasks. This is because, as the number of tasks increases in a scenario with fixed total workload (U=80% or 90%), the number of jobs which suffer underallocation at time slice boundaries (leading to rejections) during recovery, also increases. However, a closer observation shows that although #Job_Rejections increases, the rejection ratio (#Job_Rejections : #Tasks) decreases with larger number of tasks. This is due to the fact that individual task weights decrease as tasks increase in scenarios with a fixed system load and such smaller weights have higher probability of fitting into a given available slack capacity in the

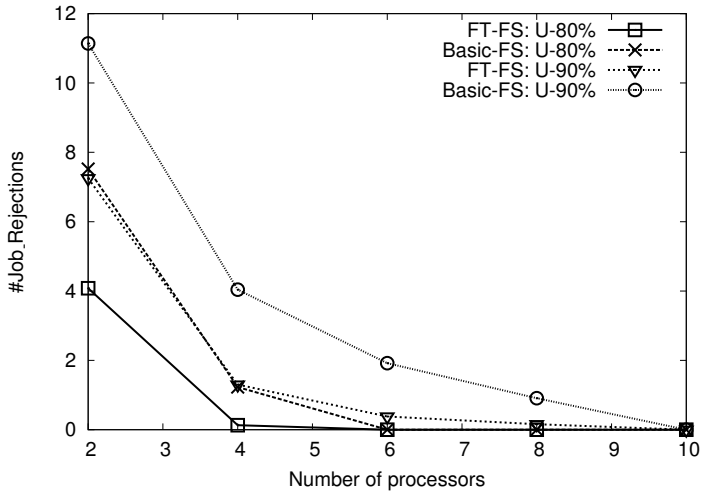


Figure 4.5: *#Job_Rejections vs. #Processors: 40 tasks, t_r = 100 ms*

system. As for the other two figures, *FT-FS* performs consistently better than *Basic-FS*.

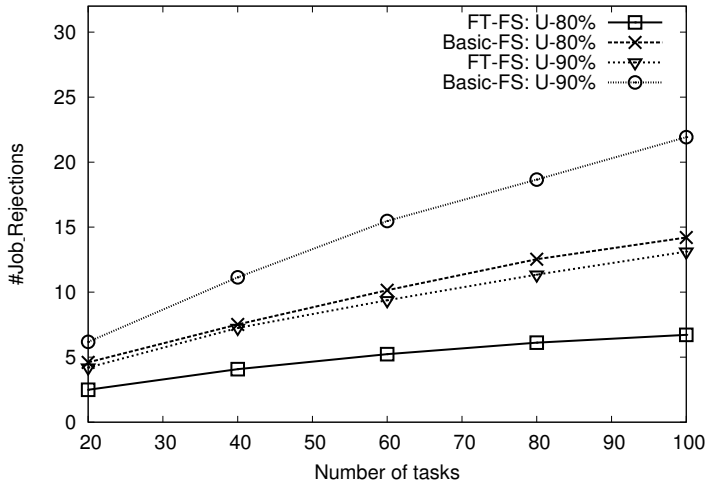


Figure 4.6: *#Job_Rejections vs. #Tasks: 2 processors, t_r = 100 ms*

Figure 4.7 compares *FT-FS* and *Basic-FS* against the performance parameter #Penalty. As expected, *FT-FS* being equipped with weight donation and backtracking capabilities incurs far low penalties compared to *Basic-FS*. Table 4.1 summarizes the comparison results obtained for *FT-FS* and *Basic-FS* on two processor systems.

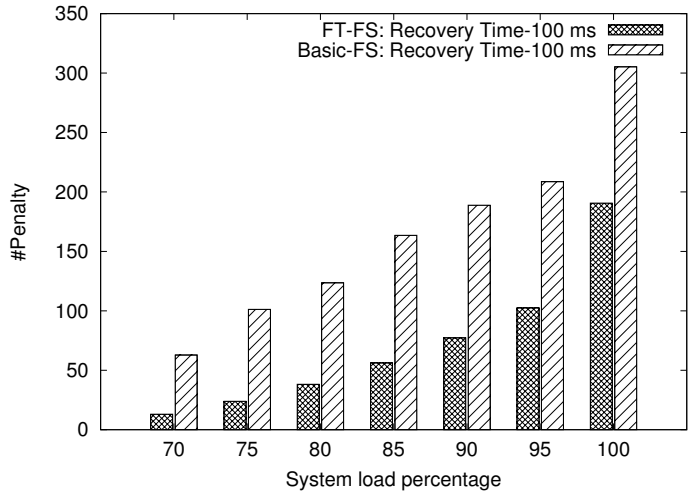


Figure 4.7: *#Penalty vs. System load: 2 processors, 40 tasks, t_r = 100 ms*

Table 4.1: *FT-FS vs. Basic-FS: Average number of jobs rejected*

U	t_r	**n=20**		**n=40**		**n=60**	
		FT-FS	**Basic-FS**	**FT-FS**	**Basic-FS**	**FT-FS**	**Basic-FS**
75%	50	1.53	3.36	2.3	5.96	2.83	7.57
	100	1.83	3.5	2.83	6.29	3.54	7.87
	150	2.4	3.76	3.77	6.37	4.66	8.43
	200	2.84	4.06	4.53	6.66	5.47	8.78
85%	50	2.99	4.97	4.97	8.65	6.38	11.85
	100	3.34	5.42	5.6	9.6	7.18	12.8
	150	4.1	5.72	6.88	9.76	8.97	13.07
	200	4.73	6.05	8.03	10.18	10.55	14.03
95%	50	4.88	6.97	8.43	12.35	11.28	17.01
	100	5.23	7.38	8.98	12.75	11.98	18.22
	150	6.1	8.21	10.65	14.12	14.29	19.72
	200	7.1	8.76	12.28	15.04	16.83	21.13
n: Total number of tasks; U: Total system load percentage; t_r: Recovery time							
#processors: 2; t_p: 10 ms; Average period length of tasks, P_{avg}: 400 ms							

4.3.4 Results: Overheads

Figure 4.8 shows plots for the normalized context switch overheads (refer Section 4.3.2) incurred by FT-FS, as the system load is varied on four and eight processor systems. In this experiment, we assume the delay corresponding to a single context switch to be $5.24\mu s$, which is the actual average context switch overhead on a 24-core Intel Xeon L7455 system under typical workloads [17]. The normalized context switch overhead (in μs) is determined as follows: we first counted the average number of context switches per processor per time slot for a given simulation run and then multiplied it with the cost of a single context switch (5.24 μs). It may be observed that for a given task set (n=20 or 40) and number of processors (m=4 or 8), the overhead increases as system load increases from 70% to 100%. This may be attributed to the fact that as system load increases, individual task weights also increase and such larger weights increase the execution times of tasks. As a result, individual task shares within time slices become larger and residual spare capacities in the system reduce. Consequently, a higher number of context switches must be incurred to feasibly accommodate and execute the tasks on the available processors. It may also be observed that, for a given workload, overhead increases with increase in the number of tasks. This is because, the number of time slices within the schedule increases proportionately with increase in the number of tasks. Consequently, the number of migrations across time slices also increases proportionately. Additionally, the number of preemptions within the time slice increases as the number of tasks increases. It may also be observed that for a given task set (n=20 or 40) and workload, the normalized context switch overhead decreases with an increase in the number of processors. This is due to the fact that the spare capacity in the system increases as the number of processors become higher with the system workload remaining the same. Due to such additional spare capacity, the tasks are able to execute continuously on the same processor for longer durations on average, without incurring migrations/preemptions. Finally, it may be observed from Figure 4.8 that the maximum normalized context switch overheads for the considered scenarios is about 1.56 μs per processor per time slot (for four processor, 40 tasks, fully loaded systems). Therefore,

considering 1 millisecond as the time slot length, about 0.156% of a slot duration may be considered to be wasted due to context switch related overheads.

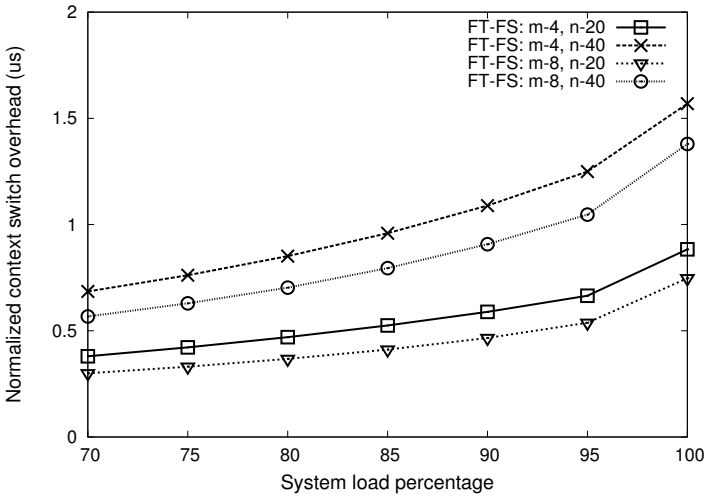


Figure 4.8: *Normalized context switch overheads of FT-FS*

Figure 4.9 shows plots for the normalized scheduling overheads (refer Section 4.3.2) incurred by *FT-FS_Normal()*, as the system load is varied on four and eight processor systems. In this experiment, the normalized scheduling overhead is determined by first finding out the average scheduling overhead for the entire schedule length incurred by *FT-FS_Normal()* over 100 simulation runs. This average scheduling overhead is then divided by the length of schedule to obtain the normalized overhead per time slot. As expected, the overhead increases when both the number of tasks and system load increases. Moreover, for a given task set (n=20 or 40) and workload, normalized scheduling overhead increases with an increase in the number of processors. Finally, it may be observed from Figure 4.9 that the maximum normalized scheduling overheads for the considered scenarios is about 2.27 μs per time slot (for eight processor, 40 tasks, fully loaded systems). This normalized overhead is experienced by tasks on all processors at any time slot. Therefore, considering 1 millisecond as the time slot length, about 0.227% of a slot duration may be considered to be wasted due to scheduling related overheads.

Figure 4.10 shows plots for the normalized scheduling overheads incurred by *FT-FS_Faulty()*, as the recovery time is varied on four and eight processor systems. In this

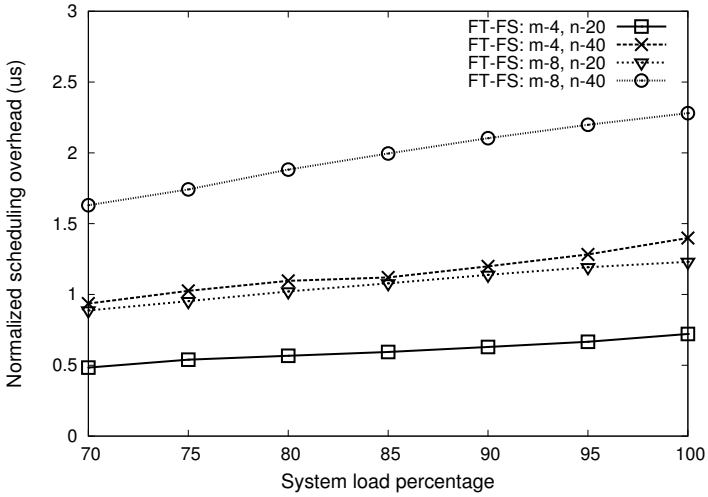


Figure 4.9: *Normalized scheduling overheads of FT-FS_Normal()*

experiment, the normalized scheduling overhead is determined by first finding out the average scheduling overhead incurred by *FT-FS_Faulty()* at the beginning of the recovery interval and then dividing it by the recovery duration. It may be noted that during the recovery period, the numbers of available processors in four and eight processor systems become three and seven, respectively. In addition, these systems are assumed to be hit by a transient overload of 5%, subsequent to the failure of a processor. So, the total system utilization has been fixed at 78.75% for four processor systems and 91.875% for eight processor systems, such that the workloads rise to 105% in both these systems, during the recovery period. As expected, for a given processor (m=4 or 8), the normalized scheduling overhead increases as the number of tasks become higher. Moreover, for a given task set (n=20 or 40), normalized scheduling overhead increases with an increase in the number of processors. It may also be observed that for a given task set (n=20 or 40), processor (m=4 or 8) and transient overload, normalized scheduling overhead decreases with an increase in the recovery time. This is due to the fact that the proportional increase in average scheduling overhead decreases as recovery time increases. It may be observed from Figure 4.10 that the maximum normalized scheduling overheads for the considered scenarios is about 3.74 μs per time slot (for eight processor, 40 tasks systems with recovery time, $t_r = 50\ ms$). This normalized overhead is experienced by

tasks on all available processors at any time slot during the recovery period. Finally, it may be noted that the overall normalized scheduling overhead of the FT-FS scheduler may be obtained as the maximum of the overheads incurred by *FT-FS_Faulty()* and *FT-FS_Normal()*. Finally, the total overhead of FT-FS per time slot is given by the sum of normalized context switch and scheduling overheads. For example, for an eight processor, 40 task system, with $t_r = 50\ ms$, the normalized scheduling overheads of *FT-FS_Faulty()* and *FT-FS_Normal()* are 2.27 μs (refer Figure 4.9) and 3.74 μs (refer Figure 4.10), respectively. Additionally, the normalized context switch overheads of FT-FS corresponding to these system parameters is 1.379 μs (refer Figure 4.8). From these values, the total normalized overhead of FT-FS may be obtained as 1.379 μs + max(2.27, 3.74) μs = 5.119 μs. Therefore, considering 1 millisecond as the time slot length, only about 0.5119% of a slot duration may be considered to be wasted due to overheads. This overhead can be easily incorporated within the schedule by inflating the execution requirement of each task by 0.5119%.

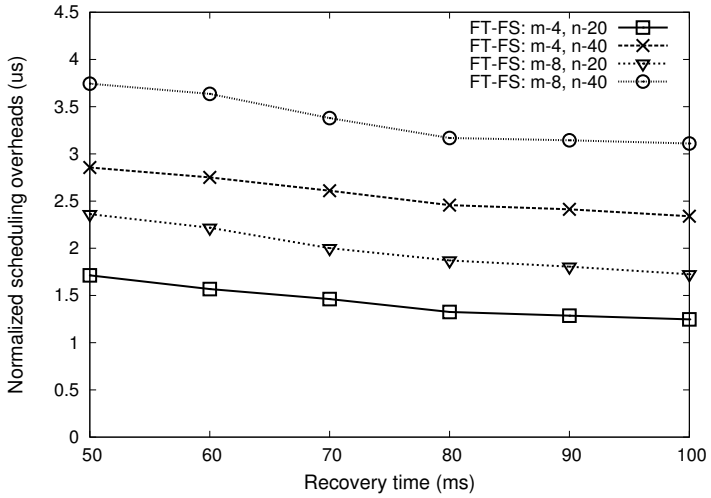


Figure 4.10: *Normalized scheduling overheads of FT-FS_Faulty()*

4.4 Case Study

In this section, we present a case study using an automated flight control system to illustrate the generic applicability of our fault recovery mechanism in real world scenar-

ios. The Flight Management System (FMS) in an aircraft performs several flight control functions, including navigation, guidance, control, etc [11]. FMS requires four separate tasks to control the aircraft during flight: Guidance, Controller, Slow Navigation and Fast Navigation. The *Guidance* task (say T_1) sets the reference trajectory of the aircraft in terms of altitude and compass heading. Based on the reference trajectory and navigation sensor values, the *Controller* task (say T_2) executes closed loop control functions that compute actuation commands for components including, elevator, ailerons, rudder and throttle, to achieve a desired reference altitude and heading for the aircraft. The elevator, ailerons and rudder generate aerodynamic forces that alter aircraft heading and airspeed. The engine throttle generates a force along the aircraft fuselage which is used in combination with the aerodynamic forces to alter aircraft airspeed and altitude. The job of both *Slow* (say T_3) and *Fast* (say T_4) *Navigation* tasks is to read sensors at low and high sampling frequencies, respectively. While slow navigation task is used to feed data to the less critical Guidance task, fast navigation task feeds data to the high critical Controller task. Now, we consider the case of an F-16 fighter aircraft which performs an additional function, *launch a missile at the enemy target during military operation*. Therefore, in addition to the basic flight control tasks (Guidance, Controller, Slow Navigation, Fast Navigation), the fighter aircraft requires a *Missile Control* task (say T_5) to monitor the aircraft radar, detect enemy targets and fire a missile if a target is detected.

Now, consider these five tasks T_1 $(100, 1000, 2)$, T_2 $(80, 200, 3)$, T_3 $(100, 1000, 2)$, T_4 $(60, 200, 3)$, T_5 $(500, 1000, 1)$ to be executed on two unit capacity processors ($m = 2$) using the FT-FS scheduling scheme. The tasks and their associated parameters used for this case study have been taken from [11]. Each of these tasks may be assigned a relative criticality value based on the importance of their usage in flight control performance and military mission operation. In this case study, we assume that flight control tasks have higher relative criticality values compared to the military mission task. Figure 4.11 depicts the FT-FS schedule for this system for the first 200 time slots (time slice ts_1).

All these tasks are real-time and periodic in nature, whose timing constraints have to

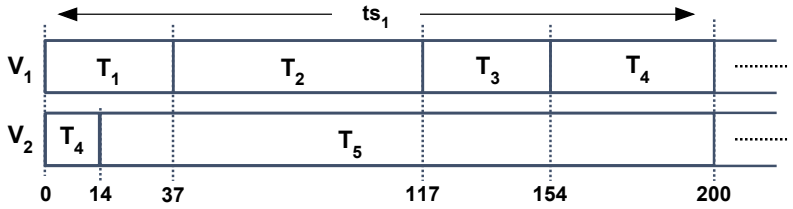


Figure 4.11: *Aircraft Flight Control: Schedules generated by FT-FS under nominal mode of operation for the first 200 time slots.*

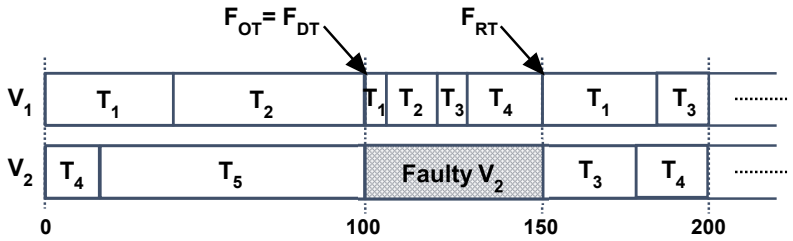


Figure 4.12: *Aircraft Flight Control: Schedules generated by FT-FS under fault mode of operation for the first 200 time slots.*

be satisfied even in the presence of faults. Let us consider the following faulty scenario: $t_p = 10$ ms, $t_r = 50$ ms, $F_{OT} = F_{DT} = 100$ and $F_{RT} = 150$. Due to the failure of any one of the two processors (say, processor V_2 fails), tasks have to execute on the remaining functional processor (V_1) until the system recovers. Figure 4.12 depicts the schedule generated by FT-FS under fault mode of operation. At time $t = 100$, the system is observed to be *unsafe* and FT-FS attempts to regenerate a feasible schedule by rejecting the least critical task T_5. After this rejection, it becomes possible to generate a feasible schedule through weight donation alone with no further task rejection being required during the recovery interval $[100, 150]$ (thereby, displaying the ability of the algorithm to remain fail-operational as far as possible). It may also be observed that under both nominal and fault modes of operation FT-FS is able to deliver optimal resource utilization. Figure 4.11 shows that FT-FS incurs only 1 task migration and 4 preemptions, thus being able to effectively control context switching related scheduling overheads.

4.5 Summary

In this chapter, we have presented a fault-tolerant proportional fair scheduling mechanism called FT-FS, for real-time homogeneous multiprocessor systems containing cold-standby spares. Subsequent to the detection of a permanent processor fault, the system requires a fixed recovery interval to boot up the spare processor to the operational state. Equipped with two novel features namely, weight donation and post rejection backtracking, the proposed scheduler FT-FS attempts to minimize rejections of critical jobs, during transient overloads within recovery intervals. The objective is to maximize the possibility of keeping the system fail-operational even in the presence of faults. The underlying scheduling structure being based on DP-Fair, FT-FS is able to ensure high resource utilization and fair rate-based execution progress while incurring low scheduling related overheads through controlled migrations and context-switches. Experimental results reveal that the FT-FS algorithm performs appreciably over an extensive sets of system scenarios pointing to the practical effectiveness of the scheme. In the next chapter, we consider a combined fault-tolerant and energy-aware design strategy for real-time safety-critical systems having heterogeneous multi-cores as the computing platform.

Chapter 5

Fault and Energy Aware Scheduling on Real-time Heterogeneous Dual-cores

In the earlier chapters, we have separately considered the energy-aware and fault-tolerant design strategies for real-time safety-critical systems, and have assumed the underlying hardware computing platform to be homogeneous. However, the nature of processing platforms used in embedded systems is changing over the years. To satisfy the computational demands of various applications, today, we observe an increased emphasis towards the integration of unrelated processing cores (i.e., heterogeneity) onto a single hardware platform [15,35]. For example, ARM has developed a heterogeneous processing architecture, called *ARM big.LITTLE* which has been deployed in cutting-edge mobile devices such as Samsung Galaxy Note 4, S10, etc. The big.LITTLE platform contains two types of cores, one of which is high-performance, called the *big* cores, while the other is of lower performance and power-efficient, referred to as *LITTLE* cores. Due to the differences in the internal microarchitectures of big and little cores, the same application/task may exhibit different timing as well as power characteristics on the different cores [100, 101]. Therefore, devising combined energy-aware and fault-tolerant design strategies for such heterogeneous platforms is a challenging and computationally demanding problem.

In this chapter, we present a standby-sparing based energy-aware fault-tolerant design strategy for heterogeneous systems. The chapter first describes the system model under consideration. Then, we present our proposed energy-aware fault-tolerant scheduling strategy to effectively handle transient processor faults. Important experimental

results which highlight the performance of the proposed methodology under various scenarios are discussed next. Finally, the chapter concludes by presenting a case study using MiBench benchmarks to illustrate the applicability of the proposed strategy in real world scenarios.

5.1 System Model and Problem Formulation

In this section, we describe various models under consideration in detail.

5.1.1 Platform and Application Model

We consider a real-time system consisting of a set of n independent periodic tasks ($T = \{T_1, T_2, ..., T_n\}$) to be executed on a heterogeneous dual-core processing platform. This system consists of a power-hungry, high-performance (big) core, and a power-efficient, relatively slow (little) core. In this work, we assume that tasks are executed in a *frame-based* manner [100, 117]. That is, all tasks in the system share same period, which is equal to the common deadline D. The worst case number of cycles required by a task T_i on a given core is denoted by C_i. However, T_i may take up to $e_i = C_i/f$ units of execution time to complete on that core when executed at the frequency level f. Due to the asymmetric nature of the cores, the same task may require different number of cycles and execution times on each of these cores. Therefore, each task T_i ($\in T$) is characterized by a two tuple (e_i^{HP}, e_i^{LP}), where e_i^{HP} and e_i^{LP} represent the worst case execution times of T_i on high-performance (denoted by HP) and low-power (denoted by LP) cores, respectively. It is assumed that e_i^{LP} and e_i^{HP} correspond to execution time under the maximum processing frequencies on LP and HP cores, (denoted by f_{max}^{LP} and f_{max}^{HP}), respectively.

5.1.2 Power Model

Due to the asymmetric nature of the cores, the HP and LP cores have different power consumption characteristics. The dynamic power consumption of a task T_i on any processing core is modeled as $P_i(f) = a_i f^3 + \alpha_i$, where a_i indicates the switching capacitance, f denotes the processing frequency of the task, and α_i is the frequency-independent

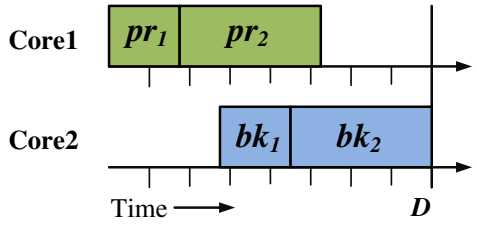


Figure 5.1: *A standby-sparing system*

power consumption [100]. Therefore, the same task may exhibit different power characteristics on different cores of a heterogeneous system.

Each processing core executes tasks in its high-power state and dissipates power as specified by the processing frequency and the characteristics of the executing task. In this work, we employ the *Dynamic Power Management (DPM)* technique on both cores to minimize the energy consumption. Therefore, when a core becomes idle (that is, not executing any tasks), DPM switch off the core to the low-power state. When a core transits between the high-power state to the low-power state, a specific amount of energy and time are consumed. Therefore, the minimum processor idle time required to compensate the cost of entering a low-power state is defined as the *break-even time* TI_{be} [66] of a processing core. In this work, we assume that the cost of entering a low-power state is negligible and so, TI_{be} is assumed to be zero. Let P_{idle}^{LP} and P_{idle}^{HP} denote the power consumption of LP and HP cores at their low-power states, respectively. The overall energy consumption within a frame is determined by aggregating the energy consumption of all cores in that frame.

5.1.3 Fault Model

In this work, we employ a standby-sparing technique in which one processing core is designated as *primary* and the other as the *spare*. Figure 5.1 depicts a standby-sparing system. Each task T_i has two versions, namely, *primary copy* (denoted by pr_i) and *backup copy* (denoted by bk_i). bk_i has exactly same timing parameters as that of pr_i. As per the motivation of energy-awareness from literature [100], we assign primary copies of tasks to low-power LP primary core and their backup copies to high-performance HP spare

core, respectively. Whenever a primary copy completes, fault detection mechanisms such as acceptance or sanity tests [62] are conducted to detect a transient fault. If a fault is not detected, (that is, primary completes successfully), the corresponding backup copy on the spare core is deallocated from the schedule dynamically. We assume that each task (primary copy) encounters at most one transient fault and at any point in time, system is able to handle at most k transient faults per frame.

Problem Formulation: Given a set of real-time tasks to be executed on a heterogeneous dual-core system and a number of transient faults to be tolerated, develop an efficient scheduling strategy which

- satisfies execution and deadline constraints of all tasks,
- tolerates a specified number of faults, and
- minimizes overall energy consumption of the system.

5.2 FENA-SCHED: Fault-tolerant Energy-aware scheduling Scheme

In this section, we describe the overall working of our proposed fault-tolerant energy-aware scheduling mechanism, *FENA-SCHED*.

Figure 5.2 depicts our proposed scheduling framework. This framework mainly consists of an offline scheduler (Module 3) which considers different models (system, power, fault models, collectively represented as Module 1 and 2) presented in Section 5.1, and generates a fault-tolerant, energy-aware schedule (Module 4). This offline schedule is further enhanced at run-time by observing the completions of the primary copies of tasks in the system (Module 5). Before describing the details of *FENA-SCHED*, we first investigate a design constraint in the standby-sparing *DPM-enabled* system, that is, designating which core to assign as the *primary* and correspondingly, *spare*. There are two configurations:

- Configuration-1: Power-hungry, high-performance HP core as primary and power-efficient, modest performance LP core as spare.

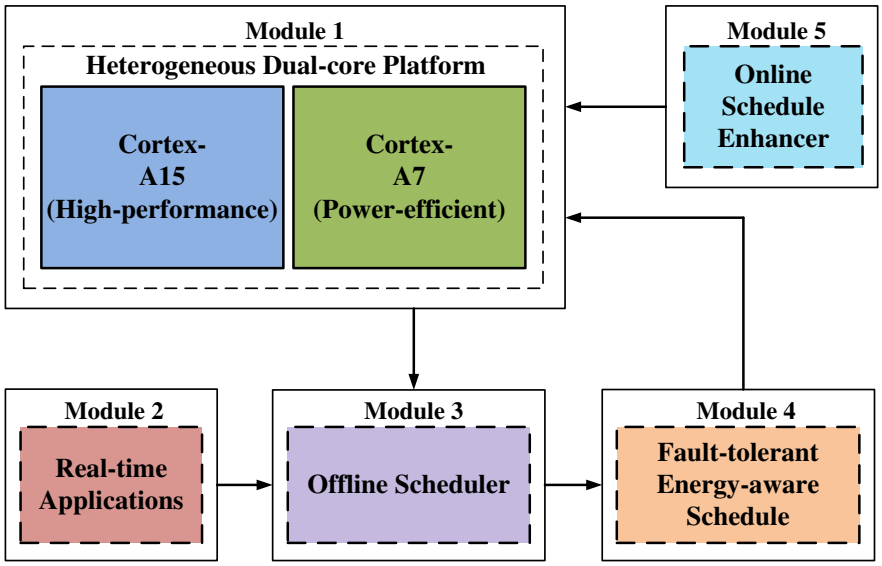


Figure 5.2: *Proposed Framework*

- Configuration-2: High-performance HP core as spare and power-efficient LP core as primary.

In a standby-sparing system, the primary copy of a task is executed in most of the cases, and corresponding backup copy is activated only when its primary fails. In a non-faulty scenario, spare core can always put to its low-power state. It may be inferred that Configuration-2 yields better energy savings as compared to Configuration-1. This is due to the fact that in Configuration-2, primary copies of tasks are assigned to the power-efficient LP core whose power characteristics are much lower than that of the power-hungry HP core.

Example 1: Consider a heterogeneous dual-core system with $f_{max}^{LP} = 0.8$ and $f_{max}^{HP} = 1.0$. Here, f_{max}^{LP} and f_{max}^{HP} denote the normalized maximum frequencies of the LP core and HP core, respectively. We assume, $P_{idle}^{LP} = 0.02, P_{idle}^{HP} = 0.05$, and $D = 100ms$. The power consumption parameters for all tasks are given by: $a_i^{HP} = 1.0, \alpha_i^{HP} = 0.1, a_i^{LP} = 0.3$, and $\alpha_i^{LP} = 0.03$. In a non-faulty and fully loaded primary core scenario (where the spare core is completely idle), Configuration-1 yields an overall energy consumption of $112mJ$, while Configuration-2 consumes only $23.36mJ$ of energy, showing an improvement of 79.14%. □

Based on the inference from Example 1, we have used Configuration-2 in our design; that is, the *high-performance HP core is used as spare while the power-efficient LP is employed as the primary.*

5.2.1 Scheduling Strategy

Now, we discuss our proposed energy-aware fault-tolerant strategy *FENA-SCHED* (refer Algorithm 9), in detail. This strategy can handle at most k (given) transient faults per frame at any point in time and takes the advantage of *backup-backup (BB) overloading* [46, 47]. In BB-overloading, backup copies of multiple tasks are scheduled during the same time interval in order to make efficient utilization of available processing core time. Therefore, in an energy-aware perspective BB-overloading reduces the overall energy consumption of the system.

Algorithm 9 first creates a list Pr_List of tasks in non-increasing order of their execution times (e_i^{LP}) on LP core (Step 2). Steps 3-10 determine a schedule for primary copies of tasks in the given task set T. Algorithm 9 extracts each task T_i from Pr_List and schedules its primary copy on LP core by assigning $start\_time$, a time at which T_i will start its execution on the corresponding core. The admission control step of Algorithm 9 (Step 5) verifies the schedulability of the given task set T. If T is schedulable, Algorithm 9 creates a list Bk_List of tasks in non-increasing order of their execution times (e_i^{HP}) on HP core (Step 12). Then it computes *BB overloading window* ($reserve\_cap$) from the first k tasks in Bk_List based on their execution times (e_i^{HP}) on HP core and reserves $reserve\_cap$ unit of backup slots on HP core as late as possible (Steps 13-15). Therefore, instead of assigning time slots for all backup copies, Algorithm 9 reserves only first k backup copies time slots on HP core to minimize overall energy consumption as well as to tolerant k number of faults.

Now, we demonstrate the working of *FENA-SCHED* through a set of illustrative examples and show its effectiveness over an existing state-of-the-art work. In this work, we focus on the energy consumption of the generated fault-tolerant offline schedule. The primary copies of tasks are scheduled in the non-increasing order of their execution times on primary core. Similarly, the backup copies of tasks are scheduled in the non-increasing

order of their execution times on spare core.

ALGORITHM 9: *FENA-SCHED*

Input: Heterogeneous dual-core system, T: Set of n real-time tasks, k: Number of transient faults

Output: Fault-Tolerant Task Schedule

`// primary: power-efficient core (LP); spare: high-performance core (HP)`

1 Initialize $start\_time = 0$, $reserve\_cap = 0$;
2 Create a list Pr_List of tasks in non-increasing order of their execution times (e_i^{LP}) on LP core;
3 **while** $Pr\_List \neq \emptyset$ **do**
4 Extract the first task T_i from Pr_List;
5 **if** $start\_time + e_i^{LP} \leq D$ **then**
6 Schedule primary copy of T_i on LP core at $start\_time$;
7 $start\_time = start\_time + e_i^{LP}$;
8 **else**
9 Task set T is not schedulable;
10 exit;
11 **if** T *is schedulable* **then**
12 Create a list Bk_List of tasks in non-increasing order of their execution times (e_i^{HP}) on HP core;
 `// Compute BB-overloading window`
13 **for** *first k tasks in* Bk_List **do**
14 $reserve\_cap = reserve\_cap + e_i^{HP}$;
15 Reserve $reserve\_cap$ unit of backup slots on HP core as late as possible;

Example 2: Consider a system consisting of a set of four real-time tasks T_1, T_2, T_3 and T_4 to be executed on a homogeneous dual-core platform. The worst case execution times (in ms) of these four tasks are assumed to be 14, 18, 10 and 6, respectively. Here, both primary as well as spare are HP cores. This system is characterized by assuming, $P_{idle}^{HP} = 0.05, f_{max}^{HP} = 1.0$ and $D = 100ms$. For all tasks, $a_i^{HP} = 1.0$ and $\alpha_i^{HP} = 0.1$. Figure 5.3 depicts the schedule of primary and backup tasks on this homogeneous system. It may be observed from Figure 5.3 that each HP core has an idle time of 52 ms. Therefore, total energy consumption of the system during this idle time becomes $52 \times 2 \times 0.05 = 5.2mJ$. Similarly, total energy consumption of the system when cores

Table 5.1: *A Sample Task Set*

	T_1	T_2	T_3	T_4
e_i^{HP}	14	18	10	6
e_i^{LP}	20	24	16	10

become active is given by $48 \times 2 \times (1.0 \times 1.0^3 + 0.1) = 105.6mJ$. Therefore, this configuration yields an overall energy consumption of $5.2 + 105.6 = 110.8mJ$. □

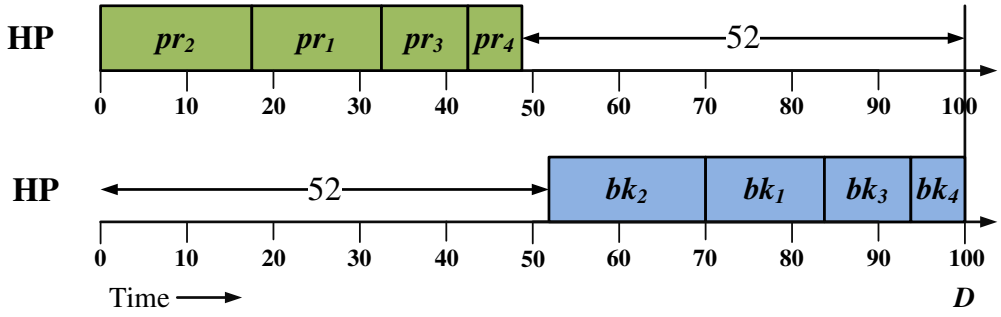


Figure 5.3: *Homogeneous System*

Example 3: Let us continue with the same system scenario discussed in Example 2, with the exception that these four real-time tasks now have to execute on a heterogeneous dual-core platform consisting of a power-efficient LP core and a high-performance HP core. In this heterogeneous system, power characteristics of the LP core are much lower than that of the power-hungry HP core. This system is characterized by the assumptions, $f_{max}^{LP} = 0.8, f_{max}^{HP} = 1.0, P_{idle}^{LP} = 0.02, P_{idle}^{HP} = 0.05$ and $D = 100ms$. Table 5.1 shows the worst case execution times (in ms) of these four tasks on HP and LP cores. For all tasks, $a_i^{HP} = 1.0, \alpha_i^{HP} = 0.1, a_i^{LP} = 0.3$ and $\alpha_i^{LP} = 0.03$. The core and task parameters are taken from [100]. Now, we consider a standby-sparing configuration named as *SlowerP* discussed in [100] which assigns primary copies of all tasks to the LP core and their backup copies to the HP core. Figure 5.4 depicts the schedule corresponding to this *SlowerP* configuration. It may be observed from Figure 5.4 that the idle times of

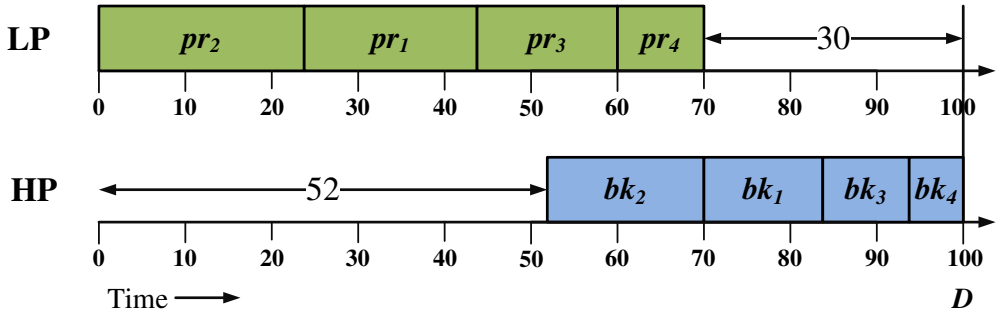


Figure 5.4: *SlowerP Configuration [100]*

LP and HP cores are 30 ms and 52 ms, respectively. Total energy consumption of the LP core during both active and idle times become $(70 \times (0.3 \times 0.8^3 + 0.03)) + (30 \times 0.02) = 13.452mJ$. Similarly, total energy consumption of the HP core during both active and idle times is given by $(52 \times 0.05) + (48 \times (1.0 \times 1.0^3 + 0.1)) = 55.4mJ$. Therefore, the overall energy consumption of this *SlowerP* configuration becomes $13.452 + 55.4 = 68.852mJ$, giving an improvement of 37.85% as compared to the homogeneous system. □

Example 4: Let us continue with the same system scenario discussed in Example 3. It may be noted from Figure 5.4 that the *SlowerP* configuration is oblivious to the number of faults to be tolerated and hence, it assigns backup slots for all primaries in the generated schedule. On the other hand, our proposed strategy *FENA-SCHED* reserves only a fixed number of backup slots based on the number of faults to be tolerated (here, $k = 2$) and allows *backup-backup (BB) overloading* within these backup slots. Figure 5.5 depicts our proposed fault-tolerant strategy *FENA-SCHED*. It may be observed from Figure 5.5 that *FENA-SCHED* increases the total idle times in the HP core using BB-overloading. This reduces total energy consumption of the HP core to $(68 \times 0.05) + (32 \times (1.0 \times 1.0^3 + 0.1)) = 38.6mJ$. Therefore, *FENA-SCHED* yields an overall energy consumption of $13.452 + 38.6 = 52.052mJ$ and produces an improvement of 24.4% over the *SlowerP* configuration and 53.02% over the homogeneous system. □

5.2.2 Run-time Behavior

During online execution, when a primary copy completes, fault detection mechanisms such as acceptance or sanity tests [62] are conducted to detect a transient fault. If a fault

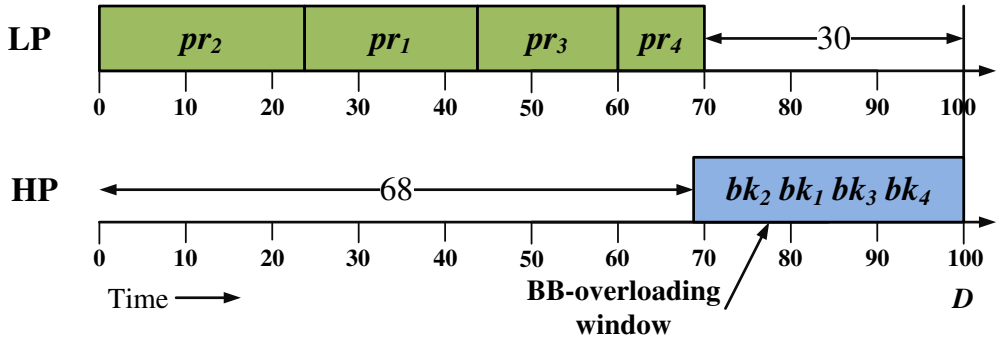


Figure 5.5: *Proposed Strategy,* FENA-SCHED

is not detected, (that is, primary completes successfully), the corresponding backup copy on the spare core is deallocated from the schedule dynamically. Moreover, the successful completion of a primary copy allows us to dynamically readjust the *BB-overloading window* in order to handle k transient faults on the remaining primary copies. Due to the successful completion of each primary copy, the size of the *BB-overloading window* is readjusted based on the HP core execution times of k remaining tasks and results in shrinking the window size dynamically. If a transient fault is detected, the backup copy of the failed primary executes up to its completion. It may be noted from the literature that the actual execution time of a task at run-time is typically lower than its worst case execution time. Therefore, at run-time tasks may finish before their offline computed completion times. This allows a provision for the remaining unexecuted tasks to start their execution earlier than their offline computed start times.

Example 5: Let us again continue with the same system scenario discussed in Example 4. The offline fault-tolerant schedule generated by *FENA-SCHED* is shown in Figure 5.5. Now, we discuss the behavior of *FENA-SCHED* at run-time (shown in Figure 5.6). Consider a scenario in which pr_2 completes its execution successfully at time=24 (at its offline computed finish time). As a result, bk_2 (the backup copy of pr_2) is deallocated from the schedule dynamically. Moreover, the size of the *BB-overloading window* is readjusted dynamically based on HP core execution times of the remaining two ($k = 2$) tasks T_1 and T_3. It may also be observed from Figure 5.6 that early completion of pr_2 allows the primary copy pr_1 to start its execution at time=20. □

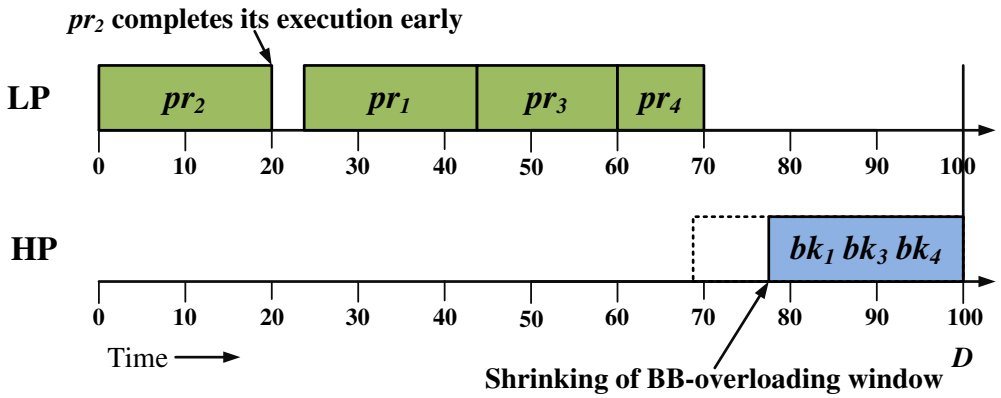


Figure 5.6: FENA-SCHED*: Run-time behavior*

5.3 Experiments and Results

We have evaluated the performance of our proposed strategy *FENA-SCHED* through an extensive set of simulation studies (conducted over an experimental framework which is described in the next subsection) and compared its performance against an existing standby-sparing configuration named *SlowerP* [100].

SlowerP: This standby-sparing configuration assigns primary copies of all tasks to LP core and their backup copies to HP core. However, this configuration is oblivious to the number of faults to be tolerated and hence, it assigns backup slots for all primary tasks on LP core to HP core. We employ DPM on both LP and HP cores to reduce the energy consumptions.

5.3.1 Experimental Setup

We have simulated different heterogeneous dual-core systems which consist of a high-performance HP core with $f_{max}^{HP} = 1.0$ and a power-efficient LP core with f_{max}^{LP} varying from 0.6 to 0.9. The experimental framework generates different task sets for various experiments. The total utilization U is computed with reference to the power-efficient LP core and normalized with its maximum frequency. Therefore, $U = (\Sigma \frac{C_i^{LP}}{D})/f_{max}^{LP}$. Given the total utilization (U) and the number of tasks to be generated (n), the individual task utilization on the low-power core have been generated from a uniform distribution. The frame deadline D is set to 200 ms for most of the experiments. For all tasks, a_i^{HP} and

α_i^{HP} are set to 1.0 and 0.1, respectively. Similarly, $P_{idle}^{LP} = 0.02$ and $P_{idle}^{HP} = 0.05$. Each result data point is the average obtained over 100 different task sets, each containing $n = 10$ tasks. The number of faults (k) to be handled by the system is varied in the range $[1, 5]$.

From literature, it is familiar that same task may require different timing as well as power consumption characteristics on heterogeneous systems [97, 100]. Therefore, we define a time-scaling factor $tscale_i = \frac{C_i^{LP}}{C_i^{HP}}$, and a power-scaling factor $pscale_i = \frac{P_i^{LP}}{P_i^{HP}}$ for each task T_i, as discussed in [100, 101]. The values of $tscale_i$ and $pscale_i$ are randomly generated within the ranges $1.4 \leq tscale_i \leq 2.3$ and $1.4 \leq 1/(tscale_i \times pscale_i) \leq 2.1$, as suggested in [97].

5.3.2 Experimental Results

Experiment 1- Impact of utilization: In this experiment, we have varied the system load on a heterogeneous system with $f_{max}^{HP} = 1.0$ and $f_{max}^{LP} = 0.8$. Figure 5.7 depicts the effect of utilization on normalized energy consumption. The values of n and k are set to 10 and 4, respectively. Here, the utilization shown in the X-axis is considered as regards the power-efficient LP core. With reference to the energy consumption of *SlowerP* at $U = 1.0$, we normalize the obtained results. It may be observed from Figure 5.7 that as expected, the energy consumption of both *FENA-SCHED* and *SlowerP* increases when system load increases. This is because for a given number of tasks, the average individual task utilization increases with an increase in the total utilization and hence, task execution time increases. This reduces the idle times of both processing cores, leads to higher energy consumption. It may also be observed that in all system load conditions, our proposed scheme *FENA-SCHED* outperforms *SlowerP*. This is due to the fact that *FENA-SCHED* reserves only a fixed amount of backup slots on HP core with respect to the number of faults to be tolerated and allows *backup-backup (BB) overloading* within these backup slots. For example, when $U = 0.6$, *FENA-SCHED* and *SlowerP* yield an overall energy consumption of $82.41mJ$ and $110.21mJ$, respectively. When utilization increases to $U = 1.0$, their overall energy consumptions become $127.8mJ$ and $174.36mJ$, respectively.

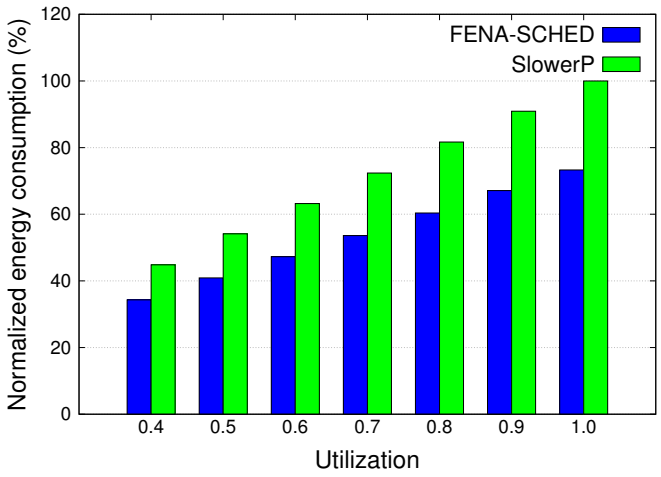


Figure 5.7: *Impact of utilization*

Experiment 2- Impact of number of faults: In this experiment, we have varied the number of faults (k) on a moderately loaded heterogeneous system, and corresponding result is shown in Figure 5.8. Here, $f_{max}^{HP} = 1.0, f_{max}^{LP} = 0.8$ and $U = 0.6$ (60% load on power-efficient LP core). The value of n is set to 10. With reference to the energy consumption of *SlowerP* at $k = 5$, we normalize the obtained results. It may be observed from Figure 5.8 that the energy consumption of *FENA-SCHED* increases when the number of faults in the system increases whereas *SlowerP* exhibits a constant energy consumption. This is because the amount of backup slots reserved on the HP core by *FENA-SCHED* increases with an increase in k. This reduces the idle times on HP processing core and results in higher energy consumption. On the other hand, *SlowerP* is oblivious to the number of faults to be tolerated and hence, it assigns backup slots for all primary tasks on LP core to HP core, results in a constant energy consumption. It may also be observed that even though the energy consumption of *FENA-SCHED* increases with k, it always performs better than *SlowerP*. This is due to the fact that *FENA-SCHED* reserves only a fixed amount of backup slots on HP core with respect to the number of faults to be tolerated and allows *BB overloading* within these backup slots. For example, when $k = 2$, *FENA-SCHED* and *SlowerP* yield an overall energy consumption of $70mJ$ and $110.21mJ$, respectively. When k increases to 5, the over-

all energy consumption of *FENA-SCHED* becomes $87.98mJ$, whereas that of *SlowerP* remains same as $110.21mJ$.

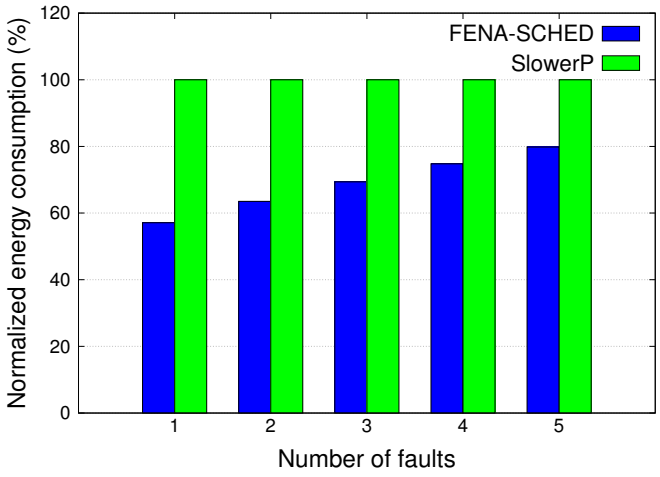


Figure 5.8: *Impact of number of faults*

Experiment 3- Impact of deadline: Figure 5.9 shows the impact of deadline on a heterogeneous system with $f_{max}^{HP} = 1.0$, $f_{max}^{LP} = 0.8$ and $U = 0.6$. The values of n and k are set to 10 and 4, respectively. With reference to the energy consumption of *SlowerP* at $D = 200$, we normalize the obtained results. It may be observed from Figure 5.9 that as expected, the energy consumption of both *FENA-SCHED* and *SlowerP* increases when deadline increases from $100ms$ to $200ms$. This is because, for a given number of tasks and total utilization, task execution time increases with an increase in the deadline, leading to higher energy consumption. For example, when $D = 100ms$, *FENA-SCHED* and *SlowerP* yield an overall energy consumption of $41.24mJ$ and $55.13mJ$, respectively. When deadline increases to $200ms$, their overall energy consumptions become $82.41mJ$ and $110.21mJ$, respectively.

Experiment 4- Impact of the maximum speed of the LP core: In this experiment, we have varied the maximum speed of the LP core (f_{max}^{LP}) on a moderately loaded heterogeneous system. Figure 5.10 depicts the effect of f_{max}^{LP} on normalized energy consumption. Here, $f_{max}^{HP} = 1.0, n = 10$ and $U = 0.6$. The value of k is set to 4. With reference to the energy consumption of *SlowerP* at $f_{max}^{LP} = 0.9$, we normalize the ob-

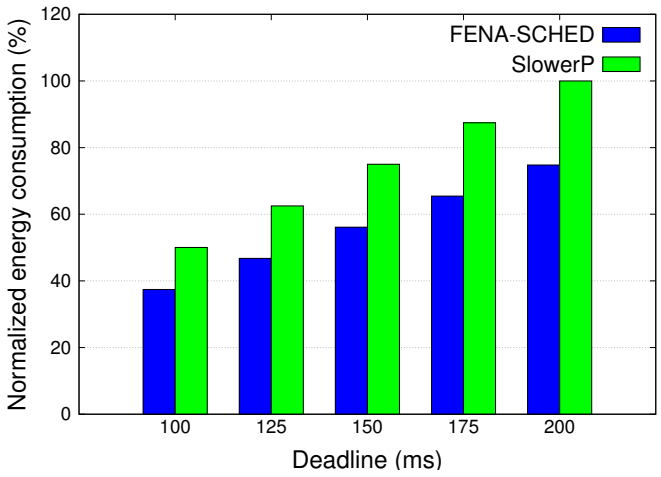


Figure 5.9: *Impact of deadline*

tained results. It may be observed from Figure 5.10 that for a fixed number of tasks and total utilization, the energy consumption of both *FENA-SCHED* and *SlowerP* increases when f_{max}^{LP} increases. This is because the power consumption characteristics of the LP core heavily depends on its operating frequency. Therefore, the power consumption of LP core increases as f_{max}^{LP} increases and results in higher energy consumption. For example, when $f_{max}^{LP} = 0.6$, *FENA-SCHED* and *SlowerP* yield an overall energy consumption of $75.52mJ$ and $96.37mJ$, respectively. When f_{max}^{LP} increases to 0.9, their overall energy consumptions become $85.86mJ$ and $117.13mJ$, respectively.

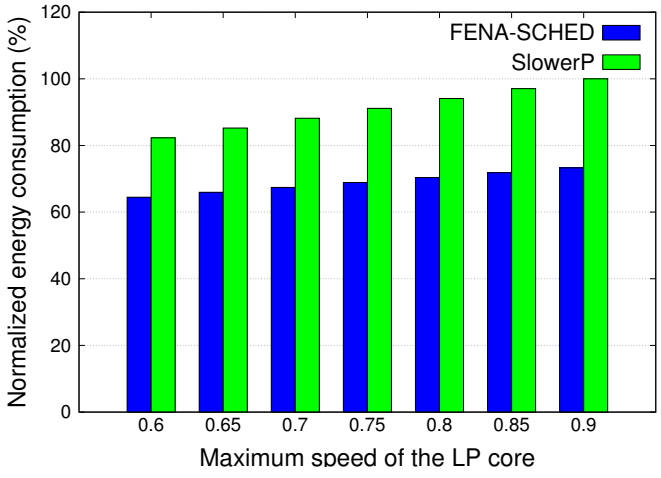


Figure 5.10: *Impact of the maximum speed of the LP core*

Experiment 5- Impact of number of tasks: Here, the number of tasks (n) on a moderately loaded heterogeneous system has been varied. Figure 5.11 depicts the effect of n on normalized energy consumption. Here, $f_{max}^{HP} = 1.0, f_{max}^{LP} = 0.8$ and $U = 0.6$. The value of k is set to 4. With reference to the energy consumption of *SlowerP* at $n = 30$, we normalize the obtained results. It may be observed from Figure 5.11 that the energy consumption of *FENA-SCHED* decreases when the number of tasks in the system increases, whereas *SlowerP* exhibits a constant energy consumption. This is due to the fact that for a fixed workload, the average individual task utilization decreases with an increase in the number of tasks and hence, task execution time decreases. This reduces the length of backup slots reserved on the HP core by *FENA-SCHED* (for handling k faults), leads to lower energy consumption. On the other hand, *SlowerP* is oblivious to the number of faults to be tolerated and hence, it assigns backup slots for all primary tasks on the LP core, to HP core, and this results in constant energy consumption. For example, when $n = 10$, *FENA-SCHED* and *SlowerP* yield an overall energy consumption of $82.41mJ$ and $110.21mJ$, respectively. When n increases to 30, the overall energy consumption of *FENA-SCHED* becomes $66.12mJ$, whereas that of *SlowerP* remains same as $110.21mJ$.

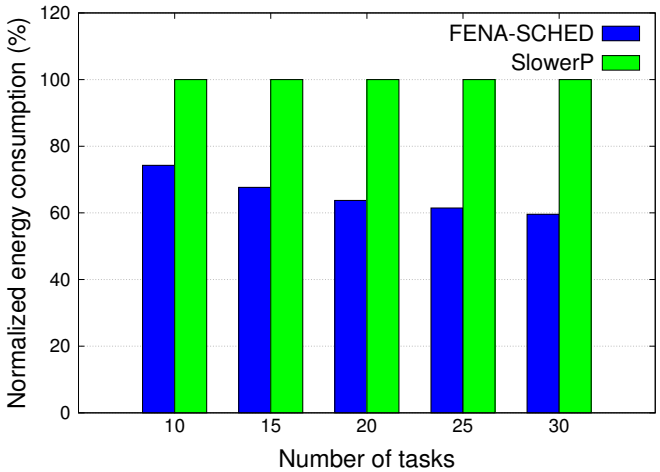


Figure 5.11: *Impact of number of tasks*

5.4 Case Study

In this section, we present a case study using *MiBench* benchmarks [50] to illustrate the generic applicability of our proposed strategy in real world scenarios.

We selected a set of 6 tasks from *MiBench* benchmarks (Automotive and Industrial Control category), and their execution times (taken from [39]) have been listed in Table 5.2. Now, we consider a scenario where these six tasks T_1, T_2, T_3, T_4, T_5, and T_6 have to execute on a heterogeneous dual-core system consisting of one high-performance HP core and one power-efficient LP core. This system is characterized by assuming, $f_{max}^{LP} = 0.8, f_{max}^{HP} = 1.0, P_{idle}^{LP} = 0.02, P_{idle}^{HP} = 0.05$, and $D = 2500ms$. We assume that the execution times of tasks listed in Table 5.2, represent their LP core execution times. Table 5.3 shows the worst case execution times (in ms) of these six tasks on HP and LP cores. For all tasks, $tscale_i = 2, a_i^{HP} = 1.0, \alpha_i^{HP} = 0.1, a_i^{LP} = 0.3$, and $\alpha_i^{LP} = 0.03$.

Table 5.2: *The execution time of the benchmark tasks [39]*

Benchmark	**Execution Time (in ms)**	**Task Label**
qsort	454	T_1
basicmath	708	T_2
bitcount	497	T_3
susan (smoothing)	259	T_4
susan (edges)	19	T_5
susan (corners)	11	T_6

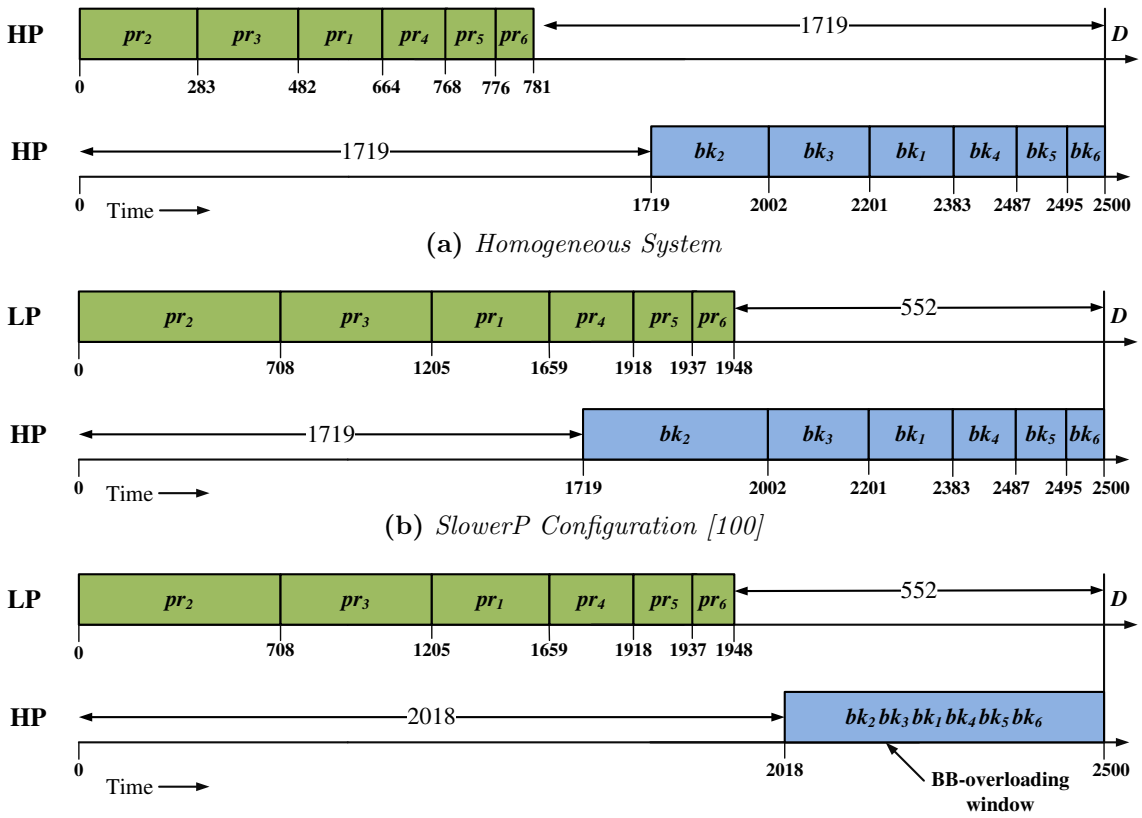
Now, consider a scenario in which these six tasks are executed on a homogeneous dual-core (HP cores) platform with $f_{max} = 1.0$. Figure 5.12(a) depicts primary and

Table 5.3: *The execution times of the benchmark tasks on HP and LP cores*

	T_1	T_2	T_3	T_4	T_5	T_6
e_i^{HP}	182	283	199	104	8	5
e_i^{LP}	454	708	497	259	19	11

backup task schedules on this homogeneous system. Here, both the primary as well as spare are HP cores. It may be observed from Figure 5.12(a) that each HP core has an idle time of 1719 ms. Therefore, total energy consumption of the system during this idle time becomes $1719 \times 2 \times 0.05 = 171.9mJ$. Similarly, total energy consumption of the system when cores become active is given by $781 \times 2 \times (1.0 \times 1.0^3 + 0.1) = 1718.2mJ$. Therefore, this configuration yields an overall energy consumption of $171.9 + 1718.2 = 1890.1mJ$.

In a heterogeneous dual-core system, we have a power-efficient LP core whose power characteristics are much lower than that of the power-hungry HP core. Now, we consider the *SlowerP* configuration, discussed in [100] which assigns primary copies of all tasks to the LP core and their backup copies to the HP core. Figure 5.12(b) depicts the schedule corresponding to this *SlowerP* configuration. It may be observed from Figure 5.12(b) that the idle times of LP and HP cores are 552 ms and 1719 ms, respectively. Total energy consumption of the LP core during both active and idle times become $(1948 \times (0.3 \times 0.8^3 + 0.03)) + (552 \times 0.02) = 368.692mJ$. Similarly, total energy consumptions of the HP core during both active and idle times are obtained as: $(1719 \times 0.05) + (781 \times (1.0 \times 1.0^3 + 0.1)) = 945.05mJ$. Therefore, the overall energy consumption of this *SlowerP* configuration becomes $368.692 + 945.05 = 1313.742mJ$, showing an improvement of 30.49% over the homogeneous system. It may be noted that this *SlowerP* configuration is oblivious to the number of faults to be tolerated and hence, it assigns backup slots for all primaries in the generated schedule. On the contrary, our proposed strategy *FENA-SCHED* reserves only a fixed number of backup slots based on the number of faults to be tolerated (here, $k = 2$) and allows BB-overloading within these backup slots. Figure 5.12(c) depicts the schedule corresponding to our proposed fault-tolerant strategy *FENA-SCHED*. It may be observed from Figure 5.12(c) that *FENA-SCHED* increases



(a) *Homogeneous System*

(b) *SlowerP Configuration [100]*

(c) *Proposed Strategy,* FENA-SCHED

Figure 5.12: *Different configuration scenarios*

the total idle times in the HP core using BB-overloading. This reduces total energy consumption of the HP core to $(2018 \times 0.05) + (482 \times (1.0 \times 1.0^3 + 0.1)) = 631.1mJ$. Therefore, *FENA-SCHED* yields an overall energy consumption of $368.692 + 631.1 = 999.792mJ$ and produces an improvement of 23.89% over the *SlowerP* configuration and 47.1% over the homogeneous system.

5.5 Summary

This chapter presents a standby-sparing based fault-tolerant energy-aware scheduling strategy, named FENA-SCHED, for heterogeneous systems. For a DPM-enabled system, we found that designating power-efficient (modest performance) core as primary and power-hungry (high-performance) core as spare yields better energy savings as compared

to its counterpart. In order to minimize overall energy consumption and to tolerate a given number of faults, FENA-SCHED reserves only a fixed number of backup slots on the high-performance core and takes the advantage of backup-backup overloading. Experimental results reveal that FENA-SCHED performs appreciably over an extensive set of system scenarios pointing to the practical effectiveness of the scheme and is able to significantly improve energy savings of the system, compared to the state-of-the-art work. In the next chapter, we present a formal fault detection and isolation (that is, fault diagnosis) framework for the design of safety-critical systems.

Chapter 6

A Formal Design Strategy for Fault Diagnosis in Safety-critical Systems

In the previous chapters, we have assumed that faults are always detectable, and have aimed towards the design of efficient fault-tolerant procedures that provide functional correctness in the presence of faults. However, enforcement of such fault tolerance can only be achieved through the incorporation of safe design methodologies which enable efficient active monitoring and detection of unobservable faults in the system. Therefore, it is desirable to incorporate efficient fault diagnosis (detection and isolation) strategies in the construction of safety-critical systems.

Several approaches towards fault diagnosis have been reported in the literature. These approaches are typically categorized based on parameters such as the degree of automation involved in diagnosis and whether the detection methodology employs abstraction models of the system, or are model-free. Semi-automated fault diagnosis, used in certain large and complex systems, necessitates the involvement of a human operator for effective fault detection [74, 98]. On the other hand, automated diagnosis is free of human intervention [104, 115]. For large and complex systems, it is also sometimes difficult to derive abstraction models which aid the process of effective diagnosis. For these systems, model-free fault detection techniques like spectrum analysis [78], limit checking [54], expert systems [73], are used. However, it has now been widely accepted that model-based representations are more suited towards all mechanisms related to automated reasoning, ranging from fault diagnosability to stability analysis in complex

systems. Examples of commonly used model-based techniques are fault-tree analysis [48], analytical redundancy (ARR) [43], Hybrid System (HS) [22], Principal Component Approach (PCA) [38], Discrete Event System (DES) models [58, 81, 104, 110], etc. DES is an important model-based technique that is often used for automated failure diagnosis in a wide range of systems primarily because of its systematic modeling approach and the simplicity of its associated algorithms [103, 104, 115]. DES allows a structured, hierarchical modeling procedure to generate composite models of complex systems from individual component models and then allows the incorporation of diagnostics over these composite models. Through this approach, DES methods are able to avoid the often tedious and involved efforts that are required to construct detailed one-shot monolithic models of the complex system to be diagnosed. Further, complex systems which even include continuous dynamics, can also be viewed as DESs at a certain level of abstract discretization. DES provides a fault detection mechanism known as *Diagnoser* which actively monitors behavior of the system and detects the occurrence of unexpected events (faults) in an effective way. It may be noted from the literature that the complexity of the diagnosis processes, that is, constructing a diagnoser and testing its diagnosability, is exponential in the number of system states [28, 103, 115]. This may lead to prohibitively huge state-space requirements in the design of diagnosers for large and complex systems.

As a spin-off from our efforts related to energy-aware and fault-tolerant scheduling, this chapter discusses the development of an efficient, low-overhead, DES-based fault diagnosis design strategy for safety-critical systems. The proposed formally constructed fault detection and isolation mechanism actively monitors the system and detects the presence of unobservable faults in the system. An important emphasis of the work in this chapter is the design of a modular light weight fault diagnosis mechanism which consumes lower state space compared to the state-of-the-art.

The chapter first describes the working of a generic DES-based fault diagnosis framework using a practical *Electronic Fuel Injection* (EFI) system. Then, we present our proposed light weight fault diagnosis mechanism. Finally, the chapter concludes by presenting important experimental results used to evaluate the performance of the proposed

fault diagnosis design strategy.

6.1 DES Modeling and Fault Diagnosis of an Electronic Fuel Injection System

In this section, we illustrate the state-based DES framework presented in Chapter 2.4, by employing it to model a practical Electronic Fuel Injection (EFI) system [44]. EFI is an electro-mechanical control system in modern automotive vehicles, which monitors and controls the operation of an engine by supplying a designated amount of air-fuel mixture combined in a precise ratio, into the engine cylinders. Figure 6.1 depicts the schematic diagram of an EFI system.

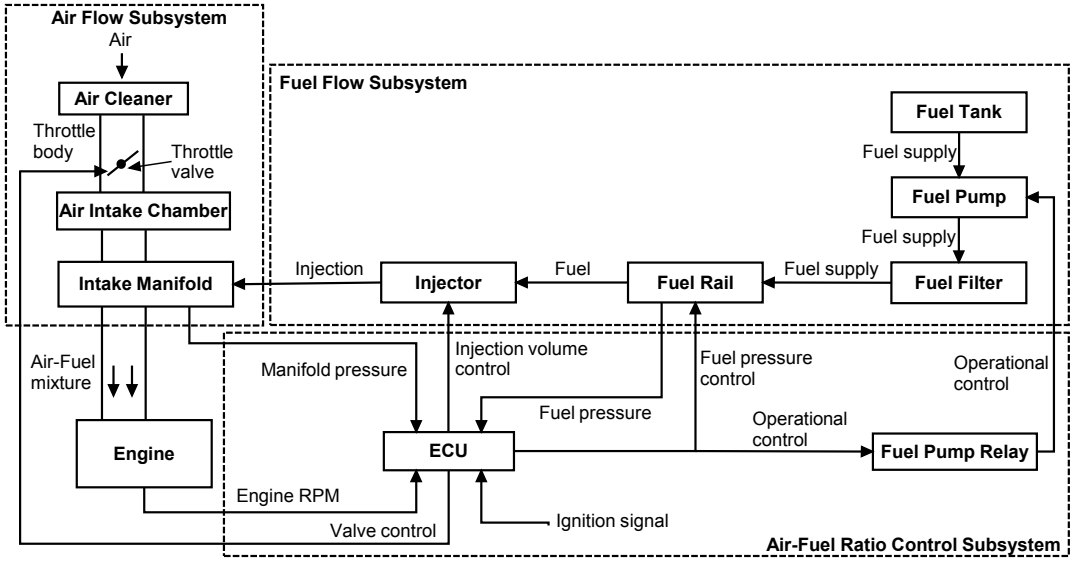


Figure 6.1: *The electronic fuel injection system.*

6.1.1 Functioning of the EFI System

Air enters into the intake manifold through the air cleaner, throttle body and air intake chamber of the '*Air Flow Subsystem*'. The amount of air entering into the engine can be controlled by adjusting the position of the throttle valve present in the throttle body. This varying air intake volume can be measured by monitoring the pressure in the intake manifold. Fuel is pumped from the tank to the injector through the fuel filter and rail

by an electric fuel pump which is controlled by a fuel pump relay (see the block named '*Fuel Flow Subsystem*' shown in Figure 6.1). The pressure regulator located at the fuel rail maintains a constant fuel pressure across the fuel injector. The injector injects the fuel into the intake manifold where it is mixed with air and the resulting mixture flows into the engine cylinders for burning. The *Electronic Control Unit* (ECU) is responsible for monitoring and controlling different engine functions by taking as input, information from various sensors located at different parts of the engine. ECU determines the right ratio for the air-fuel mixture based on the engine's RPM and the volume of air in the intake manifold. It then signals the injector to deliver the correct fuel quantity. The block named '*Air-Fuel Ratio Control Subsystem*' shown in Figure 6.1 depicts this control behavior of the ECU.

6.1.2 DES Model of the EFI System

The DES model of the EFI system is defined as $G = \langle X, S, \Im, X_0 \rangle$. Each state $x \in X$ in the EFI system model is distinguished by an enumeration of the set of state variables S. There exist two pressure sensors and one exhaust oxygen sensor in the EFI system. The control commands issued by the *ECU* and outputs of the sensors are considered as the state variables. Table 6.1 summarizes the state variables and their meaning. The EFI system represented by model G is shown in Figure 6.2. In order to model the faulty behaviors in G, we assume a 'Stuck_Closed' failure of the throttle valve in the *Air Flow Subsystem* (fault type F_1) and a 'Pump_On' failure of the fuel pump in the *Fuel Flow Subsystem* (fault type F_2). The control subsystem is assumed to be fault-free in our case.

The model G has a set of states $X = \{x_1, x_2, ..., x_{12}\}$, and a set of transitions $\Im = \{\tau_1, \tau_2, ..., \tau_{12}, \tau_{F_1}, \tau_{F_2}\}$. The occurrence of the fault types F_1 and F_2 in the system model G are represented through unmeasurable transitions τ_{F_1} and τ_{F_2}, respectively. All states reachable through the transition τ_{F_i} are faulty (of type F_i) and labeled with F_i, i.e., $x_d(C) = \{F_i\}$,[1] where $i \in \{1, 2\}$. Corresponding to the nominal states, $x_d(C) = \{N\}$. An enumeration of the variables corresponding to each state in G is

[1] d denotes the state number.

Table 6.1: *State variables and their meaning.*

State variable	Meaning
S_V	Status of the throttle valve (Open (1)/Closed (0))
S_P	Status of the fuel pump (ON (1)/OFF (0))
S_{OS}	Readings of the oxygen sensor (Presence (O)/Absence of Oxygen (NO))
S_{VS}	Readings of the air-pressure sensor (PS1) located in the intake manifold (Pressure (P_1)/No Pressure (NP_1))
S_{PS}	Readings of the fuel-pressure sensor (PS2) located in the fuel rail (Pressure (P_2) /No Pressure (NP_2))
C	Unmeasurable status variable associated with a state

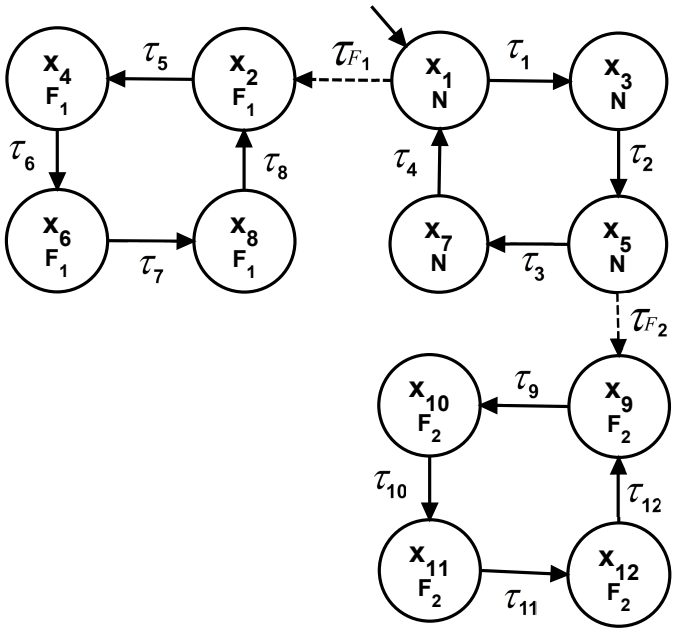


Figure 6.2: *The model of the EFI system: G.*

presented in Table 6.2. It may be noted that, all state variables except C are measurable, that is, $S_m = \{S_V, S_P, S_{OS}, S_{VS}, S_{PS}\}$ and $S_u = \{C\}$. We have modeled the

'Stuck_Closed' fault only from initial state x_1 and 'Pump_On' fault only from state x_5 because the corresponding fault causing transitions become unmeasurable from these states. The fault causing transitions from other states are measurable resulting in trivial fault diagnosability and hence, we have not considered them. The behavior of G

Table 6.2: *State variables of the model G.*

State	**State variables**	**State**	**State variables**
x_1	$S_V = 0, S_P = 0, NO, NP_1, NP_2$	x_6	$S_V = 1, S_P = 1, NO, NP_1, P_2$
x_3	$S_V = 1, S_P = 0, NO, P_1, NP_2$	x_8	$S_V = 1, S_P = 0, NO, NP_1, NP_2$
x_5	$S_V = 1, S_P = 1, O, P_1, P_2$	x_9	$S_V = 1, S_P = 1, O, P_1, P_2$
x_7	$S_V = 1, S_P = 0, NO, P_1, NP_2$	x_{10}	$S_V = 1, S_P = 0, O, P_1, P_2$
x_2	$S_V = 0, S_P = 0, NO, NP_1, NP_2$	x_{11}	$S_V = 0, S_P = 0, NO, NP_1, P_2$
x_4	$S_V = 1, S_P = 0, NO, NP_1, NP_2$	x_{12}	$S_V = 1, S_P = 0, O, P_1, P_2$

(shown in Figure 6.2) is explained as follows:

- The nominal behavior of G is represented through the set of states $\{x_1, x_3, x_5, x_7\}$ and transitions $\tau_1, \tau_2, \tau_3, \tau_4$. The initial state $X_0 = \{x_1\}$ defined by $S_V = 0, S_P = 0, S_{OS} = NO, S_{VS} = NP_1, S_{PS} = NP_2$, denotes the 'Closed' status of the throttle valve and 'OFF' status of the fuel pump. The pressure sensors PS_1 and PS_2 do not detect air or fuel flows in the system at this state and hence, $S_{VS} = NP_1$ and $S_{PS} = NP_2$. When the ECU issues the command to open the throttle valve ($S_V = 1$), G moves from state x_1 to state x_3 through the transition τ_1. Here, the sensor reading $S_{VS} = P_1$ shows the flow of air in the system due to the opening of the valve. G moves to state x_5 from state x_3 through the transition τ_2 when the ECU issues the command to start the fuel pump ($S_P = 1$). State x_5 with its state variables $S_V = 1, S_P = 1, S_{OS} = O, S_{VS} = P_1, S_{PS} = P_2$ represents the functioning of the engine due to the flow of air and fuel in the system. When the ECU issues the command to stop the fuel pump ($S_P = 0$), G moves from state x_5 to state x_7 through the transition τ_3. Here, the sensor PS_2 gives the reading $S_{PS} = NP_2$

which shows no fuel flow in the system. G moves back to its initial state x_1 through the transition τ_4 when the 'Closed' command is issued to the throttle valve from the ECU ($S_V = 0$).

- Due to the occurrence of the fault F_1, G moves from nominal to its faulty behavior represented through the set of states $\{x_2, x_4, x_6, x_8\}$ and the set of transitions $\{\tau_5, \tau_6, \tau_7, \tau_8\}$. G moves from the nominal state x_1 to the faulty state x_2 through the unmeasurable transition τ_{F_1}. All the states after τ_{F_1} have the label F_1, that is, $x_d(C) = \{F_1\}$, where $d \in \{2, 4, 6, 8\}$. All state variables in x_2 barring C assume the same set of values as that of x_1 and hence, these two states are considered as measurement equivalent (represented as $x_1 E x_2$; see Definition 2.4.1). From x_2, G moves to faulty state x_4 on transition τ_5 representing the situation when the ECU has issued the command to open the valve ($S_V = 1$) but sensor PS_1 does not detect any air pressure ($S_{VS} = NP_1$). G moves from x_4 to x_6 on τ_6 representing the situation when there is fuel flow ($S_P = 1, S_{PS} = P_2$) but no air flow ($S_{VS} = NP_1$) even though $S_V = 1$. Subsequently, when the ECU issues the command to stop the fuel pump, G moves from x_6 to faulty state x_8 on transition τ_7. At x_8, $S_V = 1, S_P = 0, S_{OS} = NO, S_{VS} = NP_1, S_{PS} = NP_2$. G moves back to state x_2 through the transition τ_8 when the ECU issues the command to close the valve.

- Similarly, as a result of the fault F_2, G moves from nominal to its faulty behavior represented by the states $x_9, x_{10}, x_{11}, x_{12}$ and the transitions $\tau_9, \tau_{10}, \tau_{11}, \tau_{12}$. G moves from the nominal state x_5 to its measurement equivalent faulty state x_9 ($x_5 E x_9$) through the unmeasurable transition τ_{F_2}. All the states after τ_{F_2} have the label F_2, that is, $x_d(C) = \{F_2\}$, where $d \in \{9, 10, 11, 12\}$. From x_9, G moves to faulty state x_{10} on transition τ_9 representing the situation when there is both air and fuel flow in the system ($S_{VS} = P_1, S_{PS} = P_2$) even though the desired status of the pump is 'OFF' ($S_P = 0$). G moves from x_{10} to x_{11} on τ_{10} to model the situation when fuel pressure is detected ($S_{PS} = P_2$) even though both air and fuel flow are desired to be stopped ($S_V = 0, S_P = 0$). Subsequently, when the ECU

issues the command to open the valve, G moves from x_{11} to faulty state x_{12} on transition τ_{11}. G moves back to state x_9 through the transition τ_{12} when the ECU issues the command to start the pump.

6.1.3 Fault Diagnosis of the EFI System

In this subsection, we briefly illustrate the diagnosis of the EFI system modeled as G shown in Figure 6.2. According to the diagnosis theory discussed in Section 2.4, a diagnoser G_{diag} (say, *global diagnoser*) is constructed from G which detects the presence of a fault of type F_i, (either F_1 or F_2) in the system. Figure 6.3 depicts the global diagnoser G_{diag} for the the EFI system G. The construction of G_{diag} is explained (see Section 2.4.3 for detailed steps) as follows.

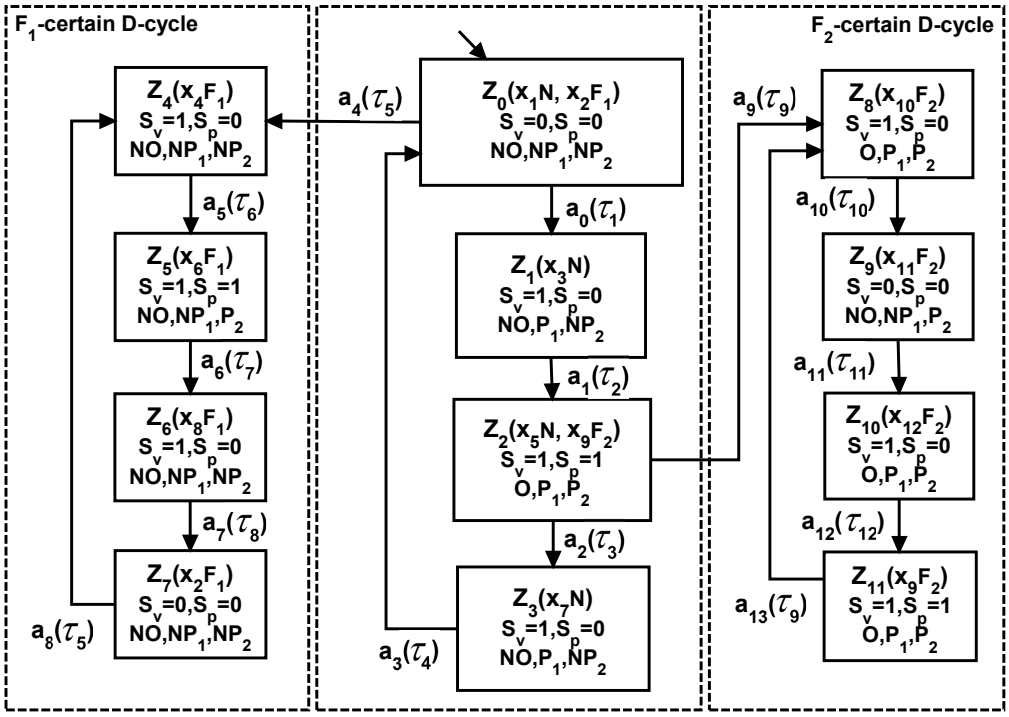


Figure 6.3: *The global diagnoser G_{diag} for the DES model G*

The initial D-state of the diagnoser Z_0 is obtained as $Z_0 = \mathcal{U}^*(X_0) = \{x_1, x_2\}$, where $X_0 = \{x_1\}$, the set of initial states of G and $\mathcal{U}^*(X_0)$ denotes the unmeasurable reach of X_0. It may be noted that Z_0 is F_1-uncertain as $X_1(C) = N$ while $X_2(C) =$

F_1. With $\Im_{mz_0} = \{\tau_1, \tau_5\}$ being the set of measurable G-transitions from the states $x_i \in Z_0$ the corresponding measurement equivalence classes of transitions become $A_{z_0} = \{\{\tau_1\}, \{\tau_5\}\}$. While $\{\tau_1\}$ is designated as D-transition a_0, $\{\tau_5\}$ becomes a_4. Now, the successor state Z_1 through transition a_0 is obtained as $Z_1 = \mathcal{U}^*(Z^+_{0_{a_0}}) = \{x_3\}$, where, $Z^+_{0_{a_0}} = \{final(\{\tau_1)\}\} = \{x_3\}$. Similarly Z_4, the successor state through transition a_4 is obtained as $Z_4 = \mathcal{U}^*(Z^+_{0_{a_4}}) = \{x_4\}$. In a similar way, the whole diagnoser G_{diag} is constructed. Therefore, G_{diag} has a set of states $Z = \{Z_0, Z_1, ..., Z_{11}\}$ and a set of transitions $A = \{a_0, a_1, ..., a_{13}\}$.

It can be observed that G_{diag} shown in Figure 6.3 contains three cycles, the first one being a cycle of F_1-certain D-states, the second being a cycle of F_2-certain D-states, while the third is a cycle over the normal and F_i-uncertain D-states Z_0, Z_1, Z_2 and Z_3. So, G_{diag} does not contain any cycle over F_i-uncertain states only, where $i \in \{1, 2\}$ and therefore, there is no F_i-indeterminate cycle in the diagnoser G_{diag} (see Section 2.4-Definition 2.4.12). Hence, the EFI system modeled as G is diagnosable with respect to the fault type F_i where $i \in \{1, 2\}$. The diagnosis of the EFI system by the diagnoser G_{diag} is described as follows.

G_{diag} monitors the behavior of the EFI system by measuring the values of the state variables. The state change transition a_4 from F_1-uncertain D-state Z_0 to F_1-certain D-state Z_4 in G_{diag} detects occurrence of the 'Stuck_Closed' failure of the throttle valve (fault type F_1). This is accomplished by observing the measurement variations in the state variables S_V and S_{VS}. Under normal operation, whenever the status of the throttle valve is 'Open' ($S_V = 1$) there should be a pressure in the intake manifold ($S_{VS} = P_1$). However, for D-state Z_4, $S_V = 1$ but $S_{VS} = NP_1$, thus establishing the fault F_1. Similarly, the transition a_9 from F_2-uncertain D-state Z_2 to F_2-certain D-state Z_8 identifies the presence of the 'Pump_On' failure of the fuel pump (fault type F_2). This is accomplished by observing the measurement variations in the state variables S_P and S_{PS} which should have the co-enumerations $\{S_P = 1, S_{PS} = P_2\}$ when operating normally. However, for D-state Z_8, $S_P = 0$ but $S_{PS} = P_2$, which establishes the fault F_2.

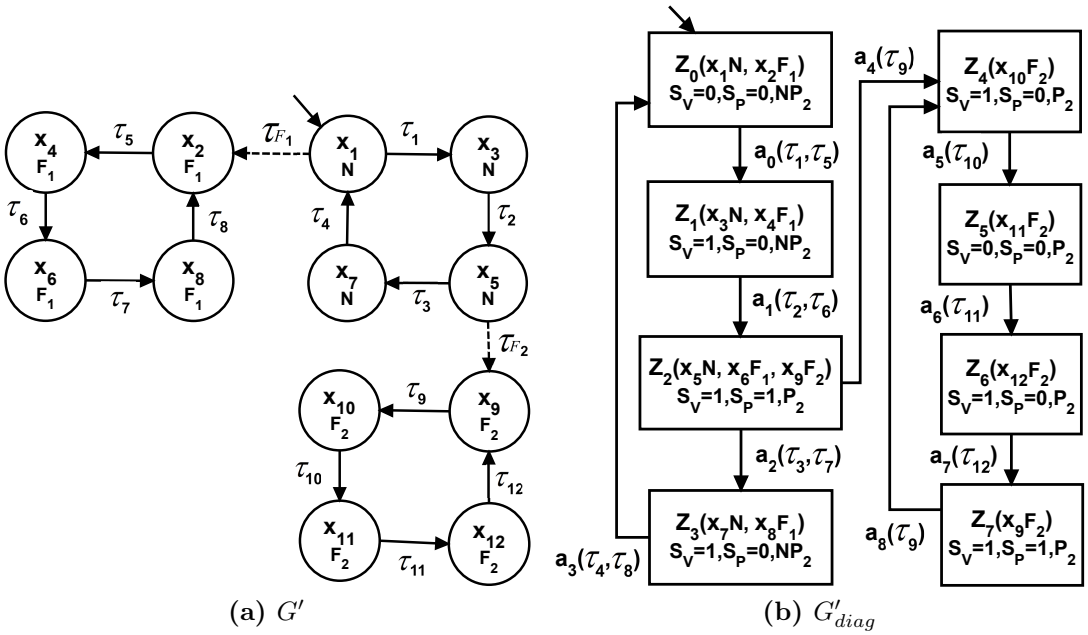
Remark: Since all variables considered in the design of G_{diag} are measurable, it has full knowledge of the system at any point in time and this helps G_{diag} to precisely identify any fault. Let us consider a modified model G' of the EFI system whose state variable measurements are listed in Table 6.3. Here, $S_m = \{S_V, S_P, S_{PS}\}$ and $S_u = \{S_{VS}, S_{OS}, C\}$. The system model G' and its corresponding diagnoser G'_{diag} are shown in Figure 6.4. It can be noticed that G'_{diag} contains an F_1-indeterminate cycle formed by F_1-uncertain D-states Z_0, Z_1, Z_2 and Z_3. Therefore, G' is F_2-diagnosable, but not F_1-diagnosable. This shows that measurement limitation on a subset of state variables compromises diagnosability of the system. □

Table 6.3: *State variables of the model G'.*

State	State variables	State	State variables
x_1	$S_V = 0, S_P = 0, NP_2$	x_6	$S_V = 1, S_P = 1, P_2$
x_3	$S_V = 1, S_P = 0, NP_2$	x_8	$S_V = 1, S_P = 0, NP_2$
x_5	$S_V = 1, S_P = 1, P_2$	x_9	$S_V = 1, S_P = 1, P_2$
x_7	$S_V = 1, S_P = 0, NP_2$	x_{10}	$S_V = 1, S_P = 0, P_2$
x_2	$S_V = 0, S_P = 0, NP_2$	x_{11}	$S_V = 0, S_P = 0, P_2$
x_4	$S_V = 1, S_P = 0, NP_2$	x_{12}	$S_V = 1, S_P = 0, P_2$

Global diagnosers typically consume spatially huge state spaces, their sizes being very sensitive to the number of model states. State space of the model in turn is exponential in the number of state variables. For n state variables, the number of model states is upper bounded by $O(2^n)$. The number of diagnoser states in turn can possibly be exponential with respect to the number of model states, thus making the state space complexity of such monolithic diagnosers to be $O(2^{2^n})$ in the number of state variables. From this observation, it can be concluded that a reduction in the number of measurable state variables may possibly lead to drastic reduction in the state space involved in diagnoser synthesis.

Typically for a practical system, the number of state variables ranges from 4 to 10 [28].



(a) G' **(b)** G'_{diag}

Figure 6.4: *DES model G' and its global diagnoser G'_{diag}.*

Let us consider a system modeled with five state variables whose corresponding diagnoser has a state space upper bounded by $O(2^{32})$ ($O(2^{2^5})$). If we are able to measurement limit even a single variable (say), the state space complexity of the diagnoser constructed from the resulting abstracted model reduces to $O(2^{16})$ ($O(2^{2^4})$). Therefore, measurement limitation of state variables in a system can potentially provide significant reductions in state space complexity of the model G as well as the diagnoser G_{diag}. This essential insight was employed by Zad et al. [115] where they applied measurement limitation on redundant state variables to achieve significant state space reductions involved in diagnoser synthesis with respect to conventional approaches. It may be noted that further measurement limitation beyond that proposed by Zad et al. cannot be achieved without compromising diagnosability (at least partially) of the system. However through a deeper look, we realized that full diagnosis with significantly lower overall state spaces may be obtained by deploying multiple intelligently constructed partially compromised diagnosers in parallel. For example, let us consider a system constituted of five variables $S_1, ..., S_5$ of which one variable (say, S_5) is redundant, and thus, its limitation do not

affect diagnosability of the system. So, Zad's approach would reduce the required state space from $O(2^{32})$ to $O(2^{16})$. Further, let us assume that the system may be affected by two fault types F_1 and F_2. Also, measurement limiting S_3 and S_4 compromises the diagnosability of F_1 and F_2, respectively. Thus, measurement limiting S_3 and S_5 (S_4 and S_5) will lead to the generation of a diagnoser say, G_{diag1} (G_{diag2}), which is F_2-diagnosable (F_1-diagnosable) only and consumes state space $O(2^8)$ ($O(2^{2^3})$). Further, it may be observed that the diagnosability of all faults is achieved by deploying G_{diag1} and G_{diag2} in parallel, resulting in a combined state space of $2 \cdot O(2^8) = O(2^9)$. Thus, by utilizing the additive nature of the rise in the state space complexity when multiple diagnosers are deployed together, it is possible to achieve complete diagnosability with far lower state space complexity compared to Zad's method. This insight has been the fundamental motivation towards the design approach *MLAD*, which is proposed in this work.

6.2 Proposed Fault Diagnosis Scheme

This section discusses our proposed fault diagnosis approach in detail.

6.2.1 Measurement Limitation based Abstract DES Diagnosis (MLAD)

In this section, we present a procedure to construct a set of partially compromised behaviorally abstracted diagnosers whose additive combination ensures diagnosability of all faults. Before proceeding further, we provide the following few definitions.

Definition 6.2.1 (**Model Abstraction**)**.** Model abstraction is defined as the mechanism for generating a reduced/abstracted model HG from a given model G by forcefully limiting the measurement of a subset of measurable state variables $S_C = \{S_j, S_{j+1}, ..., S_l\}$, where $S_C \subseteq S_m$ of G. So, the set of measurable and unmeasurable state variables in HG becomes $HS_m = S_m \setminus S_C$ and $HS_u = S_u \cup S_C$, respectively.

Model abstraction may or may not lead to compromised fault diagnosis. This brings us to the next important definition.

Definition 6.2.2 (F_C**-Model Abstraction**). Given a set of k fault types, F_C-Model Abstraction is defined as the mechanism for generating a reduced model HG_j from a given model G, which carefully chooses a designated subset of variables S_C whose limitation can possibly lead to compromised diagnosability of only a stipulated subset of (say k') faults while not affecting the diagnosability of the remaining $(k - k')$ faults.

It means that, a F_C-model abstraction leads to the compartmentalization of faults in the system and results in the non-diagnosability of a subset of faults $F_C = \{F_{C_1}, ..., F_{C_{k'}}\}$ in the generated reduced model HG_j.

Due to the measurement limitation of $S_C \subseteq S_m$ in G, its set of measurable and unmeasurable state variables are updated as $S_{mc} = S_m \setminus S_C$ and $S_{uc} = S_u \cup S_C$, respectively. In this situation, many states in G may become measurement equivalent which leads to the existence of a new set of unmeasurable transitions (say, $\Im_C$) in G. Therefore, the set of measurable transitions in G after limiting the measurement of S_C becomes $\Im_{mc} = \Im_m \setminus \Im_C$.

Hence, a F_C-model abstraction can be viewed in terms of a projection operation $P : \Im^* \rightarrow \Im^*_{mc}$, referred to as *controlled projection*, defined over two transition sets $\Im$ and $\Im_{mc}$, where $\Im$ denotes the transition set of original model G and $\Im_{mc}$ denotes the set of measurable transitions after the measurement limitation of S_C.

Let $P_1, ..., P_L$ denote the controlled projection operations, denoted as $P_j : \Im^* \rightarrow \Im^*_{mc_j}$, where $1 \leq j \leq L$ which possibly lead to L controlled measurement limitations and each such projection operation P_j results in a F_C-model abstracted subsystem model HG_j.

Definition 6.2.3. F_i**-compartmentalized Diagnosability**: A DES model G is said to be F_i-compartmentalized diagnosable with respect to a set of projection operations $P_1, ..., P_L$ for fault F_i, if the following holds:

$$(\exists n \in \mathbb{N})[\forall q \in \Psi(X_{F_i})](\forall r \in L_f(G)/q)[|r| \geq n \Rightarrow D] \tag{6.1}$$

where the condition D is $(\forall u \in P_1^{-1}[P_1(qr)], final(u) \in X_{F_i}) \vee ... \vee (\forall u \in P_L^{-1}[P_L(qr)], final(u) \in X_{F_i})$.

Informally, Definition 6.2.3 means the following: let q be any finite prefix of a trace of

G that ends in an F_i-state and let r be any sufficiently long continuation of q. Condition D then requires that every sequence of transitions, measurement equivalent with qr and belonging to $P_j^{-1}[P_j(qr)]$, where $\exists j \in \{1, ..., L\}$, shall end into an F_i-state. This implies that, along every continuation r of q, there exists at least one F_C-model abstracted subsystem HG_j which can detect the occurrence of fault F_i within a finite delay, more specifically in at most n transitions after q.

Definition 6.2.4 ($REACH(x_i)$)**.** $REACH(x_i)$ is defined as the set of all states reachable from state x_i in G including x_i. So, $REACH(x_i) = \{x_i\} \cup \{x_j | \exists \tau_{ij} = \langle x_i, x_j \rangle : x_i, x_j \in X\}$.

The model abstraction mechanism fundamentally involves the following steps: i) Partitioning G into subcomponents based on nominal and faulty behaviors, ii) Determining the connected components in each such subcomponent, iii) Reducing each connected component obtained in the previous step, by merging states which have become measurement equivalent due to forceful limitation of the set of variables S_C in G (refer Definition 6.2.1), and adjusting corresponding transitions.

Now, we formally present the methodology to construct an abstracted model $HG = \langle HX, HS, H\Im, HX_0 \rangle$ from $G = \langle X, S, \Im, X_0 \rangle$, in Algorithm 10. Line 1 of the algorithm initializes the tuple variables $HX, H\Im$ and HX_0 of HG. Line 2 determines the set of measurable and unmeasurable state variables in HG and line 3 removes fault causing transitions τ_{F_i} in G, from $H\Im$. It may be noted that the removal of fault causing transitions of a particular type τ_{F_i}, transforms HG into a forest of connected components. One of these components represents the nominal and faulty behaviors, excluding those behaviours induced by F_i. The remaining connected components represent distinct faulty behaviors in different subsystems induced by failure type F_i. Let HX_h denote either, the set of states in the nominal behavior or the states in the forest of faulty behaviors corresponding to a distinct fault type. Line 4 partitions HX into disjoint subsets such that $HX = \bigcup_{h=N,F_1,...,F_p} HX_h$. The *for* loop in lines 5-27 merges mutually measurement equivalent states in each subset HX_h and constructs the reduced automata corresponding to the behaviors involving states in HX_h. Line 6 determines all the equivalence classes of

ALGORITHM 10: Synthesis of an abstracted model, HG

Input: $G = \langle X, S, \Im, X_0 \rangle$, S_C
Output: $HG = \langle HX, HS, H\Im, HX_0 \rangle$

1 $HX \leftarrow X$, $H\Im \leftarrow \Im$, $HX_0 \leftarrow \phi$;
2 $HS_m = S_m \setminus S_C$, $HS_u = S_u \cup S_C$, $HS = HS_m \cup HS_u$;
3 $H\Im = H\Im \setminus (\tau_{F_i} = \langle u, v \rangle)$, where $u \in X_N$, $v \in X_{F_i}$, $X_N, X_{F_i} \in X$ and $i \in \{1, 2, ..., k\}$, k denotes the number of failure types in G;
4 Partition state set HX into disjoint subsets such that $HX = HX_N \cup HX_{F_1} \cup ... \cup HX_{F_k}$;
5 **for** *each* $HX_h \subseteq HX$, *where* $h = N, F_1, ..., F_k$ **do**
6 Determine the equivalence classes ($HX_{h_j}, 1 \leq j \leq l$) of measurement equivalent states in HX_h such that $HX_h = HX_{h_1} \cup HX_{h_2} \cup ... \cup HX_{h_l}$;
7 **for** *each* $HX_{h_j} \subseteq HX_h$ *where* $j = 1, 2, ..., l$ **do**
8 **while** $HX_{h_j} \neq \phi$ **do**
9 Select any state $x \in HX_{h_j}$;
10 $X_R = REACH(x)$;
11 Merge states in X_R to obtain composite state x_r;
12 $HX_{h_j} = HX_{h_j} \setminus X_R$;
13 $HX_h = HX_h \cup x_r \setminus X_R$;
14 $H\Im = H\Im \setminus \{\tau_{ij} | \exists \tau_{ij} = \langle x_i, x_j \rangle : x_i, x_j \in X_R\}$;
15 **for** *each* $x'_j \in HX_h \setminus x_r$ **do**
16 $\Im_a = \{\tau_{ij} | \exists \tau_{ij} = \langle x_i, x'_j \rangle : x_i \in X_R\}$;
17 $\Im_b = \{\tau_{ji} | \exists \tau_{ji} = \langle x'_j, x_i \rangle : x_i \in X_R\}$;
18 **if** $\Im_a \neq \phi$ **then**
19 $H\Im = H\Im \setminus \Im_a \cup \langle x_r, x'_j \rangle$;
20 **if** $\Im_b \neq \phi$ **then**
21 $H\Im = H\Im \setminus \Im_b \cup \langle x'_j, x_r \rangle$;
22 Add faulty transitions $\tau_{F_i} = \langle x, y \rangle$, where $x \in HX_N$ and $y \in HX_{F_i}$ such that x is generated by collapsing u with zero or more other state(s) of G and y is generated by collapsing v with zero or more other state(s) of G and $\exists \tau_{F_i} = \langle u, v \rangle$ in G: $H\Im = H\Im \cup \tau_{F_i}$;
23 $HX_0 = \{x | x \in HX_N$ and x is generated by collapsing a G-state $x_0 \in X_0\}$;
24 $HX = HX_N \cup HX_{F_1} \cup ... \cup HX_{F_k}$;

measurement equivalent states in HX_h. Corresponding to each equivalence class HX_{h_j} considered within the next *for* loop (lines 7-26), the *while* loop in lines 8-25 merges each

mutually reachable set of states HX_{h_j} and updates corresponding transitions to realize a reduction step. Merging of a particular set of mutually reachable states occurs in lines 9 to 11. Lines 12 and 13 update HX_{h_j} and HX_h, while lines 14-24 update the transitions in $H\Im$, to account for the merge of states in X_R. Line 28 of the algorithm adds fault causing transitions τ_{F_i} back into HG. Line 29 determines the initial state HX_0 and finally, line 30 updates the final set of states HX in HG.

Now, we illustrate the construction of the model abstractions for the EFI system discussed in Section 6.1, through the steps of Algorithm 10. We discuss the generation of two reduced models HG_1 and HG_2 obtained by measurement limiting variable sets $\{S_V, S_{VS}, S_{OS}\}$ and $\{S_P, S_{PS}, S_{OS}\}$ (refer Table 6.1 for a description of the variables) respectively, in model G (see Figure 6.2). As a result of this measurement limitation, HG_1 becomes F_2-compartmentalized diagnosable (Pump_On failure of the fuel pump) but compromises F_1 (Stuck_Closed failure of throttle valve), whereas HG_2 becomes F_1-compartmentalized diagnosable only. The procedure for constructing $HG_1 = \langle HX_1, HS_1, H\Im_1, HX_{10}\rangle$ from G is described as follows:

The tuple attributes $HX_1, H\Im_1$ and HX_{10} of HG_1 are initialized as in Algorithm 10. HS_1 is determined as $HS_1 = HS_{m1} \cup HS_{u1}$, where $HS_{m1} = \{S_P, S_{PS}\}$ and $HS_{u1} = \{S_V, S_{VS}, S_{OS}, C\}$. Now, we remove fault causing transitions τ_{F_1} and τ_{F_2} from HG and partition HX_1 into $HX_1 = HX_N \cup HX_{F_1} \cup HX_{F_2}$. By only measuring the values of variables S_P and S_{PS}, two equivalence classes of measurement equivalent states $\{x_1, x_3, x_7\}$ and $\{x_5\}$ are obtained in HX_N. Similarly, HX_{F_1} and HX_{F_1} gets divided into equivalence classes $\{\{x_2, x_4, x_8\}, \{x_6\}\}$ and $\{\{x_{10}, x_{11}, x_{12}\}, \{x_9\}\}$, respectively. Application of the reach operation ($REACH()$) on the equivalence classes $\{x_1, x_3, x_7\}$, $\{x_2, x_4, x_8\}$ and $\{x_{10}, x_{11}, x_{12}\}$ produces corresponding composite states $x_1x_3x_7$, $x_2x_4x_8$ and $x_{10}x_{11}x_{12}$, because all states within a given equivalence class are mutually reachable from each other. After this, the transitions to and from these composite states to the other states are updated to obtain the reduced nominal and faulty behaviors. The final reduced model HG_1 as shown in Figure 6.5(a) is derived by adding the fault causing transitions τ_{F_1} and τ_{F_2} back into the model and assigning the initial state $HX_{10} = \{x_1x_3x_7\}$.

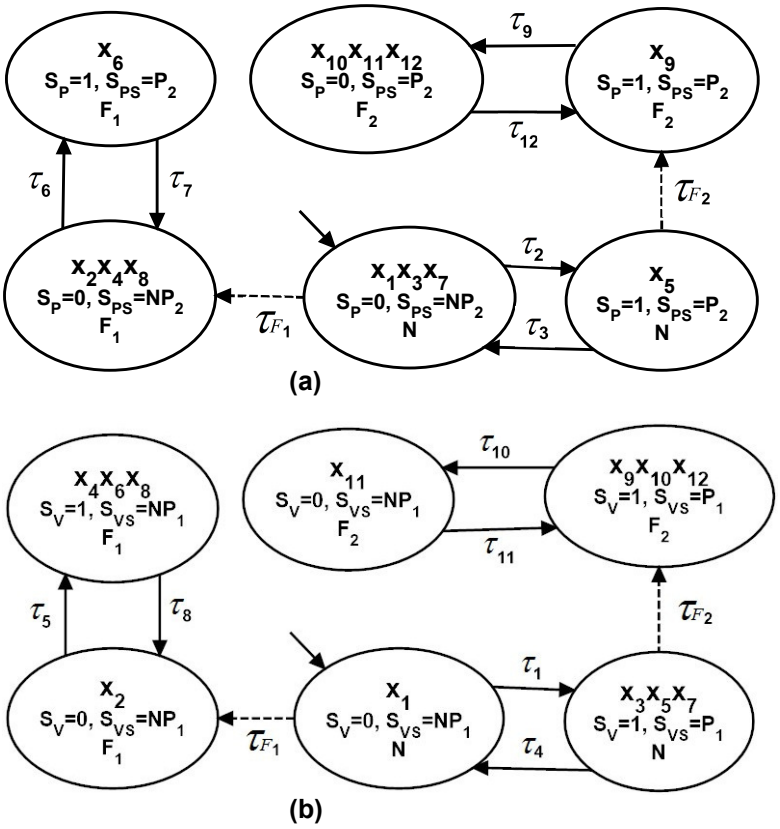


Figure 6.5: *(a) & (b) The abstracted models HG_1 and HG_2, respectively of G.*

Likewise, the reduced model HG_2 shown in Figure 6.5(b) is obtained by only allowing measurement of state variables S_V and S_{VS}. The model abstraction mechanism is similar to the one described for HG_1 above.

It may be noted that HG_1 and HG_2 are obtained through F_C-model abstractions where $F_C = \{F_1\}$ for HG_1 and $F_C = \{F_2\}$ for HG_2. While HG_1 which compromises the 'Stuck_Closed' failure of the throttle valve is obtained by measurement limiting variables related to *Air Flow Subsystem*, HG_2 limits variables related to *Fuel Flow Subsystem* and makes the 'Pump_On' failure of the fuel pump non-diagnosable. Thus, as an empirical inference, it may be concluded that measurement limitation of a set of carefully chosen variables related to a particular subset of faulty behaviors F_C leads to F_C-model abstractions producing reduced models that compromises F_C. Conversely, this highlights that the model abstraction mechanism discussed here is based on the

compartmentalization of faults, rather than modularization based on functionality.

6.2.1.1 Fault Diagnosis under Behavioral Abstraction

In this subsection, we describe the fault diagnosis mechanism proposed in *MLAD* by constructing two diagnosers G_{diag1} and G_{diag2} (see Figure 6.6) corresponding to the abstracted models $HG_1$and HG_2 of the EFI system.

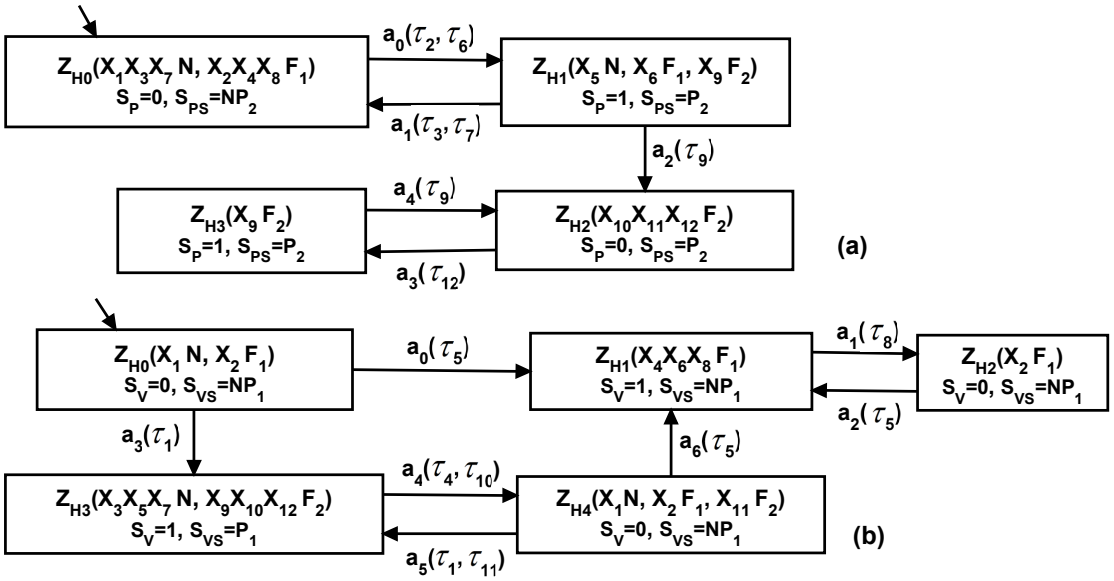


Figure 6.6: *The MLAD approach: (a) G_{diag1}, (b) G_{diag2}.*

G_{diag1} shown in Figure 6.6(a) monitors the behavior of the EFI system by measuring the state variables S_P and S_{PS} only. In a nominally functioning system or subsequent to the occurrence of the 'Stuck_Closed' failure of the throttle valve (fault type F_1), G_{diag1} moves from D-state Z_{H0} to Z_{H1} (on transition τ_2 under normal operation or τ_6 after fault F_1) or vice versa (on τ_3 under normal operation or τ_7 subsequent to F_1). Thus, transition pairs τ_2, τ_6 and also τ_3, τ_7 are measurement equivalent and represented through D-transitions a_0 and a_1, respectively as shown in the figure. From the above statements, it also follows that both Z_{H0} and Z_{H1} are F_1-uncertain and so, the cycle involving these two states with corresponding transitions a_0, a_1, becomes F_1-uncertain. Validation with G shows that this F_1-uncertain cycle is actually F_1-indeterminate (refer Definition 2.4.12). This is because corresponding to this D-cycle, there exists two measurement equivalent

cycles q and y in HG_1, among which q ($= \langle \tau_2, \tau_3 \rangle$) comprises only of normal G-states and y ($= \langle \tau_6, \tau_7 \rangle$) comprises of F_1 G-states only (see Figure 6.5(a)). Therefore, the diagnoser G_{diag1} cannot detect fault type F_1. G_{diag1} however, can precisely diagnose the 'Pump_On' failure of the fuel pump by detecting fuel pressure in the fuel rail ($S_{PS} = P_2$) even when the ECU instructs otherwise (by changing the value of S_P from 1 to 0). In this situation, G_{diag1} transits from D-state Z_{H1} to F_2-certain state Z_{H2} on transition a_2 and thereby, establishes F_2. It may be noted that there are no F_2-indeterminate cycles in G_{diag1}, and hence, the abstracted model HG_1 is F_2-compartmentalized diagnosable.

Similarly, G_{diag2} (see Figure 6.6(b)) derived from the abstracted model HG_2 monitors the behavior of the EFI system by measuring the state variables S_V and S_{VS}. Here, we observe that contrary to HG_1, HG_2 is not F_2-compartmentalized diagnosable due to the presence of the F_2-indeterminate cycle over F_2-uncertain states Z_{H3} and Z_{H4} in G_{diag2}. However, fault type F_1 is diagnosed by detecting the absence of air pressure in the intake manifold ($S_{VS} = NP_1$) even when the throttle valve is opened ($S_V = 1$). In this situation, G_{diag2} moves to the F_1-certain D-state Z_{H1} from D-states Z_{H0} or Z_{H4}, thus establishing F_1. Further, it may be noted that G_{diag2} does not contain any F_1-indeterminate cycle in it, and hence HG_2 is F_1-compartmentalized diagnosable.

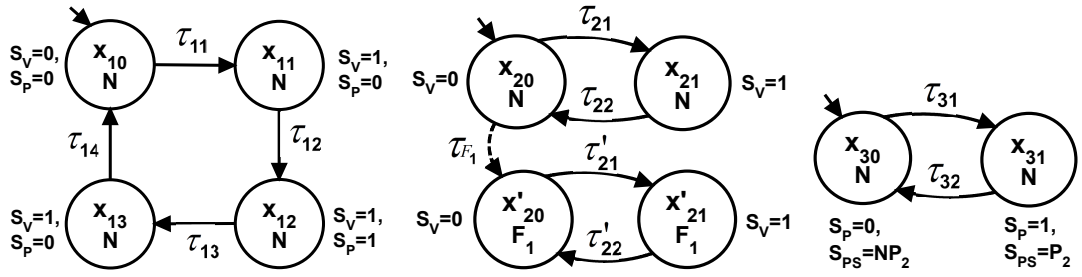
From the above discussion, we see that although either of G_{diag1} or G_{diag2} cannot ensure the diagnosability of all faults in seclusion, this limitation can be eradicated by concurrently deploying both diagnosers in parallel. The principal advantage of this approach is the reduction in state space that is achieved with respect to the single monolithic diagnoser that is obtained directly from the original model G. Figures 6.3 and 6.6 show that while the diagnoser G_{diag} obtained from model G of the EFI system contains 12 states, the combined state space of G_{diag1} and G_{diag2} contains only 9 states. In general, it may be concluded that as diagnoser sizes are doubly exponential with respect to the number of state variables in the system, the diagnosis methodology based on, the additive combination of partially compromised diagnosers presented here, is expected to provide handsome reductions in state space for larger, more complex systems.

Example: We now discuss an example to show that, the state space of a modular

diagnoser (synthesized using Debouk's approach [36]) may become as big as, or even larger than a monolithic diagnoser (built using Sampath's approach [103]), in case, one or more faults associated with a particular subcomponent of the system cannot be detected by the local subsystem diagnoser. However, it may still be possible that such a system can be compartmentalized based on faults, allowing the generation of diagnosers with far lower state spaces when synthesized using MLAD. The example uses a slightly modified model of the EFI system presented in Section 6.1. Specifically, we assume that: i) The air-pressure sensor which measures the flow of air in the *Air Flow Subsystem*, is now physically unavailable. ii) The system can suffer only one failure, "Stuck_Closed" failure of the throttle valve in the air flow subsystem (denoted as F_1). All other components are fault-free.

Let G_1, G_2 and G_3 represent DES models corresponding to *Air-Fuel Ratio Control Subsystem*, *Air Flow Subsystem* and *Fuel Flow Subsystem*, respectively (see Figure 6.7).

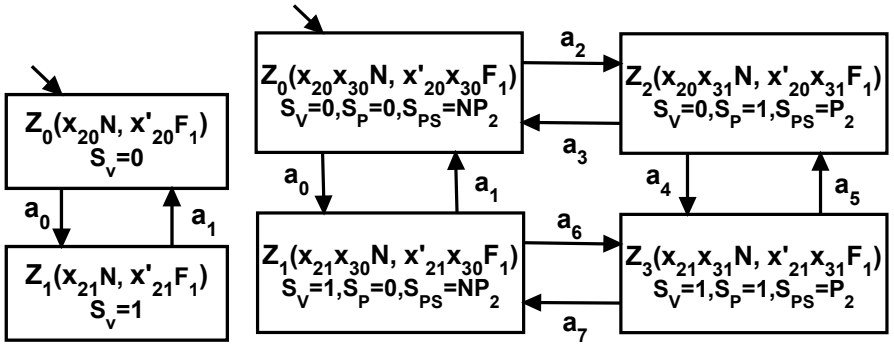
It may be seen from Figure 6.8(a) (depicting the local diagnoser G_{diag2}, for subsys-



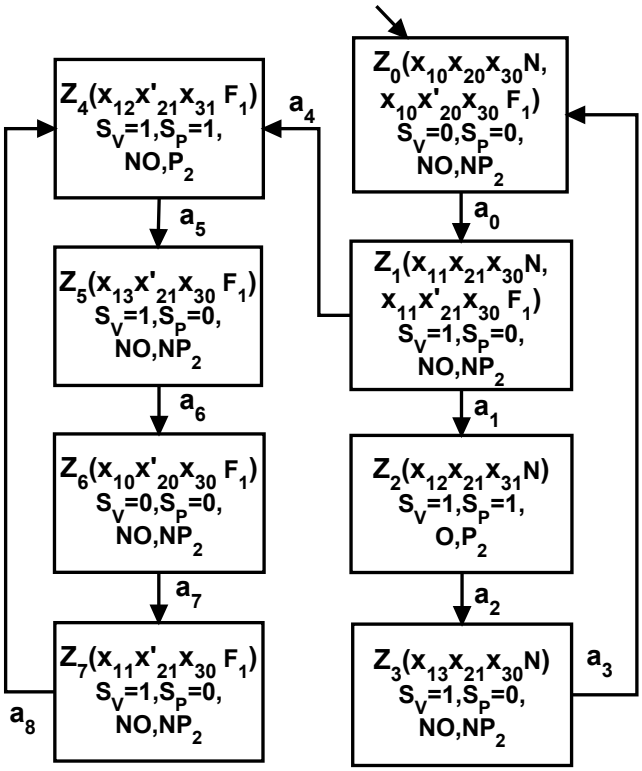
(a) *Air-fuel ratio control subsystem model* G_1 **(b)** *Air flow subsystem model* G_2 **(c)** *Fuel flow subsystem model* G_3

Figure 6.7: *Component models of the EFI modular system.*

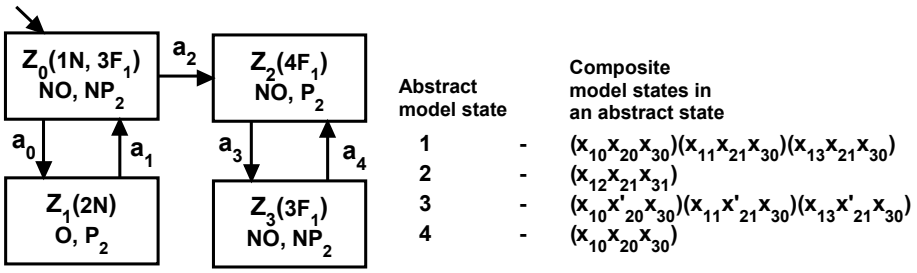
tem model G_2) that G_{diag2} contains an F_1-indeterminate cycle. So, fault F_1 cannot be detected by G_{diag2}. In this situation, Debouk's approach composes the subsystem models G_2 and G_3 producing $G_{23} = G_2 \parallel G_3$ and builds diagnoser G_{diag23} as shown in Figure 6.8(b). However as the figure reveals, G_{diag23} is also unable to detect the fault F_1 as it contains an F_1-indeterminate cycle. Thus now, Debouk's approach is forced to build a diagnoser G_{diag} (Figure 6.8(c)) corresponding to the overall system model



(a) *Local modular diagnoser* G_{diag2} (b) *Intermediate modular diagnoser* G_{diag23}



(c) *Composed modular diagnoser* G_{diag}



(d) *MLAD diagnoser*

Figure 6.8: *Fault diagnosis of the EFI modular system.*

Table 6.4: *A qualitative comparison among related works*

Diagnosis Framework	**System Structure**	**Modeling Framework**	**Projection/ Measurement Limitation**	**Diagnosis**	**Diagnoser Design Strategy**
Sampath et al. [103]	Monolithic	Event-based	Physical restriction	Monolithic diagnoser	Top-down
Zad et al. [115]	Monolithic	State-based	Limits redundant variables	Monolithic diagnoser	Top-down
Modular Approaches (Debouk et al. [36])	Modular	Event-based	Projections based on local observations	local diagnosers	Bottom-up
Our MLAD	Monolithic, Modular	State-based	Controlled projections w.r.t. fault compartmentalization	local diagnosers	Top-down

G $(= G_1 \parallel G_{23})$ and therefore, incurs the same state space complexity as Sampath's approach. It may be seen from the figure that G_{diag} contains 8 states and is free from F_1-indeterminate cycles.

On the other hand, the diagnoser shown in Figure 6.8(d), build from a fault compartmentalization oriented reduced model using the MLAD approach, contains only 4 states, and therefore, is able to achieve significant state space reduction compared to Debouk/Sampath's approach. Here, the compartmentalization of fault F_1 is done by applying a projection on the composite model of the system in a controlled manner which measures only the readings of the oxygen sensor and the fuel-pressure sensor. □

Since the composite DES model has both nominal and faulty behaviors, our approach is naturally applicable to modular systems where faults are always contained in a subset of components. Even if the overall system is non-modular or monolithic, we can do a fault compartmentalization using F_C-model abstraction. Therefore, our fault diagnosis approach is applicable to both monolithic as well as modular systems. A detailed comparative study of our proposed scheme with the state-of-the-art is given in Table 6.4.

6.3 Experimental Evaluation

In this section, we briefly discuss the experimental evaluation of our proposed *MLAD* approach on different standard practical benchmark systems and show its effectiveness over the conventional monolithic diagnosis mechanisms proposed by Sampath et al. [104] and Zad et al. [115]. For this purpose, we have considered nominal and faulty behaviors corresponding to two benchmark systems presented in [104]: i) the Air Handling Unit (AHU) of a HVAC (Heating, Ventilation and Air Conditioning) system and ii) the Nitric Acid Cooling (NAC) system.

6.3.1 System I

AHU is one of the principal components of a HVAC system. When there is a demand for heating (termed as heating load), the essential job of an AHU under normal operation is as follows: AHU accepts cold air at a given temperature, heats it to the stipulated degree inside its heating coil and drives out appropriate amount of heated air through a valve into a set of rooms, such that a desired comfort level is maintained. The degree of heating within the heating coil is regulated by a controller which commands a pump to drive in appropriate amounts of heated water or stream from a boiler, as required. Similarly, the operation of the valve is also regulated by the controller. This nominal system behavior gets affected due to the occurrence of the following failures: i) *Valve Stuck-Open (fault type F_1):* Valve remains 'Open' even when the controller signals it to be closed, ii) *Valve Stuck-Closed (fault type F_2):* Valve remains 'Closed' even when the controller signals it to be opened, iii) *Controller Failed-On (fault type F_3):* Controller always assumes a positive load and instructs the system to run regardless of whether a load is actually present, iv) *Controller Failed-Off (fault type F_4):* Controller does not detect the presence of load on the system and so, fails to issue appropriate control signals. Simultaneous occurrence of more than one fault is not considered here. For the purpose of diagnosability, we assume that the system is always in power-on state. Otherwise, it is obvious that if the system is not activated, it is not possible to diagnose occurrences of the above mentioned failures.

Table 6.5 summarizes the diagnosability of the HVAC system discussed above. From

Table 6.5: *The effectiveness of MLAD: HVAC System.*

	Diagnosability					
Diagnoser	F_1	F_2	F_3	F_4	**Number of States**	**Measurable Variables**
Global Diagnoser G_{diag} (Sampath et al. [104])	✓	✓	✓	✓	45	All (refer, Section IV-A in [104]).
Reduced Diagnoser G_{diag1} (Zad et al. [115])	✓	✓	✓	✓	27	All except status of the fan, pump and boiler.
Abstract Diagnoser G_{diag2}	✓	✓	χ	χ	9	Status of the valve and valve flow sensor reading.
Abstract Diagnoser G_{diag3}	χ	χ	✓	✓	14	Status of the valve and the load

Table 6.5, it can be observed that the global diagnoser G_{diag} corresponding to the HVAC system model G (constructed based on Sampath et al. [104]) ensures the diagnosability of all fault types. G_{diag} contains 45 states. The global model G of the HVAC system consists of 6 measurable state variables: the valve flow sensor readings (S_{FS}), status of valve (S_V), status of fan (S_F), status of pump (S_P), status of boiler (S_B) and status of load (S_L). Therefore, the set of measurable state variables (say, S_m) in G is denoted as $S_m = \{S_{FS}, S_V, S_F, S_P, S_B, S_L\}$. Each state variable except S_L takes values from boolean domain whereas S_L takes values from domain $= \{0, 1, 2\}$. Table 6.6 depicts the global model G of the HVAC system under consideration. The attributes of this state-transition table are explained as follows: $\tau = \langle x, x^+ \rangle$ denotes a transition from a state x to another state x^+. S_x and S_{x^+} denote the set of values corresponding to measurable state variables in states x and x^+, respectively. For example, the transition τ_2 in G (see Table 6.6) is defined from state 4 having $S_4 = \{S_{FS} = 0, S_V = 0, S_F = 1, S_P = 0, S_B = 0, S_L = 0\}$ to another state 7 with $S_7 = \{S_{FS} = 0, S_V = 0, S_F = 1, S_P = 0, S_B = 0, S_L = 2\}$. Out of these six state variables, S_F, S_P and S_B are redundant with respect to the diagnosability of faults F_1, F_2, F_3 and F_4. By limiting these redundant variables following the mechanism by Zad et al. [115], we obtain the reduced diagnoser G_{diag1}

Table 6.6: *State-Transition Table: Global Model G*

x	S_x	τ	x^+	S_{x^+}	x	S_x	τ	x^+	S_{x^+}	x	S_x	τ	x^+	S_{x^+}
1	0,0,0,0,0,0	τ_1	4	0,0,1,0,0,0	4	0,0,1,0,0,0	τ_2	7	0,0,1,0,0,2	7	0,0,1,0,0,2	τ_4	10	0,1,1,0,0,2
		τ_{F_1}	2	0,0,0,0,0,0			τ_3	28	0,0,1,0,0,1			τ_{F_1}	8	0,0,1,0,0,2
		τ_{F_2}	3	0,0,0,0,0,0			τ_{F_1}	5	0,0,1,0,0,0			τ_{F_2}	9	0,0,1,0,0,2
		τ_{F_3}	31	0,0,0,0,0,0			τ_{F_2}	6	0,0,1,0,0,0	16	1,1,1,1,1,2	τ_7	19	1,1,1,1,1,1
		τ_{F_4}	79	0,0,0,0,0,0	13	1,1,1,1,0,2	τ_6	16	1,1,1,1,1,2			τ_{F_1}	17	1,1,1,1,1,2
10	0,1,1,0,0,2	τ_5	13	1,1,1,1,0,2			τ_{F_1}	14	1,1,1,1,0,2			τ_{F_2}	92	1,1,1,1,1,2
		τ_{F_1}	11	0,1,1,0,0,2			τ_{F_2}	91	1,1,1,1,0,2	25	0,0,1,0,1,1	τ_{10}	28	0,0,1,0,0,1
		τ_{F_2}	12	0,1,1,0,0,2	22	0,0,1,1,1,1	τ_9	25	0,0,1,0,1,1			τ_{F_1}	26	0,0,1,0,1,1
19	1,1,1,1,1,1	τ_8	22	0,0,1,1,1,1			τ_{F_1}	94	0,0,1,1,1,1			τ_{F_2}	27	0,0,1,0,1,1
		τ_{F_1}	20	1,1,1,1,1,1			τ_{F_2}	24	0,0,1,1,1,1	11	0,1,1,0,0,2	τ_{16}	14	1,1,1,1,0,2
		τ_{F_2}	93	1,1,1,1,1,1	2	0,0,0,0,0,0	τ_{12}	5	0,0,1,0,0,0	14	1,1,1,1,0,2	τ_{17}	17	1,1,1,1,1,2
28	0,0,1,0,0,1	τ_{11}	7	0,0,1,0,0,2	5	0,0,1,0,0,0	τ_{13}	8	0,0,1,0,0,2	17	1,1,1,1,1,2	τ_{18}	20	1,1,1,1,1,1
		τ_{F_1}	29	0,0,1,0,0,1			τ_{14}	29	0,0,1,0,0,1	20	1,1,1,1,1,1	τ_{19}	23	1,0,1,1,1,1
		τ_{F_2}	30	0,0,1,0,0,1	8	0,0,1,0,0,2	τ_{15}	11	0,1,1,0,0,2	23	1,0,1,1,1,1	τ_{20}	26	0,0,1,0,1,1
26	0,0,1,0,1,1	τ_{21}	29	0,0,1,0,0,1	29	0,0,1,0,0,1	τ_{22}	8	0,0,1,0,0,2	3	0,0,0,0,0,0	τ_{23}	6	0,0,1,0,0,0
6	0,0,1,0,0,0	τ_{24}	9	0,0,1,0,0,2	9	0,0,1,0,0,2	τ_{26}	12	0,1,1,0,0,2	12	0,1,1,0,0,2	τ_{27}	15	0,1,1,1,0,2
		τ_{25}	30	0,0,1,0,0,1	15	0,1,1,1,0,2	τ_{28}	18	0,1,1,1,1,2	18	0,1,1,1,1,2	τ_{29}	21	0,1,1,1,1,1
21	0,1,1,1,1,1	τ_{30}	24	0,0,1,1,1,1	24	0,0,1,1,1,1	τ_{31}	27	0,0,1,0,1,1	27	0,0,1,0,1,1	τ_{32}	30	0,0,1,0,0,1
30	0,0,1,0,0,1	τ_{33}	9	0,0,1,0,0,2	31	0,0,0,0,0,0	τ_{34}	34	0,0,1,0,0,0	34	0,0,1,0,0,0	τ_{35}	37	0,0,1,0,0,2
37	0,0,1,0,0,2	τ_{37}	40	0,1,1,0,0,2	49	0,0,1,0,0,1	τ_{38}	52	0,1,1,0,0,1			τ_{36}	49	0,0,1,0,0,1
40	0,1,1,0,0,2	τ_{39}	43	1,1,1,1,0,2	52	0,1,1,0,0,1	τ_{40}	55	1,1,1,1,0,1	46	1,1,1,1,1,2	τ_{43}	58	1,1,1,1,1,1
43	1,1,1,1,0,2	τ_{41}	46	1,1,1,1,1,2	55	1,1,1,1,0,1	τ_{42}	67	1,1,1,1,1,1	67	1,1,1,1,1,1	τ_{44}	70	1,1,1,1,1,2
58	1,1,1,1,1,1	τ_{45}	61	1,1,1,1,1,1	70	1,1,1,1,1,2	τ_{46}	73	1,1,1,1,1,2	61	1,1,1,1,1,1	τ_{47}	64	1,1,1,1,1,1
73	1,1,1,1,1,2	τ_{48}	76	1,1,1,1,1,2	64	1,1,1,1,1,1	τ_{49}	67	1,1,1,1,1,1	76	1,1,1,1,1,2	τ_{50}	46	1,1,1,1,1,2
79	0,0,0,0,0,0	τ_{51}	82	0,0,1,0,0,0	82	0,0,1,0,0,0	τ_{52}	85	0,0,1,0,0,2	85	0,0,1,0,0,2	τ_{54}	88	0,0,1,0,0,1
							τ_{53}	88	0,0,1,0,0,1	88	0,0,1,0,0,1	τ_{55}	85	0,0,1,0,0,2

having 27 states (constructed from the reduced model G_1) and with the same power as G_{diag} in terms of diagnosability. Table 6.7 depicts the reduced model G_1. Since

Table 6.7: *State-Transition Table: Reduced Model* G_1

x	S_x	τ	x^+	S_{x^+}	x	S_x	τ	x^+	S_{x^+}	x	S_x	τ	x^+	S_{x^+}
x_1	0,0,-,-,-,0	τ_1	x_2	0,0,-,-,-,2	x_2	0,0,-,-,-,2	τ_3	x_4	0,1,-,-,-,2	x_3	0,0,-,-,-,1	τ_7	x_2	0,0,-,-,-,2
		τ_2	x_3	0,0,-,-,-,1			τ_{F_1}	x_8	0,0,-,-,-,2			τ_{F_1}	x_9	0,0,-,-,-,1
		τ_{F_1}	x_7	0,0,-,-,-,0			τ_{F_2}	x_{15}	0,0,-,-,-,2			τ_{F_2}	x_{16}	0,0,-,-,-,1
		τ_{F_2}	x_{14}	0,0,-,-,-,0	x_4	0,1,-,-,-,2	τ_4	x_5	1,1,-,-,-,2	x_5	1,1,-,-,-,2	τ_5	x_6	1,1,-,-,-,1
		τ_{F_3}	x_{19}	0,0,-,-,-,0			τ_{F_1}	x_{10}	0,1,-,-,-,2			τ_{F_1}	x_{12}	1,1,-,-,-,2
		τ_{F_4}	x_{26}	0,0,-,-,-,0			τ_{F_2}	x_{17}	0,1,-,-,-,2			τ_{F_2}	x_{29}	1,1,-,-,-,2
x_6	1,1,-,-,-,1	τ_6	x_3	0,0,-,-,-,1	x_7	0,0,-,-,-,0	τ_8	x_8	0,0,-,-,-,2	x_{10}	0,1,-,-,-,2	τ_{11}	x_{12}	1,1,-,-,-,2
		τ_{F_1}	x_{13}	1,1,-,-,-,1			τ_9	x_9	0,0,-,-,-,1	x_{12}	1,1,-,-,-,2	τ_{12}	x_{13}	1,1,-,-,-,1
		τ_{F_2}	x_{30}	1,1,-,-,-,1	x_8	0,0,-,-,-,2	τ_{10}	x_{10}	0,1,-,-,-,2	x_{13}	1,1,-,-,-,1	τ_{13}	x_{11}	1,0,-,-,-,1
x_{11}	1,0,-,-,-,1	τ_{14}	x_9	0,0,-,-,-,1	x_{14}	0,0,-,-,-,0	τ_{16}	x_{15}	0,0,-,-,-,2	x_{15}	0,0,-,-,-,2	τ_{18}	x_{17}	0,1,-,-,-,2
x_9	0,0,-,-,-,1	τ_{15}	x_8	0,0,-,-,-,2			τ_{17}	x_{16}	0,0,-,-,-,1	x_{17}	0,1,-,-,-,2	τ_{19}	x_{18}	0,1,-,-,-,1
		τ_{34}	x_{11}	1,0,-,-,-,1	x_{16}	0,0,-,-,-,1	τ_{21}	x_{15}	0,0,-,-,-,2	x_{18}	0,1,-,-,-,1	τ_{20}	x_{16}	0,0,-,-,-,1
x_{19}	0,0,-,-,-,0	τ_{22}	x_{20}	0,0,-,-,-,2	x_{20}	0,0,-,-,-,2	τ_{24}	x_{22}	0,1,-,-,-,2	x_{21}	0,0,-,-,-,1	τ_{25}	x_{23}	0,1,-,-,-,1
		τ_{23}	x_{21}	0,0,-,-,-,1	x_{22}	0,1,-,-,-,2	τ_{26}	x_{24}	1,1,-,-,-,2	x_{23}	0,1,-,-,-,1	τ_{27}	x_{25}	1,1,-,-,-,1
x_{24}	1,1,-,-,-,2	τ_{28}	x_{25}	1,1,-,-,-,1	x_{25}	1,1,-,-,-,1	τ_{29}	x_{24}	1,1,-,-,-,2	x_{26}	0,0,-,-,-,0	τ_{30}	x_{27}	0,0,-,-,-,2
x_{27}	0,0,-,-,-,2	τ_{32}	x_{28}	0,0,-,-,-,1	x_{29}	1,1,-,-,-,2	τ_{35}	x_{17}	0,1,-,-,-,2			τ_{31}	x_{28}	0,0,-,-,-,1
x_{28}	0,0,-,-,-,1	τ_{33}	x_{27}	0,0,-,-,-,2	x_{30}	1,1,-,-,-,1	τ_{36}	x_{18}	0,1,-,-,-,1					

the set of measurable state variables in G_1 becomes $\{S_{FS}, S_V, S_L\}$, we represent the symbol '-' as the value of an unmeasurable state variable in S_x. Now, we apply our proposed *MLAD* approach on the model G for the HVAC system under consideration to obtain two partially compromised reduced diagnosers G_{diag2} and G_{diag3}. These reduced diagnosers are constructed from the abstract models HG_1 (refer Table 6.8) and HG_2 (refer Table 6.9), respectively. The sets of measurable state variables corresponding to HG_1 and HG_2 become $\{S_{FS}, S_V\}$ and $\{S_V, S_L\}$, respectively. G_{diag2} which consists

Table 6.8: *State-Transition Table: Abstract Model HG_1*

x	S_x	τ	x^+	S_{x^+}	x	S_x	τ	x^+	S_{x^+}	x	S_x	τ	x^+	S_{x^+}
x_1	0,0,-,-,-,-	τ_1	x_2	0,1,-,-,-,-	x_2	0,1,-,-,-,-	τ_2	x_3	1,1,-,-,-,-	x_3	1,1,-,-,-,-	τ_3	x_1	0,0,-,-,-,-
		τ_{F_1}	x_4	0,0,-,-,-,-			τ_{F_1}	x_5	0,1,-,-,-,-			τ_{F_1}	x_6	1,1,-,-,-,-
		τ_{F_2}	x_8	0,0,-,-,-,-			τ_{F_2}	x_9	0,1,-,-,-,-			τ_{F_2}	x_{15}	1,1,-,-,-,-
		τ_{F_3}	x_{10}	0,0,-,-,-,-	x_4	0,0,-,-,-,-	τ_4	x_5	0,1,-,-,-,-	x_6	1,1,-,-,-,-	τ_6	x_7	1,0,-,-,-,-
		τ_{F_4}	x_{13}	0,0,-,-,-,-	x_5	0,1,-,-,-,-	τ_5	x_6	1,1,-,-,-,-	x_7	1,0,-,-,-,-	τ_7	x_4	0,0,-,-,-,-
x_8	0,0,-,-,-,-	τ_9	x_9	0,1,-,-,-,-	x_{10}	0,0,-,-,-,-	τ_{12}	x_{11}	0,1,-,-,-,-	x_4	0,0,-,-,-,-	τ_8	x_7	1,0,-,-,-,-
x_9	0,1,-,-,-,-	τ_{10}	x_8	0,0,-,-,-,-	x_{11}	0,1,-,-,-,-	τ_{13}	x_{12}	1,1,-,-,-,-	x_{15}	1,1,-,-,-,-	τ_{11}	x_9	0,1,-,-,-,-

Table 6.9: *State-Transition Table: Abstract Model HG_2*

x	S_x	τ	x^+	S_{x^+}	x	S_x	τ	x^+	S_{x^+}	x	S_x	τ	x^+	S_{x^+}
x_1	-,0,-,-,-,0	τ_1	x_2	-,0,-,-,-,2	x_2	-,0,-,-,-,2	τ_3	x_4	-,1,-,-,-,2	x_3	-,0,-,-,-,1	τ_6	x_2	-,0,-,-,-,2
		τ_2	x_3	-,0,-,-,-,1			τ_{F_1}	x_7	-,0,-,-,-,2			τ_{F_1}	x_8	-,0,-,-,-,1
		τ_{F_1}	x_6	-,0,-,-,-,0			τ_{F_2}	x_{12}	-,0,-,-,-,2			τ_{F_2}	x_{13}	-,0,-,-,-,1
		τ_{F_2}	x_{11}	-,0,-,-,-,0	x_4	-,1,-,-,-,2	τ_4	x_5	-,1,-,-,-,1	x_5	-,1,-,-,-,1	τ_5	x_3	-,0,-,-,-,1
		τ_{F_3}	x_{16}	-,0,-,-,-,0			τ_{F_1}	x_9	-,1,-,-,-,2			τ_{F_1}	x_{10}	-,1,-,-,-,1
		τ_{F_4}	x_{21}	-,0,-,-,-,0			τ_{F_2}	x_{14}	-,1,-,-,-,2			τ_{F_2}	x_{15}	-,1,-,-,-,1
x_6	-,0,-,-,-,0	τ_7	x_7	-,0,-,-,-,2	x_7	-,0,-,-,-,2	τ_9	x_9	-,1,-,-,-,2	x_8	-,0,-,-,-,1	τ_{12}	x_7	-,0,-,-,-,2
		τ_8	x_8	-,0,-,-,-,1	x_9	-,1,-,-,-,2	τ_{10}	x_{10}	-,1,-,-,-,1	x_{10}	-,1,-,-,-,1	τ_{11}	x_8	-,0,-,-,-,1
x_{11}	-,0,-,-,-,0	τ_{13}	x_{12}	-,0,-,-,-,2	x_{12}	-,0,-,-,-,2	τ_{15}	x_{14}	-,1,-,-,-,2	x_{15}	-,1,-,-,-,1	τ_{17}	x_{13}	-,0,-,-,-,1
		τ_{14}	x_{13}	-,0,-,-,-,1	x_{14}	-,1,-,-,-,2	τ_{16}	x_{15}	-,1,-,-,-,1	x_{13}	-,0,-,-,-,1	τ_{18}	x_{12}	-,0,-,-,-,2
x_{16}	-,0,-,-,-,0	τ_{19}	x_{17}	-,0,-,-,-,2	x_{17}	-,0,-,-,-,2	τ_{21}	x_{19}	-,1,-,-,-,2	x_{18}	-,0,-,-,-,1	τ_{22}	x_{20}	-,1,-,-,-,1
		τ_{20}	x_{18}	-,0,-,-,-,1	x_{19}	-,1,-,-,-,2	τ_{23}	x_{20}	-,1,-,-,-,1	x_{20}	-,1,-,-,-,1	τ_{24}	x_{19}	-,1,-,-,-,2
x_{21}	-,0,-,-,-,0	τ_{25}	x_{22}	-,0,-,-,-,2	x_{22}	-,0,-,-,-,2	τ_{27}	x_{23}	-,0,-,-,-,1					
		τ_{26}	x_{23}	-,0,-,-,-,1	x_{23}	-,0,-,-,-,1	τ_{28}	x_{22}	-,0,-,-,-,2					

of just 9 states is F_1 and F_2-diagnosable but compromises the diagnosability of F_3 and F_4. On the other hand, G_{diag3} which consists of 14 states can diagnose faults F_3 and

F_4 only. Now, we observe that G_{diag2} and G_{diag3} when deployed in parallel can detect all four faults F_1, F_2, F_3 and F_4 with their combined additive state space having size $9 + 14 = 23$. Thus, G_{diag2} and G_{diag3} acting in union attains the same power as the monolithic diagnosers G_{diag} and G_{diag1} while at the same time consuming a much lower state space.

6.3.2 System II

The nitric acid cooling (NAC) system under normal operation maintains a desired set-point value for the temperature of the acid that is fed to the reactor. The degree of cooling is regulated by a controller which commands a valve to provide appropriate amounts of cooling water by sensing the output temperature. Similarly, the flow of incoming nitric acid is regulated by a nitric acid shutdown system which commands a control valve to operate accordingly. This nominal system behavior gets affected due to the occurrence of the following failures: i) *NAS_Failure (fault type F_1):* Control valve remains 'Open' due to failure of nitric acid shutdown system, ii) *Controller Failed-Off (fault type F_2):* Controller fails to issue control signals.

Table 6.10: *The effectiveness of MLAD: Nitric Acid Cooling System*

	Diagnosability			
Diagnoser	F_1	F_2	**Number of States**	**Measurable Variables**
Global Diagnoser G_{diag} (Sampath et al. [104])	✓	✓	312	All (refer, example 2.2 of Section II in [104]).
Reduced Diagnoser G_{diag1} (Zad et al. [115])	✓	✓	228	All except status of the load.
Abstract Diagnoser G_{diag2}	✓	χ	11	Status of the control valve and the pump.
Abstract Diagnoser G_{diag3}	χ	✓	16	Status of the cooling water valve, readings of the temperature sensor and the valve stem-position sensor.

The diagnosability of the NAC system discussed above is summarized in Table 6.10.

It can noticed that its G_{diag} containing 312 states ensures the diagnosability of all fault types. The reduced diagnoser G_{diag1} obtained by limiting a redundant variable, following the mechanism of Zad et al. [115], contains 228 states and has the same power as G_{diag} in terms of diagnosability. Now, we apply our proposed *MLAD* approach on the model for the NAC system to obtain two partially compromised reduced diagnosers G_{diag2} and G_{diag3}. G_{diag2} which consists of just 11 states can diagnose fault F_1 only. On the other hand, G_{diag3} which consists of 16 states is F_2-diagnosable but compromises the diagnosability of F_1. From Table 6.10, it can be further observed that G_{diag2} and G_{diag3} when deployed in parallel can detect both the faults F_1 and F_2, with their combined additive state space being $11 + 16 = 27$. So, there is a drastic reduction in state space compared to monolithic diagnosers. From Table 6.5 and Table 6.10, it may be observed that *MLAD* based diagnosis can achieve the same power as that of the monolithic diagnosers, however with much lower state spaces, and the gain gets enhanced with size of the systems.

6.4 Summary

In this chapter, we have presented a new fault diagnosis approach called MLAD which is capable of producing controlled partially compromised diagnosers with state spaces far lower than those obtained through the existing state-of-the-art mechanisms. MLAD carefully chooses a designated subset of variables whose forceful limitation can possibly lead to compromised diagnosability of only a stipulated subset of faults from a given set of faults, while not affecting the diagnosability of the remaining faults. For fault diagnosis, MLAD performs a stipulated number of such controlled limitations on the original model to obtain a set of partially compromised reduced diagnosers whose combination ensures diagnosability of all faults. We have exhibited this capability of MLAD through the case study of a state-based DES model of an electronic fuel injection system. Here, we have discussed the idea of fault compartmentalization using the concept of measurement limitation and experimentally shown the effectiveness of our approach. The next chapter summarizes the contributions of this dissertation and discusses a few possible extensions

to this research.

Chapter 7

Conclusions and Future Perspectives

In this chapter, we summarize the contributions of this dissertation and outline some possible directions for future work.

7.1 Discussion and Summarization

This thesis deals with the design of safety-critical systems in general and real-time safety-critical systems in particular. During the development of a safety-critical system, the design methodology may need to consider various stringent constraints, including those related to timeliness, resource utilization, fault-tolerance, power dissipation, cost, etc. We now enumerate a few important challenges that must be considered in the design of safety-critical systems.

- The first challenge relates to *timing constraints* associated with various safety-critical applications/tasks that co-execute in the system. The timing constraints of these applications are captured by their execution requirements and deadlines.

- The second challenge deals with the design of *safety and performance related constraints* such as fault tolerance, energy minimization, etc.

- The third challenge relates to *resource constraints* imposed by the underlying computing platform on which the system is implemented. Over the years, the nature of computing platforms used in real-time systems has seen a distinct transformation

from uni-cores to homogeneous multi-cores to heterogeneous multi-core systems. These computing platforms typically consist of a limited number of processing elements (i.e., resources). Therefore, the design strategies for safety-critical systems must be able to effectively utilize the processing capacity of the underlying platform to satisfy the computational demands of applications.

7.1.1 Overall Summary of Chapters and Thesis

This dissertation presents a few novel ideas towards the design of energy-efficient and fault-tolerant strategies for safety-critical systems keeping in view the challenges/hurdles discussed above. We now present brief summaries of these works in more detail.

In our first contributory chapter, Chapter 3, we have presented a novel energy-efficient scheduling strategy that aims to minimize static energy consumption in a real-time multiprocessor system. The underlying scheduling structure being based on ERfair, the proposed optimal proportional fair scheduler, named *ERfair Scheduler with Suspension on Multiprocessors* (ESSM), attempts to reduce system wide energy consumption by locally maximizing the processor suspension intervals while not sacrificing the ER-fairness timing constraints of the system. The proposed technique takes advantage of higher execution rates of tasks in underloaded ERfair systems and uses a novel procrastination scheme to search for time points within the schedule where shutdown interval lengths may be locally maximized. ESSM not only ensures 100% resource utilization but also guarantees that fairness accuracy of no task will ever be violated due to the procrastination applied. We have designed, implemented, and evaluated the ESSM algorithm and proved the feasibility of this scheme. The simulation-based experimental results are promising.

In Chapter 3, we have assumed the underlying hardware computing platform to be fault-free. However, the processing platforms are subject to a variety of faults. Therefore, apart from guaranteeing the timely execution of tasks in a resource-constrained environment, ensuring the proper functioning of the system even in the presence of faults has currently become a design constraint of paramount importance. Hence, in Chapter 4, we have presented a fault-tolerant proportional fair scheduling mechanism

called *Fault Tolerant Fair Scheduler* (FT-FS), for real-time multiprocessor systems containing cold-standby spares. Subsequent to the detection of a permanent processor fault, the system requires a fixed recovery interval to boot up the spare processor to the operational state. Equipped with two novel features namely, weight donation and post rejection backtracking, the proposed scheduler FT-FS attempts to minimize rejections of critical jobs, during transient overloads within recovery intervals. The objective is to maximize the possibility of keeping the system fail-operational even in the presence of faults. The underlying scheduling structure being based on DP-Fair, FT-FS is able to ensure high resource utilization and fair rate-based execution progress while incurring low scheduling related overheads through controlled migrations and context switches. Experimental results reveal that the FT-FS algorithm performs appreciably over an extensive set of system scenarios pointing to the practical effectiveness of the scheme.

In the earlier chapters, we have assumed the underlying hardware computing platform to be homogeneous. However, the nature of processing platforms used in embedded systems is changing over the years. To satisfy the computational demands of various applications, today, we observe an increased emphasis towards the integration of unrelated processing cores (i.e., heterogeneity) onto a single hardware platform [15, 35]. In Chapter 5, we have presented a combined fault-tolerant and energy-aware design strategy for real-time safety-critical systems having heterogeneous multi-cores as the computing platform. This chapter proposed a standby-sparing based fault-tolerant energy-aware scheduling strategy, named *FENA-SCHED*, for a heterogeneous dual-core system, consists of a power-hungry, high-performance core, and a power-efficient, relatively slow core. For a DPM-enabled system, we found that designating power-efficient (modest performance) core as primary and power-hungry (high-performance) core as spare yields better energy savings as compared to its counterpart. In order to minimize overall energy consumption and to tolerate a given number of faults, FENA-SCHED reserves only a fixed number of backup slots on the high-performance core and takes advantage of backup-backup overloading. Experimental results reveal that FENA-SCHED performs appreciably over an extensive set of system scenarios pointing to the practical effective-

ness of the scheme and is able to significantly improve energy savings of the system, compared to the state-of-the-art work.

Chapters 4 and 5 have assumed that faults are always detectable, and have aimed towards the design of efficient fault-tolerant procedures that provide functional correctness in the presence of faults. However, enforcement of such fault tolerance can only be achieved through the incorporation of safe design methodologies which enable efficient active monitoring and detection of unobservable faults in the system. Therefore, it is desirable to incorporate efficient fault diagnosis (detection and isolation) design strategies in the construction of safety-critical systems. Hence, in Chapter 6, as our last contributory chapter, we have presented a formal fault diagnosis strategy for the design of safety-critical systems. The primary objective of the research work in this chapter is to develop an efficient strategy which attempts to reduce state space complexity involved in the design of the fault diagnosis process. In this chapter, we have developed a new fault diagnosis approach called MLAD, which is capable of producing controlled partially compromised diagnosers with state spaces far lower than those obtained through the existing state-of-the-art mechanisms. MLAD carefully chooses a designated subset of variables whose forceful limitation can possibly lead to compromised diagnosability of only a stipulated subset of faults from a given set of faults, while not affecting the diagnosability of the remaining faults. For fault diagnosis, MLAD performs a stipulated number of such controlled limitations on the original model to obtain a set of partially compromised reduced diagnosers whose combination ensures diagnosability of all faults. We have exhibited this capability of MLAD through the case study of a state-based DES model of an electronic fuel injection system. Here, we have discussed the idea of fault compartmentalization using the concept of measurement limitation and experimentally shown the effectiveness of our approach.

In summary, the work conducted as part of this thesis has dealt with the development of efficient energy-aware and fault-tolerant design strategies for safety-critical systems. Figure 7.1 depicts a pictorial representation of the thesis workflow. This research work is divided into three modules. Module 1 which comprises Chapter 3 in the thesis, deals

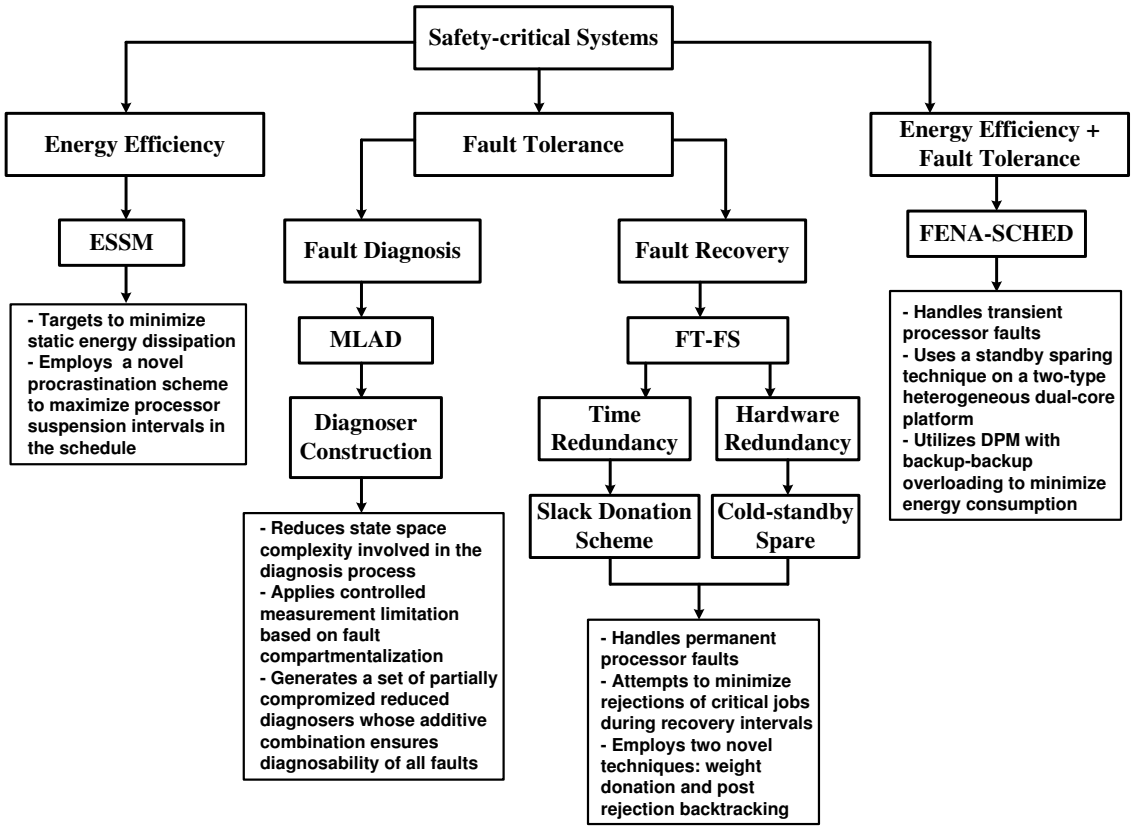


Figure 7.1: *Thesis Workflow Diagram*

with the development of an energy-efficient design strategy for safety-critical systems. The primary objective of the work presented in Module 1 is to reduce the static energy consumption in the system by employing a novel procrastination based scheduling technique. Chapters 4 and 6 together constitute Module 2 which endeavor to develop efficient fault-tolerant design mechanisms. Module 2 consists of two components: fault diagnosis (Chapter 6) and recovery (Chapter 4) sub-modules. In the fault diagnosis module, faults are assumed to be unobservable. Therefore, this module deals with the development of a formal automata based fault detection mechanism, named as *Diagnoser*, to actively monitor the system and detect the presence of unobservable faults in the system. On the contrary, the fault recovery module performs a set of recovery operations to make the system operational subsequent to the detection of faults. The research work presented in this module employs a combined time and hardware redundancy based fault-tolerant

technique to handle the effects of faults in the system. The scheme handles faults using cold-standby spare hardwares. However, such spare hardwares become available only after an interval called recovery time subsequent to the detection of a permanent processor fault. In order to keep the system fail-operational during this recovery interval, the scheme uses redundant time or slack capacity to reconfigure the workload priorities so that all tasks can meet their timeliness demands during the recovery period. The third module (Module 3) comprises Chapter 5 in the thesis and deals with the development of a combined energy-efficient and fault-tolerant design strategy for heterogeneous systems. We employ the well known standby-sparing technique for fault tolerance and DPM to minimize energy consumption.

7.1.2 Comparison Across the Proposed Techniques

This section discusses a few important system scenarios that may be used to compare the scheduling techniques developed across the chapters in this dissertation. They are listed as follows: i) Comparison of basic scheduling schemes, ii) Fault tolerance in the ERfair system, iii) Fault tolerance in the ESSM system iv) Energy awareness in the DP-Fair system v) Fault tolerance and Energy awareness in the DP-Fair system. Among these, the last four scenarios (ii-v) may be considered as possible immediate extensions to the works presented in Chapter 3 and Chapter 4. We now discuss each of these scenarios in detail.

i) Comparison of basic scheduling schemes: In Chapter 3, we have presented an energy-efficient fair scheduling scheme, called ESSM for homogeneous multiprocessor systems. ESSM uses an optimal, strictly fair scheduling scheme called as ERfair [6] as the underlying scheduling mechanism. ERfair is a work-conserving version of Proportionate fair (Pfair) scheduling introduced by Baruah et al. in [16], and so, it never allows a processor to be idle in the presence of runnable/ready tasks. Since ERfair follows a strict proportional fair strategy, it is an attractive alternative for applications where meeting fairness is not an option but a necessity, as in applications like real-time audio processing, streaming video, interactive gaming, and so on. However, ERfair is a

global scheduling scheme and thus, incurs unrestricted preemption/migration overheads to maintain fair proportional progress for all tasks at all time slots. On the other hand, the fault-tolerant scheme presented in Chapter 4 uses a work conserving version of the DP-Fair algorithm [71] as the underlying scheduling mechanism. DP-Fair is able to deliver optimal resource utilization while enforcing strict proportional fairness (ERfairness) only at period/deadline boundaries. Therefore, it is an approximate proportional fair scheduler. Moreover, DP-Fair is a semi-partitioned scheduling technique that allows at most $m-1$ task migrations and $n-1$ preemptions within a time slice and thus incurs much lower overheads compared to ERfair. It may be noted that the scheduling works presented in Chapters 3 and 4 have assumed the underlying hardware computing platform to be *homogeneous*. As different from these works, Chapter 5 presents a low-overhead heuristic scheme called FENA-SCHED, for the energy-aware fault-tolerant scheduling of real-time applications on *heterogeneous* dual-core systems. FENA-SCHED follows a primary-backup fault-tolerant scheduling strategy and does not impose any fairness constraint on the task system. FENA-SCHED utilizes dynamic power management (DPM) with backup-backup overloading [46, 47] to minimize energy consumption while guaranteeing tolerance against a given number of transient processor faults.

ii) Incorporating fault tolerance in the ERfair system: Let us consider a fault model similar to one defined in Chapter 4, for the ERfair system. The objective of any fault-tolerant scheme in an ERfair system is to maintain an ERfair schedule within the recovery period, which means that every task has to meet its pseudo-deadlines within the recovery period. To generate an effective fault-tolerant scheme for such systems, it is desirable to have a few assumptions at the time of fault detection (represented as F_{DT}). As a part of the fault model, we assume that one processor was faulty for some duration of time before the detection of a fault, and no task was actually executed on that faulty processor for that interval. We also assume that no task has missed its pseudo-deadline at F_{DT}. At F_{DT}, a few tasks may have actually executed more than their stipulated execution rates, and a few may have executed less than their required rates. So, everybody has a different amount of slacks at F_{DT}. In this situation, a slack

distribution strategy similar to the weight donation scheme used in FT-FS, may be formulated so that this slack distribution provides a best possibility for every task to meet its pseudo-deadline within the recovery interval. If we assume that some tasks have missed their pseudo-deadlines at F_{DT}, then we have to reject this deadline missed tasks and have to allocate time slots of the rejected tasks to others for meeting their pseudo-deadlines, as one similar to FT-FS.

iii) Incorporating fault tolerance in the ESSM system: The basic technique used for achieving energy savings in ESSM and fault tolerance in FT-FS is the efficient distribution of slacks generated in an underloaded system. Thus, the goals of energy efficiency and that of fault tolerance may be considered to be conflicting in nature. Hence, simultaneously satisfying both these goals will necessitate a careful trade-off. In safety-critical systems, fault tolerance always has the highest priority over energy efficiency. As discussed in the above paragraph, we first consider a fault model similar to one defined in Chapter 4, for the ESSM system. In this fault model, we assume that faults occur on processors when they are active, that is, no faults are allowed to occur on processors when they are suspended. We now determine the amount of slack accumulated by each task in the system at fault detection time F_{DT}. Subsequently, we design a slack distribution strategy that initiates the donation of slacks to the required set of tasks so that the system becomes fault-tolerant during the recovery period. Then we find out how much additional slack is available after providing the required fault tolerance to the system. If there is such an additional slack available in the system, then it may be used for energy minimization.

Now, we provide an illustrative example that demonstrates how the accumulated slack in the system can be used for both fault tolerance and energy awareness, by considering ESSM as the underlying scheduling scheme. In this example, we first utilize a portion of the slack available in the system to provide one processor fault tolerance for a given recovery period, and then we use the additional slack available to minimize energy consumption in the system.

Example: Consider a set of five tasks, $T_1(10, 50)$, $T_2(12, 60)$, $T_3(30, 150)$, $T_4(30, 150)$, and $T_5(20, 50)$ to be executed on two unit capacity processors ($m = 2$) using the ESSM scheduling scheme. As similar to the fault model discussed in Chapter 4, we assume that the system can handle at most one transient/permanent processor fault at any given time; no further faults are assumed to occur during the recovery period $[F_{DT}, F_{RT})$. Here, $F_{OT} = F_{DT} = 34$ and $F_{RT} = 50$. $u_1 = u_2 = u_3 = u_4 = 1/5$ and $u_5 = 2/5$. The total system utilization at $t = 0$ is given by $U_{A \cup C} = U_R = \sum_{i=1}^{5} u_i = 6/5$. The system is feasibly schedulable at $t = 0$ since it is underloaded (that is, $U_{A \cup C}$ $(= 6/5) < m$ $(= 2)$). As system utilization $U_R = 6/5$ and system capacity $\lceil U_{A \cup C} \rceil = 2$ at time $t = 0$, the effective execution rates of the *Running* tasks are $eu_1 = eu_2 = eu_3 = eu_4 = 1/3$ and $eu_5 = 2/3$. The slack generation rate of these tasks is 2/3. At t=0, the estimated finish times of T_1, T_2, T_3, T_4 and T_5 are $f_1 = 30$, $f_2 = 36$, $f_3 = 90$, $f_4 = 90$, and $f_5 = 30$. Since T_1 and T_5 have the earliest finishing time $f_1 = f_5 = 30$, the next potential suspension point ($EPSP$) is obtained at $t = 30$. Figure 7.2 depicts this scenario.

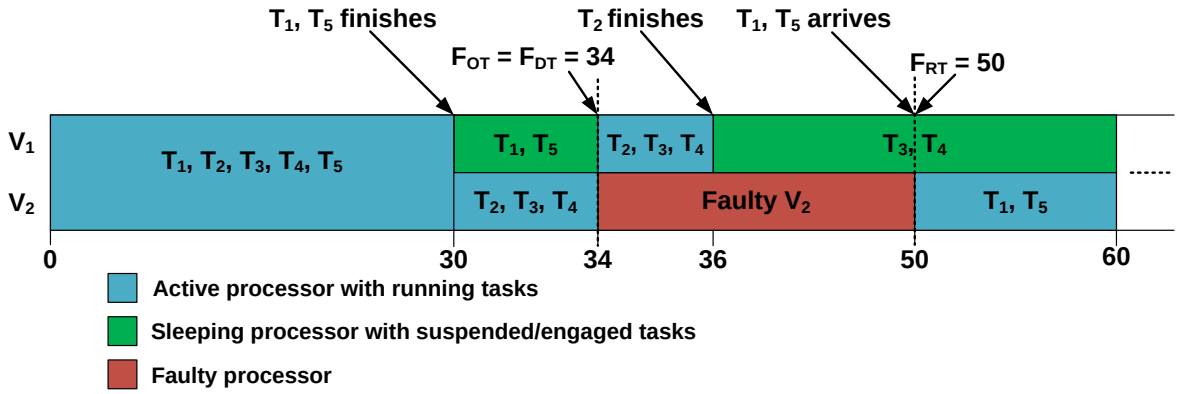


Figure 7.2: *ESSM with fault tolerance*

Both T_1 and T_5 finish their executions at $t = 30$, and are moved to the set of $Free$ tasks. The slacks of T_1, T_2, T_3, T_4, and T_5 are 20. At $t = 30$, $Slack_1(30) = Slack_5(30) = 20$, and $u_1 + u_5 \geq U_{R \cup F} - \lfloor U_{R \cup F} \rfloor$. Now, ESSM generates an admissible task set using the $Free$ tasks T_1 and T_5 with a group slack of 20. Therefore, one processor (say, V_1) can be shutdown for the next 20 time slots at $t = 30$ with the next earliest potential wakeup point ($EPWP$) being $t = 50$. The *Running* tasks at $t = 30$ are T_2, T_3, and T_4.

U_R becomes 3/5 and slack generation rate is $SR = 2/3$. The effective execution rates of T_2, T_3, and T_4 become 1/3, and their corresponding estimated finish times are $f_2 = 36$, $f_3 = 90$, and $f_4 = 90$. Therefore, $EPSP = 36$.

The processor V_2 now suffers a fault at time $F_{OT} = 34$. The fault is detected at $F_{DT} = 34$ and the recovery interval t_r being 16, we get $F_{RT} = 50$. As part of our fault-tolerant strategy, we wake up the sleeping processor V_1 to handle the transient overload caused by the fault. At $t = 34$, the slacks of T_1 and T_5 are used for handling the fault, instead of allowing a processor to sleep. Now, the wake up processor V_1 is able to accommodate the *Running* tasks T_2, T_3, and T_4. U_R becomes 3/5 and slack generation rate is $SR = 2/3$. The effective execution rates of T_2, T_3, and T_4 become 1/3 and their corresponding estimated finish times are $f_2 = 36$, $f_3 = 90$, and $f_4 = 90$. Therefore, $EPSP = 36$. At time $t = 36$, T_2 finishes and is moved to the set of *Free* tasks. The slacks of T_2, T_3, and T_4 are same (= 24). Here, the fault-tolerant ESSM generates an admissible task set using the *Running* tasks T_3, and T_4 with a group slack of 24. Therefore, the non-faulty processor V_1 will be suspended between $t = 36$ and $t = 60$. Figure 7.2 depicts this scenario.

At time $t = F_{RT} = 50$, the faulty processor V_2 becomes operational and the system recovers from the fault. The next instances of tasks T_1 and T_5 also arrive at this time. At $t = 50$, $U_R = 3/5$ and the slack generation rates of the *Running* tasks T_1 and T_5 become $SR = 2/3$. $eu_1 = 1/3$, $eu_5 = 2/3$, and $f_1 = f_5 = 80$. Therefore, $EPSP = 80$, the finishing time of T_1 and T_5. The next $EPWP$ is obtained at $t = 60$, the wake up time of V_1. The rest of the schedule continues as ESSM schedule. □

iv) Incorporating energy awareness in the DP-Fair system: In an underloaded DP-Fair system, tasks allocated to a processor within a time slice execute in an EDF-like fashion starting from the beginning of the slice and complete their execution before the end of the time slice is reached. After executing the allocated task shares, each processor idles up to the end of the current time slice. One straight forward energy-saving method is to suspend processors during such idle intervals provided the duration of these intervals is at least equal to the break-even time TI_{be}. We can also employ our energy-aware

strategy used in ESSM to the DP-Fair scheduling scheme. For this purpose, we first have to design a work-conserving version of DP-Fair, similar to one discussed in Chapter 4. Then we can apply a similar slack generation strategy used in ESSM to accumulate slacks in the system and use a procrastination scheme to suspend the processors. This ESSM based energy-aware strategy can be improved further by applying the concept of slack donation (as one discussed in Chapter 4). This slack donation enhances the group slack available at viable suspension points and thus, maximizes sleep durations.

v) Incorporating fault tolerance and energy awareness in the DP-Fair system: To incorporate an energy-aware strategy over the proposed fault-tolerant scheduling scheme FT-FS presented in Chapter 4, we can make use of the slack distribution strategy in the third system scenario (ESSM with fault tolerance), discussed above. Here, we first utilize a portion of the slack available in the system to provide one processor fault tolerance for a given recovery period, and then we use the additional slack available to minimize energy consumption in the system.

In the next section, we outline some possible directions for future work.

7.2 Future Works

The work presented in this thesis leaves several open directions and there is ample scope for future research in this area. In this section, we present four such future perspectives.

- **Design strategies for total system energy minimization:** Techniques for controlling power/energy consumption are being applied at all system levels starting from hardware and firmware to the architectural, system, and even application levels. As discussed in Chapter 2, at the operating system level, two primary mechanisms are generally used to reduce energy consumption: (1) Dynamic Power Management (DPM) [21,67] and (2) Dynamic Voltage and Frequency Scaling (DVFS) [89,108]. The first mechanism involves suspending/shut-downing the system when the processor is idling because the energy consumed during the suspension period is negligible. DVFS, on the other hand, involves lowering the pro-

cessor's operating frequency by appropriately scaling its supply voltage when the full speed is not required. As energy dissipated per cycle in CMOS circuits scale quadratically to the supply voltage, this strategy is able to provide large energy savings in DVFS-enabled processors. In Chapters 3 and 5, we have utilized the DPM technique to minimize energy consumption in real-time multiprocessor/multi-core systems. It would be interesting to extend the works presented in these chapters by integrating the DPM technique with existing DVFS schemes in DVFS-enabled processors that support a set of discrete voltage/frequency levels. This integration endeavors to minimize total energy consumption (including both static and dynamic) in the system.

- **Design strategies for peak power constrained safety-critical systems:** Nowadays, modern multi-core chips are designed with a *Thermal Design Power* (TDP) value which is considered to be the highest sustainable power that a chip can consume without triggering any performance throttling mechanism such as *Dynamic Thermal Management* (DTM) [88]. Whenever the power dissipation in the system crosses this specified TDP value, DTM is triggered to ensure thermal stability of the system. However, activation of DTM introduces unpredictability in the timing behavior of the safety-critical real-time applications executing on the system [88]. Therefore, in order to ensure the timely execution of these applications, we need to keep the cumulative peak power consumption of the cores to be within the specified TDP value. As future work, we plan to develop efficient scheduling mechanisms which not only guarantee to satisfy the timing requirements of safety-critical applications but also minimize peak power consumption within the TDP constraint. In addition, we also plan to develop a reliability-aware model over this peak power-aware scheduling scheme.

- **Design strategies for parallel real-time applications:** In Chapters 3, 4, and 5 of this dissertation, we have considered the scheduling of independent real-time applications on a multiprocessor/multi-core platform. Applications in many safety-critical real-time systems, ranging from avionic, automotive and industrial control

to telecommunication systems, health care and even a significant class of consumer electronics systems, are often highly parallelizable [41]. One of the most generic mechanisms for modeling parallel real-time applications is the *Directed Acyclic Graph* (DAG) [31, 32, 96]. In the DAG model of an application, nodes correspond to the tasks, and edges denote inter-task dependencies. In the near future, we intend to propose an efficient energy-aware and fault-tolerant scheduling strategy for parallel real-time applications represented as DAGs, executing on a multi-core platform.

- **Design strategies for safety-critical systems on distributed processing platforms:** The scheduling (energy-aware and fault-tolerant) mechanisms presented in this dissertation assumed tightly-coupled interconnection of individual processing elements as in a symmetric multi-core system. Hence, inter-processor communication delays have been ignored in the system design process. However, this assumption is not valid in the case of a safety-critical system to be implemented on loosely-coupled distributed processing platforms [109]. The applications that execute on different computing elements of a distributed system communicate with each other typically by exchanging the messages through an interconnected communication network. In such a distributed environment, the real-time requirements of applications not only depend on the processing speed of the computing elements but also rely on specifications of the underlying communication network. As future work, we plan to extend the research presented in this dissertation to develop efficient design strategies for safety-critical distributed systems.

Disseminations out of this Work

Journal Papers

1. Piyoosh Purushothaman Nair, Arnab Sarkar, N. M. Harsha, Megha Gandhi, P. P. Chakrabarti, and Sujoy Ghose. "ERfair scheduler with processor suspension for real-time multiprocessor embedded systems." ACM Transactions on Design Automation of Electronic Systems (TODAES), Volume 22, Issue 1, Pages 19:1-19:25, 2016.

2. Piyoosh Purushothaman Nair, Santosh Biswas, and Arnab Sarkar. "Design of light weight exact discrete event system diagnosers using measurement limitation: case study of electronic fuel injection system." International Journal of Systems Science, Volume 49, Issue 8, Pages 1760-1783, 2018.

3. Piyoosh Purushothaman Nair, Arnab Sarkar and Santosh Biswas. "Fault-tolerant Real-time Fair Scheduling on Multiprocessor Systems with Cold-standby." IEEE Transactions on Dependable and Secure Computing (TDSC). (Early Access). doi: 10.1109/TDSC.2019.2934098.

4. Piyoosh Purushothaman Nair, Arnab Sarkar and Santosh Biswas. "FENA-SCHED: Fault and Energy Aware Scheduling on Heterogeneous Dual-cores." Microprocessors and Microsystems-Embedded Hardware Design, Elsevier. (Under review).

Conference Papers

1. Piyoosh Purushothaman Nair, Rajesh Devaraj, Argha Sen, Arnab Sarkar, and Santosh Biswas. "DES based Modeling and Fault Diagnosis in Safety-critical Semi-Partitioned Real-time Systems." IFAC-PapersOnLine, Volume 50, Issue 1, Pages 5029-5034, 2017.

2. Piyoosh Purushothaman Nair, Rajesh Devaraj, and Arnab Sarkar. "FEST: Fault-Tolerant Energy-Aware Scheduling on Two-Core Heterogeneous Platform." In 2018 8th International Symposium on Embedded Computing and System Design (ISED), Pages 63-68, IEEE, 2019.

References

[1] M. Agarwal, S. Biswas, and S. Nandi, "I2-diagnosability framework for detection of advanced stealth man in the middle attack in wi-fi networks," in *2015 23rd Mediterranean Conference on Control and Automation (MED)*, June 2015, pp. 349–356. [Pg.39]

[2] R. Al-Omari, A. K. Somani, and G. Manimaran, "An adaptive scheme for fault-tolerant scheduling of soft real-time tasks in multiprocessor systems," *Journal of Parallel and Distributed Computing*, vol. 65, no. 5, pp. 595–608, 2005. [Pg.34], [Pg.36]

[3] P. Altenbernd, "Deadline-monotonic software scheduling for the co-synthesis of parallel hard real-time systems," in *EDTC '95: 1995 European conference on Design and Test*. Washington, DC, USA: IEEE Computer Society, 1995, p. 190. [Pg.22]

[4] S. Andalam, P. S. Roop, A. Girault, and C. Traulsen, "A predictable framework for safety-critical embedded systems," *IEEE Transactions on Computers*, vol. 63, no. 7, pp. 1600–1612, 2013. [Pg.1]

[5] J. Anderson and A. Srinivasan, "Early-release fair scheduling," in *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, Jun 2000, pp. 35–43. [Pg.27], [Pg.28], [Pg.33], [Pg.57], [Pg.60], [Pg.71]

[6] J. H. Anderson and A. Srinivasan, "Early-release fair scheduling," in *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*. IEEE, 2000, pp. 35–43. [Pg.174]

[7] J. H. Anderson and A. Srinivasan, "Mixed pfair/erfair scheduling of asynchronous periodic tasks," *J. Comput. Syst. Sci.*, vol. 68, no. 1, pp. 157–204, 2004. [Pg.94]

[8] B. Andersson and J. Jonsson, "The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%," in *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on.* IEEE, 2003, pp. 33–40. [Pg.25], [Pg.36]

[9] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemptions," in *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on.* IEEE, 2006, pp. 322–334. [Pg.25]

[10] A. Armoush, "Design patterns for safety-critical embedded systems." Ph.D. dissertation, RWTH Aachen University, 2010. [Pg.1]

[11] T. Atdelzater, E. M. Atkins, and K. G. Shin, "Qos negotiation in real-time systems and its application to automated flight control," *IEEE Transactions on Computers*, vol. 49, no. 11, pp. 1170–1183, 2000. [Pg.115]

[12] M. Awan and S. Petters, "Energy-aware partitioning of tasks onto a heterogeneous multi-core platform," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, April 2013, pp. 205–214. [Pg.31]

[13] P. Baptiste, M. Chrobak, and C. Durr, "Polynomial time algorithms for minimum energy scheduling," in *15th Annual European Symposium on Algorithms (ESA)*, 2007, pp. 136–150. [Pg.3], [Pg.31]

[14] P. Baptiste, "Scheduling unit tasks to minimize the number of idle periods: a polynomial time algorithm for offline dynamic power management," in *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm.* Society for Industrial and Applied Mathematics, 2006, pp. 364–367. [Pg.3]

[15] S. Baruah, M. Bertogna, and G. Buttazzo, *Multiprocessor scheduling for real-time systems.* Springer, 2015. [Pg.xvii], [Pg.3], [Pg.4], [Pg.20], [Pg.119], [Pg.171]

[16] S. K. Baruah *et al.*, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996. [Pg.27], [Pg.174]

[17] A. Bastoni, B. Brandenburg, and J. Anderson, "Cache-related preemption and migration delays: Empirical approximation and impact on schedulability," *Proceedings of OSPERT*, pp. 33–44, 2010. [Pg.85], [Pg.111]

[18] M. A. Bender, R. Clifford, and K. Tsichlas, "Scheduling algorithms for procrastinators," *Journal of Scheduling*, vol. 11, no. 2, pp. 95–104, 2008. [Pg.3]

[19] L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 8, no. 3, pp. 299–316, 2000. [Pg.58]

[20] A. Bhat, S. Samii, and R. R. Rajkumar, "Recovery time considerations in real-time systems employing software fault tolerance," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 106. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018. [Pg.34], [Pg.37]

[21] M. Bhatti, M. Farooq, C. Belleudy, and M. Auguin, "Controlling energy profile of rt multiprocessor systems by anticipating workload at runtime," in *SYMPosium en Architectures nouvelles de machines*, 2009. [Pg.30], [Pg.31], [Pg.32], [Pg.179]

[22] P. Bhowal, D. Sarkar, S. Mukhopadhyay, and A. Basu, "Fault diagnosis in discrete time hybrid systems–a case study," *Information Sciences*, vol. 177, no. 5, pp. 1290–1308, 2007. [Pg.140]

[23] S. Biswas, D. Sarkar, P. Bhowal, and S. Mukhopadhyay, "Diagnosis of delaydeadline failures in real time discrete event models," {*ISA*} *Transactions*, vol. 46, no. 4, pp. 569 – 582, 2007. [Pg.39]

[24] S. Biswas, D. Sarkar, and S. Mukhopadhyay, "Diagnosability of delay-deadline failures in fair real time discrete event models," *International Journal of Systems Science*, vol. 41, no. 7, pp. 763–782, 2010. [Pg.47]

[25] S. Biswas, D. Sarkar, S. Mukhopadhyay, and A. Patra, "Diagnosability of fair discrete event systems," *Asian Journal of Control*, vol. 10, no. 6, pp. 651–665, 2008. [Pg.39]

[26] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer Science & Business Media, 2011, vol. 24. [Pg.1], [Pg.17], [Pg.20], [Pg.25]

[27] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, "A categorization of real-time multiprocessor scheduling problems and algorithms," *Handbook on scheduling algorithms, methods, and models*, pp. 30–1, 2004. [Pg.22], [Pg.25], [Pg.31]

[28] C. G. Cassandras and S. Lafortune, *Introduction to discrete event systems*. Springer Science & Business Media, 2009. [Pg.5], [Pg.6], [Pg.16], [Pg.48], [Pg.140], [Pg.148]

[29] G. Chen, K. Huang, and A. Knoll, "Adaptive dynamic power management for hard real-time pipelined multiprocessor systems," in *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*. IEEE, 2014, pp. 1–10. [Pg.33]

[30] J.-J. Chen, H.-R. Hsu, and T.-W. Kuo, "Leakage-aware energy-efficient scheduling of real-time tasks in multiprocessor systems," in *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*. IEEE, 2006, pp. 408–417. [Pg.31]

[31] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic scheduling of real-time tasks under precedence constraints," *Real-Time Systems*, vol. 2, no. 3, pp. 181–194, 1990. [Pg.181]

[32] H. S. Chwa, J. Lee, J. Lee, K.-M. Phan, A. Easwaran, and I. Shin, "Global edf schedulability analysis for parallel tasks on multi-core platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1331–1345, 2016. [Pg.181]

[33] O. Contant, S. Lafortune, and D. Teneketzis, "Diagnosability of discrete event systems with modular structure," *Discrete Event Dynamic Systems*, vol. 16, no. 1, pp. 9–37, 2006. [Pg.48], [Pg.51], [Pg.54]

[34] R. Conway, W. Maxwell, and L. Miller, *Theory of Scheduling*, ser. Dover Books on Computer Science Series. Dover, 2003. [Pg.21]

[35] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys (CSUR)*, vol. 43, no. 4, p. 35, 2011. [Pg.25], [Pg.119], [Pg.171]

[36] R. Debouk, R. Malik, and B. Brandin, "A modular architecture for diagnosis of discrete event systems," in *Decision and Control, 2002, Proceedings of the 41st IEEE Conference on*, vol. 1. IEEE, 2002, pp. 417–422. [Pg.48], [Pg.51], [Pg.52], [Pg.53], [Pg.54], [Pg.158], [Pg.160]

[37] E. Dubrova, *Fault-tolerant design*. Springer, 2013. [Pg.2], [Pg.33], [Pg.34]

[38] R. Dunia and S. J. Qin, "Joint diagnosis of process and sensor faults using principal component analysis," *Control Engineering Practice*, vol. 6, no. 4, pp. 457 – 469, 1998. [Pg.140]

[39] A. Ejlali, B. M. Al-Hashimi, and P. Eles, "A standby-sparing technique with low energy-overhead for fault-tolerant hard real-time systems," in *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. ACM, 2009, pp. 193–202. [Pg.xxiii], [Pg.38], [Pg.135]

[40] N. El-Sayed and B. Schroeder, "Understanding practical tradeoffs in hpc checkpoint-scheduling policies," *IEEE Transactions on Dependable and Secure Computing*, 2016. [Pg.2], [Pg.33], [Pg.35]

[41] D. Ferry, J. Li, M. Mahadevan, K. Agrawal, C. Gill, and C. Lu, "A real-time scheduling service for parallel tasks," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2013, pp. 261–272. [Pg.181]

[42] C. Fidge, "Real-time scheduling theory, Tech. Rep. 02-19, Apr 2002. [Pg.21]

[43] P. M. Frank, "Fault diagnosis in dynamic systems using analytical and knowledge-based redundancy: A survey and some new results," *automatica*, vol. 26, no. 3, pp. 459–474, 1990. [Pg.140]

[44] V. Ganesan, *Internal Combustion Engines*. McGraw-Hill Education, 2012. [Pg.141]

[45] E. Gascard, Z. Simeu-Abazi, and B. Suiphon, "A polynomial-time algorithm for diagnosability verification of discrete event systems," in *Complex Systems (WCCS), 2014 Second World Conference on*. IEEE, 2014, pp. 286–291. [Pg.48], [Pg.49]

[46] S. Ghosh, R. Melhem, and D. Mosse, "Fault-tolerant scheduling on a hard real-time multiprocessor system," in *Parallel Processing Symposium, 1994. Proceedings., Eighth International*. IEEE, 1994, pp. 775–782. [Pg.39], [Pg.124], [Pg.175]

[47] S. Ghosh, R. Melhem, and D. Mossé, "Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems," *IEEE Transactions on Parallel and distributed systems*, vol. 8, no. 3, pp. 272–284, 1997. [Pg.39], [Pg.124], [Pg.175]

[48] L. Guo and J. Kang, "A Hybrid Process Monitoring and Fault Diagnosis Approach for Chemical Plants," *International Journal of Chemical Engineering*, vol. 2015, 2015. [Pg.140]

[49] Y. Guo, D. Zhu, and H. Aydin, "Generalized standby-sparing techniques for energy-efficient fault tolerance in multiprocessor real-time systems," in *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013 IEEE 19th International Conference on*. IEEE, 2013, pp. 62–71. [Pg.38]

[50] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 2001, pp. 3–14. [Pg.135]

[51] K. Han, G. Lee, and K. Choi, "Software-level approaches for tolerating transient faults in a coarse-grainedreconfigurable architecture," *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 4, pp. 392–398, 2014. [Pg.2], [Pg.34]

[52] M. A. Haque, H. Aydin, and D. Zhu, "Energy-aware standby-sparing technique for periodic real-time applications," in *Computer Design (ICCD), 2011 IEEE 29th International Conference on*. IEEE, 2011, pp. 190–197. [Pg.38]

[53] H. Huang, F. Xia, J. Wang, S. Lei, and G. Wu, "Leakage-aware reallocation for periodic real-time tasks on multicore processors." in *FCST*, 2010, pp. 85–91. [Pg.31]

[54] R. Isermann, "Fault detection with limit checking," in *Fault-Diagnosis Systems*. Springer, 2006, pp. 95–110. [Pg.139]

[55] R. Jejurikar, C. Pereira, and R. Gupta, "Leakage aware dynamic voltage scaling for real-time embedded systems," in *Proceedings of the 41st annual Design Automation Conference*. ACM, 2004, pp. 275–280. [Pg.29]

[56] S. Jiang, Z. Huang, V. Chandra, and R. Kumar, "A polynomial algorithm for testing diagnosability of discrete-event systems," *IEEE Transactions on Automatic Control*, vol. 46, no. 8, pp. 1318–1321, 2001. [Pg.48], [Pg.49]

[57] S. H. Kang, H. w. Park, S. Kim, H. Oh, and S. Ha, "Optimal checkpoint selection with dual-modular redundancy hardening," *IEEE Transactions on Computers*, vol. 64, no. 7, pp. 2036–2048, 2015. [Pg.2], [Pg.33], [Pg.35], [Pg.37]

[58] E. Kilic, "Diagnosability of fuzzy discrete event systems," *Information Sciences*, vol. 178, no. 3, pp. 858–870, 2008. [Pg.140]

[59] J. Kim, K. Lakshmanan, and R. Rajkumar, "R-batch: Task partitioning for fault-tolerant multiprocessor real-time systems," in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*. IEEE, 2010, pp. 1872–1879. [Pg.34], [Pg.36], [Pg.37]

[60] N. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. Hu, M. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *Computer*, vol. 36, pp. 68–75, 2003. [Pg.30]

[61] J. C. Knight, "Safety critical systems: challenges and directions," in *Proceedings of the 24th international conference on software engineering*. ACM, 2002, pp. 547–550. [Pg.1]

[62] I. Koren and C. M. Krishna, *Fault-tolerant systems*. Elsevier, 2010. [Pg.2], [Pg.34], [Pg.36], [Pg.122], [Pg.127]

[63] S. Kriaa, L. Pietre-Cambacedes, M. Bouissou, and Y. Halgand, "A survey of approaches combining safety and security for industrial control systems," *Reliability engineering & system safety*, vol. 139, pp. 156–178, 2015. [Pg.1]

[64] C. Krishna, "Fault-tolerant scheduling in homogeneous real-time systems," *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, p. 48, 2014. [Pg.2], [Pg.33], [Pg.34], [Pg.35], [Pg.36]

[65] H. Kweon, Y. Do, J. Lee, and B. Ahn, "An efficient power-aware scheduling algorithm in real time system," in *PacRim '07: IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Aug 2007, pp. 350–353. [Pg.30]

[66] I. Lee, J. Leung, and S. H. Son, Eds., *Handbook of Real-Time and Embedded Systems*. Chapman & Hall/CRC Press, 2007. [Pg.58], [Pg.121]

[67] V. Legout, M. Jan, and L. Pautet, "An off-line multiprocessor real-time scheduling algorithm to reduce static energy consumption," in *First Workshop on Highly-Reliable Power-Efficient Embedded Designs (HARSH)*, 2013, pp. 7–12. [Pg.30], [Pg.31], [Pg.32], [Pg.179]

[68] V. Legout, M. Jan, and L. Pautet, "A scheduling algorithm to reduce the static energy consumption of multiprocessor real-time systems," in *Proceedings of the 21st International conference on Real-Time Networks and Systems*. ACM, 2013, pp. 99–108. [Pg.31], [Pg.32]

[69] V. Legout, M. Jan, and L. Pautet, "Scheduling algorithms to reduce the static energy consumption of real-time systems," *Real-Time Systems*, vol. 51, no. 2, pp. 153–191, March 2015. [Pg.31], [Pg.32]

[70] J. Leung, "A new algorithm for scheduling periodic, real-time tasks," *Algorithmica*, vol. 4, no. 1, pp. 209–219, 1989. [Pg.25]

[71] G. Levin *et al.*, "Dp-fair: A simple model for understanding optimal multiprocessor scheduling," in *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*. IEEE, 2010, pp. 3–13. [Pg.27], [Pg.28], [Pg.175]

[72] G. Levitin, L. Xing, H. Ben-Haim, and Y. Dai, "Effect of failure propagation on cold vs. hot standby tradeoff in heterogeneous 1-out-of-n: G systems," *IEEE Transactions on Reliability*, vol. 64, no. 1, pp. 410–419, 2015. [Pg.34]

[73] J. Lieslehto, J. T. Tanttu, and H. N. Koivo, "An expert system for multivariable controller design," *Automatica*, vol. 29, no. 4, pp. 953–968, 1993. [Pg.139]

[74] Y. Lin and W. Zhang, "Towards a novel interface design framework: function–behavior–state paradigm," *International journal of human-computer studies*, vol. 61, no. 3, pp. 259–297, 2004. [Pg.139]

[75] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973. [Pg.22], [Pg.23], [Pg.24]

[76] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973. [Pg.1], [Pg.29], [Pg.31]

[77] J. Liu, *Real-Time Systems*. Prentice Hall, 2000. [Pg.21], [Pg.22], [Pg.23], [Pg.24]

[78] Z. Liu, X. Yin, Z. Zhang, D. Chen, and W. Chen, "Online rotor mixed fault diagnosis way based on spectrum analysis of instantaneous power in squirrel cage induction motors," *IEEE Transactions on Energy Conversion*, vol. 19, no. 3, pp. 485–490, 2004. [Pg.139]

[79] J. Malkevitch, "Bin packing and machine scheduling," *Feature Column from the AMS: Monthly Essays on Mathematical Topics*, Jun 2004. [Online]. Available: http://www.ams.org/featurecolumn/archive/packings2.html [Pg.22], [Pg.26]

[80] G. Manimaran and C. S. R. Murthy, "A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 11, pp. 1137–1152, 1998. [Pg.33]

[81] A. Mannani and P. Gohari, "A framework for synthesis of communicating decentralised supervisors for discrete-event systems," *International Journal of Systems Science*, vol. 45, no. 5, pp. 947–969, 2014. [Pg.140]

[82] R. McNaughton, "Scheduling with deadlines and loss functions," *Management Science*, vol. 6, no. 1, pp. 1–12, 1959. [Pg.92], [Pg.94]

[83] P. Mejía-Alvarez and D. Mossé, "A responsiveness approach for scheduling fault recovery in real-time systems," in *Real-Time Technology and Applications Symposium, 1999. Proceedings of the Fifth IEEE.* IEEE, 1999, pp. 4–13. [Pg.20]

[84] R. Melhem, D. Mossé, and E. Elnozahy, "The interplay of power management and fault recovery in real-time systems," *IEEE Transactions on Computers*, vol. 53, no. 2, pp. 217–231, 2004. [Pg.4]

[85] V. Moghaddas, M. Fazeli, and A. Patooghy, "Reliability-oriented scheduling for static-priority real-time tasks in standby-sparing systems," *Microprocessors and Microsystems*, vol. 45, pp. 208 – 215, 2016. [Pg.38]

[86] A. K. Mok, "Fundamental design problems of distributed systems for the hard-real-time environment," Cambridge, MA, USA, Tech. Rep., 1983. [Pg.24]

[87] M. H. Mottaghi and H. R. Zarandi, "Dfts: A dynamic fault-tolerant scheduling for real-time tasks in multicore processors," *Microprocessors and Microsystems*, vol. 38, no. 1, pp. 88–97, 2014. [Pg.37], [Pg.106]

[88] W. Munawar, H. Khdr, S. Pagani, M. Shafique, J.-J. Chen, and J. Henkel, "Peak power management for scheduling real-time tasks on heterogeneous many-core systems," in *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS).* IEEE, 2014, pp. 200–209. [Pg.180]

[89] B. V. Naik, S. Das, and H. K. Kapoor, "Rt-dvs for power optimization in multiprocessor real-time systems," in *2014 International Conference on Information Technology.* IEEE, 2014, pp. 24–29. [Pg.3], [Pg.30], [Pg.179]

[90] G. Nelissen, "Efficient optimal multiprocessor scheduling algorithms for real-time systems," Ph.D. dissertation, Université libre de Bruxelles, 2012. [Pg.16]

[91] A. Paoli and S. Lafortune, "Safe diagnosability for fault-tolerant supervision of discrete-event systems," *Automatica*, vol. 41, no. 8, pp. 1335–1347, 2005. [Pg.47]

[92] A. Paoli, M. Sartini, and S. Lafortune, "Active fault tolerant control of discrete event systems using online diagnostics," *Automatica*, vol. 47, no. 4, pp. 639–649, 2011. [Pg.47]

[93] R. M. Pathan, *Scheduling algorithms for fault-tolerant real-time systems*, 2010. [Pg.20]

[94] R. M. Pathan, "Real-time scheduling algorithm for safety-critical systems on faulty multicore environments," *Real-Time Systems*, vol. 53, no. 1, pp. 45–81, 2017. [Pg.2], [Pg.33], [Pg.34], [Pg.35]

[95] R. M. Pathan and J. Jonsson, "Ftgs: Fault-tolerant fixed-priority scheduling on multiprocessors," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*. IEEE, 2011, pp. 1164–1175. [Pg.2], [Pg.33], [Pg.34], [Pg.35]

[96] M. Pinedo, *Scheduling*. Springer, 2012. [Pg.20], [Pg.181]

[97] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, "Power-performance modeling on asymmetric multi-cores," in *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*. IEEE, 2013, pp. 1–10. [Pg.130]

[98] J. Rasmussen and K. J. Vicente, "Coping with human errors through system design: implications for ecological interface design," *International Journal of Man-Machine Studies*, vol. 31, no. 5, pp. 517–534, 1989. [Pg.139]

[99] M. Roth, S. Schneider, J.-J. Lesage, and L. Litz, "Fault detection and isolation in manufacturing systems with an identified discrete event model," *International Journal of Systems Science*, vol. 43, no. 10, pp. 1826–1841, 2012. [Pg.47]

[100] A. Roy, H. Aydin, and D. Zhu, "Energy-aware standby-sparing on heterogeneous multicore systems," in *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*. IEEE, 2017, pp. 1–6. [Pg.xviii], [Pg.5], [Pg.39], [Pg.119], [Pg.120], [Pg.121], [Pg.126], [Pg.127], [Pg.129], [Pg.130], [Pg.136], [Pg.137]

[101] A. Roy, H. Aydin, and D. Zhu, "Energy-efficient primary/backup scheduling techniques for heterogeneous multicore systems," in *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*. IEEE, 2017, pp. 1–8. [Pg.5], [Pg.119], [Pg.130]

[102] R. Samet, "Recovery device for real-time dual-redundant computer systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 3, pp. 391–403, May 2011. [Pg.2]

[103] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis, "Diagnosability of discrete-event systems," *IEEE Transactions on automatic control*, vol. 40, no. 9, pp. 1555–1575, 1995. [Pg.5], [Pg.6], [Pg.16], [Pg.42], [Pg.44], [Pg.47], [Pg.48], [Pg.49], [Pg.50], [Pg.51], [Pg.52], [Pg.140], [Pg.158], [Pg.160]

[104] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. C. Teneketzis, "Failure diagnosis using discrete-event models," *Control Systems Technology, IEEE Transactions on*, vol. 4, no. 2, pp. 105–124, 1996. [Pg.39], [Pg.42], [Pg.44], [Pg.47], [Pg.139], [Pg.140], [Pg.161], [Pg.162], [Pg.166]

[105] K. Schmidt, "Abstraction-based failure diagnosis for discrete event systems," *Systems & Control Letters*, vol. 59, no. 1, pp. 42–47, 2010. [Pg.48], [Pg.50], [Pg.51]

[106] K. W. Schmidt, "Verification of modular diagnosability with local specifications for discrete-event systems," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 43, no. 5, pp. 1130–1140, 2013. [Pg.48], [Pg.51], [Pg.54]

[107] A. Srinivasan, P. Holman, and J. Anderson, "The case for fair multiprocessor scheduling," in *Proceedings of the 11th International Workshop on Parallel and Distributed Real-time Systems, Nice, France*, Apr 2003. [Pg.22], [Pg.26], [Pg.27], [Pg.31]

[108] D. Suleiman, M. Ibrahim, and I. Hamarash, "Dynamic voltage frequency scaling (dvfs) for microprocessors power and energy reduction." [Pg.3], [Pg.30], [Pg.179]

[109] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE transactions on parallel and distributed systems*, vol. 13, no. 3, pp. 260–274, 2002. [Pg.181]

[110] P. Wang and K.-Y. Cai, "Supervisory control of discrete event systems with state-dependent controllability," *International Journal of Systems Science*, vol. 40, no. 4, pp. 357–366, 2009. [Pg.140]

[111] W. Wang and P. Mishra, "Leakage-aware energy minimization using dynamic voltage scaling and cache reconfiguration in real-time systems," in *2010 23rd International Conference on VLSI Design*. IEEE, 2010, pp. 357–362. [Pg.30]

[112] Y. wen Zhang, "Energy-aware mixed partitioning scheduling in standby-sparing systems," *Computer Standards and Interfaces*, vol. 61, pp. 129 – 136, 2019. [Pg.38]

[113] L. Xing, O. Tannous, and J. B. Dugan, "Reliability analysis of nonrepairable cold-standby systems using sequential binary decision diagrams," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, vol. 42, no. 3, pp. 715–726, 2012. [Pg.34]

[114] T.-S. Yoo and S. Lafortune, "Polynomial-time verification of diagnosability of partially observed discrete-event systems," *IEEE Transactions on automatic control*, vol. 47, no. 9, pp. 1491–1495, 2002. [Pg.48], [Pg.49]

[115] S. H. Zad, R. H. Kwong, and W. M. Wonham, "Fault diagnosis in discrete-event systems: Framework and model reduction," *IEEE Transactions on Automatic Control*, vol. 48, no. 7, pp. 1199–1212, 2003. [Pg.5], [Pg.6], [Pg.16], [Pg.39], [Pg.43], [Pg.44], [Pg.47], [Pg.48], [Pg.50], [Pg.51], [Pg.54], [Pg.139], [Pg.140], [Pg.149], [Pg.160], [Pg.161], [Pg.162], [Pg.166], [Pg.167]

[116] J. Zaytoon and S. Lafortune, "Overview of fault diagnosis methods for discrete event systems," *Annual Reviews in Control*, vol. 37, no. 2, pp. 308–320, 2013. [Pg.47]

[117] B. Zhao, H. Aydin, and D. Zhu, "Shared recovery for energy efficiency and reliability enhancements in real-time applications with precedence constraints," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 18, no. 2, p. 23, 2013. [Pg.120]

[118] D. Zhu and H. Aydin, "Reliability-aware energy management for periodic real-time tasks," *IEEE Transactions on Computers*, vol. 58, no. 10, pp. 1382–1397, 2009. [Pg.4]

[119] D. Zhu, X. Qi, D. Mossé, and R. Melhem, "An optimal boundary fair scheduling algorithm for multiprocessor real-time systems," *Journal of Parallel and Distributed Computing*, vol. 71, no. 10, pp. 1411–1425, 2011. [Pg.79], [Pg.84]