IMPROVING RELIABILITY OF REAL-TIME EMBEDDED SYSTEMS

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by

Yue Ma

----------------------------------------

Xiaobo Sharon Hu, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
November 2018
IMPROVING RELIABILITY OF REAL-TIME EMBEDDED SYSTEMS

Abstract

by

Yue Ma

Multi-processor systems on a chip (MPSoCs) provide high performance and power efficiency. They have been widely used in many real-time embedded applications such as automotive electronics, industrial automation, and avionics. Most of these applications must satisfy deterministic or probabilistic timing constraints. However, due to CMOS technology scaling, MPSoCs increasingly have higher power density and temperature, which reduce system lifetime reliability. Meanwhile, the decreasing feature size of transistors and low supply voltage and frequency make MPSoCs more vulnerable to soft errors and degrade soft-error reliability. Maintaining quality of service, improving lifetime reliability and soft-error reliability, and satisfying real-time requirements have become major concerns in MPSoCs, especially for such systems deployed in harsh environments.

This thesis focuses on developing techniques to improve reliability for MPSoCs having different architectural features. Two methods are proposed to improve lifetime reliability and soft-error reliability respectively for homogeneous MPSoCs. The first method aims at improving lifetime reliability by dynamically reducing operating temperature. In order to maximize soft-error reliability and recover failed tasks caused by soft errors, the second method introduces a novel approach to dynamically determine which failed tasks should be recovered.

This thesis further studies reliability improvements for “big–little” type MPSoCs,
which are widely used in many real-time applications. For such systems, both soft-error reliability and lifetime reliability are key concerns. After exploring the power features of the “big” core and “little” core, a framework is developed to improve soft-error reliability while satisfying a lifetime reliability constraint. Based on the run-time cores’ frequencies and utilizations, this framework dynamically increases core frequencies and selects the most power efficient cores to execute tasks to achieve improved soft-error reliability.

Finally, this thesis also focuses on MPSoCs with integrated CPU and GPU. Thanks to the massively parallel computing ability offered by GPU and the low power design of CPU, this type of MPSoCs has been widely used in many real-time applications such as autonomous vehicles and robots. For such MPSoCs, lifetime reliability and soft-error reliability of both CPU and GPU need to be considered. Through detailed execution profiling, we reveal that for a task relying on GPU resources to complete, its execution time on CPU significantly increases if executing on a different CPU core from the operating system. Based on this observation, a framework is introduced to map tasks and manage core frequencies of both CPU and GPU in order to improve soft-error reliability while satisfying a lifetime reliability constraint.
# CONTENTS

Figures .................................................. v
Tables .................................................. vii
Acknowledgments ........................................ viii

Chapter 1: Introduction .................................. 1
  1.1 Reliability of Real-Time Embedded Systems ............. 1
  1.2 Challenges in Improving Reliability ..................... 3
    1.2.1 Temperature and thermal cycling affecting lifetime reliability . 3
    1.2.2 Lifetime reliability and soft-error reliability trade-off problem . 5
    1.2.3 Real-time, power, energy, and peak temperature constraints . 5
    1.2.4 Varying workload ................................ 6
  1.3 Contributions ....................................... 6
    1.3.1 Improving lifetime reliability through utilization and thermal management .................. 6
    1.3.2 Improving soft-error reliability through dynamic recovery .... 7
    1.3.3 Improving reliability of real-time systems on “big-little” type MPSoCs .................. 8
    1.3.4 Improving reliability of real-time systems on integrated CPU and GPU platforms ................ 9

Chapter 2: Improving Lifetime Reliability through Utilization and Thermal Management .................. 10
  2.1 Introduction .......................................... 10
  2.2 Related Work ......................................... 12
  2.3 System Models ........................................ 14
    2.3.1 Hardware platform ................................ 14
    2.3.2 Task model ...................................... 14
    2.3.3 System-level lifetime reliability .................. 15
  2.4 Problem Formulation and Framework Overview ............ 18
    2.4.1 Problem formulation .............................. 18
    2.4.2 Framework overview ................................ 19
  2.5 Reliability-Aware Utilization Control Details .......... 20
    2.5.1 Dynamic utilization model ....................... 21
2.5.2 Utilization and thermal management .................................. 22
2.5.3 Sampling window controller design ................................... 26
2.6 Experimental Setup ......................................................... 28
  2.6.1 Tasks ................................................................. 29
  2.6.2 Comparison targets ................................................. 30
  2.6.3 Experimental platforms .......................................... 30
2.7 Experimental Evaluation .................................................. 33
  2.7.1 Experiments on TK1 chip ........................................... 35
  2.7.2 Simulation results .................................................. 37
2.8 Summary ............................................................................ 39

Chapter 3: Improving Soft-Error Reliability through Dynamic Recovery ...... 41
  3.1 Introduction .................................................................. 41
  3.2 Preliminaries ............................................................... 43
    3.2.1 Task model ......................................................... 43
    3.2.2 Soft-error reliability ............................................ 44
    3.2.3 Lifetime reliability .............................................. 45
    3.2.4 Motivating examples ............................................ 45
  3.3 Problem Formulation and Solution Overview .............................. 48
    3.3.1 Problem formulation ............................................. 48
    3.3.2 System-level soft-error reliability .......................... 49
    3.3.3 Overview of framework ....................................... 51
  3.4 Improving Soft-Error Reliability ....................................... 52
    3.4.1 A fast scheduling algorithm for a specific type of task sets ... 52
    3.4.2 A general scheduling algorithm .............................. 56
  3.5 Lifetime Reliability Satisfaction ....................................... 58
  3.6 Evaluation .................................................................... 61
    3.6.1 Experimental setup .............................................. 61
    3.6.2 Experimental results ............................................ 62
  3.7 Summary .................................................................... 65

Chapter 4: Improving Reliability of Real-Time Systems on “Big-Little” Type
        MPSocs ................................................................. 67
  4.1 Introduction ............................................................... 67
  4.2 Related Work .............................................................. 69
  4.3 System Model and Problem Formulation ................................ 71
    4.3.1 Hardware model .................................................. 71
    4.3.2 Task model ........................................................ 72
    4.3.3 Soft-error reliability ............................................ 73
    4.3.4 Lifetime reliability .............................................. 74
  4.4 “Big-Little” Type MPSocs and Core Power Features ................... 75
  4.5 Problem Formulation and Framework Overview ....................... 79
    4.5.1 Problem formulation ............................................. 80
    4.5.2 Overview of reliability improvement framework .......... 83
FIGURES

1.1 The amplitude, period, and peak temperature of thermal cycling. . . 4
1.2 The system-level lifetime reliability with different thermal profiles. . 4

2.1 System-level lifetime reliability depends on thermal cycle (a) ampli-
tude, (b) period, and (c) peak temperature, and on (d) temperature
without thermal cycles. . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.2 The percentage error and overhead due to Monte Carlo trials. . . . 17
2.3 High-level overview of RUC. . . . . . . . . . . . . . . . . . . . . . . 21
2.4 Temperature profile with four fully loaded cores running at the highest
frequency. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
2.5 Steady-state temperature with (a) different numbers of fully loaded
cores running at different frequencies and (b) one core running at dif-
fferent frequencies and utilizations. . . . . . . . . . . . . . . . . . . . . 33
2.6 Temperature readings from thermal sensors and estimated by Hotspot
for various core frequencies. . . . . . . . . . . . . . . . . . . . . . . . . . 34
2.7 Normalized MTTF with TK1 and constructed simulator. . . . . . . 34
2.8 Normalized MTTF and percentage of feasible solutions when tasks are
frame-based and running on TK1. . . . . . . . . . . . . . . . . . . . . . . 35
2.9 Normalized MTTF and percentage of feasible solutions when tasks’
periods are random and running on TK1. . . . . . . . . . . . . . . . . . . 36
2.10 Normalized MTTF and percentage of feasible solutions when tasks are
frame-based and running on a TK1 based simulator. . . . . . . . . . . . 37
2.11 Normalized MTTF and percentage of feasible solutions when tasks’
periods are random and running on a TK1 based simulator. . . . . . . . 38
2.12 Normalized MTTF and percentage of feasible solutions when running
on a simulator with high power density chips with (a) frame-based
tasks; (b) tasks with random periods. . . . . . . . . . . . . . . . . . . . . 39

3.1 Motivating examples illustrate dynamic recovery allocation is better
than static recovery allocation, and the scheduling can affect system-
level soft-error reliability. . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
3.2 Normalized probability of failures when the task set has 5 randomly
generated tasks. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
3.3 Normalized probability of failures when task set is composed of (a) 10 randomly generated tasks; (b) tasks from MiBench benchmark suite.  
3.4 Comparison of RIF-FRIS and RIF-GRIS with different lengths of slack.  
4.1 The power consumption of an HP (Denver) core and an LP (ARM) core on TX2 under different utilization and frequency levels.  
4.2 The steady state temperature of an HP core and an LP core on TK1 under different utilization and frequency levels.  
4.3 High-level overview of DRIF.  
4.4 Probability of failures due to soft errors and percentage of feasible solutions for a frame-based task set running on TK1.  
4.5 Probability of failures due to soft errors and percentage of feasible solutions for a general periodic task set running on TK1.  
4.6 Probability of failures due to soft errors and percentage of feasible solutions for a frame-based task set running on TX2.  
4.7 Probability of failures due to soft errors and percentage of feasible solutions for a general periodic task set running on TX2.  
5.1 The measured additional CPU times on TK1 (in milliseconds and percentage) if a task executed on different CPU cores with the operating system.  
5.2 The measured additional CPU times on TK1 (in millisecond and percentage) if a task executed on different CPU cores with the I/O service xorg.  
5.3 The measured additional CPU times on TX2 (in milliseconds and percentage) if a task executed on different ARM cores with the operating system.  
5.4 The measured additional CPU times on TX2 (in millisecond and percentage) if a task executed on different CPU cores with the I/O service xorg.  
5.5 The measured additional CPU times on TK1 (in milliseconds and percentage) if an CUDA task calling the CUDA API, cudaMemcpy, different times.  
5.6 The measured additional CPU times on TK1 (in milliseconds and percentage) if copying different size of data between CPU’s and GPU’s memory space.  
5.7 Tasks rely on the OS and I/O services to complete.  
5.8 High-level overview of HyFRO.  
5.9 The probability of failures for (a) the CPU and (b) the GPU.
### TABLES

2.1 Real-Time Tasks from MiBench Benchmark Suite ........................................ 29
2.2 Parameters in Power Model ........................................................................... 32
3.1 Average Execution Times of Tasks ................................................................. 62
4.1 Task Mapping and Migration Guideline for TX2 ............................................. 78
4.2 Task Mapping and Migration Guideline for TK1 ............................................. 79
4.3 Tasks’ Execution Times on TK1 ..................................................................... 96
4.4 Tasks’ Execution Times on TX2 ..................................................................... 96
4.5 The Task Allocation for TX2 .......................................................................... 97
5.1 Eight CUDA Tasks in Experiments ................................................................. 113
5.2 The measured additional GPU Time on TK1 ............................................... 114
5.3 CPU, GPU, and Additional CPU Times of Tasks on TX2 ............................. 130
I would like to thank my advisor, Dr. Xiaobo Sharon Hu, for her guidance, understanding, and strong support for my graduate study and future career. Dr. Hu has always made time for me and have taught me the value of hard work and persistence. Thanks for taking me on as a graduate student to a researcher today. Studying as her student is one of the best choices in my life.

I would like to thank Dr. Robert P. Dick, Dr. Micheal Niemier, and Dr. Yiyu Shi for their serving on my Ph.D. dissertation committee. Dr. Dick has given me lots of valuable and inspiring suggestions in my research. Dr. Niemier and Dr. Shi taught me how to be a good lecturer and gave me important suggestions in my Ph.D. candidacy oral exam.

I would like to thank Dr. Thidapat Chantem, Dr. Shige Wang, and Dr. Dong Wang. I am grateful for their wisdom, high standards, and inspiring suggestions.

Thanks to my colleagues whom I have received help from (in no particular order): Dr. Junlong Zhou, Dr. Shengyan Hong, Dr. Kai Xiao, Dr. Li Tang, Dr. Indranil Palit, Dr. Robert Perricone, Qiuwen Lou, Xunzhao Yin, and Yue Zhang.

Thanks to my family and friends. Thank you mom and dad for your unconditional love and silent support in my life, regardless of good days or bad days. I love you all. I also owe a big thank to all my friends around the world.

This work is supported in part by National Science Foundation (NSF) under grant numbers CNS-1319904, CNS-1319718, and CNS-1319784.
CHAPTER 1

INTRODUCTION

Multi-processor systems on a chip (MPSoCs) offer good performance with reasonable power consumption and are widely used in many real-time applications such as automotive electronics, industrial automation, and avionics [62, 107, 46, 53, 39, 20, 87, 120, 64, 65, 33, 58, 10]. For embedded systems used in such applications, both lifetime reliability and soft-error reliability are major concerns. Meanwhile, these applications often require the associated tasks completed by specified deadlines to ensure certain level of Quality of Service (QoS). The goal of this thesis is to improve lifetime reliability and/or soft-error reliability of real-time embedded systems. This chapter first presents the lifetime reliability and soft-error reliability requirements. Then, it discusses the challenges in improving reliability and describes the major contributions of this thesis.

1.1 Reliability of Real-Time Embedded Systems

The increase in the number of transistors per chip has greatly improved processor performance. However, the corresponding increase in power density leads to elevated operating temperature and large and frequent temperature variations. For a given transistor, the lifetime reliability due to permanent fault, measured by mean time to failure (MTTF), is temperature dependent. Generally, a higher temperature increases the effects of the gradual displacement and mass transportation of the atoms in metal wires, and the deterioration of the gate dielectric layer. Changes in temperature over time also influence reliability. Since integrated circuit materials have different thermal
expansion coefficients, thermal fluctuation produces mechanical stress, which leads to failures [99, 100, 8].

The system-level lifetime reliability is related to not only the operating lifespan of each transistor, but also the number of transistors, the distribution of permanent faults, and the mechanisms of fault propagation [108, 67, 37]. Hence, reliability modeling and optimization techniques should consider these divergent wear processes and the microarchitecture of a chip.

Soft errors, also referred to as single event upsets, are caused by high energy neutrons resulting from cosmic rays colliding with particles in the atmosphere [96, 96, 69, 26, 110, 93, 117]. Soft errors, which are transient, do not permanently damage or reduce the lifetime of circuit. However, they can fail the executing tasks. Soft-error reliability is used to describe the probability that tasks fail due to soft errors, and generally, a higher core frequency leads to a low soft error rate as well as high soft-error reliability [93, 3, 117, 118]. Although some techniques, such as error-correcting codes and parity, are proposed to protect system avoiding soft errors, improving soft-error reliability and reducing soft error rate are still necessary especially for safety-critical applications.

To reduce the cost of repairing or replacing broken systems, maintain quality of service, and guarantee more tasks can complete successfully, improving lifetime reliability and soft-error reliability becomes a major concern in MPSoCs, especially for those deployed in harsh environments. However, reducing power, energy, and temperature to improve lifetime reliability may reduce soft-error reliability, and vice versa. Hence, the design to improve a system’s reliability should consider this lifetime reliability and soft-error reliability tradeoff. In the next section, we discuss the challenges that should to be addressed in improving soft-error reliability and/or lifetime reliability.
1.2 Challenges in Improving Reliability

This thesis tackles four important challenges in improving reliability of real-time embedded systems. First of all, both the temperature and thermal cycling can affect lifetime reliability. Secondly, improving lifetime reliability and soft-error reliability is a trade-off problem. Thirdly, most embedded systems should satisfy the real-time, power/energy, and peak temperature constraints. Finally, system’s workload may vary at runtime. Each of these challenges is briefly discussed below.

1.2.1 Temperature and thermal cycling affecting lifetime reliability

In this thesis, we consider four main circuit failure mechanisms: electromigration (EM), time dependent dielectric breakdown (TDDB), stress migration (SM), and thermal cycling [99]. EM refers to dislocation of metal atoms caused by momentum imparted by electrical current in wires and vias. TDDB is the deterioration of the gate oxide layer. SM is caused by the directionally biased motion of atoms in metal wires due to mechanical stress caused by thermal mismatch among metal and dielectric materials. Wear due to EM, SM, and TDDB is strongly dependent on temperature.

Thermal cycling refers to wear caused by thermal stress resulting from mismatched coefficients of thermal expansion for adjacent material layers. Run-time temperature variation results in inelastic deformation, eventually leading to failure. For thermal cycling, we use the amplitude, period, and peak temperature to describe its features (see Fig. 1.1). Fig. 1.2 illustrates that for thermal profiles with different periods, amplitudes, and peak temperatures, the lifetime reliability, measured by MTTF, can be very different. Hence, methods that only minimize average temperature or peak temperature sometimes increase thermal cycling, so it may reduce lifetime reliability in the end. All lifetime reliability improvement techniques and frameworks proposed in this thesis consider both temperature and thermal cycling.
Figure 1.1. The amplitude, period, and peak temperature of thermal cycling.

MTTF: 3.2 years

MTTF: 0.6 years

MTTF: 2.5 years

MTTF: 1.6 years

Figure 1.2. The system-level lifetime reliability with different thermal profiles.
1.2.2 Lifetime reliability and soft-error reliability trade-off problem

An efficient method to improve lifetime reliability is to reduce operating temperature and thermal cycling by using dynamic voltage and frequency scaling (DVFS) technique. However, a lower core frequency leads to a higher soft error rate. For a given hardware platform and workload, reducing core frequency is efficient to reduce operating temperature and may improve lifetime reliability, but also leads to a high soft error rate and a low soft-error reliability.

Besides increasing core frequency, allocating recoveries to failed tasks, i.e., allowing a failed task re-execution, is another efficient method to improve soft-error reliability. However, allocating recoveries increases the system’s overhead and causes a high operating temperature, which leads to a lower lifetime reliability. The techniques to improve soft-error reliability, regardless of increasing core frequency or allocating recoveries to failed tasks, reduce lifetime reliability. Hence, improving lifetime reliability and soft-error reliability is a trade-off problem.

1.2.3 Real-time, power, energy, and peak temperature constraints

To achieve better performance and power efficiency, MPSoCs have been widely used in many applications, such as in-vehicle infotainment systems [62, 53, 39, 18], avionics [33, 10], industrial automation [87, 120], and mobile devices [2, 64, 65, 74, 52]. In these systems, tasks are expected to complete successfully before their deadlines. At the same time, MPSoCs in such applications are usually driven by battery. Hence, it is necessary to reduce power and/or energy consumption. Meanwhile, in order to avoid thermal throttling and protect MPSoCs from overheating, the peak operating temperature must be smaller than a temperature constraint.

For a real-time embedded system, all these constraints should be satisfied, but they are tradeoff. In order to guarantee the real-time constraint, one typical method is to increase the core frequency. However, a higher core frequency leads to a higher
power and energy consumption as well as a higher temperature. In this thesis, we address these trade-off problems and propose to satisfy all the real-time, power/energy, and peak temperature constraints.

1.2.4 Varying workload

In some applications, such as mobile applications [64, 65] and in-vehicle information systems [62, 53, 39], the workload may vary at runtime, and it is hard to predict offline. Hence, methods to improve reliability must be adaptive to the varying of workload, otherwise, it may violate the real-time, power/energy, and peak temperature constraints. At the same time, the computational complexity of such methods should be low enough to use online. In this thesis, we consider this low complexity requirement and develop multiple computing efficient methods and frameworks to improve lifetime reliability and/or soft-error reliability.

1.3 Contributions

The contributions of this thesis is summarized in this section. Two methods are developed to improve lifetime reliability and soft-error reliability respectively for homogeneous MPSoCs. We also develop frameworks to improve reliability for “bit–little” type MPSoCs and MPSoCs with integrated CPU and GPU.

1.3.1 Improving lifetime reliability through utilization and thermal management

Since system-level lifetime reliability is less explored, we first contribute to improve such reliability of real-time embedded systems. Considering that increasing utilization via decreasing core frequencies reduces core temperature, we design a model predictive controller (MPC) based method that keeps system utilization at a desired value. This method is effective to reduce operating temperature as well as satisfying the real-time requirement.
Based on this method, we design an on-line framework for homogeneous MPSoCs to improve system-level lifetime reliability. By periodically invoking this MPC based method, our framework reduces the effects of EM, SM, and TDDB. The length of the period in invoking this method can affect i) lifetime reliability via peak temperature and thermal cycling and ii) the MPC computational overhead. Hence, we develop a heuristic algorithm in our framework to dynamically adjust the length of period to balance these considerations.

We have conducted a large set of experiments on Nvidia’s TK1 chip [76] and a simulator to validate our approach and compared it with two existing control methods: temperature-aware [31] and utilization control [57]. For 10 tasks from MiBench benchmark suite [25], our proposed framework improves MTTF by 50% on average, and by up to 141% for processors with high power density.

1.3.2 Improving soft-error reliability through dynamic recovery

In this work, we are interested in improving soft-error reliability through dynamic recovery. Since the probability that a task fails twice is generally much lower than the probability that the task fails once, recovery is quite efficient in increasing soft-error reliability [111, 116]. Hence, a dynamic recovery allocation method is developed. In this method, slack is shared and any failed task can be recovered if there is sufficient amount of remaining slack for re-executing the failed task.

Based on this method, we introduce a novel framework for homogeneous MPSoCs to improve system-level soft-error reliability while satisfying real-time and lifetime reliability constraints. This framework schedules tasks to guarantee more tasks can be recoveries. Meanwhile, core frequencies are dynamically scaled to guarantee the peak temperature, real-time, and lifetime reliability constraints.

We have evaluated our framework in a simulator, which is constructed based on the Nvidia’s TK1 chip [76]. For both randomly generated tasks and tasks from the
MiBench benchmark suite [25], this framework reduces the probability of failure by at least 8% and 73% on average when compared to existing approaches.

1.3.3 Improving reliability of real-time systems on “big–little” type MPSoCs

Our next contribution improves reliability of the “big–little” type MPSoCs. To address power and energy concerns, this “big–little” architecture is composed of pairs of high-performance (HP) cores and low-power (LP) cores [107, 5, 82]. We introduce an on-line framework, referred to as DRIF (for Dynamic Reliability Improvement Framework), to improve soft-error reliability while satisfying lifetime reliability, real-time, power consumption, and peak operating temperature constraints. After exploiting the power and performance features of the “big” and “little” cores, we dynamically activate the most power efficient cores to execute tasks, which is efficient to reduce power consumption and operating temperature.

In this work, we first experimentally establish a migration guideline for migrating tasks between HP and LP cores based on power and performance features. We then propose a computationally efficient method to determine whether a given temporal thermal profile would result in the corresponding lifetime reliability to be larger than a threshold. Based on this tool, we develop an on-line framework to maximize soft-error reliability under real-time, peak operating temperature, power, and lifetime reliability constraints by scaling cores’ frequencies and selecting the most power efficient cores to execute tasks.

We have implemented and validated DRIF on two chips: Nvidia’s TK1 [76] and TX2 [78]. Based on the results obtained from running tasks in MiBench benchmark suite [25], we show that our framework can increase the no soft error occurring time at least 2 more days than existing approaches.
1.3.4 Improving reliability of real-time systems on integrated CPU and GPU platforms

MPSoCs consisting of integrated CPU and GPU are desirable platforms for real-time embedded applications requiring massively parallel processing capabilities. This work systematically addresses reliability concerns for real-time systems running on this type MPSoCs. We aim at solving the problem of maximizing soft-error reliability under real-time, lifetime reliability, and peak temperature constraints [61]. Since Nvidia’s chips have been used in many real-world applications [80], we are especially interested in CUDA tasks, but our proposed framework can also be readily applied to other GPUs and programming models.

In order to improve reliability for such MPSoCs, we first explore how the mapping of CUDA tasks affects the tasks’ CPU times and then develop a hybrid framework to achieve the design target. This framework i) statically maps tasks to CPU cores to improve soft-error reliability, ii) dynamically migrates tasks among CPU cores to achieve a higher lifetime reliability by balancing the wear states of cores, and iii) dynamically scales frequencies of CPU and GPU cores to increase soft-error reliability under temperature, real-time, and lifetime reliability constraints.

We have implemented and evaluated our framework on Nvidia’s TX2 chip [78]. Experimental results show that this framework allows the CPU and GPU to execute without soft errors for about 2 days and 42 days longer on average than existing approaches, respectively.
CHAPTER 2

IMPROVING LIFETIME RELIABILITY THROUGH UTILIZATION AND THERMAL MANAGEMENT

This chapter addresses the problem of maximizing system-level lifetime reliability, an important design consideration for many real-time embedded systems. A model predict controller based utilization and thermal management method is proposed to reduce operating temperature by manipulating core frequencies. Integrated with this method, an on-line framework that maximizes system-level lifetime reliability is proposed in this chapter. It selectively employs a comprehensive reliability estimation tool to deal with a variety of failure mechanisms at the system level. This framework also adjusts the controller sampling window length to decrease the reliability effects of thermal cycling.

2.1 Introduction

MPSoCs offer good performance with reasonable power consumption and are widely used in many real-time applications such as automotive electronics, industrial automation, and avionics [62, 107, 46, 53, 39]. Ongoing increases in device densities and operating frequencies increase the power density and temperature. Generally, a higher temperature leads to a lower lifetime reliability due to the increasing effects of the gradual displacement and mass transportation of the atoms in metal wires, and the deterioration of the gate dielectric layer. Changes in temperature over time also influence lifetime reliability. Hence, lifetime reliability modeling and optimization techniques should consider these divergent wear processes.
Most existing work on temperature management focuses on improving lifetime reliability by minimizing chip’s peak temperature [28, 40, 12, 31] or increasing the quality of service given a temperature constraint [11, 90, 41, 94]. However, minimizing peak temperature sometimes implies increasing thermal cycling, so it may reduce lifetime reliability [13]. The lifetime reliability maximization problem has its own unique characteristics and requires an in-depth analysis of the temporal temperature profile, including its peaks and valleys.

There exist several efforts that directly address increasing lifetime reliability via task mapping, scheduling [43, 44, 21, 22], and dynamic voltage and frequency scaling [64, 23]. However, the complexity of the lifetime reliability and thermal models (often based on Monte Carlo simulations [108, 38, 37, 116]) result in high execution overheads, making them impractical for online use. In order to reduce overhead, a common strategy is to use device-level MTTF model to directly replace the system-level model [43, 44, 21, 22, 24]. This, however, is a potentially inaccurate approximation as the temporal fault distributions of individual devices and million-device systems differ greatly. Only system-level lifetime reliability models can accurately determine system-level MTTF.

In this work, we first develop a model predictive controller (MPC) based utilization and thermal management method to reduce operating temperature, which is effective in improve lifetime reliability. Integrated with this method, we introduce an on-line framework called reliability-aware utilization control (RUC) to improve the lifetime reliability in the presence of wear caused by failure mechanisms that strongly depend on temperature and temperature variation, such as EM, TDDB, SM, and thermal cycling. We pay special attention to reduce the complexity of thermal management and lifetime reliability estimation, enabling on-line, in-system reliability estimation.

Our main contributions follow.
• Based on the observation that increasing utilization via decreasing core frequencies reduces core temperature, we design a MPC based utilization and thermal management method that keeps system utilization at a desired value. Our algorithm deployed in MPC incorporates various design considerations including real-time constraints and MTTF dependency on peak temperature and load balancing.

• We develop a framework, which utilizes the method to reduce the effects of EM, SM, and TDDB by achieving a low operating temperature. This framework also dynamically adjusts the length of period in invoking the above method, thereby influencing (i) the system-level lifetime reliability via peak temperature and thermal cycling and (ii) the MPC computational overhead. Our heuristic algorithm balances these considerations.

• We establish a simulator whose parameters are calibrated based on the measurement from Nvidia’s TK1 chip [76]. The simulator’s parameters are calibrated based on the measurement from TK1, but it can be with a higher and/or lower power density and allows cores executing at different core frequencies. We implement our on-line framework on both TK1 and the simulator.

We have conducted a large set of experiments on the Nvidia’s TK1 chip [76] and above simulator to validate our approach and compared it with existing frameworks. For 10 tasks from MiBench benchmark suite [25], our framework improves MTTF by 50% on average, and by up to 141% for processors with high power density.

2.2 Related Work

Researchers have modeled system lifetime reliability in several ways, from device level to system level. An architecture-level model was designed to calculate a processor’s lifetime due to EM, SM, TDDB, and thermal cycling [99]. The model was used to analyze the effects of CMOS technology scaling on system lifetime [99]. A statistical model and simulation methodology were used to determine multi-core system-on-chip reliability [16]. System lifetime has also been estimated in the presence of simulated sequences of failures [38]. A fast and accurate Monte Carlo based modeling framework that integrates device-, component-, and system-level models was proposed [108].
Several papers have described using the above models and hardware to improve lifetime reliability through offline optimization strategies. For periodic tasks running on a multiprocessor system-on-chip, Huang et al. proposed an analytical model to estimate the lifetime reliability of MPSoCs and a task mapping and scheduling algorithm to guard against aging effects [43]. Based on the same model, Das et al. extended the work to improve the lifetime of network-on-chips and also solve the energy-reliability tradeoff problem for multimedia MPSoCs [21, 22]. Based on a system-level lifetime reliability model, Zhou et al. presented a unified metric combining system-level lifetime reliability and soft-error reliability, and maximized the soft-error reliability under the lifetime reliability constraint [116]. These approaches can improve lifetime reliability but require that tasks and their timing and power characteristics be known at design time, and remain static. Our work supports tasks with characteristics that are unknown at design time and change during system operation, using dynamic voltage and frequency scaling as a mechanism.

Several dynamic reliability management approaches have been developed. Hartman et al. describe a run-time task mapping technology to maximize system-level lifetime reliability by utilizing wear sensors [37]. However, wear sensors are very frequently unavailable or inaccessible to system software. Bolchini et al. dynamically determined the most effective mapping of tasks to minimize network-on-chip energy consumption and maximize the lifetime [6]. Mandelli et al. proposed a run-time distributed energy-aware mapping technique that balances thermal distribution and improves reliability [63]. Bolchini et al. analyzed how load distribution strategies influence system lifetime reliability [7]. Das et al. proposed a machine learning based algorithm to handle inter- and intra-application variations and reduce peak temperature and thermal cycling [24]. All the above work improves lifetime reliability and captures variations in workload and run-time environments. However, none of them use a system-level reliability model, potentially undermining accuracy and fidelity.
In this work, we control a core’s frequency and dynamically determine the length of the control period to improve the system-level reliability of real-time embedded systems.

2.3 System Models

In this section, we present the hardware platform as well as the task and lifetime reliability models.

2.3.1 Hardware platform

We focus on homogeneous MPSoCs. Let $M = \{\rho_1, \ldots, \rho_m\}$ denote the set of $m$ cores in a given system. Power consumption is the fundamental cause of rising temperature and hence system failure. A core dissipates idle power when it is idle and consumes additional active power when it performs operations [31]. Both active and idle power are related to the core’s frequency. Let the utilization of a core in a given time interval $\Delta t$ be $u = \frac{\Delta t_a}{\Delta t}$, where $\Delta t_a$ is the amount of time that the core executes operations [31]. Run-time temperature can be estimated by using an RC thermal modeling tool or measured by thermal sensors. We use both thermal sensors and Hotspot [98], an integrated circuit thermal modeling tool, in experimental evaluations when validating our approach.

2.3.2 Task model

We consider independent real-time tasks with soft deadlines. A task that misses its deadline is immediately terminated. Tasks are periodic and follow partitioned scheduling, i.e., they are mapped to cores statically and no migration is allowed. Furthermore, tasks can be arbitrarily preempted. Such an execution model is prevalent in real-time systems since it incurs low development cost and runtime overhead.
Tasks on each core are scheduled according to a real-time scheduling policy such as earliest deadline first or rate monotonic scheduling [55].

We use $\tau_i$ to denote the $i^{th}$ task. Since all the jobs of the $i^{th}$ task have the same properties, $\tau_i$ also denotes the jobs of the $i^{th}$ task. $\tau_i$ is associated with a tuple $\{e_i, d_i\}$, where $e_i$ is the execution time and $d_i$ is the relative deadline, as well as the period. For a given duration, hereafter referred to as sampling window $k$ ($SW_k$), if core $\rho_i$’s frequency is fixed at $f_i(k)$, its utilization can be calculated as

$$u_i(k) = \sum_{\tau_j \in \Gamma(k)} \frac{e_j}{d_j} = \sum_{\tau_j \in \Gamma(k)} \frac{e_j^* + e'_j/f(k)}{d_j}, \quad (2.1)$$

where $e'_j$ reflects the portion of job execution that is dependent on the core’s frequency and $e_j^*$ is the independent portion. $\Gamma(k)$ is the set of jobs executed in $SW_k$.

2.3.3 System-level lifetime reliability

We consider four main integrated circuit failure mechanisms in this work: EM, TDDB, SM, and thermal cycling [13]. EM is the dislocation of metal atoms and TDDB is the deterioration of the gate oxide layer. SM is caused by directionally biased motion of atoms in metal wires. Wear due to EM, SM, and TDDB is strongly dependent on temperature. thermal cycling refers to integrated circuit fatigue failures caused by thermal mismatch deformation. We focus on short-term thermal cycling in this work, as it dramatically decreases reliability [13] and accelerates system failure [73]. In this work, we propose a framework to improve system-level reliability by reducing the effects of both temperature and thermal cycling. System-level reliability is calculated by a system-level Monte Carlo reliability modeling tool [108].

Wear due to EM, SM, and TDDB is exponentially dependent on temperature. However, the wear due to thermal cycling depends on the amplitude (e.g., the difference between the peak and valley temperature), period, and maximum temperature.
of thermal cycles. Fig. 2.1 summarize some system MTTF data obtained from the system-level reliability tool with default settings [108]. Figs. 2.1(a), (b), and (c) depict the MTTF of an example system as a function of the amplitude, period, and peak temperature of thermal cycles, respectively. As a comparison, Fig. 2.1(d) shows the system MTTF due to temperature alone without thermal cycles. As can be seen from Fig. 2.1, system MTTF is generally increases at lower temperatures and with smaller thermal cycles, but the precise relationship is complicated.

We used a Monte Carlo simulation based modeling tool [108] to calculate system-level reliability [13, 59]. In Monte Carlo simulation, numerous trials allow high accuracy but increase computation overhead (i.e., the execution time to complete MTTF calculation). Hence, analyzing the accuracy-overhead tradeoff is necessary. We use the tool [108] to calculate the system MTTF with default settings and show how the number of Monte Carlo trials affects overhead and accuracy (see Fig. 2.2). We set
Figure 2.2. The percentage error and overhead due to Monte Carlo trials.

the total number of cycles to failure to $10^6$, the default of this tool, as the baseline for comparison. The metric, percentage error, is used to quantify the accuracy. Increasing trial count decreases speed and increases accuracy. However, the accuracy saturates. The values in Fig. 2.2 are chip dependent, but the accuracy-overhead tradeoff is more general. Fig. 2.2 helps to find the most appropriate number of trials. For example, 100 is a good choice if 0.5% error is acceptable. Note that other techniques, e.g., importance sampling, may be used to improve the computational efficiency of reliability modeling if supported by the modeling tool in use.
2.4 Problem Formulation and Framework Overview

In this section, we first formulate the problem, and then briefly introduce how RUC solves this problem.

2.4.1 Problem formulation

A system’s lifetime reliability is determined by its operating temperature and thermal cycles. Given that lower frequencies and voltages lead to higher utilizations (see (2.1)) but lower temperatures, utilization control in a DVFS-enabled system can be used to manage temperature. Based on this observation, we propose to indirectly improve the system MTTF by directly controlling system utilization.

Our goal is to minimize the deviation of the cores’ utilization from a set point in each time interval, referred to as sampling window (SW). This concept has previously been used in controller based resource management [57] and temperature-aware control strategies [109]. The length of sampling window is chosen such that a core’s temperature can be considered constant within a sampling window. Before formulating the utilization control problem, we introduce the following notation:

- $u_i(k)$ is the utilization of core $\rho_i$ during the $k^{th}$ sampling window, $SW_k$;
- $u^s$ is the utilization set point;
- $u_{\text{max}}$ is the upper bound on the utilization to ensure schedulability;
- $f_i(k)$ is the frequency of core $\rho_i$ during $SW_k$;
- $f_{\text{min}}$ and $f_{\text{max}}$ are the lowest and highest core’s frequencies allowed, respectively; and
- $\bar{f}(k)$ is the average frequency over all cores during $SW_k$.

We aim to solve the following problem:

$$\min_{\rho_i \in M} \sum_{i} (u^s - u_i(k))^2.$$  \hspace{1cm} (2.2)
The solution to (2.2) must satisfy these constraints:

\[ u_i(k) \leq u_{\text{max}}, \text{for } \rho_i \in M, \quad (2.3) \]
\[ -f_{\text{th}} \leq f_i(k) - \bar{f}(k) \leq f_{\text{th}}, \text{for } \rho_i \in M, \text{ and } \]
\[ f_{\text{min}} \leq f_i(k) \leq f_{\text{max}}, \text{for } \rho_i \in M. \quad (2.5) \]

The first constraint ensures that no cores exceed the schedulability bound. Based on the discussion in Section 2.3.3, the second constraint is introduced to bound the differences in the core frequencies, which in turn bounds the core temperature differences. This constraint requires the differences between each core’s frequency and the average frequency of all cores, \( \bar{f}(k) \), to be smaller than a threshold \( f_{\text{th}} \). The third constraint limits core frequencies.

Solving the above problem by controlling a core’s frequency reduces operating temperature under the real-time constraints. A shorter sampling window results in a lower temperature, but frequently altering a core’s frequency causes more fluctuation in operating temperature and increases the effects of thermal cycling. We tune the length of the sampling window to balance peak temperature and thermal cycling dependent wear.

2.4.2 Framework overview

Real computing systems (and especially embedded systems) frequently operate in environments with ambient temperature and task execution time variations that cannot be predicted at design time, motivating us to develop an on-line reliability optimization framework. This framework improves system MTTF by using a utilization and thermal management method to solve the optimization problem defined in (2.2)–(2.5) and dynamically tuning the length of the sampling window.

The framework, RUC, consists of two main components: a global utilization con-
controller (GUC), which implements utilization and thermal management, and a sampling window controller (SWC) (see Fig. 2.3). The GUC reduces the peak temperature by dynamically adjusting core frequencies to adhere to the utilization set point. The SWC minimizes thermal cycling wear by dynamically adjusting the length of the sampling window. We assume that each core has a utilization monitor (UM) and a temperature sensor (TS).

At the end of the $k^{th}$ sampling window $SW_k$, the utilization and temperature of each core are measured and sent to the GUC and SWC (see the solid lines in Fig. 2.3), respectively. Based on the utilization, the GUC solves the optimization problem defined in (2.2)–(2.5). The solution sets core frequencies for the next sampling window, which are sent (see the dashed lines in Fig. 2.3) to each core before the start of $SW_{k+1}$. Unlike the GUC, the SWC stores temperature history. Specifically, the temperatures measured in the most recent $s$ sampling windows are saved. We refer to the $s$ sampling windows as one profiling window, PW. At the end of each profiling window, a temperature profile is generated. SWC then analyzes this temperature profile and determines whether to increase or reduce the sampling window length for the next profiling window.

2.5 Reliability-Aware Utilization Control Details

This section presents the detailed design of the GUC and SWC. We introduce a new utilization model describing how a core’s utilization changes with frequency. Based on this model, we discuss how the GUC solves the problem defined in (2.2)–(2.5) using a controller. Finally, we elaborate on how the heuristic used in the SWC adjusts the length of the sampling window dynamically at runtime to improve system MTTF.
2.5.1 Dynamic utilization model

In order to control utilization, we must first model it. We use a dynamic model to determine utilization as a function of core frequency. We first rewrite the non-linear utilization model in (2.1) in linear form

\[ u_j(k) = \sum_{\tau_i \in \Gamma(k)} \frac{c^*_i}{d_i} + \sum_{\tau_i \in \Gamma(k)} \frac{c'_i}{d_i} \times \hat{f}_j(k), \]  

(2.6)

where \( \hat{f}_j(k) = 1/f_j(k) \) is the clock period, which we refer to as the manipulated variable. The dependence of utilization on frequency can be modeled as

\[ u_j(k + 1) = u_j(k) + g_j(k) \times \Delta \hat{f}_j(k), \]  

(2.7)
where $g_j(k) = \sum_{\tau_i \in \Gamma(k)} \frac{t_i}{d_i}$ and $\Delta \hat{f}_j(k) = \hat{f}_j(k+1) - \hat{f}_j(k)$. Since we have $m$ cores, we use a matrix to describe the dynamic utilization model as

$$U(k + 1) = U(k) + G(k) \times \Delta \hat{F}(k). \quad (2.8)$$

$G(k)$ is a diagonal matrix and $G(k)_{i,i} = g_i(k)$. $\Delta \hat{F}(k) = [\Delta \hat{f}_1(k), \ldots, \Delta \hat{f}_m(k)]^T$ and $U(k) = [u_1(k), \ldots, u_m(k)]^T$.

Although (2.8) indicates the behavior of utilization as a function of the manipulated variable, it cannot be used directly because $G(k)$ is only known at runtime. According to feedback control theory, if stable control can be achieved, the value of $G(k)$ does not affect the final result [57]. Therefore, assuming the system is stable, we can set $G(k)$ to constant $G$ and rewrite (2.8) as

$$U(k + 1) = U(k) + G \times \Delta \hat{F}(k). \quad (2.9)$$

### 2.5.2 Utilization and thermal management

We solve the constrained optimization problem defined in (2.2)–(2.5) using a model predictive controller (MPC). The basic idea behind any MPC is to optimize a cost function. Hence, we first find similarities between the constrained optimization problem defined in (2.2)–(2.5) and an MPC cost function optimization problem. After this, we transform the MPC cost function optimization problem to a standard quadratic programming problem and solve it using existing solvers.

Inside the $k^{th}$ sampling window, MPC minimizes the cost function,

$$J(k) = \sum_{i=1}^{N_1} \delta_i [U(k + i) - \xi(k + i)]^2 + \sum_{j=1}^{N_2} \lambda_j \left[ \frac{1}{\Delta f(k + j - 1)} \right]^2, \quad (2.10)$$

where $N_1$ is the prediction horizon and $N_2$ is the control horizon. $\delta_i$ is the tracking error weight and $\lambda_j$ is the control penalty error [31]. The user-specified reference
trajectory $\xi(k + i)$ defines the ideal trajectory along which the utilization should converge to the set point.

Although this standard cost function in (2.10) is not our proposed optimization function in (2.2), the solution of the cost function is also the solution of (2.2). The first term in the cost function (2.10) is a variation of the function in (2.2). The second term in (2.10) minimizes the changes in the manipulated variable and does not affect the final result of the optimization problem. It only helps to make the control more stable. Hence, minimizing the cost function (2.10) under the constraints in (2.3)–(2.5) would also lead to an optimal solution to the problem defined in (2.2)–(2.5).

We propose using a quadratic programming solver to solve the optimization problem. A standard quadratic programming problem can be written as

$$
\min_{\varepsilon} \{ \frac{1}{2} \varepsilon^T \Omega \varepsilon + \zeta^T \varepsilon \}, \text{ s.t. } \begin{cases}
A_0 \times \varepsilon \leq b_0 \\
A_1 \times \varepsilon = b_1 \\
b_l \leq \varepsilon \leq b_u
\end{cases},
$$

where $\Omega$, $A_0$, and $A_1$ are matrices. $\zeta$, $b_0$, $b_1$, $b_l$, and $b_u$ are vectors and $\varepsilon$ denotes the change in a core’s frequency.

Since this standard quadratic programming problem can be directly solved by existing tools, the key point in MPC design is to transform the cost function defined in (2.10) and the constraints defined in (2.3)–(2.5) to (2.11). Based on $\varepsilon$ and the core’s frequency inside $SW_k$, the core’s frequency at $SW_{k+1}$ can be directly calculated. Because the solution to (2.10) is also the solution to (2.2), the optimal solution to the quadratic programming problem is also the optimal solution to the proposed optimization problem defined in (2.2)–(2.5).

We first transform the MPC cost function in (2.10) to the target function in standard quadratic problem. Suppose we have $h = \max\{N_1, N_2\}$, then the cost
function can be rewritten as

$$J(k) = [Y'(k) - \Phi(k)]^T \Pi [Y'(k) - \Phi(k)] + \varepsilon^T \Theta \varepsilon,$$

(2.12)

where $Y'(k) = [U(k+1), \ldots, U(k+h)]^T$ and $\Phi = [\xi(k+1), \ldots, \xi(k+h)]^T$. $\Pi$ and $\Theta$ are two diagonal matrices where $\Pi_{i,i} = \delta_i$ and $\Theta_{i,i} = \lambda_i$. Based on (2.9),

$$Y'(k) = Y(k) + \Lambda \times \varepsilon,$$

(2.13)

where $Y(k) = [U(k), \ldots, U(k)]^T$. $\Lambda$ is a low triangular matrix and $\Lambda_{i,j} = G$ if $i \geq j$ and 0 otherwise. Therefore, (2.10) can be simplified as

$$J(k) = \varepsilon^T (\Lambda^T \Pi \Lambda + \Theta) \varepsilon + 2 (Y(k)^T \Pi \Lambda - \Phi(k)^T \Pi \Lambda) \varepsilon + H.$$

(2.14)

$H$ is independent of the manipulated variable $\varepsilon$. The cost function is successfully transformed to the objective function in a standard quadratic programming problem in (2.11), where $\Omega = 2(\Lambda^T \Pi \Lambda + \Theta)$ and $\zeta = 2[Y(k)^T \Pi \Lambda - \Phi(k)^T \Pi \Lambda]^T$.

The next step is to transform the constraints to the standard form. The first constraint in (2.3) can be described as

$$\Lambda \times \varepsilon \leq U^* - Y(k),$$

(2.15)

where $U^* = [U^\text{max}, \ldots, U^\text{max}]^T$. This is a standard form where $\Lambda$ and $U^* - Y(k)$ correspond to $A_0$ and $b_0$ in (2.11), respectively.

The second constraint in (2.4) enforces the even distribution of core frequencies. The constraint on core frequencies constrains core clock periods,

$$-\bar{f}_\text{th} \leq \hat{f}_i(k) - \bar{f}(k) \leq \bar{f}_\text{th},$$

(2.16)
where the $\hat{f}_{th}$ is the threshold related to $f_{th}$, and $\bar{f}(k)$ is related to average frequency of all cores at the $k^{th}$ sampling window $\hat{f}(k)$. The clock period of core $\rho_i$ at the $(k + h)^{th}$ sampling window, $\hat{f}_i(k + h)$, is

$$\Psi_0 = \Psi_2 + \Psi_1 \times \varepsilon,$$  \hspace{1cm} (2.17)

where $\Psi_1$ is a block matrix where the element $\Psi_1(i, j)$ is a unit matrix if $j \leq i$, and 0 otherwise. $\Psi_0 = [\hat{f}_1(k + 1), \ldots, \hat{f}_m(k + 1), \ldots, \hat{f}_1(k + h), \ldots, \hat{f}_m(k + h)]^T$ and $\Psi_2 = [\hat{f}_1(k), \ldots, \hat{f}_m(k), \ldots, \hat{f}_1(k), \ldots, \hat{f}_m(k)]^T$. Based on (2.16) and (2.17), the second constraint can be expressed as

$$(\Psi_3 \Psi_1 - \Psi_3 \Psi_5 \Psi_1) \varepsilon \leq \Psi_4 - \Psi_3 \Psi_2 + \Psi_3 \Psi_5 \Psi_2,$$  \hspace{1cm} (2.18)

where $\Psi_4 = [\hat{f}_{th}, \ldots, \hat{f}_{th}, -\hat{f}_{th}, \ldots, -\hat{f}_{th}]^T$ and $\Psi_3 = [E, -E]^T$. $\Psi_5$ is diagonal matrix and $\Psi_{5_{i,i}} = \Psi_6$, where $\Psi_6$ is a matrix where all elements are equal to $1/m$.

The last constraint in (2.5) sets the upper and lower bounds on core frequencies. Based on (2.17), it can be expressed as

$$\Psi_3 \Psi_1 \varepsilon \leq \Psi_7 - \Psi_3 \Psi_2,$$  \hspace{1cm} (2.19)

where $\Psi_7 = [\hat{f}_{max}, \ldots, -\hat{f}_{max}, -\hat{f}_{min}, \ldots, -\hat{f}_{min}]^T$, $\hat{f}_{max} = \frac{1}{f_{max}}$, and $\hat{f}_{min} = \frac{1}{f_{min}}$.

After these transformations, the problem in (2.2)–(2.5) is a standard quadratic programming problem. This problem can be directly solved using standard solvers such as COPL_QP [14] or Python’s package CVXOPT [19].

GUC aims to reduce the peak and average temperature for the entire chip and to balance the temperatures across cores. Although a lower temperature tends to increase the system MTTF, the latter also depends on the amplitudes, periods, peak and valley temperature of thermal cycles. Hence, we propose a sampling window
controller to reduce the aging effect of thermal cycling.

2.5.3 Sampling window controller design

We design SWC to minimize the aging effect of thermal cycling by dynamically changing the length of the sampling window, $L_{SW}$. Since a core’s frequency can change from one sampling window to the next, the length of sampling window directly impacts thermal cycling. In general, a shorter sampling window means scaling core’s frequency more frequently, and helps to reduce the peak temperature and amplitude of thermal cycles, but may increase the frequency of thermal cycles.

It is difficult to precisely model how the sampling window affects the system reliability, and the complicated reliability model also increases the overhead of searching for the best sampling period. Hence, to balance the impact of peak temperature against that of thermal cycles on the system MTTF, we design an efficient online heuristic based on binary search to adjust $L_{SW}$ at runtime.

Our heuristic algorithm in the SWC uses the concept of a profiling window to adjust the $L_{SW}$. A profiling window is composed of $s$ equal-length sampling windows and referred to as PW. In each profiling window, the $s$ temperature points make up a temperature profile. At the end of the profiling window, we calculate the system MTTF from the temporal temperature profile using a reliability modelling tool [108]. Since this tool assumes the input temperature profile is repeated until the chip fails, the established temperature profile must have enough temperature points. Although a larger $s$ makes the calculation of system MTTF more precise, it increases the overhead of the reliability modeling tool. In our experimental evaluations, we find that setting $s = 50$ achieves an accurate system MTTF with an acceptable overhead. We store the history of the system MTTF for the various length of the sampling window and rely on binary search to find the most appropriate $L_{SW}$ for the sampling windows in the next profiling window.
To perform binary search, we need to determine the lower and upper bound of $L_{SW}$. We set the lower bound of $L_{SW}$ ($L_{SW}^{lb}$) to 1 second. Since the quadratic programming solver executes at the end of each sampling window, and the execution time of a typical solver, e.g., COPL$_{QP}$ [14], is less than 10 ms on our platform (with ARMv7 1.23 GHz), setting $L_{SW}^{lb}$ to 1s keeps the overhead due to the GUC under 1% of the sampling window length. To determine the upper bound of the $L_{SW}^{ub}$, we consider two factors. Since the GUC expects a constant temperature in a given sampling window, the sampling window length must be smaller than some constant, say $C_{HW}$. $C_{HW}$ is dependent on the hardware platform. A low power density chip leads to a large $C_{HW}$ and vice versa. As for the other factor, since thermal cycles can essentially be avoided if the $L_{SW}$ equals the hyperperiod of the periodic task set, any length larger than the hyperperiod increases temperature and thermal cycling. Hence, we only need to consider lengths of the sampling window less than or equal to the hyperperiod. As a result, we set $L_{SW}^{ub} = \min\{|hp|, C_{HW}\}$, where $|hp|$ is the length of hyperperiod.

The pseudo-code for the SWC is given in Alg. 1. Lines 2–3 initialize the needed variables, where $L_{SW}^{c}$ is the length of the current sampling window. The infinite while-loop (Lines 4–26) allows the algorithm to run throughout system lifetime. At the end of each sampling window, the temperature of each core is read from temperature sensors (Lines 6–8). At the end of the $j^{th}$ profiling window ($i = s$), a temperature profile, $TP(j)$, is constructed based on the collected temperature of the $m$ cores in the current $s$ sampling windows (Line 11). The corresponding MTTF, $MTTF_j$, is obtained using a reliability modeling tool [108] (Line 12). The $L_{SW}^{c}$ is next determined for the sampling windows in the upcoming profiling window (Lines 13–22). $L_{SW}^{c}$ is set to $(L_{SW}^{lb} + L_{SW}^{ub})/2$ at the second sampling window (Lines 13–14), and is then dynamically changed based on the comparison of MTTFs (Lines 16–21). $L_{SW}^{lb}$ and $L_{SW}^{ub}$ are updated and $L_{SW}^{c}$ converges to the most appropriate value. Finally, $i$ and $j$
Algorithm 1 Sampling Window Control

1: procedure SWC($L_{sw}^{lb}$, $L_{sw}^{ub}$)
2:    $i \leftarrow 1$, $j \leftarrow 1$
3:    $L_{sw}^{c} \leftarrow L_{sw}^{lb}$
4:    while True do
5:        if at the end of $SW_i$ then
6:            for each core $\rho_k$ do
7:                measure temperature $T_k(i)$
8:            end for
9:            $i \leftarrow i + 1$
10:       if $i = s$ then
11:           $tp(j) \leftarrow \{T_i(1), \ldots, T_m(s)\}$
12:           calculate $MTTF_j$
13:           if $j = 1$ then
14:              $L_{sw}^{c} \leftarrow (L_{sw}^{lb} + L_{sw}^{ub})/2$
15:           else
16:              if $MTTF_j \geq MTTF_{j-1}$ then
17:                  $L_{sw}^{lb} \leftarrow L_{sw}^{c}$
18:              else
19:                  $L_{sw}^{ub} \leftarrow L_{sw}^{c}$
20:              end if
21:           end if
22:       end if
23:       $i \leftarrow 1$, $j \leftarrow j + 1$
24:    end if
25: end procedure

are updated to indicate the beginning of the next profiling window (Line 23).

2.6 Experimental Setup

To evaluate the proposed framework, we conducted experiments to compare the proposed framework with two existing approaches. In this section, we present the platforms, tasks, and the existing frameworks used for comparison in our experiments.
TABLE 2.1

REAL-TIME TASKS FROM MIBENCH BENCHMARK SUITE

<table>
<thead>
<tr>
<th>Task</th>
<th>Application</th>
<th>Mapping to</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitcount</td>
<td>Automotive</td>
<td>Core 2</td>
<td>960 ms–969 ms</td>
</tr>
<tr>
<td>susan</td>
<td>Automotive</td>
<td>Core 1</td>
<td>129 ms–151 ms</td>
</tr>
<tr>
<td>qsort</td>
<td>Automotive</td>
<td>Core 1</td>
<td>176 ms–195 ms</td>
</tr>
<tr>
<td>dijkstra</td>
<td>Network</td>
<td>Core 3</td>
<td>130 ms–141 ms</td>
</tr>
<tr>
<td>patricia</td>
<td>Network</td>
<td>Core 3</td>
<td>445 ms–459 ms</td>
</tr>
<tr>
<td>stringsearch</td>
<td>Office</td>
<td>Core 2</td>
<td>3 ms–10 ms</td>
</tr>
<tr>
<td>blowfish</td>
<td>Security</td>
<td>Core 1</td>
<td>1352 ms–1369 ms</td>
</tr>
<tr>
<td>sha</td>
<td>Security</td>
<td>Core 2</td>
<td>54 ms–60 ms</td>
</tr>
<tr>
<td>crc32</td>
<td>Telecommunication</td>
<td>Core 3</td>
<td>843 ms–874 ms</td>
</tr>
<tr>
<td>cjpeg</td>
<td>Consumer</td>
<td>Core 3</td>
<td>16 ms–20 ms</td>
</tr>
</tbody>
</table>

2.6.1 Tasks

A total of 10 tasks were chosen from Mibench [25] which include different tasks from automotive, network, office, security, telecommunication, and consumer applications. We measured the execution times of the tasks on a quad-core ARM Cortex-A15 chip, Nvidia’s TK1 [76], when a core’s frequency is 1.24 GHz and then assigned them to appropriate cores (see Table 2.1). Note that Core 0 is reserved for the operating system. Based on the execution time data, we designed two groups of task setups. In the first group, tasks are frame-based and share the same period and deadline. We will measure the system-level MTTFs when the deadline is 1.6, 1.8, 2.0, 2.2, and 2.4 seconds. In the second group, a task’s deadline and period are random in the ranges 1.2–1.6, 1.6–2.0, 2.0–2.4, and 2.4–2.8 seconds.
2.6.2 Comparison targets

We compared our proposed RUC with two existing frameworks: pure utilization control (UC) [57] and temperature-aware control (TA) [41, 31]. In contrast with RUC, the length of sampling window in UC is fixed. Generally, it is set to 1 s [57]. In addition, since UC is not designed for lifetime reliability, it does not bound the differences in core frequencies. TA has a similar design to UC, but the goal is to control the temperature to a specific set-point, which is lower than the maximum temperature allowed by the chip. Hence, TA is an effective way to guarantee that the operating temperature is safe.

Two metrics are considered in the comparison. Reliability is quantified by the system MTTF, which is obtained using the system-level reliability modeling tool [108]. Real-time performance is quantified as the percentage of feasible solutions (FS), which is the ratio of the number of tasks satisfying their real-time requirements to the total number of tasks.

2.6.3 Experimental platforms

We conducted experiments on two platforms: an ARMv7 chip, and a simulator. The first experimental platform is Nvidia’s TK1, a quad-core Cortex-A15 ARM chip [76]. This SoC is designed for mobile and automotive applications. The quad-core ARM Cortex-A15 CPU and 192 Kepler GPU cores provide high performance with low power requirement. As a low-power chip, TK1 supports 12 different frequencies from 1.24 GHz to 2.32 GHz, but all cores are required to have the same frequency. Except for the primary core (core 0), all cores can be powered on and off dynamically. There are 11 thermal sensors to sample the CPU, GPU, and other component temperatures every 50 ms. However, the default interface only provides one CPU temperature for all CPU cores. TK1 is shipped with an operating system based on Ubuntu, so most of desktop-level tasks can be directly executed on it.
In order to evaluate RUC on different platforms, we constructed a hardware platform simulator. We extracted the parameters from TK1 to build power model. In the simulator, the temperature is obtained using Hotspot, a thermal modeling tool [98]. Our simulator is tested via a comparison to the TK1.

We first obtained the parameter values for Hotspot. Since TK1 only reports one temperature for all cores, we made an assumption that all cores have the same thermal capacitance, $C$, and thermal resistance $R$. We used no fan when measuring the temperature. The temperature profile for four fully loaded core running at the highest frequency is shown in Fig. 2.4. Note that due to power variation and other uncontrollable parameters, e.g., small changes in ambient temperature, the obtained temperature profile fluctuated slightly. In this configuration, the chip’s power is 6.98 W [77], so $R$ can be estimated as $10.24 \, ^\circ\text{C/W}$. For the same reasons, we estimated $C$ to be in the range of $2.34–7.34 \, \text{J/}^\circ\text{C}$. The capacitance used in Hotspot is randomly generated in this range.

Figure 2.4. Temperature profile with four fully loaded cores running at the highest frequency.
Based on $R$ and $C$, we determined the power model for the simulator. Let $P_{act}(f)$ be the core’s active power at frequency $f$. $P_{oth}(f)$ is the power independent of core utilization but related to core frequency. $P_{oth}(f)$ includes the core’s idle power and other components’ power, e.g., GPU and memory. The average power ($\bar{P}$) can be calculated as

$$\bar{P} = U \times P_{act}(f) + P_{oth}(f).$$

The value of $P_{act}(f)$ and $P_{oth}(f)$ are extracted from the measurement of TK1. It is difficult to measure TK1 power consumption directly. Instead, we used steady state temperatures to determine values of $P_{oth}(f)$ and $P_{act}(f)$. Fig. 2.5(a) shows the steady state temperature with different numbers of fully loaded cores running at different frequencies. Fig. 2.5(b) depicts the steady state temperatures for one core running at different frequencies and utilizations. Note that since the resolution of the thermal sensors is 0.5°C, manipulating a core’s frequency may not change the measured steady-state temperature, especially when the workload is light. Based on these measurements, the power values are derived and summarized in Table 2.2.
Figure 2.5. Steady-state temperature with (a) different numbers of fully loaded cores running at different frequencies and (b) one core running at different frequencies and utilizations.

In order to validate this power model, we compared the temperature readings from the thermal sensors with those estimated by Hotspot (see Fig. 2.6). As can be seen from Fig. 2.6, the parameter values that we have obtained resulted in accurate temperature estimation. The average temperature difference between TK1 and the simulator is less than 2°C, and Fig. 2.7 shows that this small difference does not impact the system-level MTTF. The data in Fig. 2.7 indicate that RUC has a similar performance on both TK1 and the simulator. The maximum difference is about 2%, and the average is less than 1%. These experiments demonstrated that the parameter values that we have obtained indeed lead to reliable accuracy of the simulator.

2.7 Experimental Evaluation

In this section, we examine the performance of the proposed RUC compared to the utilization control (UC) and temperature-aware control (TA) approaches.
Figure 2.6. Temperature readings from thermal sensors and estimated by Hotspot for various core frequencies.

Figure 2.7. Normalized MTTF with TK1 and constructed simulator.
2.7.1 Experiments on TK1 chip

We compared the proposed RUC with UC and TA to determine whether RUC can improve lifetime reliability without sacrificing real-time performance. For RUC and UC, the utilization set-point is close to the schedulability boundary, which was set to 70%. We used a temperature set point of 70°C in TA, which is lower than the peak operating temperature.

We first studied the overhead of the solver to the optimization problem defined in (2.2)–(2.5). If four cores are required to have the same frequency, such as TK1 [76], a brute-force search solver consumes less than 1 ms to find the optimal solution. This overhead is tolerable in our RUC. However, if cores can run at different frequencies, the overhead of brute-force search increases to 74 ms. In this case, brute-force search is impractical. The overhead can be reduced if we use standard solvers, e.g., COPL_QP [14] or CVXOPT [19]. The overhead of COPL_QP is about 4 ms and CVXOPT is about 25 ms. Both their overheads are small, and we can choose one of them offline.

Fig. 2.8 shows the experimental results when tasks are frame-based. RUC achieves a higher MTTF than UC and TA in all the cases. Thanks to the proposed sam-

Figure 2.8. Normalized MTTF and percentage of feasible solutions when tasks are frame-based and running on TK1.
pling window control, when compared to UC, RUC improves the MTTF by 9.4% on average, and up to 32%. TA keeps the run-time temperature at 70°C, and the corresponding MTTF is much smaller than RUC and UC when the workload is light. RUC doubles the MTTF when compared to TA. In terms of real-time performance, all the tested frameworks have a large and similar percentage of feasible solutions (FS). Since RUC may increase the length of the sampling window and reduce the DVFS frequency, it may lead to more tasks missing their deadlines. However, its percentage of FS is only 2% smaller than TA, and also close to 100%. In most soft real-time systems, this is tolerable [57].

In Fig. 2.9, each task has the same deadline and period, which is randomly generated in different ranges for different tasks. The average MTTF improvement of RUC over UC is 6.4%, and up to 9.5%. Compared to TA, this improvement increases to over 100%. For real-time performance, RUC does not have the highest percentage of FS, but the small loss in number of feasible solutions does not limit its application in many soft real-time systems.

Figure 2.9. Normalized MTTF and percentage of feasible solutions when tasks’ periods are random and running on TK1.
2.7.2 Simulation results

Leveraging the simulator, we conducted two sets of experiments to further assess the pros and cons of RUC in a more general setting. Tasks have been assigned to appropriate cores, and each task’s execution time is the average of measured execution time on TK1 (see Table 2.1). We extended the simulator so that different cores can operate at different frequencies on this simulator. In the first set of experiments, we assumed the chip has the same low power density as TK1. We increased the power density in the second set of experiments. We used the CVXOPT [19] Python package, to solve the quadratic programming problem.

Fig. 2.10 and 2.11 depict the results of the first set of experiments. Fig. 2.10 shows the MTTF and percentage of FS when tasks are frame-based and Fig. 2.11 shows the results when tasks’ period is randomly generated. Compared to UC, the average improvement of RUC is 5.9% when tasks are frame-based and 6.5% when task periods are randomly generated. As in the experiment on TK1, RUC also doubles the MTTF when compared to TA. All three frameworks allow almost the same number of deadlines to be met. Note that the absolute value of the MTTF and percentages of FS could be different in the experiments on TK1 and on the simulator. This is
because the estimated temperature and tasks’ execution times may differ from those on the TK1. Both of these experiments show that RUC improves the system MTTF and also guarantees that almost all task deadlines are met.

All the above experiments demonstrate that RUC has a better system reliability performance. However, for a low-power chip, different core frequencies do not lead to significant power changes, which weakens the effect of DVFS and also the sampling window control. For this kind of chip, the MTTF improvement of RUC over UC is less than 32%.

In order to study the effectiveness of RUC on a high power density chip, we used the same simulator as in the previous experiments but increased the chip’s power density by 3× the power density. We increasing the power density by 3× because it guarantees that the maximum operating temperature still remains below the temperature bound. We assumed that the Connectland 1504002 heatsink [15] was used. With this heatsink, the thermal resistance is estimated to be 7.02 J/°C, and the thermal capacitance is in the range of 1.92–25.87 watt/°C. For RUC and UC, we used the same CVXOPT [19] to solve the quadratic programming problem.

The experimental results are shown in Fig. 2.12. In all cases, the average im-
Figure 2.12. Normalized MTTF and percentage of feasible solutions when running on a simulator with high power density chips with (a) frame-based tasks; (b) tasks with random periods.

Improvement in MTTF by RUC is 39.5% for frame-based tasks and 54.9% when tasks’ periods are random. At higher power density, reducing core frequency dramatically reduces run-time temperature. At the same time, frequent changes in DVFS settings introduce more temperature fluctuations, which increases aging from thermal cycling. Hence, sampling window control in RUC benefits lifetime reliability. RUC can achieve a more stable temperature profile than UC, which is one of the most important factors that positively impacts the system-level MTTF. The behaviors of RUC and UC have similar real-time performance to that in the previous experiments. The percentage of FS is close to 100%. RUC allows most tasks complete before their deadlines even when the workload is heavy.

2.8 Summary

This work proposes a utilization and thermal management method to reduce peak temperature by adjusting core frequencies and voltages within each sampling window. Based on this method, a framework is developed to maximize the lifetime reliability of MPSoCs under soft real-time constraints. This online framework lowers peak
temperature, balances the temperature differences among cores, and reduces thermal cycling. We have conducted experiments with different types of tasks on multiple platforms, including a quad-core ARM chip and a simulator, and considering chips with different power densities. The results reveal that, on all the platforms and with a variety of task setups, our approach is effective in increasing lifetime reliability of soft real-time systems compared to existing temperature-aware and utilization control approaches.
CHAPTER 3

IMPROVING SOFT-ERROR RELIABILITY THROUGH DYNAMIC RECOVERY

This work introduces a framework for improving soft-error reliability while satisfying lifetime reliability and real-time constraints. It describes a new system-level soft-error reliability model that enables our primary contribution, a dynamic recovery allocation method that allows any failed task to be recovered if the remaining slack is adequate. Based on this method, we propose two scheduling algorithms for task sets to improve system-level soft-error reliability. Lifetime reliability requirements are satisfied by reducing core frequencies for appropriate tasks, thereby reducing wear due to high temperature and thermal cycling.

3.1 Introduction

Since many real-time embedded systems are used in critical applications and may be expensive as well as difficult to replace, soft-error reliability due to transient faults as well as lifetime reliability due to permanent faults are important design considerations. However, improving lifetime reliability and soft-error reliability is a trade-off problem. Techniques for reducing power, energy, and/or temperature to improve lifetime reliability, however, reduce soft-error reliability, and vice versa.

Motivated by applications such as resource reservation servers [103], we focus on improving system-level soft-error reliability while satisfying some pre-defined lifetime reliability and deadline requirements since system-level soft-error reliability is usually
smaller than lifetime reliability. Although increasing core frequency may allow more tasks to complete successfully, errors can still occur.

A complementary method, which handles errors, is to allocate recoveries, i.e., re-execute failed tasks [45, 111, 113, 34, 106, 115, 35, 119]. Since the probability that a task fails twice is generally much lower than the probability that the task fails once, this method is quite efficient in increasing soft-error reliability [116]. Izosimov et al. [45] determined how to allocate recoveries to tasks offline. A failed task can be recovered only when a recovery has been allocated. A more flexible technique is one in which all tasks share slack and the recovery allocation is determined online. For example, Zhou et al. [111, 113, 34, 115] proposed a shared recovery technique where the first $k$ failed tasks can use slack, but the value of $k$ is determined offline. Since only the first $k$ tasks can be recovered, the effectiveness of this technique is very much dependent on the choice of $k$ as well as when and where soft errors occur. Although a large $k$ leads to a higher reliability, in this approach, $k$ must be small enough to satisfy the real-time requirement. While all the above work focuses on increasing soft-error reliability, none of them consider lifetime reliability.

In this work, we first develop a method to allocate recoveries to failed tasks dynamically. Based on this method, we introduce a novel framework for improving the system-level soft-error reliability while satisfying real-time and lifetime reliability constraints. We make three main contributions in this chapter.

- We propose a new dynamic recovery allocation method and derive the corresponding system-level soft-error reliability. In our method, slack is shared and any failed task can be recovered if there is sufficient amount of remaining slack for executing the task. Compared to existing approaches [111, 113], our method does not limit the number of recoveries and thus provides the highest level of slack sharing flexibility.

- We introduce two scheduling algorithms for different types of task sets to improve soft-error reliability. A fast and simple algorithm is designed for task sets with special properties: execution times of different tasks are close and the slack is small. We also propose a computationally efficient method to obtain
system-level soft-error reliability for such task sets. A more powerful algorithm is designed for general task sets.

We have implemented the above contributions in a reliability improvement framework, called RIF. RIF consists of two components. The first component aims at increasing system-level soft-error reliability through task scheduling and dynamically allocating recoveries. The second component handles lifetime reliability and real-time constraints by reducing core frequencies for tasks. We evaluated RIF in a simulator, which is constructed based on the Nvidia’s TK1 chip [76]. For both randomly generated tasks and tasks from the MiBench benchmark suite [25], RIF reduces the probability of failure by at least 8% and 73% on average when compared to existing approaches.

3.2 Preliminaries

In this section, we introduce the task model, the soft error model and the lifetime reliability model.

3.2.1 Task model

We consider a frame-based task set composed of $n$ independent tasks, denoted by $\{\tau_1, \ldots, \tau_n\}$. Such tasks are often found in resource reservation servers [103]. Tasks are executed on a dynamic voltage/frequency scaling capable uniprocessor that supports a discrete set of frequencies varying from the minimum, $f_{\text{min}}$, to the maximum, $f_{\text{max}}$. All tasks should complete their executions by a common deadline, $d$. Slack, $s$, is defined as $d - \sum_{i=1}^{n} e_i$ and used to re-execute failed tasks. The $i^{th}$ task, $\tau_i$, is associated with a tuple $\{e_i, f_i, p_i\}$, where $e_i$ is the worst-case execution time measured at $f_{\text{max}}$, $f_i$ is the core frequency when executing $\tau_i$, and $p_i$ is the priority, where a large value for $p_i$ denotes a higher priority. Since tasks have the same period, we do not need to distinguish instances (i.e., jobs) of a task.
3.2.2 Soft-error reliability

The soft-error rate represents the expected number of failures occurring per second and is generally modeled as an exponential distribution with an average arrival rate \([116]\). The average rate highly depends on the core frequency and can be modeled \([116]\) as

\[
\lambda(f) = \lambda_0 \times 10^{\frac{c \times (f_{\text{max}} - f)}{f_{\text{max}} - f_{\text{min}}}},
\]

where \(f_{\text{min}}\) and \(f_{\text{max}}\) are the minimum and maximum core frequencies, respectively. \(\lambda_0\) is the average arrival rate when the processor is executing at \(f_{\text{max}}\) and \(f\) is the running core frequency. \(c\) is a hardware specific constant that indicates the sensitivity of fault rates to frequency scaling.

Let \(r^t_i(f_i)\) denote the probability that no soft error occurs in the execution of task \(\tau_i\). \(r^t_i(f_i)\) depends on the execution time of \(\tau_i\) and the processor frequency when executing \(\tau_i\), i.e., \(f_i\). \(r^t_i(f_i)\) can be calculated as \([111]\)

\[
r^t_i(f_i) = e^{-\lambda(f_i) \frac{t_i}{f_i}}.
\]

It is clear from (3.1) and (3.2) that reducing the core frequency for a task leads to exponentially increasing soft-error rate.

With recovery, even if a task fails in its first iteration due to soft error, it is considered to be successful if it has a recovery, i.e., task re-execution, and the recovery completes successfully. Let \(r^s_i\) denote the probability that \(\tau_i\) successfully completes in the recovery, i.e., the second execution. The value of \(r^s_i\) depends on the recovery methods used, i.e., how to allocate and execute recovery. Section 3.3.2 explains how to calculate \(r^s_i\) for our proposed dynamic recovery method.
3.2.3 Lifetime reliability

Lifetime reliability depends on multiple wear-out effects and is typically measured by the mean time to failure model [13]. Wear due to EM, SM, and TDDB is exponentially dependent on operating temperature. Wear due to thermal cycling depends on the amplitude (e.g., the difference between the peak and valley temperature), period, and cycle maximum temperature [13]. To improve lifetime reliability, both the operating temperature and the effect of thermal cycling should be reduced.

Lifetime reliability is commonly obtained using the sum of failure rate model calculated at the device level [43]. However, this device-level reliability can deviate significantly from the system-level reliability since the temporal fault distribution of individual devices and that of the corresponding million-device system can differ greatly [108]. We use a Monte Carlo simulation based modeling tool [108] to calculate system-level lifetime reliability [13, 59]. This tool considers wears influenced by operating temperature and thermal cycling, and integrates device-, component-, and system-level models. Our framework tunes temperature by scaling core frequency when executing different tasks, and this tool uses the corresponding thermal profile to calculate system-level lifetime reliability.

3.2.4 Motivating examples

We use an example task set to illustrate how recovery allocations and task schedules impact the system-level soft-error reliability. The considered task set is composed of four tasks sharing a common period of 15 s. The worst-case execution times of the tasks are 1 s, 2 s, 3 s, and 4 s, respectively, and the slack is 5 s. We assume all tasks are executed at the highest core frequency. The soft-error reliability of the tasks is 0.904, 0.819, 0.741, and 0.670, respectively, based on an existing model [111].

The system-level soft-error reliability is the probability that all tasks complete successfully. If no recovery is allowed as shown in Fig. 3.1(a), one can readily see that
the system-level reliability is 0.368. A higher system-level reliability can be achieved if some tasks can recover. One existing method is to allocate recoveries to tasks offline [116]. Fig. 3.1(b) represents the optimal solution for maximizing the soft-error reliability, in which task $\tau_2$ and $\tau_3$ have recoveries $r_2$ and $r_3$. In this case, the system-level reliability is 0.547. Another existing method allocates recovery online [111, 113]. Fig. 3.1(c) shows a scenario where the first failed task has a recovery [111]. The system-level reliability is 0.686, which is higher than that in Fig. 3.1(b). However, although the slack is dynamically used in Fig. 3.1(c), only one task can be recovered.

In our proposed dynamic recovery allocation method, a failed task is recovered if the remaining slack is adequate, and tasks consume slack on a first-come, first-served basis (see Fig. 3.1(d)). For example, task $\tau_2$ can recover even if $\tau_1$ fails. However, task $\tau_3$ cannot recover if both $\tau_1$ and $\tau_2$ fail because the remaining slack for $\tau_3$ is only $2s$. Task $\tau_4$ can recover only when all tasks succeed or only $\tau_1$ fails. Hence, the probabilities of recoveries for $\tau_3$ and $\tau_4$ are 0.983 and 0.607, and the system-level reliability is 0.716.

Now, let us consider the impact of task scheduling on the system-level soft-error reliability. Fig. 3.1(e) represents a new schedule where the task’s priority is the inverse of its execution time. In this case, the probabilities of recovering $\tau_1$, $\tau_2$, $\tau_3$, and $\tau_4$ are 0.792, 0.670, 0.670, and 1.000. Compared with Fig. 3.1(d), the task with the lowest reliability, $\tau_4$, can always recover, but the system-level reliability is 0.692. Hence, a scheduling algorithm that simply improves the probability of recovery for some specific tasks may not be a good solution. Based on these observations, we design a simple and fast scheduling algorithm for special task sets and a powerful scheduling algorithm for general task sets.
Figure 3.1. Motivating examples illustrate dynamic recovery allocation is better than static recovery allocation, and the scheduling can affect system-level soft-error reliability.
3.3 Problem Formulation and Solution Overview

In this section, we first outline the problem of interest. We then introduce the formulation for the system-level soft-error reliability, which is key to solving the problem under consideration. Finally, we present an overview of our framework.

3.3.1 Problem formulation

We propose a new approach to dynamically allocate recoveries to failed tasks. In this method, available slack is shared by all tasks and dynamically assigned on a first-come, first-serve basis. A recovery is allocated to task $\tau_i$ if the remaining slack is no smaller than $e_i$. This first-come, first-serve design is efficient in computation and memory. On the contrary, other designs, such as allocating recoveries after all tasks completed, consume memory to store the completion status of each task and have a heavier computational overhead.

In order to maximize soft-error reliability, the recoveries are required to execute at the highest core frequency. Although a recovery may still fail due to the occurrence of soft errors, the probability that both a task and its recovery fail is very low. Therefore, we only allocate one recovery for each task to prevent a task from consuming too much slack. Note that the method presented in this work can be extended to allow some critical tasks to receive more than one recovery. The details are left to future work. Since a recovery is allocated only when the remaining slack is adequate, the recovery itself will not violate the real-time requirement. At the same time, whether a higher priority task consumes slack affects recovery allocation of a lower-priority task since the higher priority task is executed first. Hence, task scheduling directly impacts soft-error reliability.

Given the importance of soft-error reliability, lifetime reliability, and real-time requirements, we aim to solve the problem of maximizing system-level soft-error reliability through dynamic recovery while satisfying lifetime reliability and real-time
constraints. System-level soft-error reliability is defined as the probability that all
tasks complete successfully. Note that a task is considered to be successful if it en-\ncounters no soft error during its first execution or successfully completes in the second
execution (where the second execution is the recovery). Specifically, let $R(S, F)$ be
the system-level soft-error reliability. The problem to be tackled is formulated as
finding a schedule, $S$, and a frequency assignment $F = \{f_1, \ldots, f_n\}$ to maximize the
system-level soft-error reliability

$$\max R(S, F).$$

(3.3)

The solution should satisfy the following constraints:

$$\sum_{\tau_i} \frac{e_i \times f_{\max}}{f_i} \leq d,$$

(3.4)

$$LTR(S, F) \geq LTR_{TH}.$$ 

(3.5)

$LTR_{TH}$ is the minimum lifetime reliability that the system must achieve. Note that
$LTR_{TH}$ reflects the concept of product warranty period. Before considering any
methods to solve the above optimization problem, we need to know how to evaluate
$R(S, F)$. Below we introduce the formulation of $R(S, F)$ for the dynamic recovery
scheme that we have introduced.

### 3.3.2 System-level soft-error reliability

In order to solve the optimization problem, we need to evaluate $R(S, F)$. Before
presenting the formulation of $R(S, F)$, let us first derive the probability that a task
successfully completes. For task $\tau_i$, $r_{i}^{1}(f_i)$ is the probability that no soft-error occurs
during its normal execution. In a given schedule, $S$, $r_{i}^{*}(S)$ denotes the probability
that $\tau_i$ has a recovery and the recovery completes successfully. Hence, the probability
that a task completes successfully is

\[ r_i(f_i, S) = 1 - (1 - r_i^t(f_i)) \times (1 - r_i^s(S)). \] (3.6)

It follows that the system-level soft-error reliability is

\[ R(F, S) = \prod_{i=1}^{n} \{1 - (1 - r_i^t(f_i))(1 - r_i^s(S))\}, \] (3.7)

where \( F = f_1, \ldots, f_n \) is the frequency assignment to tasks.

The variable \( r_i^s(S) \) is the key to compute the overall system-level soft-error reliability. We use a concept called execution pattern to calculate \( r_i^s(S) \). Suppose \( S = \{\tau_1, \ldots, \tau_n\} \) where task \( \tau_i \) has a higher priority than \( \tau_{i+1} \). We use the execution pattern, \( P_i \), to indicate the successful and failed tasks before \( \tau_i \). Inside an execution pattern, \( \tau_k^+ \) denotes the successful completion of \( \tau_k \) and \( \tau_k^- \) indicates that \( \tau_k \) has failed. Since one execution pattern is composed of \( i - 1 \) tasks, there are \( 2^{i-1} \) patterns for \( \tau_i \), and we use \( P_{i,j} \) indicates the \( j^{th} \) pattern.

The value of \( r_i^s(S) \) can be determined if we know the time used by the recovered tasks in each pattern, denoted by \( Time(P_{i,j}) \), and the probability that each pattern appears, denoted by \( Prob(P_{i,j}) \). Suppose there are \( m \) patterns satisfying

\[ Time(P_{i,j}) + e_i \leq s, \] (3.8)

where \( s \) is the shared slack. Then \( r_i^s(S) \) can be calculated by

\[ r_i^s(S) = \begin{cases} 
 r_i^t(f_{\text{max}}) \times \sum_{j=1}^{m} Prob(P_{i,j}), & \text{if } m > 0, \\
 0, & \text{otherwise},
\end{cases} \] (3.9)

where \( m = 0 \) means that \( \tau_i \) can never be recovered. If all execution patterns satisfy the condition (3.8), then \( r_i^s(S) = r_i^t(f_{\text{max}}) \) since \( \tau_i \) can always be recovered.
Time(\mathcal{P}_{i,j}) can be obtained as follows. A task \tau_i uses slack to recover only when the remaining slack is larger than \epsilon_i, so it is possible that a failed task with large execution time does not use the slack. We search each failed task in \mathcal{P}_{i,j} and calculate Time(\mathcal{P}_{i,j}) iteratively. We first initialize Time(\mathcal{P}_{i,j}) = 0. Then, for each failed task \tau_i in \mathcal{P}_{i,j}, if \epsilon_i \leq s, update Time(\mathcal{P}_{i,j}) = Time(\mathcal{P}_{i,j}) + \epsilon_i and s = s - \epsilon_i. For Prob(\mathcal{P}_{i,j}), note that it simply depends on the soft-error reliability of each task included in the pattern. That is,

\begin{equation}
Prob(\mathcal{P}_{i,j}) = \prod_{\tau_i \in \mathcal{P}_{i,j}} r^i_k(f_k) \times \prod_{\tau_k \in \mathcal{P}_{i,j}} (1 - r^i_k(f_k)).
\end{equation}

(3.10)

3.3.3 Overview of framework

Based on the formulation of \(R(S, F)\) and \(LTR(S, F)\), we propose a reliability improvement framework (RIF) to solve the problem defined in 3.3. RIF consists of two components. One focuses on improving system-level soft-error reliability by scheduling tasks, i.e., determining the task priorities, and the other focuses on satisfying lifetime reliability by scaling core frequency. Since system-level soft-error reliability is usually much smaller than lifetime reliability, RIF assigns \(f_{\text{max}}\) to each task and schedule tasks offline to improve soft-error reliability defined in (3.7). RIF employs a fast and simple scheduling algorithm for task sets with some special properties or a more powerful scheduling algorithm for general task sets. RIF then checks whether the schedule satisfies the lifetime reliability constraint in (3.5). If not, core frequencies of lower priority and higher power consumption tasks are reduced iteratively to increase lifetime reliability until the lifetime constraint is satisfied. Meanwhile, core frequencies of tasks are required to be high enough to satisfy the real-time constraint in (3.4). In the subsequent sections, we will discuss the details of the two components.
3.4 Improving Soft-Error Reliability

In this section, we present algorithms to schedule tasks online to improve the system-level soft-error reliability. Since recovery allocation depends on the remaining slack, and the task with the highest priority is scheduled first, specifying a high priority to a task increases the probability that it can be recovered but reduces the soft-error reliability of low priority tasks. In other words, the probability that a task can be recovered depends on whether previously executed tasks require recovery. Based on the observations in Fig. 3.1(d) and 3.1(e), we design two scheduling algorithms. The first algorithm targets task sets with special properties while the second algorithm can be applied to general task sets. As pointed out in Section 3.3.3, tasks are assumed to run at the highest core frequency. We omit the frequency parameter in the following discussion when not required to improve readability.

3.4.1 A fast scheduling algorithm for a specific type of task sets

We propose a fast reliability improvement scheduling algorithm (FRIS) to improve system-level soft-error reliability if the task sets satisfy certain conditions.

The algorithm is built on the observation that a schedule allowing more tasks to recover does not always lead to a higher system-level soft-error reliability. For a task with very large execution time, there may be insufficient slack for the task to re-execute. However, the reliability of this task can significantly reduce the overall system-level soft-error reliability. Hence, a schedule that has a higher system-level soft-error reliability would have tasks with execution times that are close to one another. We summarize this observation in Theorem 1 below. Before presenting the theorem, we first define a few useful terms. For a given task set \( \{\tau_i, \ldots, \tau_n\} \), we define 
\[ e_{\text{max}} = \max\{e_i, \ldots, e_n\}, \quad r^t_{\text{min}} = \min\{r^t_i, \ldots, r^t_n\}, \quad \text{and} \quad r^t_{\text{max}} = \max\{r^t_i, \ldots, r^t_n\}. \]

**Theorem 1.** If a task set with \( n \) tasks, and its corresponding slack, \( s \), satisfy,
1. \( s < e_{\text{max}} \),

2. \( (2r_{\text{max}}^t - (r_{\text{max}}^t)^2)^{n-1} \leq (r_{\text{min}}^t)(2r_{\text{min}}^t - (r_{\text{min}}^t)^2) \)

a schedule allowing more tasks to recover always leads to a higher system-level soft-error reliability.

**Proof.** We assume a schedule, \( S_1 \), guarantees recovery for \( k \) tasks and that these tasks are \( \{\tau_{n-k+1}, \ldots, \tau_n\} \). The corresponding system-level soft-error reliability, \( R(S_1) \), is

\[
R(S_1) = \prod_{i=1}^{n-k} \{1 - (1 - r_i^t)(1 - r_i^s(S_1))\} \times \prod_{i=n-k+1}^{n} \{1 - (1 - r_i^t)^2\}. \tag{3.11}
\]

Suppose there exists another schedule, \( S_2 \), which guarantees recovery for \( k - 1 \) tasks. The corresponding soft-error reliability, \( R(S_2) \), is

\[
R(S_2) = \prod_{i=1}^{n-k+1} \{1 - (1 - r_i^t)(1 - r_i^s(S_2))\} \times \prod_{i=n-k+2}^{n} \{1 - (1 - r_i^t)^2\}. \tag{3.12}
\]

We wish to show that \( R(S_1) \geq R(S_2) \). Since \( \prod_{i=n-k+2}^{n} \{1 - (1 - r_i^t)^2\} \) is in both \( R(S_1) \) and \( R(S_2) \), we omit it in the following prove. For a given task set with \( n \) tasks, there exist \( 2^n \) non-preemptive schedules. Let \( r_{\text{min}}^s = \min\{r_1^s(S_1), \ldots, r_n^s(S_{2^n})\} \) and \( r_{\text{max}}^s = \max\{r_1^s(S_1), \ldots, r_n^s(S_{2^n})\} \) for all possible schedules. The lower bound of \( R(S_1) \) and upper bound of \( R(S_2) \) are,

\[
R(S_1) \geq (1 - (1 - r_{\text{min}}^t)(1 - r_{\text{min}}^s))^n - k(2r_{\text{min}}^t - (r_{\text{min}}^t)^2) \tag{3.13}
\]

\[
R(S_2) \leq (1 - (1 - r_{\text{max}}^t)(1 - r_{\text{max}}^s))^{n-k+1}. \tag{3.14}
\]

If condition (i) in Theorem 1 is satisfied, there is a task whose execution time is larger than \( s \). For this task, it can never be recovered in any schedules. Hence, the
minimum probability of recovery for all tasks, \( r_{\text{min}}^s \), is equal to 0 and

\[
R(S_1) \geq (r_{\text{min}}^t)^{n-k}(2r_{\text{min}}^t - (r_{\text{min}}^t)^2).
\]  (3.15)

Given \( 1 < k < n \) and \( r_{\text{min}}^t < 2r_{\text{min}}^t - (r_{\text{min}}^t)^2 \), we have

\[
R(S_1) \geq (r_{\text{min}}^t)(2r_{\text{min}}^t - (r_{\text{min}}^t)^2).
\]  (3.16)

Based on the definition of \( r_i^t \) in (3.9), for each task, \( r_i^t \) is larger than or equal to \( r_i^s \), so \( r_{\text{max}}^t \geq r_{\text{max}}^s \) and

\[
R(S_2) \leq (2r_{\text{max}}^t - (r_{\text{max}}^t)^2)^{n-1}.
\]  (3.17)

If condition (ii) in Theorem 1 is satisfied, \( R(S_1) > R(S_2) \).

Since a task’s reliability depends on its execution time (in 3.2), Theorem 1 provides conditions to determine whether the execution times of tasks are close and whether the slack is relatively small. It also provides some intuitions on how to schedule tasks. That is, the schedule that maximizes the number of tasks capable of being recovered would maximize the system-level soft-error reliability. Observe that a task with a high priority and which executes earlier has a larger probability of recovering because less slack is consumed by prior tasks. In order to preserve slack for later tasks, high priority tasks should have short execution times. This is because a task with short execution time not only tends to have a higher reliability but also consumes less slack to recover. Hence, we propose a fast scheduling algorithm (FRIS) that assigns the priority to tasks according to their execution times. Given a task set that satisfies the conditions in Theorem 1 and is scheduled according to FRIS, i.e., \( p_i < p_j \) if \( e_i < e_j \); then the system-level soft-error reliability is maximized.

Since obtaining the system-level soft-error reliability can not only be used to evaluate our proposed framework, but also in other work, e.g., reliability-oriented
energy management [111], we provide a computationally efficient method to obtain the system-level soft-error reliability for task sets that satisfy Theorem 1 and which are scheduled by FRIS. This method reduces the overhead in finding the execution patterns satisfying (3.8). We simplify $r^s_i(S)$ to $r^s_i$ whenever doing so does not cause confusions.

Suppose a schedule of $n$ tasks is $S = \{\tau_1, \ldots, \tau_n\}$, where $p_i > p_{i+1}$ and $e_i < e_{i+1}$.

For $\tau_i$, we introduce two concepts:

- **Heavy set**, $\Omega^+_i$, is a subset of tasks in $\{\tau_1, \ldots, \tau_{i-1}\}$ where if all tasks in $\Omega^+_i$ fail, the slack needed to recover the tasks in $\Omega^+_i$ is larger than $s - e_i$.
- **Light set**, $\Omega^-_i$, is a subset of tasks in $\{\tau_1, \ldots, \tau_{i-1}\}$ where if all tasks in $\Omega^-_i$ fail, the slack needed to recover is smaller than or equal to $s - e_i$.

Tasks in both $\Omega^+_i$ and $\Omega^-_i$ are sorted by decreasing priority. Let $\Omega^-_{i,j}$ be the $j^{th}$ light set for task $\tau_i$. If $\Omega^-_{i,j} = \{\tau_1, \tau_2\}$, it means that the combined slack needed by $\tau_1$ and $\tau_2$ to recover is smaller than or equal to $s - e_i$. Based on this definition, one light set coincides with one execution pattern satisfying (3.8). Hence, finding all the light sets is equivalent to finding all the execution patterns satisfying (3.8), which can then be used to efficiently compute the system-level soft-error reliability. Below we state three lemmas which are used to find light sets.

**Lemma 1.** With FRIS, a set $\Omega_i$ is a heavy set for $\tau_i$ only when $\sum_{\tau_j \in \Omega_i} e_j > s - e_i$.

**Lemma 2.** If $\Omega^+_i$ is a heavy set of $\tau_i$, $\Omega^+_i$ is also a heavy set if $\Omega^+_i$ is a super set of $\Omega^+_i$, i.e., $\Omega^+_i \subset \Omega^+_i$.

**Lemma 3.** With FRIS, if $\Omega^+_i$ is a heavy set of $\tau_i$, it is also a heavy set of $\tau_{i+1}$.

For a task $\tau_i$, we divide all its light sets into groups. $G_{i,k}$, the $k^{th}$ group of $\tau_i$’s light sets, consists of light sets in which each set has exactly $k$ tasks. Based on this definition, for task $\tau_i$, there are $i$ groups, i.e., $G_{i,0}$ to $G_{i,i-1}$. Based on Lemmas 1–3 and the definition of $G_{i,k}$, we provide a guideline on how to find light sets for each task.
Theorem 2. For task sets satisfy Theorem 1 and are scheduled by FRIS, if a light set, $\Omega^-$, in $G_{i-1,j}$ satisfies $\sum_{\tau_k \in \Omega^-} e_k \leq s - e_i$, $\Omega^-$ is also a light set in $G_{i,j}$. If $\Omega^-$ is a light set in $G_{i-1,j-1}$, it is also a light set in $G_{i,j}$ if $e_{i-1} + \sum_{\tau_k \in \Omega^-} e_k \leq s - e_i$.

Based on Theorem 2, we find all light sets for each task iteratively following a dynamic programming strategy. The details of our method is shown in Alg. 2. We initialize $G_{i,0}$ for each task in Lines 2-8 and construct $G_{i,j}$ for each tasks in Lines 9–28. $G_{i,0}$ is initialized to $\emptyset$ if $e_i \leq S$, which means $\tau_i$ can be recovered if the slack is not consumed by other tasks. $G_{i,0} = \{\emptyset\}$ means $\tau_i$ can never be recovered. Based on Theorem 2, $G_{i,j}$ is first set to $\emptyset$ (in Line 11) and then additional light sets are added to $G_{i,j}$ in Lines 12-22. For each light set in $G_{i-1,k}$, if it is still a light set for $\tau_i$, it is added into $G_{i,k}$ (in Lines 12–16). We determine whether a light set in $G_{i-1,k}$ is also a light set in $G_{i,k}$ (in Line 13). Similarly, for each light set in $G_{i-1,k-1}$, we determine whether the set remains light if $\tau_{i-1}$ is added (Lines 17–22). Based on Lemma 2, if $G_{i,k} = \emptyset$, groups from $G_{i,k+1}$ to $G_{i,j-1}$ are also equal to $\emptyset$ (in Lines 23–26). Finally we return groups of light sets for each task, $\{G_{i,0}, \ldots, G_{i,i-1}\}$ in Line 29. We can use Alg. 2 to find light sets for all tasks in pseudo-polynomial time.

3.4.2 A general scheduling algorithm

Although FRIS is efficient and effective in scheduling tasks satisfying the conditions in Theorem 1, for general task sets, it may lead to suboptimal schedule. In this subsection, we present a general reliability improvement scheduling algorithm (GRIS) for general task sets where a task’s execution time can be any arbitrary value. The aim of GRIS is to guarantee that the system-level soft-error reliability is always higher than the optimal solution to the static recovery allocation problem.
Algorithm 2 Find Light Sets

1: procedure FIND_SET($S = \{\tau_1, \tau_2, \ldots, \tau_n\}$)
2: for task $\tau_i$ in $\{\tau_1, \tau_2, \ldots, \tau_n\}$ do
3: \hspace{1em} if $e_i \leq s$ then
4: \hspace{2em} $G_{i,0} = \{\emptyset\}$
5: \hspace{1em} else
6: \hspace{2em} $G_{i,0} = \emptyset$
7: \hspace{1em} end if
8: end for
9: for task $\tau_i$ in $\{\tau_2, \tau_3, \ldots, \tau_n\}$ do
10: \hspace{1em} for $k$ in $\{1, 2, \ldots, i-1\}$ do
11: \hspace{2em} $G_{i,k} = \emptyset$
12: \hspace{2em} for light set $\Omega^-$ in $G_{i-1,k}$ do
13: \hspace{3em} if $\sum_{\tau_j \in \Omega^-} (c_j) \leq s - e_i$ then
14: \hspace{4em} $G_{i,k} = \{G_{i,k} \cup \Omega^-\}$
15: \hspace{3em} end if
16: \hspace{2em} end for
17: \hspace{2em} for light set $\Omega^-$ in $G_{i-1,k-1}$ do
18: \hspace{3em} if $\sum_{\tau_j \in \Omega^-} (e_j) + e_{i-1} \leq s - e_i$ then
19: \hspace{4em} $\Omega^- = \{\Omega^- \cup \tau_{i-1}\}$
20: \hspace{4em} $G_{i,k} = \{G_{i,k} \cup \Omega^-\}$
21: \hspace{3em} end if
22: \hspace{2em} end for
23: \hspace{2em} if $G_{i,k} = \emptyset$ then
24: \hspace{3em} $G_{i,k+1} = \cdots = G_{i,i-1} = \emptyset$
25: \hspace{3em} break
26: \hspace{2em} end if
27: \hspace{1em} end for
28: \hspace{1em} end for
29: \hspace{1em} return $\{G_{i,0}, \ldots, G_{i,i-1}\}$ for $i$ in $\{1, 2, \ldots, n\}$
30: end procedure
The static allocation problem is defined as follows. For a given task set, find a subset of tasks $\Phi$ to allocate recovery to in order to maximize the system-level soft-error reliability. GRIS uses two steps. It first finds the optimal solution, set $\Phi$, for the static recovery allocation problem. It then elevates the priority of tasks in $\Phi$ to further improve the soft-error reliability.

It can be seen that solving the static recovery allocation problem is the key to GRIS. This static recovery allocation problem is a variation of the knapsack problem [32], which can be solved using dynamic programming. Let $\Phi\{i, s'\}$ be a set of tasks that achieves the maximum system-level soft-error reliability when (i) some tasks in $\{\tau_1, \tau_2, \ldots, \tau_i\}$ can be recovered and (ii) the slack is less than or equal to $s'$. Given this construction, a dynamic program for solving the knapsack problem [32] can be leveraged to find the optimal solution to the static allocation problem, $\Phi\{n, s\}$. We omit the details due to space limit.

Based on the solution to the static allocation problem, tasks in $\Phi$ have a higher priority than tasks not in $\Phi$. The priority is assigned in such a way that $p_i < p_j$ if $e_i > e_j$ for both $\tau_i$ and $\tau_j$ in $\Phi$, or both $\tau_i$ and $\tau_j$ not in $\Phi$. With dynamic recovery allocation, this scheduling algorithm guarantees that tasks in $\Phi$ can always schedule its recovery. Tasks not in $\Phi$ can still have a recovery if some tasks in $\Phi$ successfully complete their execution. Hence, GRIS achieves a higher system-level soft-error reliability than the optimal solution to the static recovery allocation problem. Compared to FRIS, although GRIS is more complicated, simulation results in Section 3.6 show that GRIS is more efficient in improving system-level soft-error reliability than FRIS in some cases.

3.5 Lifetime Reliability Satisfaction

To improve soft-error reliability, it is desirable to execute tasks at the highest core frequency. However, this core frequency may violate the lifetime reliability constraint
in (3.5). To address this problem, we propose a DVFS based frequency reduction heuristic to reduce some tasks’ core frequencies and guarantee the lifetime reliability requirement.

We consider the lifetime reliability due to both operating temperature and thermal cycling. Reducing core frequencies for tasks is an effective way to reduce the operating temperature. However, reducing core frequency may also introduce thermal cycles. For the task model and recovery allocation method considered in Section 3.2, the CPU is only active in the earlier part of a period. Hence, if all tasks are running at the same frequency, there is only one thermal cycle in each period. However, if the frequency of task $\tau_i$ is lower or higher than both $\tau_{i-1}$ and $\tau_{i+1}$, more thermal cycles may be introduced. Hence, we reduce the core frequencies of tasks while avoiding additional thermal cycles.

Reducing a task’s core frequency increases its execution time and reduces both its soft-error reliability and available slack. This task, in the end, has a higher failure probability and is more likely to consume slacks for recovery. Since a high priority task first consumes slack to recover, and may affect the reliability of low priority tasks, which are executed later, we adopt the general principle of executing tasks with higher priorities at the highest frequency while lowering the core frequency when execute tasks with lower priorities. The key question now is whether to only reduce core frequency for the task with the lowest priority, or whether the core frequencies of all low priority tasks can be reduced.

We present a method to choose a task to reduce its core frequency in such a way that doing so maximizes the power saving and minimizes the influence on other tasks’ reliability. Suppose the processor supports $l$ frequency levels, and level 1 is the lowest frequency, i.e., $f(1) = f_{\text{min}}$, and level $l$ is the highest frequency, i.e., $f(l) = f_{\text{max}}$. $f(j)$ represents the core frequency at the $j^{th}$ level. We define $R_t((f(j), f(k)))$ as the
Algorithm 3 Frequency Reduction

```
1: procedure freq_red(S, F)
2:  while LTR(F, S) > LTR_{TH} do
3:    for τ_i ∈ {τ_1, ..., τ_{n-1}} do
4:      if d > s + \frac{e_i f(j)}{f_i - 1} + \sum_{\tau_j \neq \tau_i} \frac{e_j f(j)}{f_j} then
5:        if \( \mathcal{R}_i(f_i, f_i - 1) > \mathcal{R}_{\text{max}} \) and \( f_i > f_{i+1} \) then
6:          \( \mathcal{R}_{\text{max}} = \mathcal{R}_i(f_i, f_i - 1) \)
7:          \( \tau_i = \tau_t \)
8:      end if
9:    end for
10:   reduce core frequency for \( \tau_t \)
11:   update F and calculate LTR(Sched, F)
12: end while
13: end procedure
```

The power-time ratio of task \( \tau_i \) when reducing core frequency from \( f(j) \) to \( f(k) \),

\[
\mathcal{R}_i(f(j), f(k)) = \frac{\text{Power}(f(j)) - \text{Power}(f(k))}{\text{Time}_i(f(k)) - \text{Time}_i(f(j))},
\] (3.18)

where \( \text{Power}(f(j)) \) is the power consumption when the core frequency is \( f(j) \). The power consumption depends on the hardware platform and can be obtained using an existing model [111]. \( \text{Time}_i(f(j)) \) is the execution time at \( f(j) \) and \( \text{Time}_i(f(j)) = f(l) \times e_i/f(j) \). We iteratively reduce the core frequency of the tasks, starting with the one with the highest power-time ratio, until the lifetime reliability is satisfied. The details are shown in Alg. 3. This heuristic starts with a given schedule \( S = \{\tau_1, \tau_2, ..., \tau_n\} \) and frequency assignment \( F = \{f_1 = \cdots = f_n = f(l)\} \). We iteratively reduce tasks’ core frequencies until the lifetime reliability is larger than a given threshold (in Lines 2–13). In each iteration, we find a task where reducing one level of this task’s core frequency achieves the largest \( \mathcal{R} \) (in Lines 3–10). \( f_i - 1 \) denotes reducing one frequency level for \( \tau_i \). After reducing the core frequency of the chosen task, \( \tau_t \), the frequency assignment \( F \) is updated and a new MTTF is calculated (in Line 12).
3.6 Evaluation

We evaluate the proposed RIF by conducting simulations and comparing it with existing approaches.

3.6.1 Experimental setup

We compared RIF with two representative approaches: generalized shared recovery approach (GSR) [113] and partial replication and speedup approach (PRS) [116]. GSR represents a series of work in which recovery is dynamically allocated to failed tasks, but the total number of recoveries is fixed and determined offline [111, 113]. GSR is composed of two modules; one allocates recovery and the other reduces core frequency to minimize power consumption. We kept the first GSR module but replaced the second module with Alg. 3 to satisfy the lifetime reliability constraint. PRS is a greedy algorithm based approach to maximize the minimum of lifetime reliability and soft-error reliability. However, the allocation of recovery is determined offline. We evaluated RIF with FRIS in Section 3.4.1 (RIF-FRIS) and with GRIS in Section 3.4.2 (RIF-GRIS). The probability of failure (PoF) due to soft errors, which is defined as $1 - R$ and $R$ is the system-level soft-error reliability, is used as a metric for comparison. We normalized the measured probability of failure to the probability of failure of every task having a recovery. The average soft-error rate at $f_{\text{max}}$, $\lambda_0$, is $10^{-6}$, and the parameter, $c$, in (3.1) is 3 [116].

The comparisons were conducted on a simulator, which is constructed based on Nvidia’s TK1 chip [76]. The operating temperature is calculated using a RC thermal modeling tool [31] and thermal parameters are extracted from the TK1. We specified three different configurations for the task set. In the first configuration, the task set is composed of 5 randomly generated tasks, and a task’s execution time is random in the range of 0.75–1.25 seconds. The task set is composed of 10 tasks in the second configuration where each task is similarly generated as in the first configuration.
TABLE 3.1

AVERAGE EXECUTION TIMES OF TASKS

<table>
<thead>
<tr>
<th>Name</th>
<th>bitcount</th>
<th>susan</th>
<th>qsort</th>
<th>dijkstra</th>
<th>patricia</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time</td>
<td>965 ms</td>
<td>140 ms</td>
<td>185 ms</td>
<td>135 ms</td>
<td>4525 ms</td>
</tr>
<tr>
<td>Name</td>
<td>stringsearch</td>
<td>blowfish</td>
<td>sha</td>
<td>crc32</td>
<td>cjpeg</td>
</tr>
<tr>
<td>Execution Time</td>
<td>6 ms</td>
<td>1360 ms</td>
<td>57 ms</td>
<td>858 ms</td>
<td>18 ms</td>
</tr>
</tbody>
</table>

In the last configuration, the task set is composed of 10 tasks from the MiBench benchmark suite [25], which represent automotive, network, consumer, and security applications. The average execution time of each task was measured with the highest core frequency on the TK1 (see Table 3.1). For a task set, the deadline is the sum of the slack and tasks' execution times. We assume the slack is 1 s, 1.5 s, 2 s, 2.5 s, and 3 s, respectively.

3.6.2 Experimental results

We first compared RIF to GSR and PRS when the task set is composed of 5 randomly generated tasks. The results are summarized in Fig. 3.2. As can be readily seen, both RIF-FRIS and RIF-GRIS achieve a lower probability of failure than GSR and PRS in all cases. RIF-FRIS and RIF-GRIS have similar performance, and their difference can be ignored when compared to GSR and PRS. When the slack is 1 s, RIF, either with FRIS or with GRIS, reduces the probability of failure by about 29% compared to GSR and by about 16% compared to PRS. The benefit of our proposed RIF is more obvious when the slack is large. Compared to PRS, RIF allocates more recoveries to tasks and the average probability of failure of RIF is only about of 0.0009%, 0.0007%, 0.0003%, and 0.0002% of PRS when the slack is 1.5 s, 2 s, 2.5 s,
and 3 s, respectively. Compared to GSR, RIF allocates more recoveries when the slack is 1.5 s, 2 s, and 2.5 s, and the average probability of failure of RIF is only about 0.0007%, 0.0005%, 0.0008% of GSR. The number of recoveries in GSR depends on the length of the slack, and a larger slack leads to more recoveries. Although GSR can allocate more recoveries when the slack is 3 s, RIF reduces the probability of failure by about 52%. This simulation shows that GSR is effective in reliability improvement only when the slack is large, but our proposed RIF works well in all cases.

We extended our evaluation of RIF when the workload is heavy (see Fig. 3.3(a)) and when a task’s execution time does not follow any distributions (see Fig. 3.3(b)). In Fig. 3.3(a), when the slack is 1 s, 1.5 s, 2 s, 2.5 s, and 2.5 s, the average probability of failure of RIF is only 75%, 0.005%, 0.004%, 0.001%, and 19% of GSR, and 91%, 0.006%, 0.005%, 0.002%, and 0.001% of PRS, respectively. This result is similar to that shown in Fig. 3.2. The task set in Fig. 3.3(b) is composed of tasks from MiBench benchmark suite, where some tasks’ execution times are short and some are large (see Table 3.1). The benefit of RIF becomes smaller since the execution times of tasks, e.g., stringsearch and cjpeg, are very short, both RIF and PRS can allocate recovery to them. Thanks to the dynamic recovery allocation method we proposed, RIF can
allocate recovery to other tasks such as blowfish and bitcount, but the improvement in the system-level soft-error reliability is relatively small. Since the execution time of patricia is larger than the slack, less recoveries are allocated in GSR. Hence, the probability of failure of GSR is much higher than RIF and PRS. Both Figs. 3.2 and 3.3 show RIF is better than GSR and PRS in terms of soft-error reliability improvement.

We compared RIF-FRIS and RIF-GRIS (see Fig. 3.4). RIF-FRIS performs better than RIF-GRIS when the slack is small, and the probability of failure is reduced by about 1.15% and 0.24% when the slack is 1 s and 1.5 s, respectively. In this case, the task set and the slack satisfy the conditions in Theorem 1 (in Section 3.4.1), so RIF-

Figure 3.3. Normalized probability of failures when task set is composed of (a) 10 randomly generated tasks; (b) tasks from MiBench benchmark suite.
Figure 3.4. Comparison of RIF-FRIS and RIF-GRIS with different lengths of slack.

FRIS is a better approach in scheduling tasks. However, RIF-GRIS is better when the slack is 2 s, and 2.5 s, as RIF-GRIS reduces the probability of failure by about 0.98% and 2.02%, respectively. Note that although the difference of the probability of failure in RIF-FRIS and RIF-GRIS is small, it can result in a large difference in soft-error reliability with a high soft-error rate. This comparison shows that although GRIS is more complicated than FRIS, it is sometime necessary and can achieve a higher soft-error reliability.

3.7 Summary

This work develops a method to improve soft-error reliability by allocating recoveries to any failed tasks. Utilizing this method, a reliability improvement framework is designed to maximize system-level soft-error reliability under lifetime reliability and hard real-time constraints. The system-level soft-error reliability is improved by...
statistically scheduling tasks leading more tasks can be recovered. Lifetime reliability is satisfied by reducing core frequencies for tasks having low priority but with high power consumption. The simulation results show that our work is effective in improving soft-error reliability compared to existing approaches without violating real-time and lifetime reliability constraints.
CHAPTER 4

IMPROVING RELIABILITY OF REAL-TIME SYSTEMS ON “BIG-LITTLE” TYPE MPSoCs

This chapter introduces an on-line framework to maximize soft-error reliability while satisfying a lifetime reliability constraint for soft real-time systems executed on MPSoCs composed of high-performance cores and low-power cores. Based on the runtime cores’ frequencies and utilizations, this framework performs workload migration between the high-performance cores and low-power cores to achieve improved soft-error reliability.

4.1 Introduction

To address power/energy concerns, various heterogeneous MPSoCs have been introduced [107]. A popular MPSoC architecture that is often used in power/energy-conscious real-time embedded applications is composed of pairs of high-performance (HP) cores and low-power (LP) cores. Following the terminology introduced by ARM [5], we refer to this architecture as the “big–little” architecture. Nvidia’s variable symmetric multiprocessing (vSMP) [82] is such an example. Such HP and LP cores present unique performance, power/energy, and reliability tradeoffs, which are investigated in this chapter.

Resource management in heterogeneous MPSoCs has been widely studied [38, 12, 44, 24, 60], but few work targets the “big–little” architecture [56, 4, 9, 97]. In this architecture, HP (LP) cores are homogeneous and both HP and LP cores have the same instruction set architecture (ISA). However, “big–little” type MPSoCs may
support different execution models. In one model, represented by Nvidia’s TK1 [76] and Samsung’s Exynos 5410 [95], one HP core is paired with one LP core, and the HP and LP cores in one pair cannot work simultaneously. In another model, represented by Nvidia’s TX2 [78] and NXP’s i.MX8 [84], although HP and LP cores can work simultaneously, all HP (all LP) cores must execute at the same frequency. We aim to design a resource management framework that is adaptive to different execution models.

Since many real-time embedded systems that use MPSoCs are deployed in critical applications and are expensive as well as inconvenient to replace, lifetime reliability due to permanent faults as well as soft-error reliability due to transient faults are important design considerations. Although there exist several efforts that either target soft-error reliability [111, 114, 113] or lifetime reliability [17, 43, 21, 13, 60], only a few papers have examined both soft-error reliability and lifetime reliability together [23, 48, 116, 61]. In addition, runtime workload variations further complicate the problem of improving the system overall reliability. Hence, designing an online approach considering both lifetime reliability and soft-error reliability becomes necessary.

This chapter systematically addresses the reliability concerns for real-time systems running on “big–little” type MPSoCs. Since transient faults occur much more frequently than permanent faults [112], we focus on increasing soft-error reliability without sacrificing lifetime reliability. Specifically, we solve the problem of maximizing soft-error reliability while satisfying temperature, real-time, and lifetime reliability requirements. Motivated by many real world applications such as mobile devices and in-vehicle infotainment systems [62], we are particularly interested in developing an on-line framework to address unavoidable workload and environment variations.

Our on-line framework, referred to as DRIF (for Dynamic Reliability Improvement Framework), solves the problem outlined above by dynamically and judiciously
scaling core frequencies to increase soft-error reliability. By leveraging the power and performance features of the “big–little” type MPSoCs, we dynamically migrate workload and activate the most power efficient cores to execute tasks. Meanwhile, in order to reduce the computational overhead to check whether the lifetime reliability caused by a thermal profile is larger than a lifetime reliability constraint, we design a tool, referred to as LTR-Checker, which is computation efficient to use at run time.

This work makes three main contributions.

- We propose a computation efficient method to determine whether a given temporal thermal profile would respect the corresponding lifetime reliability threshold.

- By performing extensive experiments on a hardware platform, we experimentally establish a suitable task migration guideline allowing tasks executed on the most power efficient cores.

- We develop an on-line framework to maximize soft-error reliability under temperature, real-time, and lifetime reliability constraints by scaling cores’ frequencies and selecting the most power efficient cores to execute tasks.

We have implemented and validated our framework on Nvidia’s TK1 [76] chip and TX2 [78] chip, respectively. Based on the results obtained from running tasks from the MiBench benchmark suite [25], we show that DRIF increases the no soft error occurring time at least 2 more days than existing approaches.

4.2 Related Work

As a special type of heterogeneous MPSoCs, the “big–little” type MPSoCs use two types of cores: the LP cores offer high power efficiency while the HP cores provide maximum computing performance [5]. This type of MPSoCs provides flexibility to balance the performance and power, and facilitates ease of use [121]. Since different execution models introduce unique constraints, e.g., the the HP core and LP core in the same pair cannot work simultaneously, or all HP (all LP) cores must execute at the same frequency, most resource management approaches for heterogeneous MPSoCs
are not applicable for the “big–little” architecture [88, 43, 12, 13, 31]. Focusing on this architecture, Liu et al. [56] proposed an iterative approach for mapping multi-threaded applications on MPSoCs composing of multiple core types to achieve high performance and power efficiency. Annamalai et al. [4] designed a novel technique to dynamically swap threads between HP cores and LP cores and change the core frequency to achieve a high throughput/Watt. Considering the constraints for HP cores and LP cores, Carroll et al. [9] investigated the mechanisms for frequency scaling, and proposed a technique to reduce energy consumption. Singla et al. [97] designed an on-line method to predict and reduce power and runtime temperature for “big–little” type MPSoCs. While the above work considers the specific features of the “big–little” architecture, none of them focuses on lifetime reliability or soft-error reliability.

There exist several efforts that directly aim to increase soft-error reliability [111, 27, 42, 114] or lifetime reliability by using scheduling [43, 6, 22, 24]. In order to improve soft-error reliability, Zhao et al. proposed a method to allocate recoveries for tasks [111, 114] while Nahar et al. assigned redundancies to tasks statically [70]. Fan et al. proposed a dynamic voltage and frequency scaling based method to reduce power consumption under soft-error reliability constraint. Although these methods are effective in improving and ensuring soft-error reliability, they usually reduce lifetime reliability with a high operating temperature. For periodic tasks running on an MPSoC, Huang et al. proposed an analytical model to estimate lifetime reliability of MPSoCs and a task mapping and scheduling algorithm to guard against aging effects [43]. Bolchini et al. dynamically determined the most effective mapping of tasks to minimize network-on-chip energy consumption and maximize lifetime reliability [6]. Das et al. proposed a machine learning based algorithm to handle inter- and intra-application variations and reduce peak temperature and thermal cycling [24]. These methods are designed to increase lifetime reliability but weaken soft-error re-
liability.

Our proposed framework considers soft-error reliability and lifetime reliability, both of which have not typically been examined together. The work by Das et al. aims to jointly improve soft-error reliability and lifetime reliability by mapping tasks to all cores and scaling core frequencies [23]. However, their solution is too computationally intensive to use at run time. Kapadia et al. proposed a framework to optimize performance and energy [48]. Although transient and permanent faults are considered, their work does not increase reliability but only focuses on reducing power under lifetime reliability and soft-error reliability constraints. Zhou et al. proposed an off-line technique to maximize system availability by allocating replicated tasks and determining the core frequency statically [116]. Although these works consider both lifetime reliability and soft-error reliability, they are off-line approaches and ignore the specific features of “big–little” type MPSoCs. In this chapter, we focus on “big–little” type MPSoCs and propose to maximize soft-error reliability under lifetime reliability constraint.

4.3 System Model and Problem Formulation

In this section, we present the hardware platform as well as the task and reliability models used in our framework.

4.3.1 Hardware model

We consider on “big–little” type MPSoCs with \( n \) HP and \( m \) LP cores. We assume that both HP cores and LP cores support DVFS and have multiple frequency levels [76, 78]. A core dissipates static power when it is idle and consumes additional active power when it performs operations [31]. Both active and static power are related to the core’s frequency. Let the utilization of a core in a given time interval \( |t| \) be \( u = \frac{|t_o|}{|t|} \), where \( |t_o| \) is the amount of time that the core performs operations [31]. A
core’s utilization is commonly used to estimate real-time performance and soft-error reliability.

We consider two execution models of “big–little” MPSoCs in this chapter. In the first execution model, referred to as Hetero-Paired model and represented by Nvidia’s TK1 [76] and Samsung’s Exynos 5410 [95], HP cores and LP cores are paired, and the paired HP core and LP core cannot be active simultaneously. In the second execution model, referred to as Homo-Grouped model and represented by Nvidia’s TX2 [78] and NXP’s i.MX8 [84], all cores can work simultaneously, but HP (LP) cores must execute at the same frequency. There exist other execution models, where HP and LP cores can run simultaneously with their own core frequencies, but such models are not widely supported by MPSoCs.

4.3.2 Task model

Similar as in Chapter 2, we assume that MPSoCs execute independent periodic tasks with soft deadlines such as those found in multimedia and communication applications. A task is associated with a tuple \( \tau_i = \{ d_i, e^H_i, e^L_i \} \) where \( d_i \) is the deadline, and \( e^H_i \) and \( e^L_i \) represent the worst-case execution time when running on an HP core and LP core, respectively. Generally \( e^H_i \leq e^L_i \). Since all the jobs of the \( i^{th} \) task have the same properties, \( \tau_i \) also denotes the jobs of the \( i^{th} \) task. Tasks on each core are scheduled according to a real-time scheduling policy such as earliest deadline first or rate monotonic scheduling [55]. In this work, we adopt a mapping approach where tasks are assigned to cores at design time to balance the workload of cores [59]. We guarantee the real-time constraint by ensuring that the utilization of each core is lower than utilization bound for schedulability [30, 60].
4.3.3 Soft-error reliability

In this chapter, we aim to maximize reliability in the presence of soft errors caused by transient faults. Same as in Chapter 3, the soft-error reliability of a single core in a time interval is the probability that soft errors occur during the time interval [116],

\[ r(f, t) = e^{-\lambda(f) \times u \times |t|}. \] (4.1)

The \( f \) is the core frequency, \( |t| \) is the length of time interval, and \( u \) is the core’s utilization in this time interval. \( \lambda(f) \) is the average fault rate depending on \( f \) [116],

\[ \lambda(f) = \lambda_0 \times 10^{cx(f_{\text{max}} - f)} f_{\text{max}} - f_{\text{min}}. \] (4.2)

\( \lambda_0 \) is the average faults rate at highest core frequency. \( f_{\text{min}} \) and \( f_{\text{max}} \) are the minimum and maximum core frequency and \( c (c > 0) \) is a hardware specific constant that indicates the sensitivity of fault rates to frequency scaling. The core frequency and utilization can be treated as constant in the time interval [116, 61]. This model indicates that improving core frequency is effective in improving soft-error reliability.

For a “big–little” type MPSoC with \( n \) active HP cores and \( m \) active LP cores, the soft-error reliability in the \( i^{th} \) time interval, \( t_i \), is

\[ R(t_i) = \prod_{j=1}^{n} r_j^{HP}(f_j, t_i) \times \prod_{j=1}^{m} r_j^{LP}(f_j, t_i), \] (4.3)

where \( r_j^{HP}(f_j, t_i) \) and \( r_j^{LP}(f_j, t_i) \) are the soft-error reliability of the \( j^{th} \) HP (LP) core in the time interval \( t_i \). The aim of this chapter is to maximize soft-error reliability of the MPSoC in each time interval.
4.3.4 Lifetime reliability

Lifetime reliability, measured by MTTF, is dependent on multiple wear-out effects [13]. The existing tool [108] provides a precise system-level lifetime reliability estimation but with a heavy computational overhead. For applications, which tolerates less precise lifetime reliability estimation but requires a low computation overload, a simple and device-level lifetime reliability modeling tool should be considered. For the sake of simplicity, we consider EM as the primary source of permanent faults in this work. Other device fault mechanisms can be incorporated using the sum-of-fault rate model [21, 23]. Since the tasks are executed periodically, the temperature variance with respect to time will be also periodical after the system stabilization, so we assume the thermal profiles are same in each task set’s hyperperiod, $hp$. Based on the thermal profile in a hyperperiod, the MTTF can be calculated by

$$MTTF = |hp| \times \sum_{i=0}^{\infty} e^{-(i \times A)^\beta}, \quad (4.4)$$

where $|hp|$ is the length of the hyperperiod and $\beta$ is the slope parameter in the Weibull distribution [43]. $A$ is a temperature-related parameter. If one hyperperiod can be divided by $p$ time intervals of the same length, and the operating temperature is constant in each time interval, we calculate $A$ as

$$A = \sum_{i=1}^{p} \frac{|t|}{\alpha(T_i)}, \quad (4.5)$$

where $|t|$ and $T_i$ are the length of the time interval and the temperature at the $i^{th}$ time interval, respectively. $\alpha(T_i)$ relates to the arrival rate of permanent faults and depends on the hardware and temperature $T_i$ [43].
4.4 “Big-Little” Type MPSoCs and Core Power Features

In this section, we describe the “big–little” type MPSoCs consisting of LP and HP cores, especially explore their unique power features. We first observe that executing tasks on a LP core may consume more power and energy than executing on a HP core. We provide a measurement-based method to quantitatively compare the power and energy consumption of HP and LP cores. Based on this method and the measurement results, we establish a suitable task mapping and migration guideline to migrate tasks between cores and reduce a chip’s power consumption.

Whereas the primary goal of “big–little” MPSoCs is to reduce power consumption by executing a light workload on the LP cores, an LP core may consume more power than an HP core. To totally capture the power consumption behavior of “big–little” MPSoCs, we have conducted a series of measurement-based experiments. We measure the power consumption of the HP core and LP core in Nvidia’s TX2 chip and TK1 chip\(^1\) when executing tasks from MiBench benchmark suite [25]. We use FLUKE AC/DC current clamp meters [29] and National Instruments USB-6216 data acquisition system [72] to acquire power consumption when cores execute at different core frequencies and at different utilizations.

To generally evaluate the power features of HP and LP cores, we propose a measurement-based method to quantitatively compare power consumption of HP and LP cores. This method measures and compares the power consumption of cores with different frequencies and utilizations, and the comparison results can guide the mapping of tasks. A low utilization means that the workload is light, a core consumes less active power, and the leakage power may be dominated. In order to maintain the core’s utilization at a specific level, we develop a feedback-based tool which can

---

\(^1\)Note that TX2 is composed of ARM Cortex A57 cores geared for multithreading, and Nvidia’ Denver cores for high single-thread performance with dynamic code optimization [83]. In this measurement, we only consider single-thread applications for TX2, therefore the Denver core is an HP core and the ARM core is an LP core.
Figure 4.1. The power consumption of an HP (Denver) core and an LP (ARM) core on TX2 under different utilization and frequency levels.

maintain the core’s utilization at a specific value.

The measured power consumption of HP and LP cores on TX2 are illustrated in Fig. 4.1. The results show that for any core frequency, both HP and LP cores have a higher power consumption with a heavier workload. However, LP cores are not always power efficient. The LP core consumes less power than the HP core only when the core frequency is low and the workload is light. For other platforms such as Nvidia’s TK1, we have similar observations. Note that since TK1 has only one LP core, it is difficult to directly measure or precisely estimate the power consumption of this LP core. Hence, we use the steady-state temperatures without external heatsink
as an indirect method to measure power consumption. When using steady-state temperature to measure power consumption, measurements must be performed with the same ambient temperature and heat transfer conditions. A higher steady-state temperature indicates higher power consumption. Fig. 4.2 illustrate that the LP core operates with a lower temperature as well as has a lower power consumption than the HP core only when the utilization and core frequency are low.

One possible reason to explain this phenomenon is that the HP and LP core have different microarchitectures, such as on TX2. Although HP and LP cores on TK1 have the same microarchitecture, the transistors in the HP core and LP core have different threshold voltages. The LP core consumes low leakage power but requires high voltage to operate at high frequencies. On the contrary, the HP core can work at high frequency with a low voltage. The measurement results reveal that in order to reduce power consumption of MPSoCs, we should keep the workload light in the LP cores, and it is necessary to migrate tasks between HP and LP cores if cores’ utilizations vary at run time.
Based on the data collected from our extensive experiments, we can establish a suitable task mapping and migration guideline guiding the selection of cores for executing workload to balance the power consumption and performance. This guideline indicates that whether the LP core or the HP core consumes less power for each given core frequency and core utilization. With this guideline, we should map and migrate tasks to the core consuming less power. As an example, TABLE 4.1 and TABLE 4.2 present the guideline for Nvidia TX2 and TK1, respectively. In this table, “HP” (“LP”) indicates the HP (LP) core is more power efficient with the corresponding core frequency and utilization, so the workload should be running on an HP (LP) core. Note that due to small variations in ambient temperature, as well as chip operating voltage and current, the power consumption may vary slightly even for exactly the same workload. Therefore, it is not sufficient to conclude that a core always consumes less power when its measured power is lower than that of another core by a small amount. For TX2, we treat two measured power values as the same if their difference is smaller than 0.1 W, which is the resolution of our sensors. For TK1, we
TABLE 4.2

TASK MAPPING AND MIGRATION GUIDELINE FOR TK1

<table>
<thead>
<tr>
<th>Utilization</th>
<th>Core Frequency (in GHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.092</td>
</tr>
<tr>
<td>100%</td>
<td>HP</td>
</tr>
<tr>
<td>85%</td>
<td>HP</td>
</tr>
<tr>
<td>70%</td>
<td>HP</td>
</tr>
<tr>
<td>55%</td>
<td>HP</td>
</tr>
<tr>
<td>40%</td>
<td>HP</td>
</tr>
</tbody>
</table>

treat two temperatures are same if their difference is smaller than 0.5°C, which is the resolution of our thermal sensors. In TABLE 4.1, “-” indicates that the difference in power consumption of an HP core and an LP core is smaller than this threshold. In this case, workload can run either on an HP core or an LP core.

In our work to dynamically improve reliability, we will use this guideline to migrate tasks between HP and LP cores at run time to guarantee tasks are always executed on the most power efficient cores. This task migration reducing power consumption and operating temperature allows the cores to execute at a high core frequency and achieves a high soft-error reliability.

4.5 Problem Formulation and Framework Overview

In this section, we first formulate the problem addressed in this chapter and then describe our solution DRIF at high level.
4.5.1 Problem formulation

The problem that we aim to solve is motivated by applications such as in-vehicle infotainment systems. For such systems, tasks are expected to complete before their deadlines, and both lifetime and soft-error reliability are critical to guarantee the safety of human drivers and passengers [62]. Infotainment systems have soft real-time, power, lifetime reliability, and soft-error reliability constraints, and are typically run on operating systems [62]. At the same time, the infotainment and other in-vehicle computational subsystems should be power efficient especially for electric vehicles [68]. Hence, such systems have soft real-time, peak temperature, lifetime reliability, and soft-error reliability constraints. Furthermore, the workload in these systems can vary significantly at run time due to variations in input data and the environment.

Before formulating the problem, we first introduce two definitions.

**Definition 1.** A sampling window (SW) is defined as a time interval during which the temperature is constant.

**Definition 2.** A profiling window (PW) is composed of multiple equal-length sampling windows.

We determine the core frequencies and cores’ workloads for each sampling window, and the profiling window is used to estimate lifetime reliability. The soft-error reliability, frequency, utilization, and operating temperature of the \( j^{th} \) HP (LP) core at the \( i^{th} \) sampling window are denoted by \( r(SW_i, HP_j) \) (\( r(SW_i, LP_j) \)), \( f(SW_i, HP_j) \) (\( f(SW_i, LP_j) \)), \( u(SW_i, HP_j) \) (\( u(SW_i, LP_j) \)), and \( T(SW_i, HP_j) \) (\( T(SW_i, LP_j) \)).

Assume that a profiling window is composed of \( p \) sampling windows and the MPSoC has \( n \) HP cores and \( m \) LP cores. Our objective is to maximize the system-
level soft-error reliability in each profiling window,

\[ R = \prod_{i=1}^{p} \left( \prod_{j=1}^{n} r(SW_i, HP_j) \times \prod_{j=1}^{m} r(SW_i, LP_j) \right). \quad (4.6) \]

The solution to (4.6) must satisfy these constraints:

\[ T(SW_i, HP_j) \leq T_{th}, \forall SW_i, \forall HP_j \quad (4.7) \]
\[ T(SW_i, LP_j) \leq T_{th}, \forall SW_i, \forall HP_j \quad (4.8) \]
\[ u(SW_i, HP_j) \leq u_{th}, \forall SW_i, \forall HP_j \quad (4.9) \]
\[ u(SW_i, LP_j) \leq u_{th}, \forall SW_i, \forall HP_j \quad (4.10) \]
\[ LTR(TP(PW)) \geq LTR_{th}. \quad (4.11) \]

The first two constraints require the temperature of both HP and LP cores are less than the thresholds \( T_{th} \) in any sampling window. Note that this temperature constraint also limits the power consumption of the system. The third and fourth constraints capture the real-time requirement, where \( u_{th} \) is the upper bound on utilization to satisfy schedulability. The last constraint requires the lifetime reliability resulting from the thermal profile, \( TP(PW) \), to be not less than a threshold \( LTR_{th} \). For soft real-time systems, temporarily violating the real-time and lifetime reliability constraints is acceptable, but the temperature constraint must be satisfied to avoid thermal throttling. One parameter in (4.6)–(4.11) is cores’ frequencies and we assume a core’s frequency is 0 when the core is idle.

Different execution models of “big–little” type MPSoCs introduce different execution related constraints. For the Hetero-Paired execution model, the paired HP core and LP core cannot work simultaneously. If the \( j^{th} \) HP core is paired with the
\( j^{th} \) LP core, one of them must be idle, i.e.,

\[ f(SW_i, HP_j) \times f(SW_i, LP_j) = 0. \quad (4.12) \]

We assume that a core whose frequency is 0 is powered-off. For the Homo-Grouped execution model, all HP (LP) cores should have the same core frequency, i.e.,

\[ f(SW_i, HP_j) = f(SW_i, HP_{j+1}), \forall j \quad (4.13) \]
\[ f(SW_i, LP_j) = f(SW_i, LP_{j+1}), \forall j. \quad (4.14) \]

Our framework is applicable to both execution models and dynamically improves the soft-error reliability under the temperature, real-time, and lifetime reliability constraints in each profiling window.

In this formulated problem, the system-level soft-error reliability, lifetime reliability, operating temperature, and power consumption are functions of core frequencies, and also related to mapping of tasks. At the end of each profiling window, our proposed framework solves the formulated problem for each sampling windows in the next profiling window. The solution indicates the core frequencies of HP and LP cores as well as mapping of tasks. In order to solve the formulated problem, there are two main challenges that we need to overcome: (i) since the history (i.e., tasks’ execution times) does not always reflect the future, it is possible for the constraints to be violated when using history-based predictions, and (ii) a highly efficient algorithm is needed to avoid excessive overhead. We address these challenges by proposing an on-line framework to (i) obtain system runtime status, and (ii) dynamically migrate tasks between cores, power off idle cores, and determine core frequencies based on the system status in history.
4.5.2 Overview of reliability improvement framework

As stated earlier in the chapter, to better respond to workload and environmental changes that are unavoidable in real-time embedded systems, we aim to develop an on-line approach to solve the problem defined in (4.6)–(4.11) by taking into consideration of execution models given in (4.12) or (4.13)–(4.14). The basic idea of our framework, DRIF, is to incrementally solve the optimization problem by using the history of system states in the previous profiling window. The system state includes which cores are active and each active core’s frequency, operating temperature, and utilization. Note that our method can be easily applied to any arbitrary history window size. DRIF consists of three main components: a schedule generator (SG), which is triggered at the beginning of each profiling window, a schedule executor (SE), which is triggered at the beginning of each sampling window, and a state collector (SC), which collects the system state in each sampling window (see Fig. 4.3).

DRIF works as follows. In each sampling window, SC collects and saves the system state. At the end of each profiling window, the system state during this profiling window is sent to SG. Based on the state information, SG then generates
a solution, called schedule, which specifies cores’ workloads and frequencies for each sampling window in the next profiling window (see Section 4.7.1). The migration guideline given in TABLE 4.1 is used by SG to migrate tasks between cores to achieve a lower power consumption as well as operating temperature. In order to reduce the computational cost, we design a tool, referred to as LTR-Checker, to efficiently check whether the lifetime reliability constraint is satisfied. In each sampling window, SE either adopts the schedule generated by SG or modifies the schedule to adapt to runtime variations (see Section 4.7.2).

We highlight the effectiveness of DRIF. First of all, DRIF is adaptive to different types of “big–little” MPSoCs and different number of cores and/or pairs of cores. Meanwhile, considering that the workload in systems like infotainment may vary at runtime, DRIF periodically obtain the utilization of each core. Based on the obtained runtime status, DRIF determines the most appropriate cores to execute tasks satisfying the real-time, lifetime reliability and operating temperature constraints. In order to reduce the computational overhead, we propose heuristics to periodically migrate tasks and tune core frequencies in linear time. Note that the execution order of tasks in each core can be determined by some existing scheduling policies. DRIF is adaptive to and can work on any scheduling policy such as rate monotonic and earliest deadline first [55]. The details of our DRIF are elaborated in the next section.

4.6 LTR-Checker: A Tool to Check Lifetime Reliability Constraint

In this section, we design a tool LTR-Checker, which computational efficiently checks whether the lifetime reliability caused by a given thermal profile in a task set’s hyperperiod is larger than a pre-specified constraint, \( LTR_{th} \). Calculating lifetime reliability by using (4.4) is extremely time consuming and may not be practical to use at runtime. Hence, the target of LTR-Checker is reducing the runtime computational overhead by allowing some calculations are operated offline.
We first introduce a concept called super hyperperiod, \( sp \), which is a set of multiple adjacent hyperperiods. Let the length of this super hyperperiod be \(|sp|\), then \(|sp| = |hp| \times k\), where \( k \) is a positive integer. We further assume one hyperperiod can be divided into \( p \) time intervals, and the temperature in each interval is a constant. Since one super hyperperiod is composed of multiple adjacent hyperperiods and thermal profiles are same in each super hyperperiod, the lifetime reliability, measured by MTTF, can also be expressed as

\[
MTTF = |sp| \times \sum_{i=0}^{\infty} e^{-(i \times A^*)^\beta},
\]

where

\[
A^* = \sum_{i=1}^{k \times p} \frac{|t|}{\alpha(T_i)}.
\]

For a given thermal profile in the hyperperiod, LTR-Checker checks whether the corresponding lifetime reliability is larger than \( LTR_{th} \). LTR-Checker reduces the online computational overhead by operating the accumulation offline and only calculating \( A^* \) online.

The aim of the offline part in LTR-Checker is to find a threshold for \( A^* \), referred to \( A^*_{th} \), such that if \( A^* \leq A^*_{th} \), the corresponding MTTF is larger than \( LTR_{th} \). We first arbitrarily determine the length of super hyperperiod \(|sp|\). Since \(|hp|\) is usually in seconds and \( LTR_{th} \) is in years, setting \(|sp|\) to months can satisfy that \(|sp|\) can be evenly divided by any possible \(|hp|\). After determining the value of \(|sp|\), we can find the threshold \( A^*_{th} \) such that

\[
|sp| \times \sum_{i=0}^{\infty} e^{-(i \times A^*_{th})^\beta} = LTR_{th}.
\]

If the \( A^* \) caused by a thermal profile is smaller than \( A^*_{th} \), the corresponding system’s MTTF is larger than \( LTR_{th} \).
The online part of LTR-Checker calculates $A^*$ based on the thermal profile in a hyperperiod. With the determined $|sp|$, we first find the relationship between $A$ (in (4.5)) and $A^*$, which is described in Lemma 4.

**Lemma 4.** If one super hyperperiod is composed of $k$ hyperperiods, i.e., $|sp| = k \times |hp|$, then $A^* = A \times k$.

**Proof.** Since thermal profiles are same in each hyperperiod, each hyperperiod’s $i^{th}$ time interval has the same temperature, i.e., $T_i = T_{i+p} = \cdots = T_{i+kp}$. Furthermore, $\alpha(T_i) = \alpha(T_{i+p}) = \cdots = \alpha(T_{i+kp})$. Hence,

$$A^* = \sum_{i=1}^{k \times p} \frac{|t|}{\alpha(T_i)} = k \times \sum_{i=1}^{p} \frac{|t|}{\alpha(T_i)} = A \times k.$$  \hspace{1cm} (4.18)

Since $|sp|$ is arbitrarily determined offline and $|hp|$ is constant for a given task set, we only need to calculate $A$ in order to obtain $A^*$. $A$ can be obtained by using (4.5), and its computational overhead only depends on the value of $|hp|$, which is much smaller than $|sp|$ and $LTR_{th}$. Comparing to obtain lifetime reliability directly by using (4.4) and (4.5), the online operation of LTR-Checker is only obtaining $A$ by using (4.5). Hence, LTR-Checker dramatically reduces the online computational overhead and it can be easily used even when the computational resources are limited.

In DRIF, we require the length of the profiling window is multiple of the length of the task set’s hyperperiod, and the SG utilizes the LTR-Checker to determine whether operating temperature can guarantee the lifetime reliability constraint.

### 4.7 Design of Reliability Improvement Framework

We provide the details of our framework DRIF to improve the soft-error reliability under the temperature, real-time, and lifetime reliability constraints.
4.7.1 Schedule generator

The goal of schedule generator is to generate a schedule, i.e., each core’s workload and frequency, for the next profiling window based on the state information in the current profiling window. Though it is possible to use an optimization solver to generate an optimal schedule for the problem defined in (4.6)–(4.11), such a solver would be too time consuming for online use. Instead, we design a computational effective heuristic migrating tasks and dynamically scaling core frequencies.

As pointed out earlier, we assume that the workload has already been mapped and the workload is balanced between cores. Considering the runtime variations of workload, SG determines the frequencies of all cores to maximize soft-error reliability and meet all constraints in (4.7)–(4.11) by considering the execution models of “big–little” MPSoCs given in (4.12) or (4.13)–(4.14).

Before we present the algorithm in SG, we first introduce some concepts. System state, \( \text{St}(\text{PW}_j) \), denotes the state in the profiling window \( \text{PW}_j \), which includes the utilization, frequency, and operating temperature of each core in the sampling windows of \( \text{PW}_j \). \( \text{St}(\text{SW}_i) \), a subset of \( \text{St}(\text{PW}_j) \), represents the state in the sampling window \( \text{SW}_i \). System schedule, \( \text{Sc}(\text{PW}_j) \), specifies each core’s workload and frequency in all sampling windows in \( \text{PW}_j \). Similarly, \( \text{Sc}(\text{SW}_i) \) represents schedule in the sampling window \( \text{SW}_i \).

SG is invoked at the end of each profiling window and takes \( \text{St}(\text{PW}_j) \) and \( \text{Sc}(\text{PW}_j) \) as inputs. SG generates a schedule for Homo-Grouped MPSoCs (in Alg. 4) or for Hetero-Paired MPSoCs (in Alg. 5), respectively. We provide the details to generate a schedule for Homo-Grouped MPSoCs first. The idea is that we check whether the lifetime reliability constraint is satisfied, and try to increase core frequencies if the lifetime reliability is larger than its constraint, otherwise, reduce core frequencies (in Lines 5–19). Since all HP or all LP cores run at the same core frequency, we use \( hl \) or \( lf \) to represent cores running at high or low core frequencies, respectively. For each
Algorithm 4 Schedule Generator for Homo-Grouped MPSoCs

1: $hf$ and $lf$: the cores with high and low core frequencies, respectively
2: $l(lf, SW_i)$: frequency level of $lf$ cores at sampling window $SW_i$
3: $TP_j$: thermal profile in the $q$th profiling window
4: **procedure** `GENERATORHOMOGROUP(Sc(PWj), St(PWj))`

5: if $LTR(TP_j) < LTR_{th}$ then
6:   for each sampling window $SW_i$ do
7:     if $u(l(hf, SW_i) - 1) < u_{th}$ then
8:       $l(hf, SW_i) = l(hf, SW_i) - 1$
9:     else if $u(l(hf, SW_i) - 1) < u_{th}$ then
10:       $l(lf, SW_i) = l(lf, SW_i) - 1$
11:   end if
12: end for
13: else
14:   for each sampling window $SW_i$ do
15:     if $T(l(lf, SW_i) + 1) < T_{th}$ then
16:       $l(H, SW_i) = l(H, SW_i) + 1$
17:     end if
18:   end for
19: end if
20: for each sampling window $SW_i$ do
21:   $Sc^*(SW_i) \leftarrow$ migrate workload based on TABLE I
22: end for
23: $Sc(PW_{j+1}) \leftarrow \{Sc^*(SW_1), \ldots, Sc^*(SW_p)\}$
24: end procedure

sampling window, if the system state in the previous profiling window, $St(PW_j)$, violates the lifetime reliability constraint, SG reduces the core frequencies of cores running at high frequency if doing so does not violate the real-time constraint (in Lines 6–7). Otherwise, reduce the core frequencies of cores with low core frequency if not violate the real-time constraint (in Lines 9–10). Meanwhile, if $St(PW_j)$ meets the lifetime reliability constraint, SG increases frequencies for cores with low core frequency to improve soft-error reliability under the temperature constraint (in Lines 14–18). After determining core frequencies, SG migrates tasks between cores to reduce the power consumption and temperature (in Lines 20–22). We provide the details of task migration in Alg. 6. After determining core frequencies and migrating tasks between cores, the schedule for the next profiling window, $Sc(PW_{j+1})$, is generated (in Line 23).
Algorithm 5 Schedule Generator for Hetero-Paired MPSoCs

1: $\rho_k$: the $k^{th}$ active core
2: $l(\rho_k, SW_i)$: HP’s frequency level at sampling window $SW_i$
3: $TP_j$: thermal profile in the $q^{th}$ profiling window
4: procedure $\text{GENERATORHETEROPair}(Sc(PW_j), St(PW_j))$
5: \hspace{1em} if $\text{LTR}(TP_j) < \text{LTR}_{th}$ then
6: \hspace{2em} for each sampling window $SW_i$ do
7: \hspace{3em} Sort core with their core frequencies
8: \hspace{3em} for $\rho_k$ (starting form the core with high frequency) do
9: \hspace{4em} if $u(l(\rho_k, SW_i) - 1) < u_{th}$ then
10: \hspace{5em} $l(\rho_k, SW_i) = l(\rho_k, SW_i) - 1$
11: \hspace{5em} break
12: \hspace{3em} end if
13: \hspace{3em} end for
14: \hspace{2em} end for
15: \hspace{1em} else
16: \hspace{2em} for each sampling window $SW_i$ do
17: \hspace{3em} Sort core with their core frequencies
18: \hspace{3em} for $\rho_k$ (starting form the core with low frequency) do
19: \hspace{4em} if $T(l(\rho_k, SW_i) + 1) < T_{th}$ then
20: \hspace{5em} $l(\rho_k, SW_i) = l(\rho_k, SW_i) + 1$
21: \hspace{5em} break
22: \hspace{4em} end if
23: \hspace{3em} end for
24: \hspace{2em} end for
25: \hspace{1em} end if
26: \hspace{2em} for each sampling window $SW_i$ do
27: \hspace{3em} $Sc^*(SW_i) \leftarrow$ migrate workload based on TABLE 4.1
28: \hspace{2em} end for
29: $Sc(PW_{j+1}) \leftarrow \{Sc^*(SW_1), \ldots, Sc^*(SW_p)\}$
30: end procedure
Algorithm 6 Migrate Workload

1: \( Ty(\rho_j) \): the type of \( \rho_j \), its HP or LP
2: \( u(\rho_j, SW_i) \): \( \rho_j \)'s utilization at \( SW_i \)
3: \( u(\rho_j, W) \): \( \rho_j \)'s utilization if executing workload \( W \)
4: \( e_R^p \): the execution time of task \( \tau_k \) on core \( \rho_p \)
5: \textbf{procedure} migrate(Sc\((SW_i)\), St\((SW_i)\), TABLE 4.1)
6: \textbf{if} Homo-Grouped MPSoCs \textbf{then}
7: \textbf{for} each core (\( \rho_j \)) \textbf{do}
8: \( \tau_k \): the task on \( \rho_j \) with shortest execution time
9: \( \rho_p \): the lowest utilization core at different type of \( \rho_j \)
10: Search TABLE 4.1 with \( u(\rho_j, SW_i) \) and \( f(\rho_j, SW_i) \)
11: \( T \leftarrow \) the type of the most power efficient core
12: \textbf{while} \( Ty(\rho_j) \neq T \) \textbf{do}
13: \textbf{if} \( u(\rho_p) + \frac{e_R^p}{\Delta_k} < u_{th} \) \textbf{then}
14: \( \text{Migrate } \tau_k \text{ to core } \rho_p \)
15: \textbf{end if}
16: \( T \leftarrow \) search TABLE 4.1
17: \textbf{end while}
18: \textbf{end for}
19: \textbf{end if}
20: \textbf{if} Hetero-Paired MPSoCs \textbf{then}
21: \textbf{for} each active core \( \rho_j \) \textbf{do}
22: \( T \leftarrow \) search TABLE 4.1 with \( u(\rho_j) \) and \( f(\rho_j) \)
23: \( W \): the workload on \( \rho_j \)
24: \( \rho_p \): \( \rho_j \)'s paired core
25: \textbf{if} \( Ty(\rho_j) \neq T \) \textbf{and} \( u(W, \rho_p) < u_{th} \) \textbf{then}
26: \( \text{Migrate all workload to } \rho_p \text{ paired core} \)
27: \textbf{end if}
28: \textbf{end for}
29: \textbf{end if}
30: \textbf{for} each core \( \rho_j \) \textbf{do}
31: \textbf{if} \( \rho_j \)'s workload is empty \textbf{then}
32: \( \text{Power off } \rho_j \)
33: \textbf{end if}
34: \textbf{end for}
35: \textbf{end procedure}
SG generates a schedule for Hetero-Paired MPSoCs in Alg. 5. If the system state in the previous profiling window, $St(PW_j)$, violates the lifetime reliability constraint, SG tries to reduce the core frequency for the core which executes at the highest frequency if doing so does not violate the real-time constraint (in Lines 6–14). On the contrary, if $St(PW_j)$ satisfies the lifetime reliability constraint, SG increases the core frequency of cores with low core frequency under the temperature constraint (in Lines 16–24). Similar to Homo-Grouped MPSoCs, SG migrates tasks (in Lines 26–28) and finally generates a new schedule $Sc(PW_{j+1})$ (in Line 29). The computational complexity of Alg. 5 is $O(p \times (n+m) \times \log(n+m))$ where $p$ is the number of sampling windows in a profiling window, and $n$ and $m$ is the number of HP cores and LP cores, respectively.

We provide the details on how to migrate tasks and select power efficient cores to execute tasks are in Alg. 6. This task migration algorithm is called by Alg. 4 and Alg. 5 at each sampling window, and its inputs are the migration guideline given in TABLE 4.1 and the system state and schedule at each sampling window. The key idea is that we search the migration guideline with each core’s utilization and frequency, and migrate tasks based on the search results. For the Homo-Grouped MPSoCs, for a core, $\rho_j$, if the migration guideline indicates we should tune $\rho_j$’s utilization to save power, we migrate the task with shortest execution to an LP or HP core (in Lines 6–19). We iteratively migrate tasks between cores until the results of search migration guideline match the types of all cores. For the Hetero-Paired MPSoCs, the paired HP and LP core works exclusively. Hence, if tasks are ready optimally mapped to each pair initially, we only need to select the HP or LP core to use for each pair. If the searching results from the task migration guideline do not match the type of the core $\rho_j$, migrate all tasks on $\rho_j$ to its paired core if doing so does not violate the real-time constraint (in Lines 21–28). For both Homo-Grouped and Hetero-Paired MPSoCs, if a core’s workload is empty, power off this core to save energy (in Lines 91
30–34). For the Homo-Grouped MPSoCs, the computational complexity of Alg. 6 is \( O(\varphi \times (m + n)) \), where \( \varphi, m, n \) are the number of tasks, HP cores, and LP cores, respectively. For Hetero-Paired MPSoCs, the complexity is \( O(m + n) \).

4.7.2 Schedule executor

The schedule executor determines the active cores’ frequencies at the beginning of each sampling window. A straightforward approach is to simply follow the schedule generated by SG. However, since the schedule \( Sc(PW_{j+1}) \) is generated based on the system state \( St(PW_j) \), but the utilization in the profiling window \( PW_j \) can be different from that in the \( PW_{j+1} \), \( Sc(PW_{j+1}) \) may actually violate some or all of the constraints during run time. For soft real-time systems, it is acceptable to temporarily violate the real-time and lifetime reliability constraints in (4.9)–(4.11) as they can be compensated in the next profiling window. However, violating the temperature constraint may either cause timing faults or unexpected throttling. Therefore, SE should be designed to avoid the occurrence of such a case.

SE adjusts core frequency for each core. At the beginning of each sampling window, SE receives the initial temperatures from SC, the state collector, which is the temperature of the previous sampling window, and gets the cores’ frequencies from \( Sc(PW_{j+1}) \). We statically design a table that for all possible initial temperature and core frequencies. This table indicates the worst-case temperature in a sampling window by assuming the core utilization is 100%. SE checks whether the worst-case temperature can remain below the thermal threshold. If not, it reduces the core frequency one level lower than that specified in the schedule \( Sc(PW_{j+1}) \). Since we establish such a table statically, the computational complexity of SE is \( O(1) \).
4.8 Experimental Setup

To evaluate the proposed DRIF, we conducted experiments to compare with two existing approaches. In this section, we present the platforms, workloads, and the existing frameworks used for comparison in our experiments.

4.8.1 Comparison targets

We compared the performance of DRIF to two representative frameworks. The multi-objective optimization of system reliability (MOO) finds the Pareto optimization of soft-error reliability and lifetime reliability by using a genetic algorithm [23]. Since the genetic algorithm based solver is too costly to be used at runtime, core frequencies are determined off-line and cannot be changed on-line. In order to evaluate the benefits of migrating tasks between cores, we compare DRIF with a framework, called simplified DRIF (S-DRIF), which scales core frequencies as in DRIF, but does not migrate tasks between cores.

Three metrics are considered in the comparison. The probability of failures (PoF) due to soft errors quantifies the soft-error reliability. The PoF is defined as $1 - R$, where $R$ is the system-level soft-error reliability. An approach achieving a lower PoF is the same as achieving a higher soft-error reliability. We used the percentage of feasible solutions for real-time constraint (FS-RT) to describe the capability of satisfying real-time constraint. In experiments, the jobs of each task are periodically released. We checked which job meeting its deadline and the percentage of FS-RT is quantified as the ratio of the number of jobs meeting its deadline over the total number of all jobs. Similarly, the percentage of feasible solution for lifetime reliability constraint (FS-LTR) describes the capability of satisfying lifetime reliability. In experiments, we utilized LTR-Checker to check whether the lifetime reliability is satisfied at each profiling window. The percentage of FS-LTR is quantified as the ratio of the number of profiling windows achieving a higher lifetime reliability than the lifetime reliability
constraint over the total number of profiling windows.

4.8.2 Experimental platforms

The experiments are conducted on two boards containing Nvidia’s TK1 [76] and TX2 [78] chip, respectively. The TK1 chip provides 4 HP cores and 1 LP core, but the HP cores and the LP core cannot work simultaneously. Hence, the TK1 chip is a Hetero-Paired type MPSoC, and it only provides 1 HP–LP core pair. In our experiments, the workload for TK1 is designed to be light enough to fit on one HP or LP core for TK1. The TX2 chip includes 2 HP cores (with Nvidia’s Denver2 microarchitecture [83]) and 4 LP cores (with ARM Cortex A57 microarchitecture). Hence, TX2 chip is a Homo-Grouped type MPSoC. Note that we only consider single-thread tasks, so the Denver core has a better performance than the ARM core [83].

We obtained the chip’s operating temperature by reading their integrated thermal sensors. Note that although TK1 and TX2 only report one CPU temperature, it is enough to show that DRIF can achieve a lower temperature and guarantee the temperature constraint. For both HP and LP cores in TK1, we use the core frequencies 1.092 GHz, 0.96 GHz, 0.828 GHz, 0.696 GHz, and 0.564 GHz. For TX2, we select the core frequencies 1.881 GHz, 1.574 GHz, 1.267 GHz, 0.960 GHz, 0.652 GHz, and 0.345 GHz.

4.8.3 Workloads

We now discuss the tasks set for experiments on TK1 and TX2. Considering the low performance of cores on TK1, we chose 3 tasks from Mibench benchmark suite [25] and measured their execution times when the core’s frequency is 1.092 GHz (see TABLE 4.3). TK1 only provides one 1 HP–LP core pair, so tasks execute either on the HP core or the LP core. For experiments on TX2, we used 2 ARM cores and 1 Denver core to execute 8 tasks from Mibench [25]. We first measured the execution
times of the tasks on an ARM and Denver core with the highest core frequency (see TABLE 4.4). Based on the measurements, we mapped these tasks to ARM and Denver cores and balanced the workloads of cores (see TABLE 4.5). Note that although TX2 provides 4 ARM cores and 2 Denver cores, we only used 1 Denver core and 2 ARM cores because the workload is light. If allocating the selected tasks to 3 ARM cores and/or 2 Denver cores, the utilization of each core is such low that a core can always execute at the lowest frequency. Meanwhile, we aim at independent tasks and the soft-error reliability achieved by DRIF is related to a cores utilization but independent to the number of cores. Hence, executing tasks on 2 ARM cores and 1 Denver core is sufficient to validate the capability of DRIF to improve soft-error reliability.

We designed two task groups. In the first group, tasks are frame-based and share the same period and deadline. For experiments on TX2, tasks’ periods and deadlines are 150 ms, 200 ms, 250 ms, and 300 ms, and for experiments on TK1, they are 700 ms, 800 ms, 900 ms, and 1000 ms. In the second group, a task’s deadline and period are set to be the same but random in the ranges between 150 ms–200 ms, 200 ms–250 ms, 250 ms–300 ms for TX2, and for TK1, the ranges are 700 ms–800 ms, 800 ms–900 ms, 900 ms–1000 ms. We used the deadline-monotonic scheduling policy to schedule tasks where a task with shorter deadline is assigned a higher priority and executed earlier [55]. Also, change from tasks to jobs to be consistent. Such setups ensure that tasks are schedulable, and represent multiple workloads ranging from heavy to light.

4.9 Experimental Results

In this section, we examine the performance of the proposed DRIF compared to the S-DRIF and MOO.
### TABLE 4.3

**TASKS’ EXECUTION TIMES ON TK1**

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Execution time</th>
<th>HP ARM Core</th>
<th>LP ARM Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>qsort</td>
<td>145 ms</td>
<td>145 ms</td>
<td></td>
</tr>
<tr>
<td>blowfish</td>
<td>150 ms</td>
<td>152 ms</td>
<td></td>
</tr>
<tr>
<td>crc42</td>
<td>195 ms</td>
<td>196 ms</td>
<td></td>
</tr>
</tbody>
</table>

### TABLE 4.4

**TASKS’ EXECUTION TIMES ON TX2**

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Execution time</th>
<th>Denver Core</th>
<th>ARM Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>cjpeg</td>
<td>24 ms</td>
<td>33 ms</td>
<td></td>
</tr>
<tr>
<td>qsort</td>
<td>49 ms</td>
<td>69 ms</td>
<td></td>
</tr>
<tr>
<td>dijkstra</td>
<td>47 ms</td>
<td>64 ms</td>
<td></td>
</tr>
<tr>
<td>blowfish</td>
<td>26 ms</td>
<td>52 ms</td>
<td></td>
</tr>
<tr>
<td>susan</td>
<td>52 ms</td>
<td>78 ms</td>
<td></td>
</tr>
<tr>
<td>stringsearch</td>
<td>2 ms</td>
<td>3 ms</td>
<td></td>
</tr>
<tr>
<td>crc42</td>
<td>30 ms</td>
<td>75 ms</td>
<td></td>
</tr>
<tr>
<td>patricia</td>
<td>12 ms</td>
<td>16 ms</td>
<td></td>
</tr>
</tbody>
</table>
TABLE 4.5

THE TASK ALLOCATION FOR TX2

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Mapping to</th>
</tr>
</thead>
<tbody>
<tr>
<td>cjpeg</td>
<td>ARM Core 0</td>
</tr>
<tr>
<td>qsort</td>
<td>ARM Core 0</td>
</tr>
<tr>
<td>dijkstra</td>
<td>ARM Core 1</td>
</tr>
<tr>
<td>blowfish</td>
<td>ARM Core 1</td>
</tr>
<tr>
<td>susan</td>
<td>Denver Core 0</td>
</tr>
<tr>
<td>stringsearch</td>
<td>Denver Core 0</td>
</tr>
<tr>
<td>crc42</td>
<td>Denver Core 0</td>
</tr>
<tr>
<td>patricia</td>
<td>Denver Core 0</td>
</tr>
</tbody>
</table>

4.9.1 Experiments on TK1 chip

We first validated our approach on a TK1 chip with Hetero-Paired execution model. We compared the proposed DRIF with MOO and S-DRIF to determine whether DRIF can improve soft-error reliability without violating temperature, real-time, and lifetime reliability constraints.

Fig. 4.4 shows the experimental results when tasks are frame-based. DRIF and S-DRIF have similar performance when the workload is heavy, but DRIF achieves a lower PoF than MOO and S-DRIF in all the cases. The PoF of DRIF is 97.89%, 95.64%, 37.9%, and 18.89% of S-DRIF when the period is 700 ms, 800 ms, 900 ms, and 1000 ms, respectively. This reduced PoF guarantees the system can work without soft errors at least 2 minutes more than with S-DRIF, and up to 10 hours. Meanwhile, since our task migration considers the real-time and lifetime constraints, the percentages of FS-RT and FS-LTR of DRIF, S-DRIF and MOO are close, especially
when the workload is light. For the soft-error reliability, the PoF of DRIF is only 29.18%, 51.21%, 16.29%, and 15.04% of MOO. It means that the system can work successfully without soft errors 1.1 hours, 0.4 hour, 12.7 hours, and 100.8 hours more than MOO, respectively.

We extended the experiment to validate DRIF for a general periodic task set where tasks’ periods and deadlines are equal but randomly generated in different ranges (see Fig. 4.5). The average PoF of DRIF is 81% of S-DRIF and 51% of MOO, which translates to DRIF allowing the system to successfully work for 17 minutes more than S-DRIF on average, and 63 minutes more than MOO on average. Comparing to the results in Fig. 4.4, DRIF provides less benefits when tasks have different periods. The reason is that the workload in each sampling window varies dramatically, and DRIF guarantees the lifetime reliability constraint with a low core frequencies, which limits the performance in improving soft-error reliability. However, DRIF is still a better approach than S-DRIF and MOO, and achieves a lower PoF.

We measured the time and power consumption of DRIF on an ARM core. DRIF consumes less than 1 ms to complete and we cannot observe power changes when operating DRIF because the resolution of our power measurement tool is about 0.1 W. Based on these measurements, we claim that the time and power consumption of DRIF on TK1 can be ignored.

We also compared DRIF with a brute force search based approach which finds the optimal solution at each sampling window. This approach, although can guarantee the highest soft-error reliability at each sampling window, is computation intensive and cannot be used at the runtime. The execution time of this approach is about 30 s if running on the TK1’s HP core. Compared to this approach, the computation time of DRIF is less than 1 ms even if one profiling window has 100 sampling windows. Since both approaches determine core frequencies for each profiling window, which is typically in minutes, the exhaustive search may not be a good choice to use at
Figure 4.4. Probability of failures due to soft errors and percentage of feasible solutions for a frame-based task set running on TK1.

runtime, especially when the workload is heavy.

Although the brute force search based approach can find optimal solutions at each sampling window, it is too computational complicated to apply at runtime. On the contrary, DRIF determines the core frequencies at each profiling window by tuning the operating core frequencies one level at one time. Our experiments show that DRIF can find the best solution starting from the fifth profiling window. Since the length of a profiling window is in minutes, not finding the best solution in the first five profiling windows (less than 10 minutes) has negligible effect on lifetime reliability and soft-error reliability.
Figure 4.5. Probability of failures due to soft errors and percentage of feasible solutions for a general periodic task set running on TK1.
4.9.2 Experiments on TX2 chip

We conducted experiments on TX2 chip to evaluate the performance of DRIF in the platform with Homo-Grouped execution model. On this platform, DRIF scales core frequencies and migrates tasks to increase soft-error reliability under temperature, real-time and lifetime reliability constraints.

Similar to the experiments on TK1, we validated DRIF for i) a frame-based task set (see Fig. 4.6) and ii) a general periodic task set (see Fig. 4.7). For the frame-based task set, the PoF of DRIF is 47.25%, 81.95%, 0.1%, and 0.003% of S-DRIF when the period is 150 ms, 200 ms, 250 ms, and 300 ms, respectively. This low PoF guarantees the system can successfully work 158 hours more than S-DRIF on average and up to 24 days. Thanks to the dynamic task migration, DRIF dynamically selects the most appropriate cores to execute tasks. DRIF can also dynamically power off any
idle cores to reduce power consumption and allow active cores running at high core frequency. Hence, the benefits of DRIF are clearer than the experiments on TK1 in Fig. 4.4. Since DRIF migrates workload between cores, and the migration decision is based on current workload, it may lead more tasks missing their deadlines. However, its percentage of feasible solutions is only 1.2% smaller than S-DRIF, and also close to 100%. In most soft real-time systems, this is tolerable [61, 60]. Comparing to MOO, DRIF achieves a lower PoF in all cases, and leads to a system that can successfully work about 6.6 days more than MOO on average and up to 26 days. In terms of satisfying real-time and lifetime reliability constraints, both DRIF and S-DRIF achieve a similar percentage of FS-RT and FS-LTR to MOO especially when the workload is light.

Fig. 4.7 shows the performance of DRIF when the workload is a general periodic
task set. The PoF of DRIF is about 98%, 86%, and 0.001% of S-DRIF when periods of tasks in ranges 150 ms–200 ms, 200 ms–250 ms, and 250 ms–300 ms, respectively. It means that DRIF guarantees the system successfully work without soft errors 7.6 days more than S-DRIF on average, and up to 22.8 days. Meanwhile, the soft-error reliability improvement of DRIF over MOO is similar to that over S-DRIF. Comparing to MOO, DRIF increases the system’s successful execution time about 7.6 days on average, and up to 22.8 days. Finally, the execution time of DRIF is less than 1 ms either on the ARM core or the Denver core. The power consumption of DRIF on TX2, similar as on TK1, is also too small to be observed. In summary, the above experiments confirm that our approach DRIF has a better performance in improving soft-error reliability in all cases, and this improvement is clearer especially when the workload is light.

4.10 Summary

Focusing on two execution models of “big–little” type MPSoCs, we propose a dynamic reliability improvement framework to maximize soft-error reliability under temperature, real-time, and lifetime reliability constraints. We design a computation efficient tool to check whether the lifetime reliability caused by a thermal profile is larger than a pre-specified constraint. In order to reduce power consumption, we empirically study the power features of the high-performance and low-power cores and establish a task migration guideline to indicate the most appropriate core to execute tasks. Based on these contributions, our framework dynamically migrates tasks between cores and adjusts the core frequencies to satisfy all constraints. The experimental results on both Nvidia’s TK1 and TX2 chips show that our approach is effective in increasing soft-error reliability compared to existing approaches.
CHAPTER 5

IMPROVING RELIABILITY OF REAL-TIME SYSTEMS ON INTEGRATED
CPU AND GPU PLATFORMS

MPSO Cs consisting of integrated CPU and GPU are desirable platforms for real-
time embedded applications requiring massively parallel processing capabilities. For
many such applications, transient faults due to soft errors and permanent faults due
to wear processes are major concerns. Through detailed execution profiling, this
work reveals that a CUDA task’s execution time on CPU significantly increases if
executing such task on a different CPU core from the operating system. Based on
this observation, an extended task model is introduced to consider the dependencies
among tasks and the operating system. A hybrid framework is proposed to improve
soft-error reliability while satisfying a lifetime reliability constraint for soft real-time
systems executed on CPU and GPU.

5.1 Introduction

To help meet high performance and low power demands in many applications,
various heterogeneous MPSO Cs have been introduced [54]. One type of MPSO Cs
is composed of CPU and GPU. Thanks to the massively parallel computing ability
offered by GPU and the low power design of CPU, this type of MPSO Cs has been
widely used in many real-time embedded systems [80]. In such systems, soft-error
reliability due to transient faults and lifetime reliability due to permanent faults are
typically important design considerations.
Resource management for MPSoCs with integrated CPU and GPU has been studied [47, 86, 89]. Since the CPU and GPU share bandwidth, Jeong et al. improved the performance of GPU by dynamically adapting the priorities of CPU and GPU memory requests [47]. In order to improve the performance per watt of GPU, Pathania et al. developed a power-performance model and proposed an efficient power management strategy [86]. Prakash et al. proposed a temperature management method to improve the performance for mobile gaming applications [89]. Targeted on Nvidia's TX2 chip [78], Amert et al. deeply studied the GPU's scheduling policy [1]. All above efforts are benefit to reduce power and energy consumption, but cannot improve, even may reduce, the reliability.

Since many MPSoCs are deployed in safety critical applications and expensive as well as inconvenient to replace, lifetime reliability due to permanent faults as well as soft-error reliability due to transient faults are important design considerations. Although most existing work focuses on CPU reliability [112, 105, 13, 60], there exist several efforts that have examined soft-error reliability [101, 85, 102] or lifetime reliability [71, 91, 51] for GPU. Tan et al. developed a framework to estimate the soft-error reliability of general-purpose GPUs (GPGPU) [101] and proposed to leverage a special memory to mitigate both the soft-error reliability and energy consumption of the register files in GPGPU [102]. Based on the observation that a system may not fail just with small magnitude soft errors, Palframan et al. proposed an approach to protect the GPU execution logic from large magnitude soft errors in architecture level [85]. In order to improve soft-error reliability for GPU, Tan et al. proposed to leverage a special memory to mitigate both the soft-error reliability and energy consumption of the register files in GPGPU [102]. Namaki-Shoushtari et al. proposed to balance the wear states of register files to improve lifetime reliability for GPGPU [71]. In order to minimize the aging of GPU, Rahimi et al developed an aging-aware instruction assignment scheme to evenly distribute the stress of instruc-
tions [91]. Although this technique is benefit to improve GPU’s lifetime reliability, it suffers from a considerable amount of performance overhead. In order to reduce this overhead, Lee et al. proposed to group GPU cores based on their aging information, and evenly distribute instructions to groups [51]. All the above papers improve lifetime reliability or soft-error reliability at low levels. However, not all MPSoCs support these techniques. In addition, these papers improve GPU reliability but ignore the reliability of CPU. For a MPSoC with integrated CPU and GPU, faults from either CPU or GPU can cause the system to fail. Hence, examining the reliability of CPU and GPU together is necessary.

Our work in this chapter systematically addresses reliability concerns for real-time systems running on integrated CPU and GPU platforms. Since transient faults occur much more frequently than permanent faults [112], we aim at solving the problem of maximizing soft-error reliability under the lifetime reliability constraint. Meanwhile, in order to avoid thermal throttling, we require that the system’s operating temperature remains lower than a threshold. Since Nvidia’s chip have been used in many real-world applications [80], we are specially interested in CUDA tasks. However, the techniques can be readily applied to other GPUs and programming models. A CUDA task uses GPU resources through the driver in the operating system (OS) and may rely on some I/O services\(^1\) to complete.

In order to solve the above problem, we first explore how the mapping of CUDA tasks affects the tasks’ CPU times and then develop a hybrid framework, referred to as HyFRO (for Hybrid Framework for Reliability Optimization). This framework i) statically maps tasks to CPU cores to improve soft-error reliability, ii) dynamically migrates tasks among CPU cores to balance the wear states of the cores and achieve a higher lifetime reliability, and iii) dynamically and judiciously scales frequencies of

---

\(^1\)In this work, we refer to the operating system as the operating system kernel including hardware drivers, and refer to the I/O services as default services shipped with the operating system such as video and audio services.
CPU and GPU cores to increase soft-error reliability under temperature, real-time, and lifetime reliability constraints.

This work makes four main contributions.

• Our experiments on multiple hardware platforms reveal that for a CUDA task, its CPU time is increased if the OS and/or related I/O services are running on different CPU cores from this CUDA task. Based on this observation, we generalize a real-time task model to consider the impact of the OS and I/O services on task execution times. This model captures the dependencies between tasks, the OS, and I/O services to assist system design and analysis.

• Since a lower core utilization causes a higher soft-error reliability, we develop an off-line task mapping policy to reduce CPU times of tasks and the utilization of cores.

• By considering uneven wear states of cores and unavoidable workload and operating environment variations, we design two on-line algorithms to i) migrate tasks to balance the wear states of cores to achieve higher lifetime reliability, and ii) scale frequencies of CPU and GPU cores to further improve soft-error reliability.

We implemented and evaluated HyFRO on the Nvidia’s TX2 [78] chip. Experimental results show that HyFRO allows the CPU and GPU to execute without soft errors for about 2 days and 42 days longer on average than existing approaches, respectively.

5.2 Preliminaries

In this section, we introduce the hardware, soft-error reliability, and lifetime reliability models used in this chapter.

5.2.1 Hardware model

We focus on MPSoCs composed of one GPU ($\rho_G$) and $m$ homogeneous CPU cores, denoted by $M = \{\rho_1, ..., \rho_m\}$. We assume both the CPU and GPU support frequency scaling. Although the GPU is also composed of multiple cores, the Nvidia’s
GPU manager automatically schedules specific cores to execute our tasks. Hence, we abstract the GPU as a single processing core, which is idle only when no operations execute on it [66]. We assume the CPU core $\rho_i$ supports $L_C$ frequency levels and the core frequencies are $\{f(1), ..., f(L_C)\}$ (from lowest to highest). Similar as the CPU core, the GPU core supports $L_G$ frequency levels. We define the utilization of a core in a given time interval $\Delta t$ (whose length is $|\Delta t|$) as $u = \frac{|\Delta t_a|}{|\Delta t|}$, where $|\Delta t_a|$ is the amount of time that the core executes operations. A core’s utilization is commonly used to estimate soft-error reliability and guarantee deadline constraint. Operating temperature of both CPU and GPU cores can be estimated using an RC thermal modeling tool or measured by thermal sensors.

5.2.2 Soft-error and lifetime reliability model

We consider both soft-error reliability due to transient faults and lifetime reliability due to permanent faults. Since the CPU and GPU are identical at the device level, the device-level soft-error reliability model and lifetime reliability model are applicable to both CPU and GPU.

The soft-error reliability in a time interval $\Delta t$ is the probability that no soft errors occur [116], i.e.,

$$r = e^{-\lambda(f) \times u \times |\Delta t|}.$$  \hspace{1cm} (5.1)

$\lambda(f)$ is the average fault rate which highly depends on the core frequency ($f$). Since soft errors from either CPU and GPU fail the operating tasks, the soft-error reliability for both CPU and GPU is

$$SER = r_G \times \prod_{i=1}^{m} r_i,$$  \hspace{1cm} (5.2)

where $r_G$ and $r_i$ are the soft-error reliability of $\rho_G$ and $\rho_i$, respectively [61].

For lifetime reliability, we consider multiple wear-out effects. Wear due to electromigration, stress migration, and time-dependent dielectric breakdown is exponentially dependent on operating temperature [108]. Wear due to thermal cycling
depends on the amplitude, period, and peak temperature [108]. Hence, in order to improve lifetime reliability, both the operating temperature and thermal cycling should be considered. For the lifetime reliability, we use the tool developed in Section 4.6, LTR-Checker, to determine that for given thermal profiles of CPU and GPU, whether the corresponding lifetime reliability of CPU and GPU are larger than their constraints. Note that our work is independent of the soft-error reliability and lifetime reliability modeling tools. The proposed framework in this work can work with any modeling tools regardless in the device level or the system level.

Based on above models, in this work, we design a framework to improve soft-error reliability under temperature, real-time and lifetime reliability constraints by i) statically mapping tasks, I/O services, and the OS to appropriate CPU cores, and ii) dynamically migrating tasks between CPU cores and scaling cores’ frequencies for CPU and GPU.

5.3 Empirical Study: Effects of Mapping on Task Execution Times

In this section, we discuss one of our major contributions in this work. We focus on CUDA tasks, which are executed on both CPU and GPU and use GPU resources through the driver in the OS and relies on some I/O services to complete. The question we want to answer is whether a CUDA task’s CPU time and GPU time varies when the OS and/or related I/O services are executed on different CPU cores. We explore the answers to these questions through conducting experiments on different hardware platforms.

We first conduct experiments on Nvidia’s TK1 [76] chip (with CUDA 6.5) to observe the change of a CUDA task’s CPU time. We use default settings where the CPU frequency is 2.1 GHz and the GPU frequency is 72 MHz. To obtain general conclusions, we execute 8 CUDA tasks from multiple benchmark suites (see TABLE 5.1). We illustrate that executing a CUDA task on a different core from the OS can result
in a significant increase in the CPU time but not the GPU time. The increase in CPU
time, called additional CPU time, of each task is shown in Fig. 5.1 and the average
of additional GPU times are shown in TABLE 5.2. For all the tasks, the additional
CPU times can be significant and vary with different inputs. Although the additional
CPU times increase or decrease with different inputs, they can be predicted if tasks’
inputs are ready given. We use the worse-case additional CPU times when designing
our framework. In contrast to the additional CPU time, the additional GPU time is
negligible. The additional GPU time is less than 0.1% for all measured CUDA tasks.
This increase can be ignored in most applications.

We also consider a category of tasks which rely on I/O services to complete.
Among the 8 tasks, YOLO and ThunderStruck fall into this category. They display
results by using xorg, an I/O service in Linux’s display system. For a task in this
category, we explore whether its CPU time and GPU time increase if executing on
different cores from related services. The additional CPU times of YOLO and Thun-
derStruck are shown in Fig. 5.2. Similar to the results in Fig. 5.1, the additional
CPU time can be significant and must be considered. We also check the their addi-
tional GPU times, which are much smaller than the additional CPU times and can
be simply ignored.

To understand our observations are platform independent, we also measure the
execution times of these tasks on Nvidia’s TX2 [78] chip with CUDA 8.0. TX2 consists
of 4 ARM cores and 2 Denver cores. Both the ARM core and the Denver core support
the same instruction set architecture (ISA), but have different microarchitectures.
Since the OS must execute on the primary core (an ARM core) and a task’s execution
time is different on an ARM core and an Denver core, we only execute tasks on the
ARM cores and power off all Denver cores. We set the frequency of the ARM cores
to 2.0 GHz, and the GPU frequency is 115 MHz. The additional CPU times are
illustrated in Fig. 5.3 and Fig.5.4. Similar as the observations on TK1, executing
CUDA tasks on different ARM cores from the OS and I/O services increases the tasks’ CPU times.

In order to explore the sources of the additional CPU time, we further measure the CPU time of each CUDA API by using nvprof, a Nvidia’s profiling tool [81]. The results show that the CPU time due to calling synchronized CUDA APIs, such as cudaMemcpy, significantly increases if the CUDA task executes on a different core from the OS. Fig. 5.5 illustrates how the additional CPU time varies if a task calling the CUDA API, cudaMemcpy, different times. Generally, a task frequently calling synchronized CUDA APIs suffers a larger additional CPU time. Meanwhile, the data copy between CPU’s and GPU’s memory space is another reason to cause the additional CPU time. On TK1, we measured the additional CPU time when calling the memory copy CUDA API, cudaMemcpy, only one time but copying different size of data between CPU’s and GPU’s memory space. Fig. 5.6 illustrates how the size of the copied memory causes the additional CPU time. The percentage of additional CPU time is a constant to the size of copied memory, relatively. Above experimental results can be used as a guideline that in order to reduce the additional CPU time, a CUDA task should avoid calling synchronized CUDA APIs frequently and transferring more data in each memory copy to reduce the number of data transferring.

Based on all the above experiments, we conclude that if a CUDA task relies on the OS and/or I/O services to complete, its CPU time increases if executing on a different core from the OS and/or I/O services, but its GPU time does not change. We extend the real-time task model by considering this observation. Since soft-error reliability is reduced if the system have heavier workload, we develop a mapping policy to minimize the additional CPU times of tasks.

---

2 Although for TK1 and TX2, their CPU and GPU share the main memory, memory copy APIs, such as cudaMemcpy, still copy data between CPU’s and GPU’s memory space
Figure 5.1. The measured additional CPU times on TK1 (in milliseconds and percentage) if a task executed on different CPU cores with the operating system.
TABLE 5.1

EIGHT CUDA TASKS IN EXPERIMENTS

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>VectorAdd</td>
<td>Element by element vector addition</td>
<td>CUDA Samples [75]</td>
</tr>
<tr>
<td>SimpleTexture</td>
<td>Texture usages</td>
<td></td>
</tr>
<tr>
<td>MatrixMul</td>
<td>Matrix multiplication</td>
<td></td>
</tr>
<tr>
<td>Gaussian</td>
<td>Gaussian elimination solving equations</td>
<td>Rodinia [104]</td>
</tr>
<tr>
<td>BFS</td>
<td>Breadth first search</td>
<td></td>
</tr>
<tr>
<td>Backprop</td>
<td>Back propagation</td>
<td></td>
</tr>
<tr>
<td>YOLO</td>
<td>Real-time object detection</td>
<td>Redmon et al. [92]</td>
</tr>
<tr>
<td>ThunderStruck</td>
<td>Output tracking</td>
<td>Hare et al. [36]</td>
</tr>
</tbody>
</table>

Figure 5.2. The measured additional CPU times on TK1 (in millisecond and percentage) if a task executed on different CPU cores with the I/O service xorg.
# Table 5.2

The Measured Additional GPU Time on TK1

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Additional GPU Time</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>In Millisecond</td>
<td>In Percentage (%)</td>
<td></td>
</tr>
<tr>
<td>VectorAdd</td>
<td>0.38</td>
<td>0.01</td>
<td></td>
</tr>
<tr>
<td>SimpleTexture</td>
<td>0.09</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>MatrixMul</td>
<td>0.22</td>
<td>0.11</td>
<td></td>
</tr>
<tr>
<td>Gaussian</td>
<td>0.38</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>BFS</td>
<td>0.003</td>
<td>0.20</td>
<td></td>
</tr>
<tr>
<td>Backprop</td>
<td>0.31</td>
<td>0.68</td>
<td></td>
</tr>
<tr>
<td>YOLO</td>
<td>0.44</td>
<td>0.02</td>
<td></td>
</tr>
<tr>
<td>ThunderStruck</td>
<td>0.02</td>
<td>0.01</td>
<td></td>
</tr>
</tbody>
</table>
Figure 5.3. The measured additional CPU times on TX2 (in milliseconds and percentage) if a task executed on different ARM cores with the operating system.
Figure 5.4. The measured additional CPU times on TX2 (in millisecond and percentage) if a task executed on different CPU cores with the I/O service xorg.

Figure 5.5. The measured additional CPU times on TK1 (in milliseconds and percentage) if an CUDA task calling the CUDA API, cudaMemcpy, different times.
5.4 Integrated Task, OS, and I/O Services Model

We extend the real-time task model to account for the fact that a task depends on the OS and I/O services to complete. In order to describe the dependencies, one method is to simply treat the OS and I/O services as real-time tasks and use existing models such as synchronous data flow graphs [50]. However, this method may not be correct since it ignores some special features of the OS and I/O services. Specifically, different from real-time tasks, the OS and I/O services can start any time and respond to multiple requests from other tasks simultaneously. Also, there is usually only one process for each I/O service, and so the service cannot be described by multiple tasks. Finally, the OS and I/O services are not periodic and do not have deadline requirement. To deal with these features, we extend the real-time task model by adding necessary dependencies between real-time tasks, and the OS and I/O services.

The real-time tasks considered here are independent and periodic and have soft deadlines. Task $\tau_i$ is associated with a tuple $\{e_i, d_i, A_i\}$ where $e_i$ is the execution time (i.e., the elapsed time from $\tau_i$’s start time to finish time when $\tau_i$ is the only task being

---

Figure 5.6. The measured additional CPU times on TK1 (in milliseconds and percentage) if copying different size of data between CPU’s and GPU’s memory space.
executed), \( d_i \) is the deadline, and \( A_i \) captures the dependencies of \( \tau_i \) on the OS and I/O services. We assume a task that missing its deadline is immediately terminated.

For a CUDA task, its execution time consists of two parts: the execution time on CPU, \( e_i^C \), and the execution time on GPU, \( e_i^G \). Since a CUDA task may call multiple synchronized and asynchronized CUDA APIs, the relationship between \( e_i^C \), \( e_i^G \), and \( e_i \) is generally complicated. However, for CUDA tasks executing on Nvidia’s TK1 or TX2, CPU busy waits when GPU operates [1, 49]. Hence, it is safe to assume that \( e_i^C = e_i \) and a longer (shorter) \( e_i^G \) leads to a longer (shorter) \( e_i \).

For \( \tau_i \), we use \( A_i \) to describe its dependencies on the OS and I/O services. \( A_i \) is a hash table where the key is the name of an I/O service or the OS, and the value is the additional CPU time. This table can be built off-line through profiling. For example, for the task YOLO described in Sec. 5.3, we have \( A = \{ \text{OS}: 0.7, \text{xorg}: 0.5 \} \) representing that YOLO depends the OS xorg, an I/O service, to complete. The additional CPU times are 0.7 ms and 0.5 ms if executing YOLO on different cores with the OS and xorg, respectively. For each dependency to the OS or an I/O service, although the additional CPU time varies with different inputs, we choose the worst-case additional CPU time to guarantee the real-time constraint.

We can use an undirected graph to describe how tasks depend the OS and I/O
services to complete. Tasks, the OS, and I/O services are represented by nodes, and edges represent their dependencies. The value of each edge is the addition CPU time if the task and the OS or I/O services executing on different cores. For example, Fig. 5.7 illustrates that the tasks YOLO and ThunderStruck rely on both xorg and the OS to complete, but vectorAdd and BFS only depend on the OS. For the task BitCount, which is from the MiBench Benchmark Suite [25] and only uses CPU resources, it is independent to any I/O services and the OS. This graph shows each task’s dependencies and the I/O service that most tasks rely on. We use it when develop a mapping method to minimize additional CPU times of tasks and reduce the workload of the task set.

5.5 Problem Formulation and Framework Overview

In this section, we first formulate the problem addressed in this work and then describe our solution HyFRO at high level.

5.5.1 Problem formulation

We formulate our problem motivated by applications such as mobile and in-vehicle infotainment systems. These systems have soft real-time, temperature, lifetime reliability, and soft-error reliability requirements [79]. Before formulating the problem, we first introduce two concepts: sampling window (SW) and profiling window (PW). Similar as in Chapter 3, a sampling window is defined as a time interval in which the temperature can be treated as constant, and task migration is not allowed inside a sampling window [61]. Based on the concept of sampling window, a profiling window is composed of $n$ equal-length sampling windows and used to estimate lifetime reliability.
Specifically, the problem we want to solve is the following

$$\max \left\{ \prod_{j=1}^{n} (r_{G,j} \times \prod_{i=1}^{m} r_{i,j}) \right\}. \quad (5.3)$$

The solution to (5.3) must satisfy the following constraints:

$$\max \left\{ T(\rho_{i}, SW_{j}) \right\} \leq T_{th}^{C}, \max \left\{ T(\rho_{G}, SW_{j}) \right\} \leq T_{th}^{G}, \quad (5.4)$$

$$u(\rho_{i}, SW_{j}) \leq u_{th}, \quad (5.5)$$

$$\min \left\{ LTR(\rho_{i}), LTR(\rho_{G}) \right\} > LTR_{th}. \quad (5.6)$$

$r_{G,j}$ and $r_{i,j}$ are the soft-error reliability of $\rho_{G}$ and $\rho_{i}$ in the $j^{th}$ sampling window, respectively. For any CPU core $\rho_{i}$ and the GPU core $\rho_{G}$, constraints in (5.4)–(5.6) must be satisfied. The first constraint requires the temperature of CPU cores and the GPU core be less than some thresholds $T_{th}^{C}$ and $T_{th}^{G}$ in any sampling window. For soft real-time systems, temporarily violating the real-time and lifetime reliability constraints is acceptable, but the temperature constraint must be satisfied to avoid thermal throttling. The second constraint captures the real-time requirement, where $u(\rho_{i}, W_{j})$ is the utilization of the CPU core and $u_{th}$ is the upper bound on utilization to satisfy schedulability. The last constraint requires the MTTF resulting from the run-time temperature be larger than a threshold $LTR_{th}$. In our work, the CPU and GPU are embedded on a single chip and permanent faults from either the CPU or GPU can cause the system to fail. Hence, we require that the minimum lifetime reliability of CPU and GPU cores be larger than the lifetime reliability threshold.

The parameters in (5.3)–(5.6) are cores’ frequencies.

Since the soft-error reliability is related to the core’s utilization, we first design an off-line heuristic to map tasks, the OS, and I/O services to reduce the total utilization of cores. Then, we dynamically migrate tasks among CPU cores to achieve a higher
lifetime reliability and scale frequencies for CPU and GPU cores in each sampling window to improve soft-error reliability. We integrate these two work into a hybrid framework to solve this formulated problem efficiently.

5.5.2 Framework overview

To better respond to the facts that i) mapping of tasks affects the tasks’ execution times, and ii) workload and environmental changes that are unavoidable in real-time embedded systems, we aim to develop a hybrid off-line/on-line framework, HyFRO, to solve the problem defined in (5.3)–(5.6). Since a higher core utilization causes a lower soft-error reliability, HyFRO maps tasks, I/O services, and the OS to appropriate CPU cores to minimize the additional CPU time and reduce the total utilization of cores. Although this mapping can improve soft-error reliability, it weakens lifetime reliability for uneven core wear states. Hence, the on-line part in HyFRO migrates tasks to balance the core wear states. Responding to the workload and operating environment variations, HyFRO improves soft-error reliability by scaling frequencies of CPU and GPU cores dynamically. Fig. 5.8 depicts the major components in HyFRO. This framework is designed based on the consideration that i) the impact of task mapping on task execution times, ii) unbalanced wear states of cores, and iii) unavoidable workload and runtime environment variations.

The off-line component in HyFRO maps tasks, I/O services, and the OS to appropriate CPU cores (the left part of Fig. 5.8). A common mapping method is to achieve spatial and temporal load balancing, which tends to even off core wear states and effectively improves the system-level lifetime reliability [13]. However, this method weakens the soft-error reliability since it introduces additional CPU times and increases the total utilization of cores. Based on the soft-error reliability model in (5.1), soft-error reliability is improved with shorter task execution times. We develop a task mapping policy allowing a task, the OS, and its related I/O services
to execute on the same core. This mapping minimizes additional CPU times and reduces task execution times, which improves soft-error reliability but weakens lifetime reliability. We dynamically improve lifetime reliability by migrating task and balancing the wear states of CPU cores in the on-line component in HyFRO.

The on-line component in HyFRO consists of four main parts (the right part of Fig. 5.8). The state collector (SC) collects and saves the system states in each sampling window. The system states include each core’s temperature, utilization, and frequency. In order to compensate for the impact of the off-line task mapping on lifetime reliability, the workload balancer (WB) swaps the workload among CPU cores at each profiling window to improve lifetime reliability by balancing the wear states of cores.

To address unavoidable workload and environment variations, the schedule generator (SG) and schedule executor (SE) work together to dynamically improve soft-error reliability by increasing CPU and GPU cores’ frequencies. This on-line component works as follows. At the beginning of each profiling window, the schedule generator determines cores’ frequencies for both CPU and GPU for each sampling window in

Figure 5.8. High-level overview of HyFRO.
the next profiling window. This decision, called a schedule, is based on the past system states. However, since the workload of the task set can change at run time, the schedule generated by schedule generator may not meet all the constraints in (5.4)–(5.6). Although a slight violation to real-time and/or lifetime reliability constraints is acceptable, the temperature constraint must be satisfied to avoid thermal throttling. Hence, in each sampling window, the schedule executor may modify the generated schedule to adapt to run-time variations while always meeting the temperature constraint. Since the target of HyFRO’s schedule executor is same as the schedule executor DRIF (in Sec. 4.7.2), these two schedule executors have the exactly same algorithm. We elaborate on the details of HyFRO in the next section. Note that although we design HyFRO based on CUDA tasks, this framework also works for regular tasks that do not rely on the OS and any I/O services.

5.6 Framework Details

We provide the details of our framework to improve the soft-error reliability under the temperature, real-time, and lifetime reliability constraints.

5.6.1 Task mapper

The task mapper maps tasks, I/O services, and the OS to appropriate CPU cores. Although existing methods balance the workload of cores to increase lifetime reliability, they reduce soft-error reliability by introducing additional CPU times and increasing the total utilization of cores [13]. Hence, it is necessary to design a new mapping method to minimize additional CPU times. We can formulate this mapping problem as an integer linear program (ILP) problem, and find the optimal mapping using existing methods [12, 105]. However, the ILP problem under consideration is NP-complete, and its complexity is too high even for off-line mapping if the number of tasks and the number of cores are large [12].
In order to reduce the computational complexity, we design an efficient method to find a good mapping in terms of minimizing the additional CPU time (see Alg. 7). Let $\delta_{i,j}$ represent the additional CPU time if the I/O service $s_i$ and task $\tau_j$ are in different cores. Note that in this algorithm, we treat the OS and I/O services are same and use $s_{os}$ representing the OS. We iteratively find the largest additional CPU time (supposing it is $\delta_{i,j}$), and map $s_i$ and $\tau_j$ to appropriate cores (Lines 7–26). If both $s_i$ and $\tau_j$ have not been mapped, we attempt to map them to a single core if doing so would not violate the real-time requirement. Otherwise, map $s_i$ and $\tau_j$ to two cores with lowest utilizations (Lines 9–16). If $s_i$ ($\tau_j$) is already mapped, we map $\tau_j$ ($s_i$) to the same core as $s_i$ ($\tau_j$) if allowed, and map $\tau_j$ ($s_i$) to a core with lowest utilization otherwise (Lines 17–24). Since this heuristic checks every $\delta$, its complexity is linear in terms of the number of dependencies among tasks, I/O services, and the OS.

### 5.6.2 Workload balancer

In order to maximize soft-error reliability, the aim of the task mapper in HyFRO is to avoid the additional CPU times and reduce the total utilization of cores. This mapping may lead to unbalanced workloads on cores and thus uneven core wear states. Generally, a heavier workload increases a core’s temperature, resulting in more wear. Hence, to achieve a higher lifetime reliability, it is necessary to migrate tasks and balance the wear states of cores at run time. In order to not weaken soft-error reliability, we design a heuristic to balance the core wear states by swapping the workload between cores.

The basic idea of the workload balancer is to swap the entire workload on a core having high temperature with that on another core having low temperature (see Alg. 8). Swapping workload is efficient to balance the wear states, but it may cause thermal cycling which weakens lifetime reliability since it reduces the temperature of
the core wearing fast and increases the temperature of the core wearing slow. Hence, we only swap the workload between cores if the difference in the cores’ temperatures is smaller than a pre-specified threshold, $T_{cyc}$ (Lines 3–10). $T_{cyc}$ is used to ensure that the thermal cycling will not become the dominant reason for permanent faults [13]. We can determine the value of $T_{cyc}$ offline using an existing lifetime reliability modeling tool [108]. Since the workload balancer treats the tasks on a core as one entity and does not separate tasks, related I/O services, and/or the OS to different cores, it would not weaken soft-error reliability. The complexity of Alg. 8 is $O(m)$ where $m$ is the number of CPU cores.

---

**Algorithm 7** Task mapping

1: $u(s_i)$: utilization of the service $s_i$, which can be measured offline
2: $u(p_i)$: utilization of the core $p_i$
3: $\delta_{i,j}$: the additional CPU time if $s_i$ and $\tau_j$ on different cores
4: $\Delta$: \{ $\delta_{i,j}$, for all dependencies among tasks and services \}
5: **procedure** MAPPING
6: \hspace{1em} Set $u(p_i) = 0$ for all cores
7: \hspace{1em} while $\Delta$ is not empty do
8: \hspace{2em} Assume $\delta_{i,j}$ is the largest additional CPU time
9: \hspace{2em} if Both $s_i$ and $\tau_j$ have not been mapped then
10: \hspace{3em} $p_l$: the core with lowest utilization
11: \hspace{3em} if $u(p_l) + u(s_i) + \frac{\delta_{i,j}}{\theta_j} \leq u_{th}$ then
12: \hspace{4em} Map $s_i$ and $\tau_j$ to $p_l$
13: \hspace{2em} else
14: \hspace{3em} Map $s_i$, $\tau_j$ to two cores with lowest utilizations
15: \hspace{2em} end if
16: \hspace{2em} end if
17: \hspace{2em} if $\tau_j$ (or $s_i$) is already mapped then
18: \hspace{3em} $p_l$: the core $\tau_j$ (or $s_i$) executes on
19: \hspace{3em} if $u(p_l) + u(s_i) \leq u_{th}$ (or $u(p_l) + \frac{\delta_{i,j}}{\theta_j} \leq u_{th}$) then
20: \hspace{4em} Map $s_i$ (or $\tau_j$) to $p_l$
21: \hspace{3em} else
22: \hspace{4em} Map $s_i$ (or $\tau_j$) on the core with lowest utilization
23: \hspace{3em} end if
24: \hspace{2em} end if
25: \hspace{2em} Remove $\delta_{i,j}$ from $\Delta$
26: \hspace{2em} end while
27: **end procedure**
Algorithm 8 Balance workload among CPU cores

1: procedure BALANCER
2: disorder cores with temperature: $T(\rho_i) > T(\rho_{i+1})$
3: for $i = \{1, 2, 3, \ldots, m\}$ do
4:   for $j = i + 1, i + 2, \ldots, m$ do
5:     if $T(\rho_i) - T(\rho_j) < T_{\text{cyc}}$ and not swapping $\rho_j$ then
6:       Swap the workload between $\rho_i$ and $\rho_j$
7:       break
8:     end if
9:   end for
10: end for
11: end procedure

5.6.3 Schedule generator

In order to improve soft-error reliability, the scheduler generator is triggered at the beginning of each profiling window to generate a frequency schedule for the next profiling window based on the past system states. The schedule for the $(q + 1)^{th}$ profiling window, $S_{c}(PW_{q+1})$, is generated by tuning the schedule $S_{c}(PW_{q})$, along with the system states $S_{t}(PW_{q})$. The initial schedule used in first profiling window sets the lowest frequencies of both CPU and GPU cores to guarantee the temperature constraint.
Algorithm 9 Determine core frequencies in the scheduler generator

1: \( SW_{p,q} \): the \( p \)th sampling window in the \( q \)th profiling window
2: \( l(\rho_i, SW_{p,q}) \): core frequency level of \( \rho_i \) at \( SW_{p,q} \)
3: \( f(\rho_i, SW_{p,q}, l) \): core frequency of \( \rho_i \) with frequency level \( l \)
4: \( u(\rho_i, SW_{p,q}, l) \): utilization of \( \rho_i \) at \( SW_{p,q} \)
5: \( T(\rho_i, SW_{p,q}) \): operating temperature of \( \rho_i \) at \( SW_{p,q} \)
6: procedure \( \text{Generator}(Sc(PW_q), St(PW_q)) \)
7: if \( \text{LTR}(GPU) < \text{LTR}_{\text{th}}^G \) then
8: Sort W: \( T(\rho_G, SW_{p,q}, l) > T(\rho_G, SW_{p+1,q}, l) \)
9: for \( i = \{1, 2, 3, \ldots, n\} \) do
10: if \( u_{i,q} + \frac{u(\rho_G, SW_{i,q}, l_{\text{max}}) \times f(\rho_G, SW_{i,q}, l)}{f(\rho_G, SW_{i,q}, l-1)} \leq u_{\text{th}} \) then
11: \( l(\rho_G, SW_{i,q+1}) = l(\rho_G, SW_{i,q}) - 1 \)
12: Break
13: end if
14: end for
15: else
16: while \( T(\rho_G, SW_{p,q}, l + 1) \leq T_{\text{th}}^G \) do
17: \( l(\rho_i, SW_{p,q+1}) = l(\rho_i, SW_{p,q}) + 1 \)
18: end while
19: end if
20: if \( \text{LTR}(CPU) < \text{LTR}_{\text{th}}^C \) then
21: \( SW_{h,q} \): the CPU has highest average temperature at \( SW_{h,q} \)
22: for \( i = \{1, 2, 3, \ldots, m\} \) do
23: if \( u(\rho_i, SW_{h,q}, l) \times f(\rho_i, SW_{h,q}, l) \leq u_{\text{th}} \) then
24: \( l(\rho_i, SW_{h,q+1}) = l(\rho_i, SW_{h,q}) - 1 \)
25: end if
26: end for
27: else
28: \( SW_{l,q} \): the CPU has lowest average temperature at \( SW_{l,q} \)
29: for \( i = \{1, 2, 3, \ldots, m\} \) do
30: if \( T(\rho_i, SW_{l,q}, l + 1) \leq T_{\text{th}}^C \) then
31: \( l(\rho_i, SW_{l,q+1}) = l(\rho_i, SW_{l,q}) + 1 \)
32: end if
33: end for
34: end if
35: Return the schedule for the next profiling window
36: end procedure
The main procedure of scheduler generator is given in Alg. 9. We introduce symbols in Lines 1–5 and then start our heuristic in Line 6. This heuristic is triggered at the end of each profiling window. The thermal profiles of CPU and GPU, which is used to estimate lifetime reliability, are generated from the operating temperature of all cores at each sampling window. If the lifetime reliability of GPU is lower than the lifetime reliability constraint, we find a sampling window where the GPU has the highest temperature, and reduce the core frequency at that sampling window if doing so does not violate the utilization constraint (Lines 7–15). If the current schedule satisfies the lifetime reliability constraint for GPU, we increase the frequency of GPU in each sampling window to maximize soft-error reliability if doing so does not violate the temperature constraint (Lines 16–18). We adjust the CPU core frequencies in a similar manner. The core frequencies at the sampling window with highest (lowest) temperature are reduced (increased) if the previous schedule cannot (can) guarantee the lifetime reliability constraint (Lines 20–34). After tuning the schedule in the previous profiling window, a new schedule for the next profiling window is generated and determine the frequencies of all CPU and GPU cores (Line 35). The complexity of Alg. 9 is $O(m \times n + n)$ where $m$ is the number of CPU cores and $n$ is number of sampling windows in a profiling window.

5.7 Evaluation

We evaluate our framework HyFRO by conducting multiple experiments and comparing it with two existing approaches.

5.7.1 Experimental setup

We conducted multiple experiments to evaluate the effectiveness of HyFRO in improving soft-error reliability under temperature, real-time, and lifetime reliability constraints. For the hardware platform, we select Nvidia’s TX2 board [78] targeted
at edge computing such as robotics and medical devices. For the CPU, TX2 contains two Denver [83] cores and four ARM Cortex-A57 cores. We implemented HyFRO (in the user space) and deployed it on one Denver core, and tasks, the OS, and I/O services are executed on ARM cores. All these cores can be active simultaneously, but all ARM (Denver) cores must work at the same voltage and frequency. Except for the primary ARM core, core 0, all cores can be powered on and off dynamically. In the experiments, we only use the four ARM cores to execute tasks. TX2 also contains a Pascal architecture based GPU which is highly power efficient and fully supports all modern graphics APIs. As a low-power chip, the CPU supports multiple frequencies from 0.35 GHz to 2.04 GHz, and the GPU can work from 0.11 GHz to 1.30 GHz. Thermal sensors are deployed to sample temperatures of the CPU, GPU, and other components. Since the default interface only provides one CPU temperature for all CPU cores, we assume CPU cores have the same temperature. TX2 is shipped with an OS based on Ubuntu and is capable of executing some widely used benchmarks. The kernel of OS must execute on the primary ARM core but the I/O services like xorg are allowed to execute on different cores.

We selected 6 tasks from multiple benchmark suites: MiBench [25], CUDA samples [75], and Rodinia [104] (see TABLE 5.3). Tasks VectorAdd, MatrixMul, Backprop, and Gaussian are independent of any I/O services but use GPU computational resources through the OS and have long execution times. CRC and Dijkstra only use CPU computational resources to complete. Their execution times are short and independent of mapping. We measured the CPU and GPU times of these tasks on TX2’s ARM core at 2.04 GHz and set the GPU at 1.30 GHz. Based on the execution time, we map tasks and the OS to appropriate CPU cores by using our mapping method in Alg. 7. Considering different real-time requirements, we designed two groups of task setups. In the first group, tasks are frame-based and share the same period and deadline. We evaluated HyFRO when the deadline is 0.50 s, 0.75 s, 1.00 s, and
TABLE 5.3

CPU, GPU, AND ADDITIONAL CPU TIMES OF TASKS ON TX2

<table>
<thead>
<tr>
<th>Task</th>
<th>GPU Time</th>
<th>CPU Time</th>
<th>Additional CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>VectorAdd [75]</td>
<td>3.9 ms</td>
<td>356 ms</td>
<td>4 ms</td>
</tr>
<tr>
<td>MatrixMul [75]</td>
<td>23.0 ms</td>
<td>120 ms</td>
<td>6 ms</td>
</tr>
<tr>
<td>Gaussian [104]</td>
<td>16.6 ms</td>
<td>196 ms</td>
<td>12 ms</td>
</tr>
<tr>
<td>Backprop [104]</td>
<td>6.1 ms</td>
<td>125 ms</td>
<td>7 ms</td>
</tr>
<tr>
<td>CRC [25]</td>
<td>0 ms</td>
<td>65 ms</td>
<td>0 ms</td>
</tr>
<tr>
<td>Dijkstra [25]</td>
<td>0 ms</td>
<td>60 ms</td>
<td>0 ms</td>
</tr>
</tbody>
</table>

1.25 s. In the second group, a task’s deadline and period are random in the ranges 0.50 s–0.75 s, 0.75 s–1.00 s, and 1.00 s–1.25 s.

We compared HyFRO with two representative approach as: the dynamic reliability improvement framework (DRI) [61] and multi-objective optimization of reliability (MOO) [23]. Similar to HyFRO, DRI dynamically scales core frequencies to improve soft-error reliability under temperature, real-time, and lifetime reliability constraints. However, it assumes tasks are mapped to balance the workload of cores and task migration is not allowed at run time. Since both DRI and MOO ignore the reliability requirements for GPU, they use the default setting when system starts where the GPU’s frequency is 115 MHz. For both HyFRO and DRI, the upper bound on utilization, $u_{th}$, is set to 80%, and the temperature threshold is 70°C, to avoid thermal throttling [60]. Based on the worst-case run-time environment, MOO maximizes the minimum of soft-error reliability and lifetime reliability by statically determining the frequencies of CPU cores [23]. Lifetime reliability and soft-error reliability are measured by using existing tools [43, 60, 61]. Note that the improvement of HyFRO is
independent on the methods to obtain lifetime reliability and soft-error reliability. Since we aim to improve soft-error reliability under lifetime reliability constraint, the probability of failure due to soft errors for CPU (and GPU), $PoF_C = \prod_{i=1}^{m}(1 - r_i)$ (and $PoF_G = 1 - r_G$), is used as a metric for comparison.

### 5.7.2 Experimental results

HyFRO maps tasks, the OS, and I/O services to appropriate cores and reduce the total utilization of cores. It also dynamically balances the core wear states to improve lifetime reliability and increases the frequencies of CPU and GPU to improve soft-error reliability. Both the on-line and off-line parts are effective in increasing soft-error reliability under temperature, real-time, and lifetime reliability constraints.

We summarize the experimental results in Fig. 5.9. As can be readily seen, HyFRO achieves a lower probability of failure than DRI regardless of the tasks under consideration. HyFRO reduces the $PoF_C$ by 28% on average to DRI. This improvement in soft-error reliability leads to the CPU that can execute without errors for about 53 hours longer on average than DRI. Although the performance of our framework is limited by the hardware limitation that all ARM cores must execute at the same frequency, the improvement of HyFRO is still meaningful. Especially when the period is long, HyFRO allows more tasks to execute on the same core with the OS, which reduces the total utilization of cores and achieves a large improvement over DRI. Thanks to the on-line component, HyFRO improves soft-error reliability of CPU significantly compared to MOO, an off-line approach. HyFRO reduces the probability of failure by 27% on average for different periods, which leads to the CPU that can successfully execute for about 2 days longer on average than MOO. Since both DRI and MOO ignore the reliability of GPU and use the default low frequency for GPU, HyFRO reduces the $PoF_G$ about $2 \times 10^5$ times on average comparing to both DRI and MOO for different tasks’ periods. This improvement of soft-error re-
Figure 5.9. The probability of failures for (a) the CPU and (b) the GPU.

liability leads to the GPU that can execute without errors for about 42 days longer on average than DRI.

We also compare the real-time performance of these approaches and the results show HyFRO results in 5% more tasks meeting their deadlines than both DRI and MOO while achieving the higher soft-error reliability as discussed above. In terms of the overhead of HyFRO, its execution time is less than 10 ms, which is much shorter than the typical length of the sampling window (about 1 s) and acceptable for most applications.
5.8 Summary

In this work, we aim to improve the soft-error reliability of real-time systems running on integrated CPU and GPU platforms. For CUDA tasks, we observe that their execution times on CPU may vary with different mappings. Based on this observation, we first extend the real-time task model by considering the dependencies between tasks, the operating system, and I/O services. We then propose a hybrid reliability improvement framework to improve soft-error reliability under temperature, real-time, and lifetime reliability constraints. For the off-line component in this framework, a mapping policy is developed to reduce the total utilization of cores and improve soft-error reliability. The on-line component dynamically migrates tasks between CPU cores to achieve a higher lifetime reliability and adjusts the frequencies of CPU and GPU cores to improve soft-error reliability. Experimental results show that our approach is effective in increasing soft-error reliability without violating temperature, real-time, and lifetime reliability constraints.
CHAPTER 6

CONCLUSIONS

Improving reliability of real-time embedded systems is an important problem that has received significant research attention in the past several years. In this thesis, we systematically discuss the lifetime reliability and soft-error reliability requirements. Motivated by different applications, we propose multiple methods and frameworks for MPSoCs with different architectures to i) improve their lifetime reliability under real-time constraint and ii) improve their soft-error reliability under lifetime reliability and real-time constraints.

To improve lifetime reliability, we proposed a framework to dynamically reduce core frequencies and maintain the operating temperature at a low level in Chapter 2. To improve soft-error reliability, a framework is developed to schedule tasks and guarantee more tasks can be recovered in Chapter 3. Chapter 4 and Chapter 5 focus on the “big–little” type MPSoCs and the MPSoCs with integrated CPU and GPU, respectively. We have analyzed the power features of the “big” core and the “little” core in Chapter 4, and an on-line framework is developed to maximize soft-error reliability. It selects the most power efficient cores to execute tasks and dynamically scales core frequencies. In Chapter 5, we focused on CUDA tasks and analyzed the impacts of task mapping to tasks’ execution times. A framework is proposed to improve soft-error reliability by mapping CUDA tasks and migrating them between cores. All these works have been validated on multiple hardware platforms, and the results illustrate their effectiveness in improving reliability.

134


