6-26-2015

Energy-aware Fault-tolerant Scheduling for Hard Real-time Systems

Qiushi Han
Florida International University, qhan001@fiu.edu

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

ENERGY-AWARE FAULT-TOLERANT SCHEDULING FOR HARD REAL-TIME SYSTEMS

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

ELECTRICAL ENGINEERING

by

Qiushi Han

2015

To: Dean Amir Mirmiran
College of Engineering and Computing

This dissertation, written by Qiushi Han, and entitled Energy-aware Fault-tolerant Scheduling for Hard Real-time Systems, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Kang K. Yen

Jean H. Andrian

Nezih Pala

Deng Pan

Gang Quan, Major Professor

Date of Defense: June 26, 2015

The dissertation of Qiushi Han is approved.

Dean Amir Mirmiran
College of Engineering and Computing

Dean Lakshmi N. Reddi
University Graduate School

Florida International University, 2015

DEDICATION

I would like to dedicate this Doctoral dissertation to my beloved wife, Lu Shen, my sister, Xiao Han and my dearest mom, Yaqin Guo. Without their love, understanding, support, and encouragement, the completion of this endeavor would never have been possible.

ACKNOWLEDGMENTS

First, I would like to express my heartfelt appreciation to my major advisor, Dr. Gang Quan, for his constant guidance and endless encouragement during the last six years of my doctoral study. I truly admire his dedication to science and research. In addition, I would also like to express my gratitude to my Ph.D. committee members, Dr. Kang K. Yen, Dr. Jean H. Andrian, Dr. Nezih Pala and Dr. Deng Pan, for their insightful feedbacak, comments and suggestions in improving the quality of this dissertation. I am extremely proud to have such wonderful and knowledgeable people serving on my dissertation committee.

Next, I would like to thank my lab mates, Mr. Ming Fan, Mr. Shuo Liu, Mr. Tianyi Wang, Mr. Shi Sha, Mr. Soamar Homsi, Mr. Gustavo A.Chaparro-Baquero, Dr. Vivek Chaturvedi, Dr. Huang Huang and Dr. Guanglei Liu, for creating a wonderfully collaborative and friendly work environment.

Last, but not least, my deepest gratitude goes to my family for their constant love and support during this journey. I am very grateful to my beloved wife, Lu Shen, for accompanying and encouraging me through all these years. I want to give my life-long gratitude to my dearest sister, Ms. Xiao Han, and my mother, Ms. Yaqin Guo, for all the love and affection they have showered upon me. I am thankful to my mother-in-law, Mrs. Yunhui Luo, and farther-in-law, Mr. Qing Shen, for their care and encouragement.

My Ph.D. research was supported in part by US National Science Foundation (NSF) grants CNS-0969013, CNS-0917021 and CNS-1018108.

ABSTRACT OF THE DISSERTATION

ENERGY-AWARE FAULT-TOLERANT SCHEDULING FOR HARD REAL-TIME SYSTEMS

by

Qiushi Han

Florida International University, 2015

Miami, Florida

Professor Gang Quan, Major Professor

Over the past several decades, we have experienced tremendous growth of real-time systems in both scale and complexity. This progress is made possible largely due to advancements in semiconductor technology that have enabled the continuous scaling and massive integration of transistors on a single chip. In the meantime, however, the relentless transistor scaling and integration have dramatically increased the power consumption and degraded the system reliability substantially. Traditional real-time scheduling techniques with the sole emphasis on guaranteeing timing constraints have become insufficient.

In this research, we studied the problem of how to develop advanced scheduling methods on hard real-time systems that are subject to multiple design constraints, in particular, timing, energy consumption, and reliability constraints. To this end, we first investigated the energy minimization problem with fault-tolerance requirements for dynamic-priority based hard real-time tasks on a single-core processor. Three scheduling algorithms have been developed to judiciously make tradeoffs between fault tolerance and energy reduction since both design objectives usually conflict with each other. We then shifted our research focus from single-core platforms to multi-core platforms as the latter are becoming mainstream. Specifically, we launched our research in fault-tolerant multi-core scheduling for fixed-priority tasks as fixed-priority scheduling is one of the most commonly used schemes in the industry today. For such systems, we developed several checkpointing-based partitioning strategies with the joint consideration of fault tolerance and energy minimization. At last, we exploited the implicit relations between real-time tasks in order to judiciously make partitioning decisions with the aim of improving system schedulability.

According to the simulation results, our design strategies have been shown to be very promising for emerging systems and applications where timeliness, fault-tolerance, and energy reduction need to be simultaneously addressed.

TABLE OF CONTENTS

CHAPTER PAGE

1. INTRODUCTION 1
1.1 Real-time systems 2
1.2 Power/Energy management in real-time systems 5
1.3 Fault tolerance/reliability in real-time systems 7
1.4 The research problem and our contributions 11
1.5 Structure of the dissertation 13

2. BACKGROUND AND RELATED WORK 14
2.1 Real-time scheduling 14
2.1.1 Preliminaries of real-time scheduling 14
2.1.2 Related works on real-time scheduling 18
2.2 Energy-efficient real-time scheduling 20
2.2.1 Preliminaries on power consumption in real-time systems 20
2.2.2 Related works on energy management in real-time systems 21
2.3 Fault-tolerant real-time scheduling 23
2.3.1 Energy-oblivious fault-tolerant techniques 24
2.3.2 Energy-aware fault-tolerant techniques 26
2.4 Summary 29

3. ENERGY EFFICIENT FAULT-TOLERANT EARLIEST DEADLINE FIRST SCHEDULING FOR HARD REAL-TIME SYSTEMS 30
3.1 Related works 31
3.2 Preliminaries 32
3.2.1 Real-time application model 33
3.2.2 Power and energy model 33
3.2.3 Fault model 34
3.2.4 Problem formulation 35
3.3 Fault-tolerant speed schedule 35
3.4 Fault-tolerant speed schedule with shared recovery slacks 44
3.5 Other considerations of the proposed methods 54
3.5.1 Dealing with the limitations of practical processors 55
3.5.2 System reliability and imperfect fault coverage 56
3.6 Simulation results 57
3.6.1 System with continuous speeds 58
3.6.2 System with discrete speed levels 61
3.6.3 Real-life periodic task sets 62
3.6.4 Further validation of LPSSR 63
3.7 Summary 65

4. ENERGY MINIMIZATION FOR FAULT TOLERANT REAL-TIME APPLICATIONS ON MULTI-CORE PLATFORMS USING CHECKPOINTING . . . 66
4.1 Related works . . . 67
4.2 Preliminaries . . . 68
4.2.1 Application model . . . 68
4.2.2 Fault model and checkpointing . . . 69
4.2.3 Platform and energy model . . . 69
4.3 Optimal checkpointing scheme for minimizing the worst case latency on a single core . . . 71
4.4 Energy-aware fault-tolerant task allocation . . . 76
4.5 Experimental results . . . 78
4.6 Summary . . . 82

5. ENERGY MINIMIZATION FOR FAULT-TOLERANT SCHEDULING OF PERIODIC FIXED-PRIORITY APPLICATION ON MULTI-CORE PLATFORMS . . . 83
5.1 Related works . . . 83
5.2 Preliminaries . . . 85
5.2.1 Application model . . . 85
5.2.2 Fault model and checkpointing . . . 85
5.2.3 Platform and energy model . . . 87
5.3 Feasible checkpointing configuration for fixed-priority tasks on a single-core processor . . . 88
5.4 Energy-aware task allocation . . . 95
5.5 Experimental results . . . 98
5.5.1 Timing complexity evaluation . . . 98
5.5.2 Energy performance evaluation . . . 100
5.6 Summary . . . 103

6. ENHANCED FIXED-PRIORITY FAULT-TOLERANT SCHEDULING OF HARD REAL-TIME TASKS ON MULTI-CORE PLATFORMS . . . 105
6.1 Related work . . . 106
6.2 Preliminaries . . . 108
6.2.1 Application and system model . . . 108
6.2.2 Fault-tolerance/reliability requirement . . . 109
6.2.3 Problem formulation . . . 110
6.2.4 Motivation example . . . 110
6.3 Fault-tolerant schedulability analysis for fixed-priority task sets . . . 112
6.4 Compatibility index and its properties . . . 116
6.5 Fault-tolerant task partitioning . . . 119
6.6 Task set with checkpointing . . . 122
6.7 Simulation results . . . 125

6.7.1 Experiment 1, acceptance ratio vs. system average utilization. 126
6.7.2 Experiment 2, acceptance ratio vs. the number of faults. 128
6.7.3 Experiment 3, acceptance ratio vs. checkpointing 129
6.8 Summary and future directions 131

7. CONCLUSIONS AND FUTURE WORK 133
7.1 Summary 133
7.2 Future work 135
7.2.1 Lifetime and fault model 136
7.2.2 Preliminary results 140

BIBLIOGRAPHY 148

VITA 162

LIST OF FIGURES

FIGURE PAGE

1.1 Embedded system market [117] 2

1.2 Demand for multi-core based devices 4

1.3 Transistor count from 1971-2011 [125] 5

1.4 Power consumption for portable and stationary devices 6

1.5 Soft Error Rate, FIT: faults in time (a billion hour operation) 8

3.1 MLPEDF vs. EMLPEDF. K is set to 1, a dark grey rectangle represents a reserved recovery block and a shaded rectangle indicates that a recovery block becomes active, i.e. a fault has been encountered. Figure 3.1(a) and 3.1(b) show the schedules when the fault affects the job with the longest execution time, i.e.J_2 under MLPEDF and EMLPEDF, respectively. The reserved recovery blocks are not shown in the fault-free schedules. 38

3.2 Monotonicity violation example 39

3.3 EMLPEDF vs. LPSSR 44

3.4 An example of LPSSR 49

3.5 (a)d'_i is deadline to be assigned after the removal of critical interval, which is $t_s + RS(J_i)$, t_i is the finishing time of J_i or its recoveries. (b) t^* is the completion time of all the jobs and recoveries in $\mathcal{J}(I^*)$, a'_i is extended into I^* by $RS(J_i)$. 50

3.6 Energy savings with different numbers of jobs, K = 1 59

3.7 Energy savings with increasing number of faults, # of jobs = 15 60

3.8 Energy savings with increasing number of jobs under PentiumM, K=1, . 61

3.9 Energy savings with increasing number of faults under PentiumM, # of jobs = 15 61

3.10 LPSSR vs FTUniCk 64

4.1 Upper and lower bounds of $L(\Gamma, SR)$ 75

4.2 20 tasks on a 4-core processor, K = 1 80

4.3 40 tasks on a 8-core processor, K = 2 80

4.4 80 tasks on a 16-core processor, K = 4 81

4.5 Performance of two speed-up techniques 81

5.1 Varying the number of tasks 99

5.2 Varying the number of tasks 99

5.3 Varying checkpoint overhead 100

5.4 40 tasks on 4-core processors, K = 2 102

5.5 80 tasks on 8-core processors, K = 5 102

5.6 160 tasks on 16-core processors, K = 10 103

6.1 Task partition based on HAPS. Task τ_2 misses deadline under the worst case. 111

6.2 An alternative partition, all tasks are schedulable under the worst case. 111

6.3 32 tasks on 4-core platform, K=2. 126

6.4 64 tasks on 8-core platform, K=2. 127

6.5 32 tasks on 4-core platform, system average utilization is 0.5. 128

6.6 32 tasks on 4-core platform, checkpoint overhead is 5 percent of execution time, K=2. 129

7.1 Simulation framework 141

7.2 MTTF VS. Temperature 142

7.3 Speed Schedule 143

7.4 Thermal profiles 144

7.5 Reliability distribution for core 1(2) 145

7.6 Impacts of m-oscillation on system reliability 146

7.7 MTTF of TC vs. the number of oscillations 146

CHAPTER 1

INTRODUCTION

For the past several decades, we have experienced tremendous growth of real-time systems and applications largely due to the remarkable advancements of IC technology. From simple electronic devices such as cell phones, to large and complex systems such as ICU patient monitoring systems, Unmanned Aerial Vehicles (UAV), industry controls, etc, real-time systems have become indispensable to our personal and social lives. However, as transistor scaling and massive integration continue, the dramatically increased power/energy consumption and degraded reliability of IC chips have posed significant challenges to the design of real-time systems. Power/energy management on computing systems has already been one of the primary concerns in both academia and industry for several decades [19, 37]. At the same time, the impacts of system failures become more and more substantial, ranging from personal inconvenience, disruption of our daily lives, to some catastrophic consequences such as huge financial loss. For example, Knight Capital lost an estimated of $400 million and almost fell to the edge of bankruptcy due to a computer glitch in less than one hour in 2013. Conceivably, guaranteeing the reliability of computing systems has also been raised to a first-class design concern. Left unchecked, the high power/energy consumption and deteriorating reliability of IC chips will handicap the availability of future generations of real-time computing systems.

Our research focuses on studying and developing effective and efficient resource-management schemes that address the constraints of power consumption and reliability in the design of real-time systems. In what follows, we first introduce the basics of real-time systems. We then discuss the opportunities and challenges associated with the design of real-time systems with power consumption and reliability constraints. Next, we introduce the contributions of our research for energy minimization and reliability enhancement. We also discuss the organization of the dissertation at the end of this chapter.

1.1 Real-time systems

Real-time systems refer to computing systems that are subject to "real-time" constraints where the correctness of an output depends on not only its logic correctness but when the output is produced. The requirement of real-time capability is pervasive in embedded systems, which account for a large portion of modern computing systems. According to a study in [3], the embedded system market was valued at $121 billion in 2011, and is predicted to reach $194 billion by 2018. As shown in Figure 1.1, more than half (61%) of all the embedded systems are equipped with real-time capability. This feature is manifested by associating tasks (workloads) with deadlines in such systems.

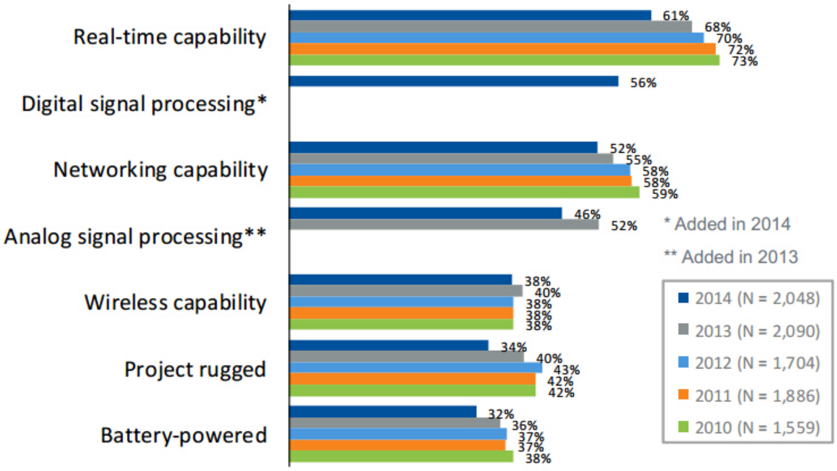


Figure 1.1: Embedded system market [117]

In general, real-time systems can be broadly classified into two types, i.e. *hard real-time systems* and *soft real-time systems*, according to consequences of missing a deadline. Hard real-time systems have very stringent timing constraints. Deadline misses in such systems can potentially lead to catastrophic consequences such as an automatic train fails to stop in time. On the contrary, soft real-time systems can tolerate certain deadline misses, with degraded quality of service (Qos) of a system [105]. Examples of such systems include media streaming in distributed systems and non-mission-critical tasks in control systems. In this research, we focus on hard real-time systems as such systems are safety-critical in nature and therefore, demand higher reliability.

In order to guarantee the timeliness of hard real-time embedded systems, *real-time scheduling* that determines the order of real-time task executions and manages the resource allocations has been extensively studied in the literature over the past several decades. [87, 105, 24]. The research on real-time scheduling can be categorized along different dimensions, such as static/dynamic, periodic/aperiodic, priority driven/non-priority driven and single-core processor/multi-core processor, and many scheduling algorithms have been introduced. For example, for a set of periodic real-time tasks executed on a single-core processor, Rate Monotonic Scheduling (RMS) and Earliest Deadline First (EDF) scheduling have been identified as the optimal scheduling policies for static and dynamic priority based scheduling algorithms, respectively [86].

While there has been significant real-time scheduling research based on single-core platforms, there are growing interests in studying the real-time scheduling problem on multi-core platforms. Nowadays, there is an increasing number of real-time embedded systems that are adopting multi-core processors as the underlying architecture for higher performance, reliability, and overall greater flexibility of operations [84]. To keep pace with the demands for increasing processor performance, silicon vendors no longer concentrate wholly on increasing the clock frequency of

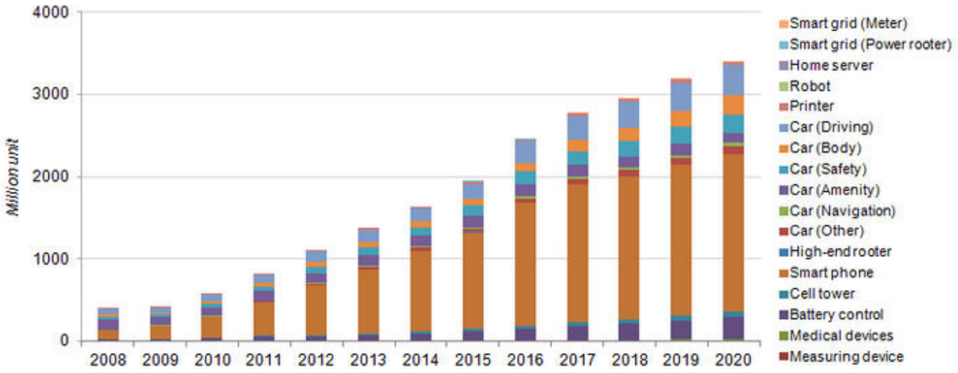


Figure 1.2: Demand for multi-core based devices

a single-core platform, as this approach leads to excessive power consumption and heat dissipation [7]. Instead, multi-core platforms have attracted more attention and become mainstream in the industrial market. Since 2007, many chip manufactures, e.g. AMD and Intel, have been releasing their new multi-core chips into the market with increasing number of cores, e.g Intel Xeon Series [69]. The demand of multi-core processors for various real-time embedded systems is illustrated in Figure 1.2 [2]. Since 2012, there has been an annual increase of 40% in the number of delivered multi-core processors. As computing paradigms shift towards multi-core processors, there is a growing need to develop appropriate multi-core real-time scheduling algorithms to efficiently utilize system resources in order to guarantee timing constraints for hard real-time systems.

The real-time scheduling problem on multi-core platforms is a challenging one. Different from real-time scheduling on single-core platforms, multi-core real-time scheduling needs to decide not only when but where to execute real-time tasks. The real-time scheduling on multi-core systems with only the timing constraints has been identified as a NP-hard problem [105].

In addition, as transistor miniaturization and mass transistor integration continue, they present unprecedented challenges to researchers, i.e. soaring power consumption and significantly degraded reliability of modern processors, which makes the real-time scheduling problem even harder to study. In the following sections, we discuss these challenges in details.

1.2 Power/Energy management in real-time systems

Energy consumption has emerged as a critical design concern for computing systems. Following Moore's Law as illustrated in Figure 1.3, the number of transistors being integrated into a single chip approximately doubles every two years to keep providing desirable processor performance.

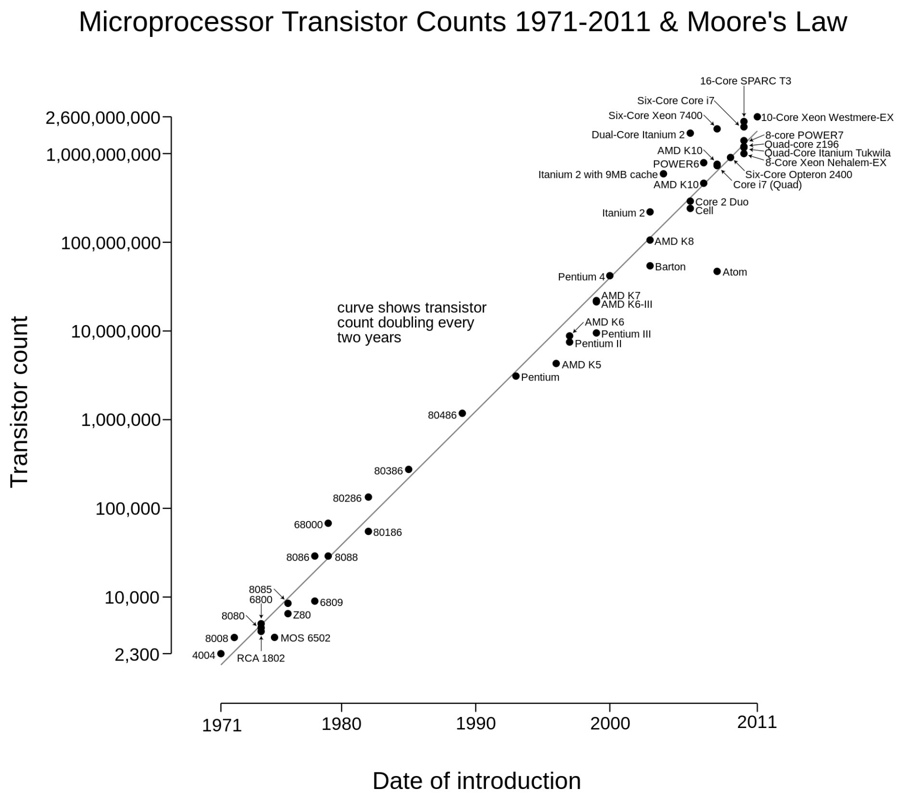
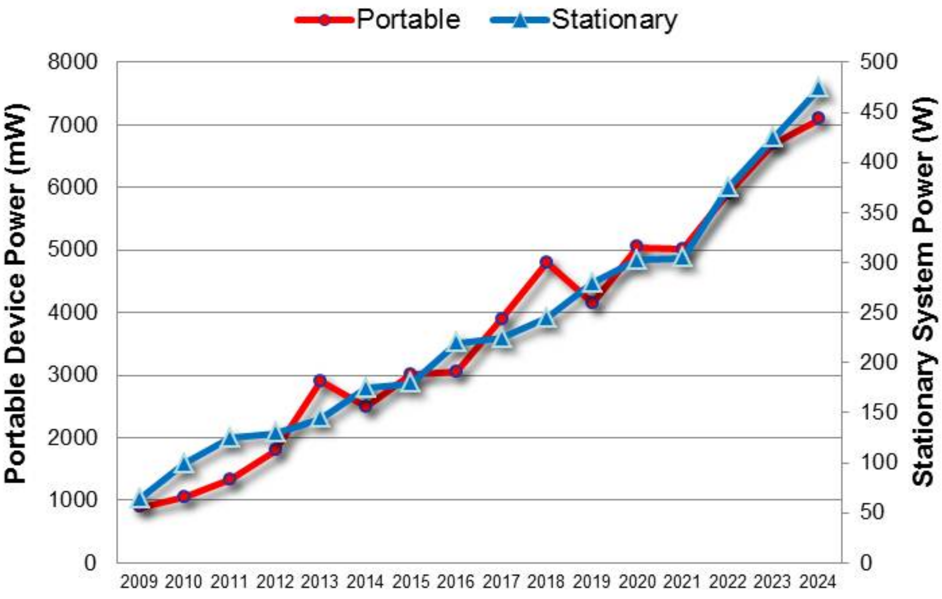


Figure 1.3: Transistor count from 1971-2011 [125]

However, one consequence of such progress is high power dissipation, which decreases the lifetime of battery-powered real-time systems, e.g. mobile phones, medical devices, [109, 129]. On the other hand, high power consumption dramatically increase the maintenance cost of large-scale computing systems such as data centers and server farms. As shown in Figure 1.4, the energy consumption of these systems has dramatically increased over the past few years and is predicted to keep increasing in the foreseeable future [36]. Even worse, the soaring power consumption has resulted in an ever-increasing chip temperature which adversely affects the performance, reliability, and packaging and cooling cost [66].



source ITRS 2010

Figure 1.4: Power consumption for portable and stationary devices

Therefore, it is imperative to develop efficient and effective power/energy management techniques for real-time systems while satisfying the timing constraints. For the past two decades, extensive power management techniques (e.g. [20, 21, 50]) have been developed on energy minimization for real-time systems. Among these techniques, dynamic voltage and frequency scaling (DVFS) is one of the most popular and widely deployed schemes. Most modern processors, if not all, are equipped with DVFS capabilities, such as Intel Xeon [69] and AMD G-series [5]. DVFS dynamically adjusts the supply voltage and working frequency of a processing core to reduce power consumption at the cost of extended circuit delay. Although there are a number of works in the literature that are focused on guaranteeing timing constraints while minimizing energy consumption for real-time systems [91, 102, 103, 74], they do not explicitly take system reliability into consideration, which makes them insufficient for systems that require both energy efficiency and high reliability.

1.3 Fault tolerance/reliability in real-time systems

A system fault occurs when a delivered service deviates from the desired service. In other words, a system fails when it cannot provide the desired service [82]. Even a perfectly designed computer system can be subject to different faults and therefore fail unpredictably. As shown in [115], processor faults can be broadly classified into two categories: *transient* and *permanent* faults. Transient faults, also termed *soft errors*, are often caused by electromagnetic interference and cosmic ray radiations. They may cause errors in computation and corruption in data, but are not persistent. On the other hand, permanent faults, also called *hard errors* can cause hardware damages to processors and bring them to halt permanently. Permanent faults can be further divided into extrinsic faults and intrinsic faults. The extrinsic faults occur due to process and manufacturing defects and the intrinsic faults are those related to wear-out [115]. According to [30] and [71], transient faults occur more frequently than permanent faults.

As real-time computing systems continue to grow rapidly in both scale and complexity, maintaining high reliability becomes an increasingly challenging issue. As semiconductor technology continues to scale, computing systems become less robust. The aggressive scaling in transistor size makes transistor more vulnerable to external impacts such as electromagnetic interference and cosmic ray radiations. According to [106], the soft error rate (SER) per chip of logic circuits increased nine orders of magnitude from 600nm to 50 nm technology. As the scaling process continues, it is predicted that there will be one failure per day per computer chip when the size of transistors shrinks to 16nm as shown in Figure 1.5 [78].

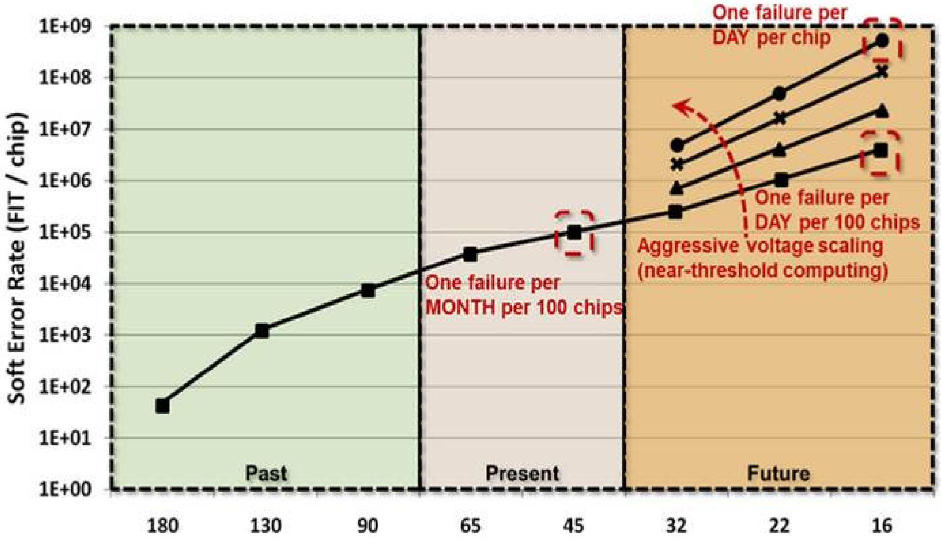


Figure 1.5: Soft Error Rate, FIT: faults in time (a billion hour operation)

Moreover, as more and more transistors are integrated into a single chip (Figure 1.3), the power consumption has been increased exponentially. One immediate consequence of high power consumption is the high operating temperature, which in turn poses severe threats on system reliability. As reported in [112], the maximum temperature reached by a 65nm processor is 15 degrees Kelvin higher than that reached by a 180nm processor, and the corresponding hard error rate is increased as much as 316%.

Faults in real-time systems that are not addressed properly in a timely fashion will lead to violations of timing constraints, which can cause catastrophical consequences if the systems are safety-critical, e.g. aircraft, nuclear power plant. Therefore, providing fault-tolerance features (the property that enables a system to continue operating properly in the event of failure(s)) to achieve high reliability is particularly sought after in such systems.

Traditional fault-tolerance techniques to deal with faults consist of two components, i.e. *fault detection* followed by *fault recovery* [123]. Examples of techniques that can detect the processor faults timely and effectively are listed below [123, 98]:

1. a fail-signal processor to send notifications to other processors when faults occur,
2. watchdog processors for concurrent control flow checking,
3. signatures that can be used for detection of hardware and software faults
4. sanity or consistence checks

To tolerate or recover from faults, many hardware/software replication techniques have been developed. For example, two task replication schemes have been proposed to support fault tolerance in multi-core systems [16].

- *Passive replication*- One or more backups of a task are assigned either to the same core or to a backup core. The backup is executed only when a fault occurs.
- *Active replication*- One or more independent active copies of a task run concurrently on different cores.

Instead of duplicating the execution of the entire program, checkpointing [131] in conjunction with backward error recovery is also a well-known fault-tolerant strategy. Checkpointing refers to the scheme that diagnoses system states after a period of time and stores a snapshot if no fault is detected. In case of a fault detection, the system rolls back to its pervious correct state. Note that checkpointing is a special passive replication scheme.

Active replication schemes usually require extra system resources, e.g, processing cores, and consume more energy even under the fault-free scenarios, but they can tolerate run-time faults timely and promptly. On the contrary, passive replications are only invoked in the event of run-time failure(s), and therefore, does not consume system resources when no faults occur. However, passive replications take longer to recover from faults and put the system at risk when timing constraints are very stringent. The selection of the appropriate replication schemes for various hard real-time systems is a design decision problem and requires careful investigations.

Conceivably, traditional techniques for ensuring the timing constraints for real-time systems without explicitly considering fault-tolerance requirements are becoming ineffective. It is imperative to explore advanced methodologies to ensure the timeliness in the presence of faults for real-time systems. Moreover, both fault tolerance and energy reduction are essentially achieved by exploiting system slack time, therefore they are two conflicting goals in nature. Even worse, DVFS has been shown to have adverse impacts on system reliability [107, 136, 135]. It is desirable that different constraints, i.e. timing, power, reliability, and their interplays be studied in a comprehensive and systematic way to achieve various design goals for different real-time systems. In what follows, we present our research problem in this dissertation and briefly summarize our contributions.

1.4 The research problem and our contributions

The objective of this research is to develop advanced fault-tolerant yet power efficient resource-management techniques for real-time computing systems. Researchers from both industry and academia have been studying this problem from different levels of abstraction, e.g. gate level, circuit level, architecture level, and system level. We endeavor to explore system-level methodologies and techniques in solving this problem. Specifically, we are interested in developing reliability-aware/fault-tolerant real-time scheduling to satisfy different design constants, i.e. reliability and timeliness, and in the meantime, to optimize different performance metrics such as energy consumption. To this end, we have made the following contributions.

1. First, we studied the problem of minimizing energy consumption while ensuring the timing constraints of fault-tolerant real-time tasks scheduled on a single-core platform. We developed several techniques for the co-management of energy reduction and fault tolerance. The goal is to utilize the least amount of system resources to tolerate transient faults and leave more space for energy minimization. Compared with the existing works, we found that our algorithm can achieve at least 13% energy reduction while guaranteeing that all task deadlines can be met under the worst case scenario.

2. Second, we investigated the energy minimization problem for fault-tolerant fixed-priority tasks scheduled on a multi-core platform. Real-time tasks with identical deadlines were considered. An efficient optimal checkpointing scheme was proposed for such tasks in order to minimize the overall schedule length in the presence of transient faults when they are executed on the same core. Then, we developed a novel task allocation algorithm that is based on this checkpointing scheme to judiciously make partitioning and checkpointing de-

cisions with the joint consideration of energy minimization and fault tolerance. Simulation results have shown that the proposed algorithm can outperform two related approaches by 11% and 50% in terms of energy savings, respectively.

3. Third, we further extended our research problem to more general fixed-priority tasks, i.e. tasks with arbitrary deadlines, and we explored solutions for effective checkpointing configuration and energy reduction that can tolerate transient faults with the least amount of energy consumption. A quick and accurate checkpointing algorithm was derived to determine if there exists a feasible checkpointing configuration for a set of tasks executed on the same processing core. It can achieve a speedup of two orders of magnitude over the state-of-art technique, therefore, it is more favorable to design space explorations. Moreover, we introduced a task partitioning approach in conjunction with the checkpointing algorithm to minimize energy consumption while ensuring the fault-tolerant capability of the system. The effectiveness of this approach has also been demonstrated using extensive simulations.

4. Finally, we explored the problem of mapping tasks to multi-core platforms with the focus on maximizing system schedulability in the presence of transient faults. By taking the task characteristics into consideration, we proposed a metric named "compatibility index" to measure how "compatible" a set of tasks are when they are mapped to the same core. Grouping tasks with lower compatibility index (more compatible) and assigning them to the same core are more likely to result in higher system utilization and better schedulability. We developed several techniques based on this concept, and they can at least improve the current approaches by 24% in terms of system schedulability.

1.5 Structure of the dissertation

The rest of this dissertation is organized as follows. In Chapter 2, we introduce the pertinent background to this dissertation and discuss existing works that are closely related to our research. In Chapter 3, we study the problem of fault-tolerant scheduling for dynamic-priority based real-time tasks on single-core systems to guarantee the timing constraints while minimizing the energy consumption. In Chapter 4, we focus our research on partitioning fixed-priority tasks with identical deadlines on multi-core platforms with the joint consideration of fault tolerance and energy reduction. We also propose an optimal checkpointing scheme and an efficient task allocation algorithm. In Chapter 5, we extend our research problem presented in Chapter 4 to more general fixed-priority tasks. We propose an efficient checkpointing scheme that can guarantee the schedulability of a set of real-time tasks in the presence of transient faults. A task partitioning approach to minimize energy consumption while ensuring the fault-tolerance capability of the system is presented. In Chapter 6, we investigate partitioning techniques for fixed-priority real-time tasks on multi-core platforms with a focus on maximizing system schedulability under the influence of transient faults. Finally, in Chapter 7, we conclude this dissertation and discuss possible future works.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, we introduce the pertinent research background and discuss the related works. Specifically, we present some important basics on real-time scheduling, energy-efficient scheduling, fault-tolerant scheduling, and energy-efficient fault-tolerant scheduling and discuss the existing works related to these topics, respectively.

2.1 Real-time scheduling

In this section, we first introduce some preliminaries of real-time scheduling and then we review the existing works with regard to this topic.

2.1.1 Preliminaries of real-time scheduling

As explained in Chapter 1.1, a real-time system is responsible for delivering logically correct computations within the predefined deadlines. A *task* is defined as a set of related computations that jointly provide some system functions, and a *job* is an invocation of a task. The violations of task deadlines in real-time systems can potentially lead to catastrophical consequences [87, 105]. To guarantee the timing constraints, real-time scheduling that primarily determines the resource allocation and management has been widely adopted as one of the most effective techniques. In general, real-time scheduling determines when, where, and how to execute a set of real-time tasks such that all deadlines can be met and other design metrics, e.g. power consumption and reliability can be optimized.

Real-time scheduling can be classified into various categories from different perspectives. According to the stringency of task deadlines, real-time scheduling can be categorized into *hard real-time scheduling* and *soft real-time scheduling* [28, 27]. Based on job arrival patterns, it can be classified into *periodic* and *aperiodic* [61, 110]. From the perspective of scheduling mechanisms, it can be further characterized as *static* or *dynamic*, *priority driven* or *non-priority driven* and *preemptive* or *non-preemptive* [24, 40]. Finally, real-time scheduling can be categorized as *single-core* and *multi-core* scheduling according to the underlying system architectures [57]. In the following, we discuss the real-time scheduling from several categories in details.

Hard real-time scheduling vs. Soft real-time scheduling.

Hard real-time scheduling focuses on providing deterministic guarantees to all task deadlines since a deadline miss may have catastrophic consequences. Typical hard real-time scheduling can be found in avionic systems, industrial control systems, etc. On the contrary, soft real-time scheduling allows a certain degree of deadline misses, where the effects on normal operations of a system will not be fatal, but degrade the delivered quality of service. In this dissertation, we focus our efforts on hard real-time scheduling.

Periodic vs. Aperiodic.

In a real-time system, a task can be periodic or aperiodic. Specifically, periodic tasks, as their name implies, execute on a regular basis. A periodic task can potentially generate an infinite number of jobs. The jobs/instances of the same task are released following a certain pattern where two consecutive jobs are separated by a fixed length of inter-arrival time (period). A periodic task is represented by its worst case execution time (WCET), period and relative deadline. The utilization of a periodic task is determined by the ratio of its WCET over period. According to the relationships between relative deadline and period, periodic tasks are further characterized by *implicit deadline* (task deadlines are equal to their periods), *constrained deadline* (task deadlines are less than their periods) and *arbitrary deadlines* (tasks deadlines can be arbitrary), respectively. There are a large variety of real-times systems that are concerned with real-time scheduling for periodic tasks, such as the Inertial Navigation System (INS) in [76] and the Generic Avionic Platform (GAP) in [41]. In contrast, an aperiodic task is a single invocation of computation. The two terms, i.e. job and task, are considered equivalent in this case. An aperiodic task is usually characterized by its arrival time, worst case execution time and relative deadline. For example, Anti-lock Braking System (ABS) in modern cars is a typical system that employs aperiodic real-time scheduling.

Static vs. Dynamic

For static scheduling, the schedules for each task need to be determined in advance, therefore it requires prior knowledge of the characteristics of tasks. It only incurs little runtime overhead. In contrast, dynamic scheduling calculates the schedules during runtime, hence it can provide more flexibility to react to uncertainties of task characteristics at the cost of large runtime overhead. As deterministic guarantees for timing constraints are of critical importance in hard real-time systems, whether dynamic scheduling is suitable for such systems is highly debated [40].

Priority-Driven vs. Non-Priority-Driven

One of the critical problems in real-time scheduling is in what order should the tasks be executed. One method is to assign tasks with different priorities, and a higher priority task is favored over a lower priority tasks when they are competing for system resources, e.g. CPU. Additionally, priority-driven scheduling can be further characterized as *fixed-priority* and *dynamic-priority* scheduling according to the priority assignment policy for real-time tasks. For fixed-priority scheduling, all the jobs from a task share the same priority and maintain it during their lifetime. Rate Monotonic Scheduling (RMS) [86, 44, 73, 43] is a popular fixed-priority scheduling method for periodic real-time tasks where task priorities are directly related to the periods. The larger the period is, the lower the priority is. It has been proven in [86] that RMS is an optimal scheduling policy for fixed-priority tasks on a single-core processor.

By comparison, instead of statically assigning priorities to real-time tasks, dynamic scheduling determines the priority for each job during runtime. Potentially, jobs from the same tasks can have various priorities. Among all dynamic scheduling techniques, Earliest Deadline First (EDF) has attracted a lot of researchers' attention [86, 128, 57, 26] and it has been proven to be the optimal dynamic scheduling algorithm for hard real-time tasks on a single-core platform.

On the other hand, in non-priority driven scheduling, the order of task executions is determined by other criterias. For instance, Round Robin scheduling assigns a fixed amount of computation time to each task and cycles through them. The behavior of such scheduling is hard to predict. Therefore, it is not appropriate for hard real-time systems.

Preemptive vs. Non-Preemptive:

In preemptive scheduling, the execution of a job can be suspended (most likely by a higher priority job) and restarted later, without affecting the behavior of that job other than its completion time. On the contrary, non-preemptive scheduling does not has this feature; once a job starts executing, it continues until completion [29].

Single-Core vs. Multi-Core

Real-time scheduling can be categorized into single-core scheduling [86] and multi-core scheduling [105], on the basis of the underlying system architecture. Different from single-core scheduling, multi-core scheduling needs to decide not only when but where a task should be executed. Multi-core scheduling, known as a NP-hard problem [105], is more complicated than single-core scheduling.

2.1.2 Related works on real-time scheduling

The real-time scheduling has been studied for decades, and a plethora of techniques has been proposed for various task and system models.

The primary focus of real-time scheduling is to provide deterministic guarantees to timing constraints in hard real-time systems through schedulability analysis. One efficient way is to study the utilization bound (least achievable utilization) [86] of a system such that the system is deemed to be schedulable if this bound is never exceeded. For single-core platforms, there exist a number of techniques for improving the utilization bound and achieving more accurate schedulability analysis for periodic tasks scheduled under RMS policy in preemptive systems [86, 80, 81, 53, 87]. Since the utilization bound is only the sufficient condition to determine system schedulability, exact timing analysis has been conducted by [85, 83, 118] for fixed-priority preemptive scheduling of periodic real-time tasks. Similar problems has been investigated for dynamic-priority (e.g. EDF) tasks in [14, 47, 128]. All the aforementioned approaches are restricted to single-core processors.

As multi-core platforms are becoming mainstream, multi-core scheduling has attracted more and more researchers' attention lately. Multi-core scheduling can be broadly classified into *partitioned scheduling* and *global scheduling* [40]. In partitioned multi-core scheduling, each task is allocated to a core and all of its jobs have to be executed on that core, i.e. no migration is permitted. On the contrary, in global scheduling, the jobs of a task can be executed on any available cores. A new paradigm named *semi-partitioned scheduling*, which is a combination of the two previous concepts and allows a certain degree of migrations, has recently emerged in multi-core scheduling [77, 45]. Further, multi-core scheduling can be classified as *homogenous* and *heterogenous* according to the characteristics of the underlying multi-core systems. In homogenous systems, all processing cores are identical in terms of processing speed, power/thermal characteristics, and so forth. By comparison, the cores in a heterogenous system can vary widely. This feature further complicates the multi-core scheduling problem.

The multi-core scheduling essentially solves two problems, 1) task/job allocation; 2)the order of task/job execution on each core which is mostly determined by priority assignment. There are a great number of literatures targeting on these two problems.

First, the partitioning scheme is well studied, and various techniques for improving system schedulability have been proposed. For fixed-priority (e.g. RMS) periodic tasks scheduled on multi-core platforms, different allocation schemes such as traditional Bin-packing approaches, i.e. First Fit (FF), Best Fit (BF), and Worst Fit (WF) have been evaluated in [87], and how the ordering of tasks can affect the task-allocation results is investigated in [94]. Later, the characteristics of real-times tasks were exploited to develop more effective task partitioning schemes in [23, 44, 43]. For example, as shown in [44], by grouping harmonic tasks into the same core, system schedulability can be greatly enhanced. On the other hand, partitioning of dynamic-priority periodic tasks on multi-core processors is explored in [26, 13, 10]. Simple heuristics such as BF, FF, and WF have been evaluated, and extensions to (variants of) these approaches are proposed. As shown in [10], ordering tasks in decreasing utilization fashion can significantly improve system schedulability.

Second, there is also a great number of literatures on global scheduling of both fixed-priority and dynamic-priority periodic hard real-time tasks [12, 11, 40]. A new schedulability test for global scheduling of fixed-priority tasks with arbitrary deadlines on identical multi-core processors has been proposed in [12]. Later, Baruah et al. [11] proposed a new global EDF schedulability test and presented some theoretical advantages of this test.

Finally, the effects of semi-partitioning on improving system schedulability are examined in [45, 73]. By allowing a limited number of tasks to be split and assigned to different cores, the utilization bound of the system is increased, and hence the system schedulability can be improved.

All these works predominately focus on guaranteeing the timing constraints for hard real-time tasks. As discussed in Chapter 1, other design constraints, e.g energy consumption and relaibility are becoming increasingly critical in the design of real-time systems. In what follows, we introduce some important real-time scheduling techniques that explicitly account for these design constraints.

2.2 Energy-efficient real-time scheduling

In this section, we first present some preliminaries on energy-management methods in real-time systems, and then we review the existing works that are closely related to this topic.

2.2.1 Preliminaries on power consumption in real-time systems

Power consumption in computing systems mainly consists of two parts, namely *dynamic power* and *leakage power* [33]. The dynamic power is associated with the switching activities of the circuits and is also related to the supply voltage and frequency. To better understand the dependency of dynamic power consumption on these factors, the following power model is established in [33] and shown in equation (2.1).

$$P_{dyn} = CV^2 f, \tag{2.1}$$

where C is the switching capacitance, V and f are the supply voltage and frequency/speed respectively. Moreover, the frequency is usually linearly proportional to the supply voltage, i.e. $f \propto V$.

The leakage power, also termed as *static power*, is mainly incurred by electronic devices attached to the capacitors, such as transistors or diodes, which conduct a small amount of current (leakage current) even when they are turned off. The leakage current is inter-dependent with the chip temperature [31]. High power consumption leads to high temperature which in turn aggravates the power situation. The leakage power is formulated as

$$P_{leak} = N_{gate} V I_0 [A T^2 e^{\frac{\alpha V + \beta}{T}} + B e^{\gamma V + \delta}] \tag{2.2}$$

where T and V are the current temperature and supply voltage, respectively. N_{gate} is the number of gates in the circuit, and T_0 is the reference leakage current. A, B, α, β, and γ are technology dependent constants [119].

Traditionally, dynamic power consumption is the dominating factor in the overall power consumption of a system. However, as the semiconductor technology enters into the sub-micro domain, leakage power is becoming increasingly important. In the following section, we present the related works on energy management.

2.2.2 Related works on energy management in real-time systems

Researchers in both academia and industry have resorted to various techniques to minimize energy consumption in computing systems. Among these, Dynamic

Voltage and Frequency Scaling has emerged as one the most effective system-level techniques for energy reductions [9]. DVFS scheduling reduces the supply voltage and frequency when possible, therefore, its effects on conserving energy consumption are evident according to equation (2.1 and 2.2) where supply voltage and frequency directly affect the system energy consumption. However, one consequence of applying DVFS is the extended circuit delay which may undermine the schedulablity of a real-time system. As a result, a great number of techniques studying the problem of minimizing the energy consumption without jeopardizing the timing constraints on single-core platforms are proposed in the literature [9, 127, 131, 91, 102, 103, 63] for various task models. Yao et al. [127] developed a DVFS scheme for a set of aperiodic real-time tasks scheduled under EDF policy with a focus of minimizing dynamic power consumption. Similar problems for fixed-priority apeorodic/periodic real-time tasks were investigated in [91, 103]. As leakage power consumption is becoming prominent, Huang et al. [63] considered the temperate and leakage dependencies and proposed an efficient DVFS scheme to minimize the overall energy consumption while guaranteeing the timing constraints of a real-time system.

For multi-core systems, various techniques [15, 19, 18, 4], which exploit DVFS scheduling to minimize dynamic energy consumption, have also been developed. For example, AlEnawy et al. [4] studied the combination of task partitioning and DVFS scheme for real-time periodic tasks scheduled under RMS policy on homogenous multi-core platforms. A constant speed was determined for each core under a given partition result. They have shown that WF dominates other traditional bin-packing techniques in terms of dynamic energy saving. Different from single-core platforms, to judiciously minimize the overall system energy consumption with the consideration of temperature and leakage dependencies is extremely difficult. Therefore, pessimistic approximations of leakage power consumption using constant values are adopted by many researchers, and various DVFS-based heuristics are proposed in [62, 116, 68, 46]. However, these fault-oblivious approaches are becoming insufficient due to the fact that the reliability of computing systems are severely degraded. It is desirable to develop efficient and effective approaches that can provide the fault-tolerance feature. Next, we introduce the concept of fault-tolerance and elaborate on the existing works that are closely related to our research.

2.3 Fault-tolerant real-time scheduling

For a fault-tolerant system, fault detections accompanied by fault recoveries are usually required. Different software- and hardware- based fault-detection techniques have been developed, such as watchdog processors and sanity checks [98]. As for fault recovery, either space or time redundancy/backup is needed. Specifically, there are two major backup policies, namely active backup and passive backup. Under active-backup scheme, each task is replicated a number of times on different processing cores, and all the copies run concurrently. Run-time failures can be countered promptly and effectively, but extra system resources are consumed even under fault-free scenarios. By comparison, if a task is passively replicated, the backup copies can be assigned either to the same core or different cores, and they are only invoked when run-time faults are detected. This scheme can save system resources when the system is fault-free, however, it takes more time to recover from faults (a fault detected at the end of a job requires a re-execution of the entire job). A special case of passive backup that worths mentioning is checkpointing, where the status of a system is checked on a regular basis, and a checkpoint is inserted if no fault is detected or otherwise rollback to the latest saved checkpoint. Checkpointing has been shown to be very effective in reducing the recovery overhead at the cost of delaying the normal execution of a task/job, i.e. inserting checkpoints [130, 56].

In recent years, extensive studies have been done in improving the reliability of real-time systems through fault-tolerance, and many interesting techniques have been proposed. We categorize these work into the following categories: *energy-oblivious fault-tolerant techniques* and *energy-aware fault-tolerant techniques.*

2.3.1 Energy-oblivious fault-tolerant techniques

First, we discuss several advanced fault-tolerant techniques that are very effective in guaranteeing the schedulability of hard real-time systems in the present of faults, but they do not explicitly account for energy-consumption constraints. In what follows, we elaborate on energy-oblivious fault-tolerant techniques with regard to hard errors and soft errors, respectively.

A hard error occurs when a processing core loses its capability of computation permanently. Due to the nature of hard errors, to be able to maintain the schedulability of a system, the target platforms have to be multi-core systems. Many different replication methods were explored to make tradeoffs between fault tolerance and system resource usage, e.g. the number of cores required for a feasible schedule.

Bertossi et al. [16] proposed a fault-tolerant scheduling for periodic task sets. Both active and passive backups can be used. The objective is to reduce the number of cores required. However, only one permanent fault can be tolerated. For more general fault scenarios, Chen et al. [34] introduced several replication schemes to tolerate a fixed number of faults for periodic real-time tasks on homogenous multi-core systems. Two problems are studied in the paper. One is to minimize the maximum utilization in a system with a specified number of precessing cores. The other is to minimize the number of cores required for deriving a feasible schedule. In that work, only active backups are considered. Later on, two heuristics referred to R-BFD (Reliable Best-Fit Decreasing) and R-BATCH (Reliable Bin-packing Algorithm for Tasks with Cold standby and Hot standby) were introduced in [79]. The Cold standby and Hot standby are in fact the active backup and passive backup, respectively. The main idea is to reduce the number of required cores by utilizing the passive backups to the greatest extend.

Additionally, there have been significant research efforts on dealing with the soft errors. As mentioned before, soft errors occur more frequently than hard errors in modern computing systems. While soft errors can occur in both single-core and multi-core platforms, a majority of current researches are focused on single-core platforms [52, 39, 8, 89, 131] and only a few on multi-core platforms.

Han et al. [52] proposed a combined primary and backup scheme to tolerate at least one transient fault. The backup is assumed to be fault-free and of lower quality yield. The timing constraint is guaranteed by scheduling the backups with higher priority at the cost of quality loss. To study the schedulability under more general fault models, schedulability analysis for fixed-priority systems was extended to take fault recoveries into account [39]. In [131], the schedulability analysis for fixed-priority tasks with checkpoints was investigated, and an effective checkpointing scheme was proposed. Subsequently in [8], a dynamic programming approach was proposed to evaluate the feasibility of aperiodic task sets under preemptive Earliest Deadline First (EDF) scheduling given a fault-tolerance constraint, i.e maximum K-fault.

For multi-core systems, Pop et al. [96] proposed a more comprehensive approach to the synthesis of fault tolerant schedule for applications on heterogeneous distributed systems. They used the combination of checkpointing and active replication to deal with the fault-tolerance problem. A meta-heuristic (Tabu search) was constructed to decide the fault-tolerance policy, the placement of checkpoints, and the mapping of tasks to processing cores with the aim of minimizing the overall schedule length. Similar analysis was conducted in [64] where only the passive-backup scheme was employed. In [108], a process-level redundancy was exploited to tolerate transient faults where fault detection, coverage, and tolerance were carefully studied on a practical platform.

All these works are either computationally inhibitive (meta-heuristic based approaches) or limited by optimistic simplifications in terms of task or fault model. Moreover, they do not consider energy consumption as a design constraint, which makes them insufficient for energy-constrained real-time systems.

2.3.2 Energy-aware fault-tolerant techniques

In this section, we present the research efforts on scheduling techniques with the joint consideration of energy efficiency and fault tolerance. To enable the system to tolerate hard errors while minimizing its energy consumption, a popular concept termed *standby sparing* [58] has been proposed in the literature. The main idea is to replicate the entire schedule on a primary core to backup core(s), and the execution of tasks on backup core(s) is delayed as much as possible. A number of techniques employing this mechanism have been developed for various task and system models [42, 58, 59, 49]. For example, Haque et al. employed the standby sparing technique on a dual-core platform where the real-time tasks on primary core are scheduled under EDF policy whereas the tasks on the backup core are scheduled according to Earliest Deadline Latest policy (a policy where the execution of a task/job is delayed as much as possible without violating the timing constraints). Due to the fact that the probability of an error occurrence is relatively low, it can potentially save system energy consumption. Note that, these techniques can also be used to tolerant soft errors.

Since soft errors are more common in computing systems, most of researches related to fault tolerance are focusing on soft errors. In [136], Zhu et al. proposed a linear and an exponential model to capture the effects of dynamic voltage frequency scaling (DVFS) on transient fault rate. They showed that energy management through DVFS can reduce the system reliability. Based on this model, they proposed a recovery scheme to schedule an recovery for the each scaled job to compensate the reliability loss caused by DVFS in [135]. Later on, an enhanced approach was developed in [132] to further reduce energy consumption by reserving only one share recovery block and leaving more space for DVFS. In 2010, Liu et al. proposed a heuristic scheme that minimizes the energy consumption when no fault occurs and preserves feasibility under the worst case of fault occurrences, i.e. up to K fault occur during an operational cycle of the system [88]. These works suffer a common drawback that all the approaches can only be applied to frame-based task sets, i.e. all tasks share the same deadline.

To guarantee the reliability of fixed-priority real-time tasks, Zhang et al.[130] introduced a combination of checkpointing and DVS scheme for tolerating faults for periodic task sets while minimizing energy consumption. The author used exhaustive search to find the optimal speed assignment, which is computationally impractical for large task set with a considerable amount of frequencies available on the processor. Melhem et al. [90] investigated the same problem for periodic task sets scheduled under EDF on a single-core processor with the restriction that there is at most one failure. Wei et al. [124] further extended the approach in [130] for the development of combined offline and online DVFS schedules. These techniques are only applicable to single-core platforms.

For multi-core platforms, there are only a few number of techniques presented in the literature. Pop et al. [97] presented a constraint logic programming method to develop fault-tolerant DVFS schedules for real-time tasks with precedence constraints on distributed heterogeneous platforms. The task allocation is assumed to be known a priori. Considering the negative effects of DVFS on system reliability, Guo et al [48] studied the similar problem and proposed several heuristics to minimize system energy consumption while maintaining system reliability. Qi et al. [100] investigated global scheduling in conjunction with energy management, i.e. DVFS, for a set of frame-based real-time tasks running on a homogeneous multi-core system. Additionally, the standby-sparing techniques in [42, 58, 59, 49] can also address the combinatorial problem of energy minimization and fault tolerance to transient faults (reliability guarantee). However, all these techniques rely on replicating the entire execution of a task to enable the fault-tolerance capability. As shown in [90, 130], checkpointing scheme can reduce the fault-recovery overhead significantly at the cost of runtime overhead, i.e. inserting checkpoints, which may potentially improve system schedulability and leave more space for energy management. However, if not carefully studied, checkpointing may undermine system schedulability due to its run-time overhead. Therefore, the problem of how to make judicious checkpointing decisions together with other design techniques in multi-core systems, e.g. task allocation and DVFS deployment, is yet to be studied.

2.4 Summary

In this section, we present the essential pertinent of our research and review some closely related works in the literature. We first introduce the basic concepts and different types of real-time scheduling. Existing researches on real-time scheduling for various task and system models are discussed. Then, we present some preliminaries on power management in real-time system and particularly introduce an effective system-level power-management technique, i.e. DVFS. We discuss the related works on employing DVFS in real-time scheduling in details. Finally, we present the concept of fault tolerance in real-time systems and elaborate on the the related research in fault-tolerant real-time scheduling for distinct task and system models with different design emphasis, e.g. timing and power. Based on the above discussions, we can see that fault-tolerant scheduling under various constraints still poses a grand challenge for researchers. Studying the interplay of different design constraints in a comprehensive and systematic way is becoming more and more critical.

In this dissertation, the goal of our research is to develop effective and efficient scheduling methods for hard real-time systems to provide deterministic guarantees of timing constraints under transient faults and also to optimize other design objectives, e.g. power consumption and the number of required processing cores. In the following chapters, i.e. Chapter 3, 4, 5 and 6, we present our contributions on this subject. We then conclude this dissertation in Chapter 7.

CHAPTER 3

ENERGY EFFICIENT FAULT-TOLERANT EARLIEST DEADLINE FIRST SCHEDULING FOR HARD REAL-TIME SYSTEMS

We first present our research on energy-efficient fault-tolerant scheduling on single-core platforms. Specifically, in this chapter, we focus on EDF-scheduled tasks with hard real-time constraints that are subject to a fixed number of transient faults, K. We adopt DVFS mechanism as our power management technique. As energy reduction and fault tolerance are two conflicting goals, the challenge is how to make the tradeoff between these two objectives such that the timing constraints can be guaranteed when no more than K faults occur while the system energy consumption is minimized. This is particularly important in the design of systems, such as surveillance and satellite systems, that demand both energy efficiency and fault tolerance.

In this regard, three scheduling algorithms are presented in this chapter. The first algorithm is an extension of a well-known fault oblivious low-power scheduling algorithm. The second algorithm intends to minimize the energy consumption under the fault-free situation while reserving adequate resources for recovery when faults strike. The third algorithm improves upon the first two by sharing the reserved resources and thus can achieve better energy efficiency.

The rest of this chapter is organized as follows. Section 3.1 discussed the works related to our research problem. In Section 3.2, we introduce system models and formally formulate our research problem. Our three algorithms are presented in Sections 3.3 and 3.4. Section 3.5 extends our algorithms to deal with several practical issues. Section 3.6 discusses our simulation results. Finally, we summarize this chapter in Section 3.7.

3.1 Related works

Recently, the problem to address energy conservation with reliability improvement has drawn considerable attention from many researchers.

When considering the reliability requirement, one approach is to formulate the reliability of a real-time system analytically. For example, Zhu et al. [136] formulated the reliability of a real-time system as the probability to complete executions of all tasks, with or without fault occurrences. They also proposed a linear and an exponential model to capture the effects of DVFS on transient fault rate and showed that energy management through DVFS could reduce the system reliability. Based on this model, they proposed a recovery scheme to schedule real-time tasks that can reduce energy consumption without degrading the reliability. They further proposed to reserve computing resources that can be shared by different tasks to improve the energy-saving performance [133]. These algorithms work only for frame-based real-time systems, i.e. tasks with same arrival times and deadlines. Zhao et al. [134] considered a more general real-time periodic task model. Different tasks may have different periods. For each task, its deadline is equal to its period. Algorithms were proposed to determine the processor speed and resource reservation for each task to achieve the goal of energy minimization under the task-level reliability requirement. The advantage of this approach is that the reliability can be quantified and the impacts of DVFS to reliability can also be taken into consideration. However, to precisely identify the parameters for the reliability model can be challenging, especially when faults usually occur in a burst manner [89].

Another more intuitive approach is to require that a system can still function properly as long as fault occurrences do not exceed a predefined number. For example, Zhang et al. [130] introduced a combination of checkpointing and DVFS scheme for tolerating K faults for periodic task sets while minimizing energy consumption. To guarantee the timing constraints, they incorporated the worst case fault recovery time into fixed-priority exact timing analysis to obtain the worst case response time, based on which the energy efficient schedule is determined. Melhem et al. [90] investigated the same problem for periodic task sets scheduled under EDF on a single processor, assuming $K = 1$. Wei et al. [124] further extended the approach in [130] for the development of combined offline and online DVFS schedules. Since the probability of fault occurrence can be very small, the energy saving performance of the proposed algorithm can be limited. Liu et al. [88] proposed a heuristic scheduling algorithm that minimizes the energy consumption under the fault-free scenarios and preserves feasibility under the worst case fault occurrences, i.e. up to K faults occur during an operational cycle of the system. This algorithm can only be applied for frame-based real-time task sets. For frame-based real-time task sets, reserved computing resources can be readily shared by different jobs. However, if jobs have different priorities and deadlines, to share the reserved resources becomes much more challenging.

We are interested in developing scheduling techniques to minimize the energy consumption and enhance the reliability of a real-time system. In what follows, we first present the pertinent background of our research and introduce some necessary notations used throughout this chapter.

3.2 Preliminaries

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

3.2.1 Real-time application model

We model a real-time system as a job set $\mathcal{J} = \{J_1, J_2, ..., J_n\}$, where J_i denotes the i_{th} job in a job set and is characterized by a tuple (a_i, c_i, d_i). The definition of theses parameters is given in the following:

- a_i: the time when J_i is ready for execution, referred to as arrival time;
- c_i: the worst case execution time of J_i under s_{max}, where s_{max} is the maximum speed that the processor supports;
- d_i: the absolute deadline of J_i.

This model is rather general and can be readily extended to other real-time models such as the general periodic task model. All jobs are considered to be independent and scheduled under preemptive EDF policy on a single processor.

3.2.2 Power and energy model

For ease of our presentation, we assume the speed/frequency (two terms are used interchangeably throughout this chapter) of a processor can be changed continuously in $[s_{min}, s_{max}]$ with $0 \leq s_{min} \leq s_{max} = 1$. Later in this chapter, we extend our algorithms to processors supporting only a set of discrete levels of processor speed. A job is assumed to execute with only one speed. Therefore, when J_i is executed under speed s_i, the execution time of J_i becomes $\frac{c_i}{s_i}$. A speed schedule for an entire job set is denoted as $S = \{s_1, s_2, s_3, ..., s_n\}$ where s_i is the speed for J_i.

Our system-level power model is similar to that in [133] by distinguishing the frequency-independent and frequency-dependent power components. Specifically, the overall power consumption (P) can be formulated as

$$P = P_{ind} + P_{dep} = P_{ind} + C_{ef}s^{\alpha} \tag{3.1}$$

where P_{ind} is the frequency-independent power, including the power consumed by off-chip devices such as main memory and external devices and constant leakage power. C_{ef} is the effective switching capacitance. α is a constant usually no smaller than 2. P_{dep} is the frequency-dependent active power, including the CPU power, and any power that depends on the processing speed s. Hence, the energy consumption of a job J_i running at the speed s_i can be expressed as:

$$E_i(s_i) = (P_{ind} + C_{ef}s_i^{\alpha}) \cdot \frac{c_i}{s_i} \tag{3.2}$$

As $E_i(s_i)$ is a convex function, the minimum system energy is achieved when s_i is as small as possible, provided it is larger than so-called *critical speed* (s_c) [136]. In this study, we assume that $s_{min} \geq s_c$.

3.2.3 Fault model

We assume that the system is subject to a maximum of K transient faults (e.g., bit flips in architectural registers or timing errors in CMOS circuit). Faults usually are detected at the end of each job J_i's execution using *acceptance* or *sanity tests* [99] and the timing and energy overhead for detection are denoted as TO_i and EO_i, respectively. Furthermore, we assume that the overheads of fault detections are not subject to frequency variations. There is no assumption regarding the occurrence pattern of faults, i.e. faults can occur anywhere at any time during an operational cycle of the system, multiple faults may hit a single job. A fault is tolerated by re-executing the affected job. Therefore, the maximum recovery overhead for job J_i executing at s_{max} under a single failure, denoted as R_i, is c_i, or $R_i = c_i$. When a fault happens during the execution of J_i, a recovery job that of the same deadline d_i is released. The recovery jobs are subject to preemption as well.

3.2.4 Problem formulation

We formulate our problem formally as follows:

Problem 3.2.1. *Given a real-time job set $\mathcal{J}$ scheduled under EDF on a single processor, find a speed schedule S for all the jobs in $\mathcal{J}$ (including the recoveries) such that the processor energy consumption is minimized without any deadline miss when no more than K faults occur.*

3.3 Fault-tolerant speed schedule

In this section, we introduce an approach to the development of a fault tolerant DVFS schedule for a hard real-time job set to reduce the energy consumption. The algorithm is developed based on LPEDF presented in [127]. To ease the presentation of our approach, we first introduce several definitions and then reiterate briefly the general idea of LPEDF.

Definition 3.3.1. *Given a real-time job set $\mathcal{J}$,*

- *$\mathcal{J}(I)$ denotes the set of jobs contained in the interval $I = [t_s, t_f]$, i.e. $\mathcal{J}(I) = \{J_i | t_s \leq a_i < d_i \leq t_f\}$;*
- *the **workload** $W(I)$ of an interval $I = [t_s, t_f]$ is the accumulated execution time of jobs completely contained in the interval, i.e $W(I) = \sum_{J_i \in \mathcal{J}(I)} c_i$;*
- *the **intensity** of interval I is defined as*

$$s(I) = \frac{W(I)}{L(I)}, \tag{3.3}$$

  where $L(I)$ is the length of interval I, i.e. $L(I) = t_f - t_s$;
- *the interval $I = [t_s, t_f]$ is called a **critical interval** if it has the highest intensity and t_s and t_f are the arrival time and the deadline of some job(s), correspondingly.*

- *the* ***fault-related overhead*** *of an interval I is denoted as $W_{ft}(I) = W_r(I) + W_{TO}(I)$, where $W_r(I)$ represents the reserved workload to be used for recovery in the worst case, i.e. $W_r(I) = K \times (R_x + TO_x)$ and x denotes the index of the job with the longest recovery time in $\mathcal{J}(I)$, i.e. $J_x = \{J_i | \max(R_i + TO_i), J_i \in J(I)\}$ and $W_{TO}(I)$ denotes the overhead imposed by fault detections from regular jobs, i.e. $W_{TO}(I) = \sum\limits_{J_i \in \mathcal{J}(I)} TO_i$.*

Given a real-time job set $\mathcal{J}$, LPEDF can be employed to minimize the energy consumption (assuming $s_{min} \geq s_c$) as follows [127] :

1. Step 1: Identify a critical interval $I = [t_s, t_f]$ using equation (3.3);
2. Step 2: Remove the critical interval and all jobs contained in the interval, set the speeds of all jobs in $\mathcal{J}(I)$ to $s(I)$ and modify the arrival times and deadlines of other jobs accordingly. Specifically, let $\mathcal{J} \leftarrow \mathcal{J} - \mathcal{J}(I)$; change deadline d_i to t_s if $d_i \in [t_s, t_f]$, or to $d_i - (t_f - t_s)$ if $d_i \geq t_f$; set a_i to t_s if $a_i \in [t_s, t_f]$, or to $a_i - (t_f - t_s)$ if $a_i \geq t_f$.
3. Step 3: Repeat step $1) - 2)$ until $\mathcal{J}$ is empty.

To make the above LPEDF fault-tolerant, one intuitive approach (we call this approach as **MLPEDF**) is to take the fault recovery into consideration and increase the workload of an interval when calculating its intensity, that is, to replace $s(I)$ with $s_m(I)$, as defined in equation (3.4),

$$s_m(I) = \frac{W(I) + K \times R_x}{L(I) - W_{TO}(I) - K \times TO_x}, \tag{3.4}$$

where x is the index of the job with longest recovery in $\mathcal{J}(I)$ and $W_{TO}(I)$ denotes the total fault-detection overheads for regular jobs as defined in Definition 3.3.1.

We summarize the feasibility condition of an arbitrary EDF-scheduled job set on a single processor that is subject to a maximum number of K transient faults in the following.

Theorem 3.3.2. *[8] Given a real-time job set $\mathcal{J}$ with K faults to be tolerated and $s_{max} = 1$, if for each interval I, we have*

$$\frac{W(I) + W_{ft}(I)}{L(I)} \leq 1, \tag{3.5}$$

then the job set $\mathcal{J}$ is feasible.

Note that, when a fault occurs, MPLEDF executes the recover copy of a job using a scaled processor speed. This helps to reduce the total energy consumption for both the original jobs and their recovery copies. However, this may not be energy efficient in a practical scenario when the possibility of fault occurrence is low.

An alternative approach (we call this approach as **EMLPEDF**), is to run the recovery backups using the maximum possible processor speed. The intensity calculation of interval I can be modified correspondingly, as equation (3.6).

$$s_e(I) = \frac{W(I)}{L(I) - W_{ft}(I)} \tag{3.6}$$

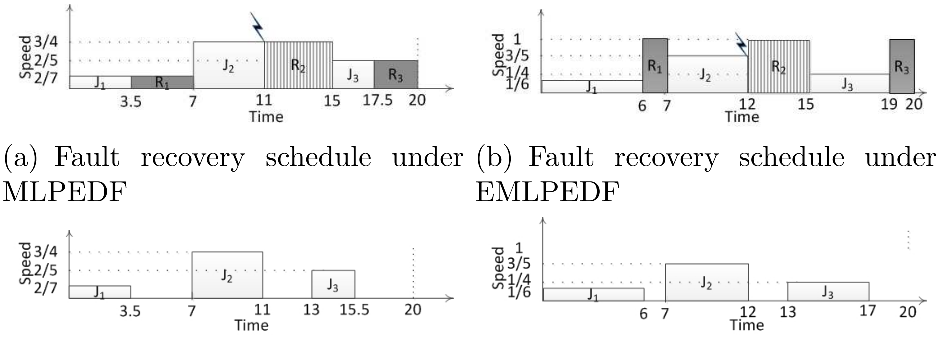
It can be easily verified that $s_e(I) \leq s_m(I)$ for a given interval I if $W(I) + W_{ft}(I) \leq L(I)$, which always holds for a feasible schedule. The advantage of this approach is that it requires the least amount of resource reservation to guarantee the timely recovery, and thus can reduce the energy consumption under the fault-free scenario. This can be further illustrated using the following example.

Consider the simple real-time job set, shown in Table 3.1 and Figure 3.1.

Table 3.1: A real-time system with three jobs

	a_i	c_i	d_i
J_1	0	1	9
J_2	7	3	15
J_3	13	1	20

Let $\alpha = 2$, $P_{ind} = 0.02$ and $C_{ef} = 1$. For simplicity, the timing and energy overhead are considered negligible. We can calculate that the fault recovery schedule by MLPEDF (Figure 3.1(a)) consumes less energy than that by EMLPEDF (Figure 3.1(b)), i.e. 5.41 vs. 5.56. However, the fault-free schedule by MLPEDF (Figure 3.1(c)) consumes much more energy than that by EMLPEDF (Figure 3.1(d)), i.e. 3.08 vs. 2.48 (20% more). Since the fault rate is usually very low in practice, EMLPEDF can have a much better energy saving performance than MLPEDF.



(a) Fault recovery schedule under MLPEDF

(b) Fault recovery schedule under EMLPEDF

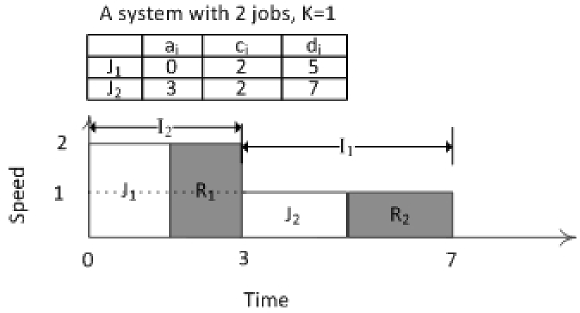
(c) Fault free schedule under MLPEDF

(d) Fault free schedule under EMLPEDF

Figure 3.1: MLPEDF vs. EMLPEDF. K is set to 1, a dark grey rectangle represents a reserved recovery block and a shaded rectangle indicates that a recovery block becomes active, i.e. a fault has been encountered. Figure 3.1(a) and 3.1(b) show the schedules when the fault affects the job with the longest execution time, i.e.J_2 under MLPEDF and EMLPEDF, respectively. The reserved recovery blocks are not shown in the fault-free schedules.

To ensure the deadlines, when removing a critical interval and updating the arrivals or deadlines of remaining jobs in each iteration (similar to each round of Step 1 and Step 2 in LPEDF), we assume that all K faults will affect the longest job in the critical interval under the worst case. This assumption is rather pessimistic because each critical interval demands computing resources reserved for tolerating K faults, which may potentially cause a feasible job set infeasible. We use an example to illustrate this problem.

Consider a system with two jobs specified in Figure 3.2 and at most one fault to be tolerated. For ease of presentation, we set the overheads of fault detections to 0. According to EMLPEDF, the first critical interval is interval [3,7] with intensity 1



A system with 2 jobs, K=1

	a_i	c_i	d_i
J_1	0	2	5
J_2	3	2	7

Figure 3.2: Monotonicity violation example

based on equation (3.6). After the removal of interval [3,7] along with job J_2, d_1 is updated as 3 and the second critical interval is [0,3] with intensity 2. We have the schedule drawn in Figure 3.2, where I_1 and I_2 denote the first and second critical interval, respectively. We can see that $s_e(I_2)$ is larger than $s_e(I_1)$(we refer to this situation as the **monotonicity violation**). Moreover, $s_e(I_2)$ exceeds the highest speed available in the system ($s_{max} = 1$), so the required speed is unachievable. However, it is not hard to see that the job set is in fact feasible under constant speed 1. From the above discussion, it is clear that the energy minimization problem with fault tolerance requirement cannot be solved by simply modifying the LPEDF solution. Provisions are required during the scheduling process to ensure that the resulting schedule is valid.

To handle monotonicity violations, we observed that any critical interval that violates monotonicity must be adjacent to the critical interval found in the previous iteration. Specifically, we have the following lemma.

Lemma 3.3.3. *Let I_i and I_{i-1} be two critical intervals identified by EMLPEDF from i_{th} and $(i-1)_{th}$ iteration*[1]*, respectively. If $s_e(I_i) > s_e(I_{i-1})$, I_i and I_{i-1} are adjacent.*

[1] Each iteration of EMLPEDF refers to one round of the Step 1-2 in LPEDF except the intensity function is defined in equation (3.6)

Proof. When removing interval I_{i-1}, the workload distribution is not changed in the intervals that have no overlap with I_{i-1}. Only the intervals overlapping I_{i-1} are shortened by Δ, $0 < \Delta \leq L(I_{i-1})$; therefore, they may experience an increase in intensity in the next iteration. □

As implied in the proof of Lemma 3.3.3, a monotonicity violation occurs when the removed critical interval contains slacks that need to be reserved as recoveries for jobs in its overlapping intervals. Therefore, the execution space for these jobs are shortened due to its removal. To eliminate such monotonicity violations, we can incorporate these jobs into the previously found critical interval. We formulate this conclusion in Lemma 3.3.4.

Lemma 3.3.4. *Let I_i and I_{i-1} be two critical intervals identified by EMLPEDF from i_{th} and $(i-1)_{th}$ iteration, respectively. If $s_e(I_i) > s_e(I_{i-1})$, the minimum constant speed to maintain feasibility of jobs contained in I_i and I_{i-1} is $s_e(I_{i-1})$.*

Proof. Before removing the critical interval I_{i-1} $(i > 1)$, all the remaining jobs in $\mathcal{J}$(jobs left after first $i-2$ iterations) are feasible under the constant speed $s_e(I_{i-1})$. Therefore, the combined jobs in I_i and I_{i-1} are definitely feasible under this speed. □

Lemma 3.3.3 and Lemma 3.3.4 help us to keep track of the monotonicity violation and remove it whenever it occurs.

Up to now, we can formulate our EMLPEDF algorithm in Algorithm 1. Line 4 identifies the current critical interval and its speed. Lines 5-8 check if the current desired speed is less than the minimal available speed, and terminate the iteration if so. Lines 9-12 remove monotonicity violation whenever it occurs. Line 14 backs up the timing information of jobs in case a rollback operation is needed. Lines 15-17 remove the critical interval and update the job set. The complexity of EMLPEDF mainly comes from calculations of critical intervals (line 4), i.e. $O(n^2)$ with a straightforward implementation. The overall complexity of EMLPEDF is same as LPEDF, i.e. $O(n^3)$.

Algorithm 1 EMLPEDF algorithm

```
Require:
    1) Job set : $\mathcal{J} = \{J_1, J_2, ...J_n\}$;
    2) Number of faults: $K$
    3) minimum frequency available: $s_{min}$
 1: $s_i = s_{max}$, for $i = 1, 2, ..., n$;
 2: $p = 1$; {the critical interval index}
 3: while $\mathcal{J} \neq \emptyset$ do
 4:   Identify the next critical interval $I_p^* = [t_s, t_f]$ and its intensity $s_{e,p}$ based on
      equation (3.6);
      {$s_{e,p}$: the intensity of $p_{th}$ critical interval}
 5:   if $s_{e,p} < s_{min}$ then
 6:     $s_i = s_{min}, \forall J_i \in \mathcal{J}$;
 7:     break;
 8:   end if
 9:   if $s_{e,p} > s_{e,p-1}$ AND $p > 1$ then
10:     Restore the timing information from the previous iteration;
11:     Merge the interval $I_p^*$ with $I_{p-1}^*$;
12:     $p - -$;{Roll back the critical interval index}
13:   end if
14:   Back up the timing information;
15:   $s_i = s_{e,p}, \forall J_i \in \mathcal{J}(I_p)$;
16:   remove all jobs in $I_p$ from $\mathcal{J}$;
17:   update timing information of remaining jobs according the step 2 in
      LPEDF[127]
18:   $p + +$;
19: end while
20: return {$s_1, s_2, ...,$ }
```

We have the following theorem regarding the lowest constant speed that guarantees the feasibility of a job set.

Theorem 3.3.5. *Let $s_{e1}, s_{e2}, s_{e3}, ...$ be the intensities for the critical intervals from iteration 1, 2, 3... in EMLPEDF. s_{e1} is the lowest constant speed that can be em-*

ployed throughout the entire job set without causing any deadline miss as long as no more than K faults happen.

Proof. This theorem can be proved directly in light of Theorem 3.3.2. During the first iteration of EMLPEDF, we have $\frac{W(I)}{s_{e1}} \leq \frac{W(I)}{s_e(I)}$ for each interval I, since $s_{e1} \geq s_e(I)$ considering the definition of critical interval. Take equation (3.6) into the right-hand side of the above inequality and add $W_{ft}(I)$ to both sides. We have $\frac{W(I)}{s_{e1}} + W_{ft}(I) \leq L(I)$. Therefore, the job set is feasible under constant speed s_{e1}.

Moreover, assume s_{e1} is the resulting intensity from interval I_1, i.e. $s_{e1} = \frac{W(I_1)}{L(I_1)-W_{ft}(I_1)}$ and s^* is the lowest constant speed that maintains the feasibility of the job set and $s^* < s_{e1}$. We have the scaled workload in I_1 as $\frac{W(I_1)}{s^*} + W_{ft}(I_1) > \frac{W(I_1)}{s_{e1}} + W_{ft}(I_1) = L(I_1)$, which violates the feasibility condition in Theorem 3.3.2. □

In addition, by applying Algorithm 1, we have the following theorem regarding the characteristics of critical interval speeds.

Theorem 3.3.6. *Let $s_{e1}, s_{e2}, ...s_{em}$ be the intensities for the critical intervals from iteration $1, 2, ...m$ in EMLPEDF. We have $s_{e1} \geq s_{e2} ... \geq s_{em}$.*

Proof. Because all monotonicity violations are eliminated in Algorithm 1, the non-increasing relationship between subsequent critical intervals can be easily determined. □

More importantly, if EMLPEDF can be successfully applied for a job set, then the feasibility of the result DVFS schedule is guaranteed. This is summarized in the following theorem.

Theorem 3.3.7. *EMLPEDF can guarantee that all jobs can meet their deadlines as long as the following two constraints are satisfied : (1) no more than K faults occur; (2) $\forall i \in [1, m]$, where m is the total number of iterations, we have $s_{ei} \leq 1$.*

Proof. In EMLPEDF, a critical interval I_i is exclusively reserved for executing jobs and their recovery copies in the interval. For any higher priority job (e.g. J_h) with possible execution overlapping with I_i, it is forced to finish before the I_i in EMLPEDF. Similarly, for any lower priority job (e.g. J_l) with possible execution overlapping with I_i, the interval I_i is excluded for its execution by adjusting its arrival time and deadline in EMLPEDF. Therefore, to prove the theorem, we only need to prove that if we set the processor speed to be s_{ei}, i.e. the intensity of I_i, throughout I_i, then the schedulability of all jobs in I_i is guaranteed in the worst case (i.e. against K faults), as long as $s_{ei} \leq 1$.

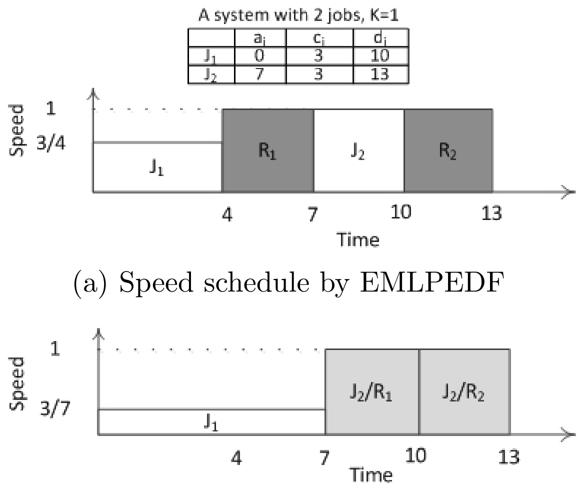
We prove this by contradiction. Let $J_c = (r_c, c_c, d_c) \in \mathcal{J}(I_i)$ miss its deadline when processor speed is set to s_{ei}. Then we must be able to find a time $t \leq r_c$, such that for interval $I' = [t, d_c]$, we have $\frac{W(I')}{s_{ei}} + W_{ft}(I') > L(I')$. Since $s' = \frac{W(I')}{L(I')-W_{ft}(I')} > s_{ei}$ and $I' \subseteq I_i$. This violates the assumption that I_i is a critical interval.

Since all jobs are associated with a critical interval in EMLPEDF and all jobs within a critical interval are schedulable when the corresponding speed is applied, we prove the theorem.

□

While EMLPEDF can guarantee the feasibility of a real-time job set under maximum K faults, and can also achieve better energy saving performance than MLPEDF, each critical interval needs to reserve computing resource separately for timely recovery when faults happen. It is desirable that different critical intervals can share the reserved resources and conceivably the energy saving performance can be further improved. We develop a new algorithm for this purpose, which is introduced next.

3.4 Fault-tolerant speed schedule with shared recovery slacks


A system with 2 jobs, K=1

	a_i	c_i	d_i
J_1	0	3	10
J_2	7	3	13

(a) Speed schedule by EMLPEDF

(b) Speed schedule by LPSSR

Figure 3.3: EMLPEDF vs. LPSSR

This section presents an improved approach to the development of energy efficient fault tolerant schedule for a given job set $\mathcal{J}$. We call this algorithm LPSSR. Specifically, LPSSR improves upon EMLPEDF by allowing different critical intervals to share reserved computing resources. We also execute recovery under s_{max} in LPSSR and focus on determining the speed schedule S for regular jobs. Before we introduce the algorithm in details, we first use an example to motivate our research.

Consider a simple job set with two jobs specified in Figure 3.3. Note that, we set the overheads of fault detection to 0 for easy presentation. The speed schedule by EMLPEDF is shown in Figure 3.3(a). Note that in Figure 3.3(a), interval R_1 (i.e. interval $[4, 7]$) and interval R_2 (i.e. interval $[10, 13]$) are the recovery blocks used for fault recovery. However, since $K = 1$, at most one of the recovery blocks can be used. If the fault occurs during J_1's execution, R_1 will be used for recovery. In that case, R_2 will never be used since no fault will happen during J_2's execution. Same problem occurs if the fault affects J_2's execution.

A better fault tolerant schedule is shown in Figure 3.3(b). Note that, when the fault affects J_1, the interval $[7, 10]$ can serve as the reserved block to run the backup of J_1, and J_2 can be executed at interval $[10, 13]$. If the fault affects J_2, since there is no fault during J_1's execution, J_2 can be executed at interval $[7, 10]$, and later recovered at interval $[10, 13]$ if necessary. For either case, the system is always feasible. By sharing the recovering slacks, the speed of J_1 is reduced to 3/7. Using the same system parameters as in the previous example, the energy consumption of the new schedule is more than 30% lower than that by EMLPEDF. The example clearly shows that significant energy savings can be obtained without compromising the system feasibility if the reserved computing resource can be shared. The problem is how to judiciously share the reserved resource to maximize the energy saving performance. In what follows, we develop an approach to explore the shared slacks to improve the energy efficiency.

When removing a critical interval in EMLPEDF, its reserved slacks can only be shared by jobs that have potential execution overlaps with it. To ease our presentation, we classify these jobs into the following categories as defined below.

Definition 3.4.1. *For a given interval $I = [t_s, t_f]$ a job J_i is referred to as **deadline overlapping** with I if $a_i \notin [t_s, t_f]$ and $d_i \in [t_s, t_f]$, and **arrival overlapping** with I if $a_i \in [t_s, t_f]$ and $d_i \notin [t_s, t_f]$, and **fully overlapping** with I if $I \subseteq [a_i, d_i]$.*

Specifically, for interval I, we denote all deadline overlapping jobs, arrival overlapping jobs, and fully overlapping jobs as $\mathcal{J}_I^{do}$ and $\mathcal{J}_I^{ao}$, and $\mathcal{J}_I^{fo}$, respectively.

In EMLPEDF, when a critical interval is identified, it is removed with all jobs inside it to make sure that the interval is exclusively used for running jobs and their backups that are completely located within the interval. Also, the arrival times and deadlines of the others are updated to the boundary of the interval such that their executions will never interfere with jobs in the critical interval. In LPSSR, we allow a job to share the reserved slacks in the critical interval by "extending" its deadline or arrival time "into" the critical interval. We discuss each category of jobs separately as follows. Let $I^* = [t_s, t_f]$ be a critical interval with length $L(I^*)$, and intensity $s_e(I^*)$, which is calculated the same way (i.e. equation (3.6)) as that in EMLPEDF. Let $R_{max}(I^*) = W_r(I^*)$, $R_{min}(I^*) = K \times \min\{R_j + TO_j | J_j \in \mathcal{J}(I^*)\}$ be the upper and lower bound of the reserved slacks. Also, let J_i be a job with execution interval (i.e. $[a_i, d_i]$) partially or fully overlapped with I^*, the overlap length is represented as $L(I_i^{op})$. Additionally, the maximum amount of reserved slack shared by a job J_i is denoted by $RS(J_i)$. Consider the following three cases:

- $J_i \in \mathcal{J}_{I^*}^{do}$: To share the reserved slack, the deadline of J_i will be *extended into* interval I^*. Specifically, instead of t_s, d_i is set to $t_s + RS(J_i)$ where $RS(J_i) = \min(K \times (R_i + TO_i), K \times R_{min}(I^*), L(I_i^{op}))$.
- $J_i \in \mathcal{J}_{I^*}^{ao}$: In this case, the new arrival time of J_i will be *extended into* interval I^* in order to share the reserved slack. To share the slacks, after removing the critical interval I^* (only subinterval $[t_s, t_f - RS(J_i)]$ is effectively removed, where $RS(J_i) = \min(K \times (R_i + TO_i), K \times R_{min}(I^*), L(I_i^{op})))$, we set J_i's deadline as $d_i = d_i - L(I^*) + RS(J_i)$, and update a_i to t_s.
- $J_i \in \mathcal{J}_{I^*}^{fo}$: In this case, all the reserved slacks in I^* can be potentially used by J_i. To share the slack, after removing the critical interval I^*, we set J_i's deadline as $d_i = d_i - L(I^*) + RS(J_i)$, where $RS(J_i) = \min(R_{max}(I^*), K \times (R_i + TO_i))$.

Accordingly, we formulate a new algorithm (i.e. LPSSR), as shown in Algorithm 2. Without loss of generality, we ignore the overheads of fault detections. The work flow of Algorithm 2 is similar to EMLPEDF, i.e. iteratively identifying

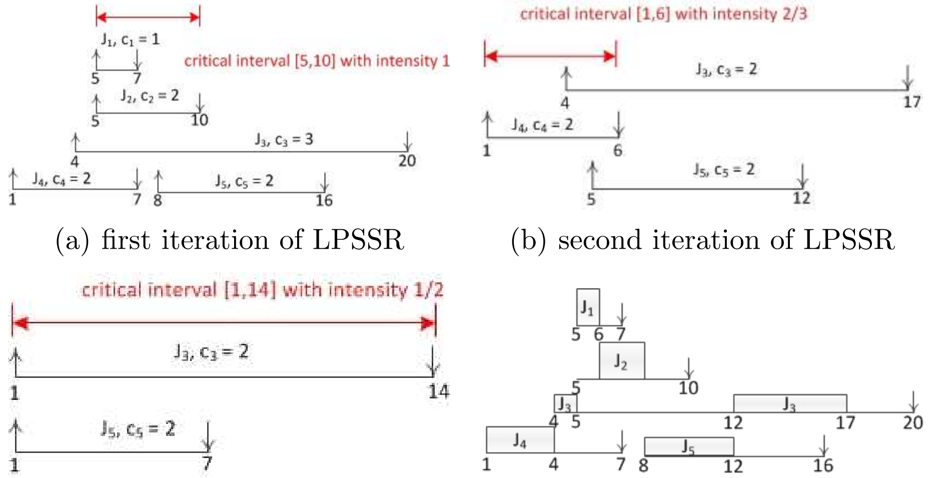
Algorithm 2 LPSSR algorithm

```
Require:
    1) Job set : $\mathcal{J} = \{J_1, J_2, ...J_n\}$;
    2) Number of faults: $K$
$s_i = s_{max}$, for $i = 1, 2, ..., n$;
$p = 1$;{critical interval index}
while $\mathcal{J} \neq \emptyset$ do
    Identify the critical interval $I_p = [t_s, t_f]$ and its intensity $s_{e,p}$ based on equation (3.6);
    $\{s_{e,p}$: the intensity of $p_{th}$ critical interval$\}$
    if $s_{e,p} < s_{min}$ then
        $s_i = s_{min}, \forall i \in \mathcal{J}$;
        break;
    end if
    if $s_{e,p} > s_{e,p-1}$ AND $p > 1$ then
        Restore the timing information from the previous iteration;
        Merge the interval $I_p$ with $I_{p-1}$;
        $p - -$;{Roll back the critical interval index}
    end if
    $L(I_p) = t_f - t_s$;
    for all $J_i \in \mathcal{J}$ do
        Backup timing information of $J_i$;
        $RS(J_i) = \min(K \times (R_i + TO_i), K \times R_{min}(I_p), L(I_i^{op}))$;//$R_{min}(I_p)$ is the minimum recovery time for jobs in $\mathcal{J}$
        if $J_i \in J_{I_p}^{do}$ then
            $d_i \leftarrow \min\{d_i, t_s + RS(J_i)\}$;
        else if $J_i \in J_{I_p}^{ao}$ then
            $d_i \leftarrow d_i - (L(I_p) - RS(J_i))$
            $a_i \leftarrow t_s$;
        else if $J_i \in J_{I_p}^{fo}$ then
            $RS(J_i) = \min(K \times (R_i + TO_i), K \times R_{max}(I_p))$;
            $d_i \leftarrow d_i - (L(I_p) - RS(J_i))$;
        else
            $a_i \leftarrow a_i - L(I_p)$;
            $d_i \leftarrow d_i - L(I_p)$;
        end if
        for all $J_q|[a_q, d_q] \subseteq I_p$ do
            $s_q = s_{e,p}$;
            $\mathcal{J} \leftarrow \mathcal{J} - \mathcal{J}(I_p)$;
        end for
    end for
    $p + +$;
end while
return $\{s_1, s_2, ...,\}$
```

critical intervals, removing the critical interval and the jobs inside the critical interval, and then updating the timing parameters for the rest of the jobs, until the job queue becomes empty. Different from EMLPEDF, we apply our sharing technique when updating the timing parameters and eliminate monotonicity violation whenever it occurs. In Algorithm 2, line 4 identifies p_{th} critical interval for the current real-time job set. Lines 5 to 8 are simply the application of Theorem 3.3.5. Lines 9 to 13 roll back to the previous iteration and merge the current critical interval with the previous one once monotonicity violation is found. Lines 16 to 29 backup and update the timing parameters of each remaining jobs according to the sharing technique discussed above. At last, lines 30 to 33 remove all jobs inside the critical interval.

The main computation complexity per iteration comes from identifying the critical interval, which is $O(n^2)$, where n is the number of jobs. The outer loop can at most repeat n times. Therefore, the complexity of Algorithm 2 is $O(n^3)$. In what follows, we first use an example to illustrate the procedures of LPSSR. We then prove that the algorithm can guarantee the schedulability of all jobs under K faults.

Consider a system with 5 jobs whose timing information is given in Figure 3.4(a). We assume that $K = 1$. We use $\uparrow$ and $\downarrow$ to denote a job's arrival time and deadline, respectively. The fault-detection overheads are considered negligible in this example. For each step, the critical interval is identified with intensity function in equation (3.6) and is shown as $| \leftrightarrow |$. For the first iteration in Figure 3.4(a), the critical interval is identified as [5,10] with intensity $s_e([5, 10]) = \frac{c_1+c_2}{10-5-R_2} = 1$. When we remove interval [5,10], J_1 and J_2 are removed and speed 1 is assigned to both jobs, and then we need to update the timing information of the remaining jobs.



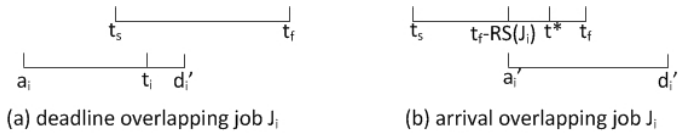
(a) first iteration of LPSSR (b) second iteration of LPSSR

(c) third iteration of LPSSR (d) fault-free schedule under the speeds from LPSSR

Figure 3.4: An example of LPSSR

Note that J_3 is a *fully overlapping* job with respect to the critical interval, and all the slacks that reserved in interval [5,10] can be used by J_3. Therefore, $RS(J_3) = \min(R_{max}[5,10], R_3) = \min(2,3) = 2$. Consequently, we have $d_3 = d_3 - L([0,5]) + RS(J_3) = 17$. For J_4, it is *deadline overlapping* with the critical interval and thus $RS(J_4) = \min(R_4, R_{min}([5,10]), L_i^{op}) = \min(2,1,2) = 1$. As a result, d_4 is set to 6, i.e. the boundary of the critical interval(5) plus the slacks that can be shared by J_4. For J_5, which is a *arrival overlapping* job with respect to the critical interval, the slacks that be shared by J_5 is $RS(J_5) = \min(R_5, R_{min}([5,10]), L_i^{op}) = \min(2,1,2) = 1$ and its arrival and deadline are set to 5 and 12, respectively. The resulting job set is illustrated in Figure 3.4(b).

Based on the new job set, we identify the critical interval as [0,6] with intensity 2/3. After the assign speed 2/3 to J_4 and remove the critical interval and repeat the same procedures as discussed in the previous iteration, we have a consequent job set as shown in Figure 3.4(c). Finally, the last critical interval is [1,14] with intensity 1/2 and J_3 and J_5 are removed after being allocated a speed 1/2. The LPSSR algorithm



(a) deadline overlapping job J_i

(b) arrival overlapping job J_i

Figure 3.5: (a)d_i' is deadline to be assigned after the removal of critical interval, which is $t_s + RS(J_i)$, t_i is the finishing time of J_i or its recoveries. (b) t^* is the completion time of all the jobs and recoveries in $\mathcal{J}(I^*)$, a_i' is extended into I^* by $RS(J_i)$.

terminates and we have the resulting speed schedule $S = \{1, 1, 1/2, 2/3, 1/2\}$. The final schedule is shown in Figure 3.4(d), and it can be verified that no matter when the failure occurs, there is no deadline miss with this schedule.

Moreover, the feasibility of the schedule output from Algorithm 2 is guaranteed, which is formulated in Theorem 3.4.2.

Theorem 3.4.2. *Given a real-time job set $\mathcal{J}$ and a constant K, all the jobs in $\mathcal{J}$ can meet their deadlines if they are executed based on the processor speeds determined by Algorithm 2 and no more than K faults occur.*

Proof. The proof of Theorem 3.4.2 is similar to that of Theorem 3.3.7. Let $I^* = [t_s, t_f]$ be the critical interval and $s_e(I^*)$ be its speed. We consider the three types of jobs separately.

Case 1: let $J_i \in \mathcal{J}_{I^*}^{do}$ and d_i' denote the deadline after the removal of the critical interval I^*, i.e, $d_i' = t_s + RS(J_i)$. If J_i and its recovery workload finishes at t_s or earlier, it has no impact to the execution for jobs in $\mathcal{J}(I^*)$. Hence all jobs in $\mathcal{J}(I^*)$ are schedulable under K faults in the worst case. If J_i and its recovery workload finishes at d_i', this means that all K faults must occur before d_i'. Otherwise, one more fault occurs at d_i' will cause J_i to miss deadlne. As a result, there will be no faults occurring in interval I^*. Since $d_i' - t_s = RS(J_i) \leq K \times R_{min}(I^*)$, this implies that the slack time occupied by J_i is smaller than the minimum amount of reserved slack in interval I^* that can be exploited by every job to execute the recovery workload. Therefore, all jobs in $\mathcal{J}(I^*)$ must be schedulable. The question now becomes what if J_i and its recoveries finishes at t_i, where $t_s < t_i < t_s + RS(J_i)$, refer to Figure 3.5-(a).

We consider the following two cases.

- Case 1-a: $R_i + TO_i >= R_{min}(I^*)$.

  Then there are at most K' faults, where $K' = \lfloor (d_i' - t_i)/(R_i + TO_i) \rfloor$ left after $t > t_i$. Otherwise, if more than K' faults occurring at (or after) t_i will cause J_i to miss its deadline. In other words, there must be $K - K'$ faults occurred before t_i. Note that J_i consumes a slack of $t_i - t_s$ from I^*. In the meantime, each job at least has an additional slack of $(K - K') \times R_{min}(I^*)$ to spare. Since $K \times R_{min}(I^*) \geq d_i' - t_s = RS(J_i)$, we have

  $$\begin{aligned}(K - K') \times R_{min}(I^*) &\geq (K - \lfloor (d_i' - t_i)/(R_i + TO_i) \rfloor) \times R_{min}(I^*) \\ &\geq (K - (d_i' - t_i)/(R_i + TO_i))R_{min}(I^*) \\ &\geq d_i' - t_s - (d_i' - t_i)/R_{min}(I^*) \times R_{min}(I^*) \\ &= t_i - t_s.\end{aligned}$$

  Therefore, all jobs in $\mathcal{J}(I^*)$ can be schedulable.

- Case 1-b: $R_i + TO_i < R_{min}(I^*)$.

  Then there are at least K' faults, where $K' = \lceil (t_i - t_s)/(R_i + TO_i) \rceil$ before t_i. Otherwise, assume that there are $K' - 1$ faults before t_i, then there can be $K - K' + 1$ faults after(or at) t_i, we have $(K - K' + 1)(R_i + TO_i) > (K - (t_i - t_s)/(R_i + TO_i))(R_i + TO_i) \geq RS(J_i) - (t_i - t_s) = d_i' - t_i$, which causes J_i to miss its deadline according to Theorem 3.3.2. Since K' faults have already occurred before t_i, this implies that each job in $\mathcal{J}(I^*)$ at least has an additional slack of $K' \times R_{min}(I^*)$ to spare. Since

$$\begin{aligned} K' \times R_{min}(I^*) &= \lceil (t_i - t_s)/(R_i + TO_i) \rceil \times R_{min}(I^*) \\ &\geq (t_i - t_s)/(R_i + TO_i) \times R_{min}(I^*) \\ &\geq t_i - t_s, \end{aligned}$$

all jobs in $\mathcal{J}(I^*)$ are schedulable.

From the above discussions, we can then conclude that d'_i is a valid deadline for any $J_i \in \mathcal{J}^{do}_{I^*}$.

Case 2: let $J_i \in \mathcal{J}^{ao}_{I^*}$ and a'_i represent the new arrival time, $a'_i = t_s$ and d'_i the updated deadline, i.e. $d'_i = d_i - L(I^*) + RS(J_i)$. Note that J_i has lower priority than all the jobs in $\mathcal{J}(I^*)$. Therefore, we only need to show the changes made to the arrival time and deadline of J_i will not compromise the resource savings to guarantee the schedulability of J_i.

If all the jobs in $\mathcal{J}(I^*)$ and their recoveries finish at or before $t = t_f - RS(J_i)$, then J_i will not experience any interference from jobs in $\mathcal{J}(I^*)$ and its feasibility will not be affected. Now the question becomes what if all jobs in $\mathcal{J}(I^*)$ and their recoveries, if any, finish at t^*, where $t_f - RS(J_i) < t^* \leq t_f$, see Figure 3.5-(b).

We consider the following two cases.

- Case 2-a: $R_i + TO_i >= R_{min}(I^*)$. Then there are at least K' faults, where $K' = \lceil (t^* + RS(J_i) - t_f)/R_{min} \rceil$ before t^*. Otherwise, similar to the proof of Case 1-b, more than $K - K'$ faults occurring at $t = t^*$ will cause at least one job in $\mathcal{J}(I^*)$ to miss its deadline. In the meantime, this implies that J_i at least has an additional slack of $K' \times (R_i + TO_i)$ to spare. We have

$$\begin{aligned} K' \times (R_i + TO_i) &= \lceil (t^* + RS(J_i) - t_f)/R_{min}(I^*) \rceil \times (R_i + TO_i) \\ &\geq t^* + RS(J_i) - t_f. \end{aligned}$$

This ensures that J_i has reserved enough resource for fault recovery.

- Case 2-b: $(R_i + TO_i) < R_{min}(I^*)$. Then there are at most K' faults, where $K' = \lfloor (t_f - t^*)/R_{min}(I^*) \rfloor$ that may occur after t^*. Otherwise, one more fault at t_f will cause some job(s) in $\mathcal{J}(I^*)$ to miss deadline(s). In other words, there must be at least $K - K'$ faults that occurred before t^*. A portion of the shared slacks with the amount of $t^* + RS(J_i) - t_f$ is used by the jobs(recoveries) in $\mathcal{J}(I^*)$. However, job J_i reclaims an additional slack of $(K - K') \times (R_i + TO_i)$ to spare. Since $K \times (R_i + TO_i) \geq RS(J_i)$, we have

$$\begin{aligned}(K - K') \times R_i &\geq (K - \lfloor (t_f - t^*)/R_{min} \rfloor) \times (R_i + TO_i)\\ &\geq (K - (t_f - t^*)/(R_i + TO_i))(R_i + TO_i)\\ &\geq RS(J_i) - (t'_f - t^*)/(R_i + TO_i) \times (R_i + TO_i)\\ &= t^* + RS(J_i) - t_f.\end{aligned}$$

  Therefore, J_i also reserves enough resource.

Case 3: let $J_i \in \mathcal{J}^{fo}_{I^*}$. Similarly we want to prove that the change of deadline for J_i will not compromise the resource savings to guarantee its schedulability with the possible of maximum K faults. Since there are $K \times R_{max}(I^*)$ slacks reserved in I^*, it just requires additional slacks of $\max(0, K \times (R_i + TO_i) - K \times R_{max}(I^*))$ for J_i with the sharing mechanism.

We consider two cases below.

- Case 3-a: $(R_i + TO_i) > R_{max}(I^*)$. In this case, additional slacks of $K \times (R_i + TO_i) - K \times R_{max}(I^*)$ is reserved for J_i. Assume that K' faults occurred during the critical interval I^*. J_i can immediately claim the unused reserved slacks of $(K - K')R_{max}(I^*)$ in I^* to spare. Since there will be at most $K - K'$ faults striking J_i and we have the remaining reserved resources for J_i as

$$\begin{aligned}(K-K')R_{max}(I^*)+&(K\times(R_i+TO_i)-K\times R_{max}(I^*))\\ &=K\times(R_i+TO_i)-K'R_{max}(I^*)\\ &\geq(K-K')(R_i+TO_i).\end{aligned}$$

This ensures that J_i has reserved enough resources for fault recovery.

- Case 3-b: $(R_i + TO_i) \leq R_{max}(I^*)$. Then there is no additional slacks needed for J_i. Assume that there are K' faults in I^*. This implies J_i can reclaim $(K - K')R_{max}(I^*)$ from the critical interval I^* to spare. In addition, there will be at most $K - K'$ faults affecting J_i. Since $(K - K')R_{max}(I^*) >= (K - K')(R_i + TO_i)$, J_i has enough resources for its execution and recovery.

Since all jobs are associated with a critical interval in LPSSR and all jobs within a critical interval are schedulable when the corresponding speed is applied and the feasibility of the remaining jobs is not affected after the removal of a critical interval, we prove the theorem.

□

Algorithm 2 allows reserved slacks to be shared by different critical intervals and thus can achieve better energy efficiency. By far, both EMLPEDF and LPSSR assume that speeds can be continuously varied between $[s_{min}, s_{max}]$. In the next section, we extend our LPSSR algorithm to systems with only a limited number of frequencies.

3.5 Other considerations of the proposed methods

In this section, we relax our assumptions about system and fault model and explicitly address some practical issues in modern processors.

3.5.1 Dealing with the limitations of practical processors

Up to now, we assume that the processor speed can be varied continuously. However, current commercial variable voltage processors only have a finite number of speeds [122],[111]. In addition, it takes time for a processor to change its running modes. These factors must be taken into consideration to provide a practical, valid and efficient voltage schedule.

One intuitive way to deal with discrete frequency levels is to round up the required frequency to the next available level. Unfortunately, this can be extremely pessimistic and energy inefficient, especially for processors with only a few frequencies available. In fact, we can adopt the similar approach as in the work [92] to deal with both the problem of discrete levels of working frequencies and non-zero timing overhead. As shown in [92], non-zero timing overhead can cause **monotonicity violation** similar to the scenario when we insert recovery blocks for fault tolerance. Therefore, the transition overhead can be efficiently handled by adding it to the reserved blocks. For discrete frequency levels, we can take this factor into consideration when constructing critical intervals. Specifically, when a critical interval is found according to Algorithm 2, its speed needs to be raised to the next level available. Once a higher than necessary speed is used, idle slacks will be generated in the critical interval. Then we reduce the idle slacks by identifying the **latest finishing time** of the critical interval.

Given the jobs in the interval and a higher speed than required, we can find the latest finishing time of the workload including recoveries under the worst case as follows. Let $I^* = [t_s, t_f]$ be the critical interval and s_h be its speed. In addition, the set of jobs in I^* is denoted by $\mathcal{J}(I^*)$ and $\mathcal{J}_{hp}(i)$ is the set of jobs of priority higher than that of job J_i. Therefore, the latest finishing time($LFT(I^*)$) is obtained by equation (3.7),

$$LFT(I^*) = \max_{\forall J_i \in \mathcal{J}(I^*)} \{ \frac{c_i}{s_h} + \sum_{\forall J_j \in \mathcal{J}_{hp}(i) \cap d_j > a_i} \frac{c_j}{s_h} + K \times Re(i) + a_i \} \qquad (3.7)$$

where the first part denotes the execution requirement from J_i itself and the second and third part represent the interference from the higher priority jobs and the worst case recovery time, i.e. $Re(i) = \max\{R_p + TO_p | J_p \in \{J_i\} \cup \mathcal{J}_{hp}(i)\}$ that J_i can suffer, respectively. Note that not all the workload from higher priority jobs are considered because only those with deadlines after the arrival of J_i, i.e. a_i may delay the execution of J_i. Therefore, the actual critical interval to be removed is $[t_s, \min(t_f, LFT(I^*))]$.

To update our LPSSR to deal with discrete frequency levels, we only need to calculate the latest finishing time and update the ending point of the critical interval before line 14. Similarly, this technique can be incorporated into MLPEDF and EMLPEDF as well.

3.5.2 System reliability and imperfect fault coverage

Our proposed approach enhances the system reliability by ensuring the K-fault-tolerance capability of the system through advanced backup policies. Let the system reliability be defined in 3.5.1.

Definition 3.5.1. *The* reliability *of the system, denoted as R_{sys}, is the probability that the system functions correctly during an operational cycle (its length is represented by L_{cyc}) of the system.*

Let $Pr(q, L_{cyc})$ denote the probability that exactly q faults occur during L_{cyc} and ρ denote the fault coverage of the given fault detection method, i.e. $0 < \rho \leq 1$. To ensure the system to function correctly, two conditions have to be met: 1) no more than K faults occur during L_{cyc}; 2) all failures are appropriately detected. As the event of fault occurrence is independent of the process of fault detection, the system reliability R_{sys} can be calculated in equation (3.8),

$$R_{sys} = \sum_{q=1}^{K} Pr(q, L_{cyc}) \cdot Pr_{det}(q), \tag{3.8}$$

where $Pr_{det}(q) = \rho^q$, i.e. the probability that q faults are detected. If the failure distribution is modeled as a Possion process with a failure rate λ as in [136, 134, 133], then the reliability function is shown in equation (3.9),

$$R_{sys} = \sum_{q=1}^{K} \frac{(\lambda L_{cyc})^q \cdot e^{-\lambda L_{cyc}}}{q!} \cdot \rho^q \tag{3.9}$$

As can be seen from both equation (3.8) and (3.9), the larger the number of faults, i.e. K that the system can tolerate, the higher the system reliability. Note that, this reliability model is not limited to any particular failure distribution, as long as $Pr(q, L_{cyc})$ is well defined, it can be readily applied. Given a reliability goal and the length of an operational cycle of the system, a corresponding K can be determined under any given fault detection technique.

From equation 3.9, the fault coverage factor can play an important role in system reliability. To study the tradeoffs between different fault coverage techniques is an interesting research problem and will be our future work.

3.6 Simulation results

In this section, we compare the performance of four algorithms: NPM, MLPEDF, EMLPEDF, and LPSSR. NPM represents the speed schedule with no power management involved, i.e. all jobs or recoveries are executed under s_{max} and is used as a reference schedule. MLPEDF and EMPLEDF are fault tolerant algorithms discussed in Section 3.3 and LPSSR is the algorithm presented in Section 3.4. All energy consumptions plotted were normalized to NPM.

We assumed that $\alpha = 2, C_{ef} = 1$, $P_{ind} = 0.05$, and s_{min} was set to 0.25. We tested our algorithm with job sets randomly generated as follows: for each job, the arrival time a_i is uniformly distributed in the interval [0s,100s] while its relative deadline rd_i is in [50s,100s]. Therefore, the absolute deadline was calculated as $d_i = a_i + rd_i$. In addition, the worst case execution time c_i was less than rd_i and also randomly generated. For each job, the timing and energy overhead of fault detection is set to 10% of its worst case execution time and its energy consumption, respectively. The choices of K were based on the characteristics of the task sets and typical fault arrival rates. As indicated in [130], the typical fault arrival rate in safety-critical real-time system is in the range of 10^{-10} to $10^{-5}/hour$. However, for systems that operate in harsh environment, the fault arrival rate can be much higher, in the range of 10^{-2} to $10^{2}/hour$. Only the job sets running at s_{max} that are feasible under K faults are of interest to us.

Two sets of simulations were conducted to study the performance of our algorithm in terms of energy savings under continuously varied speeds and discrete speed levels, respectively.

3.6.1 System with continuous speeds

First, we studied how energy saving performance changes with the number of jobs. We set the fault rate to be 10^{-5} and varied the number of jobs from 10 to 50. For simulations with the same number of jobs, we generated at least 1000 different test cases. With our settings, the number of fault in our job set is no more than 1. Therefore, we set $K = 1$. For each job set, we collected the energy consumption

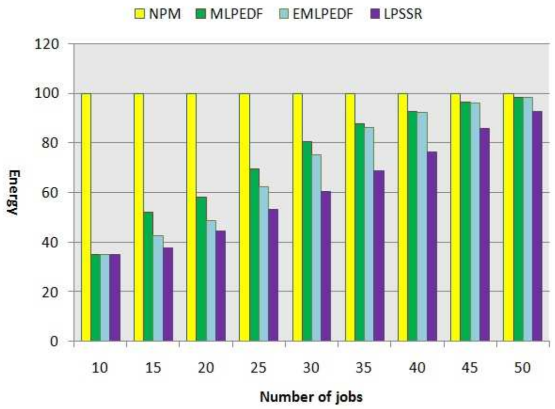


Figure 3.6: Energy savings with different numbers of jobs, K = 1

of the speed schedule by each of the four approaches. The result is illustrated in Figure 3.6.

From Figure 3.6, we can see that the energy consumption of LPSSR, EMLPEDF and MLPEDF increases as the job set becomes larger. This is reasonable since the workload is increasing while the slacks that can be used for DVFS are diminishing. LPSSR always dominates the other three algorithms because, by sharing reserved slacks, LPSSR reserves fewer resources for fault recovery and uses more for slowing down the execution of jobs. When the workload is very low, i.e., only 10 jobs, the energy savings achieved by all three algorithms are almost the same, this is due to the fact that most of the test cases are feasible under constant speed s_{min}. When the workload is high enough, most of the slacks are used for fault recovery and no room is left for DVFS. Moreover, if the number of jobs is increased to a certain point, no fault-tolerant speed schedule can be found. In average, additional 13% and 10% energy saving can be achieved by LPSSR when comparing with MLPEDF and EMLPEDF, respectively.

In our second set of simulations, we wanted to investigate how the number of faults affects the performance of our algorithm. In this simulation, the number of

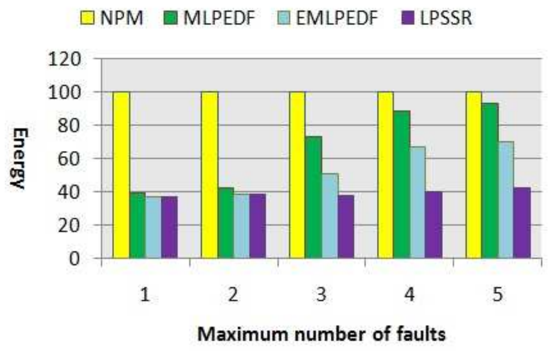


Figure 3.7: Energy savings with increasing number of faults, # of jobs = 15

jobs is fixed to 15 and the fault rates to be tolerated varies from 10^{-2} to $10^{2}/h$, i.e. K changes from 1 to 5. Again, no less than 1000 different test cases were generated for simulations with the same fault numbers. The average results are shown in Figure 3.7.

From Figure 3.7 we can see that the energy consumptions by MLPEDF and EMLPEDF increase rapidly as the increase of the number of faults. The energy consumption by LPSSR, on the other hand, grows but less dramatically. From Figure 3.7, the energy consumption difference is around 6% between tolerating 1 fault and 5 faults under LPSSR. This is due to the fact that the recovery slacks are shared to the maximum extent by employing the sharing mechanism in LPSSR. On the contrary, MLPEDF(EMLPEDF) is affected significantly by the increasing number of faults in the system and more than 40%(33%) additional energy is consumed when fault occurrences increase from 1 to 5.

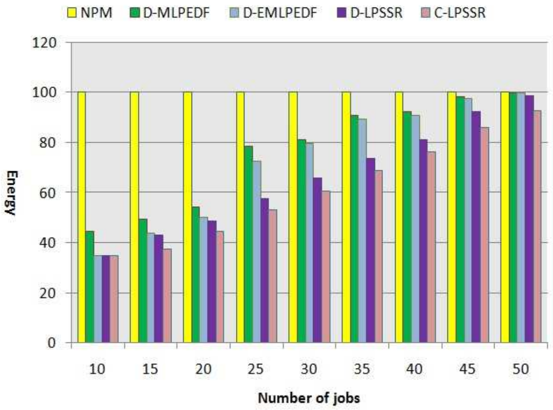


Figure 3.8: Energy savings with increasing number of jobs under PentiumM, K=1,

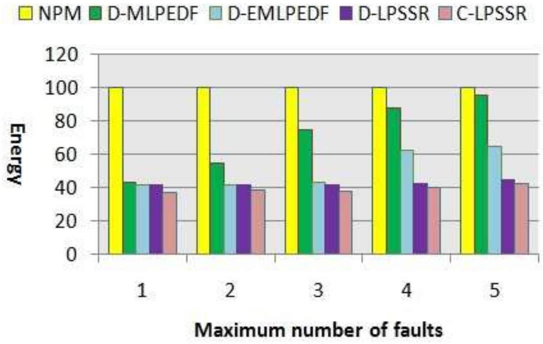


Figure 3.9: Energy savings with increasing number of faults under PentiumM, # of jobs = 15

3.6.2 System with discrete speed levels

In this section, we also evaluate the four algorithms using two different sets of simluations. The technique discussed in Section 3.5.1 is used to deal with discrete speed levels.

We adopt PentiumM processor with 8 frequency levels (1.00, 0.86, 0.76, 0.67, 0.57, 0.47, 0.38, 0.28) as our target system as used in [88]. Two simulations under the same configuration as those in Section 3.6.1 are performed and their results are shown in Figure 3.8 and Figure 3.9, respectively. Again, four algorithms with limited number of speeds are evaluated, which are NPM, D-MLPEDF, D-EMLPEDF and

D-LPSSR, respectively. To better illustrate the performance of our algorithm under discrete speed levels, we compare it with that of continuous speeds, which is denoted by C-LPSSR.

The advantages of our algorithm D-LPSSR over the other two in terms of energy savings are manifested in Figure 3.8, and the additional energy savings only drops around 3% compared with continuous varied speeds. In average, the difference between D-LPSSR and C-LPSSR is only 5%.

Moreover, for the second simulation, algorithm LPSSR performs even better as shown in Figure 3.9. This is due to the fact that it extensively explores the slacks that can be shared among different critical intervals and significantly reduce the amount of recoveries. Therefore, increasing the number of faults has little impact on the resulting speed schedule. Comparing with C-LPSSR, only 3% more energy is consumed for tolerating 1 to 5 faults.

3.6.3 Real-life periodic task sets

In this section, we verify the proposed algorithms using three real-life periodic task sets, which are a CNC task set, an inertial navigation system(INS) task set, and a generic aviation platform(GAP) task set, respectively. The specifications of these task sets can be found in [130] and omitted here due to space limitation. Based on our simulations, no task set can tolerate more than 2 faults. Therefore, only the results of K = 1, 2 are recorded and are normalized to NPM. Processors with continuous frequencies and discrete frequency levels are considered, separately.

As shown in Table 3.2, all three algorithms can achieve energy savings compared with NPM while maintaining the feasibility of the task sets, where A1, A2 and A3 stand for MLPEDF, EMLPEDF and LPSSR, respectively. The two algorithms, i.e.

Table 3.2: Energy-performance comparison for CNC, INS, and GAP

		Continuous frequencies			PentiumM		
Task Set	K	A1(%)	A2 (%)	A3 (%)	A1(%)	A2 (%)	A3 (%)
CNC	1	69.9	60.4	59.9	72.8	62.6	61.9
	2	86.4	80.5	71.4	91.8	84.8	77.4
INS	1	96.3	93.0	88.5	98.7	96.2	92.1
	2	NF	NF	NF	NF	NF	NF
GAP	1	91.5	89.4	87.2	98.4	96.9	93.3
	2	100	100	92.2	100	100	96.8

EMLPEDF and LPSSR have similar performance when tolerating 1 fault, because the shared slacks are negligible considering a relatively small execution time to period ratio. For all three algorithms, we noticed a consequent increase in energy consumption when K increases. This increase mostly comes from the first iteration of the algorithm, the intensity of the first critical interval is much higher for a larger K, especially for a task set with large utilization where slacks are already scarce. However, when K = 2, LPSSR stills attains another 8.5%(12%) energy reduction compared with EMLPEDF (MLPEDF), which is a strong demonstration of the benefits from slack-sharing. Under a processor with a limited number of frequencies, i.e. PentiumM in Section 3.6.2, the performance of our algorithms is slightly degraded as expected.

3.6.4 Further validation of LPSSR

To our best knowledge, there is no other existing works in the literature addressing the exactly same problem. However, to demonstrate the efficacy of our LPSSR, we compared LPSSR against the method fault-tolerant uniform checkpointing with DVFS (FTUniChK) from the work [90] that studied the fault-tolerant energy reduction for periodic task sets scheduled under EDF on a single processor. FTUniChK first identified the the checkpointing interval and then derived a constant speed

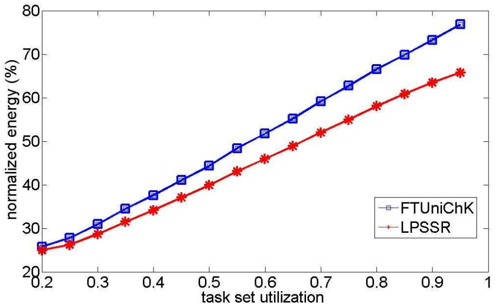


Figure 3.10: LPSSR vs FTUniCk

to execute the entire task set, but it is only applicable when no more than one fault can occur, i.e. $K = 1$. Note that, our LPSSR exploits the slacks that can be shared among different jobs and acts on top of any checkpointing scheme. Therefore, we directly adopted the uniform checkpointing scheme from [90] before employing LPSSR.

The simulation parameters were set as follows. We had $\alpha = 2, C_{ef} = 1$, $P_{ind} = 0.05$, and s_{min} was set to 0.25. Each task set consisted of 10 periodic tasks, whose periods were uniformly generated in the range of [5s 50s]. The checkpointing overhead of each task was set to 5% of its worst case execution time under s_{max}. The total utilization of the task set was varied from 0.2 to 0.95 with a step of 0.05. For each utilization value, we generated 1000 different task sets according to UUNISORT in [17], and the average energy consumption of one LCM was reported. We again normalized the energy consumption with respect to that of NPM.

According to Figure 3.10, our LPSSR consistently outperforms FTUniChK. When the processor is light-loaded, both methods use close-to-minimum speed to execute the task set, therefore the energy performance is close. However, as the utilization increase, LPSSR can reduce the amount of slacks reserved for fault-tolerance and use more for energy reduction compared to FTUniChK. For instance, when the utilization is 0.8, LPSSR achieves around 10% more energy savings.

Through extensive simulations, we have shown that the three proposed algorithms can save a significant amount of energy comparing with NPM. Specifically, our LPSSR algorithm is more energy efficient by reserving the least amount of slacks.

3.7 Summary

In this chapter, we investigate the problem of minimizing energy consumption when scheduling a set of real-time jobs in presence of up to K transient faults under EDF policy. We explore the reserved slacks in the system and maximize its utility by providing a slack sharing mechanism. Under the notion of shared recovery slacks, we propose an algorithm that reduces the energy consumption and maintains feasibility under the worst case, i.e. up to K faults occur during one operational cycle of the system. We then extend our algorithm to systems with discrete speed levels to provide practical and energy efficient solutions. Theoretical validation of our approach is provided and the simulation results have shown that our approach consistently results in lower energy consumption compared with other algorithms.

CHAPTER 4

ENERGY MINIMIZATION FOR FAULT TOLERANT REAL-TIME APPLICATIONS ON MULTI-CORE PLATFORMS USING CHECKPOINTING

In the previous chapter, we study the problem of fault-tolerant real-time scheduling for EDF-scheduled real-time tasks on a single-core platform. As more and more real-time systems are adopting multi-core architecture as the underlying structure, it is imperative that we develop energy-efficient fault-tolerant scheduling on multi-core platforms. Thereby, in this chapter, we study the energy minimization problem for real-time applications on multi-core platforms while tolerating K transient faults using checkpointing.

A key to solve this problem is to make the judicious tradeoffs between the number of checkpoints for each task and the amount of reserved resources for fault recovery. In this chapter, we first study the problem on how to identify the appropriate numbers of checkpoints for tasks on a single core to minimize the worst case response time. Based on the results, we then develop an efficient method to optimize the energy consumption for a real-time application while ensuring that K transient faults can be tolerated.

The rest of the chapter is organized as follows. We first review the related works in Section 4.1. Section 4.2 introduces the system models and notations used throughout this chapter. Section 4.3 presents our method to minimize the worse case latency for real-time tasks on a single-core processor. We then present our energy efficient fault-tolerant algorithm in section 4.4. The effectiveness and efficiency of our algorithms are evaluated in Section 4.5. Finally, section 4.6 summarizes the chapter.

4.1 Related works

A plethora of techniques has been presented in the literature on real-time scheduling with both fault tolerance and energy minimization requirements. For example, Zhang et al. [130] introduced a static combination of checkpointing and DVFS scheme for fixed-priority tasks for tolerating K transient faults while minimizing energy consumption. This approach was extended by Wei et al. [124] to explore run-time slacks for further reducing energy consumption. Zhao et al. [134] considered the negative effects of DVFS on transient fault rate and proposed a task-level reliability model. They developed algorithms to determine DVFS schedules and resource-reservation schemes to minimize energy consumption while meeting task-level reliability requirements. All these approaches are restricted to single-core platforms.

As more and more transistors are integrated to the same chip, and due to problems such as the power/thermal issues and limitations in instruction level parallelism [7], multi-core platforms are becoming mainstream. As a result, most of the research efforts are turned to multi-core platforms.

Pop et al. [97] presented a constraint logic programming method to design low-power fault-tolerant hard real-time applications on distributed heterogeneous platforms. They assumed that the task allocation is fixed and known a priori, and an entire task needs to be re-executed when a transient fault occurs. Qi et al. [101] derived a reliability-aware global scheduling scheme aiming at reducing the system energy consumption for a set of frame-based tasks running on a homogeneous multi-core platform. They assumed that different tasks can share the same reserved sources to recover when faults happen. Again, the entire task has to be re-executed in case of faults, which can greatly affect the energy efficiency of the system. Pop et al. [96] proposed a more comprehensive approach to the synthesis of fault tolerant schedule for applications on heterogeneous distributed systems. They used the combination of checkpointing and active replication to deal with the fault tolerance problem. A meta-heuristic (Tabu search) is constructed to decide the fault-tolerance policy, the placement of checkpoints and the mapping of tasks to processing cores, but energy consumption is not considered in their approach.

We are interested in the problem of minimizing energy consumption while tolerating up to K transient faults with checkpointing scheme for a real-time system running on a homogeneous multi-core platform. In the following, we introduce some important preliminaries on our research and explain the notations used throughout this chapter.

4.2 Preliminaries

In this section, we present the background pertinent to our research. Specifically, we introduce the models and notations that are critical to our research.

4.2.1 Application model

The real-time applications considered in this chapter consist of n independent tasks, denoted as $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$. All tasks in Γ have the same deadline D, but with different execution requirements. We denote the execution time of τ_i as c_i. The utilization of task τ_i is represented as $u_i = \frac{c_i}{D}$. The system utilization U is therefore calculated as $U_{total} = \sum_{i=1}^{n} \frac{c_i}{D}$.

4.2.2 Fault model and checkpointing

In this chapter, we only consider transient faults that can be tolerated by backward rollback recoveries. We assume that the system needs to tolerate K faults, and the faults can happen on any of the processing cores and at any time, even in a burst manner. Run-time faults are countered by rolling back to the latest checkpoints and re-executing the corrupted segments. Checkpointing is considered to be self fault-tolerant.

The timing and energy overhead of inserting one checkpoint to task τ_i (saving the fault-free state) are denoted by o_i and eo_i, respectively. In addition, we use r_i (er_i) to denote the time (energy) it takes to retrieve the information needed to rollback to the latest checkpoint when a fault happens during the execution of τ_i. Fault detection is performed at each checkpoint to ensure the correctness of the saved state. The timing and energy overhead for such an operation are represented as q_i and eq_i, respectively. Assuming m_i checkpoints inserted into τ_i, fault detections will be performed for total $(m_i + 1)$ times (including one fault detection at the end of τ_i's execution). Therefore, the fault-free execution time of τ_i with m_i number of checkpoints, denoted as $c_i'(m_i)$, can be specified as shown in equation (4.1a). The recovery time of τ_i with a single failure, denoted as $R_i(m_i)$, is shown in equation (4.1b).

$$c_i'(m_i) = c_i + m_i(o_i + q_i) + q_i \tag{4.1a}$$

$$R_i(m_i) = r_i + \frac{c_i}{m_i + 1} + q_i \tag{4.1b}$$

4.2.3 Platform and energy model

We consider a homogeneous multi-core platform Ψ with m cores, i.e. $\Psi = \{\psi_1, ..., \psi_m\}$. We assume all the cores are identical in terms of processing frequency and power characteristics. For the ease of presentation, we assume the speed/frequency of a core can be changed continuously in $[f_{min}, f_{max}]$ with $0 \leq f_{min} \leq f_{max} = 1$. As discussed later in this chapter, this constraint can be easily relaxed to accommodate the fact that most practical processors support a set of discrete levels of frequencies.

Our system-level power model is similar to that in [88] by distinguishing the dynamic and leakage power components. Specifically, the overall power consumption P can be formulated as

$$P = P_{leak} + P_{dyn} = P_{leak} + C_{ef} f^{\alpha} \tag{4.2}$$

where P_{leak} is the constant leakage power and can be only eliminated by turning down the processing core. C_{ef} is the effective switching capacitance. α is a constant usually larger than 1. P_{dyn} is the dynamic power consumed when the device changes logic states. Hence, the energy consumption of a task τ_i with m_i checkpoints running under the frequency f_i can be expressed as:

$$\begin{aligned} E_i(f_i) = &(P_{leak} + C_{ef} f_i^{\alpha}) \cdot \frac{c_i}{f_i} \\ &+ m_i(eo_i + o_i P_{leak}) + (m_i + 1)(eq_i + q_i P_{leak}), \end{aligned} \tag{4.3}$$

which includes the energy consumption incurred by executing task τ_i and the energy overheads caused by checkpointing and fault detections. Similar to [130], we consider checkpointing, fault detections and checkpoint retrievals are frequency independent, but leakage power is still consumed during their operations. As $E_i(f_i)$ is a convex function, the minimum system energy is achieved when f_i is as small as possible, provided it is larger than so-called *critical frequency* ($f_c = \sqrt[\alpha]{\frac{P_{leak}}{(\alpha-1)C_{ef}}}$) [88].

We use Γ_j to represent the set of tasks assigned to the core ψ_j. The energy consumption of core ψ_j can be calculated using equation (4.4).

$$E(\Gamma_j) = \sum_{\tau_i \in \Gamma_j} E_i(f_i). \tag{4.4}$$

The total energy consumption of the system is thus $E(\Gamma) = \sum_{j=1}^{m} E(\Gamma_j)$.

4.3 Optimal checkpointing scheme for minimizing the worst case latency on a single core

Our goal is to develop a method that can minimize the energy consumption while ensuring the K-fault tolerance using checkpointing. A key to solve this problem is to make judicious decisions on inserting checkpoints to each task. As shown in the previous section, increasing the numbers of checkpoints for real-time tasks incurs larger checkpointing overhead which may compromise the feasibility and/or energy efficiency of real-time systems. On the other hand, however, increasing the checkpoint numbers decreases the needs of larger resource reservation for fault recovery, which can be in favor of both system feasibility and energy efficiency. As a result, the number of checkpoints (or the checkpointing interval) must be carefully chosen to balance the checkpointing overhead with the fault recovery cost.

As a closely related work, Zhang et al. [130] showed that the optimal number of checkpoints to minimize the worst case latency of a single task τ_i, denoted as m_i^*, can be calculated as

$$m_i^* = \begin{cases} \lceil \sqrt{\frac{K*c_i}{o_i+q_i}} - 1 \rceil & \text{if } c_i > \frac{(m_i^-+1)(m_i^-+2)(o_i+q_i)}{K} \\ \lfloor \sqrt{\frac{K*c_i}{o_i+q_i}} - 1 \rfloor & \text{if } c_i \leq \frac{(m_i^-+1)(m_i^-+2)(o_i+q_i)}{K} \end{cases}$$

where $m_i^- = \lfloor \sqrt{\frac{K*c_i}{o_i+q_i}} - 1 \rfloor$. However, when considering multiple tasks that share recovery resources on a single-core processor, the individual optimal checkpointing configuration does not necessarily lead to the global optimal result. Pop et al. [96] resorted to meta-heuristic (i.e. Tabu search) to search for the global optimal solution. It is desirable that a more efficient and effective method can be developed to identify the optimal global checkpointing settings, especially during the design space exploration process.

Assuming all tasks on the same core share the same recovery resources, to tolerate K faults, we must reserve enough CPU time, i.e. $K \times SR$, to re-execute the corresponding program segments, where $SR = \max\limits_{i=1,...n} \{R_i(m_i)\}$, m_i is the number of checkpoints for τ_i, and $R_i(m_i)$ is defined in equation (4.1b). We call SR as the *shared recovery block*. Considering the task set $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$ is allocated to the same core, the worst case latency of task set Γ with shared recovery block of SR, denoted as $L(\Gamma, SR)$, can be formulated in equation (4.5)

$$L(\Gamma, SR) = \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} (m_i * o_i + m_i * q_i + q_i) + K * SR. \tag{4.5}$$

To find the optimal checkpointing scheme that minimize the worst case latency, i.e. $L(\Gamma, SR)$, we have the following theorem.

Theorem 4.3.1. *If $\{m_1, m_2, ..., m_n\}$ is the optimal checkpointing configuration to minimize $L(\Gamma, SR)$, then we have $\forall i, m_i \leq m_i^*$, where m_i^* is the optimal number of checkpoints to run a task τ_i individually.*

Proof. We prove this theorem by contradiction.

Let $\{m_1, m_2, ..., m_n\}$ be the optimal checkpointing configuration but $\exists i \in [1, n], m_i > m_i^*$. Let $\{m_1, m_2, ..., m_i^*, ...m_n\}$ be another configuration that distinguishes the former one only by the number of checkpoints for task τ_i. SR and SR' denote the sizes of the shared recovery blocks under two configurations, respectively. δ represents the difference between the two worst case latencies, i.e. $\delta = L(\Gamma, SR) - L(\Gamma, SR')$. Then, we have $\delta = m_i(o_i + q_i) + K * SR - (m_i^*(o_i + q_i) + K * SR')$ according to equation (4.5).

Note that SR can be potentially increased after reducing m_i to m_i^*, we discuss the two possible scenarios separately in the following.

- *Case 1:* $R_i(m_i^*) \leq SR$. In this case, reducing m_i to m_i^* does not change the size of the shared recovery block, i.e. $SR' = SR$. Because $m_i > m_i^*$, we know $\delta > 0$.
- *Case 2:* $R_i(m_i^*) > SR$. This means that the share recovery block is increased due to the decrease in the checkpointing number of task τ_i and $SR' = R_i(m_i^*)$. Since $SR \geq R_i(m_i)$, if we replace SR (SR') with $R_i(m_i)$ ($R_i(m_i^*)$), respectively, we have

$$\delta \geq m_i(o_i + q_i) + K * R_i(m_i) - (m_i^*(o_i + q_i) + K * R_i(m_i^*)). \tag{4.6}$$

Note that the right hand side of equation (4.6) represents the difference of two worst case latencies when running τ_i individually using two different checkpointing schemes. Since m_i^* is the optimal checkpoint solution, we must have $\delta > 0$.

For both cases, we have $\delta > 0$. This contradicts our assumption that M is optimal. □

Theorem 4.3.1 helps to prune the search space for the checkpointing configurations. However, a brute-force method based on Theorem 4.3.1 still has a very high computational complexity, i.e. $\prod_{i=1}^{n} m_i^*$, which can be computationally prohibitive for large task sets with a considerable amount of possible values of m_i^*. In what follows, we introduce a novel approach to further prune the search space.

Since $SR = \max\limits_{i=1,\ldots n}\{R_i(m_i)\}$, from equation (4.1b), for a given SR, we have

$$m_i = \lceil \frac{c_i}{SR - (r_i + q_i)} - 1 \rceil. \tag{4.7}$$

Therefore, equation (4.5) can be transformed to

$$\begin{aligned} L(\Gamma, SR) = \sum_{i=1}^{n} (c_i + q_i) + \sum_{i=1}^{n} \lceil \frac{c_i}{SR - (r_i + q_i)} - 1 \rceil (o_i + q_i) \\ + K * SR \end{aligned} \tag{4.8}$$

Therefore, to search for the optimal checkpointing configurations, we only need to search the optimal value of SR that can optimize $L(\Gamma, SR)$. To achieve this purpose, we first introduce the following lemma.

Lemma 4.3.2. *If $M = \{m_1, m_2, ..., m_n\}$ is the optimal checkpointing configuration for task set $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$, then the size of the shared recovery block SR under configuration M is no less than $\max\limits_{i=1,...n}\{R_i(m_i^*)\}$.*

Proof. The proof of Lemma 4.3.2 is similar to that of Theorem 4.3.1. We also prove it by contradiction. The configuration M is assumed to be optimal but the resulting $SR < \max\limits_{i=1,...n}\{R_i(m_i^*)\}$. Let task τ_k have the longest recovery time, i.e. $R_k(m_k^*) = \max\limits_{i=1,...n}\{R_i(m_i^*)\}$. According to equation (4.7), the number checkpoint of τ_k is calculated as $m_k = \lceil \frac{c_k}{SR-(r_k+q_k)} - 1 \rceil > \lceil \frac{c_k}{R_k(m_k^*)-(r_k+q_k)} - 1 \rceil = \lceil m_k^* \rceil$. This contradicts Theorem 4.3.1. □

From Lemma 4.3.2, we can immediately set up a lower bound for SR as

$$SR \geq \max_{i=1,...n}\{\frac{c_i}{m_i^*+1}\}. \tag{4.9}$$

Moreover, based on the properties of ceiling(floor) functions and equation (4.8), we can set an upper bound and a lower bound as follows:

$$L_{upper}(\mathcal{T}, SR) = \sum_{i=1}^{n}(c_i + q_i) + \sum_{i=1}^{n}\frac{c_i}{SR - \phi_{max}}(o_i + q_i) + K * SR \tag{4.10a}$$

$$L_{lower}(\mathcal{T}, SR) = \sum_{i=1}^{n}(c_i - o_i) + \sum_{i=1}^{n}\frac{c_i}{SR - \phi_{min}}(o_i + q_i) + K * SR \tag{4.10b}$$

where $\phi_{max} = \max\limits_{i=1,2,...,n}(r_i + q_i)$ and $\phi_{min} = \min\limits_{i=1,2,...,n}(r_i + q_i)$.

Note that the two curves defined in equation (4.10a) and (4.10b) constrain the optimal SR as shown in Figure 4.1. Moreover, from equation (4.10a), we can readily calculate the minimum upper bound by setting

$$\frac{\partial L_{upper}(\Gamma, SR)}{\partial SR} = 0. \tag{4.11}$$

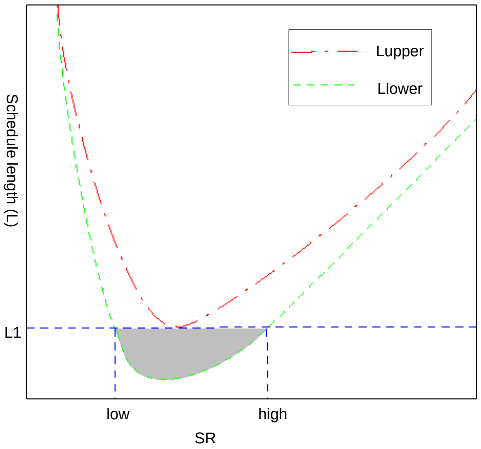


Figure 4.1: Upper and lower bounds of $L(\Gamma, SR)$

As can be seen from Figure 4.1, the optimal SR can only be located in the shaded range between $[low, high]$, beyond which L is always greater than $L1$, which is the solution of equation (4.11). The exact values of *low* and *high* can be calculated accordingly by solving the following equation

$$L_{lower}(\Gamma, SR) = L1. \tag{4.12}$$

Algorithm 3 OPT_CHK(Γ, K)

```
obtain $m_i^*$, for $i = 1, 2, ..., n$ according to [96]
$L_{min} = INF$;
$\mathcal{S} = \{SR_{i,k} | SR_{i,k} = R_i(k), i = 1, ..., n; k = 1, ...m_i^*\}$;
Prune $\mathcal{S}$ based on equation (4.9) and solutions of equation (4.12);
for $i = 1$; $i \leq sizeof(\mathcal{S})$; $i++$ do
    calculate $L(\Gamma, \mathcal{S}(i))$ according to equation (4.8);
    if $L(\Gamma, \mathcal{S}(i)) < L_{min}$ then
        $L_{min} = L(\Gamma, \mathcal{S}(i))$;
        $SR_{opt} = \mathcal{S}(i)$;
    end if
end for
$m_i = \lceil \frac{c_i}{SR_{opt} - (r_i + q_i)} - 1 \rceil \ \forall i = 1, ...n$;
return $L_{min}, SR_{opt}, M = m_i, i = 1, ...n$
```

As such, equation(4.9) and solutions of equation (4.12) can be effectively used for pruning the solution space for the optimal checkpoint configurations. We summarize the procedures in Algorithm 3. It is not difficult to see that the complexity of Algorithm 3 is linear to the possible values of SR. In section 4.5, we use experimental results to test the efficiency of our approach.

4.4 Energy-aware fault-tolerant task allocation

With our analysis results and algorithm to search for the optimal checkpointing scheme on a single-core processor, we are now ready to present our algorithm to minimize the overall energy consumption while tolerating K transient faults on multi-core platforms.

Without fault tolerance requirement, one intuitive method is to spread the workload among multi-core platforms as even as possible [4]. When fault tolerance requirements are taken into consideration, however, extra care must be taken since both resource reservation and DVFS compete for system slack time. Aggressively packing as many tasks as possible into one core helps to reduce the resource reservation since the reserved resource can be shared by all tasks in the same core. However, with too much workload stacked in one core, it becomes difficult for a core to scale down the processing speed. On the contrary, spreading tasks around helps to balance the workload among different cores and thus effectively reduces the processing speed. The problem is that potentially more resources need to be reserved since tasks allocated to different cores cannot share the same reserved resources. Moreover, as indicated in our analysis results before, different sets of tasks may lead to totally different optimal checkpointing results, i.e. resource-reservation schemes.

It is well known that the multi-objective task allocation problem is a NP-hard problem in the strong sense [4]. Therefore, we focus our effort on developing an effective heuristic solution for this problem. Our task allocation scheme for energy

Algorithm 4 EATA(Γ, Ψ, K)

```
obtain $m_i^*$, for $i = 1, 2, ..., n$ according to [96]
$E_{total} = 0$;
$\Gamma_j$ = NULL, for $j = 1, 2, ..., m$;
for $i = 1$; $i \leq n$; $i++$ do
   $\Delta E = \infty$;
   assigned = 0;
   $M_{new}$ = NULL;
   for $j = 1$; $j \leq m$; $j++$ do
      $\{L_{temp}$, $SR_{temp}$, $M_{temp}\}$ = OPT_CHK($\Gamma_j \cup \{\tau_i\}$, K);
      $E_{temp} = E(\Gamma_i \cup \{\tau_i\})$
      if $L_{temp} \leq D$ and $E_{temp} < \Delta E$ then
         assigned = j;
         $\Delta E = E_{temp}$, $M_{new} = M_{temp}$
      end if
   end for
   if $assigned == 0$ then
      return "not feasible";
   else
      $\Gamma_{assigned} \leftarrow \Gamma_{assigned} \cup \{\tau_i\}$;
      $M = M_{new}$;
      $E_{total} \leftarrow E_{total} + \Delta E$;
   end if
end for
return $\{\Gamma_1, ..., \Gamma_m\}, E_{total}, M$
```

minimization with K fault tolerance guarantee is developed based on the algorithm OPT_CHK. Specifically, when allocating a new task τ_i, we assign τ_i to the core that leads to the minimum energy consumption increase. Note that, when assigning τ_i to a core (e.g. ψ_j), the optimal checkpoint configurations can be obtained using algorithm OPT_CHK. We assume that the re-execution of a faulty task is always performed at the highest speed and the checkpointing overhead is independent to the core's running mode. Then the core speed for ψ_j, i.e. f_j, can be determined by

$$f_j = \max(\frac{\sum\limits_{\tau_i \in \Gamma_j} c_i}{D - \sum\limits_{\tau_i \in \Gamma_j} c_i'(m_i) - K * \max\limits_{\tau_i \in \Gamma_j} R_i(m_i)}, f_c) \tag{4.13}$$

where f_c is the critical speed, $c_*'(*)$ and $R_*(*)$ are obtained through equations (4.1a) and (4.1b), respectively. Also, the energy consumption of core ψ_j, i.e $E(\Gamma_j)$, can be calculated according to equation (4.4). Note that even though we assume the frequency of a core can be continuously varied, we can still adopt the traditional approach [70] to deal with the scenario when only a set of discrete levels of frequencies are available. Specifically, if the desired constant frequency, i.e. f_i, is not available, we identify two available neighboring frequencies of f_i to run the task set Γ_j on ψ_j. The overall algorithm is described in Algorithm 4. It is not difficult to see that the overall complexity of Algorithm 4 is $O(n \times m \times |\mathcal{S}|)$, where $|\mathcal{S}|$ is the worst case possible values of the shared reservation block on a core.

4.5 Experimental results

In this section, we study the effectiveness and efficiency of our proposed algorithms. To our best knowledge, there is no existing approach targeting the exact same problem. As a result, to study the energy saving performance of $EATA$, we compared it with two well-known fault-oblivious approaches, i.e. Best-Fit(BF) and Worst-

Fit(WF). Especially, WF is a commonly used energy optimization heuristic and has been shown to be quite effective in the absence of processor faults due to its load-balancing characteristic [4]. To maintain the feasibility under the K faults for both BF and WF, the reserved resource on each core was considered as part of the workload, and different tasks can share the reserved resource. BF(WF) allocates a task to a feasible core with the least(most) remaining capacity. Individual optimal number of checkpoints was inserted to each task under these two heuristics. We then evaluate how many speedups that $EATA$ can achieve with the techniques proposed in Section 4.3 to prune the search space of OPT_CHK. To evaluate the energy saving performance, we set up the simulation platforms as follows. For a fixed number (m) of cores, we varied the average utilization, i.e. $\frac{U_{total}}{m}$ from 0.1(light load) to 1 (heavy load). The utilization of each task τ_i was uniformly distributed in the range $[0.01, 0.6]$. The deadline of the application, i.e. D, was set to 100. The fault detection, checkpointing and state retrieval overhead was identically set to 0.5, 1 and 1 respectively for each task. The corresponding energy overhead was set to 0.05, 0.1 and 0.1. In addition, we set $P_{leak} = 0.1$, $C_{ef} = 1$ and $\alpha = 3$ and we assumed the existence of four normalized frequency levels given by $\{0.4, 0.6, 0.8, 1.0\}$.

Due to page limits, we only show three sets of experimental results with different numbers of tasks, cores and total transient faults. Figure 4.2 shows the energy consumption for 20 tasks and 4-core processors with K=1. Each point in the figure was averaged over 1000 test cases. As we can see, the energy consumption increases when the system workload becomes heavier for all three techniques, but our approach EATA always outperforms the other two. For instance, when the core average utilization is 0.55, 12%(46%) energy saving is achieved by EATA over WF(BF). In average, our algorithm reduces energy consumption by 11% (59%) compared to WF (BF).

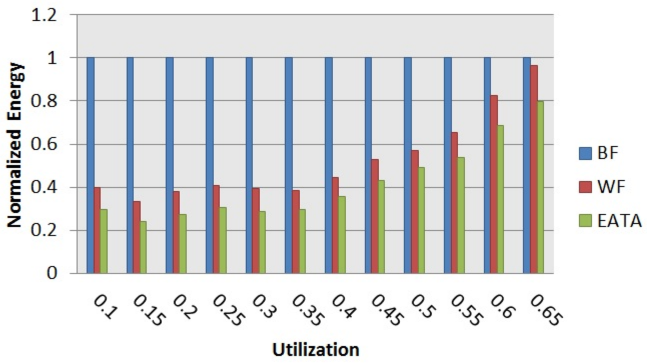


Figure 4.2: 20 tasks on a 4-core processor, K = 1

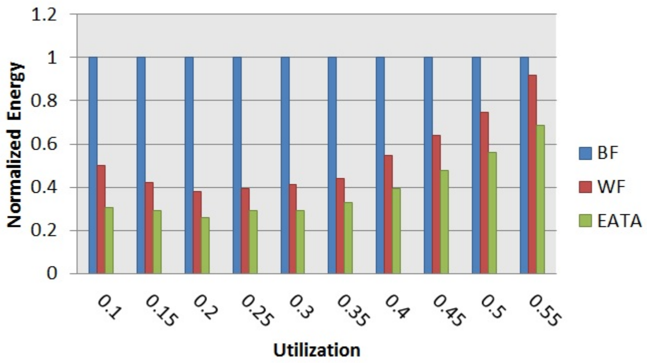


Figure 4.3: 40 tasks on a 8-core processor, K = 2

The energy savings are more substantial in Figure 4.3, with 8-core processors and 40 tasks to tolerate maximum 2 faults, with over 16% and 62% energy savings in average compared to WF and BF, respectively.

Similar results are observed for the case of 16-core processors and 80 tasks with at most 4 faults as shown in Figure 4.4, where 19% and 65% energy savings are achieved over WF and BF respectively. In general, we can see that our approach can achieve better energy savings for test cases with higher system utilizations, larger numbers of tasks and cores. This is due to the fact that our approach tries to find the best combination of task allocation, checkpointing scheme and speed assignment at each

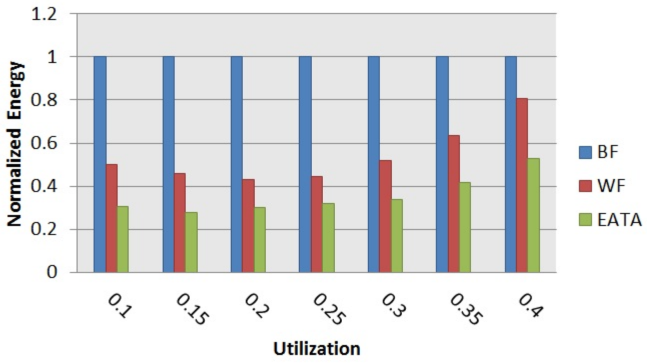


Figure 4.4: 80 tasks on a 16-core processor, K = 4

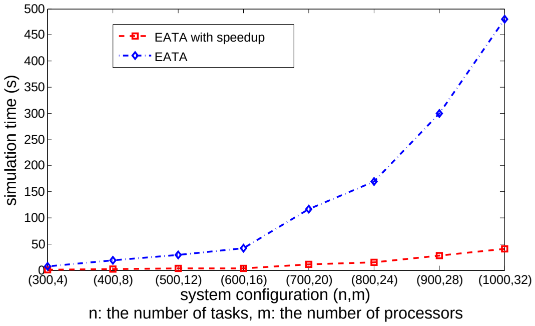


Figure 4.5: Performance of two speed-up techniques

step. High energy savings are achieved by reserving as less resources as possible and leaving more slacks for DVFS.

Next, we evaluated the benefits of our approach proposed in Section 4.3. The complexity of EATA heavily depends on that of OPT_CHK. Therefore, the computational efficiency of OPT_CHK is critical to the success of EATA. To study the computational efficiency of $EATA$ brought by the speedup techniques for OPT_CHK, we set the average utilization, i.e. $\frac{U_{total}}{m}$ to be 0.8. The utilization of each task was randomly generated to be uniformly distributed in [0.01, 0.06]. The deadline, i.e. D was set to 100. The timing overhead of checkpointing, fault detection and state retrieval were considered as 1% of the average task execution time. We varied the numbers of tasks and cores and recorded the results in Figure 4.5. In each step, we increase the number of tasks by 100 and the number of cores by 4. As we can see, as the system size grows, the time consumed by both simulations increase. However, our approach proposed in section 4.3 can easily achieve a speed up of at least 10X. As the number of tasks and cores increases, the efficiency of the two speed-up technique becomes more prominent and make the algorithm EATA efficiently scalable.

4.6 Summary

As IC technology continues its evolution into the deep sub-micron domain, the exponentially increased energy consumption and the deteriorated reliability have become serious concerns in computer system design. In this chapter, we study the energy minimization problem for a real-time application on a multi-core platform that can tolerate K transient faults using the checkpointing method. We first develop an efficient method to determine the checkpointing scheme that can minimize the worst case response time for a task set that shares the reserved resources for fault recovery on a single-core processor. We then present a task assignment algorithm to minimize the overall energy while guaranteeing the fault-tolerance capability. Our experimental results also demonstrate the effectiveness and efficiency of our proposed approach.

CHAPTER 5

ENERGY MINIMIZATION FOR FAULT-TOLERANT SCHEDULING OF PERIODIC FIXED-PRIORITY APPLICATION ON MULTI-CORE PLATFORMS

Despite the fact that the techniques proposed in the previous chapter can achieve significant energy savings while enabling the system to tolerate transient faults. It can only be applied to frame-based tasks, i.e. all tasks share the same deadline. In this chapter, we study the problem of energy minimization for scheduling general periodic fixed-priority applications on multi-core platforms with fault-tolerance requirements. Specifically, We first introduce an efficient method to determine the checkpointing scheme that guarantees the schedulability of an application under the worst-case scenario, i.e. up to K faults occur, on a single-core processor. Based on this method, we then present a task allocation scheme aiming at minimizing energy consumption while ensuring the fault-tolerance requirement of the system.

The rest of this chapter is organized as follows. Existing works that are related to our research problem are discussed in Section 5.1. Section 5.2 introduces the system models and notations used throughout this chapter. We introduce our efficient algorithm for obtaining a feasible checkpointing solution for a given task set on a single-core processor in Section 5.3. We then present our energy efficient fault-tolerant task-allocation algorithm in section 5.4. The effectiveness and efficiency of our algorithms are evaluated in Section 5.5. Finally, section 5.6 summarizes the chapter.

5.1 Related works

When dealing with both energy conservation and fault tolerance, one big challenge is how to balance the resource usage between the two, since energy conservation strategies need additional resources for lowering down system speed, and fault tolerance strategies need additional resources for fault detection and recovery.

There are also several papers published that are closely related to our research. Pop et al. [97] presented a constraint logic programming method to develop fault-tolerant DVFS schedules for real-time tasks with precedence constraints on distributed heterogeneous platforms. The task allocation is assumed to be known a priori. Fault tolerance is achieved by reserving passive backup(s) for a task on the same core and activating it in case of failure. With the slacks mostly being occupied by reserved recoveries, the space for DVFS is severely limited. Haque et al. [60] proposed a stand-sparing technique for fixed-priority applications on a dual-core platform. Active replication with delayed starting time is employed for the purpose of maintaining task reliability and reducing energy consumption. Again, an entire task needs to be re-executed in presence of a failure and active replication can consume extra energy even under fault-free scenario. Han et al [56] proposed an optimal checkpointing scheme for minimize the worst case response time of an application on a single-core processor and developed a task allocation scheme for energy minimization. However, this approach is limited to frame-based task sets, hence it does not apply to a much more complicated fixed-priority periodic task model.

In this chapter, we study the problem of minimizing the energy consumption for periodic fixed-priority hard real-time systems running on homogeneous multi-core platforms while ensuring that the systems can tolerate up to K transient faults. We adopt the widely used DVFS and checkpointing as the energy management method and the fault-tolerance policy, respectively. We focus our efforts on fixed-priority scheduling due to its simpler implementation and better practicability [39] compared with dynamic priority-based scheduling. In what follows, we introduce some preliminaries that are essential to our research.

5.2 Preliminaries

In this section, we introduce the pertinent background of our research problem and present the system models and notations.

5.2.1 Application model

The real-time application considered in this chapter consists of n independent sporadic tasks, denoted as $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$. Each task is characterized by a tuple (C_i, D_i, T_i). C_i denotes the worst-case execution time of a task τ_i, whereas D_i and T_i represent its deadline and minimum inter-arrival time (period), respectively. Each task can generate an infinite number of instances or jobs, we use these two terms interchangeably in this chapter. The utilization of task τ_i is represented as $u_i = \frac{C_i}{T_i}$. The system utilization U is therefore calculated as $U_{total} = \sum_{i=1}^{n} \frac{C_i}{T_i}$.

5.2.2 Fault model and checkpointing

In this chapter, we assume that there are at most K transient faults within one least common multiple (LCM) of all the task periods in Γ but we do not make any assumptions regarding the fault pattern. In other words, the transient faults can strike any task instance at any time, and multiple faults may affect the same task instance. Once a fault is detected, the task instance being affected rolls back to the last saved checkpoint and re-executes the faulty segment. We consider the checkpoint to be self fault-tolerant.

We assume all the jobs of the same task have the identical number of checkpoints. Inserting one checkpoint to an instance of task τ_i refers to the operation of saving its current state and condition to memory, with its the timing and energy overhead denoted as o_i and eo_i, respectively. Before inserting a checkpoint, a fault detection is always performed to ensure the sanity of the to-be-saved state. We use q_i and eq_i to denote the timing and energy overhead for such an operation. Moreover, once a fault is detected during the execution of an instance of task τ_i, it needs to rollback to the latest checkpoint, i.e. to retrieve the latest-saved correct information. The time and energy overhead of this operation are represented by r_i and er_i, respectively.

The fault-free execution time of an instance of task τ_i is a function of the number of checkpoints, and is formulated in equation (5.1a). Note that, with m_i checkpoints, the fault detections are performed $m_i + 1$ times including the one at the end of the job's execution. The recovery time of τ_i with m_i checkpoints under a single failure includes three parts, namely the time to rollback to the latest checkpoint, the time to re-execute the faulty segment and the time to perform a fault detection operation at the end. We denote it as $F_i(m_i)$ and formulate it in equation (5.1b).

$$C_i(m_i) = C_i + m_i o_i + (m_i + 1)q_i \tag{5.1a}$$

$$F_i(m_i) = r_i + \frac{C_i}{m_i + 1} + q_i \tag{5.1b}$$

Since a lower priority task τ_i is subject to the workload interference (including recoveries) from higher priorities tasks, the worst case recovery time for τ_i is the longest recovery time among all tasks with higher priority and τ_i itself. Specifically, we denote it as

$$MR_i = \max(F_1, F_2, ..., F_i). \tag{5.2}$$

Regarding $\tau_i's$ schedulability, adding more checkpoints to its higher priorities tasks increases the interference caused by fault-free workloads, which may undermine $\tau_i's$ schedulability. However, it may decrease the recovery time needed for τ_i, i.e. MR_i, which is in favor of $\tau_i's$ schedulability. Therefore, to determine the appropriate number of checkpoints for scheduling real-time tasks under the fault tolerance constraint is not a trivial task.

5.2.3 Platform and energy model

We assume that there are a total number of ϕ cores on a homogeneous multi-core platform Ψ, i.e. $\Psi = \{\psi_1, ..., \psi_\phi\}$ and there exist a set of L-level discrete speeds/frequencies for each core, which is denoted as $FR = \{f_1, f_2, ..., f_L\}$. Without loss of generality, we assume $0 \leq f_L \leq f_{L-1} \leq ... \leq f_1 = 1$.

We adopt the power model in [56, 133] by considering the frequency-independent and frequency-dependent power components. Specifically, the overall power consumption P can be formulated as

$$P = P_{ind} + P_{dep} = P_{ind} + C_{ef} f^{\alpha}, \tag{5.3}$$

where P_{ind} is the frequency-independent power, including the power consumed by off-chip devices such as main memory and external devices and constant leakage power. C_{ef} is the effective switching capacitance. α is a constant usually no smaller than 2. P_{dyn} is the dynamic power consumed by switching transistor state. As a result, the fault-free energy consumption of a job from task τ_i with m_i checkpoints executed under speed f_i is calculated as:

$$\begin{aligned} E_i(f_i) = {} & (P_{ind} + C_{ef} f_i^{\alpha}) \cdot \frac{C_i}{f_i} \\ & + m_i(eo_i + o_i P_{ind}) + (m_i + 1)(eq_i + q_i P_{ind}), \end{aligned} \tag{5.4}$$

where the first part is the energy consumed by executing the job (the scaled execution time of task τ_i under frequency f_i is $\frac{C_i}{f_i}$), and the second and the third part represent the energy overhead from checkpointing and fault detections, respectively. Similar to [130, 56], we assume that checkpointing, fault detections and checkpoint retrievals are not affected by processing frequency. Note that, during those operations, the frequency-independent power is still consumed. As $E_i(f_i)$ is a convex function, one intuition to save energy is to lower the operating frequency as much as possible, provided it is larger than so-called *critical frequency* ($f_c = \sqrt[\alpha]{\frac{P_{ind}}{(\alpha-1)C_{ef}}}$) [88].

Γ_j is used to denote the set of tasks assigned to the core ψ_j. As we only study the energy consumption within one LCM of the task periods, the energy consumption of core ψ_j is formulated in equation (5.5).

$$E(\Gamma_j) = \sum_{\tau_i \in \Gamma_j} \frac{LCM}{T_i} E_i(f_i). \tag{5.5}$$

The total energy consumption of the system is thus $E(\Gamma) = \sum_{j=1}^{\phi} E(\Gamma_j)$.

5.3 Feasible checkpointing configuration for fixed-priority tasks on a single-core processor

Our goal is to minimize the energy consumption while being able to tolerate, in the worst case, K faults when scheduling a fixed-priority task set on a multi-core platform. One key to this problem is to choose an appropriate number of checkpoints for each task. Adding more checkpoints to tasks may reduce the recovery overheads, which is in favor of system schedulability. However, excessive checkpointing overheads may outweigh the benefits of decreasing recovery overheads, which might undermine the schedulability of the system. Therefore, to determine the number of checkpoints for each task is not a trivial problem and must be carefully studied.

As a closely related work, Zhang et al. [130] showed that the optimal number of checkpoints to minimize the worst case latency of a single task τ_i, denoted as m_i^*, can be calculated as

$$m_i^* = \begin{cases} \lceil \sqrt{\frac{K*c_i}{o_i+q_i}} - 1 \rceil & \text{if } c_i > \frac{(m_i^-+1)(m_i^-+2)(o_i+q_i)}{K} \\ \lfloor \sqrt{\frac{K*c_i}{o_i+q_i}} - 1 \rfloor & \text{if } c_i \leq \frac{(m_i^-+1)(m_i^-+2)(o_i+q_i)}{K} \end{cases}$$

where $m_i^- = \lfloor \sqrt{\frac{K*c_i}{o_i+q_i}} - 1 \rfloor$. However, when considering multiple fixed-priority tasks on a single-core processor, the individual optimal checkpointing configuration does not necessarily lead to a feasible checkpointing configuration for a task set.

To this end, Zhang et al. [130] proposed a recursive approach for identifying a feasible checkpointing scheme for a given fixed-priority task set on a single-core processor. Specifically, the recursive algorithm, i.e. (ZCP(p,q)), takes two parameters p and q as inputs, where p and q are the indexes for the first and last task in the sub-task set with checkpoint numbers to be determined. The algorithm works as follows:

1. Initially, let $m_i = 0$ and obtain m_i^* for $1 \leq i \leq n$. Set $p = 1, q = n$.
2. ZCP(p,q): Starting from the first task τ_p, evaluates the schedulability of each task in decreasing order of task priorities, and finishes successfully if all tasks are determined schedulable.
3. If task $\tau_j, j \in [p, q]$ is not schedulable, the task $\tau_h, h \in [1, j]$ with the longest recovery is found and one more checkpoint is added to it, i.e. $m_h = m_h + 1$ to reduce its recovery time, i.e. $F_h = F_h(m_h)$. Since the addition of checkpoints to τ_h affects the schedulability of the tasks from τ_h to τ_j, we need to set $p = h, q = j$ and recursively call ZCP(p,q).
4. ZCP(p,q) terminates and reports that the task set is unschedulable if, for each task $\tau_i, i \in [1, p]$, the number of checkpoints is larger than m_i^*, i.e. the optimal value for a single task.

This approach works well only for small task sets and/or tasks with small optimal checkpoint numbers. Otherwise, it can be extremely time consuming. Note that, a task τ_i is considered unschedulable only when the checkpoint numbers of all tasks in $\{\tau_1, \tau_2, ..., \tau_i\}$ exceed their individual optimal numbers. In addition, each time when a checkpoint is added to a task τ_i, the schedulability of task τ_i along with all the lower-priority tasks has to be re-evaluated. These two factors contribute the most to the excessive running time of ZCP and make it extremely computational expensive for design space explorations for our multi-core energy-efficient fault-tolerant real-time scheduling problem, which is NP-hard in nature.

It is therefore desirable that a more efficient and effective method can be developed to rapidly determine the checkpointing configuration for tasks on a single core. In what follows, we introduce several theorems, and based on which, we develop a much more efficient algorithm.

Theorem 5.3.1. *Given a checkpointing configuration $M = \{m_1, ..., m_p, ..., m_n\}$, assume that there exists a task τ_p with $m_p > m_p^*$. Let $M' = \{m_1, ..., m_p^*, ..., m_n\}$. Then if the task set Γ is unschedulable under M', it must also be unschedulable under M.*

Proof. Reducing the number of checkpoints of τ_p does not affect the schedulability of the tasks with higher priorities than τ_p. We only need to consider each task $\tau_i, i \in [p, n]$. To ease our proof, we use $W_*(t)$ and $W'_*(t)$ to denote the total workload demand before t for task τ_* under the checkpointing scheme M and M', respectively.

Since τ_i is schedulable under M, then there must exist at least one time instance $t \in [0, D_i]$ such that

$$W_i(t) = \sum_{j=1}^{i} \lceil \frac{t}{T_j} \rceil C_j(m_j) + K \times MR_i \leq t. \tag{5.6}$$

Let MR'_i denote the longest recovery overhead affecting τ_i under M'. Since the number of checkpoints in M is no smaller than that in M', according to equation (5.1b) and (5.2), it is clear that $MR'_i \geq MR_i$. We consider two cases separately.

Case 1:$MR'_i = MR_i$. In this case, we have

$$\begin{aligned} W'_i(t) = & \sum_{j=1, j \neq p}^{i} \lceil \frac{t}{T_j} \rceil C_j(m_j) \\ & + \lceil \frac{t}{T_p} \rceil C_p(m^*_p) + K \times MR'_i \\ < & W_i(t) \leq t, \end{aligned} \tag{5.7}$$

as a result, τ_i must be schedulable.

Case 2:$MR'_i > MR_i$. In this case, reducing m_p to m^*_p leads to an increase in fault recovery overhead for τ_i. In other words, τ_p become the task with the longest recovery, i.e. $MR'_i = F_p(m^*_p)$. Let $\kappa_i = o_i + q_i, \forall i$. Additionally, we know that $MR_i \geq F_p(m_p)$ and $m_p\kappa_p + K \times F_p(m_p) \geq m^*_p\kappa_p + K \times F_p(m^*_p)$ because m^*_p is the optimal number of checkpoints for τ_p when considered individually. Consequently, let $\delta = W_i(t) - W'_i(t)$, we have

$$\begin{aligned} \delta = & \lceil \frac{t}{T_p} \rceil (m_p\kappa_p - m^*_p\kappa_p) \\ & + K \times MR_i - K \times MR'_i \\ \geq & m_p\kappa_p - m^*_p\kappa_p + K \times MR_i - K \times F_p(m^*_p) \\ \geq & m_p\kappa_p + K \times F_p(m_p) - (m^*_p\kappa_p + K \times F_p(m^*_p)) \\ \geq & 0; \end{aligned}$$

Thus, $W'_i(t) \leq W_i(t) \leq t$ and τ_i must be schedulable. □

Theorem 5.3.1 implies that, if the task set is not schedulable when the number of checkpoints of any task has already exceeded its individual optimal number, this task set is deemed to be unschedulable. As a result, there is no need to increase the numbers of checkpoints for other tasks until all of them exceed their individual optimal numbers, as in ZCP algorithm stated above. With larger task sets and larger optimal checkpoint numbers for each tasks, Theorem 1 can improve the computational efficiency tremendously.

In addition, changing the number of checkpoints of a higher priority task also changes its preemption impacts to the low priority tasks and thus results in time-consuming schedulability checking operations. The following theorem helps to greatly reduce the computational cost for schedulability checking.

Theorem 5.3.2. *Let τ_q be the unschedulable task with the highest priority under the checkpointing configuration $M = \{m_1, ..., m_q, ..., m_n\}$. Assume that τ_q becomes schedulable under a new configuration $M' = \{m'_1, ..., m'_q, ..., m'_n\}$, $\forall i, m'_i \geq m_i$ when gradually adding checkpoints to tasks with the largest recovery cost. Then, for any higher priority task τ_i, where $i \in [1, q)$, if it is schedulable under M then it must be schedulable under the new configuration M'.*

Proof. If checkpoints are increased only for tasks with priorities lower than τ_i, i.e. $\forall j, j \in [1, i], m_j = m'_j$, then τ_i's schedulability is not affected.

Now consider the case where there are checkpoints added to tasks with priorities higher than τ_i. Let the checkpoint configuration before increasing the checkpoints of τ_i and any task with higher priorities than τ_i be $\tilde{M} = \{\tilde{m}_1, \tilde{m}_2, ..., \tilde{m}_n\}$. We have $m_j = \tilde{m}_j, j = 1, ..., i$ and $m_j \leq \tilde{m}_j \leq m'_j, j = i + 1, ..., q$, and $\tilde{MR}_i = \tilde{MR}_q$, where $\tilde{MR}_i$ and $\tilde{MR}_q$ are the longest recovery overhead under the checkpointing scheme $\tilde{M}$ for τ_i and τ_q, respectively. Note that τ_q is not schedulable under $\tilde{M}$ but schedulable under M'.

For ease of our proof, we let $W'_*(t)$ and $\tilde{W_*}(t)$ denote the total workloads before t for task τ_* under the scheme M' and $\tilde{M}$, respectively.

We prove this theorem by contradiction. We assume that τ_i is schedulable under M but not schedulable under M'. Let MR_i' be the longest recovery overhead for τ_i under M'. Then, with $t_1 \in [0, D_i]$ we have

$$W_i'(t_1) = \sum_{j=1}^{i} \lceil \frac{t_1}{T_j} \rceil C_j(m_j') + K \times MR_i' > t_1. \tag{5.8}$$

Since τ_i is schedulable under $\tilde{M}$ and without loss of generality, we let

$$W_i\tilde{}(t_1) = \sum_{j=1}^{i} \lceil \frac{t_1}{T_j} \rceil C_j(\tilde{m}_j) + K \times M\tilde{R}_i \leq t_1. \tag{5.9}$$

Subtracting equation (5.9) from equation (5.8) and letting $H_*(t) = W_*'(t) - W_*\tilde{}(t)$ and $\kappa_i = o_i + q_i, \forall i$, we obtain the following result

$$H_i(t_1) = \sum_{j=1}^{i} \lceil \frac{t_1}{T_j} \rceil (m_j' - \tilde{m}_j)\kappa_j + K \times MR_i' - K \times M\tilde{R}_i > 0 \tag{5.10}$$

Similarly, because τ_q is schedulable under M', we know

$$W_q'(t_2) = \sum_{j=1}^{q} \lceil \frac{t_2}{T_j} \rceil C_j(m_j') + K \times MR_q' \leq t_2, \tag{5.11}$$

where t_2 is an time instance in $[0, D_q]$ and MR_q' denotes the longest recovery for τ_q under M'.

In what follows, we consider two cases regarding t_2.

Case 1: $t_2 \leq D_i$. In this case, it can be seen that $W_i'(t_2) \leq W_q'(t_2) \leq t_2$, since $i < q$ and $MR_i' \leq MR_q'$. Therefore, τ_i must be schedulable under M', which is contradictory to our assumption.

Case 2: $t_2 > D_i$. As a result, we have $t_2 > t_1$. Furthermore, since τ_q is not schedulable under the scheme $\tilde{M}$, we have the following condition,

$$\sum_{j=1}^{q} \lceil \frac{t_2}{T_j} \rceil C_j(\tilde{m}_j) + K \times M\tilde{R}_q > t_2. \tag{5.12}$$

Subtracting equation (5.12) from equation (5.11), we have

$$H_q(t_2) = \sum_{j=1}^{q} \lceil \frac{t_2}{T_j} \rceil (m'_j - \tilde{m_j})\kappa_j + K \times MR'_q - K \times \tilde{MR}_q \leq 0 \tag{5.13}$$

Since $i < q$, $MR'_i \leq MR'_q$ and $\tilde{MR}_i = \tilde{MR}_q$, the following contradiction can be readily derived,

$$0 < H_i(t_1) \leq H_q(t_2) \leq 0. \tag{5.14}$$

Thus far, this theorem is proved. □

According to Theorem 2, for the first task τ_q that misses its deadline under a checkpoint scheme, if we are able to incrementally add checkpoints to its higher-priority tasks or itself to make it schedulable, all tasks with priorities higher than τ_q are guaranteed to be schedulable. This theorem can eliminate the computational efforts for re-evaluating the schedulability of higher priority tasks when inserting the checkpoints to them. Based on Theorem 5.3.1 and 5.3.2, we formulate an efficient and effective algorithm for finding a feasible checkpointing configuration for fixed-priority tasks on a single core, as shown in Algorithm 5.

ECHK evaluates the schedulability of each task from the highest priority to the lowest. If an unschedulable task τ_i is encountered, ECHK searches for the checkpointing configuration to make τ_i schedulable by repeatedly adding checkpoints to a higher priority task or τ_i that currently contributes the most to $\tau_i's$ recovery, an termination condition is set according to Theorem 5.3.1. If an feasible checkpointing configuration is found, then the schedulability of all the tasks with higher priorities than τ_i is guaranteed based on Theorem 5.3.2.

Algorithm 5 greatly simplifies the process of searching for a feasible checkpointing combination for a given task set on a single core. The complexity of ZCP is $O(\prod_{i=1}^{n} m_i^* \cdot nT)$, where T is the longest time for evaluating the schedulability of a

Algorithm 5 ECHK(Γ, K)

```
Require:
    1) Task set : $\Gamma = \{\tau_1, \tau_2, ...\tau_n\}$;
    2) Number of faults: $K$
flag = “task set schedulable”
obtain $m_i^*$, for $i = 1, 2, ..., n$ according to [96]
$\forall i, i =, 1, 2..., n$, initialize $m_i$ to 0;
for $(i = 1; i < n + 1; i + +)$ do
    while $\tau_i$ is not feasible do
        $F_h = \max(F_1, ..., F_i)$;
        $m_h = m_h + 1$;
        if $m_h > m_h^*$ then
            $flag$ = “task set unschedulable”
            return $flag$
        end if
    end while
end for
return $flag$, $M = \{m_1, m_2, ..., m_n\}$
```

task using exact response time analysis, whereas our ECHK has a complexity of at most $O(\sum_{i=1}^{n} m_i^* \cdot nT)$. Moreover, given that our algorithm can determine a task set to be unschedulable as soon as the number of checkpoints of any task exceeds its individual optimal value, ECHK is much more efficient in practice.

5.4 Energy-aware task allocation

Based on our algorithm ECHK that guarantees the single-core fault tolerance, we now present an algorithm determining the task allocation and the corresponding DVFS schedule on multi-core platforms to minimize the overall energy consumption.

Without the fault tolerance requirement, one intuitive method is to balance the workload among multi-core platforms as much as possible [4] such that each core can run at a relatively low speed. When we take fault tolerance into account, however, extra care must be taken since both recovery reservation and energy management compete for system resources. The amount of reserved resources heavily depends on the feasible checkpointing scheme that can be obtained for a given task set. Balancing the workload does not necessarily leads to a favorable checkpointing scheme, since the system utilization itself does not provide any information regarding the fault-tolerant schedulability of a task set. On the other hand, packing as many tasks as possible into one core helps to reduce the number of cores to be utilized, but leaves less space for slowing down the processing core.

In what follows, we focus our effort on developing an effective heuristic for jointly determining the task allocation, checkpointing configuration and DVFS schedule for fixed-priority task sets scheduled on multi-core platforms, as it is a NP-Hard problem in strong sense [96].

Our task allocation scheme for energy minimization with K-fault tolerance capability is developed based on the algorithm $ECHK$. The overall algorithm is described in Algorithm 6. Specifically, when allocating a new task τ_i, we tentatively assign τ_i to each core and determine whether a feasible checkpointing can be obtained. For each feasible candidate core ψ_j, we search for the lowest constant speed that can guarantee the schedulability of all the tasks assigned to it according to Algorithm 7. As excessive frequency switching can cause significant overhead, we use a constant speed for each core. Then, τ_i is allocated to the core with the lowest possible speed among all the feasible candidates.

In Algorithm 7, we reduce the speed of a core one level at a time until the lowest speed that yields a feasible checkpointing scheme is reached. Therefore, the complexity of our algorithm greatly hinges on that of ECHK. We assume that the re-execution of a faulty task is always performed at the highest speed, given the probability of failure is low. The checkpointing overhead is considered independent of the core's running mode.

Algorithm 6 TACHK(Γ, Ψ, K)

```
$\Gamma_j$ = NULL, for $j = 1, 2, ..., \phi$;
for $i = 1$; $i \leq n$; $i++$ do
    $feasible\_speed_i = f_{max}$;
    assigned = 0;
    for $j = 1$; $j \leq \phi$; $j++$ do
        $\{flag, M_{temp}\}$ = ECHK($\Gamma_j \cup \tau_i$, K);
        if ($!flag$) then
            continue;
        end if
        $speed_{temp} = determine\_core\_speed(\Gamma_j \cup \tau_i, K)$;
        if $speed_{temp} < feasible\_speed$ then
            assigned = j;$feasible\_speed_i = speed_{temp}$;
        end if
    end for
    if  $assigned == 0$ then
        return "not schedulable";
    else
        $\Gamma_{assigned} \leftarrow \Gamma_{assigned} \cup \{\tau_i\}$;
    end if
end for
calculate the energy consumption $E_{total}$ according to equation (5.4) and (5.5);
return  $\{\Gamma_1, ..., \Gamma_\phi\}, E_{total}$
```

Algorithm 7 *determine_core_speed*(Γ, K)

```
$lowest\_feasible\_speed = f_{max}$;
sort the available discrete speeds of the cores,i.e. $FR$ in decreasing order;
for $i = 1$; $i \leq |FR|$; $i++$ do
    $\Gamma_{temp}$: temporary task set resulting from $\Gamma$ scaled by frequency FR[i];
    flag = ECHK($\Gamma_{temp}$, K);
    if ($!flag$) then
        break;
    else
        $lowest\_feasible\_speed = FR[i]$;
    end if
end for
return  $lowest\_feasible\_speed$
```

It is not difficult to see that the overall complexity of Algorithm 7 is $O(nL\sum_{i=1}^{n} m_i^* \cdot T)$, where L is the number of available processor frequencies and T is the longest time for evaluating the schedulability of a task using exact response time analysis. Furthermore, the complexity of Algorithm 6 is $O(n^2 \cdot \phi \cdot L\sum_{i=1}^{n} m_i^* \cdot T)$.

5.5 Experimental results

In this section, we use simulations to verify the effectiveness and efficiency of our proposed algorithms.

5.5.1 Timing complexity evaluation

Firstly, we evaluate the timing complexity of our algorithm $ECHK$ against the method proposed in [130], i.e. ZCP, on a single-core platform.

We set the system utilization to be 0.8. Note that, we fixed the system utilization to a high value such that the task set generated was not schedulable under faults without checkpointing. The period of each task was randomly selected in the range [10,1000]. The rest of the task parameters were generated according to UUNIFAST in [17]. In our experiments, ZCP can easily fail even with a small number of task when the execution ratio, i.e. $\frac{C_{max}}{C_{min}}$ is very large (e.g. > 100), where C_{max} and C_{min} are the longest and shortest execution time in the task set, respectively. This is due to the fact that it may keep adding checkpoints to the task with a number of checkpoints already larger than its optimal value and thus incurs unnecessary recursions. Therefore, we first modified the ZCP according to Theorem 5.3.1. The running times of ECHK and ZCP greatly rely on the following three factors: the number of tasks, checkpointing overhead and the number of faults. We conducted

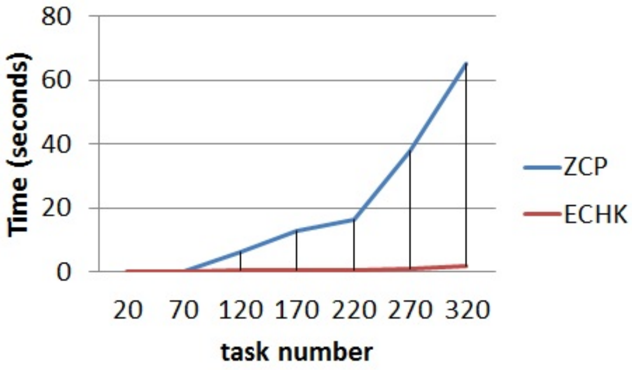


Figure 5.1: Varying the number of tasks

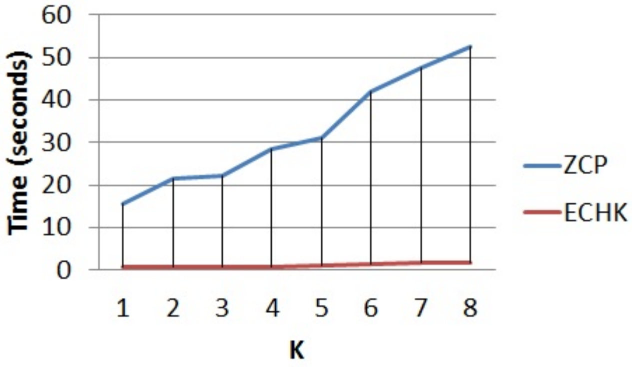


Figure 5.2: Varying the number of tasks

experiments regarding each factor and recorded the results as shown in the following figures. The result of each test case is the average from over 1000 task sets.

In Figure 5.1, we set $K = 2$ and the checkpointing overhead of each task τ_i as 3% of its worst case execution time, i.e. C_i. The number of tasks was varied from 20 to 320 with a step of 50. As can be seen from the figure, our approach ECHK significantly outperforms the method ZCP. ECHK can achieve a speedup with the maximum of 38X and 20X in average.

Next we evaluated the impact of increasing K on running time of ECHK and ZCP, respectively. The number of tasks was set to 200 and the checkpointing overhead of each task τ_i was fixed at 3% of its worst case execution time, i.e. C_i. As expected,

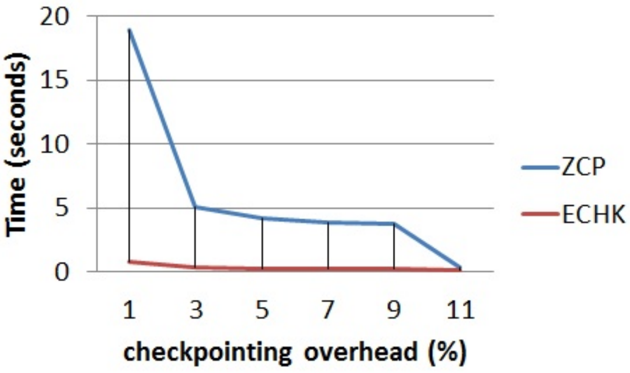


Figure 5.3: Varying checkpoint overhead

our ECHK performs much better in terms of timing complexity. In this case, ECHK can achieve a maximum speedup of 37X and an average speedup of 30X.

Finally, we studied the effects of increasing checkpointing overhead. The checkpointing overhead was varied from 1% to 11% of the worst case execution of each task, and the numbers of faults and tasks were set to 2 and 200, respectively. As shown in Figure 5.3, when the checkpointing overhead increases, the individual optimal number of checkpoints for each task decreases, hence the search space becomes smaller. While running time of both algorithms decrease, our algorithm can achieve a speedup of 16X in average.

In conclusion, our algorithm ECHK is significantly more efficient than ZCP and more scalable in terms of task numbers, the number of faults and checkpoint overhead.

5.5.2 Energy performance evaluation

Next, we evaluated the effectiveness of our algorithm $TACHK$.

To our best knowledge, there is no existing approach that solves the exact same problem. Therefore, we evaluated our algorithm against two widely used fault-oblivious approaches, i.e. Best-Fit (BF) and Worst-Fit (WF). In particular, WF is well-known for its effectiveness in fault-oblivious energy reduction as it balances the workload among different cores [4].

To make BF and WF fault-tolerant, we propose a two-step approach. The first step is to identify a feasible task allocation solution. Similar to our approach TACHK, we tentatively allocate the current task to each core. We use ECHK to check if a core is a feasible candidate. BF (WF) allocates a task to a feasible core with the least (most) remaining capacity, i.e. the spare utilization. After obtaining a feasible allocation solution, Algorithm 7 is used to find the lowest constant speed for each core and then the total energy consumption is calculated. The energy consumptions of $TACHK$ and WF are normalized with respect to that of BF.

To evaluate the energy saving performance, we set up the simulation platform as follows. For a fixed number (ϕ) of cores, we varied the average utilization, i.e. $\frac{U_{total}}{\phi}$ from 0.2 (light load) to 0.8 (heavy load). The period of each task τ_i was uniformly distributed in the range $[10, 1000]$. The rest of task parameters were generated according to UUNIFAST [17]. The fault detection, checkpointing and state retrieval overhead were identically set to 1%, 3% and 3% respectively for each task. The corresponding energy overheads were set to 1%, 3% and 3% of the dynamic energy under f_{max} for each task. In addition, we set $P_{ind} = 0.1$, $C_{ef} = 1$ and $\alpha = 3$ [56] and we assumed the existence of the normalized frequency in the range of [0.2, 1] with a step of 0.05.

We present three sets of experimental results with various numbers of tasks, cores and total transient faults. Each value reported in the figure is averaged over 1000 test cases. Figure 5.4 shows the energy consumption for a 4-core processor with 40 tasks and K=2. We can see the energy consumption increases for all three techniques as the system workload becomes heavier, but our approach TACHK always outperforms

Figure 5.4: 40 tasks on 4-core processors, K = 2

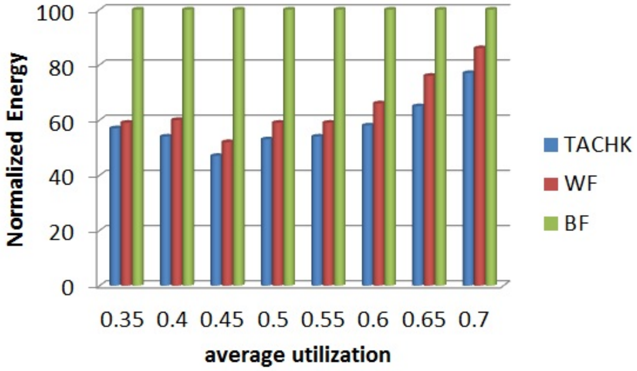


Figure 5.5: 80 tasks on 8-core processors, K = 5

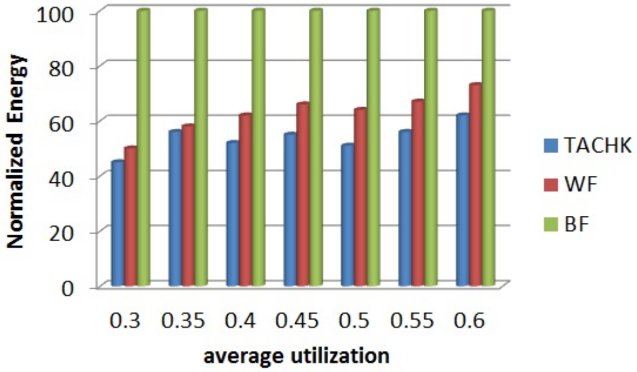
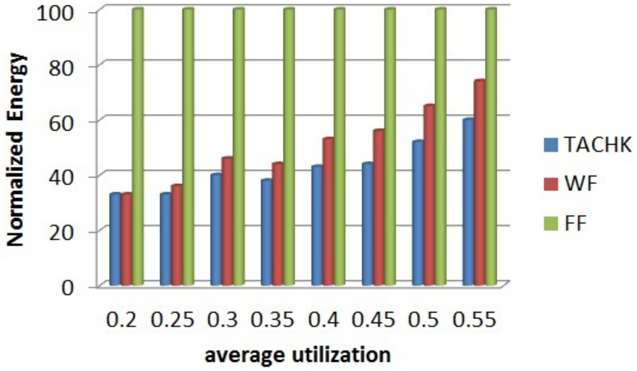


Figure 5.6: 160 tasks on 16-core processors, K = 10



the other two. For instance, when the processor average utilization is 0.65, 12%(34%) energy saving is achieved by TACHK over WF (BF). Our algorithm achieves a energy reduction of 7.5% (38.4%) in average when comparing with WF (BF). The energy savings are more substantial in Figure 5.5, on a 8-core processor with 80 tasks that can tolerate 5 faults,

TACHK in average saves 10% and 46% energy over WF and BF, respectively. Similarly, for the case of a 16-core system with 160 tasks that can tolerate at most 10 faults as shown in Figure 5.6, 13% and 59% energy savings are achieved over WF and BF, respectively. In general, we can see that our approach becomes more effective when system utilizations and/or the number of tasks/cores become larger. The reason is that at each step, our approach TACHK tries to determine the best combination of task allocation, checkpointing configuration and speed assignment.

5.6 Summary

With relentless technology scaling and mass integration of transistors into a single chip, the exponentially increased power consumption and the severely degraded reliability have become first-class design issues in modern computing systems. In this chapter, we study the energy minimization problem for hard real-time fixed-priority systems running on multi-core platforms that can tolerate up to K transient faults. We propose a solution to this problem by jointly considering the task allocation, checkpoint configuration and speed assignment. We first develop an efficient method to judiciously determine the checkpointing scheme that can guarantee the schedulability of a task set on a single-core processor. From our theoretical analysis and simulation results, we can see that this algorithm is much more efficient than the state-of-art technique. We then present an algorithm that comprehensively takes the task allocation, checkpointing scheme and speed assignment into account for designing systems with high energy-efficiency and fault-tolerance requirements. Its efficiency and effectiveness are clearly validated by extensive simulation results.

CHAPTER 6

ENHANCED FIXED-PRIORITY FAULT-TOLERANT SCHEDULING OF HARD REAL-TIME TASKS ON MULTI-CORE PLATFORMS

In the previous two chapters, we have developed real-time scheduling algorithms to minimize energy consumption for real-time tasks—from frame-based tasks to more general fixed-priority tasks—on multi-core platforms. Specifically, we have developed algorithms to judiciously set up the checkpoints for each task and the heuristics to partition real-time tasks to different processing cores accordingly. When we group and allocate real-time tasks to different cores, we simply employ the standard bin-packing heuristics such as such as First-fit (FF), Best-fit (FF), Worst-fit (WF), which does not take real-time tasks characteristics into considerations. Existing work [43, 73] has clearly shown that, by appropriately incorporating real-time task characteristics such as periods into task partitioning phase, the performance of real-time scheduling on multi-core platforms can be greatly improved. Therefore, we intend to study the problem on how to take task specifications into considerations when scheduling real-time tasks with fault-tolerance constraints on multi-core platforms.

The rest of the chapter is organized as follows. Section 6.1 discusses the related works in the literature. Section 6.2 introduces the preliminaries and notations used throughout this chapter. Section 6.3 studies the schedulability of rate-monotonic fault-tolerant tasks. Two partitioning techniques are presented in Section 6.5. In Section 6.6, we extend our partitioning algorithm to incorporate the checkpointing feature to further enhance system schedulability. Simulation studies are conducted in Section 6.7. Finally, we summarize our chapter in Section 6.8.

6.1 Related work

Searching for the optimal task partitioning is essentially a design space exploration problem. The key to the success of a partitioned algorithm is to efficiently and accurately evaluate a design alternative, i.e. a task allocation. A task allocation is considered to be feasible if the timing constraints of all tasks can be guaranteed under the influence of faults. To determine if such condition can be met, a number of fault-tolerant schedulability analysis techniques are proposed.

Pandya et al. [95] developed an utilization bound of 0.5 for hard real-time tasks scheduled under RMS policy on single-core platforms when at most one failure can occur. It is an efficient condition to test the schedulability of tasks under the influence of a failure. However, 0.5 is far from being a tight bound of the system utilization and the constraint that the system can only experience one failure is too stringent. Burns et al. [25] extended the traditional Worst Case Response Time Analysis (WCRT) for fixed-priority tasks to incorporate run-time faults. A necessary and sufficient schedulability test was derived. However, they considered failure as a special sporadic task that each failure is separated by a minimum inter-arrival time. This assumption severely limits the applicability of this approach. Zhang et al. [130] relaxed the constraints regarding the fault pattern and proposed an exact timing analysis based on the WCRT for fixed-priority tasks subject to a maximum number of faults. Despite of the accuracy of the exact timing analysis, it is computationally prohibitive and is not suitable for design space explorations. These aforementioned approaches are either too computationally expensive or too pessimistic, and are unsuitable for design space explorations.

Task partitioning is well-known as a NP-complete problem [40]. Therefore, developing effective and efficient heuristics to achieve sub-optimal results is reasonable and practical. A plethora of papers have been published on partitioned multi-core scheduling of fixed-priority periodic tasks.

Andersson et al. [6] showed that the maximum utilization a fixed-priority multi-core scheduling can achieve on each core is no more than 50%. AlEnawy et al. [4] studied the schedulability and energy performance for periodic tasks scheduled on a homogeneous multiprocessor platform with different allocation methods, e.g. Best-Fit, Worst-Fit and First-Fit and speed assignments. They concluded that the overall performance of Best-Fit dominates the other well-known heuristics in terms of schedulability. Task partitioning under multiple resource constraints was studied in [32] and efficient heuristics were proposed to improve system schedulability considering resource assignment. Fan [43] et al. exploited the fact that harmonic tasks (tasks that have periods being integer multiples of each other) can achieve higher system utilization and developed a metric to quantify how harmonic a task set is. Based on this metric, they proposed an partition approach by grouping the most harmonic tasks together and showed that it can significantly outperform traditional bin-packing approaches. Unfortunately, these approaches are fault-oblivious.

There are only a few papers which are closely related to our research. Pop et al. [96] investigated the problem of guaranteeing the schedulability and reliability of tasks with precedence constraints on a heterogenous multi-core platform. They used the combination of checkpointing and active replication to deal with the fault tolerance problem. A meta-heuristic approach, i.e. Tabu search was adopted to search for the best task allocation and fault-tolerance policy for each task. However, this approach is computational inhibitive and it is not scalable with increasing number of tasks and cores. Guo et al. [49] developed a standby-sparing technique to tolerate faults by replicating task schedules on spare cores. This approach requires extra processing cores and the aim is to save energy rather than to improve system schedulability. Our research focuses on improving the system feasibility by judiciously partitioning tasks, and later in this chapter, we discuss how this approach can be integrated into our approach for tasks with multiple checkpoints.

In what follows, we first introduce some preliminaries crucial to this chapter and use an example to motivate our research. Then we formulate our research problem formally.

6.2 Preliminaries

In this section, we introduce some basic concepts and notations used throughout this paper.

6.2.1 Application and system model

The application under investigation is modeled as a periodic task set Γ with n tasks, i.e. $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$. Each task τ_i is associated with a tuple (C_i, D_i, T_i) where C_i, D_i and T_i denote the worst case execution time, relative deadline and minimum inter-arrival time (period) of τ_i, respectively. We consider implicit-deadline tasks, i.e. $D = T$ in this paper. Each task can release an infinite number of jobs. We assume that Γ is sorted by non-decreasing period order, i.e. for $\forall \tau_i, \tau_j \in \Gamma$, $T_i \leq T_j$ if $i < j$. We use $u_i = \frac{C_i}{T_i}$ to denote the utilization of task τ_i. The total utilization of task set Γ is represented by

$$U(\Gamma) = \sum_{\tau_i \in \Gamma} \frac{C_i}{T_i}. \tag{6.1}$$

We consider a multi-core platform that consists of M homogenous preemptive cores, i.e. $\mathcal{P} = \{P_1, P_2, ..., P_M\}$. The system average utilization is denoted as

$$U_{avg} = \frac{U(\Gamma)}{M} \tag{6.2}$$

Partitioned scheduling is adopted in this paper and the tasks assigned to each core are scheduled according to RMS. We let Γ_{P_j} denote the set of tasks assigned to core P_j.

6.2.2 Fault-tolerance/reliability requirement

In this paper, we focus our efforts on tolerating transient/soft errors that do not cause permanent damage to a processing core. Transient/soft errors are the predominant type of failures in modern computing systems [30]. In particular, we consider that the system is subject to a maximum of K faults during one operation cycle of the system (its length is the least common multiple (LCM) of all task periods and is denoted by L). We adopt this K-fault model for the following three reasons: 1) it is a widely accepted fault model and well studied in the literature [124, 57, 54, 130]; 2) it is more general in a sense that it does not assume any particular fault pattern; 3) it can be readily translated to the statistical reliability requirement, as explained in [57].

To deal with the fault, we first consider the option to re-execute the entire task once a fault is detected at the end of the execution. Then the worst case recovery time for τ_i under a single failure is denoted as

$$F_i = \max_{j=1,\ldots,i} C_j. \tag{6.3}$$

The max() function is used since a lower priority job of task τ_i can be preempted by job(s) from any higher priority task $\tau_j, j \in [1, i-1]$. Therefore, the worst-case

Table 6.1: Example I: a task set with five real-time periodic tasks arranged in decreasing priority on a 2-core processor with K =1

τ_i	C_i	T_i	u_i
1	3.5	10	0.35
2	3.1	10	0.31
3	6	19	0.32
4	3	19	0.16
5	4	19	0.21

delay it may suffer due to a failure is the longest re-execution of a job among all higher-priority tasks and τ_i itself.

6.2.3 Problem formulation

With the system models defined above, we formally formulate our research problem as follows.

Problem 6.2.1. *Given a task set Γ scheduled under RMS on a multi-core platform $\mathcal{P}$, develop efficient and effective task partitioning methods such that all tasks in Γ can meet their deadlines when no more than K faults occur.*

6.2.4 Motivation example

Problem 6.2.1 is a traditional NP-complete problem even without the fault-tolerance requirements. To understand the unique challenges of Problem 6.2.1, we first present a motivate example.

Consider a 2-core platform and a task set consists of 5 tasks, the task parameters are shown in Table 6.1. Assume that in order to satisfy the reliability requirement of the task set, the task set needs to tolerate 1 fault in the worst case scenario. It is a well-known fact that, when real-time tasks are scheduled according to RMS,

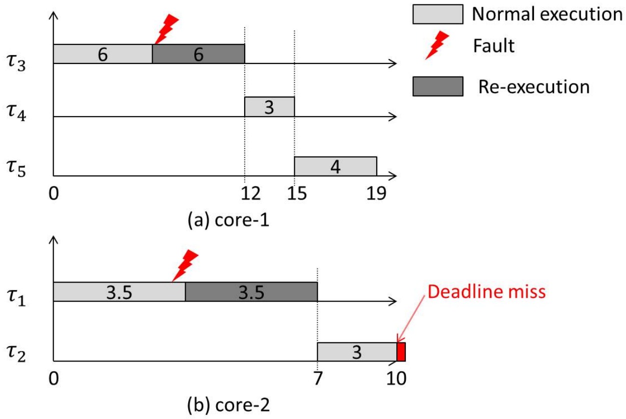


Figure 6.1: Task partition based on HAPS. Task τ_2 misses deadline under the worst case.

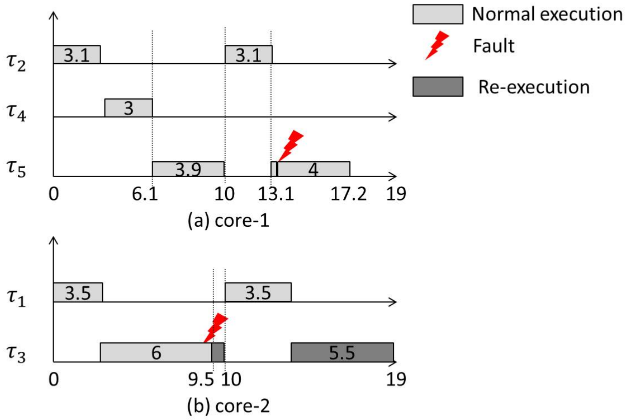


Figure 6.2: An alternative partition, all tasks are schedulable under the worst case.

allocating the harmonic tasks to the same processor can achieve the maximum utilization of 1. As shown in [43], algorithm HAPS takes advantage of this fact and, by grouping harmonic tasks together, it can significantly improve the system schedulability. Note that the sub task set $\{\tau_3, \tau_4, \tau_5\}$ and $\{\tau_1, \tau_2\}$ are perfect harmonic. Therefore, one intuitive approach is to assign $\{\tau_3, \tau_4, \tau_5\}$ to one core and $\{\tau_1, \tau_2\}$ to a different core, as shown in Figure 6.1.

As shown in Figure 6.1(a), processing core 1 is fully utilized when the worst case, i.e. a fault strikes τ_3, occurs. Still, all tasks can meet their deadlines. However, as shown in Figure 6.1(b), if a fault strikes τ_1, τ_2 will miss its deadline.

An alternative partition is to assign tasks τ_1 and τ_3 to core-2 and the rest to core-1. As shown in Figure 6.2, even though τ_1 and τ_3 are not entirely harmonic, neither are τ_2, τ_4 and τ_5, it can be readily verified that with this partition, all tasks can meet their deadlines under the worst case as shown in Figure 6.2.

The above motivation example implies that, while harmonic task sets can achieve high system utilization, making partition decisions without considering fault-tolerance requirements may undermine the schedulability of a system. In what follows, we first conduct the feasibility analysis for real-time tasks with fault-tolerance requirements and see how we can enhance the real-time system schedulability by partitioning tasks appropriately.

6.3 Fault-tolerant schedulability analysis for fixed-priority task sets

While it is a common sense that harmonic task sets can better utilize processor resource, as indicated in our motivation example above, grouping harmonic tasks together does not necessarily always lead to the best solution when fault-tolerance requirement is considered. To uncover the fundamental reason for this problem, we start with the feasibility analysis for tasks with fault-tolerance requirements since the key to a successful partitioning algorithm is to evaluate a partition result effectively in a efficient manner. One advantage of partitioning algorithms over global algorithms is that well-established single-core scheduling methods (e.g. RMS) can be readily adopted in partitioned settings. In what follows, we first introduce an existing method with pseudo-polynomial running timing for determining the schedulability of RMS-scheduled real-time tasks under the influence of transient faults. Then, we present a much more efficient schedulability test by exploiting the implicit harmonic relations between task periods.

For a task set Γ with K-fault-tolerance requirement, its feasibility can be determined using the traditional exact worst case timing analysis. Specifically, the following theorem is established in [124] for this purpose.

Theorem 6.3.1. *A task $\tau_i \in \Gamma$ is schedulable if and only if there exists a scheduling point $t \in [0, T_i]$, such that*

$$C_i + \sum_{j=1}^{i-1} \lceil \frac{t}{T_j} \rceil \cdot C_j + K \cdot F_i \leq t, \tag{6.4}$$

where t is defined in the set $\{t_x | t_x = n \cdot T_j, n \in [1, \lfloor \frac{T_i}{T_j} \rfloor], j \in [1, i]\}$. Therefore, a task set Γ is schedulable if $\forall \tau_i, \tau_i \in \Gamma$ is schedulable.

Note that, while the exact worst case response time analysis in Theorem 6.3.1 helps to identify the exact schedulability of a *given* real-time task set, it does not provide any guidance, except for being applied in traditional heuristics such as bin-packing methods, on which tasks should be grouped together and assigned to the same core to improve the system schedulability. In addition, since the complexity of this test is pseudo-polynomial, it is not suitable for design space explorations when designing large and complex systems.

As a harmonic task set is schedulable if its total utilization is no more than 1 [53], the computational complexity for schedulability checking is greatly reduced. Similarly, for a harmonic task set with K-fault-tolerance requirement, the feasibility condition can also be greatly simplified as shown in the following lemma and theorem.

Lemma 6.3.2. *Given a harmonic task set* Γ*, a task* τ_i *is schedulable with no more than K fault occurrences if and only if the following condition is met,*

$$C_i + \sum_{j=1}^{i-1} \lceil \frac{T_i}{T_j} \rceil \cdot C_j + K \cdot F_i \leq T_i \tag{6.5}$$

Proof. We prove this lemma in two steps.

Sufficient condition:if equation (6.5) is met, then τ_i must be schedulable according to Theorem 6.3.1.

Necessary condition: if task τ_i is schedulable, there must exist a scheduling point t such that equation (6.4) is satisfied. Furthermore, according to the definition of scheduling points, t must be some arrival time(s) of higher priority task(s). As a result, T_i must be some integer multiple of t, we denote it by $T_i = a \cdot t$ where a is an arbitrary integer. Moreover, T_i can be divided by any period $T_j, j \in [1, i]$. Therefore, we have the following property,

$$\begin{aligned} C_i + \sum_{j=1}^{i-1} \lceil \frac{T_i}{T_j} \rceil \cdot C_j + K \cdot F_i &= C_i + \sum_{j=1}^{i-1} \frac{T_i}{T_j} \cdot C_j + K \cdot F_i \\ &\leq a \cdot C_i + a \cdot \sum_{j=1}^{i-1} \lceil \frac{t}{T_j} \rceil \cdot C_j + a \cdot K \cdot F_i \\ &\leq a \cdot t = T_i. \end{aligned}$$

In other words, if task τ_i is feasible, then equation (6.5) must be met. Thus far, this lemma is proved. □

Theorem 6.3.3. *A harmonic task set Γ is schedulable with no more than K fault occurrences if and only if the following condition holds,*

$$\max_{i=1,\ldots,n}(U_{eff,i} + UF_i) \leq 1 \tag{6.6}$$

*where $U_{eff,i} = \sum_{j=1}^{i} u_j$ and $UF_i = K \cdot \frac{F_i}{T_i}$ denotes the **effective utilization** and the **recovery utilization** of task τ_i, respectively.*

The proof of this Theorem 6.3.3 can be readily obtained from Lemma 6.3.2 and is therefore omitted. Thus far, we develop an efficient and effective schedulability tests for harmonic task sets. However, it is a very stringent constraint for tasks to be strictly harmonic. Therefore, we relax this constraint and extend our method to more general task sets in this section. For a given task set Γ, a corresponding transformed harmonic task set Γ'_i is defined as follows.

Definition 6.3.4. *Given a task set $\Gamma = \{\tau_1, ..., \tau_i, ..., \tau_n\}$ where $\tau_i = (C_i, T_i)$ is the **base task**, then*

$$\Gamma'_i = \{\tau'_{1,i}, ..., \tau_i, ..., \tau'_{n,i}\} \tag{6.7}$$

*is a transformed **harmonic** task set with $\tau'_{j,i} = (C'_{j,i}, T'_{j,i}), \forall j \neq i$ where $C'_{j,i} = C_j$ and $T'_{j,i}$ is the largest possible period that is less than T_j and can form a harmonic relationship with all the other task periods. For two arbitrary tasks, i.e. $\tau'_{j,i}$ and $\tau'_{k,i}$ and $j < k$, the period $T'_{j,i}$ divides $T'_{k,i}$ (denoted as $T'_{j,i}|T'_{k,i}$). The utilization of task $\tau'_{j,i}$ is denoted as $u'_{j,i} = \frac{C'_{j,i}}{T'_{j,i}}$.*

In this paper, we adopt the DCT algorithm [53] to construct harmonic task sets from an arbitrary task set Γ. Note that our algorithms proposed in this paper are not restricted to any transformation method. To make this paper self-contained, we reiterate the steps of the DCT algorithm as below,

- sort task set Γ with non-decreasing period;

- using each $\tau_i \in \Gamma$, transform Γ to Γ'_i

$$T'_{j,i} = \begin{cases} T'_{j+1,i}/(\lceil T'_{j+1,i}/T_j \rceil), & \text{if} j < i \\ T_j, & \text{if} j = i \\ T'_{j-1,i} \cdot \lfloor T_j/T'_{j-1,i} \rfloor, & \text{if} j > i \end{cases} \tag{6.8}$$

Under DCT, task execution times and task orderings remain the same, but task periods become smaller. Therefore we can determine the schedulability of task set Γ from that of its transformed harmonic task sets, which is formulated in the theorem below.

Theorem 6.3.5. *Given a task set Γ with n tasks and its transformed harmonic task set through DCT, i.e. $\Gamma'_i, 1 \leq i \leq n$, if there exists i, such that Γ'_i is schedulable with the maximum of K fault-occurrences, then Γ is also schedulable with the maximum of K fault-occurrences.*

Theorem 3 can be readily proved noting that periods for tasks in the transformed task set is no larger than that in the original task set. A straightforward implementation of Theorem 6.3.3 has a computational complexity of $O(n)$. Therefore, the computational complexity to check the schedulability based on Theorem 6.3.5 is $O(n^2)$, which is usually much smaller than that of Theorem 6.3.1.

6.4 Compatibility index and its properties

The feasibility analysis results presented above clearly reveal the reason why allocating harmonic tasks to the same core can in fact lead to inferior solutions. Note that, from equation (6.6), the schedulability of a harmonic task set with fault-tolerance requirement depends not only on the task set utilization itself but also recovery utilization as well. Therefore, to partition tasks with fault-tolerance requirements, we need to consider not only if tasks are harmonic but also if they are "compatible". To this end, we design a new metric to quantify the *compatibility* of tasks to be allocated to the same processing core.

Definition 6.4.1. *Given an arbitrary task set* Γ *and its transformed harmonic task set* Γ'_i *as defined in Definition 6.3.4, then the* **compatibility index** *of task* τ_j*,* $\tau_j \in \Gamma$ *measured under configuration* Γ'_i *is defined as*

$$COMP(\tau_j, \Gamma'_i) = \Delta H_{j,i} + \Delta EF_{j,i}, \tag{6.9}$$

where $\Delta H_{j,i} = u'_{j,i} - u_j$ *and* $\Delta EF_{j,i} = K \cdot \frac{F_j - C_j}{T'_{j,i}}$ *denote the* **harmonic distance** *of task* τ_j *to its counterpart in* Γ'_i *and the* **extra recovery utilization** *task* τ_j *has to endure considering all the higher-priority tasks in* Γ*.*

In what follows, we study the impacts of each factor exclusively. A *harmonic distance* $\Delta H_{j,i}$ [43] quantifies how much utilization of a task τ_j needs to be increased in order to transform a task set Γ to a harmonic task set Γ'_i. In other words, it measures how harmonic the task τ_j is with respect to all the remaining tasks in Γ. The less the harmonic distance for each task is, the better the system schedulability is. We formally formulate this property in the following theorem.

Theorem 6.4.2. *Consider a task set* Γ *and its two transformed harmonic task set* Γ'_p *and* Γ'_q *(using* τ_p *and* τ_q *as base tasks, respectively), where* $\Delta EF_{i,p} = \Delta EF_{i,q}$ *and* $\Delta H_{i,p} \leq \Delta H_{i,q}$*, for* $\forall \tau_i$*. The task set* Γ'_p *must be schedulable if* Γ'_q *is schedulable.*

Proof. If the task set Γ'_q is schedulable, then for each task $\tau'_{i,q} \in \Gamma'_q$, the following condition must be satisfied according to Theorem 6.3.3,

$$\frac{C_i}{T'_{i,q}} + \sum_{j=1}^{i-1} \frac{C_j}{T'_{j,q}} + K \cdot \frac{F_i}{T'_{i,q}} \leq 1. \tag{6.10}$$

Since $\Delta EF_{i,p} = \Delta EF_{i,q}$ and $\Delta H_{i,p} \leq \Delta H_{i,q}$, $\forall \tau_i$, we have

$$u'_{i,p} \leq u'_{i,q} \Longleftrightarrow T'_{i,p} \geq T'_{i,q}. \tag{6.11}$$

Then equation (6.10) can also be satisfied with a larger period $T'_{i,p}$, which means that task set Γ'_p is schedulable. $\square$

The *extra recovery utilization* represents the extra recovery overheads that task τ_j needs to tolerate when it is subject to preemptions from all higher-priority tasks in Γ. A task is more likely to be schedulable when there is less extra recovery overhead. Therefore, with less extra recovery overhead for each task, the system can potentially achieve better schedulability. We summarize this property in Theorem 6.4.3.

Theorem 6.4.3. *Given two harmonic task sets Γ_1 and Γ_2 with identical number of tasks, let $\tau_{j,1}$ and $\tau_{j,2}$ be their j_{th} task in Γ_1 and Γ_2, respectively. Assume that $\forall j$, $u_{j,1} = u_{j,2}$ and $\Delta EF_{j,1} \leq \Delta EF_{j,2}$. If Γ_2 is schedulable, then Γ_1 must also be schedulable.*

Proof. If the task set Γ_2 is schedulable, then for each task $\tau_{j,2} \in \Gamma_2$, the following condition must be satisfied according to Theorem 6.3.3,

$$u_{j,2} + \sum_{i=1}^{j-1} u_{i,2} + K \cdot \frac{F_{j,2}}{T_{j,2}} \leq 1. \tag{6.12}$$

Since $u_{j,1} = u_{j,2}$, and $\Delta EF_{j,1} \leq \Delta EF_{j,2}$, we have

$$\begin{aligned}
\Delta EF_{j,1} - \Delta EF_{j,2} = K \cdot \frac{F_{j,1} - C_{j,1}}{T_{j,1}} \\
- K \cdot \frac{F_{j,2} - C_{j,2}}{T_{j,2}} \\
= K \cdot \frac{F_{j,1}}{T_{j,1}} - K \cdot \frac{F_{j,2}}{T_{j,2}} - K \cdot (u_{j,1} - u_{j,2}) \\
= K \cdot \frac{F_{j,1}}{T_{j,1}} - K \cdot \frac{F_{j,2}}{T_{j,2}} \leq 0.
\end{aligned}$$

Therefore, for each task $\tau_{j,1} \in \Gamma_1$, the following condition is met,

$$u_{j,1} + \sum_{i=1}^{j-1} u_{i,1} + K \cdot \frac{F_{j,1}}{T_{j,1}} \leq 1. \tag{6.13}$$

In other words, Γ_1 is schedulable. Thus far, this theorem is proved. □

The above two theorems show that both of the factors in compatibility index, i.e. harmonic distance and extra recovery utilization, can play significant roles in reflecting the schedulability of a task set. We consider both factors equally important, and we define the compatibility index of a task set as follows.

Definition 6.4.4. *The* **compatibility index** *of a task set* Γ *consisting of* n *tasks is defined as*

$$COMPTS(\Gamma) = \min_{i=1,\dots,n} \sum_{j=1}^{n} COMP(\tau_j, \Gamma_i'), \tag{6.14}$$

where $COMP(\tau_j, \Gamma_i')$ *is formulated in Definition 6.4.1. The less the value* $COMPTS(\Gamma)$ *is, the more compatible* Γ *is.*

This metric measures not only the harmonicity of a task set but also the fault-compatibility among all tasks. Let us use the example in Section 6.2.4 to illustrate the efficacy of this metric. We have $COMPTS(\{\tau_1, \tau_2\}) = 0.04$ where $COMPTS(\{\tau_1, \tau_3\}) = 0.018$. Then τ_1 and τ_3 are deemed to be more compatible, though their periods are not strictly harmonic.

6.5 Fault-tolerant task partitioning

In light of Section 6.4, task sets with lower "compatibility index" (more compatible) are more likely to be schedulable under the influence of transient faults.

We are now ready to present our multi-core partition algorithm "Compatibility Aware Task Partition (CATP)" in Algorithm 8.

Algorithm 8 *CATP*(Γ, $\mathcal{P}$, K)

```
Require:
    Γ - task set with n tasks, 𝒫 - multi-core platform with m cores, K- number of faults.
 1: sort tasks in non-increasing utilization order;
 2: for $i = 1$ to $n$ do
 3:     $p\_index = 0$; $c\_min = +\infty$;
 4:     for $j = 1$ to $M$ do
 5:         if $\tau_i$ can be assigned to $P_j$ then
 6:             if $COMPTS(\{\tau_i, \Gamma_{P_j}\}) < c\_min$ then $p\_index = j$
 7:         end if
 8:     end for
 9:     if $core\_index == 0$ then return"FAILURE";
10:     else $\tau_i \rightarrow P_{p\_index}$ ;
11: end for
12: return "SUCCESS" and partition results;
```

The task set Γ is first arranged in non-increasing utilization fashion (Line 1). The algorithm allocates one task at a time. Within each step, it tentatively assigns the current task to each core and measures how compatible the task is with the existing tasks on the core (Lines 3-8). If the current task can not be allocated to any of the cores, the algorithm reports that a feasible allocation can not be found (Line 9). Otherwise, the task is assigned to the core with the minimum compatibility value (Line 10). The partition result is returned if all tasks can be successfully allocated.

Algorithm 8 is simple yet effective. It is a greedy approach as it intends to find the best candidate core in each step when assigning a task. However, the limitation of assigning task one at time comes at the ignorance of the fact that a task to be assigned in later stage may not be packed with the most compatible tasks due to schedulability constraints.

Let us revisit Example 1. By running Algorithm CATP, a feasible partition can be found with tasks τ_1, τ_4, τ_5 assigned to core-1 and tasks τ_2 and τ_3 to core-2. If we add another task τ_6 with parameters $C_6 = 2.6$ and $T_6 = 19$, it can be verified that

τ_6 can not be allocated to neither of the core, which results in a FAILURE. With a careful examination, we can see that the most compatible tasks are τ_1 and τ_3 with a $COMPTS(\{\tau_1, \tau_3\}) = 0.018$, if we first group these two tasks together and assign them to core-1, the rest of the tasks with a $COMPTS(\{\tau_2, \tau_4, \tau_5, \tau_6\}) = 0.053$ to core-2, all tasks are schedulable under the worst case.

Next, we present our "Group-wise Compatibility Aware Task Partition (G-CATP)" method in Algorithm 9.

Algorithm 9 *G-CATP*(Γ, K)

```
Require:
    $\Gamma$ - task set with n tasks, $\mathcal{P}$ - multi-core platform with m cores, $K$- number of faults.
 1: sort tasks in non-decreasing period order;
 2: while isNOTempty($\Gamma$) AND isNOTempty($\mathcal{P}$) do
 3:     $\Gamma_{opt} = \emptyset$;
 4:     for $i = 1$ to $|\Gamma|$ do
 5:         Transform $\Gamma$ into $\Gamma'_i$ with base task $\tau_i$;
 6:         Find a subset $\Gamma'_{sub}$ from task set $\Gamma'_i$ (corresponding to $\Gamma_{sub}$ from $\Gamma$) such that
            1. $\Gamma'_{sub}$ is schedulable;
            2. $U(\Gamma'_{sub})$ is maximized;
            3. $COMPTS(\Gamma'_{sub})$ is minimized.
 7:         if $U(\Gamma_{sub}) > U(\Gamma_{opt})$ then $\Gamma_{opt} = \Gamma_{sub}$;
 8:     end for
 9:     if $U(\Gamma_{opt}) == \emptyset$, then return "FAILURE";
10:     else $\Gamma = \Gamma - \Gamma_{opt}$; $\Gamma_{opt} \rightarrow$ an empty core;
11: end while
12: if isNOTempty($\Gamma$) then return "FAILURE";
13: else return "SUCCESS" and partition results
```

Different from Algorithm 8, in each step, Algorithm 9 assigns a group of tasks together to a core. Under each harmonic transformation, determining the most compatible subset of tasks while simultaneously guaranteeing all three conditions at Line 6 is not a trivial task. A brute-force exhaustive search is apparently computationally inhibitive and impractical. Therefore, we use the heuristic as follows. With a given base task τ_i, we first assign τ'_i to Γ'_{sub}. Then, we scan all the remaining tasks in Γ'_i and find the task that results in the minimum increase of $COMPTS(\Gamma'_{sub})$ if it is assigned to Γ'_{sub}. We repeat the process until no more tasks can be added to Γ'_{sub}. After a group of tasks with the largest total original utilization (utilizations before harmonic transformation) are determined (Lines 3-8), they will be assigned to the first available core and removed from Γ (Line 10). The algorithm reports "SUCCESS" if all tasks can be assigned but otherwise report "FAILURE".

Next, we extend our partitioning algorithms to incorporate the checkpointing feature to further enhance system schedulability.

6.6 Task set with checkpointing

Till now, we assume that an entire job is re-executed once a fault is detected. As shown in [55], checkpointing with roll-back recovery is a very efficient technique to reduce recovery overhead and improve system schedulability. To our best knowledge, there is no work that targets on improving system schedulability for fixed-priority tasks on multi-core platforms based on exploring the combination of task partitioning and checkpointing. Different task partitions can result in different checkpointing configurations. Moreover, without the knowledge of task partitioning, a predefined checkpointing scheme will most likely lead to poor schedulability performances. Therefore, it is not a trivial problem to search for the best combination of checkpointing and task allocation. In what follows, we endeavor to develop efficient and effective heuristics with the joint consideration of checkpointing and task allocation in order to maximize system schedulability. Specifically, we extend our partitioning algorithms, i.e. CATP and G-CATP, to incorporate the checkpointing scheme. We first introduce some basics on checkpointing for ease of presentation.

Under checkpoint scheme, instead of rolling back to the beginning of the execution of a job, the last saved checkpoint is retrieved and the job is executed thereafter. For a task τ_i with m_i number of checkpoints, the length of a re-execution segment is $\frac{C_i}{m_i+1}$. Therefore, the worst case recovery time for a job of τ_i is modified to

$$F_i = \max_{j=1,\ldots,i}(\frac{C_i}{m_i+1}). \tag{6.15}$$

Additionally, as inserting checkpoints incurs overhead, the worst case exsection time of τ_i with m_i checkpoints (its overhead is denoted by o_i) is denoted as

$$C_i(m_i) = C_i + m_i \cdot o_i. \tag{6.16}$$

With the new execution time and recovery for each task $\tau_j \in \Gamma$, the two factors, i.e. harmonic distance and extra recovery overhead, in the compatibility index defined in Definition 6.4.1 are modified accordingly to

$$\Delta H_{j,i} = \frac{C_j(m_j)}{T'_{j,i}} - \frac{C_j(m_j)}{T_{j,i}} \tag{6.17}$$

and

$$\Delta EF_{j,i} = K \cdot \frac{F_j - \frac{C_j}{m_j+1}}{T'_{j,i}}, \tag{6.18}$$

respectively.

To find a feasible checkpoint scheme for a set of fixed-priority tasks, we adopt the method ECHK in Chapter 4.6. ECHK iteratively inserts checkpoints to the task which has a higher or equal priority than the first unschedulable task and the largest recovery overhead. The algorithm ECHK returns either the checkpointing configuration if a feasible one can be found or a failure status indicating that the task set is unschedulable.

Then, algorithm CATP can be directly extended to integrate the checkpointing scheme. Tasks are assigned one at a time, and a task-to-core mapping is considered feasible only when there exists a feasible checkpointing configuration for the task set (including the to-be-assigned task) on that core. Given a checkpointing scheme, the corresponding updated task-set compatibility index can be readily obtained. Among all the feasible cores (mappings), the one with the least task-set compatibility index is selected. This process is repeated until all tasks are assigned or no core can accommodate any more tasks. We denote this algorithm as CATP-CHK. CATP-CHK essentially utilizes "compatibility index" to evaluate the fitness of task allocation and checkpointing, as "compatibility index" has been shown to be very effective in reflecting system schedulability.

Similarly, we modify algorithm G-CATP to incorporate the checkpointing scheme. Different from CATP, G-CATP tries to find the most compatible group of tasks with the largest total utilization in each step. However, this problem with the integration of checkpointing becomes more complicated, as different checkpointing configurations can lead to large variations of the "compatibility index" of a task set. Therefore, developing efficient and effective heuristics to solve this problem is practical. Following the same procedures in Algorithm 9, we search the most compatible group of tasks under each harmonic transformation, and the group is initialized with only the base task, i.e. the task used for harmonic transformation. The rationale of choosing the base task as the first task in the group is that the group will have the least amount of task-set "compatibility index", i.e. 0, at the beginning. Then, we add tasks to the group one at a time. A task can be combined into the group only when ECHK returns a feasible checkpointing configuration. Among all the tasks that can be assigned to the group, the task which leads to the minimum increase of "compatibility index" is selected. This process repeats until no more tasks can be added to the group without jeopardizing its schedulability. We denote this algorithm as G-CATP-CHK.

In the following section, we use extensive simulations to demonstrate the effectiveness of our proposed algorithms.

6.7 Simulation results

In this section, we use simulations to evaluate the performance of our proposed partition algorithms. Specifically, we first study the impacts of different parameters, i.e. the number of tasks and cores, system average utilization, and the maximum number of faults on system schedulability. Then, we investigate the effectiveness of incorporating checkpointing scheme to improve system schedulability.

As explained in [40], a widely adopted metric to evaluate a partition algorithm is the *acceptance ratio*. First, a number of synthetic task sets are generated, and the acceptance ratio is calculated as the number of successfully partitioned task sets divided by the total number of task sets as shown in equation (6.19).

$$\text{acceptance ratio} = \frac{\text{The number of schedulable task sets}}{\text{The total number of tasks sets}}, \tag{6.19}$$

Four algorithms are evaluated in this section, namely, G-CATP, CATP, HAPS and Best-Fit Decreasing (BFD). The first two are proposed and explained in Section 6.5. The details of the algorithm HAPS is elaborated in [43], where the algorithm uses the *harmonic distance* (equation (9,14) without considering the term *extra recovery utilizaiton*) as the guideline when tasks are partitioned. $HAPS$ has been shown to be quite effective in improving system utilization for fault-oblivious systems. BFD orders the tasks in a non-increasing utilization fashion and assigns a task to the core with the minimum remaining utilization.

The experimental setup is listed in details as follows. Task sets were generated according to the algorithm UUniFast in [17]. UUniFast is an algorithm designed for single-core platform. In order to generate a task set with a total utilization,

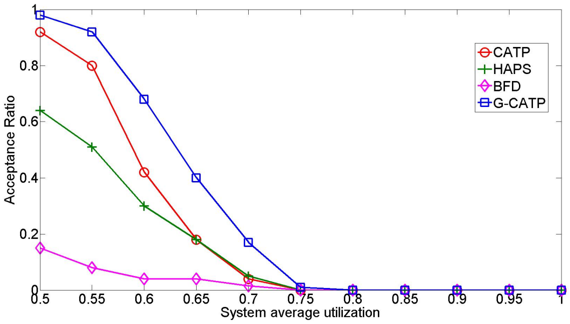


Figure 6.3: 32 tasks on 4-core platform, K=2.

i.e. U_{total} larger than 1, we need to execute this algorithm M times with the target utilization of $\frac{U_{total}}{M}$ during each run. We discarded the test cases where an individual task utilization exceeds $\frac{1}{K+1}$ since a task with a larger utilization than this can not even be scheduled by itself under the worst case scenario, i.e. K faults occur. The period of each task τ_i, i.e. T_i was randomly generated in the range of $[10, 1000]$, and its execution time C_i was calculated as $u_i \cdot T_i$.

6.7.1 Experiment 1, acceptance ratio vs. system average utilization.

In this set of experiments, we study the relationships between system average utilization and acceptance ratio. We fixed the the number of tasks and varied the system average utilization in the range $[0.5, 1]$ with a step of 0.05. We considered a 4-core platform with 32 real-time tasks. A maximum number of 2 faults was assumed in order to satisfy the system reliability constraint. For each utilization value, we generated 1000 task sets and the acceptance ratio was recorded in Figure 6.3. As we can see, the acceptance ratios for all four algorithms drop as the system average utilization increases. This is reasonable since task sets with high utilizations are difficult to be scheduled, especially when fault-tolerance is considered. BFD has

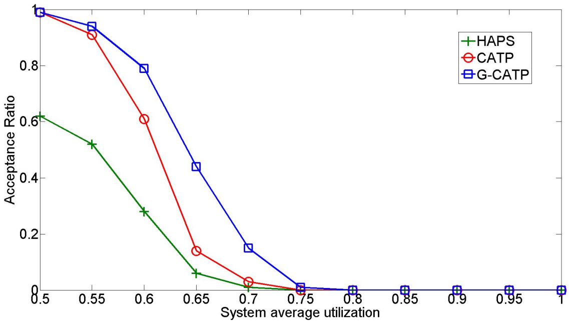


Figure 6.4: 64 tasks on 8-core platform, K=2.

the worst performance since it does not take the characteristics of tasks (e.g. harmonicity or compatibility) into consideration. With a system utilization of 0.5, BFD can only achieve an acceptance ratio less than 20%. In the remaining experiments, we excluded BFD for comparison unless otherwise specified. Both of our proposed algorithms outperforms HAPS, due to the fact that our algorithms utilize a more accurate metric to capture how compatible tasks are during the partition process. In general, CATP has a better performance than HAPS. For example, with a system average utilization of 0.55, CATP has an acceptance ratio of 80% while HAPS only achieves 52%, which is an approximate 60% improvements. $G-CATP$ has the best performance as it tries to search for the most compatible group of tasks in each step. For instance, when the system average utilization is 0.65, $G-CATP$ still achieves an acceptance ratio of 41%, while $CATP$ and $HAPS$ only have an acceptance ratio less than 20%. In average, CATP obtains a 24% improvement over HAPS, and $G-CATP$ manages to get a further 40% enhancement over CATP.

Next, we evaluated these algorithms on a 8-core platform with 64 tasks. Additionally, a maximum of 2 faults was assumed and the system average utilization was varied in the range $[0.5, 1]$ with a step of 0.05. The superiority of our proposed algorithms over HAPS is illustrated in Figure 6.4, 1000 task sets were generated for each point on the $x-axis$.

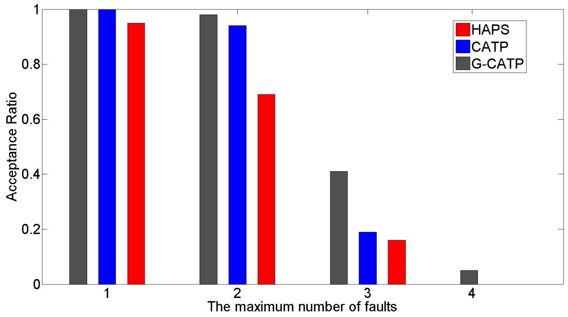


Figure 6.5: 32 tasks on 4-core platform, system average utilization is 0.5.

As the number of cores and tasks increase, all three algorithms exhibit higher acceptance ratio. This is because with a higher number of tasks, each task is likely to be associated with a smaller utilization given a fixed system average utilization and they are easier to be scheduled. CATP still exhibits significant improvement over HAPS. When the system average utilization is 0.6, HAPS has an acceptance ratios of 30%, while CATP achieves twice the value, i.e. 60%. G-CATP still has the dominated performance among all three algorithms. In average, CATP attains a 48% improvement in performance over HAPS whereas G-CATP further enhances CATP by an approximate 45%. Both our algorithms tend to have better performances for systems with more tasks and cores, since they aggressively find the most compatible tasks in each step.

6.7.2 Experiment 2, acceptance ratio vs. the number of faults.

In this section, we investigate the relationships between acceptance ratio and the number of faults that a system needs to tolerate. We fixed the system average utilization as 0.5 with a 4-core platform consisting of 32 tasks. We varied the

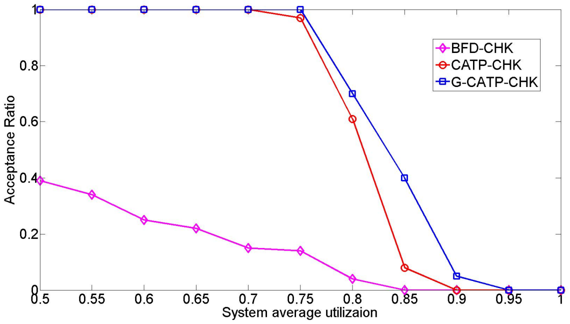


Figure 6.6: 32 tasks on 4-core platform, checkpoint overhead is 5 percent of execution time, K=2.

number of faults from 1 to 4, i.e. $K \in [1, 4]$. The result is shown in Figure 6.5, 1000 task sets were generated for each configuration.

As the number of fault increases, the acceptance ratio decreases dramatically for all three algorithms. However, both of our algorithms, i.e. CATP and G-CATP outperform the algorithm HAPS. When K = 2, HAPS has an acceptance ratio of 70% while CATP and G-CATP can achieve 92% and 95%, respectively. Our G-CATP algorithm still exhibits the best performance, it manages to achieve 40% acceptance ratio while CATP and HAPS only have an acceptance ratio less than 20% when $K = 3$.

The experimental results clearly demonstrate the effectiveness of our proposed algorithms in terms of improving system schedulability under failures. By grouping compatible tasks and assigning them to the same core, we significantly enhance system utilization and leave more space for the remaining tasks.

6.7.3 Experiment 3, acceptance ratio vs. checkpointing

In this section, we study the effects of checkpointing on system schedulability. Since there is no existing work in the literature that solves the exact same problem, we first extend the BFD to incorporate the checkpointing feature. The tasks are sorted in non-increasing order of utilization, the algorithm allocates one task at a time and use algorithm ECHK [55] to search for a feasible checkpointing scheme for this task and tasks on each candidate core. Among all feasible cores (a core is considered feasible if there exists a feasible checkpointing configuration among the tasks on the core and the task to be allocated), the task is assigned to the core with the least remanning utilization. BFD reports "FAILURE" is a task can not be assigned. We denote this algorithm as BFD-CHK. The performances of BFD-CHK and our two algorithm CATP-CHK and G-CATP-CHK were evaluated.

We considered 32 tasks on a 4-core platform. The maximum number of faults, i.e. K, was set to 2 and the checkpointing overhead was assumed to be 5% of the worst case execution time for each task. System average utilization was varied from 0.5 to 1 with a step of 0.05. 1000 task sets were generated for each test case and the acceptance ratios were plot in Figure 6.6.

Compared with Figure 6.3, the acceptance ratios are increased significantly. For instance, CATP-CHK and G-CATP-CHK and can still achieve 60% and 70% acceptance ratios under a relatively high system average utilization of 0.8 (zero under CATP and G-CATP), respectively. This is due to the fact that checkpointing can considerably reduce the recovery overhead of each task and enhance the system schedulability under faults. As can be seen, our algorithms substantially outperform BFD-CHK. With a utilization of 0.5, CATP-CHK and G-CATP-CHK both achieve an acceptance ratio of 100% whereas BFD-CHK only achieves 40%. Among all three algorithms, G-CATP-CHK still exhibits the best performance since it always tries to search for the most compatible tasks in each step. When system average utilization is 0.85, G-CATP-CHK has an acceptance ratio about 40%, while that of CATP-CHK is less than 10%. Once again, the simulation results demonstrate that *compatibility index* is an accurate metric to quantify how "compatible" a task set is and can guide task partitioning correctly.

6.8 Summary and future directions

As the computing paradigm shifts toward multi-core platforms, the need for effective and efficient multi-core scheduling is ever-growing. Also, facing the unprecedented reliability challenges brought forth by relentless transistor miniaturizations and mass integrations of transistors into a single chip, traditional multi-core scheduling without explicitly considering system reliability is becoming obsolete. In this paper, we first present an efficient test to evaluate the schedulability of tasks scheduled according to rate-monotonic method under faults. Then, we develop a novel metric to quantify the "compatibility" among tasks, which is a direct indication of system schedulability. In light of this metric, we develop two partitioning approaches CATP and G-CATP. While algorithm CATP assigns one task at a time to the most compatible core, G-CATP searches for the most compatible group of tasks in each step and assigns them to one core. We further extend our algorithms to incorporate the checkpointing scheme to further improve system utilization. Simulation results have shown that our proposed algorithms can achieve substantial improvements over other related approaches.

Adopting the concept of "compatibility index" for various fault-tolerant real-time task models and studying the corresponding partitioning problem on multi-core platforms are very interesting research directions. For example, when considering constrained-deadline tasks ($D_i \leq T_i, \forall i$) instead of implicit-deadline tasks ($D_i = T_i, \forall i$), RMS is not optimal anymore. What's worse, the *harmonic distance* and the *extra recovery utilization* that are defined solely based task execution times and periods may become inaccurate to quantify how "compatible" a set of task is. Adding the third dimension, i.e. deadline, to "compatibility index" and judiciously make partitioning decisions are not trivial problems and require careful investigation.

Another interesting direction is to extend the approaches in this paper to heterogenous multi-core platforms. Different from homogenous multi-core platform, the execution profile for each task may vary widely from core to core. To improve task set schedulability, one intuitive approach is to assign "heavy tasks" to fast cores. However, a task may take less execution time on one core but is more compatible with the tasks on the other core. How to make tradeoffs between execution speed and compatibility is worth careful studying.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this chapter, we first summarize our contributions presented in this dissertation. We then discuss the possible directions for our future research work.

7.1 Summary

Nowadays, real-time computing systems are prevalent in our daily lives. They are growing rapidly in both scale and complexity thanks to the advancements of IC technology, in particular, the transistor scaling and mass integration. However, these progresses have brought unprecedented challenges for designing real-time systems that are subject to a variety of constraints, e.g. timing, power, and reliability. In this dissertation, we focused our efforts on developing efficient and effective techniques for hard real-time systems with the purpose of providing deterministic guarantees to timing constraints under transient faults while optimizing other design metrics such as power consumption and the number of cores required for a feasible schedule.

Specifically, we started by studying the combinatorial problem of energy efficiency and fault tolerance for EDF-scheduled real-time tasks on a single-core platform. Since both energy management (i.e. DVFS) and fault recovery require system slacks, the challenge lies in how to judiciously make tradeoffs between these two factors, such that the deadlines of all real-time tasks can be met while the system energy consumption can be minimized. In this regard, we presented three algorithms, namely MLPEDF, EMLPEDF, and LPSSR. The first two algorithms were extensions of a popular fault-oblivious DVFS scheduling. LPSSR further improves the former two approaches by exploiting shared recovery reservations. By sharing the reserved resources between different tasks/jobs instead of reserving a backup for each job, system resources can be efficiently utilized and DVFS can be utilized more aggressively without jeopardizing system schedulability. The simulations results have demonstrated the efficacy of our proposed methods when compared to other related works.

Next, we ventured into the field of multi-core scheduling with multiple design constraints as there is strong evidence showing that multi-core architectures are becoming mainstream in modern computing systems. We first investigated the problem of scheduling frame-based tasks, i.e. all tasks share the same deadline, on homogenous multi-core platforms with the joint consideration of energy minimization and fault tolerance. We adopted checkpointing as our fault-recovery mechanism as it is known to be very effective in reducing fault-recovery overheads. We first developed a checkpointing scheme that is efficient and optimal in terms of minimizing the overall schedule length, i.e. OPT_CHK, for a set of tasks scheduled on the same core. Then, based on this technique, we proposed an efficient task-allocation method, i.e. EATA, that tries to find the best combination of task allocation, checkpointing scheme, and speed assignment at each step. The efficiency of OPT_CHK in reducing the computational complexity of searching for the optimal number of checkpoints for multiple tasks and the effectiveness of EATA in energy reductions have been demonstrated using extensive simulation results, respectively.

Furthermore, we relaxed the deadline constraints in the previous problem and studied the fault-tolerant scheduling of general fixed-priority tasks on multi-core platforms with the consideration of energy minimization. Similarly, checkpointing and DVFS were adopted as the fault-recovery and energy-management technique, respectively. Different from the previous problem, it is extremely difficult, if not impossible, to find the optimal checkpointing numbers for fixed-priority tasks with arbitrary deadlines. Instead, we proposed an algorithm, i.e. ECHK, to efficiently identify a feasible checkpointing configuration for a set of fixed-priority tasks scheduled on the same core. Based on this technique, we proposed a task-partitioning technique where, in each step, a task is assigned to the core with the most favorable checkpointing configuration and the lowest feasible processing speed. Again, the significant improvements of all these proposed methods over existing related approaches were validated using extensive simulations.

Finally, we investigated the problem of maximizing system schedulability under the influence of transient faults through partitioning RMS-scheduled real-time tasks on multi-core platforms. Motivated by the fact that harmonic task sets (task periods are integer multiples of each other) can result in higher system utilization [43], we explored the implicit relations between tasks and derived a metric named "compatibility index" to quantify how "compatible" a task set is. We theoretically proved that this metric can help determine the system schedulability effectively. Therefore, by grouping tasks with lower compatibility index (more compatible), the system is more likely to be schedulable. Specifically, we proposed two partitioning techniques, namely CATP and G-CATP. In CATP, tasks are allocated one at a time to the core with the lowest compatibility index whereas G-CATP first identifies a group of most compatible tasks and assigns them together to one core. Then, we further extended these two methods to incorporate the checkpointing feature. According to the simulation results, CATP and G-CAPT can significantly outperform the existing methods, such as traditional bin-packing, i.e. BF,WF and FF and a harmonic-aware technique [43].

7.2 Future work

In this dissertation, we primarily focus on developing reactive methods for fault tolerance. In other words, we put an emphasis on how to deal with run-time faults when they occur. Recently, another line of research has thrived to address the system reliability from a different perspective, i.e. prolonging system lifetime. It is becoming evidently important for systems that operate in a harsh or remote environment. For instance, for a real-time safety-critical system, such as avionic controls in space crafts with the requirement of at least 25 years of service life, maximizing system lifetime reliability is of paramount importance as meeting other restrictions, e.g. real-time constraints. It is a well-known fact that system lifetime reliability is highly influenced by temperature, a $10 - 15°C$ difference in temperature can result in a $2 - 3\times$ difference in the lifespan of a device [120, 113]. Unfortunately, aggressive scaling in semiconductor technology and increasing of transistor counts result in high power density and hence high temperature, which in turn pose unprecedented challenges on system lifetime reliability [93, 51].

We are interested in extending our research to address the problem of how to develop real-time scheduling algorithms to maximize the system lifetime while guaranteeing timing constraints and optimizing other performance metrics, e.g. energy consumption and throughput. In what follows, we present some preliminary results of our research on studying the impacts of real-time scheduling on system lifetime reliability.

7.2.1 Lifetime and fault model

In this section, we introduce three major metrics for evaluating system lifetime reliability [104] and discuss their relationships.

Reliability function

First, we will introduce several definitions and notations for ease of presentation.

Definition 7.2.1. *The observed time to failure(TTF) is a value of the random variable θ which represents the lifetime of the device, and its probability density function (PDF) is denoted as $f_\theta(t)$. Consequently, its cumulative distribution function(CDF) is $F_\theta(t) = P(\theta \leq t)$ and termed as **unreliability** at time t, which represents the probability of failure in the interval $[0, t]$. Therefore, the **reliability** function is*

$$R_\theta(t) = P(\theta > t) = \int_t^\infty f_\theta(x)dx = 1 - F_\theta(t), \tag{7.1}$$

where reliability denotes the probability of no failures in the interval $[0, t]$ or equivalently, the probability of failure after t.

Note that we implicity assume that random variable θ is continuous, which is true most of the time. For simplicity, we use $f(t)$, $F(t)$ and $R(t)$ to represent the PDF of the random variable θ, unreliability, and reliability at time t, respectively. Once we have the knowledge of the distribution of the random variable θ, we can calculate the reliability at any time t.

Failure rate

At times, specifying the distribution function of θ directly from the information that is available proves difficult. The conditional density function $h(t)$ which is referred to as the **hazard function** or **failure rate** is useful in these situations.

Definition 7.2.2. *Given an interval $[t, t + dt]$, the conditional failure rate during this interval is defined as the conditional probability of failure in the interval (given that there is no failure before t) divided by the length of the interval. It is formally* *defined by the following equation [75]:*

$$\frac{P(t < \theta < t + dt | \theta > t)}{dt} = \frac{R(t) - R(t + dt)}{R(t)dt}. \tag{7.2}$$

Definition 7.2.3. *The instantaneous failure rate at t is the limit of the equation (7.2) as $dt \to 0$. That is,*

$$h(t) = \lim_{dt \to 0} \frac{R(t) - R(t + dt)}{R(t)dt} = \frac{-R'(t)}{R(t)} = \frac{-d(\ln R(t))}{dt}. \tag{7.3}$$

Based on the the above equation, with some simple mathematical manipulations, we have

$$R(t) = \exp(-\int_0^t h(x)dx) \tag{7.4}$$

Furthermore, since $f(t) = -\frac{dR(t)}{dt}$, we know $h(t) = \frac{f(t)}{R(t)}$. Therefore, with the knowledge of any one of the three functions($h(t)$, $f(t)$ and $R(t)$), we can directly derive the others.

Mean time to failure (MTTF)

MTTF is widely used as an indicator of the system life span.

Definition 7.2.4. *The MTTF is the expected time to failure for a component or system.*

Mathematically, it is formulated as:

$$\begin{aligned} MTTF = E(\theta) &= \int_0^\infty tf(t)dt = \int_0^\infty t(\frac{-dR(t)}{dt})dt \\ &= -tR(t)|_0^\infty + \int_0^\infty R(t)dt = \int_0^\infty R(t)dt, \end{aligned}$$

if $\lim_{t \to \infty} tR(t) = 0$, which is true for a distribution whose mean exists [75]. Moreover, for many of the popular probability density functions, it is not necessary to perform the integration since their means are already known.

Failure models

In what follows, we explain in details the four wear-out failure mechanisms that are related to system reliability and are presently dominant in integrated circuits.

Electromigration refers to the transfer of metal as a result of the gradual movement of ions in the conducting path caused by the momentum tranfer between conducting electrons and diffusing metal atoms. The MTTF due to EM is given by the following equation [72]:

$$MTTF_{EM} = \frac{A_{EM}}{J^n} e^{\frac{E_{a_{EM}}}{\kappa T}}, \tag{7.5}$$

where A_{EM} and n are empirically determined constant. J is the current density in interconnect, and $E_{a_{EM}}$ is the activation energy for electormigration. κ is Boltzmann's constant, and T is the absolute temperature in Kelvin.

Time Dependent Dielectric Breakdown is a wear-out mechanism of gate oxide (or dielectric). It causes permanent failure when a conductive path forms in the dielectric. This effect is strongly influenced by temperature and is becoming worse with the advent of thin and ultra-thin gate oxides [1]. The model for the MTTF due to TDDB is defined as [72]:

$$MTTF_{TDDB} = A_{TDDB} (\frac{1}{V})^{(a-bT)} e^{(\frac{X+\frac{Y}{T}+ZT}{\kappa T})}, \tag{7.6}$$

where a,b,X,Y and Z are fitting parameters. A_{TDDB} is a empirically determined constant, and V is the supply voltage. Again, κ and T are the Boltzmann's constant and temperature respectively.

Stress Migration, much like EM, refers to the migration of metal atoms in the interconnect. It is caused by mechanical stress due to different thermal expansion rates of different materials in the device. The MTTF resulting from SM is given by the following equation [72]:

$$MTTF_{SM} = A_{SM} |T_0 - T|^{-n} e^{\frac{E_{a_{SM}}}{\kappa T}}, \tag{7.7}$$

where A_{SM} is an empirically determined constant, and T_0 is the metal deposition temperature(stress free temperature) during fabrication. T is the operating temperature, and n and $E_{a_{SM}}$ are material dependent constants. κ is Boltzmann's constant.

Thermal Cycling is caused by mismatched coefficients of thermal expansions for metallic and dielectric materials. It can result in inelastic deformations that eventually create cracks, fractures, and other related failures. The number of cycles to failure N_{TC} can be calculated using a Coffin-Mason equation [35]:

$$N_{TC} = A_{TC}(\delta T - T_{th})^{-b} e^{\frac{E_{a_{TC}}}{\kappa T_{max}}}, \tag{7.8}$$

where A_{TC} is an empirically determined constant and δT is the the thermal cycle amplitude. T_{th} is the temperature at which inelastic deformation begins. $E_{a_{TC}}$ and b are material related constants. T_{max} denotes the maximum temperature during a thermal cycle. Unlike the other three mechanisms, TC not only depends on temporal temperature but temperature variances, which make it even harder to study analytically.

Note that, given a set of operating parameters (in particular, operating temperature T and supply voltage V, etc.), the instantaneous MTTFs due to each failure mechanism can be immediately obtained.

7.2.2 Preliminary results

In this section, we provide some of our research's preliminary results. Figure 7.1 presents the overview of our simulation framework. Basically, this simulation platform consists of two major parts, 1) temperature modeling, 2) reliability modeling.

For temperature modeling, given a system architecture specification with a group of industrial benchmarks, timing simulators can gather the information of the uti-

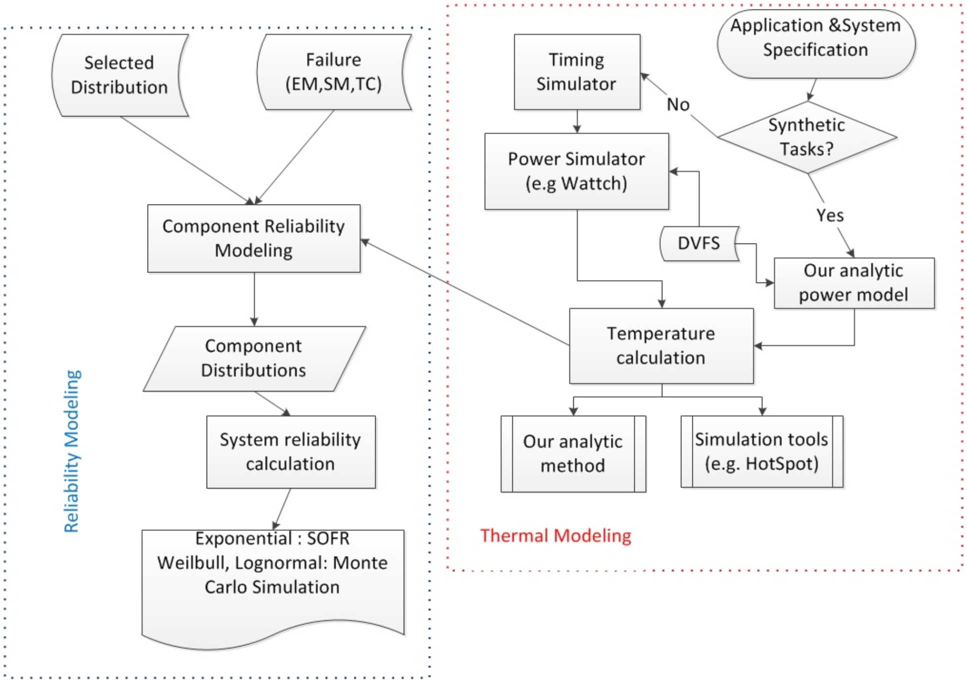

Selected Distribution
Failure (EM,SM,TC)
Component Reliability Modeling
Component Distributions
System reliability calculation
Exponential : SOFR Weilbull, Lognormal: Monte Carlo Simulation
Reliability Modeling
Timing Simulator
Application &System Specification
No
Synthetic Tasks?
Power Simulator (e.g Wattch)
Yes
DVFS
Our analytic power model
Temperature calculation
Our analytic method
Simulation tools (e.g. HotSpot)
Thermal Modeling

Figure 7.1: Simulation framework

lization trace on each core. With the utilization of each core and the specific DVFS scheme to be employed, power simulators, such as Wattch [22], can approximate the power consumption of each component in the system. Note that, for extensive simulations, synthetic task sets are randomly generated most of the time. On the other hand, we can use an analytic power modeling to directly derive the information regarding power consumption [121]. Then, after obtaining the power trace, we are able to get the corresponding thermal profiles either by using simulators like HotSpot [67] or by adopting analytic methods in [119, 121]. These methods are very accurate and much faster than HotSpot simulations. Therefore, in our simulations, we exclusively use analytic temperature calculation methods.

For reliability modeling, we first model components with respect to each failure mechanism given a selected distribution model (e.g. Exponential [113], Lognormal, and Weibull [126] etc.). Run-time temperature variations are then incorporated to obtain the distribution parameters [126, 65]. Consequently, we can complete the system reliability calculation. For a simple distribution, such as Exponential Distribution, sum-of-failure-rate (SOFR) can be used to get the system MTTF [113, 38] while for much complicated distributions, e.g Weibull and Lognormal, Monte Carlo simulation needs to be conducted [126].

First, we set the parameters of each failure mechanism according to [114] and the proportionality constants, e.g. A_{EM}, A_{SM}, and A_{TDDB}, were calculated such that the $MTTF$ due to each failure mechanism is 30 years at $70°C$.

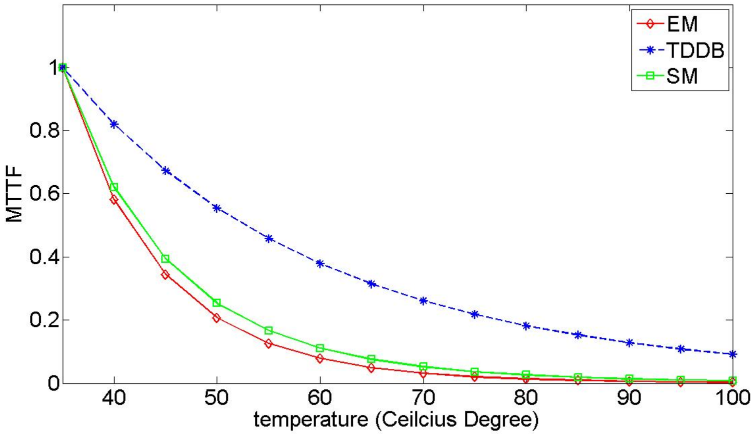
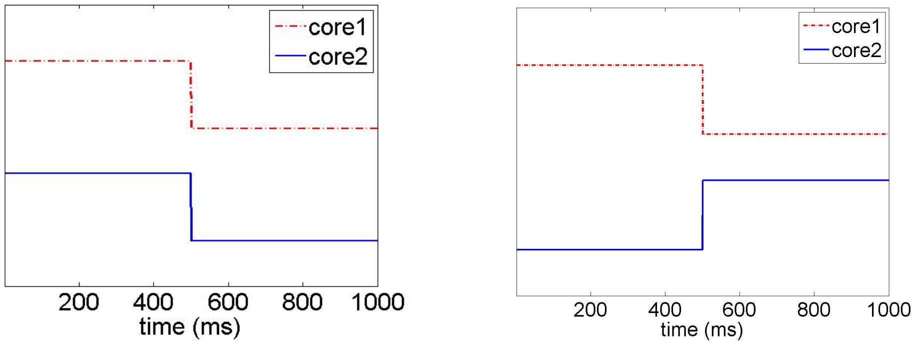


Figure 7.2: MTTF VS. Temperature

In Figure 7.2, we simply investigated the sensitivity of each failure mechanism to the change of temperature (TC was not considered since it also depends on temperature variations). For each failure type, the MTTFs was normalized to its reference MTTF at the ambient temperature, i.e. $35°C$. As can be seen, the lifetime reliability of a system with respect to each failure mechanism drops significantly as the temperature increases. Additionally, EM is the most sensitive failure mechanism with respect to temperature.

We adopted the method in [65, 126] to account for varying operating conditions (e.g. temperature, supply voltage) for obtaining system lifetime distribution. From



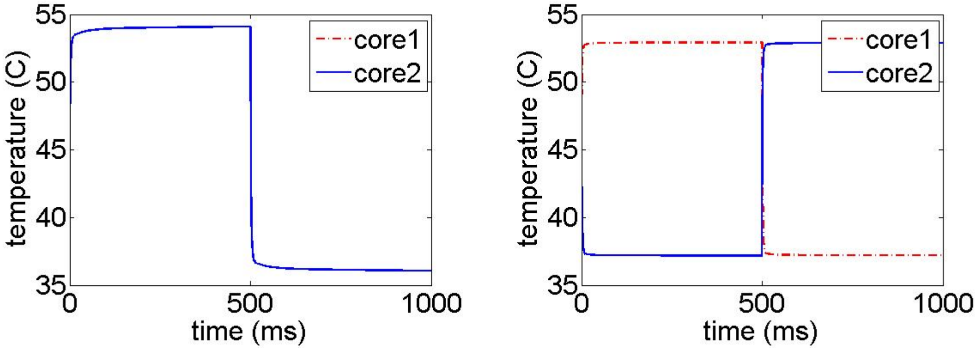
(a) Schedule1

(b) Schedule2

Figure 7.3: Speed Schedule

this point forward, we solely focused on EM since TDDB and SM can be dealt with in a similar manner. Next, we studied the impacts of power management on processor lifetime. For illustration purposes, we ran our experiments on a 2-core platform, and the thermal-related parameters were derived directly from HotSpot [67]. We used our technique in [121] to calculate the steady-state thermal profile of the system given a speed schedule. For example, we have a system of two cores where both cores are executing an identical periodic task with a period (deadline) of 1000ms and an execution time of 500ms. We utilized different two-speed schedules to execute the tasks. High speed and low speed were set to 1 and 0.4 (speeds are normalized to the highest speed available in the system), respectively. The interval length of each speed mode was determined in a way such that the task could finish exactly at its deadline. In the first schedule, both cores run simultaneously in high(low) speed mode in Figure 7.3(a) whereas the running modes on two cores are exactly opposite in Figure 7.3(b). We plotted the thermal profiles for both cores under these two different speed schedules in Figure 7.4, respectively.

As shown in Figure 7.4, for the first case, two cores have identical temperature values. On the contrary, the temperature traces oscillate differently in the second



(a) Schedule1

(b) Schedule2

Figure 7.4: Thermal profiles

case. Further, the former schedule has a peak temperature about $2.5°C$ higher than that of the latter.

Different thermal dynamics may have different lifetime reliability. As shown in Figure 7.5, the reliability of core 1(2) diminishes much faster under the second schedule due to the unfavorable temperature dynamics. Consequently, it results in a shorter MTTF.

Using the same example, we extended the "m-oscillation" technique that is well studied for peak-temperature reductions on single-core processors [61]. The main idea is to oscillate the speed of a core between high and low mode by m times. We changed the number of oscillations from 1 to 15 and started the two processors in different running mode. The results have been plotted in Figure 7.6(a) and 7.6(b). Note that, in Figure 7.6(b), MTTFs were normalized with respect to the system MTTF when $m = 1$.

As the figure shows, without careful provisions of the speed schedule on each core, increasing the number of oscillations does not lead to significant improvements in terms of reliability. Actually, the peak-temperature reduction improvement is not obvious either.

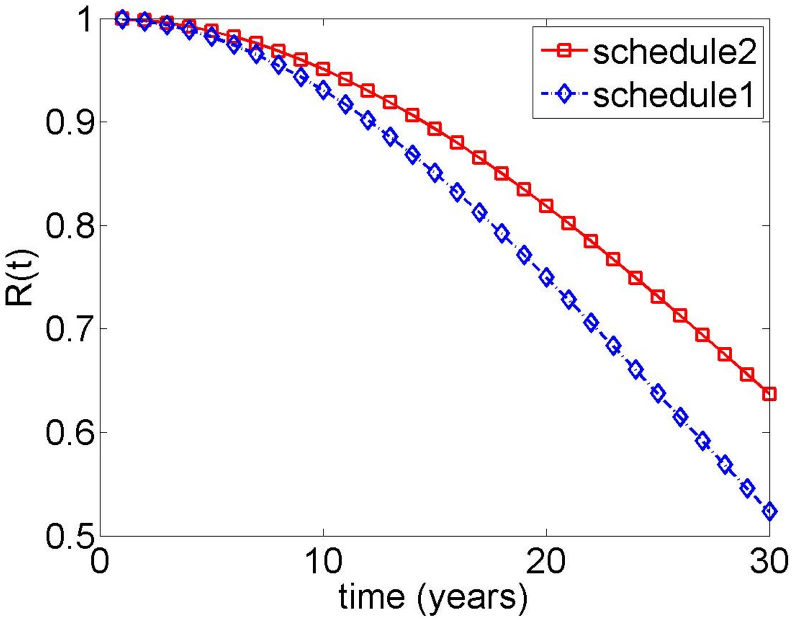
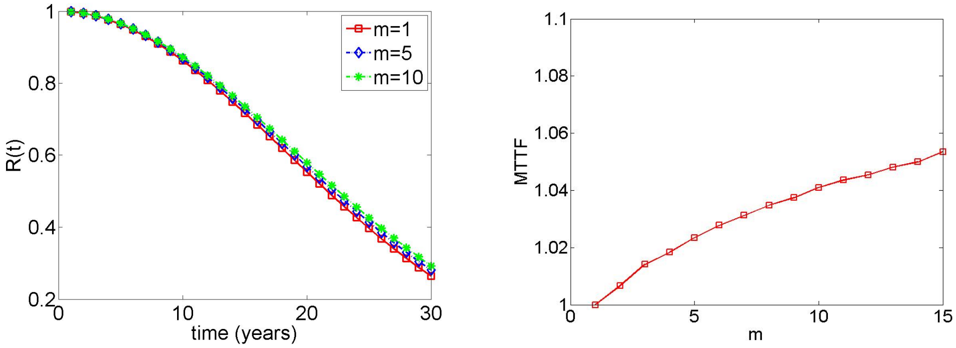


Figure 7.5: Reliability distribution for core 1(2)

TC is different from the other three failure mechanisms, and it strongly depends on temperature variances. With the aid of our temperature modeling technique in [121], we can efficiently obtain the steady-state temperature profiles, and we are able to study the TC mechanism effectively and accurately. The technique in [126] for calculating system MTTF due to TC was employed, and Rain Flow Counting was used to account for the thermal cycles within a thermal trace. Following the above example, the MTTF under schedule 2 is 1.5x of that of schedule 1. Again, we studied the impact of m-oscillation on system reliability, considering the TC mechanism exclusively. As evident in equation (7.8), the damage caused by temperature variations is determined by the constant exponent b which is material related. As shown in [72], the range of b is 1-3 for ductile metal, 3-5 for hard metal alloys, and as high as 6-9 for Si and dielectrics. We set b to 2 and 6, respectively. In this experiment, we again varied the number of oscillations. As can be observed, increasing the number of oscillation actually degrades the system reliability. This is due to the



(a) Reliability distribution for core 1(2) under m-oscillation

(b) MTTF for core 1(2) VS. m-oscillation

Figure 7.6: Impacts of m-oscillation on system reliability

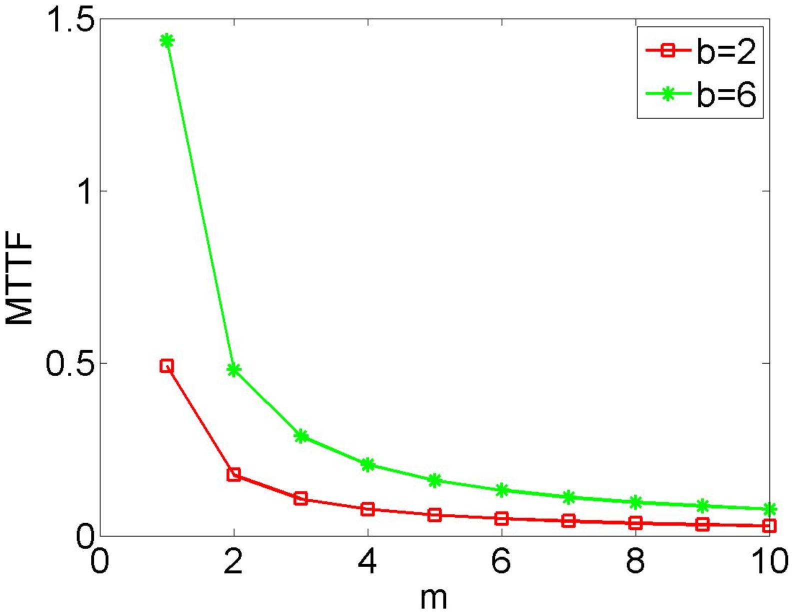


Figure 7.7: MTTF of TC vs. the number of oscillations

lack of efficiency of this simple m-oscillation scheme on multi-core platforms which encompasses oscillating the cores simultaneously and setting the speed randomly. With more available speeds and non-uniform workloads on each core, the problem becomes more complicated. Considering the fact that frequent speed switchings may lead to an increasing number of thermal cycles, we need to explore effective scheduling techniques to more substantially reduce thermal cycle amplitudes.

In summary, a simple variation of speed patterns of each core can dramatically impact the reliability of the system (Figure 7.5). Traditional techniques that are effective in reducing peak temperature and can implicitly improve the system reliability are becoming ineffective, if not detrimental, for the development of reliable multi-core platforms. This signifies the need of developing advanced techniques that explicitly take the lifetime reliability into account while considering other design constraints, e.g. timing and power.

BIBLIOGRAPHY

[1] Critical reliability challenges for the international technology roadmap for semiconductors (ITRS). Technical report, International SEMATECH, 2003.

[2] Behind the birth of m3. *IHS iSuppli*, 2012.

[3] Embedded system market - global industry analysis, size, share, growth, trends and forecast. *Transparency Market Research*, 2013.

[4] T. AlEnawy and H. Aydin. Energy-aware task allocation for rate monotonic scheduling. In *Real Time and Embedded Technology and Applications Symposium, 2005. RTAS 2005. 11th IEEE*, pages 213–223, March.

[5] AMD. Amd g-series.

[6] 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*, pages 33–40, July 2003.

[7] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick, M. J. Demmel, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, TECHNICAL REPORT, UC BERKELEY, 2006.

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

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

[10] S. Baruah. Partitioned edf scheduling: a closer look. *Real-Time Systems*, 49(6):715–729, 2013.

[11] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, and S. Stiller. Implementation of a speedup-optimal global edf schedulability test. In *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, pages 259–268, July 2009.

[12] S. Baruah and N. Fisher. Global deadline-monotonic scheduling of arbitrary-deadline sporadic task systems. In E. Tovar, P. Tsigas, and H. Fouchal, editors,

Principles of Distributed Systems, volume 4878 of *Lecture Notes in Computer Science*, pages 204–216. Springer Berlin Heidelberg, 2007.

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

[14] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 182–190, Dec 1990.

[15] R. Bergamaschi, G. Han, A. Buyuktosunoglu, H. Patel, I. Nair, G. Dittmann, G. Janssen, N. Dhanwada, Z. Hu, P. Bose, and J. Darringer. Exploring power management in multi-core systems. In *Design Automation Conference, 2008. ASPDAC 2008. Asia and South Pacific*, pages 708–713, March 2008.

[16] A. A. Bertossi, L. V. Mancini, and A. Fusiello. Fault-tolerant deadline-monotonic algorithm for scheduling hard real-time tasks. In *In Proceedings of the 11th International Parallel Processing Symposium*, pages 133–138. Bellingham, SPIE Press, 1997.

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

[18] W. L. Bircher and L. K. John. Analysis of dynamic power management on multi-core processors. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 327–338, New York, NY, USA, 2008. ACM.

[19] S. Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM.

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

[21] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 83–94, June 2000.

[22] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 83–94, 2000.

[23] A. Burchard, J. Liebeherr, Y. Oh, and S. Son. New strategies for assigning real-time tasks to multiprocessor systems. *Computers, IEEE Transactions on*, 44(12):1429–1442, Dec 1995.

[24] A. Burns. Scheduling hard real-time systems: a review. *Software Engineering Journal*, 6(3):116–128, May 1991.

[25] A. Burns, R. Davis, and S. Punnekkat. Feasibility analysis of fault-tolerant real-time task sets. In *Real-Time Systems, 1996., Proceedings of the Eighth Euromicro Workshop on*, pages 29–33, Jun 1996.

[26] A. Burns, R. Davis, P. Wang, and F. Zhang. Partitioned edf scheduling for multiprocessors using a c=d task splitting scheme. *Real-Time Systems*, 48(1):3–33, 2012.

[27] G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency (Series in Computer Science)*. Plenum Publishing Co., 2005.

[28] G. C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004.

[29] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. 2004.

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

[31] T. Chantem, Y. Xiang, X. S. Hu, and R. P. Dick. Enhancing multicore reliability through wear compensation in online assignment and scheduling. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 1373–1378, San Jose, CA, USA, 2013. EDA Consortium.

[32] B. Chattopadhyay and S. Baruah. Partitioned scheduling of implicit-deadline sporadic task systems under multiple resource constraints. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2012 IEEE 18th International Conference on*, pages 144–153, Aug 2012.

[33] V. Chaturvedi, H. Huang, and G. Quan. Leakage aware scheduling on maximum temperature minimization for periodic hard real-time systems. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1802–1809, June 2010.

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

[35] M. Ciappa, F. Carbognani, and W. Fichtner. Lifetime prediction and design of reliability tests for high-power devices in automotive applications. *Device and Materials Reliability, IEEE Transactions on*, 3(4):191–196, 2003.

[36] S. R. Corporation. International technology roadmap for semiconductors. 2010.

[37] S. R. Corporation. International technology roadmap for semiconductors. 2013.

[38] A. K. Coskun, T. Simunic, K. Mihic, G. D. Micheli, and Y. Leblebici. Analysis and optimization of mpsoc reliability. *J. Low Power Electronics*, 2(1):56–69, 2006.

[39] R. I. Davis and A. Burns. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Refuted, Revisited and Revised. Real-Time Systems*, 35:239–272, 2007.

[40] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, Oct. 2011.

[41] F. Eisenbrand, K. Kesavan, R. Mattikalli, M. Niemeier, A. Nordsieck, M. Skutella, J. Verschae, and A. Wiese. Solving an avionics real-time scheduling problem by advanced ip-methods. 6346:11–22, 2010.

[42] 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, CODES+ISSS '09, pages 193–202, New York, NY, USA, 2009. ACM.

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

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

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

[46] N. Fisher, J.-J. Chen, S. Wang, and L. Thiele. Thermal-aware global real-time scheduling on multicore systems. pages 131–140, April 2009.

[47] L. George and J.-F. Hermant. A norm approach for the partitioned edf scheduling of sporadic task systems. In *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, pages 161–169, July 2009.

[48] Y. Guo, D. Zhu, and H. Aydin. Reliability-aware power management for parallel real-time applications with precedence constraints. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–8, July 2011.

[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*, pages 62–71, Aug 2013.

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

[51] S. Hamdioui, D. Gizopoulos, G. Guido, M. Nicolaidis, A. Grasset, and P. Bonnot. Reliability challenges of real-time systems in forthcoming technology

nodes. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 129–134, 2013.

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

[53] C.-C. Han and H.-Y. Tyan. A better polynomial-time schedulability test for real-time fixed-priority scheduling algorithms. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, RTSS '97, pages 36–, Washington, DC, USA, 1997. IEEE Computer Society.

[54] Q. Han, M. Fan, L. Niu, and G. Quan. Energy minimization for fault tolerant scheduling of periodic fixed-priority applications on multiprocessor platforms. 2015.

[55] Q. Han, M. Fan, L. Niu, and G. Quan. Energy minimization for fault tolerant scheduling of periodic fixed-priority applications on multiprocessor platforms. In *Design, Automation and Test in Europe, DATE*, March 2015.

[56] Q. Han, M. Fan, and G. Quan. Energy minimization for fault tolerant real-time applications on multiprocessor platforms using checkpointing. In *Low Power Electronics and Design (ISLPED), 2013 IEEE International Symposium on*, pages 76–81, Sept 2013.

[57] Q. Han, L. Niu, G. Quan, S. Ren, and S. Ren. Energy efficient fault-tolerant earliest deadline first scheduling for hard real-time systems. *Real-Time Systems*, 50(5-6):592–619, 2014.

[58] M. 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*, pages 190–197, Oct 2011.

[59] M. Haque, H. Aydin, and D. Zhu. Energy management of standby-sparing systems for fixed-priority real-time workloads. In *International Green Computing Conference (IGCC)*, June 2013.

[60] M. Haque, H. Aydin, and D. Zhu. Energy management of standby-sparing systems for fixed-priority real-time workloads. In *Green Computing Conference (IGCC), 2013 International*, pages 1–10, June 2013.

[61] H. Huang, V. Chaturvedi, G. Quan, J. Fan, and M. Qiu. Throughput maximization for periodic real-time systems under the maximal temperature constraint. *ACM Trans. Embed. Comput. Syst.*, 13(2s):70:1–70:22, Jan. 2014.

[62] H. Huang, M. Fan, and G. Quan. On-line leakage-aware energy minimization scheduling for hard real-time systems. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 677–682, Jan 2012.

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

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

[65] L. Huang, F. Yuan, and Q. Xu. Lifetime reliability-aware task allocation and scheduling for mpsoc platforms. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 51 –56, april 2009.

[66] L. Huang, F. Yuan, and Q. Xu. On task allocation and scheduling for lifetime extension of platform-based mpsoc designs. *Parallel and Distributed Systems, IEEE Transactions on*, 22(12):2088–2099, Dec. 2011.

[67] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. Stan. Hotspot: a compact thermal modeling methodology for early-stage vlsi design. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(5):501–513, 2006.

[68] W.-L. Hung, Y. Xie, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Thermal-aware task allocation and scheduling for embedded systems. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, DATE '05, pages 898–899, Washington, DC, USA, 2005. IEEE Computer Society.

[69] Intel. Intel xeon processor.

[70] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the 1998 international symposium*

on Low power electronics and design, ISLPED '98, pages 197–202, New York, NY, USA, 1998. ACM.

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

[72] JEDEC Solid State Technology Association. Failure mechanisms and models for semiconductor devices. Technical Report JEP122C, March 2006.

[73] A. Kandhalu, K. Lakshmanan, J. Kim, and R. Rajkumar. pcompats: Period-compatible task allocation and splitting on multi-core processors. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 307–316, April 2012.

[74] J. Kang and S. Ranka. Dynamic slack allocation algorithms for energy minimization on parallel machines. *J. Parallel Distrib. Comput.*, 70(5):417–430, May 2010.

[75] K. C. Kapur and L. R. Lamberson. *Reliability in Engineering Design*. John Wiley & Sons, New York, 1977.

[76] D. Katcher, H. Arakawa, and J. Strosnider. Engineering and analysis of fixed priority schedulers. *Software Engineering, IEEE Transactions on*, 19(9):920–934, Sep 1993.

[77] S. Kato and N. Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. In *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pages 23–32, April 2009.

[78] D. Khudia and S. Mahlke. Harnessing soft computations for low-budget fault tolerance. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 319–330, Dec 2014.

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

[80] T.-W. Kuo and A. Mok. Load adjustment in adaptive real-time systems. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 160–170, Dec 1991.

[81] S. Lauzac, R. Melhem, and D. Mosse. An efficient rms admission control and its application to multiprocessor scheduling. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International ... and Symposium on Parallel and Distributed Processing 1998*, pages 511–518, Mar 1998.

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

[83] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 166–171, Dec 1989.

[84] P. Leteinturier. Multi-core processors: Driving the evolution of automotive electronics architectures. In *embedded.com*, 2007.

[85] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237 – 250, 1982.

[86] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20:46–61, January 1973.

[87] J. Liu. *Real-Time Systems*. Prentice Hall, NJ, 2000.

[88] Y. Liu, H. Liang, and K. Wu. Scheduling for energy efficiency and fault tolerance in hard real-time systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 1444 –1449, march 2010.

[89] F. Many and D. Doose. Scheduling analysis under fault bursts. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:113–122, 2011.

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

[91] B. Mochocki, X. Hu, and G. Quan. A unified approach to variable voltage scheduling for nonideal dvs processors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(9):1370 – 1377, sept. 2004.

[92] B. Mochocki, X. Hu, and G. Quan. A unified approach to variable voltage scheduling for nonideal dvs processors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(9):1370 – 1377, sept. 2009.

[93] S. Nassif, N. Mehta, and Y. Cao. A resilience roadmap. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 1011–1016, 2010.

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

[95] M. Pandya and M. Malek. Minimum achievable utilization for fault-tolerant processing of periodic tasks. *IEEE Trans. on Computers*, 47:1102–1112, 1994.

[96] P. Pop, V. Izosimov, P. Eles, and Z. Peng. Design optimization of time- and cost-constrained fault-tolerant embedded systems with checkpointing and replication. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(3):389–402, March 2009.

[97] P. Pop, K. H. Poulsen, V. Izosimov, P. Eles, and M. M. Dept. Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems, CODES+ISSS' 2007.

[98] D. K. Pradhan, editor. *Fault-tolerant computing: theory and techniques; vol. 1*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[99] D. K. Pradhan, editor. *Fault-tolerant computer system design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[100] X. Qi, D. Zhu, and H. Aydin. Global scheduling based reliability-aware power management for multiprocessor real-time systems. *Real-Time Syst.*, 47:109–142, March 2011.

[101] X. Qi, D. Zhu, and H. Aydin. Global reliability-aware power management for multiprocessor real-time systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2010 IEEE 16th International Conference on*, pages 183–192, Aug.

[102] G. Quan and X. Hu. Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 828–833, New York, NY, USA, 2001. ACM.

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

[104] M. Rausand and A. Hoyland. *System Reliability Theory: Models, Statistical Methods, and Applications, 2nd Edition*. WILEY, December 2003.

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

[106] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 389 – 398, 2002.

[107] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 389–398, 2002.

[108] A. Shye, J. Blomstedt, T. Moseley, V. Reddi, and D. Connors. Plr: A software approach to transient fault tolerance for multicore architectures. *Dependable and Secure Computing, IEEE Transactions on*, 6(2):135–148, April 2009.

[109] A. Shye, B. Scholbrock, and G. Memik. Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures. pages 168–178, 2009.

[110] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[111] R. Sridharan and R. Mahapatra. Reliability aware power management for dual-processor real-time embedded systems. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 819 –824, june 2010.

[112] J. Srinivasan, S. Adve, P. Bose, and J. Rivers. The impact of technology scaling on lifetime reliability. In *Dependable Systems and Networks, 2004 International Conference on*, pages 177 – 186, june-1 july 2004.

[113] J. Srinivasan, S. Adve, P. Bose, and J. Rivers. The impact of technology scaling on lifetime reliability. In *Dependable Systems and Networks, 2004 International Conference on*, pages 177–186, 2004.

[114] J. Srinivasan, S. Adve, P. Bose, and J. Rivers. Exploiting structural duplication for lifetime reliability enhancement. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pages 520 – 531, june 2005.

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

[116] Q. Tang, S. Gupta, and G. Varsamopoulos. Energy-efficient thermal-aware task scheduling for homogeneous high-performance computing data centers: A cyber-physical approach. *Parallel and Distributed Systems, IEEE Transactions on*, 19(11):1458–1472, Nov 2008.

[117] U. Tech. 2014 embedded marked study. *UBM Tech Electronics's Annual Survey of The Embedded Markets Worldwide*, 2014.

[118] K. W. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Syst.*, 6(2):133–151, Mar. 1994.

[119] I. Ukhov, M. Bao, P. Eles, and Z. Peng. Steady-state dynamic temperature analysis and reliability optimization for embedded multiprocessor systems. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 197–204, June 2012.

[120] R. Viswanath, V. Wakharkar, A. Watwe, V. Lebonheur, M. Group, and I. Corp. Thermal performance challenges from silicon to systems, 2000.

[121] T. Wang, M. Fan, G. Quan, and S. Ren. Heterogeneity exploration for peak temperature reduction on multi-core platforms. In *Quality Electronic Design (ISQED), 2014 15th International Symposium on*, pages 107–114, March 2014.

[122] T. Wei, X. Chen, and S. Hu. Reliability-driven energy-efficient task scheduling for multiprocessor real-time systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(10):1569 –1573, oct. 2011.

[123] T. Wei, P. Mishra, K. Wu, and H. Liang. Fixed-priority allocation and scheduling for energy-efficient fault tolerance in hard real-time multiprocessor systems. *Parallel and Distributed Systems, IEEE Transactions on*, 19(11):1511 –1526, nov. 2008.

[124] T. Wei, P. Mishra, K. Wu, and J. Zhou. Quasi-static fault-tolerant scheduling schemes for energy-efficient hard real-time systems. *J. Syst. Softw.*, 85(6):1386–1399, June 2012.

[125] Wikipedia. Moore's law.

[126] Y. Xiang, T. Chantem, R. Dick, X. Hu, and L. Shang. System-level reliability modeling for mpsocs. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 297 –306, oct. 2010.

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

[128] F. Zhang and A. Burns. Schedulability analysis for real-time systems with edf scheduling. *Computers, IEEE Transactions on*, 58(9):1250–1258, Sept 2009.

[129] S. Zhang, K. S. Chatha, and G. Konjevod. Near optimal battery-aware energy management. In *ISLPED*, pages 249–254, 2009.

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

[131] Y. Zhang, K. Chakrabarty, and V. Swaminathan. Energy-aware fault tolerance in fixed-priority real-time embedded systems. In *Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, ICCAD '03, pages 209–, Washington, DC, USA, 2003. IEEE Computer Society.

[132] B. Zhao, H. Aydin, and D. Zhu. Enhanced reliability-aware power management through shared recovery technique. In *Proceedings of the 2009 Interna-*

tional Conference on Computer-Aided Design, ICCAD '09, pages 63–70, New York, NY, USA, 2009. ACM.

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

[134] B. Zhao, H. Aydin, and D. Zhu. Energy management under general task-level reliability constraints. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 285 –294, april 2012.

[135] D. Zhu and H. Aydin. Energy management for real-time embedded systems with reliability requirements. In *Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on*, pages 528 –534, nov. 2006.

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

VITA

QIUSHI HAN

2009	B.S., Software Engineering Beijing Jiaotong University Beijing, China
2015	Ph.D. candidate, Electrical Engineering Florida International University Florida, USA

PUBLICATIONS

Qiushi Han, Ming Fan, Shaolei Ren, Gang Quan, (2015). *Temperature-Constrained Feasibility Analysis for Multicore Scheduling*, IEEE transaction on Computers (second round review).

Qiushi Han, Linwei Niu, Gang Quan, Shaolei Ren, Shangping Ren, (2014). *Energy efficient fault-tolerant earliest deadline first scheduling for hard real-time systems*, Journal,Real-Time Systems, Volume 50, Issue 5-6, Pages 592-619, Springer US.

Ming Fan, Qiushi Han, Gang Quan, Shangping Ren, (2014). *Enhanced fixed-priority real-time scheduling on multi-core platforms by exploiting task period relationship*, Journal of Systems and Software, Volume 99, Pages 85-96, Elsevier.

Qiushi Han, Tianyi Wang, Gang Quan, (2015). *Enhanced Fault-Tolerant Fixed-Priority Scheduling of Hard Real-Time Tasks on Multi-Core Platforms*, 21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (Accepted).

Qiushi Han, Ming Fan, Linwei Niu, Gang Quan, (2015). *Energy minimization for fault tolerant scheduling of periodic fixed-priority applications on multiprocessor platforms*, Proceedings of the 2015 Design, Automation & Test in Europe Conference (DATE), 830-835.

Ming Fan, Qiushi Han, Shuo Liu, Gang Quan, (2015). *On-line reliability-aware dynamic power management for real-time systems*, Quality Electronic Design (ISQED), 2015 16th International Symposium on, 361-365.

Ming Fan, Qiushi Han, Gang Quan, Shangping Ren, (2014). *Multi-core partitioned scheduling for fixed-priority periodic real-time tasks with enhanced RBound*, Quality Electronic Design (ISQED), 2014 15th International Symposium on, 284-291.

Qiushi Han, Ming Fan, Gang Quan, (2013). *Energy minimization for fault tolerant real-time applications on multiprocessor platforms using checkpointing*, IEEE International Symposium on Low Power Electronics and Design (ISLPED), 76–81. (best paper nomination)