

3-21-2014


Real-Time Scheduling of Embedded Applications on Multi-Core Platforms

Ming Fan

Florida International University, mfan001@fiu.edu

DOI: 10.25148/etd.FI14040815

Follow this and additional works at: <https://digitalcommons.fiu.edu/etd>

 Part of the [Computer and Systems Architecture Commons](#), [Power and Energy Commons](#), [Systems Architecture Commons](#), and the [Theory and Algorithms Commons](#)

Recommended Citation

Fan, Ming, "Real-Time Scheduling of Embedded Applications on Multi-Core Platforms" (2014). *FIU Electronic Theses and Dissertations*. 1243.

<https://digitalcommons.fiu.edu/etd/1243>

This work is brought to you for free and open access by the University Graduate School at FIU Digital Commons. It has been accepted for inclusion in FIU Electronic Theses and Dissertations by an authorized administrator of FIU Digital Commons. For more information, please contact dcc@fiu.edu.

FLORIDA INTERNATIONAL UNIVERSITY
Miami, Florida

REAL-TIME SCHEDULING OF EMBEDDED APPLICATIONS ON
MULTI-CORE PLATFORMS

A dissertation submitted in partial fulfillment of the
requirements for the degree of
DOCTOR OF PHILOSOPHY
in
ELECTRICAL ENGINEERING
by
Ming Fan

2014

To: Dean Amir Mirmiran
College of Engineering and Computing

This dissertation, written by Ming Fan, and entitled Real-Time Scheduling of Embedded Applications on Multi-Core Platforms, having been approved in respect to style and intellectual content, is referred to you for judgment.

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

Malek Adjouadi

Jean H. Andrian

Nezih Pala

Deng Pan

Gang Quan, Major Professor

Date of Defense: March 21, 2014

The dissertation of Ming Fan is approved.

Dean Amir Mirmiran
College of Engineering and Computing

Dean Lakshmi N. Reddi
University Graduate School

Florida International University, 2014

© Copyright 2014 by Ming Fan

All rights reserved.

DEDICATION

I would like to dedicate this Doctoral dissertation to my beloved wife, Rong Rong, and my dearest parents. Without their love, understanding, support, and encouragement, the completion of this endeavor would never have been possible.

ACKNOWLEDGMENTS

First, I would like to express my deepest appreciation to my major advisor, Dr. Gang Quan, for his constant guidance and endless encouragement during the last five years of my doctoral study. I truly admire his dedication to science and research.

I would also like to express my gratitude to my Ph.D. committee members, Dr. Jean H. Andrian, Dr. Malek Adjouadi, Dr. Nezih Pala and Dr. Deng Pan, for their helpful insights, comments and suggestions in improving the quality of this dissertation. I am extremely proud to have such a wonderful and knowledgeable people serving on my dissertation committee.

I am thankful to the staff of ECE department at FIU, specially to Mrs. Pat Brammer, Mrs. Maria Benincasa and Mrs. Ana Saenz for their great commitment to student services.

Next, I would like to thank my lab mates, Mr. Shuo Liu, Mr. Qiushi Han, Mr. Tianyi Wang, Mr. Shi Sha, Dr. Vivek Chaturvedi, Dr. Huang Huang and Dr. Guanglei Liu, for creating a wonderfully collaborative work environment.

Further, I want to thank my family for their unlimited love, faith, encouragement, blessings and prayers. I am very grateful to my beloved wife, Mrs. Rong Rong, for accompanying and encouraging me during my entire Ph.D life. I want to give my life-long gratitude to my dearest mother, Mrs. Zhenhuan Dang, and father, Mr. Wei Fan, for all the love and affection they have showered upon their children. I want to thank my sister, Mrs. Xing Fan, for being a great wall of support and inspiration in my life. I am thankful to my mother-in-law, Mrs. Xiaochun Wang, and farther-in-law, Mr. Delun Rong, for their care and encouragement.

Finally, and above all, I would like to thank the National Science Foundation (NSF) for supporting the research described in this dissertation through grants CNS-0969013, CNS-0917021 and CNS-1018108.

ABSTRACT OF THE DISSERTATION
REAL-TIME SCHEDULING OF EMBEDDED APPLICATIONS ON
MULTI-CORE PLATFORMS

by

Ming Fan

Florida International University, 2014

Miami, Florida

Professor Gang Quan, Major Professor

For the past several decades, we have experienced the tremendous growth, in both scale and scope, of real-time embedded systems, thanks largely to the advances in IC technology. However, the traditional approach to get performance boost by increasing CPU frequency has been a way of past. Researchers from both industry and academia are turning their focus to multi-core architectures for continuous improvement of computing performance. In our research, we seek to develop efficient scheduling algorithms and analysis methods in the design of real-time embedded systems on multi-core platforms. Real-time systems are the ones with the response time as critical as the logical correctness of computational results. In addition, a variety of stringent constraints such as power/energy consumption, peak temperature and reliability are also imposed to these systems. Therefore, real-time scheduling plays a critical role in design of such computing systems at the system level.

We started our research by addressing timing constraints for real-time applications on multi-core platforms, and developed both partitioned and semi-partitioned scheduling algorithms to schedule fixed priority, periodic, and hard real-time tasks on multi-core platforms. Then we extended our research by taking temperature constraints into consideration. We developed a closed-form solution to capture temperature dynamics for a given periodic voltage schedule on multi-core platforms,

and also developed three methods to check the feasibility of a periodic real-time schedule under peak temperature constraint. We further extended our research by incorporating the power/energy constraint with thermal awareness into our research problem. We investigated the energy estimation problem on multi-core platforms, and developed a computation efficient method to calculate the energy consumption for a given voltage schedule on a multi-core platform. In this dissertation, we present our research in details and demonstrate the effectiveness and efficiency of our approaches with extensive experimental results.

TABLE OF CONTENTS

CHAPTER	PAGE
1. INTRODUCTION	1
1.1 Real-Time Embedded Systems	1
1.2 The Opportunities And Challenges For Multi-Core Platforms	4
1.3 The Research Problem And Our Contributions	9
1.4 Structure Of The Dissertation	10
2. BACKGROUND AND RELATED WORK	12
2.1 Real-Time Scheduling	12
2.2 Multi-Core Scheduling	16
2.3 Power/Thermal Aware Multi-Core Scheduling	17
2.3.1 Power Aware Multi-Core Scheduling	18
2.3.2 Thermal Aware Multi-Core Scheduling	21
2.4 Summary	24
3. PARTITIONED MULTI-CORE SCHEDULING BY EXPLORING HARMONIC RELATIONSHIP AMONG REAL-TIME PERIODIC TASKS . .	26
3.1 Related Work	26
3.1.1 Different Utilization Bounds For Single-core Systems	27
3.1.2 Partitioned Scheduling	30
3.2 Preliminary	32
3.3 Motivational Examples	34
3.4 Task Partition With An Enhanced RBound	37
3.4.1 Task Set Scaling (TSS)	38
3.4.2 Feasibility Relationship Between Γ And Γ'	39
3.4.3 Enhanced RBound	43
3.4.4 The Partitioning Algorithm	45
3.5 Harmonic Advantage Exploration With CBound	47
3.5.1 Quantifying Harmonic Property	48
3.5.2 Harmonic Aware Partitioned Scheduling	49
3.5.3 Schedulability Analysis for HAPS	51
3.6 Experiments And Results	53
3.6.1 Experimental Setup	53
3.6.2 Experiment 1: Efficiency Of Our Enhanced Utilization Bound	53
3.6.3 Experiment 2: Performance Of Our Partitioned Scheduling Algorithms	56
3.7 Summary	60
4. SEMI-PARTITIONED MULTI-CORE SCHEDULING BY EXPLORING HARMONIC RELATIONSHIP AMONG REAL-TIME PERIODIC TASKS . .	61
4.1 Related Work	61
4.2 Preliminary	63

4.2.1	System Models	63
4.2.2	On Semi-Partitioned Scheduling	64
4.2.3	Motivation Examples	66
4.3	The HSP-Light Algorithm	67
4.3.1	Algorithm Details	68
4.3.2	Schedulability Analysis Of HSP-Light	70
4.3.3	Fast Schedulability Checking Method For HSP-Light	72
4.4	The HSP Algorithm	78
4.4.1	Algorithm Details	80
4.4.2	Schedulability Analysis Of HSP	82
4.5	Experiments And Results	85
4.5.1	Performance VS. Number Of Tasks	87
4.5.2	Performance VS. System Utilization	88
4.6	Summary	90
5.	TEMPERATURE-CONSTRAINED FEASIBILITY ANALYSIS FOR MULTI-CORE REAL-TIME SCHEDULING	91
5.1	Related Work	91
5.2	Preliminary	93
5.2.1	System Models	93
5.2.2	Power Model	94
5.2.3	Thermal Model	95
5.2.4	Problem Description	96
5.3	Temperature Calculation For Multi-core Scheduling	97
5.3.1	Temperature Formulation Within A State Interval	97
5.3.2	Temperature Formulation For A Periodic Schedule	98
5.3.3	Steady-State Temperature Formulation	102
5.4	Identifying The Peak Temperature	104
5.4.1	Challenging Problem In Peak Temperature Detection	104
5.4.2	Important Properties For Multi-core Temperature Variation	106
5.4.3	Peak Temperature Detection Within A State Interval	109
5.4.4	Peak Temperature Detection For A Periodic Schedule	113
5.5	Feasibility Analysis For Multi-Core Scheduling With Temperature Constraint	114
5.5.1	TmaxCheck: Feasibility Checking With Initial Temperature As T_{max}	115
5.5.2	ModeCheck: Feasibility Checking With Temperature Safe Modes	116
5.5.3	TssCheck: Feasibility Checking With Steady-State Temperature Formula	119
5.6	Experimental Evaluations	120
5.6.1	Accuracy Analysis Of Our Analytical Temperature Calculation Method	121
5.6.2	Steady-State Peak Temperature Variation Under Different Constant Speeds	124
5.6.3	Threshold Temperature Determined By TmaxCheck	125
5.6.4	Worst-Case Equilibrium Voltage Determined By ModeCheck	126

5.6.5	Performance Comparison For Different Feasibility Checking Methods	128
5.7	Summary	130
6.	LEAKAGE-AWARE ENERGY ESTIMATION FOR MULTI-CORE REAL-TIME SCHEDULING	131
6.1	Related Work	132
6.2	Preliminary	133
6.2.1	System Models	133
6.2.2	Temperature Calculation	134
6.3	Energy Calculation For Multi-Core Scheduling With Thermal Awareness	135
6.4	Experiments And Results	139
6.4.1	Experimental Setup	139
6.4.2	Accuracy Analysis	142
6.4.3	Time Efficiency Analysis	143
6.5	Summary	144
7.	CONCLUSIONS AND FUTURE WORK	145
7.1	Summary	145
7.2	Future Work	147
7.2.1	System Models And Underlying Scheduling Problem	148
7.2.2	Preliminary Results	151
	BIBLIOGRAPHY	157
	VITA	170

LIST OF TABLES

TABLE		PAGE
3.1	A task set with six real-time periodic tasks	34
3.2	A task set with four real-time periodic tasks	35
4.1	A task set with five real-time tasks	66
4.2	A task set with four real-time tasks	79
5.1	HotSpot parameters and floorplan	121
5.2	Power/thermal parameters	122
6.1	HotSpot parameters and floorplan	139
6.2	Power/thermal parameters	139

LIST OF FIGURES

FIGURE		PAGE
1.1	Embedded system market [114]	2
1.2	Demand for multi-core based devices	4
1.3	Fraction of chip reachable in one clock cycle [6, 103]	5
1.4	Time line of multi-core development [117]	7
1.5	The trend of power consumption and transistor count for a $300mm^2$ die [24]	8
2.1	Power v.s. Temperature [55]: Intel Core i5-2500K (32nm Sandy Bridge), voltage 1.26V, frequency at 1.6 GHz and 2.4 GHz, respectively. . . .	18
2.2	Illustration for RC thermal circuit on a dual-core system [116]	22
3.1	Assign tasks in Table 3.1 based on ideal harmonic relationship, and all tasks can be scheduled successfully on two processors.	34
3.2	Assign tasks in Table 3.2 based on pCOMPACTS [66], while τ_4 missing its deadline.	35
3.3	Assign tasks in Table 3.2 based on closely harmonic relationship, and all tasks can be scheduled successfully on two processors.	36
3.4	Proof of Theorem 3.4.1: given a task set Γ with $T_1 \leq T_2 \dots \leq T_N$ and $\tau_N = (C_N, T_N)$, transform Γ into Γ^* such that $\tau_N^* = (kC_N, kT_N)$ and $\tau_i^* = \tau_i, \forall i < N$	39
3.5	Efficiency of our enhanced utilization bound on a single core.	54
3.6	Experimental results for light task sets ($u_i \in [0, 0.5]$)	55
3.7	Experimental results for light task sets ($u_i \in [0, 0.5]$) by different system utilization	58
3.8	Experimental results for general task sets ($u_i \in [0, 1]$) by different system utilization	59
4.1	Allocation fails when simply grouping harmonic tasks and assigning them to the same processor.	66
4.2	Illustration of U_X^t and U_Y^t	76
4.3	(a) The task set is failed to be scheduled according to HSP-light; (b) The task set is schedulable if the heavy task τ_2 is pre-assigned. . . .	79
4.4	Experimental results for general task sets by different number of tasks. .	87

4.5	Experimental results for light task sets, $u \in [0, 0.5]$	88
4.6	Experimental results for general task sets, $u \in [0, 1]$	89
5.1	A speed schedule within 2 scheduling periods.	99
5.2	Negative interaction on temperature variation between two cores. $C_1 =$ $C_2 = 0.00035$, $G_{11} = G_{22} = 0.4$, $G_{12} = G_{21} = -0.1$, $v_1 = 0.8V$, $v_2 = 0V$, $T_1(0) = T_2(0) = 75^\circ C$	105
5.3	$T_i(t)$ increases at both time t_0 and t_1	109
5.4	$T_i(t)$ decreases at time t_0 and increases at time t_1	110
5.5	$T_i(t)$ increases at time t_0 and decreases at time t_1	111
5.6	Equilibrium voltage of core \mathcal{C}_i under processing mode k_i . a) all other cores except \mathcal{C}_i are under fixed constant processing modes; b) all other cores except \mathcal{C}_i are under any arbitrary available processing modes.	116
5.7	Accuracy analysis of our proposed temperature calculation method. . . .	123
5.8	Steady-state temperature under different constant speeds with our ana- lytical steady-state temperature formula.	124
5.9	Threshold temperature determined by TmaxCheck under different volt- ages.	126
5.10	Worst-case equilibrium voltage determined by ModeCheck under differ- ent maximum temperature constraints.	127
5.11	Feasibility ratios under different maximum temperatures.	128
6.1	Accuracy analysis, compared with the numerical method under $t_s = 0.01$	141
6.2	Time efficiency analysis, normalized with our method	143
7.1	Impact of slack on energy consumption	155
7.2	Impact of P_{ind} on energy consumption	156

CHAPTER 1

INTRODUCTION

Real-time embedded systems have been ubiquitous. From cell phones to digital cameras, from transportation to industry controls, from medical instruments to home entertainment systems, such systems affect almost every aspects of our daily life. In the meantime, to cater to the growing demand of high computing performance for these systems, the traditional approach of increasing speed for single processor has been a way of past. Instead, multi-core architecture is becoming mainstream.

In this chapter, we first introduce the basics on real-time embedded systems. Then we discuss the opportunities and critical challenges in design of real-time systems on multi-core platforms. Next, we define our research problem and briefly summarize our contributions. Finally, we present the structure of this dissertation.

1.1 Real-Time Embedded Systems

Real-time embedded systems are systems dedicated to special applications with real-time constraints in an embedded mechanical or electrical environment [54]. In a real-time embedded system, the timing constraints can be critical and need to be guaranteed, for reasons such as safety and usability. A late response, even coming with a logical correct result, can cause a degraded quality of service (QoS), or even a catastrophic accident [106].

Real-time embedded systems have been widely used in a variety of devices across a wide range of applications such as mobile phones, electronic game devices, motor vehicles, medical equipments, avionic products, etc. The embedded system market was valued at 121 billion dollars in 2011, and is expected to reach 194 billion dollars by 2018 [3]. Among all these systems/applications, it is reported in 2013 up to 68%

of all embedded system devices have the real-time capability [114] (see Figure 1.1). Real-time embedded systems have become indispensable in our daily life.

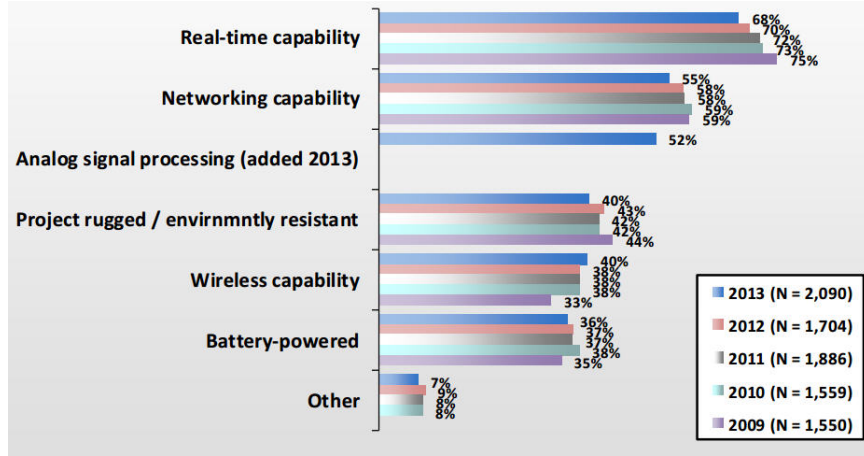


Figure 1.1: Embedded system market [114]

Due to the application nature, there are a large variety of different real-time embedded systems. In general, real-time embedded systems can be classified along different criteria. From the perspective of the nature of deadlines, real-time embedded systems can be hard or soft. Hard real-time systems require deterministic guarantee to meet all deadlines for every instance, and the failure to meet even a single deadline can be catastrophic. For example, aviation control system and automobile's ABS system are hard real-time systems. On the other hand, soft real-time systems are the systems that allow for a statistical bound on the number of deadline misses, which are neither desirable nor fatal. Examples of such systems include media streaming in distributed systems and non-mission-critical tasks in control systems. Despite large variations of real-time embedded systems, one unique common feature of real-time embedded systems is that they are usually tightly resource constrained. For instance, beside timing, real-time embedded systems are also constrained by size, weight, power/energy, temperature, reliability, etc. Due to limited resources in most of embedded devices, the problem of how to improve the perfor-

mance meanwhile satisfying other resource constraints becomes important. Take the mobile phones as an example, they have essential restrictions on size, weight, thermal and power. Power is particularly important, as these portable devices largely depend upon the battery-life to deliver high performance [107, 127]. To achieve high computational performance within limited and constrained resources, an appropriate real-time scheduling strategy for such embedded systems is desired.

The real-time scheduling is concerned with the allocation and management of the resources to complete the assigned workload within timing constraints. In a real-time embedded system, the scheduling strategy directly affects the application's execution and thus further affects the computing performance. Moreover, the scheduling strategy also brings significant impact on other system performances, i.e. power/energy, thermal, reliability, etc. There is no doubt that real-time scheduling plays a critical role in embedded systems. Thus, it is important to design effective and efficient scheduling techniques for real-time embedded systems.

From the processor architecture point of view, real-time embedded systems can be categorized as either single-core or multi-core systems. The single-core system is built by integrating only one processing core into a single chip, while the multi-core system integrates multiple processing cores into the same chip. Over two decades, it has been a common strategy to increase the computing performance by building more complex single-core architecture and increasing working frequency. Such kind of advancement has been largely driven by the continuous scaling of the transistor feature size that facilitates exponential transistor integration capacity (doubling every 2 years, Moore's law). However, under current technology, the power issue has become a critical bottleneck for further increasing the computing performance on single-core systems. Fortunately, multi-core systems can mitigate the power issue and thus provides capability to further increase the computing performance.

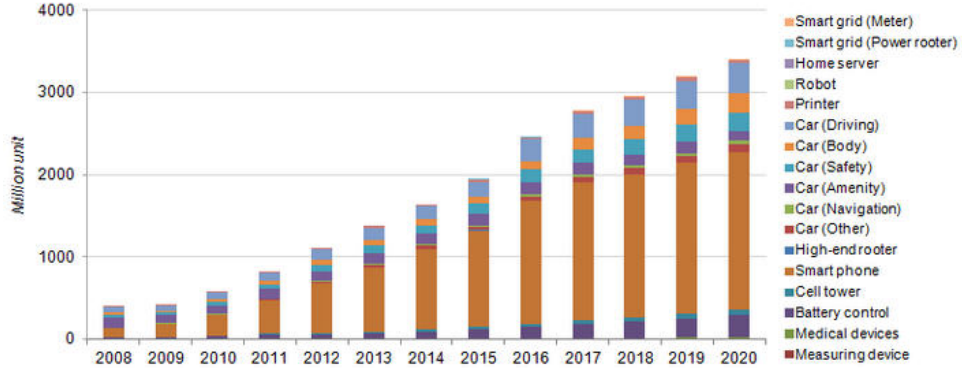


Figure 1.2: Demand for multi-core based devices

Today, multi-core systems have been the mainstream of microprocessor market in various fields. For example, multi-core platforms are widely used from personal electronic devices (e.g. smart phones, PCs, tablets and tablets) to system servers and data centers. Moreover, there is a quick increase of multi-core devices in the commercial market. The data shown in Figure 1.2 was gathered by one research company [2], and shows the annual increase in the number of multi-core processors delivered in select industries. Based on this research, starting from 2012, there is approximately a 40% annual increment in the shipment of multi-core microprocessors. As multi-core architecture is becoming more and more popular, there is a quick emerging towards multi-core for real-time embedded systems. In the following section, we discuss the opportunities as well as challenges coming with the multi-core technology.

1.2 The Opportunities And Challenges For Multi-Core Platforms

Since early 2000, industry has begun to change its focus from single-core to multi-core platforms. One major reason for this platform shift is that, in 2002, the classical

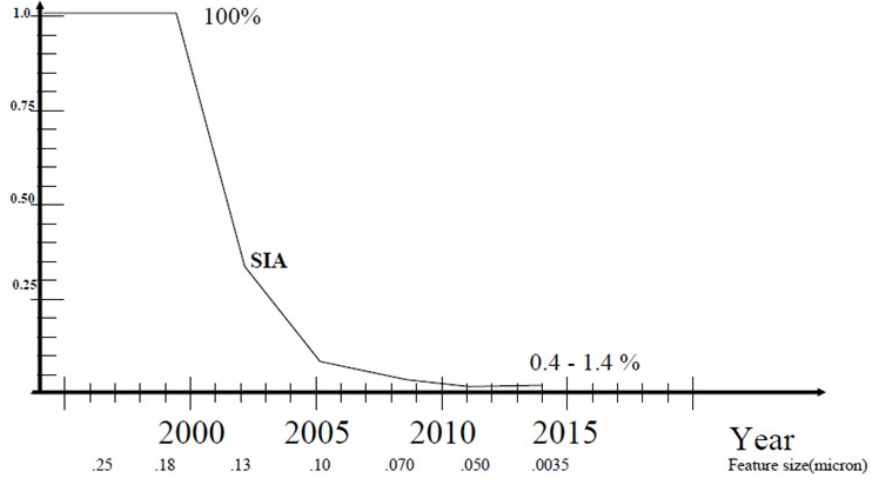


Figure 1.3: Fraction of chip reachable in one clock cycle [6, 103]

approach for increasing computing performance (by scaling the transistor size and increasing the clock frequency) reached a physical limit, i.e. the entire chip could not be reached in one clock cycle. This means that the performance could not be continuously increased under the traditional way of transistor technology scaling. Figure 1.3 shows the fraction of chip reachable in one clock cycle with respect to year and technology. From Figure 1.3, we can see that starting from 2000, there is an exponential drop in the percentage of chip achieved in one clock cycle. In fact, the trend of increasing the speed of processor to increase the computing performance is a way of past.

Multi-core platforms bring innovative solutions to overcome the limitations of single-core platforms, such as power/thermal limitation and instruction level parallelism limitation [15]. First, the power and thermal issues have become a crucial limitation in single-core design. The extremely high power consumption and excessive heat dissipation have posed critical challenges for continuously pursuing high computing performance on a single-core chip [112]. However, multi-core platforms, compared with single-core platforms, can alleviate the power and thermal issues

with the same performance achievement. Instead of continuously scaling transistor size or increasing clock frequency only on a single core, multi-core platforms can increase the computing performance by increasing the number of processing cores on the same chip with lower transistor integration density and/or clock frequency. Secondly, single-core platforms confront with instruction level parallelism limitation. Single-core architectures attempting to gain performance from techniques such as wide issue and speculative execution achieve modest increase in performance at the cost of significant overhead in area and energy [15]. Nevertheless, multi-core platforms improve the computing performance by exploiting the “thread/data level parallelism”. For instance, in a multi-core system, if all tasks are highly parallelized among all cores, it would come out with high parallel executions. Therefore, multi-core technology bring promising opportunities to further improve the computing performance.

The study on multi-core platforms is long and rich, while the history of manufactured multi-cores has only a few decades. Figure 1.4 shows some significant time line for the development of multi-core systems [117]. In 1972, one of the most early studies on multi-cores, called Illiac IV [25] consisting of 64 arithmetic logic units (ALUs), was proposed to perform parallel computing for vector and array operations. The first significant study on general purpose multi-core was performed by Hammond [46] in 1997 (shown in the figure as the Kunle study). However, the first manufactured multi-core appeared in the marketplace was in 2000 by AT&T Daytona [5]. Most of other major manufacturers followed successively to launch their multi-core chips: the C-5 by C-Port Corp. for networking, the viper by Phillips for multimedia, the OMAP by Texas Instruments for baseband processing, the MPCore by ARM for configurable design, the IXP2855 by Intel for network communication, and the Starcore by Sandbridge Technologies for signal processing. Today, multi-

core platforms have been broadly supported by most of chip vendors, including Intel, AMD, ARM, IBM, Nvidia, Freescale Semiconductor, Sun Microsystems, etc.

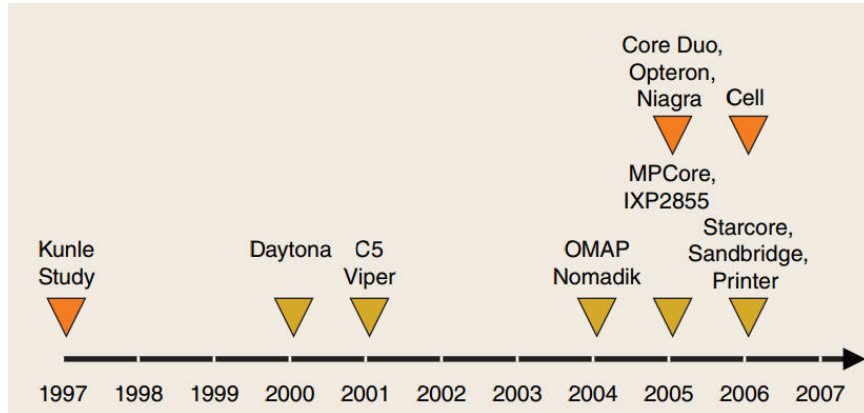


Figure 1.4: Time line of multi-core development [117]

When moving to multi-core platforms, it comes with new critical challenges in design of real-time systems. First, in a multi-core system, the scheduler needs to determine not only when a given application executes but also where it executes such that the system resources can be effectively and efficiently utilized. This problem, so called *partitioning* problem, is an NP-hard problem in nature [106]. Second, by taking the timing constraints of real-time applications into consideration, the problem of multi-core real-time computing becomes even more complicated. For example, with consideration of the dependency among different applications, the benefit coming from parallel executions of a multi-core system could be seriously suffered from guaranteeing real-time constraints [10]. There are critical challenges in design of real-time multi-core scheduling.

Secondly, the extremely high power consumption and excessive heat dissipation have also become the critical challenges in design of multi-core systems [112]. As shown in Figure 1.5¹, more than 100 billion transistors are being integrated in a $300mm^2$ die today, which results in a power consumption up to 300 watts. These

¹Figure 1.5 is plotted based on the data reported in [24].

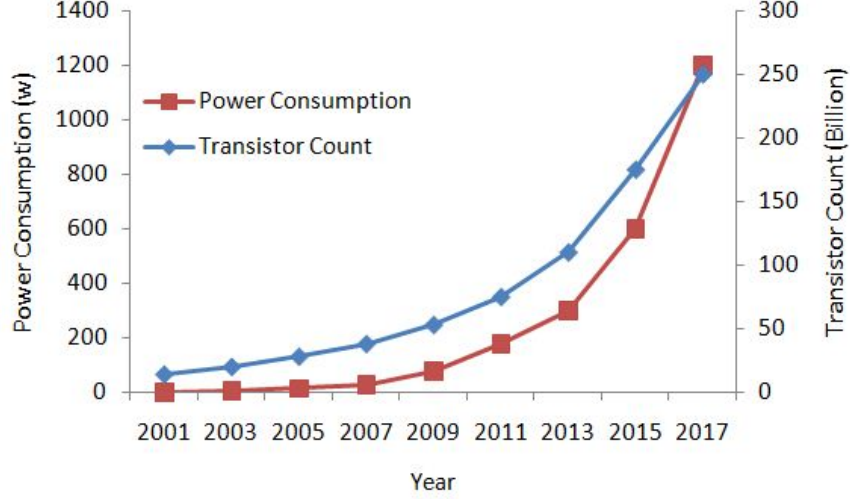


Figure 1.5: The trend of power consumption and transistor count for a $300mm^2$ die [24]

amount of power consumption has posed significant challenges in both portable devices and power-rich systems. The soaring power consumption makes the heat dissipation and the temperature control even more challenging, e.g. the severity of the thermal problem has been highlighted by Intel’s acknowledgement that it has hit a “thermal wall” [91].

Moreover, techniques and analysis methods for single-core platforms cannot be readily applicable for multi-core platforms. The traditional solutions on power, thermal and energy problems associated with single-core platforms could become ineffective without taking the multi-core characteristics (e.g. heat transform, hot spot and thermal gradient) into consideration [97]. Therefore, in order to take the opportunities and advantages of multi-core technology, it is necessary and important to appropriately consider and address the new emerging challenges.

From the above discussion, we can see that the design of multi-core systems faces new opportunities as well as various challenges. In what follows, we describe our research problem in this dissertation, and briefly summarize our contributions.

1.3 The Research Problem And Our Contributions

We are interested in the research problem on how to develop advanced techniques for real-time embedded systems on multi-core platforms. Researchers and engineers from both academia and industry have been working on this problem at different design abstraction levels, i.e. from gate level, circuit level, architecture level, to system level. Our research focuses on attacking this problem from the system level. Specifically, we are interested in developing real-time scheduling techniques and analysis methods to guarantee timing and other design constraints, and in the meantime, to optimize design criteria such as system utilization, peak temperature, power and energy consumption. Toward this problem, we have made the following contributions:

1. First, we studied the classical problem of partitioned scheduling of real-time periodic tasks on multi-core platforms, with each task executed on a dedicated core. By taking the relationship among task periods into consideration, we developed several novel partitioned scheduling approaches for scheduling fixed-priority periodic real-time tasks on multi-core systems. Our proposed algorithms can greatly improve the schedulability of real-time tasks, and thus improve the system utilization. Compared with the related work, we found that our proposed algorithms could achieve an improvement at least of 14.5% in terms of task set schedulability under high system utilizations.
2. Then, we targeted at the problem of semi-partitioned multi-core real-time scheduling, in which most of tasks were executed on dedicated cores, while some tasks could be split and executed on different cores. We developed two new semi-partitioned scheduling strategies for hard periodic real-time tasks on multi-core systems. We also developed a deterministic worst-case utilization

bound for the proposed approaches. Simulation studies showed that our approaches could outperform the related work by 15% from the perspective of task set schedulability when systems were heavy loaded.

3. Next, we incorporated the temperature constraint into the problem of multi-core real-time scheduling. We developed a closed-form solution for temperature calculation for periodic speed scheduling on multi-core platforms. We further developed an effective method that can quickly obtain the maximum temperature for a periodic multi-core schedule. To our best knowledge, this is the first work that analytically solves the temperature calculation and peak temperature detection on multi-core platforms with consideration of a linear dependency model of leakage and temperature. Based on our proposed techniques, we proposed three feasibility testing approaches for multi-core scheduling with maximum temperature constraint.
4. Finally, we studied the energy estimation problem in multi-core scheduling. We developed a fast and accurate solution of energy calculation on multi-core systems with consideration of the interdependency of leakage, temperature and supply voltage. Our solution provides a fundamental for design of energy aware multi-core systems, and can be directly used for energy efficient multi-core scheduling. The experimental results showed that our proposed method can achieve an average speedup of 15X over the existing related work, with a relative error no more than 1.5%.

1.4 Structure Of The Dissertation

The rest of this dissertation is organized as follows. In Chapter 2, we introduce some pertinent background to this dissertation, and discuss the existing work closely re-

lated to our research. In Chapter 3, we study the problem of partitioned multi-core scheduling for periodic real-time tasks, and present a new partitioned scheduling strategy by exploring the “harmonic” characteristic. In Chapter 4, we focus our research in semi-partitioned multi-core scheduling, and propose an efficient scheduling algorithm with bounded worst-case system utilization and limited count of split tasks. In Chapter 5, we study the feasibility checking problem for temperature-constrained multi-core scheduling, and propose three feasibility checking conditions for multi-core scheduling with maximum temperature constraint. In Chapter 6, we present an energy estimation approach for multi-core systems with consideration of the interdependency between leakage power and temperature. Finally, in Chapter 7, we conclude this dissertation and discuss the possible future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter presents the pertinent research background and related work. We first introduce several important concepts related to real-time scheduling. Then, we introduce some preliminaries for multi-core scheduling. Next, we discuss the research problems and related work on power/thermal aware multi-core scheduling. Finally, we summarize the contents of this chapter.

2.1 Real-Time Scheduling

In a real-time system, the correctness of an execution result depends not only on the correctness of the logical computational results, but also on the time instant at which that result is finished. We can describe a real-time system as a system that has deadlines. The violations of timing constraints in real-time systems degrade the quality of service, and in some cases result in catastrophic accidents [83, 106]. To guarantee the timing constraints, one effective way is to design an appropriate scheduling algorithm.

In general, the real-time scheduling studies the problem of how to determine when and where a given set of tasks need to be executed such that all real-time constraints (e.g. deadlines) can be guaranteed, and meanwhile some other design metrics (e.g. temperature, power/energy consumption and reliability) can be optimized.

There are different ways to categorize the real-time scheduling. From the perspective of job characteristics, real-time scheduling can be categorized into hard/soft [28, 69], periodic/non-periodic [30, 49], etc. From the perspective of scheduling mechanisms, real-time scheduling can be categorized into static/dynamic [68, 76], priority-driven/non-priority-driven [98, 27], preemptive/non-preemptive [45, 35], etc. From

the perspective of underlying architectures, real-time scheduling can be categorized into single-core/multi-core [81, 38]. From the perspective of design objectives, real-time scheduling can be categorized into single-objective/multiple-objective (e.g. timing constraints only or more other design objectives such as power/energy, thermal, reliability) [71, 37]. In what follows, we discuss the details of the above categorizations to clearly understand the behaviors of real-time scheduling.

Hard Real-Time vs. Soft Real-Time: A hard real-time scheduling requires deterministic guarantee to meet all deadlines for every instance, and the failure to meet even a single deadline can be catastrophic. Examples of hard real-time scheduling can be found in aviation control system and automobile’s ABS. In contrast, a soft real-time scheduling allows for a statistical bound on the number of deadline misses, which are neither desirable nor fatal. Examples of soft real-time scheduling can be found in multimedia player systems, in which occasionally missing of deadlines does not effect the normal operations of the system, however, the quality of service may degrade.

Periodic vs. Non-Periodic: In a periodic scheduling, jobs/instances coming from the same task are released periodically with a minimum length of inter-arrival time between any two consecutive jobs. In other words, tasks are invoked at regular intervals following a determinate pattern of time intervals. For example, in air traffic control(ATC) system, the status of each aircraft is monitored using active radars. These radars check the status periodically and update the ATC controller [83]. In an aperiodic scheduling, each task is modeled as a sequence of jobs with unknown/indeterminate release time, thus all tasks are invoked in irregular pattern and the inter-arrival time between consecutive jobs in such a task may vary widely. For instance, in a setting of radar surveillance system, the system should be responsive to operator’s commands but not on the expense of task with hard deadline.

Static vs. Dynamic: Static scheduling determines the priorities of tasks only based on the off-line available information. In other words, priorities of tasks are assigned before compile time and remain unchanged throughout the execution [81], e.g. rate monotonic scheduling (RMS). Dynamic scheduling makes scheduling decisions based on the run-time information, thus the priority of each job/task becomes known to the scheduler only after that job is released during on-line execution, e.g. earliest deadline first (EDF).

Priority-Driven vs. Non-Priority-Driven: In priority driven real-time scheduling, at any scheduling decision time instant, the jobs with the highest priorities are scheduled and executed on the available processors. Other commonly used names for this approach are greedy scheduling, list scheduling and work-conserving scheduling [83]. Some examples of priority-driven scheduling include EDF, RMS [81], etc. On the other hand, in non-priority driven scheduling, decisions are made based on, instead of priority criteria, some other criteria or policy (e.g. the round-robin policy) to determine if a task should start executing or not [113].

Preemptive vs. Non-Preemptive: If the execution of lower priority task is stopped or preempted for a higher priority task then the scheduling scheme is called preemptive scheduling and otherwise non-preemptive scheduling [12, 27].

Single-Core vs. Multi-Core: Based on type of underlying architectures, real-time scheduling can be categorized into single-core scheduling [81] and multi-core scheduling [106]. One major difference of multi-core over single-core scheduling is that, in multi-core scheduling, we need to decide not only when but also where a task should be executed. Multi-core scheduling, known as a NP-hard problem [106], is more complicated compared with single-core scheduling.

Timing-Constrained vs. Multiple-Constrained: The classical timing-constrained real-time scheduling exclusively focus on timing constraints, while the multiple-

constrained real-time scheduling incorporates other design objectives such as power/energy[56], thermal [82] and reliability [49].

Two single-core priority-based preemptive scheduling policies, i.e. *Earliest Deadline First*(EDF) and *Rate Monotonic Scheduling* (RMS), are of special interest and great importance [81]. These two scheduling policies play a fundamental role in design of real-time scheduling.

Earliest Deadline First (EDF): The EDF is a preemptive, dynamic-priority scheduling algorithm. Task’s priorities are assigned dynamically during run time. The task with the least time remaining before its deadline acquires the highest priority and thus executed before others. In fact, it is proved in [81] that if a task set is schedulable, then EDF algorithm can schedule it. Due to its 100% utilization bound, EDF becomes the underlying scheduling algorithm for a number of other scheduling techniques with different design objective, such as the “low power EDF” algorithm proposed in [119].

Rate Monotonic Scheduling (RMS): Under the fixed-priority RMS policy, tasks’ priorities are assigned based on their periods. It is shown by Liu and Layland [81] that RMS is the optimal among all fixed-priority scheduling policies. They have proved that a feasible schedule can be found by using RMS if the total utilization is less than or equal to $\ln(2)$ (69.3%).

Both EDF and RMS have been used extensively in the research domain as the underlying scheduling policy for other design metrics optimization like energy minimization ([98, 133, 99]), schedulability/feasibility analysis([4, 97]), etc.

2.2 Multi-Core Scheduling

Multi-core architecture has been widely accepted as the most important technology in the future industrial market. By providing multiple processing cores on a single chip, multi-core systems, compared with the traditional single-core systems, can significantly increase the computing performance while relaxing the power requirement. Most of the major chip manufactures have already launched 16-core chips into the market, i.e. AMD *OpteronTM 6300 Series* [7]. It is not surprising that in the coming future, hundreds or even thousands of cores will be integrated into a single chip [121]. The quickly emerging trend towards multi-core platforms brings urgent needs for effective and efficient techniques for the design of multi-core scheduling.

Multi-core scheduling can be categorized into different classes based on different criteria, i.e. homogeneous/heterogeneous (from the perspective of underlying architectures) [40, 105], global/partitioned/semi-partitioned (from the perspective of scheduling mechanisms) [9, 10, 39], timing-constrained/multiple-constrained (from the perspective of design constraints) [8, 115], etc.

Homogeneous vs. Heterogeneous: On a homogeneous platform, all processing cores are identical, hence the rate of execution of all tasks is the same among all cores. Thus, the scheduling strategy only needs to concern the execution time of each task. While on a heterogeneous platform, since the processing cores are different, hence the rate of execution of a task depends on both the core and the task. Indeed, not all tasks may be able to execute on all processors. Thus, the design of multi-core scheduling for heterogeneous platforms becomes more complicated.

Global vs. Partitioned vs. Semi-Partitioned: In the global scheduling approach, all jobs first enter a global queue, and thus each task can be potentially executed on any processor [9, 40]. In the partitioned scheduling approach, each

real-time task is assigned to a dedicated processor. All instances from the same task will be executed solely on that particular processor [10, 38]. The semi-partitioned scheduling approach is a combination of previous two approaches, i.e. some tasks are assigned to a dedicated processor, while rest can migrate among available resources [69, 72, 43, 39].

Timing-Constrained vs. Multiple-Constrained: Traditional approaches focus exclusively on timing constraints [10, 8, 34, 39], and many recent work takes other design objectives (e.g. power/energy, thermal and reliability) into considerations, which makes the scheduling problem more complicated [61, 51, 58, 57, 49].

One of the most common and useful performance metrics used to compare the effectiveness of different multi-core scheduling algorithms and schedulability analysis is the utilization bound. The utilization bound for a scheduling algorithm is defined as the minimum utilization of any taskset that is only just schedulable according to that algorithm. The best known utilization bound for either global or partitioned schedule under RMS is no more than 50% [10, 13, 9], while the utilization bound can reach up to 69.3% for semi-partitioned scheduling under RMS [43, 44, 39]. There are also other metrics to evaluate performances for different multi-core scheduling algorithms, i.e. approximation ratio (the ratio of the number of required processors of a multi-core scheduling algorithm over that of the optimal algorithm), and empirical system schedulability (the percentage of tasksets that are found to be schedulable).

2.3 Power/Thermal Aware Multi-Core Scheduling

The continuously increased power consumption has resulted in a soaring chip temperature. Moreover, as design paradigm shifts to deep sub-micron domain, high chip temperature leads to a substantial increase in leakage power consumption [60],

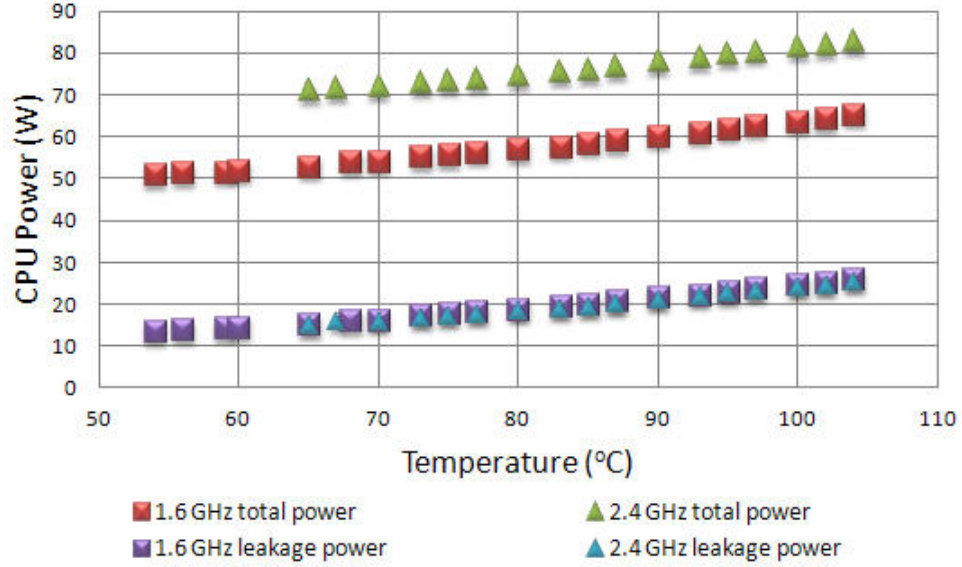


Figure 2.1: Power v.s. Temperature [55]: Intel Core i5-2500K (32nm Sandy Bridge), voltage 1.26V, frequency at 1.6 GHz and 2.4 GHz, respectively.

which in turn further deteriorates the power situation due to the interdependency between temperature and leakage power. For instance, with Intel core i5-2500K (32nm Sandy Bridge), the leakage power roughly grows up to 2X from 55°C (13W) to 105°C (26W), see Figure 2.1. Further more, the soaring chip temperature adversely impacts the performance, reliability, and packaging/cooling costs [97]. As a result, power and thermal issues have become critical and significant for advanced multi-core system design. In this section, we introduce some necessary backgrounds of multi-core scheduling with power and thermal awareness, respectively.

2.3.1 Power Aware Multi-Core Scheduling

Catalyzed by continuous transistor scaling, hundred of billions of transistors have been integrated on a single chip [60]. One of the immediate consequence caused by the tremendous increase of transistor density is the soaring power consumption [18], which further results in severe challenges in energy and temperature[57, 97]. Today,

power has become a critical and challenging design objective in front of system designers. In this subsection, we first describe our research problem on power aware multi-core scheduling, next introduce the general power model, and then discuss the related work.

Research Problem of Power Aware Scheduling

Power aware scheduling studies the problem on how to apply the *dynamic voltage and frequency scaling* (DVFS) mechanism to adjust the clock frequency and supply voltage of a processor to execute a set of tasks such that the time constraints (e.g. deadlines) can be guaranteed and meanwhile the power and/or energy consumption can be optimized.

Power Consumption

The overall power consumption of an IC chip can be divided into two categories: dynamic power and leakage power [102]. The dynamic power consumption is associated with the switching of the logic value of a gate, and thus is essential to performing useful logic operation by charging and discharging the circuit load capacitance. In general, the dynamic power is modeled as function in proportion to working frequency and square of supply voltage [102]. The leakage power, also known as static power, is consumed due to the leakage mechanism of a CMOS transistor and it does not contribute to any useful computation. Traditionally, the leakage power is modeled as a constant, and is dominated by the dynamic power. However, as the technology entering the deep sub-micron region, leakage power becomes more significant in the total amount of power consumption, and has distinct interdependency with temperature (i.e. leakage power can be approximated as a linear function of temperature and voltage [97]). This signifies the need for incorporating leakage/temperature dependency into the design and analysis of power efficiency systems.

Related Work On Power Aware Scheduling

Early research work on the problem of power aware multi-core scheduling is mainly focused on minimizing the dynamic power and its corresponding energy consumption (since dynamic power plays a dominant role over leakage power in the overall power consumption). By taking advantage of the convex relationship between the dynamic power and supply voltage, a number of methods (e.g. [76, 119]) were proposed to lower down the processor speed (i.e. supply voltage and working frequency) such that the power and/or energy consumption can be reduced meanwhile all tasks can be finished just before their “deadlines”.

As the leakage power becomes more prominent, it is no longer optimal in the power/energy reduction by only considering the characteristics regarding the dynamic consumption. This is because the saved dynamic energy might be outweighed by the increased leakage part. Moreover, by taking leakage/temperature dependency into consideration, the power aware multi-core scheduling problem becomes even more complicated.

A great number of literature are published on solving the power aware multi-core scheduling problems with consideration of leakage/power dependency [50, 52, 85, 129, 29, 125, 123, 90, 58, 126, 30, 118]. Based on different criteria, these existing work can be classified into different categories. For example, based on the target platforms, we have techniques proposed for 2-dimension multi-core platforms [50, 52], or 3-dimension multi-core platforms [85, 129]. Based on the task models, we have tasks with stochastic [84] or deterministic workload [29, 125]. Based on the timing requirement, we have soft real-time [123] or hard real-time scheduling [30, 118]. Based on the stages during which the scheduling decisions are made, we also have on-line approaches [123, 90] and off-line approaches [58, 126].

2.3.2 Thermal Aware Multi-Core Scheduling

After introducing the problem of power aware scheduling in the above section, now we introduce the problem of thermal aware multi-core scheduling. The aggressive semiconductor technology scaling has pushed the chip power density doubled every two to three years [92, 109], which immediately results in an exponential increasing in heat density. As introduced in Chapter 1, high temperature can degrade the performance of systems in various ways. Therefore, there is a great need of advanced techniques for thermal/temperature aware design of multi-core systems. In this subsection, we first introduce our research problem on thermal aware multi-core scheduling, then describe the thermal impact, and later discuss the related work.

Research Problem of Thermal Aware Scheduling

Thermal aware scheduling studies the scheduling problem in the system level with thermal/temperature awareness. Specifically, it studies the problem of how to develop effective and efficient scheduling algorithms such that the temperature requirement as well as the real-time requirement can be met, and at the same time other design metrics (i.e. peak temperature, throughput, energy consumption, etc.) can be optimized.

Thermal Modeling of Multi-Core Platforms

From the circuit-level aspect, the multi-core thermal model can be represented by an equivalent RC thermal circuit [105, 116]. Figure 2.2 shows an example of such thermal model on a dual-core platform. Basically, there are four abstraction layers in this RC-thermal model, namely die layer, thermal interface material(TIM) layer, heat spreader layer, and heat sink layer. Thermal nodes on the die layer are called active nodes, since they represent the actual processing cores of the system and consume non-zero power. In contrast, thermal nodes on the thermal package

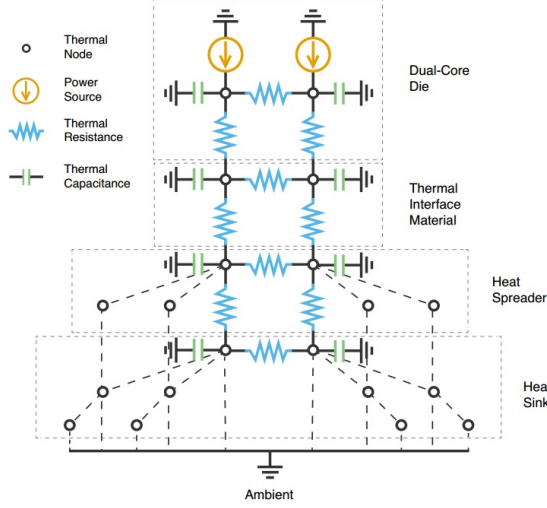


Figure 2.2: Illustration for RC thermal circuit on a dual-core system [116]

(i.e. three cooling layers) are called inactive nodes, since their power dissipation is assumed to be zero regardless of the system processing modes.

Based on the circuit-level RC thermal model, the thermal phenomena of the entire chip can be formulated as

$$\mathbf{C} \frac{d\mathbf{T}(t)}{dt} + \mathbf{G}\mathbf{T}(t) = \mathbf{P}(t) \quad (2.1)$$

where \mathbf{C} , \mathbf{G} , \mathbf{T} and \mathbf{P} are thermal capacitance matrix, thermal conductance matrix, temperature vector and power vector, respectively. From the above equation, we can see that the higher the temperature is, the larger the power will be. Moreover, as mentioned in previous subsection, the power \mathbf{P} , which is comprised of dynamic power and leakage power, also depends on temperature. Thus, we see there is an inter-dependency between power and temperature.

Related Work On Thermal Aware Scheduling

An increasing number of researches have been published on peak temperature minimization for thermal aware multi-core scheduling [70, 40, 41, 63, 124, 16, 101]. Chantem et al. [28] proposed an MILP-based solution to minimize the peak tem-

perature when executing a task graph. Lars *et al.* [104] proposed an approach to estimate the worst-case temperature for a core by searching for the worst case task/workload allocations among different cores. Ukhov *et al.* [116] presented a method to estimate the peak temperature by keeping track of temperature dynamics of a multi-core system until it reached the system steady state. Kumar *et al.* [70] proposed a stop-n-go approach to reduce the peak temperature for task with data dependencies. They distributed the slack time between jobs such that the peak temperature could be minimized and there was no make-span violation.

There are many researchers studied the thermal aware multi-core scheduling from the aspect of throughput maximization [29, 58, 82, 52]. Chantem *et al.* [29] proposed a method to run real-time tasks by frequently switching between the two speeds which are neighboring to an ideal constant speed whose stable temperature was equal to the given peak temperature. Fisher *et al.* [40] presented a method to minimize the peak temperature in a homogeneous multi-core system by deriving an ideally preferred speed for each core in a global task scheduling environment. Huang *et al.* [58] proposed two approaches to maximize the throughput for a periodic real-time system under the given peak temperature constraint, one for processor with simple active and sleep mode and the other for more complicated processors with DVFS capabilities. Hanumaiah *et al.* [53] also focused on the problem of throughput maximization, and they addressed task-to-core allocation over migration intervals and voltage speed scaling within migration intervals as a separate problem and translated task-to-core allocation into a *MILP* formulation.

Many researchers have focused on the energy minimization problem for multi-core scheduling with maximal temperature constraint [57, 118, 19, 51, 86]. Huang *et al.* [57] derived a closed-form energy calculation equation based on which they further proposed an energy minimization scheduling method for periodic task sets.

In [118], Yang et al. presented a procedure to find the optimal pattern of schedule with the minimum energy consumption at the steady state. Hanumaiah et al. [51] formulated energy minimization as a quasi-concave optimization problem and employed DVFS, task migration and cooling methods to optimize the objective function on a multi-processor system. Liu et.al [86] developed a thermal-constrained energy optimization procedure to minimize system energy consumption under peak temperature constraint.

Recently, significant amount of work targeted on 3D architectures for thermal aware multi-core scheduling has been published [33, 129, 78, 85, 90, 115]. In [78], the authors proposed a scheduling algorithm to reduce peak temperature in a 3D multi-core system by dynamically rotating tasks among different cores. Liu et al. [85] proposed a 3D thermal aware job allocation technique to reduce the peak temperature, through which hot jobs were always assigned to the cores near heat sink such that the heat could be quickly dissipated. Zhu et al. [130] presented a run-time thermal management technique that exploited the heterogeneity of processing cores in a 3D system. Coskun et al. [33] proposed an adaptive approach to balance the temperature among all cores in a 3D architecture. They adopted a second order polynomial temperature/leakage dependency model and developed a thermal aware scheduling algorithm that made the partitioning decision based on a thermal history of each core and that of its neighboring cores.

2.4 Summary

In this chapter, we discussed the essential background of our research and introduced some closely related work. We first presented a general introduction of the basic concepts and critical techniques in real-time scheduling. Particularly, we in-

introduced several different categorizations of real-time scheduling, and two important policies (e.g. RMS and EDF) in single-core scheduling. Next, we presented some preliminaries for multi-core scheduling, including categorizations from different perspectives, and several metrics used for evaluate the performance of multi-core scheduling. Then we discussed the multi-core scheduling problems with consideration of power/thermal awareness. We respectively introduced the research problems regarding to power and thermal, and discussed the related work.

In this dissertation, our goal is to develop effective scheduling methods for multi-core real-time systems to satisfy timing and other constraints, and also to optimize various objectives (e.g. system utilization, power/energy, temperature and reliability). In the following four chapters, i.e. Chapters 3, 4, 5 and 6, we present our contributions. Then we conclude this dissertation in Chapter 7.

CHAPTER 3

PARTITIONED MULTI-CORE SCHEDULING BY EXPLORING HARMONIC RELATIONSHIP AMONG REAL-TIME PERIODIC TASKS

We first present our research on classical real-time multi-core scheduling with timing as the only constraint. Specifically, in this chapter, we focus on partitioned scheduling for periodic real-time tasks on multi-core platforms under the *Rate Monotonic Scheduling (RMS)* policy. One common approach for partitioned multi-core scheduling problem is to transform this problem into a traditional bin-packing problem, with the utilization of a task being the “size” of the object and the utilization bound of a processor being the “capacity” of the bin. However, this approach ignores the fact that some implicit relations among tasks may significantly affect the feasibility of the tasks allocated to each local processor. To improve the system schedulability, we seek to exploit the fact that the utilization bound of a task set increases as task periods are closer to harmonic on single-core platforms. The challenge here, however, is how to take advantage of this fact on multi-core processor platforms while guarantee the schedulability of the real-time tasks.

3.1 Related Work

In partitioned multi-core scheduling problem, the schedulability for tasks allocated on each processor can be determined based on feasibility conditions on single processors. To search for the optimal task partition for multiple processors is essentially a design space exploration problem, with complexity increasing rapidly with the size of the problem (e.g. the numbers of tasks or processors). How to quickly and accurately evaluate the schedulability of a design alternative (i.e. task partition) is key to the success of the partitioned multi-core scheduling problem. As a result, while there

exists exact timing analysis method for feasibility checking for tasks on a single core platform ([81, 71, 73]), they are not commonly for partitioned multi-core scheduling problem due to their large computational complexity. In stead, many other timing efficient feasibility checking methods, such as the utilization-bound based feasibility checking methods, are commonly used in the search for task partitions for multi-core scheduling problem.

3.1.1 Different Utilization Bounds For Single-core Systems

A *utilization bound* $f(\Gamma)$ for a task set Γ is a function of the parameters of Γ , and can be used to determine the schedulability of Γ under certain specific scheduling policy (e.g. RMS). By applying the parameters of Γ into $f(\Gamma)$, all tasks in Γ can be guaranteed to meet their deadlines if the task set utilization (denoted as $U(\Gamma)$) is no more than that parametric utilization bound, i.e. $U(\Gamma) \leq f(\Gamma)$. Note that $U(\Gamma)$ can be calculated by summing up the task utilizations of all tasks in $U(\Gamma)$, where a task utilization is the ratio of its execution time over its period.

For single-core systems, there are several utilization bounds proposed under RMS policy [26, 81, 71, 73, 48].

- *LLBound* [81]: The *LLBound* is a function with respect to the number of tasks, and is formulated as

$$LLBound(\Gamma) = N(2^{1/N} - 1), \quad (3.1)$$

where N is the number of tasks in the task set Γ . When N goes to infinity, the *LLBound* achieves its worst-case as 69%.

- *KBound* [71]: The *KBound* has a similar form as the *LLBound*, and is formulated as

$$KBound(\Gamma) = K(2^{1/K} - 1), \quad (3.2)$$

where K , instead of being the number of all tasks as that used by *LLBound*, is the minimum number of pseudo tasks (a pseudo task is a non-empty harmonic task set in which any two tasks are period dividable).

- *RBound* [73]: The *RBound* is proposed for tasks with special characteristics, i.e. the ratio between the maximum and minimum periods needs to be less than 2. The *RBound* takes not only the number of tasks but also the relationship among periods into consideration.

$$RBound(\Gamma) = (N - 1)(r^{1/N-1} - 1) + 2/r - 1 \quad (3.3)$$

where N is the number of tasks in the task set, and r is the ratio between the maximum and minimum periods and needs to satisfy $1 \leq r < 2$.

- *CBound* [48]: The *CBound* is the utilization bound proposed for a special type of task sets, called “harmonic” task sets¹, and can be formulated as.

$$CBound(\Gamma) = 1 \quad (3.4)$$

where Γ is a harmonic task set.

Among all four utilization bounds shown in the above, it has been proved that for RMS-based single-core scheduling, the *RBound* and *CBound* higher than the other two (i.e. the *LLBound* and the *KBound*) [73, 48]. However, these two utilization bounds (*RBound* or *CBound*) have critical limitations. The *RBound* can only be applied when a given task set satisfies the period constraint (i.e. $1 \leq r < 2$), while the *CBound* can only be used directly to harmonic task sets. Hence, in order to use the *RBound* or *CBound* for checking the schedulability of an arbitrary task set, we need to first transform the task set appropriately such that it satisfies the required condition.

¹A harmonic task set is a task set in which any two tasks are period dividable.

For *RBound*, there are a few methods proposed to transform a task set to satisfy the condition of $1 \leq r < 2$, such as [73, 66]. In particular, Lauzac *et al.* [73] proposed a task set scaling method by scaling all tasks with respect to the maximum period. Specifically, given a task set Γ , $\forall \tau_i \in \Gamma$, the period as well as the execution time of τ_i was scaled by

$$\begin{cases} C'_i = C_i \cdot 2^{\lfloor \log \frac{T_{max}}{T_i} \rfloor} \\ T'_i = T_i \cdot 2^{\lfloor \log \frac{T_{max}}{T_i} \rfloor} \end{cases} \quad (3.5)$$

where T_{max} represents the maximum period among all tasks. Their method scaled all task periods with respect to, but no larger than T_{max} . They formally proved that as long as the scaled task set was feasible then the original task set was also feasible.

Kandhalu *et al.* [66] presented another method by scaling the task set with respect to the minimum period. Specifically, given a task set Γ , $\forall \tau_i \in \Gamma$, the period and the execution time of τ_i was scaled by

$$\begin{cases} C'_i = C_i / \lfloor \frac{T_i}{T_{min}} \rfloor \\ T'_i = T_i / \lfloor \frac{T_i}{T_{min}} \rfloor \end{cases} \quad (3.6)$$

where T_{min} is the minimum period among all tasks. This method scaled all task periods with respect to, but no smaller than T_{min} . However, this approach cannot always guarantee the schedulability of the original task set once the scaled task set is schedulable. For example, consider a task set Γ consisting of four tasks with execution time and periods as $\{(3, 24), (32, 100), (40, 135)\}$ and $(15, 140)$. According to the scaling method introduced in [66], we can transform the task set to a new task set Γ' as $\{(3, 24), (8, 25), (8, 27), (3, 28)\}$. It is not difficult to verify that the new task set Γ' is schedulable while the original task set Γ is not schedulable.

For *CBound*, there are also a few methods proposed to transform a task set to satisfy the harmonic condition. Han *et al.* [47, 48] proposed two methods, i.e. *Sr*

and *DCT*, to transform a task set into a harmonic one. Since both methods result in the same harmonic task set, we only introduce the *DCT* method (which has a complexity equal to N^2) as below:

- Sort Γ by T with non-increasing order.
- For each $\tau_i \in \Gamma$, transform Γ to Γ'_i by

$$T'_j = \begin{cases} T'_{j+1}/(\lfloor T'_{j+1}/T_j \rfloor), & \text{if } j < i \\ T_j, & \text{if } j = i \\ T'_{j-1} \cdot \lfloor T_j/T'_{j-1} \rfloor, & \text{if } j > i \end{cases} \quad (3.7)$$

- Find the optimal primary harmonic task Γ' that minimizes the total task set utilization among all Γ_i where $i = 1, 2, \dots, N$. In other word, $U(\Gamma') = \min_{i=1}^N U(\Gamma'_i)$.

The *RBound* and *CBound* indicate that on a single-core processor, the system utilization as well as the task set schedulability, can be greatly improved if the relationship between task periods can be appropriately exploited.

Existing work (i.e. [73, 66, 47, 48]) has shown that, with appropriate task transformation, using *RBound* and *CBound* can significantly improve the schedulability checking accuracy.

3.1.2 Partitioned Scheduling

Partitioned scheduling is originally derived based on the traditional *bin-packing* technique [106]. By mapping the utilization of a task to the “size” of the object and the utilization bound of a processor to the “capacity” of the bin, people can directly apply the common bin-packing strategies, i.e. *First-Fit* (FF), *Best-Fit* (BF) and

Worst-Fit (WF), to deal with the partitioned multi-core problem. Coffman *et al.* [31] concluded the common approaches based on the traditional bin-packing methods. For example, the FF approach assigns a task immediately to the first processor that can provide enough capacity for it, while the BF (WF) approach always assigns a task to the processor with the largest (smallest) total utilization that still can accommodate that task.

There is several work proposed to study the problem of partitioned multi-core scheduling for fixed-priority periodic real-time tasks [36, 26, 10, 34, 14, 38]. Burchard *et al.* [26] evaluated the partitioned multi-core scheduling under RMS policy by exploiting the traditional bin-packing heuristics, such as FF, BF and WF, with a decreasing order of task utilizations. Andersson *et al.* [10] developed a multi-core scheduling algorithm for fixed-priority periodic tasks, and proved that their proposed algorithm could guarantee the schedulability of any task set with system utilization no more than $1/3$. They also showed that the utilization bound of fixed-priority multi-core scheduling (for both partitioned and global scheduling) was no more than 50% [10, 13]. Lopez *et al.* [87, 88] developed more accurate but complex utilization bounds for multi-core scheduling under RMS by combining the number of processors, the number of tasks and the maximum task utilization into consideration. Later, Darera *et al.* [34] developed a specific utilization bound for partitioned multi-core scheduling under the case of a greedy RMS-based algorithm. Andersson *et al.* [14] introduced a new performance metric, named speed competitive ratio, to measure the performance of partitioned multi-core scheduling under RMS, and based on that new metric, they developed an algorithm with guaranteed schedulability under deterministic processor speedup. Most recently, Fan *et al.* [38] proposed a partitioned multi-core scheduling technique for periodic real-time tasks under RMS policy. They take the characteristic of the relationship between task periods into

the decision of task allocation and thus improved the system resource utilization. In what follows, we first introduce some preliminary concepts for our work.

3.2 Preliminary

The multi-core platform consists of M identical processors, $M \geq 2$, denoted as $\mathcal{P} = \{P_1, P_2, \dots, P_M\}$. The task model considered in this work consists of N sporadic tasks, denoted as $\Gamma = \{\tau_1, \tau_2, \dots, \tau_N\}$. Each task τ_i , where $1 \leq i \leq N$, is characterized by a tuple (C_i, T_i) . C_i is the *worst case execution time* of τ_i , and T_i is the *minimum inter-arrival time* between any two consecutive jobs of τ_i . For the sake of simplicity, we also refer to T_i as the *period* of τ_i . In this work, we assume that Γ is sorted with non-decreasing period order, i.e. for any two tasks $\tau_i, \tau_j \in \Gamma$, $T_i \leq T_j$ if $i < j$. We also use Γ_k to denote the task set on processor P_k .

To ease our presentation, we formally define several concepts as follows.

The *task utilization* of τ_i is denoted as u_i where

$$u_i = \frac{C_i}{T_i} \quad (3.8)$$

The *task set utilization* of Γ is denoted as $U(\Gamma)$ where

$$U(\Gamma) = \sum_{\tau_i \in \Gamma} u_i \quad (3.9)$$

Moreover, let $U(\Gamma_k)$ represent the total utilization of all tasks assigned to P_k .

The *system utilization* of a multi-core platform consisting of a task set Γ and M identical processors is denoted as $U_M(\Gamma)$, where

$$U_M(\Gamma) = \frac{U(\Gamma)}{M} \quad (3.10)$$

The *RBound* [73], as what we have introduced in Section 3.1.1, can be used as a feasibility test method for scheduling fixed-priority periodic tasks on single-core systems. We formally present the RBound feasibility test approach with Theorem 3.2.1.

Theorem 3.2.1. [73] *Given a task set Γ , let Γ' be the task set by scaling all tasks in Γ (i.e. $\forall \tau_i \in \Gamma$) through*

$$\begin{cases} C'_i = C_i \cdot 2^{\lfloor \log \frac{T_{max}}{T_i} \rfloor} \\ T'_i = T_i \cdot 2^{\lfloor \log \frac{T_{max}}{T_i} \rfloor} \end{cases} \quad (3.11)$$

where $T_{max} = \max_{\tau_i \in \Gamma} T_i$. Then Γ is schedulable on a single-core system under RMS if

$$U(\Gamma) \leq RBound(\Gamma') \quad (3.12)$$

The $RBound(*)$ is given by equation (3.4).

From Theorem 3.2.1, we can see that the $RBound$ feasibility test first scales all tasks in Γ with respect to the maximum period, and then predicts the schedulability of Γ by comparing its utilization with the value of $RBound$ under Γ' . In what follows, we present a new task set transformation method, based on which, we then develop a novel partitioned scheduling algorithm.

***CBound* feasibility test**

The $CBound$ [48] is another efficient utilization bound to test the feasibility of periodic tasks by taking harmonic characteristic into consideration. The $CBound$ feasibility test method is formally concluded in Theorem 3.5.3.

Theorem 3.2.2. *Given a task set Γ , let Γ' be a harmonic task set transformed from Γ by DCT method. Then Γ is schedulable on a single-core system under RMS if*

$$U(\Gamma') \leq 1 \quad (3.13)$$

Theorem 3.2.2 shows that by transforming a task set Γ into a harmonic task set Γ' , we can easily predict the feasibility of Γ by check whether the utilization of Γ' is less than or equal to “1”. Note that the $CBound$ feasibility test is different from the

Table 3.1: A task set with six real-time periodic tasks

τ_i	C_i	T_i	u_i
1	1	4	0.25
2	2	8	0.25
3	3	10	0.30
4	8	16	0.50
5	8	20	0.40
6	12	40	0.30

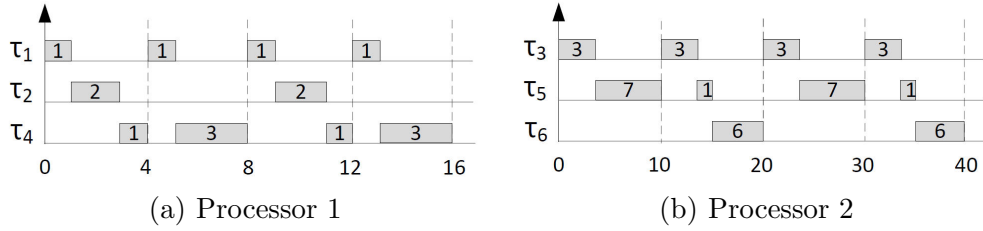


Figure 3.1: Assign tasks in Table 3.1 based on ideal harmonic relationship, and all tasks can be scheduled successfully on two processors.

RBound feasibility test in terms of the way of task set transformation, i.e. *CBound* test only scales the periods while *RBound* test scales both periods and execution times.

3.3 Motivational Examples

Before presenting our approach in detail, we first use two examples to motivate our research. In the first example, we illustrate that exploiting the harmonic relationship can significantly improve the schedulability in multi-core scheduling. In the second example, we demonstrate that we can explore this property for tasks not strictly harmonic.

Consider a multi-core platform with two processors, i.e. $M = 2$, and a task set consisting of six tasks with parameters shown in Table 3.1. When scheduling those six tasks on two processors, it is not difficult to verify that none of the existing

Table 3.2: A task set with four real-time periodic tasks

τ_i	C_i	T_i	u_i
1	4.8	10	0.48
2	5.2	11	0.47
3	5.8	15	0.39
4	9.4	19	0.49

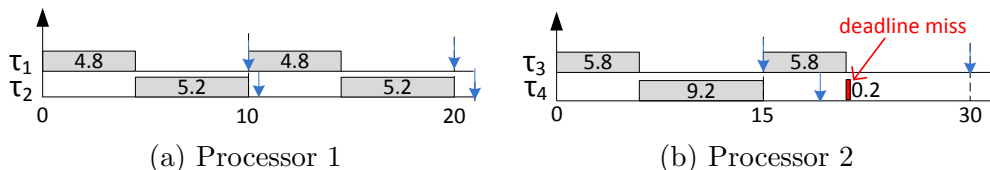


Figure 3.2: Assign tasks in Table 3.2 based on pCOMPACTS [66], while τ_4 missing its deadline.

bin-packing heuristics (e.g. “first-fit”, “best-fit” and “worst-fit”) can successfully schedule the tasks listed in Table 3.1.

Note that, current bin-packing based approaches allocate real-time tasks solely based on their utilization factors and simply ignore other factors such as the task period, which can significantly affect the schedulability of a real-time task. For example, it is a well known fact [71, 48] that a harmonic task set, i.e. the tasks with periods being integer multiples of each other, can have a much higher schedulability than other non-harmonic task sets. If we take this factor into consideration and assign τ_1 , τ_2 and τ_4 to one processor, and τ_3 , τ_5 and τ_6 to another processor, as shown in Figure 3.1(a), the task set in Table 3.1 can be perfectly scheduled on two processors.

Since tasks with ideal harmonic relationship have much higher feasibility on a single-core, one intuitive idea for partitioned multi-core scheduling would therefore be the one to group tasks with ideal harmonic relationship together and assign them to one processor. The question is what if tasks are not exactly harmonic. We use another example to illustrate this scenario.

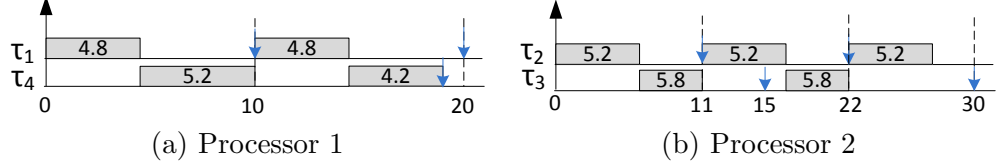


Figure 3.3: Assign tasks in Table 3.2 based on closely harmonic relationship, and all tasks can be scheduled successfully on two processors.

We consider another example to schedule a task set consisting of four tasks as shown in Table 3.2 on two processors. Different from the first example, from Table 3.2, we can see that none of any two tasks are strictly harmonic. Thus, we can not directly taking the harmonic advantage by assigning tasks according to the ideal harmonic relationship, i.e. the dividable period relationship. Once again, it is not difficult to verify that the traditional “first-fit” and “best-fit” approaches are failed in satisfying the timing constraints for all four tasks. Even some most recent partitioning approach with harmonic awareness, i.e. *pCOMPACTS* approach [66], can not successfully guarantee the timing constraints in this example. *pCOMPACTS* measures the harmonic relationship by the distance between periods under the condition of $T_{max}/T_{min} < 2$, and thus first assigns τ_1 and τ_2 to the same processor and then leaves τ_3 and τ_4 running on another processor. This could result in a failure of schedule for τ_4 , see Figure 3.2.

However, by assigning τ_1 and τ_4 to one processor and τ_2 and τ_3 to another processor, as shown in Figure 3.3, we can build a feasible solution for all four tasks. As indicated by this example, although τ_1 and τ_4 have the largest period distance, i.e. $T_4 - T_1 = 19 - 10 = 9$, among all tasks, they looks to be more closely in terms of harmonicity. In fact, by assigning τ_1 and τ_4 to one processor, that local processor can achieve a much high system utilization up to 0.97 ($0.48 + 0.49$), which is very close to the maximum ideal harmonic performance, i.e. 1.

From the above two motivational examples, we can observe that: 1) Taking

advantage of the harmonic relationship can utilize the system processing resource more efficiently; 2) Exploiting the task period relationship can also improve the system utilization, but how to quantify the harmonicity between general tasks is a challenge. In what follows, we first present a task transformation method to improve the *RBound*. We then present two partitioned scheduling algorithms by taking the harmonic relationship among task periods into consideration.

3.4 Task Partition With An Enhanced RBound

In order to apply the *RBound* to test the schedulability of a task set, one key point is to develop an effective and efficient method to transform the task set to a new one such that the ratio of the maximum and minimum period is between 1 and 2 [73]. In addition, we need to guarantee that once the new task set is schedulable, so is the original task set.

To transform a task set, one approach [73] is to fix T_{max} and scale up the rest task periods towards T_{max} so that the maximum/minimum period ratio is between 1 and 2. Another effort [66] is to keep the T_{min} unchanged and scale down task periods such that the maximum/minimum period ratio is between 1 and 2. Unfortunately, as explained before (see Section 3.1.1), this approach cannot guarantee the schedulability of the original task set even though the new task set can be schedulable. In this section, we introduce a new method to scale task periods based on the period of *any task* in the task set, and most importantly, we guarantee that the original task set is schedulable if the new task set is schedulable.

3.4.1 Task Set Scaling (TSS)

Instead of using a restricted transformation, such as scaling the entire task set only with respect to a unique task (i.e. the task with the maximum period), we introduce a more general and flexible task set transformation method, denoted as the *TSS* method, which can scale a task set with respect to the period of an arbitrary task in a given task set.

Algorithm 1 $TSS(\Gamma, \tau_k)$

Require:

```

1)  $\Gamma$ : input task set, sorted with non-decreasing period order;
2)  $\tau_k$ : the  $k^{th}$  task in  $\Gamma$ , based on which the task set is scaled.
1:  $N = |\Gamma|$ ;
2:  $T'_k = Z_k = T_k$ , and  $C'_k = C_k$ ;
3: // step 1: transform LOWER-priority tasks into harmonic;
4: for  $i = k + 1$  to  $N$   $Z_i = Z_{i-1} \cdot \lfloor \frac{T_i}{Z_{i-1}} \rfloor$  end for;
5: // step 2: scale all tasks with respect to  $\tau_k$ ;
6: for  $i = 1$  to  $k - 1$  do
7:    $R_i = 2^{\lfloor \log_2 \frac{T_k}{T_i} \rfloor}$ 
8:    $T'_i = T_i \cdot R_i$ 
9:    $C'_i = C_i \cdot R_i$ 
10: end for
11: for  $i = k + 1$  to  $N$  do
12:    $R_i = Z_i / T_k$ ;
13:    $T'_i = Z_i / R_i$ ;
14:    $C'_i = C_i / R_i$ ;
15: end for
16: return  $\Gamma'$ ;

```

Algorithm 1 shows the details of our proposed *Task Set Scaling* (*TSS*) method. We assume that the input task set Γ is sorted with non-decreasing period order, i.e. for any two tasks τ_i and τ_j , it holds $T_i \leq T_j$ if $i < j$. *TSS* method transforms the entire task set Γ into another task set Γ' by scaling all tasks with respect to τ_k 's period, i.e. T_k , where τ_k is an arbitrary task in Γ .

There are two major steps in Algorithm 1: 1) Tasks with priorities lower than τ_k are transformed into harmonic tasks with their periods being integer multiples of T_k (line 4); 2) Tasks with priorities higher than τ_k are scaled up (line 6-10), and tasks with priorities lower than or equal to τ_k are scaled such that the new period is equal to T_k (line 11-15). After all tasks in Γ are scaled appropriately, the corresponding task set Γ' is returned. In what follows, we discuss the relationship between Γ' and Γ in terms of schedulability.

3.4.2 Feasibility Relationship Between Γ And Γ'

In this subsection, we discuss the relationship between a transformed task set Γ' and its original task set Γ in terms of feasibility. we show that if Γ' is schedulable under RMS, then Γ must be schedulable under RMS. This is essential to the application of our utilization bound in schedulability test.

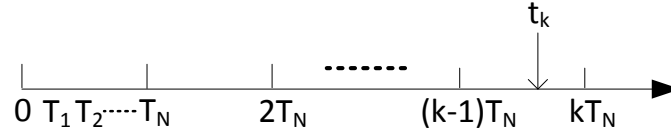


Figure 3.4: Proof of Theorem 3.4.1: given a task set Γ with $T_1 \leq T_2 \leq \dots \leq T_N$ and $\tau_N = (C_N, T_N)$, transform Γ into Γ^* such that $\tau_N^* = (kC_N, kT_N)$ and $\tau_i^* = \tau_i, \forall i < N$.

Theorem 3.4.1. *Given a task set $\Gamma = \{\tau_1, \tau_2, \dots, \tau_{N-1}, \tau_N\}$ with $T_1 \leq T_2 \leq \dots \leq T_N$, let $\Gamma^* = \{\tau_1, \tau_2, \dots, \tau_{N-1}, \tau_N^*\}$, such that $\tau_N^* = (C_N^*, T_N^*)$ satisfies that*

$$C_N^* = k \cdot C_N, \quad T_N^* = k \cdot T_N \quad (3.14)$$

where k is an arbitrary positive integer. If Γ is schedulable on a single-core system under RMS, then Γ^ must be schedulable on a single-core system under RMS.*

Proof: Since the first $N - 1$ tasks always have higher priorities than the N^{th} task in either Γ or Γ^* , their schedulability does not change. Thus, we only need

to prove that τ_N^* is schedulable in Γ^* . Consider the k^{th} instance of task τ_N in Γ . Since Γ is schedulable, we know that the k^{th} instance of task τ_N must be able to meet its deadline. In other words, there must exist a time point t_k , where $t_k \in ((k-1) \cdot T_N, k \cdot T_N]$, such that (see Figure 3.4)

$$\sum_{i=1}^{N-1} C_i \cdot \lceil \frac{t_k}{T_i} \rceil + N \cdot C_N \leq t_k \quad (3.15)$$

According to equation (3.14), we have that $C_N^* = k \cdot C_N$. Thus, the above can be rewritten as

$$\sum_{i=1}^{N-1} C_i \cdot \lceil \frac{t_k}{T_i} \rceil + C_N^* \leq t_k \quad (3.16)$$

The above inequality means that at time point t_k , τ_N^* as well as all other higher priority tasks can completely finish their execution requirements. Note that $t_k \leq k \cdot T_N = T_N^*$. Thus, τ_N^* is schedulable in Γ^* . Therefore, we can see that if Γ is schedulable, then Γ^* must be schedulable. \square

Next, for any given task set Γ , let Γ' be the task set obtained by applying *TSS* method given by Algorithm 1. We prove that the ratio between the maximum and minimum periods of all tasks in Γ' is less than 2. We formally conclude this property in Lemma 3.4.2.

Lemma 3.4.2. *Given a task set Γ sorted with non-decreasing period order and a task τ_k representing the k^{th} task in Γ , let Γ' be the scaled task set obtained by applying TSS method (see Algorithm 1). Then we have*

$$1 \leq \frac{T'_{max}}{T'_{min}} < 2 \quad (3.17)$$

where $T'_{max} = \max_{\tau'_i \in \Gamma'} T'_i$ and $T'_{min} = \min_{\tau'_i \in \Gamma'} T'_i$.

Proof: We prove this property by showing that T'_k (same as T_k) in the transformed task set Γ' is the maximum period and the ratio between T_k and any other period

is less than 2. On one hand, for any task τ_i with priority higher than τ_k , i.e. $i < k$, according to Algorithm 1, we have

$$\frac{T'_k}{T'_i} = \frac{T_k}{T_i \cdot 2^{\lfloor \log \frac{T_k}{T_i} \rfloor}} \quad (3.18)$$

from which we can derive that

$$1 = \frac{T_k}{T_i \cdot 2^{\log \frac{T_k}{T_i}}} \leq \frac{T'_k}{T'_i} < \frac{T_k}{T_i \cdot 2^{(\log \frac{T_k}{T_i} - 1)}} = \frac{2T_k}{T_i \cdot 2^{\log \frac{T_k}{T_i}}} = 2 \quad (3.19)$$

On the other hand, for any task τ_i with priority lower than or equal to τ_k , i.e. $i > k$, according to Algorithm 1, its transformed period can be represented as

$$T'_i = \frac{Z_i}{Z_i/T_k} = T_k \quad (3.20)$$

Based on the above, we can immediately get

$$\frac{T'_k}{T'_i} = \frac{T'_k}{T_k} = 1 \quad (3.21)$$

where $i > k$. Thus far, we show $T'_k(T_k)$ is the maximum period in Γ' , i.e. $\forall i$, $\frac{T'_k}{T'_i} \geq 1$, and the ratio of T'_k over any other period T'_i is less than 2. Therefore, Lemma 3.4.2 is proved. \square

Now we are ready to show that after applying *TSS* method, the schedulability of the original task set Γ can be predicted by that of Γ' . We formulate this property in Theorem 3.4.3

Theorem 3.4.3. *Given a task set Γ , let Γ' be the scaled task set obtained by applying the TSS method with respect to any task τ_k in Γ . If $U(\Gamma') \leq RBound(\Gamma')$, then Γ must be schedulable on a single-core system under RMS.*

Proof: According to Lemma 3.4.2, we have that $\Gamma' = \{\tau'_1, \dots, \tau'_{k-1}, \tau'_k, \tau'_{k+1}, \dots, \tau'_N\}$ satisfies that $1 \leq r < 2$, where r is the ratio between the maximum and minimum

periods in Γ' . Thus, if $U(\Gamma') \leq RBound(\Gamma')$, according to Theorem 3.2.1, Γ' is schedulable on a single-core system under RMS.

Next, for $\forall i > k$, according to line 11-15 in Algorithm 2, we have that

$$T'_i = Z_i/R_i = Z_i/(Z_i/T_k) = T_k = T'_k \quad (3.22)$$

Thus, $\tau'_k, \tau'_{k+1}, \dots, \tau'_N$ have the same as well as the lowest priority in Γ' . Moreover, $\tau'_1, \dots, \tau'_{k-1}$ are tasks with priorities higher than τ'_k before as well as after the transformation. Based on Lemma 2 in work [73], if Γ' is schedulable, we know that the following task set $\widehat{\Gamma}'$ must be schedulable.

$$\widehat{\Gamma}' = \{\tau_1, \dots, \tau_{k-1}, \tau_k, \tau'_{k+1}, \dots, \tau'_N\} \quad (3.23)$$

Then we construct task set Γ^* from $\widehat{\Gamma}'$ by replacing τ'_i with τ_i^* , where $i = k + 1, \dots, N$, such that $T_i^* = Z_i$ (based on line 4 in Algorithm 2) and $C_i^* = C'_i \cdot (T_i^*/T'_i)$.

$$\Gamma^* = \{\tau_1, \dots, \tau_{k-1}, \tau_k, \tau_{k+1}^*, \dots, \tau_N^*\} \quad (3.24)$$

For $i = k, \dots, N - 1$, we have that T_{i+1}^* is an integer multiple of T_i^* , thus according to Theorem 3.4.1, if $\widehat{\Gamma}'$ is schedulable, Γ^* must be schedulable.

Finally, since $T_{k+1}^* \leq \dots \leq T_N^*$ and $T_{k+1} \leq \dots \leq T_N$, thus by extending T_i^* to T_i , where $i = k + 1, \dots, N$, the schedulability of all tasks do not change. In other words, if Γ^* is schedulable, the original task set $\Gamma = \{\tau_1, \dots, \tau_{k-1}, \tau_k, \tau_{k+1}, \dots, \tau_N\}$ must be schedulable.

In sum, after applying the *TSS* method on a given task set Γ , if the scaled task set Γ' satisfies that $U(\Gamma') \leq RBound(\Gamma')$, then Γ is schedulable on a single-core system under *RMS*. \square

3.4.3 Enhanced RBound

In this part, we propose an enhanced utilization bound based on our *TSS* method, and then introduce a new feasibility test method.

First, after the transformation by *TSS*, we can apply the *RBound* function given by equation (3.4) to evaluate the schedulability of the transformed task set, and therefore that of the original task set. By applying *TSS* with different initial tasks, we can possibly attain a higher utilization bound. Subsequently, we derive our *enhanced utilization bound* in the following equation,

$$RBound^{en}(\Gamma) = \max_{\forall \tau_i \in \Gamma} \{RBound(\Gamma'_i) \mid \Gamma'_i = TSS(\Gamma, \tau_i)\} \quad (3.25)$$

where $RBound(*)$ is the utilization bound function given by equation (3.4) and $TSS(*, *)$ is our task set scaling method shown in Algorithm 1.

Next, in light of Theorem 3.4.3, we know that the task set Γ is guaranteed to be schedulable if there exists a task $\tau_i \in \Gamma$ such that the condition $U(\Gamma') \leq RBound(\Gamma')$ is satisfied. The feasibility test method based on our $RBound^{en}$ is concluded with Theorem 3.4.4 in light of Theorem 3.2.1, Lemma 3.4.2 and Theorem 3.4.3.

Theorem 3.4.4. *Given a task set Γ , if $\exists \tau_i, \tau_i \in \Gamma$, such that*

$$U(\Gamma') \leq RBound(\Gamma') \quad (3.26)$$

where $\Gamma' = TSS(\Gamma, \tau_i)$, then Γ is schedulable on a single-core system under RMS.

Theorem 3.4.4 provides a new feasibility test method by applying our proposed *TSS* method to obtain an enhanced RBound to predict the schedulability for a given task set. It is not surprising to see that our proposed feasibility test (given by Theorem 3.4.4) can always outperform the previous RBound feasibility test[73].

Corollary 3.4.5. *Given a task set Γ , if Γ can successfully pass the traditional $RBound$ feasibility test given by Theorem 3.2.1, then Γ must be able to successfully pass the enhance $RBound$ feasibility test given by Theorem 3.4.4.*

Proof: If Γ can pass the traditional $RBound$ feasibility test successfully, according to Theorem 3.2.1, we must have that

$$U(\Gamma) \leq RBound(\Gamma'_1)$$

where Γ'_1 is obtained by using equation (3.11). Note that $U(\Gamma) = U(\Gamma'_1)$. On other hand, let τ_N represent the task with the maximum period in Γ , by using τ_N to our TSS method, we can get a scaled task set, denoted as Γ'_2 . According to TSS method given by Algorithm 1, we have that Γ'_2 is exactly the same as Γ'_1 . Thus, we have

$$U(\Gamma'_2) = U(\Gamma) \leq RBound(\Gamma'_1) = RBound(\Gamma'_2)$$

According to Theorem 3.4.4, we get that Γ is schedulable. Therefore, if Γ can successfully pass the traditional $RBound$ feasibility test given by Theorem 3.2.1, then Γ must be able to successfully pass the enhance $RBound$ feasibility test given by Theorem 3.4.4. \square

From Corollary 3.4.5, we can see that our proposed feasibility test method can always outperform the previous $RBound$ feasibility test method. In fact, the traditional feasibility test condition (given by equation (3.12)) is only one of the conditions tested in our enhance feasibility test. In other words, we proposed feasibility test method completely covers the case of the traditional $RBound$ feasibility test.

For example, consider the following three tasks, $\tau_1 = (7, 10)$, $\tau_2 = (1, 11)$ and $\tau_3 = (1, 15)$. According to the traditional $RBound$ feasibility test (see Theorem 3.2.1), we get that

$$U(\Gamma) = 0.858 > RBound(\Gamma'_1) = 0.783$$

Thus Γ is not schedulable under the traditional $RBound$ test. However, by transforming Γ with respect to τ_2 under our TSS method, i.e. $\Gamma'_2 = TSS(\Gamma, \tau_2)$, we can get the $\Gamma'_2 = \{(7, 10), (1, 11), (1, 11)\}$. Directly, we can derive that

$$U(\Gamma'_2) = 0.882 < RBound(\Gamma'_2) = 0.916$$

According to Theorem 3.4.4, our proposed feasibility test can guarantee that Γ is schedulable on a single-core system under RMS.

3.4.4 The Partitioning Algorithm

In this subsection, we first present a new multi-core scheduling algorithm, *Partitioned Scheduling with Enhanced RBound (PSER)*, then we prove its schedulability after a successful partition.

PSER is a partitioned multi-core scheduling algorithm, which adopts our proposed TSS to make the partitioning decisions for a task set.

We show the details of *PSER* in Algorithm 2. During each iteration, we assign a group of tasks to a core such that the core utilization is maximized and the tasks are deemed to be schedulable according to the $RBound$. The algorithm is terminated when either all the tasks are successfully assigned or a schedulable partition can not be found.

Note that in order to find the best combination of tasks in each iteration, the unassigned task set Γ is transformed with respect to each of tasks in Γ (i.e. line 6). Thus, by exploring all transformations with different initial conditions, we can optimize the grouping decisions and as a result, maximize the system utilization.

After successfully partitioning all tasks by *PSER*, we apply the RMS on each core as the local scheduling policy. We prove that the schedulability of any task set after a successful partitioning by *PSER* can be guaranteed.

Algorithm 2 Partitioned Scheduling with Enhanced RBound (PSER)

Require:

- 1) Task set : $\Gamma = \{\tau_1, \tau_2, \dots, \tau_N\}$;
 - 2) Multi-core : $\mathcal{P} = \{P_1, P_2, \dots, P_M\}$;
 - 1: sort Γ with non-decreasing period order;
 - 2: **for** $m = 1$ to M **do**
 - 3: **if** $\Gamma == \emptyset$ **then** break, **end if**;
 - 4: $U_{opt} = 0$;
 - 5: **for** $i = 1$ to $|\Gamma|$ **do**
 - 6: $\Gamma' = TSS(\Gamma, \tau_i)$;
 - 7: sort Γ' with non-increasing period order (for tasks with same periods, sort them with non-increasing utilization order);
 - 8: $\Gamma'_{sub} = \emptyset$;
 - 9: **for** $j = 1$ to $|\Gamma'|$ **do**
 - 10: **if** $U(\Gamma'_{sub} \cup \{\tau'_j\}) \leq RBound(\Gamma'_{sub} \cup \{\tau'_j\})$ **then**
 - 11: $\Gamma'_{sub} = \Gamma'_{sub} \cup \{\tau'_j\}$;
 - 12: **end if**
 - 13: **end for**
 - 14: **if** $U(\Gamma'_{sub}) > U_{opt}$ **then**
 - 15: $\Gamma_{opt} = \Gamma'_{sub}$;
 - 16: $U_{opt} = U(\Gamma'_{sub})$;
 - 17: **end if**
 - 18: **end for**
 - 19: assign Γ_{opt} to core P_m , and remove Γ_{opt} from Γ ;
 - 20: **end for**
 - 21: **if** $\Gamma = \emptyset$ **then** return “success”; **else** return “failure”, **end if**;
-

Theorem 3.4.6. *If a task set Γ is successfully partitioned by *PSE*R on M cores and scheduled under RMS, then all tasks can meet their deadlines.*

Proof: Assume that a task set Γ is successfully partitioned by *PSE*R, then we prove that each core can guarantee the schedulability of all tasks assigned to it. Consider an arbitrary core $P_m \in \mathcal{P}$, and let Γ_m be the corresponding task set assigned to P_m . Once *PSE*R finishes successfully, according to line 9-17 in Algorithm 2, we know that there must exist $\tau_i \in \Gamma_m$, such that

$$U(\Gamma'_m) \leq RBound(\Gamma'_m) \quad (3.27)$$

where $\Gamma'_m = TSS(\Gamma, \tau_i)$. According to Theorem 3.4.4, Γ_m is schedulable on core P_m under *RMS* policy. Therefore, for an arbitrary core P_m , after the partitioning procedure *PSE*R is successfully completed, all tasks assigned to P_m can meet their deadline. Thus far, this theorem is proved. \square

From Theorem 3.4.6, we can see that any task set successfully partitioned by *PSE*R can be guaranteed to be schedulable under RMS on a multi-core system. In what follows, we will introduce another strategy for partitioned scheduling by exploring the harmonic advantage with *CBound*.

3.5 Harmonic Advantage Exploration With CBound

Instead of scaling each task with respect to both period and execution time, i.e. like TSS (shown in algorithm 1), in this section, we introduce another approach to take the harmonic advantage for multi-core scheduling by scaling and only scaling the periods of all tasks. We first introduce a new metric, called “harmonic index” to quantify the harmonic characteristic among periodic tasks. Then based on that harmonic index, we present our second partitioned scheduling algorithm, i.e. *HAPS*,

by taking the the harmonic relationship into consideration to optimize the system utilization. Finally, we analyze the schedulability of our proposed algorithm *HAPS*.

3.5.1 Quantifying Harmonic Property

Since not all tasks in a given task set are harmonic, it is desirable that we can quantify the harmonicity of a task set. We first introduce the following two concepts.

Definition 3.5.1. *Given a task set $\Gamma = \{\tau_1, \tau_2, \dots, \tau_N\}$ where $\tau_i = (C_i, T_i)$, let $\Gamma' = \{\tau'_1, \tau'_2, \dots, \tau'_N\}$ where $\tau'_i = (C_i, T'_i)$, $T'_i \leq T_i$, and $T'_i | T'_j$ if $i < j$. (Note $a|b$ means “ a divides b ” or “ b is an integer multiple of a ”.) Then Γ' is called a sub harmonic task set of Γ .*

Given task set, there may be infinite numbers of different sub harmonic task sets. There is one type of sub harmonic task sets that is of most interest to us, which we call the *primary harmonic task set* and is formally defined as follows.

Definition 3.5.2. *Let Γ' be a sub harmonic task set of Γ . Then Γ' is called a primary harmonic task set of Γ if there exists no other sub harmonic task set Γ'' such that $T'_i \leq T''_i$ for all $1 \leq i \leq n$.*

We are now ready to define a metric, i.e. the *harmonic index*, to measure the harmonicity of a real-time task set.

Definition 3.5.3. *Given a task set Γ , let $\mathcal{G}(\Gamma)$ represent all the primary harmonic task sets of Γ . Then the harmonic index of Γ , denoted as $\mathcal{H}(\Gamma)$, is defined as*

$$\mathcal{H}(\Gamma) = \min_{\Gamma' \in \mathcal{G}(\Gamma)} (U(\Gamma') - U(\Gamma)). \quad (3.28)$$

From equation (3.28), $\Delta U'$ defines the “distance” of a task set to the corresponding prime harmonic task sets in terms of its total utilization factor.

In this paper, we adopt the *DCT* algorithm [48] to find a primary harmonic task set for any given periodic task set. In the rest of this section, we will present our second partitioned scheduling algorithm *HAPS* by exploiting the harmonic metrics (i.e. harmonic index \mathcal{H}) to make our partitioning decision.

3.5.2 Harmonic Aware Partitioned Scheduling

In this subsection, we introduce our second partitioned scheduling algorithm, namely *Harmonic Aware Partitioned Scheduling (HAPS)*. *HAPS* significantly distinguishes from *PSER*, as well as the bin-packing based scheduling approaches (i.e. First-Fit, Worst-Fit and Best-Fit). Instead of assigning tasks one by one, *HAPS* assigns tasks group by group in order to allocate as more as tasks with closer harmonic relationship together to the same processor.

The basic idea of *HAPS* can be briefly described as below:

- Among all unassigned tasks, for each task τ_i , construct a sub harmonic task set Γ' with respect of T_i .
- Pick up N_i tasks, denoted as Γ_{sub} , from higher harmonic relationship to lower harmonic relationship by maximizing $U(\Gamma_{sub})$ while keeping $U(\Gamma'_{sub}) \leq 1$.
- Find the task group Γ_{opt} among unassigned tasks such that $U(\Gamma_{sub})$ is maximized.
- Allocate Γ_{opt} to an empty processor.

The *HAPS* is described in more details in Algorithm 3. Similar to *PSER*, we denote Γ as the task set containing all unassigned tasks and denote \mathcal{P} as the processor set containing all empty processors. We first sort Γ with non-decreasing order of task period (line 1). Then, when both Γ and \mathcal{P} are not empty, we pick up a group

Algorithm 3 Harmonic Aware Partitioned Scheduling (HAPS)

Require:

- 1) Task set : $\Gamma = \{\tau_1, \tau_2, \dots, \tau_N\}$;
 - 2) Multi-core : $\mathcal{P} = \{P_1, P_2, \dots, P_M\}$;
 - 1: Sort Γ with no-decreasing order of task period;
 - 2: **while** $\Gamma \neq \emptyset$ and $\mathcal{P} \neq \emptyset$ **do**
 - 3: $\Gamma_{opt} = \emptyset$;
 - 4: **for** $i = 1$ to $|\Gamma|$ **do**
 - 5: $T'_i = T_i$
 - 6: **for** $j = i + 1$ to $|\Gamma|$ **do** $T'_j = T'_{j-1} \cdot \lfloor T_j / T'_{j-1} \rfloor$;
 - 7: **for** $j = i - 1$ downto 1 **do** $T'_j = \frac{T'_{j+1}}{\lceil T'_{j+1} / T_j \rceil}$;
 - 8: $\Gamma_{sub} =$ pick up N_i tasks from Γ such that
 - (1) $U(\Gamma'_{sub}) \leq 1$, and $U(\Gamma'_{sub})$ is maximized;
 - (2) $H(\Gamma_{sub})$ is minimized;
 - 9: **if** $U(\Gamma_{sub}) > U(\Gamma_{opt})$ **then**
 - 10: $\Gamma_{opt} = \Gamma_{sub}$;
 - 11: **end if**
 - 12: **end for**
 - 13: Pick up $P_m \in \mathcal{P}$, and assign Γ_{opt} to P_m ;
 - 14: $\Gamma = \Gamma \setminus \Gamma_{opt}$;
 - 15: $\mathcal{P} = \mathcal{P} \setminus P_m$;
 - 16: **end while**
 - 17: **if** $\Gamma = \emptyset$ **then** return “success”; **else** return “fail”, **end if**;
-

of tasks with optimal combination, in terms of harmonic index and total utilization, and allocate them together to one empty processor (from line 2 to line 16). In each iteration of the “while” loop, we first initialize the objective subset of tasks as empty (line 3). The “for” loop (from line 4 to line 12) contains three steps: 1) transforming the task set Γ into a harmonic task set by using the T_i as the harmonic standard; 2) picking up a sub task set, denoted as Γ_{sub} , consisting of N_i tasks with higher harmonic relationship, meanwhile the corresponding task set utilization $U(\Gamma'_{sub})$ is maximized under the constraint of $U(\Gamma'_{sub}) \leq 1$; 3) among all $|\Gamma|$ harmonic transformations, choosing the sub task set that has the maximum utilization in order to optimize the total system utilization. After finding the optimal group of tasks by the “for” loop, assign that sub task set together to an empty processor (line 13). Accordingly, update the unassigned task set by removing the sub task set from Γ (line 14), and update the available processors by removing the occupied one from \mathcal{P} (line 15). The algorithm succeeds if all tasks could be allocated, otherwise, it fails (line 17). In what follows, we conduct further feasibility analysis for this algorithm.

3.5.3 Schedulability Analysis for HAPS

In this section, we discuss the schedulability of our proposed *HAPS* algorithm. We adopt the *RMS* policy as the priority assignment criteria for all tasks assigned to each local processor. We prove that, after successfully partitioning all tasks by *HAPS*, the schedulability of all tasks can be guaranteed under *RMS*.

First, recall that in Section 3.5.1, we define the concept of primary harmonic task set, in which for any two tasks, the period of one can divide or be divided by other. Then we introduce a feasibility test approach for real-time task set on single-core by checking its corresponding primary harmonic task set.

Theorem 3.5.4. [48] *Let Γ' be a primary harmonic task set of Γ . Then Γ is feasible on a single-core processor under RMS if $U(\Gamma') \leq 1$.*

From Theorem , we see that given a task set Γ and its corresponding primary harmonic task set Γ' , if the utilization of Γ' is no greater than 1, then scheduling Γ under *RMS* on a single-core, all tasks can meet their deadlines.

Now we are ready to draw the conclusion of the feasibility of our proposed *HAPS* algorithm. We formally conclude this property in Theorem 3.5.5.

Theorem 3.5.5. *If a task set Γ is successfully partitioned by HAPS on M processors and scheduled under RMS, then all tasks can meet their deadlines.*

Proof: Consider an arbitrary processor P_m , let Γ_m be the task set assigned to P_m . Based on Algorithm 3, we know that all tasks in Γ_m are partitioned together at one time. Moreover, according to line 8 in Algorithm 3, there exists a primary harmonic task set of Γ_m , denote as Γ'_m , such that

$$U(\Gamma'_m) \leq 1$$

According to Theorem 3.5.3, we can see that Γ_m is feasible on processor P_m under *RMS*. Consequently, the task set on each local processor can be successfully scheduled. Therefore, all tasks in Γ can meet their deadlines. \square

HAPS algorithm assigns all tasks group by group instead of one by one as the traditional bin-packing based partitioning algorithms (i.e. First-Fit, Best-Fit and Worst-Fit). Thus, *HAPS* can take the harmonic advantage by globally optimizing the harmonicity among all tasks. In the following section, we will conduct different experiments to evaluate the performance of our proposed scheduling algorithms in terms of task set schedulability and system utilization. In what follows, we use experiments to examine how effective of our proposed algorithms.

3.6 Experiments And Results

In this section, we present a detailed discussion of our experimental evaluations for the proposed partitioned scheduling algorithms. We first introduce the experimental setup used in our evaluation. Then we present two groups of experiments to investigate the performance of our proposed techniques.

3.6.1 Experimental Setup

We conducted two sets of experiments to study the performance of our proposed enhanced utilization bound (in Section 3.4.3) and partitioned scheduling algorithms (in Section 3.4.4 and Section 3.5.2), respectively. The scheduling performance for different approaches were compared by using the *success ratios*, i.e. the number of feasible tasks over the number of total tasks generated under a specific test point. In what follows, we respectively present the results of two group experiments.

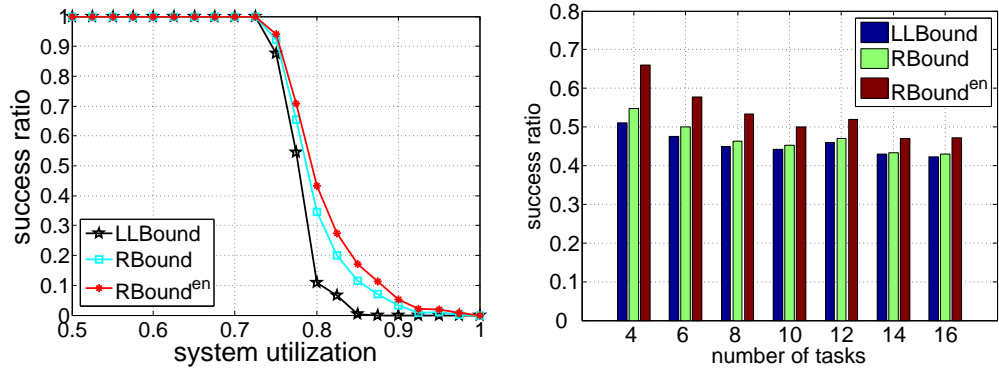
3.6.2 Experiment 1: Efficiency Of Our Enhanced Utilization Bound

In this experiment, we evaluated the efficiency of our enhanced R-Bound (see Section 3.4.3) on a single-core platform. Three different utilization bounds were implemented:

- *LLBound* [81]: Apply the *Liu&Layland's* utilization bound as shown in equation (3.1).
- *RBound* [73]: Calculate the utilization bound by equation (3.4) under the traditional task set transformation method as given by equation (3.11).

- $RBound^{en}$ (our proposed method): Calculate the utilization bound by equation (3.25).

We tested the above three utilization bounds with respect to the system utilization and the number of tasks, respectively. In the first experiment, we varied the system utilization from 0.5 to 1 with an increment of 0.025. In the second experiment, we varied the number of tasks from 4 to 16 with an increment of 2, and the total utilization of all tasks at each test point is randomly generated with $[0.5, 1]$. The task periods are randomly generated within $[10, 500]$. For each testing point, we generated 500 task sets, and the performance was evaluated by using the metric *success ratio*, which is the fraction of the number of feasible task sets over the number of total task sets. The experimental results were collected and plotted in Figure 3.5.



(a) Performance v.s. system utilization (b) Performance v.s. number of tasks

Figure 3.5: Efficiency of our enhanced utilization bound on a single core.

Figure 3.5 shows the performance of three different utilization bounds with respect to system utilization and number of tasks, respectively. From Figure 3.5(a) and 3.5(b), we can observe that our proposed $RBound^{en}$ outperforms the others, i.e. $LLBound$ and $RBound$. For example, in Figure 3.5(a), when system utilization is 0.8, $RBound^{en}$ can achieve a success ratio around 0.49, an improvement of 29%

over $RBound$ (0.38), and an improvement of 2.7 times over $LLBound$ (0.13). In Figure 3.5(b), when the number of tasks is 12, the success ratio of $RBound^{em}$ is 52%, while that ratio of $LLBound$ and $RBound$ are 47% and 46%, respectively.

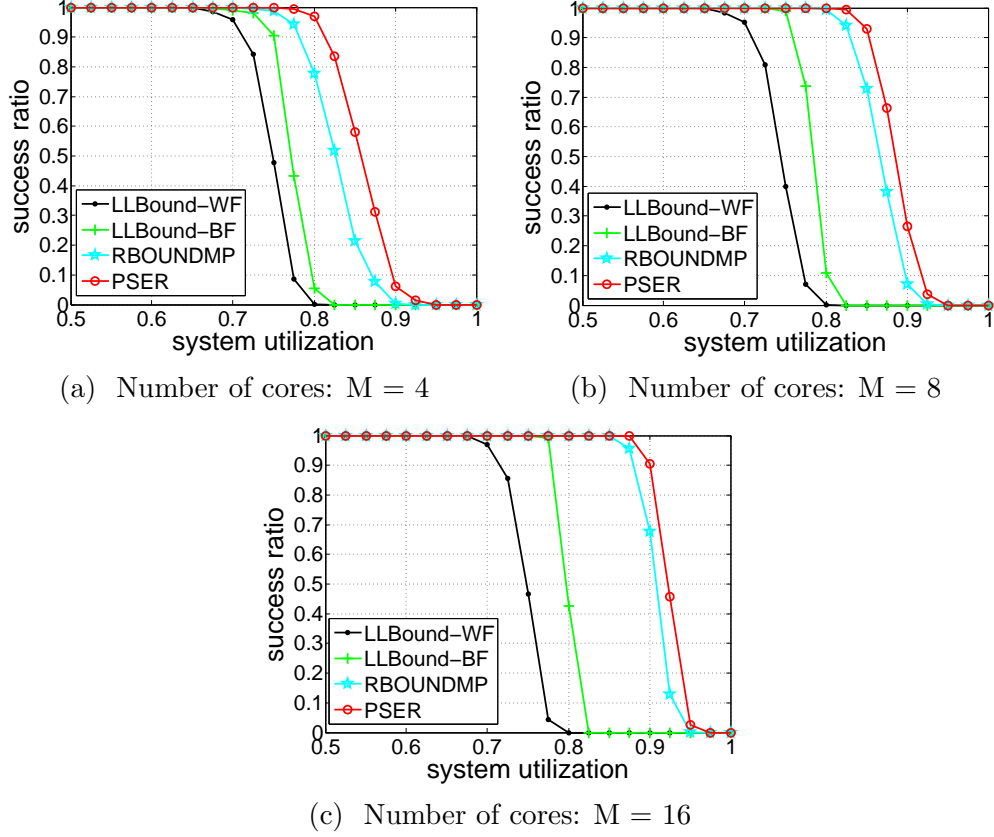


Figure 3.6: Experimental results for light task sets ($u_i \in [0, 0.5]$)

Compared with $RBound$, the improvement of our proposed utilization bound (i.e. $RBound^{en}$) comes from the fact that, instead of choosing only one task period as the task set transformation standard, $RBound^{en}$ takes all periods into consideration, and find the optimal transformation among all task set scalings. Thus our proposed $RBound^{en}$ always outperforms the traditional $RBound$.

3.6.3 Experiment 2: Performance Of Our Partitioned Scheduling Algorithms

In this experiment, we studied the performance differences by different scheduling algorithms under different system utilizations. Six algorithms were implemented in this experiment.

- *WF*: Partitions each task based on the Worst-Fit (WF) bin-packing method (which assigns each task to the core with the largest remaining capacity that can accommodate the task), and checks the capacity of each local core with the *LLBound* (see equation (3.1)).
- *BF*: Partitions each task based on the Best-Fit (BF) bin-packing method (which assigns each task to the core with the smallest remaining capacity that can successfully accommodate that task), and checks the capacity of each local core with the *LLBound*.
- *RBOUNDMP*: Exploits the *RBound* with traditional task set scaling method (see equation (3.11)), and allocates each task based on the Best-Fit strategy under the *RBound*.
- *PSER*: Our first proposed algorithm *PSER* scales the entire task set (including both periods and execution times of all tasks) with respect of each task's period, and then finds the maximal utilization bound among all scaled task sets, and further partitions each task based on the corresponding scaled task set meanwhile maximizes the total utilization of each local processor.
- *HAPS*: Our second proposed algorithm *HAPS* transforms the original task set into a harmonic counterpart by scaling and only scaling the periods of all tasks, and then based on our proposed harmonic index, assigns tasks with

closer harmonic relationship into the same processor to maximize the system utilization.

To study the performance differences among the above scheduling approaches with respect of system utilizations, we conducted two sub-sets of experiments, for light and general task sets, respectively. In light task sets, the utilization of each task was evenly distributed within $[0, 0.5]$, while in general task sets, the utilization of each task was evenly distributed within $[0, 1]$. For each experiment, we varied the system utilization from 0.5 to 1.0 with an increment of 0.025. For both sub-sets of experiments, we tested on different number of processors, i.e. $M = 4, 8$, and 16. The experimental results for all approaches are collected and shown in Figure 3.7 and Figure 3.8.

Figure 3.7 shows the experimental results for task sets containing only light tasks (i.e. $u_i \in [0, 0.5]$). From Figure 3.7, we can observe that *PSER* and *HAPS* can achieve success ratios significantly better than other four approaches. Compared with *PSER* and *HAPS*, all other four approaches, i.e. *WF*, *BF*, *RBOUNDMP* and *pCOMPATS*, can guarantee the feasibility of any task set with utilization below *Liu&Layland's bound*, the same as *PSER* and *HAPS*. The success ratio by *WF* and *BF* drop sharply when system utilization around 0.7. This is because that while *WF* and *BF* can guarantee any task sets with utilizations no more than the *Liu&Layland's bound*, it rejects any task set that cannot pass the feasibility checking condition determined by the *Liu&Layland's* approach. While *RBOUNDMP* and *pCOMPATS* may potentially schedule task sets with utilization higher than the *Liu&Layland's bound*, *PSER* and *HAPS* can achieve higher performance. For example, in Figure 3.7(a), when the system utilization is around 0.85, *PSER* and *HAPS* can respectively achieve a success ratio up to 0.55 and 0.95, while that of *RBOUNDMP* and *pCOMPATS* is around 0.3. We can also see that the performance

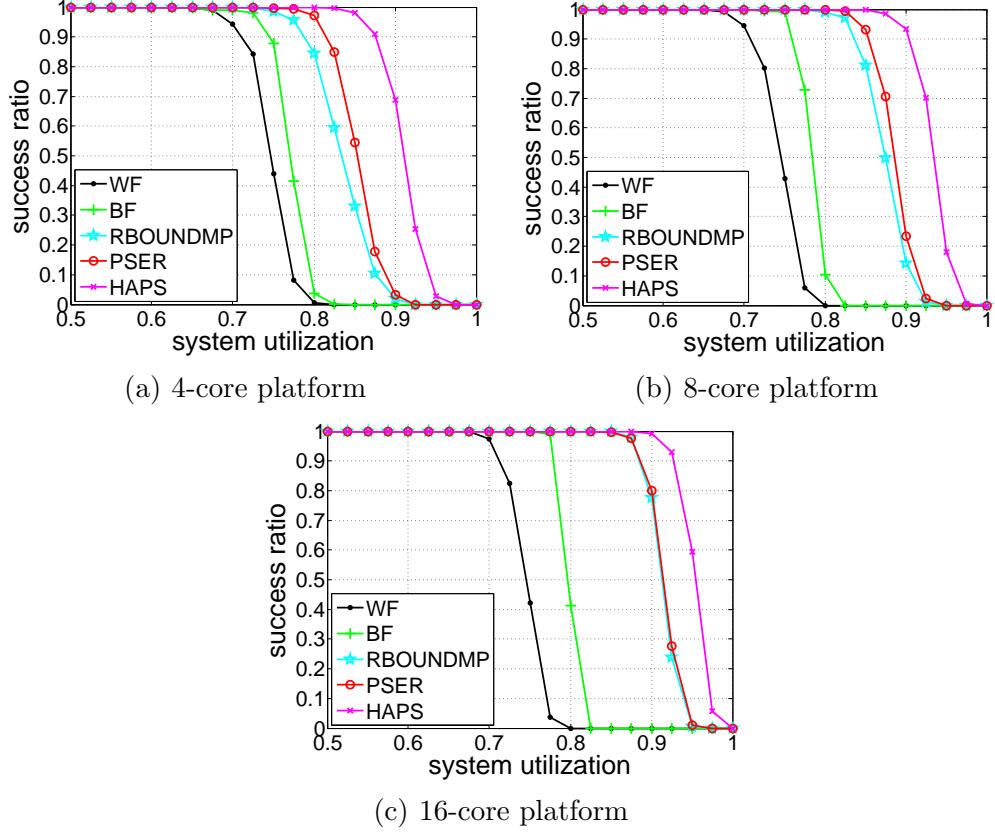


Figure 3.7: Experimental results for light task sets ($u_i \in [0, 0.5]$) by different system utilization

improvement by *PSER* and *HAPS* tends to increase as the number of processors increases. Under the system utilization of 0.9, *PSER* (*HAPS*) can achieve a success ratio of 0.05 (0.7) with 4 processors, 0.25 (0.95) with 8 processors, and increased up to 0.8 (1) with 16 processors.

Figure 3.8 shows our experimental results for general task sets containing both heavy ($u_i \in [0.5, 1]$) and light ($u_i \in [0, 0.5]$) tasks. From Figure 3.8, we can also observe that our proposed algorithms, i.e. *PSER* and *HAPS*, perform better than other four approaches. In Figure 3.8(b), when the system utilization is 0.85, *PSER* (*HAPS*) can achieve a success ratio 5 times (7 times) of that by *WF* and *BF*, and 1.25 times (1.75 times) of that by *RBOUNDMP* and *pCOMPATS*.

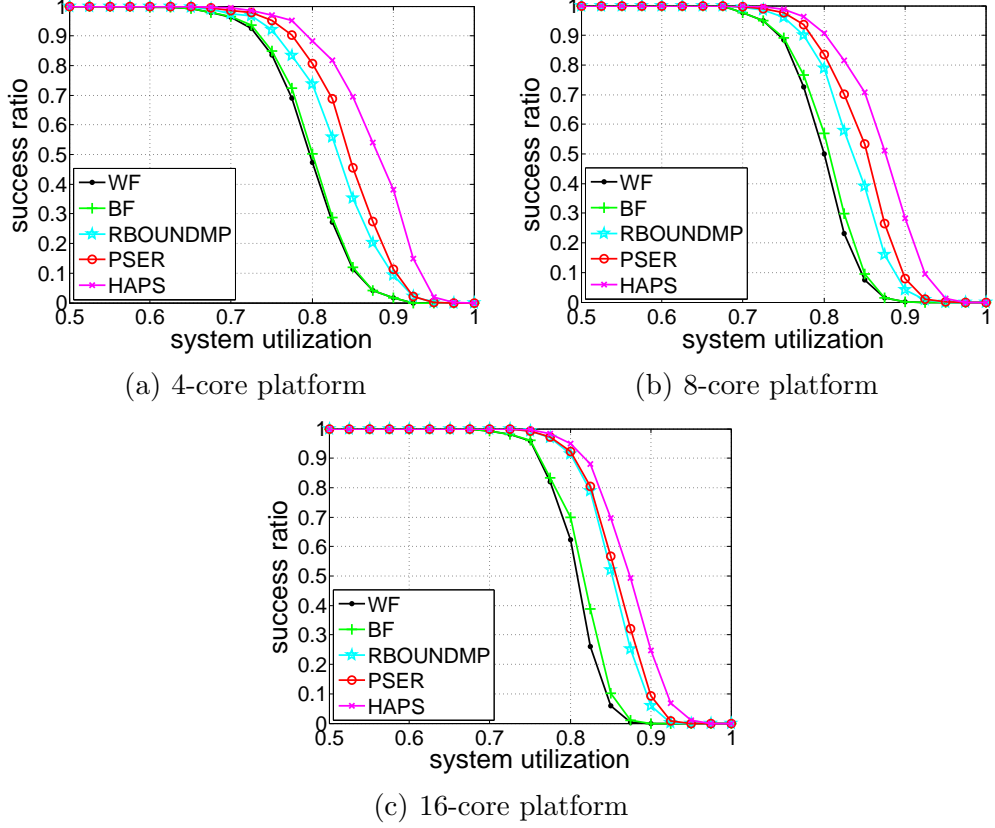


Figure 3.8: Experimental results for general task sets ($u_i \in [0, 1]$) by different system utilization

It is important to observe that for both light and general task sets, our second proposed algorithm (*HAPS*) always outperforms our first proposed algorithm (*PSER*). For example, from Figure 3.7(c) and Figure 3.8(c), we can see that as the number of processors is fixed to 16, *HAPS* achieves better performance than *PSER* when system utilization is greater than 0.85 and 0.75 for light and general task sets, respectively. The reason is that *PSER* takes the harmonic advantage by only considering the relationship among periods of tasks (i.e. see task set scaling(1)), while *HAPS* takes both period and utilization of each task into consideration (i.e. see primary harmonic task set (3.5.2) and harmonic index (3.5.3)). Although by taking the harmonic relationship among periods of tasks can potentially increase the sys-

tem utilization, we cannot consider the period factor isolated in order to optimize the system utilization, specifically for multi-core scheduling. Thus, to better improve the system performance of schedulability by taking the harmonic advantage, we need to appropriately consider not only the relationship among periods but also the utilizations of all tasks.

In summary, our experimental results clearly show, by exploiting the harmonic relationship among tasks appropriately, *PSER* and *HAPS* can significantly improve the schedulability of partitioned scheduling compared with the existing algorithms.

3.7 Summary

Multi-core scheduling problem is the most fundamental problem in real-time embedded system design. Partitioned scheduling, as one of the major types in multi-core scheduling design, becomes more important as the multi-core platform emerging as the dominant technology in both research and industry fields. In this chapter, we have presented two new partitioned approaches (i.e. *PSER* and *HAPS*) for scheduling real-time sporadic tasks on multi-core platform under *RMS*. The *PSER* algorithm first transformed a given task set with respect to each task's period, and then assigned tasks based on their scaled periods under the traditional RBound. The *HAPS* algorithm took the harmonic advantage by transforming the entire task set into a harmonic set, and based on made the partitioning decision according to a efficient utilization bound, i.e. CBound. We formally proved that our scheduling algorithms could guarantee the feasibility of any task set successfully passed the partitioned procedures. Our extensive experimental results demonstrated that the proposed algorithm can significantly improve the scheduling performance compared with previous work.

CHAPTER 4

SEMI-PARTITIONED MULTI-CORE SCHEDULING BY EXPLORING HARMONIC RELATIONSHIP AMONG REAL-TIME PERIODIC TASKS

In the previous chapter, our focus was on partitioned scheduling algorithms. In this chapter, we extended our research work to semi-partitioned scheduling, in which some of the tasks are allowed to migrate among different processing cores. Specifically, we studied the problem on how to guarantee the schedulability of a periodic task set in semi-partitioned scheduling.

4.1 Related Work

In this section, we discuss the related work from two aspects: the work that exploits the harmonic property for periodic tasks and the work on semi-partitioned scheduling.

The property of harmonic tasks, i.e. the tasks with periods being integer multiples of each other, has been widely studied on single-core systems. Compared with the *Liu&Layland's bound*, many researchers have proposed more efficient bound for RMS single-core scheduling. One known result is that if all tasks are harmonic in a task set, the utilization bound can be as high as 1 [83]. Han *et al.* [48] proposed a polynomial-time method to determine the task set schedulability through testing the schedulability of a harmonic task set derived from the original task set. They proved that any task set that can pass the schedulability test by *Liu&Layland's bound* can pass the proposed test. Kuo *et al.* [71] presented another polynomial-time schedulability test method. By combining harmonic tasks into one task, the method can reduce the effective number of tasks and then the *Liu&Layland's bound* can be used to test the schedulability. There are also a number of other researches

that study the relationship between system schedulability and task periods under RMS for single-core scheduling [22, 74, 89]. For multiple processor RMS scheduling, Jung [65] *et al.* studied the problem of scheduling harmonic tasks on a uniform multiprocessor platform. Müller [93] adopted the schedulability test by Han *et al.* [48] to minimize the number of processors, and Fan *et al.* [38] proposed a scheduling technique that improves the system schedulability by taking advantage of the harmonic relation among tasks. All these work indicate that system schedulability can be greatly improved if harmonic relations among different tasks can be appropriately exploited for RMS scheduling on both single and multiple core platforms.

Semi-partitioned scheduling, by splitting a few tasks, has been shown as an effective and practical scheduling method to improve the system utilization significantly compared with the traditional global scheduling and partitioned scheduling (e.g. [8, 67, 11, 39, 72, 69, 43, 20].) As an example, the best known utilization bound for either global or partitioned fixed-priority schedule is no more than 50% [10, 13, 9], while the utilization bound can reach much higher using semi-partitioned scheduling. For instance, Lakshmanan *et al.* [72] have shown an utilization bound of 65%, and Guan *et al.* [43, 44] improved this bound to the traditional *Liu&Layland's bound*, i.e. 69.3% as the number of tasks goes to infinite, or any valid utilization bounds (such as the *K-bound* [71] or *R-bound* [73]) established on single processor platforms. Kandhalu *et al.* [66] proposed two semi-partitioned scheduling algorithms. They show that, for task sets with each individual task utilization factor no more than 0.5, the utilization bound can increase with the number of cores and approach 100%.

We believe that taking advantage of the harmonic relationship among task periods can greatly improve the schedulability of a semi-partitioned algorithm. Some of the existing approaches (such as the ones in [44, 66]) exploit this relationship by

using the *R-Bound* [73], i.e. a utilization bound that takes the possible harmonic relationship into consideration. However, employing *R-bound* cannot determine the schedulability of a task set as accurate as the worst case analysis. Moreover, in order to use *R-bound*, all tasks have to go through a period transformation process. After the transformation, Kandhalu *et al.* [66] proposed to allocate the tasks with the smallest periods together. Unfortunately, these tasks do not necessarily form a task set closest to harmonic. In our approach, we developed a metric to quantitatively measure how harmonic a task set is, and based on this metric, to effectively allocate tasks closer to harmonic to the same processor. In addition, we can still employ the worst case analysis to determine the maximal capacity of a processor when adding a task to it and thus has a much better scheduling performance. The proposed scheduling algorithm can guarantee a utilization bound the same as *Liu&Layland's bound*.

4.2 Preliminary

We are interested in the problem of semi-partitioned scheduling of sporadic tasks on multi-core platforms based on RMS, which is known as an NP-hard problem [106]. In this section, we first present our system models used in this paper, and then we introduce some pertinent background information and concepts necessarily for our research. We then use an example to motivate our research.

4.2.1 System Models

The real-time system considered in this paper consists of N sporadic tasks, denoted as $\Gamma = \{\tau_1, \tau_2, \dots, \tau_N\}$, and executed on M identical processors, i.e. $\mathcal{P} = \{P_1, P_2, \dots, P_M\}$. Each task $\tau_i \in \Gamma$, is characterized by a tuple (C_i, T_i) , where C_i is

the *worst-case execution time* of τ_i , and T_i is the *minimum inter-arrival time* between any two consecutive jobs of τ_i . T_i is also called the *period* of τ_i in this paper. For the sake of simplicity, we use Γ_{P_m} to denote the task set on processor P_m . For the rest of this paper, we make two assumptions: 1) the deadline of each task is equal to its period; 2) Γ is sorted with decreasing priority order, i.e. task τ_i has a higher priority than τ_j if $i < j$. Similarly to Chapter 3, the task utilization of τ_i is defined as $u_i = \frac{C_i}{T_i}$, and the task set utilization of Γ is defined as $U(\Gamma) = \sum_{\tau_i \in \Gamma} u_i$.

We further define the concepts of *light task* and *heavy task* as below:

Definition 4.2.1. Task τ_i is called a *light task* if $u_i \leq \frac{1}{2}$, or a *heavy task* otherwise.

Note that, even though we used the same terminology as that in [43], our definitions of light and heavy tasks are totally different. To simplify our description, let $\Theta(N)$ represent the *Liu&Layland's bound*, i.e. $\Theta(N) = N(2^{1/N} - 1)$.

4.2.2 On Semi-Partitioned Scheduling

A semi-partitioned scheduling algorithm consists of two phases: *the partitioning phase* and *the scheduling phase*.

In the *partitioning phase*, most tasks will be assigned to one processor and can be executed only at that particular processor during running time. These tasks are called *non-split tasks* [43]. A few other tasks, so called *split tasks*, are allowed to be split into several subtasks and assigned to different processors with the purpose of maximally utilizing the processor. Let task τ_i be a task that is split into three subtasks, i.e. $\tau_i^{b_1}$, $\tau_i^{b_2}$ and τ_i^t , executed on processor P_1 , P_2 and P_3 , respectively. The total execution time of $\tau_i^{b_1}$, $\tau_i^{b_2}$ and τ_i^t equals to C_i . Specifically, the last subtask of τ_i , i.e. τ_i^t is called *tail task*, and other subtasks of τ_i , i.e. $\tau_i^{b_1}$ and $\tau_i^{b_2}$, are called *body tasks*. For ease of presentation, we use C_i^B and u_i^B to represent the total execution

time and utilization of all body tasks from a split task τ_i , respectively. Note that, once the partitioning phase is done, the assignment of a subtask to a processor is permanent and the subtask can only run on that designated processor.

In the *scheduling phase*, the scheduling strategy for each processor is determined. In our case, all tasks assigned to the same processor are scheduled strictly conforming to RMS policy, i.e. the task with a smaller period always has a higher priority. One complexity, however, is to execute multiple subtasks assigned to different processors according to the original logical order sequentially. Since the scheduler at the operating system level does not necessarily know the nature of a real-time process, to execute multiple subtasks from the same task concurrently may violate the data or control dependency and thus leads to invalid computing results. Therefore, it is vital to make sure that each subtask is executed according to its logical order and without overlapping with other subtasks.

We adopt an existing approach [69, 43, 42] to solve this problem and assume that an appropriate timer is available to monitor the execution of body/tail tasks. Specifically, the scheduler will assign a timer to a split task, e.g. τ_i in the above example. When τ_i arrives, the scheduler dispatches τ_i^{b1} to processor P_1 immediately and sets the timer to C_i^{b1} . After the timer expires, the scheduler then dispatches τ_i^{b2} to processor P_2 and sets the timer to C_i^{b2} . Then if the timer expires again, the scheduler releases τ_i^t to processor P_3 . As such, all subtasks split from the same task can only run sequentially following their logical orders to ensure the correctness of program. Therefore, the body/tail tasks from the same task can be viewed as tasks with the same periods but different starting times, and the synchronization problem for split tasks from the same task can be easily resolved in practice. For more details about the semi-partitioned scheduling, readers can refer to [43, 69, 72, 68].

4.2.3 Motivation Examples

Table 4.1: A task set with five real-time tasks

τ_i	C_i	T_i	u_i
1	2	6	0.33
2	5	10	0.50
3	3	12	0.25
4	4	20	0.20
5	15	25	0.60

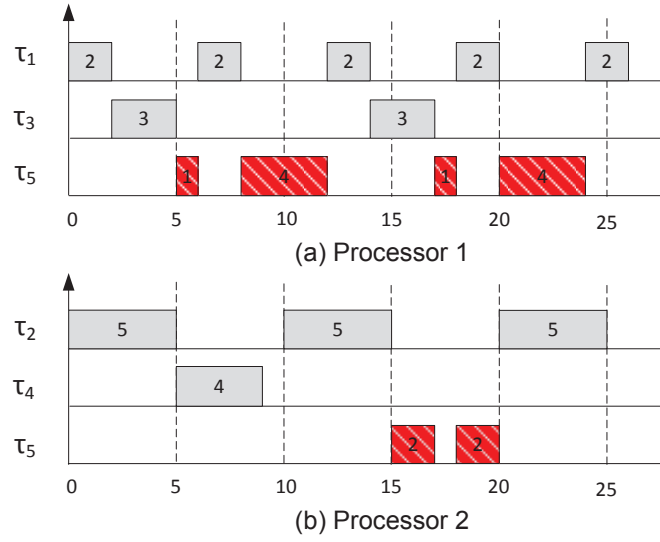


Figure 4.1: Allocation fails when simply grouping harmonic tasks and assigning them to the same processor.

Before we present our approach in detail, we first use an example to motivate our research. Since tasks with harmonic relationship have much higher schedulability on a single processor, an intuitive approach would therefore be the one that groups harmonic tasks together and assigns them to one processor. Unfortunately, such a naive approach may not work in the semi-partitioned approach.

Consider a two-processor platform with a task set shown in Table 4.1. Since τ_1 and τ_3 are harmonic, we can group τ_1 and τ_3 to one processor, i.e. Processor 1. Similarly, we can group τ_2 and τ_4 to the other processor, i.e. Processor 2.

Since no processor can accommodate τ_5 entirely, we have to split τ_5 between these two processors. There are two problems with this assignment. First, as shown in Figure 4.1(a), the maximum capacity that can be accommodated in Processor 1 is 10. Since the subtasks from τ_5 cannot be executed concurrently on two processors, at most 4 time units from Processor 2 can be utilized by τ_5 as shown in Figure 4.1(b). As a result, τ_5 cannot complete before its deadline even if all available time units are used for its execution. Second, in order to use all 4 time units on Processor 2, we need complicated process migration controls and synchronization mechanisms, which increase not only the switching overhead, but also the control complexity among different processors. Note that, if we assign τ_1 and τ_5 to one processor, and the other tasks to another processor, it is not difficult to verify that the schedule is feasible.

As indicated by this example, to take the advantage of harmonic relationship among tasks to improve the schedulability, a critical problem is how to judiciously choose the task to split and to synchronize among different processors. To solve this problem, we present two novel semi-partitioned algorithms, i.e. *HSP-light* and *HSP*, in the following sections.

4.3 The HSP-Light Algorithm

The *HSP-light* algorithm is a harmonic semi-partitioned algorithm developed for light tasks. When employing the harmonic relationship to improve the scheduling performance, it is not necessary that all tasks in the same task set are strictly harmonic. In the rest of this section, we first present a new semi-partitioned algorithm, and then study the schedulability of our proposed algorithm.

Recall that in Chapter 3, we have already defined the harmonic index, i.e. \mathcal{H} . In our proposed algorithm below, we apply \mathcal{H} to quantify the harmonic relationship among different tasks. In what follows, we introduce how we develop the HSP-light algorithm based on this index.

4.3.1 Algorithm Details

HSP-light algorithm assigns tasks to processors from lower priority to higher priority ones. A task is assigned to a processor that can accommodate it and also with the resulting task set having the lowest harmonic index. In other words, a task will be assigned to a feasible processor with the highest harmonic relationship for the resulting task set. The schedulability of the result task set can be guaranteed by performing the exact timing analysis [77] on the corresponding synchronized task set, i.e. assuming all tasks start at the same time. If a task cannot be accommodated entirely by any processor, then split occurs.

To split a task, we adopt a simple heuristic that assigns subtasks to the processor with the highest available capacity. There are two advantages using this splitting strategy: 1) It reduces the total split times by efficiently maximizing the workload for each split subtask. 2) It guarantees the priority of each body task to be the highest one on its host processor. After the split is done, the value to set up the timer for enabling the sub-task is also determined. Algorithm 4 shows the salient aspects of the HSP-light algorithm.

Given a task set Γ and a multiprocessor system \mathcal{P} , HSP-light makes the assignment decision for each task through the “while” loop from line 1 to line 17. Among all unassigned tasks left in Γ , the task τ_i with the lowest priority is selected (line 2). τ_i is assigned to the processor with the minimum harmonic index as long as that

Algorithm 4 HSP-light Algorithm

Require: $\forall \tau_i \in \Gamma, u_i \leq 1/2$;

```
1: while  $\Gamma \neq \emptyset$  do
2:    $\tau_i :=$  the task with the lowest priority in  $\Gamma$ ;
3:    $P_m :=$  the processor with minimum  $\mathcal{H}(\Gamma_{P_m} + \tau_i)$  in  $\mathcal{P}$ ;
4:   if  $\Gamma_{P_m} + \tau_i$  is feasible then
5:     Assign  $\tau_i$  to processor  $P_m$ ;
6:     Continue;
7:   end if
8:    $P_m :=$  the processor with the maximum capacity (greater than 0) for  $\tau_i$ ;
9:   if  $P_m$  does not exist, then break, end if
10:  if  $\Gamma_{P_m} + \tau_i$  is feasible then
11:    Assign  $\tau_i$  to processor  $P_m$ ;
12:  else
13:    Split  $\tau_i$  into  $\tau_{i1}$  and  $\tau_{i2}$  such that  $\Gamma_{P_m} + \tau_{i1}$  can maximally utilize  $P_m$ ;
14:    Assign  $\tau_{i1}$  to processor  $P_m$ ;
15:    Replace  $\tau_i$  by  $\tau_{i2}$ , and move  $\tau_i$  back to  $\Gamma$ ;
16:  end if
17: end while
18: if  $\Gamma = \emptyset$  then
19:   Return “Success!”;
20: else
21:   Return “Fail!”;
22: end if
```

processor has enough capacity for the task on each processor (from line 4 to line 7). If this assignment fails, we split task τ_i and make the assignment (from line 8 to line 16). We choose the processor with the maximum execution capacity for τ_i . If the corresponding capacity is large enough, then τ_i is assigned entirely. Otherwise, we split τ_i and assign part of τ_i to the processor until it is maximally utilized, i.e. no other higher priority tasks can be assigned to that processor without causing other tasks to miss deadlines. Note that, to check the schedulability of a task set (line 4, line 10) and to calculate the maximum execution capacity available for splitting a task (line 13), we can use the traditional exactly timing analysis method [77] on the corresponding synchronized task set, i.e. tasks with the same starting time. The algorithm succeeds if all tasks are allocated, and fails otherwise. In what follows, we further study the schedulability of Algorithm 4.

4.3.2 Schedulability Analysis Of HSP-Light

In this subsection, we are interested in examining how effective the algorithm HSP-light can be when scheduling real-time tasks on multi-core platforms. From the Algorithm 4, it is easy to conclude the following property.

Lemma 4.3.1. *If a task set Γ is successfully partitioned by HSP-light on M processors, then there is at most one body task on each processor; and on all processors, there are at most $(M - 1)$ tasks to be split.*

Proof: In HSP-light, splitting occurs only when no processor can accommodate one task completely. After splitting and assigning a task, the processor that accommodates the body task becomes full for higher priority tasks, and no other higher priority tasks can be assigned to it any more. The body task is the last task assigned

to its host processor. Therefore, there is at most one body task on each processor. Since there are M processors, at most $(M - 1)$ tasks will be split. \square

Lemma 4.3.1 constrains the maximum number of tasks that can be split and migrated among different processors, and thus, the extra cost associated with the migrations. From Lemma 4.3.1, we can derive the following property.

Lemma 4.3.2. *Each body task has the highest priority on its host processor.*

Proof: According to Lemma 4.3.1, we know that there is at most one body task on each processor. Moreover, Algorithm 4 guarantees that any body is the last task assigned to its host processor. Since tasks are assigned from the lowest priority to the highest priority, the priority of any body task is higher than any other tasks on its host processor. \square

More importantly, if a task set can be successfully allocated by HSP-light, all tasks can satisfy their deadlines. The conclusion is formally formulated in the following theorem.

Theorem 4.3.3. *If a task set Γ is successfully partitioned by HSP-light on M processors and scheduled according to RMS, then all tasks can meet their deadlines.*

Proof: For each body task, it has the highest priority at its host processor (Lemma 4.3.2). Therefore, it can always meet its deadline unless the worst case execution time of the original task is larger than its deadline, which is impossible. For tail tasks or any other regular tasks added to a processor, the schedulability of the entire task set is guaranteed based on the worst case response time analysis for the corresponding synchronous task set as stated above (line 4, 10 and 13). \square

From Theorem 4.3.3, HSP-light is not only an allocation method but also can serve as a schedulability test method as well. It is not surprising HSP-light is only a sufficient schedulability test method for multi-core scheduling problem. On the other

hand, however, HSP-light is too complex to be used effectively as a schedulability checking method. In what follows, we present a fast and effective schedulability checking method for our HSP-light algorithm.

4.3.3 Fast Schedulability Checking Method For HSP-Light

Before we introduce our fast and effective schedulability checking method for our proposed HSP-light algorithm, we first study the schedulability of a task set containing a *critical task*, with its formal definition presented in Definition 4.3.4.

Definition 4.3.4. *Let $\Gamma = \{\tau_1, \dots, \tau_i, \dots, \tau_N\}$ be a task set that is schedulable by RMS on a single processor. τ_i is called the critical task if when increasing the execution time of the highest priority task, τ_i is the first task to miss its deadline.*

In addition, for ease of presentation, we introduce the following definition.

Definition 4.3.5. *A processor is called to be maximally utilized by a task set if any increase of the execution time for its highest priority task will cause at least one task on the same processor to miss its deadline.*

In a semi-partitioned system, after partitioning, we divide the tasks into three types: non-split task, body task and tail task. According to Lemma 4.3.2, a body task always has the highest priority on its host processor. Thus, from Definition 4.3.4, no body task can be a critical task.

Lemma 4.3.6. *The critical task on each processor can only be a non-split task or a tail task.*

In what follows, we want to study the schedulability characteristics for processors containing non-split or tail tasks that are critical tasks. We assume that a split task

τ_i is split into B_i body tasks and one tail task, denoted as $\tau_i^{b_j}$ ($j \in [1, B_i]$) and τ_i^t , respectively.

For two different types of critical tasks, i.e. non-split tasks and tail tasks, we introduce two important properties, which are formulated in the following lemmas.

Lemma 4.3.7. *Let Γ_{P_m} be the task set allocated to processor P_m in HSP-light. If the critical task is a non-split task and P_m is maximally utilized by Γ_{P_m} , then $U(\Gamma_{P_m}) > \Theta(N)$.*

Proof: By contradiction. Assume that processor P_m is *maximally utilized* by Γ_{P_m} but

$$U(\Gamma_{P_m}) \leq \Theta(N) \quad (4.1)$$

Let N_m denote the number of tasks on P_m , and let a non-split task τ_j be the critical task on P_m . Then we know that $N_m < N$. Since $\Theta(N)$ is a monotonically decreasing function with respect to N , we have $\Theta(N) < \Theta(N_m)$. According to our assumption in equation (4.1), we get

$$U(\Gamma_{P_m}) \leq \Theta(N) < \Theta(N_m)$$

Note that Γ_{P_m} may contain some tail tasks with deadlines less than their periods. Given Γ_{P_m} , we can always construct another Γ'_{P_m} such that any tail task in Γ'_{P_m} has its deadline equal to its original period. As such, we have

$$U(\Gamma'_{P_m}) = U(\Gamma_{P_m}) \leq \Theta(N) < \Theta(N_m).$$

Also, since τ_j is a non-split critical task, processor P_m is also *maximally utilized* by Γ'_{P_m} .

Now consider the critical task τ_j . Let us keep its period (T_j) the same, but increase its execution time such that

$$\Delta u_j = \min(\Theta(N_m) - U(\Gamma_{P_m}), 1 - u_j)$$

After the above transformation, the new utilization on P_m , denoted as $U(\Gamma''_{P_m})$ still satisfies that $U(\Gamma''_{P_m}) \leq \Theta(N_m)$, which implies that Γ''_{P_m} is feasible by RMS on processor P_m even though τ_j 's execution time increases. This contradicts that P_m has been *maximally utilized* by Γ'_{P_m} and τ_j is the critical task. \square

Lemma 4.3.8. *Let Γ_{P_m} be the task set allocated to processor P_m in HSP-light. If the critical task is a tail task and P_m is maximally utilized by Γ_{P_m} , then $U(\Gamma_{P_m}) > \Theta(N)$.*

Proof: Let τ_i^t be the critical tail task on P_m . To simplify the description below, let U_X^t (U_Y^t) denote the total utilization of tasks with priorities higher (lower) than τ_i on P_m (see Figure 4.2.) From HSP-light, the processor containing the first body task $\tau_i^{b_1}$ of τ_i has the largest capacity to accommodate τ_i . Thus, we have

$$U_Y^t + u_i^{b_1} \geq \Theta(N).$$

Otherwise, $\tau_i^{b_1}$ would be assigned to P_m instead. Moreover, since τ_i is a light task, we have that $u_i^{b_1} < u_i \leq 1/2$, from the above inequality we can derive that

$$U_Y^t > \Theta(N) - \frac{1}{2}. \quad (4.2)$$

On the other hand, for processor P_m , since τ_i^t is the critical task, there will be no idle time within interval $[0, T_i - C_i^B]$, where C_i^B is the total execution time of τ_i 's body tasks. Therefore, for τ_i^t and all higher priority tasks on P_m , we have

$$\sum_{j < i} C_j \left\lceil \frac{T_i - C_i^B}{T_j} \right\rceil + C_i^t \geq T_i - C_i^B \quad (4.3)$$

Divide $(T_i - C_i^B)$ on both side of the above, we can get that

$$\sum_{j < i} u_j \cdot \left\lceil \frac{T_i - C_i^B}{T_j} \right\rceil \cdot \frac{T_j}{T_i - C_i^B} + u_i^t \cdot \frac{T_i}{T_i - C_i^B} \geq 1$$

Split the sum of the above into two parts, and rewrite as

$$\begin{aligned}
& \sum_{j < i, T_j < T_i - C_i^B} u_j \cdot \left\lceil \frac{T_i - C_i^B}{T_j} \right\rceil \cdot \frac{T_j}{T_i - C_i^B} + \\
& \sum_{j < i, T_j \geq T_i - C_i^B} u_j \cdot \left\lceil \frac{T_i - C_i^B}{T_j} \right\rceil \cdot \frac{T_j}{T_i - C_i^B} + \\
& u_i^t \cdot \frac{T_i}{T_i - C_i^B} \geq 1
\end{aligned} \tag{4.4}$$

For the first part on the left side of equation (4.4), since $\left\lceil \frac{T_i - C_i^B}{T_j} \right\rceil \leq \frac{T_i - C_i^B}{T_j} + 1$, we can derive that

$$\begin{aligned}
& \sum_{j < i, T_j < T_i - C_i^B} u_j \cdot \left\lceil \frac{T_i - C_i^B}{T_j} \right\rceil \cdot \frac{T_j}{T_i - C_i^B} \\
& \leq \sum_{j < i, T_j < T_i - C_i^B} u_j \cdot \left(1 + \frac{T_j}{T_i - C_i^B}\right)
\end{aligned}$$

Moreover, in the above, since $T_j < T_i - C_i^B$, we have $\frac{T_j}{T_i - C_i^B} < 1$. Then we can further derive

$$\sum_{j < i, T_j < T_i - C_i^B} u_j \cdot \left\lceil \frac{T_i - C_i^B}{T_j} \right\rceil \cdot \frac{T_j}{T_i - C_i^B} \leq \sum_{j < i, T_j < T_i - C_i^B} 2 \cdot u_j \tag{4.5}$$

For the second part on the left side of equation (4.4), since $T_j \geq T_i - C_i^B$, we have $\left\lceil \frac{T_i - C_i^B}{T_j} \right\rceil = 1$. Thus we can derive

$$\sum_{j < i, T_j \geq T_i - C_i^B} u_j \cdot \left\lceil \frac{T_i - C_i^B}{T_j} \right\rceil \cdot \frac{T_j}{T_i - C_i^B} = \sum_{j < i, T_j \geq T_i - C_i^B} u_j \cdot \frac{T_j}{T_i - C_i^B} \tag{4.6}$$

And further since $u_i^B < \frac{1}{2}$, then

$$\frac{T_j}{T_i - C_i^B} \leq \frac{T_i}{T_i - C_i^B} < 2, \text{ if } T_j \geq T_i - C_i^B. \tag{4.7}$$

Put equation (4.7) into (4.6), we can derive

$$\sum_{j < i, T_j \geq T_i - C_i^B} u_j < \sum_{j < i, T_j \geq T_i - C_i^B} 2 \cdot u_j \tag{4.8}$$

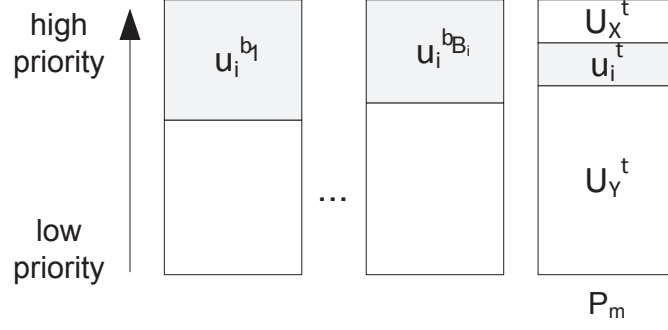


Figure 4.2: Illustration of U_X^t and U_Y^t .

For the third part on the left side of equation (4.4), by applying equation (4.7), we have

$$u_i^t \cdot \frac{T_i}{T_i - C_i^B} < 2 \cdot u_i^t \quad (4.9)$$

Apply equation (4.5), (4.8) and (4.9) into (4.4), we can get

$$\sum_{j < i, T_j < T_i - C_i^B} u_j + \sum_{j < i, T_j \geq T_i - C_i^B} u_j + u_i^t > \frac{1}{2}$$

or

$$U_X^t + u_i^t > \frac{1}{2} \quad (4.10)$$

Finally, sum up equation (4.2) and (4.10), and replace $(U_Y^t + U_X^t + u_i^t)$ by $U(\Gamma_{P_m})$, we obtain that

$$U(\Gamma_{P_m}) > \Theta(N)$$

□

Based on Lemma 4.3.7 and Lemma 4.3.8, we can derive the following property.

Lemma 4.3.9. *If all processors in \mathcal{P} are maximally utilized according to HSP-light, then we have*

$$\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) > |\mathcal{P}| \cdot \Theta(N) \quad (4.11)$$

Proof: Let \mathcal{P}^A denote the processors with critical tasks as non-split tasks, and \mathcal{P}^B denote the processors with critical tasks as tail tasks. According to Lemma 4.3.6, we have that $\mathcal{P} = \mathcal{P}^A \cup \mathcal{P}^B$ and $\mathcal{P}^A \cap \mathcal{P}^B = \emptyset$. Thus, we have

$$\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) = \sum_{P_m \in \mathcal{P}^A} U(\Gamma_{P_m}) + \sum_{P_m \in \mathcal{P}^B} U(\Gamma_{P_m}) \quad (4.12)$$

Moreover, for any $P_m \in \mathcal{P}^A$ or \mathcal{P}^B , from Lemma 4.3.7 and Lemma 4.3.8, we know that $U(\Gamma_{P_m}) > \Theta(N)$. Applying this to the above equation, we get

$$\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) > |\mathcal{P}^A| \cdot \Theta(N) + |\mathcal{P}^B| \cdot \Theta(N) \quad (4.13)$$

or

$$\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) > |\mathcal{P}| \cdot \Theta(N) \quad (4.14)$$

□

We are now ready to present our schedulability checking method, concluded in Theorem 4.3.10, to quickly and effectively predict the feasibility of any periodic task set scheduled by HSP-light algorithm.

Theorem 4.3.10. *Given a light task set Γ consisting of N tasks to be scheduled on M processors, if*

$$U_M(\Gamma) \leq \Theta(N), \quad (4.15)$$

then Γ is feasible by HSP-light under RMS.

Proof: By contradiction. Assume that Γ is not feasible by HSP-light, thus we know every processor is *maximally utilized*.

From the given condition (equation (4.15)) we have that

$$U(\Gamma) \leq M \cdot \Theta(N), \quad (4.16)$$

On the other hand, since all processors are *maximally utilized*, according to Lemma 4.3.9, we know

$$\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) > |\mathcal{P}| \cdot \Theta(N)$$

Since $|\mathcal{P}| = M$, the above can be rewritten as

$$\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) > M \cdot \Theta(N) \quad (4.17)$$

This contradicts equation (4.16). \square

Theorem 4.3.10 shows that a light task set with system utilization bounded by the well-known *Liu&Layland's bound* is guaranteed to be feasible using our proposed approach, i.e. Algorithm 4.

It is worth mentioning that Theorem 4.3.3 is valid for any general task set, which implies that if a task set can be successfully allocated using HSP-light, all tasks can meet their deadlines. However, Theorem 4.3.10 works only for light task sets. In other word, HSP-light cannot guarantee the schedulability of a general task set (which contains heavy tasks), even if its total utilization is less than *Liu&Layland's bound*. In the next section, we introduce a more advanced algorithm, i.e. *HSP*, that can guarantee the schedulability for any task sets with system utilizations no more than the utilization bound.

4.4 The HSP Algorithm

The reason that HSP-light cannot guarantee the schedulability of an arbitrary task set with utilization lower than the utilization bound is that, if a split task is a heavy task and the tail task is very *light*, the overall system utilization can be very low. We use an example to explain this observation.

Table 4.2: A task set with four real-time tasks

τ_i	C_i	T_i	u_i
1	2	50	0.04
2	49	50	0.98
3	4	90	0.044
4	4	100	0.04

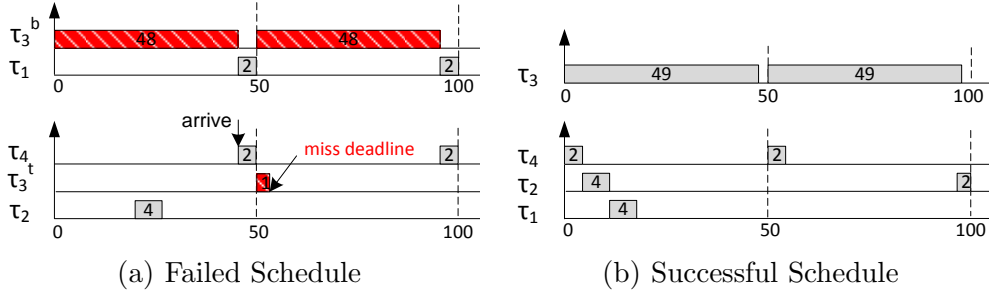


Figure 4.3: (a) The task set is failed to be scheduled according to HSP-light; (b) The task set is schedulable if the heavy task τ_2 is pre-assigned.

Consider to schedule a task set with four tasks, as shown in Table 4.2, on 2 processors. As shown in Figure 4.3(a), even though the system utilization is very small, i.e. $(2/50 + 49/50 + 4/90 + 4/100)/2 = 0.55 < 0.69$, HSP-light cannot schedule this task set successfully. Note that the tail task from τ_2 can be viewed as a task with worst case execution time of 1 and deadline of 2. Adding any higher priority task with execution time more than 1 will make τ_2 infeasible. On the other hand, if we pre-assign the heavy task τ_2 to a processor, we can see that the task set can be successfully scheduled as shown in Figure 4.3(b). Therefore, in order to take the advantage of harmonic property to schedule general task sets, a special operation, i.e. the pre-assignment, needs to be performed for heavy tasks.

4.4.1 Algorithm Details

As discussed before, HSP-light can guarantee all tasks (light or heavy) meet their deadlines if all tasks can be assigned to a processor successfully. At the same time, Figure 4.3 implies that heavy task pre-assignment can greatly improve the schedulability of the scheduling algorithm. The question becomes which heavy tasks should be pre-assigned and how other tasks should be assigned accordingly.

In HSP, the pre-assignment for heavy tasks follows the same strategy as introduced in [43]. Specifically, for any heavy task τ_i , let \mathcal{P}_i^{Emp} denote the set of empty processors before τ_i 's assignment and $|\mathcal{P}_i^{Emp}|$ denote the number of processors in this set. Then a heavy task τ_i needs to be pre-assigned to an empty processor if

$$\sum_{j>i} u_j \leq (|\mathcal{P}_i^{Emp}| - 1) \cdot \Theta(N). \quad (4.18)$$

The detailed procedure of HSP is shown in Algorithm 5. HSP is very similar to HSP-light, except for two important differences:

- At the beginning of semi-partitioning procedure, heavy tasks are pre-assigned to empty processor set, denoted as \mathcal{P}^{Pre} , if they satisfy the criteria as stated in equation (4.18) (from line 1 to line 8);
- To ensure that a body task always has the highest priority on a processor, a processor with heavy task pre-assignment may be excluded from the semi-partitioning process. According to Algorithm 5, a task can be assigned to a processor with heavy task assignment only after the heavy task pre-assigned in the processor has a lower priority (from line 12 to line 15).

Algorithm 5 HSP Algorithm

Require:

```
1) Task set :  $\Gamma = \{\tau_1, \tau_2, \dots, \tau_N\}$ ;  
2) Multiprocessor :  $\mathcal{P} = \{P_1, P_2, \dots, P_M\}$ ;  
1: // pre-assign heavy tasks;  
2:  $\mathcal{P}^{Pre} = \emptyset$ ;  
3: for  $i = 1$  to  $N$  do  
4:   if  $u_i > 1/2$  and  $\sum_{j>i} u_j \leq (|\mathcal{P}_i^{Emp}| - 1) \cdot \Theta(N)$  then  
5:     Assign  $\tau_i$  to processor  $P_m$ , where  $m = |\mathcal{P}|$ ;  
6:     Move  $P_m$  from  $\mathcal{P}$  to  $\mathcal{P}^{Pre}$ ;  
7:   end if  
8: end for  
9: // assign other tasks;  
10: while  $\Gamma \neq \emptyset$  do  
11:    $\tau_i :=$  the task with the lowest priority in  $\Gamma$ ;  
12:    $\tau_j :=$  the task with the lowest priority in  $\Gamma_{\mathcal{P}^{Pre}}$ ;  
13:   if  $\tau_i$  has higher priority than  $\tau_j$  then  
14:     Move  $P(\tau_j)$  from  $\mathcal{P}^{Pre}$  to  $\mathcal{P}$ ;  
15:   end if  
16:    $P_m :=$  the processor with minimum  $\mathcal{H}(\Gamma_{P_m} + \tau_i)$  in  $\mathcal{P}$ ;  
17:   if  $\Gamma_{P_m} + \tau_i$  is feasible then  
18:     Assign  $\tau_i$  to processor  $P_m$ ;  
19:     Continue;  
20:   end if  
21:    $P_m :=$  the processor with maximum capacity for  $\tau_i$  in  $\mathcal{P}$ ;  
22:   if  $P_m$  does not exist, then Break, end if  
23:   if  $\Gamma_{P_m} + \tau_i$  is feasible then  
24:     Assign  $\tau_i$  to processor  $P_m$ ;  
25:   else  
26:     Split  $\tau_i$  into  $\tau_{i1}$  and  $\tau_{i2}$  such that  $\Gamma_{P_m} + \tau_{i1}$  can maximally utilize  $P_m$ ;  
27:     Assign  $\tau_{i1}$  to processor  $P_m$ ;  
28:     Replace  $\tau_i$  by  $\tau_{i2}$ , and move  $\tau_i$  back to  $\Gamma$ ;  
29:   end if  
30: end while  
31: if  $\Gamma = \emptyset$  then  
32:   Return “success”;  
33: else  
34:   Return “fail”;  
35: end if
```

4.4.2 Schedulability Analysis Of HSP

First, similar to Theorem 4.3.3, for HSP, the schedulability of tasks are guaranteed as stated in the following theorem.

Theorem 4.4.1. *If a task set Γ is successfully partitioned by HSP on M processors and scheduled according to RMS, then all tasks can meet their deadlines.*

Next, two important observations, similar to that in Lemma 4.3.7 and Lemma 4.3.8, are also true and formulated in the following two lemmas.

Lemma 4.4.2. *Let Γ_{P_m} be the task set allocated to processor P_m in HSP. If the critical task is a non-split task and P_m is maximally utilized, then $U(\Gamma_{P_m}) > \Theta(N)$.*

Lemma 4.4.3. *Let Γ_{P_m} be the task set allocated to processor P_m in HSP. If the critical task is a tail task from a light task and P_m is maximally utilized, then $U(\Gamma_{P_m}) > \Theta(N)$.*

Lemma 4.4.2 and Lemma 4.4.3 can be proved in the same way as that for Lemma 4.3.7 and Lemma 4.3.8. Moreover, if a tail task from a heavy task is the critical task, we have a very important observation which is formulated in the following lemma.

Lemma 4.4.4. *Let Γ_{P_k} be the task set allocated to processor P_k in HSP. If the critical task is a tail task from a heavy task τ_i and P_k is maximally utilized, then*

$$\sum_{P_m \in \mathcal{P}^R} U(\Gamma_{P_m}) > |\mathcal{P}^R| \cdot \Theta(N) \quad (4.19)$$

where $\mathcal{P}^R = \{P(\tau_j) | j \in [i, N]\}$.

Proof: For all tasks assigned to processors in \mathcal{P}^R , we divide them into two groups: 1) tasks with priorities lower than τ_i , denoted as Γ^Y , 2) tasks with priorities equal

or higher than τ_i , denoted as Γ^X . Then we have

$$\sum_{P_m \in \mathcal{P}^R} U(\Gamma_{P_m}) = \sum_{\tau_j \in \Gamma^Y} u_j + \sum_{\tau_j \in \Gamma^X} u_j \quad (4.20)$$

On one hand, since τ_i is heavy but not pre-assigned, according to equation (4.18), we have

$$\sum_{\tau_j \in \Gamma^Y} u_j > (|\mathcal{P}^R| - 1) \cdot \Theta(N) \quad (4.21)$$

Since τ_i^t is the critical task on its host processor P_m , there will be no idle time within interval $[0, T_i - C_i^B]$. Therefore, for τ_i^t and all higher priority tasks on P_m , we have

$$\sum_{j < i, \tau_j \in \Gamma_{P_m}} C_j \left\lceil \frac{T_i - C_i^B}{T_j} \right\rceil + C_i^t \geq T_i - C_i^B$$

or

$$\sum_{j < i, \tau_j \in \Gamma_{P_m}} u_j \left\lceil \frac{T_i - C_i^B}{T_j} \right\rceil \cdot \frac{T_j}{T_i} + u_i \geq 1 \quad (4.22)$$

Note that 1) $T_j \leq T_i$ for $j < i$, 2) and $T_i - C_i^B \leq \frac{1}{2}T_i$, since $u_i^B \geq \frac{1}{2}$. By putting them into the above, we can derive

$$\sum_{j < i, \tau_j \in \Gamma_{P_m}} u_j + u_i \geq 1 \quad (4.23)$$

Therefore, for all tasks in Γ^X we have

$$\sum_{\tau_j \in \Gamma^X} u_j \geq 1 \quad (4.24)$$

Finally, apply equation (4.24) and (4.21) into (4.20), since $\Theta(N) \leq 1$, we get

$$\sum_{P_m \in \mathcal{P}^R} U(\Gamma_{P_m}) > |\mathcal{P}^R| \cdot \Theta(N)$$

□

Lemma 4.4.5. *If a system is maximally utilized through HSP, then for all processors in \mathcal{P} , we have*

$$\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) > |\mathcal{P}| \cdot \Theta(N) \quad (4.25)$$

Proof: Select the heavy task, i.e. τ_i , that is not pre-assigned and has the highest priority among the ones with its tail task being the critical task on its host processor. Let \mathcal{P}^A denote the processors to which τ_i and other lower priority tasks are assigned. Let \mathcal{P}^B denote the rest of processors besides \mathcal{P}^A . From Lemma 4.4.4 we know that

$$\sum_{P_m \in \mathcal{P}^A} U(\Gamma_{P_m}) > |\mathcal{P}^A| \cdot \Theta(N) \quad (4.26)$$

From Lemma 4.4.2 and Lemma 4.4.3, we have that

$$\sum_{P_m \in \mathcal{P}^B} U(\Gamma_{P_m}) > |\mathcal{P}^B| \cdot \Theta(N) \quad (4.27)$$

Sum up equation (4.26) and (4.27), since $\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) = \sum_{P_m \in \mathcal{P}^A} U(\Gamma_{P_m}) + \sum_{P_m \in \mathcal{P}^B} U(\Gamma_{P_m})$, we can derive

$$\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) > |\mathcal{P}| \cdot \Theta(N) \quad (4.28)$$

□

Finally, with the above conclusions, we can easily derive a schedulability checking method for HSP algorithm, which is similar to Theorem 4.3.10, by applying the *Liu & Layland's bound*. This conclusion is formally formulated in Theorem 4.4.6.

Theorem 4.4.6. *Given a task set Γ consisting of N tasks to be scheduled on M processors, if*

$$U_M(\Gamma) \leq \Theta(N), \quad (4.29)$$

then Γ is feasible by HSP under RMS.

Proof: By contradiction. Assume that Γ is not feasible by HSP. With equation (4.29), we have

$$U(\Gamma) \leq M \cdot \Theta(N), \quad (4.30)$$

Since all processors are *maximally utilized*, from Lemma 4.4.5, we have that

$$\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) > |\mathcal{P}| \cdot \Theta(N) \quad (4.31)$$

or

$$\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) > M \cdot \Theta(N) \quad (4.32)$$

This contradicts equation (4.30). \square

Theorem 4.4.6 provides a very efficient schedulability checking method for real-time task sets scheduled by HSP. Given any task set Γ , if the total utilization of Γ satisfies equation (4.29), then Γ can be successfully scheduled by HSP on M processors. Different from Theorem 4.3.10, Theorem 4.4.6 works for arbitrary task sets instead of light task sets alone.

It is worth mentioning that, based on our proofs, Theorem 4.3.10 and Theorem 4.4.6 hold true even without the consideration of period relationships, i.e. lines 3-7 of Algorithm HSP-ligh and lines 16-19 of Algorithm HSP. To study if our approach can lead to a better utilization bound is an interesting problem and will be our future study. In what follows, we use experiments to study the potential improvement that can be achieved using our methods.

4.5 Experiments And Results

In this section, we investigate the performance of our proposed algorithms with experiments. Five algorithms are implemented in our experiments.

- *SPA*: The *SPA* algorithm [43] assigns the priority of each task by RMS, and splits a task to feed the processor until “full” (e.g. utilization equal to the *Liu&Layland’s bound*). However, as long as the utilization of a task set exceeds the *Liu&Layland’s bound*, it simply aborts.

- *DM-PM*: The *DM-PM* algorithm [69] assigns task priorities by deadline monotonic scheduling (DMS) policy, and splits a task and assigns as large portion of the task as possible to a processor by computing the maximum interference to the task on each processor.
- *PUB*: The *PUB* algorithm [44], similarly to *SPA*, assigns tasks based on a parametric utilization bound, but uses exact timing analysis method for task splitting. In the following experiments, *R-Bound* [73] is applied with this algorithm.
- *pCOMPATS*: The *pCOMPATS* algorithm [66] explores the R-Bound [73] for task partitioning and splitting. R-Bound can only be applied to task sets with ratio of any two periods no smaller than 1 and no larger than 2. In our experiments, we used the same algorithm as that in [66] to scale a general task set.
- *HSP*: Our proposed algorithm. Note that *HSP* is the same as *HSP-light* when the task set is light, and can accommodate task sets containing heavy tasks.

We conducted two groups of experiments to study how performance of each algorithm changes with different numbers of tasks and different system utilizations, respectively. For each group of experiments, we tested on different number of processors, i.e. $M = 4, 8$, and 16 . For each testing point in the experiments, we randomly generated 500 task sets as test cases. The utilization of each task set varied from 0.5 to 1 (since task sets with smaller utilizations could be easily schedulable by all approaches). The minimum inter-arrival time of each task was set to have a uniform distribution within $[50, 1000]$. The scheduling performance for different approaches are compared using the *success ratios*, i.e. the number of feasible tasks over the number of total tasks generated under a specific test point.

4.5.1 Performance VS. Number Of Tasks

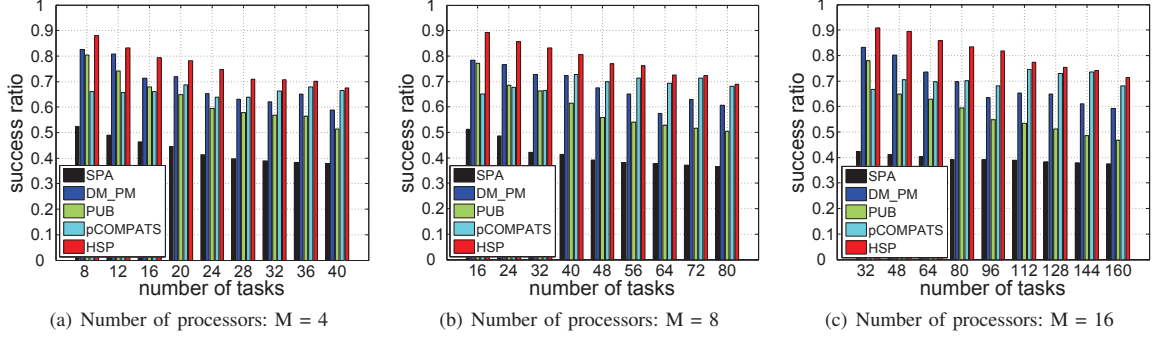


Figure 4.4: Experimental results for general task sets by different number of tasks.

In this group of experiments, we varied the number of tasks, i.e. N , in a task set from $2 \times M$ to $10 \times M$ with an increment of M (where M is the number of processors). The success ratios of all five approaches were recorded and plotted in Figure 4.4.

From Figure 4.4, we can observe that *HSP* can achieve success ratios much better than other four approaches. For example, in Figure 4.4(a), when the number of tasks is equal to 20, *HSP* can achieve a success ratio of 78%, an improvement of 1.7 times of that by *SPA* (45%), 1.1 times of that by *DM_PM* (71%), 1.2 times of that by *PUB* (64%), and 1.1 times of that by *pCOMPATS* (68%). The improvement of *HSP* comes from the fact that *HSP* takes the harmonic relationship among tasks aggressively into consideration and tries to allocate tasks closer to harmonic together among multiple processors. The exploitation of harmonicity is limited to that the utilization bounds for different processors may be different depends on how existing tasks are close to harmonic.

From Figure 4.4, we can see that, for the same number of processors (M), the success ratio of *HSP* in general decreases with the increase of task numbers (N). For example, in Figure 4.4(c) (as $M = 16$), the success ratio of *HSP* achieves 91% when

$N = 32$, but it decreases to 71% when N increases to 160. The larger the number of task is, the lower the utilization bound can be. As a result, a task set becomes more difficult to be schedulable. From Figure 4.4, it is also interesting to see that, if we assume similar average number of tasks for each processor (i.e. assuming N/M as a constant), the success ratio by *HSP* largely increases in general. For example, when $N/M = 5$, the success ratios for $M = 4, 8, 16$ are 78% (see Figure 4.4(a) at $N = 20$), 80% (see Figure 4.4(b) at $N = 40$) and 83% (see Figure 4.4(c) at $N = 80$), respectively. The reason for this is that the more processors are available, there are more opportunities that can be exploited by *HSP* to take advantage of the harmonic property among tasks to improve the processor utilization.

4.5.2 Performance VS. System Utilization

To study the performance differences by different scheduling approaches under different system utilizations, we conducted three sub-groups of experiments, for light and general task sets, respectively. In light task sets, the utilization of each task was evenly distributed within $[0, 0.5]$, while in general task sets, the utilization of each task was evenly distributed within $[0, 1]$. For each experiment, we varied the system utilization from 0.5 to 1.0 with an increment of 0.025. The experimental results for all approaches are collected and shown in Figure 4.5 and Figure 4.6.

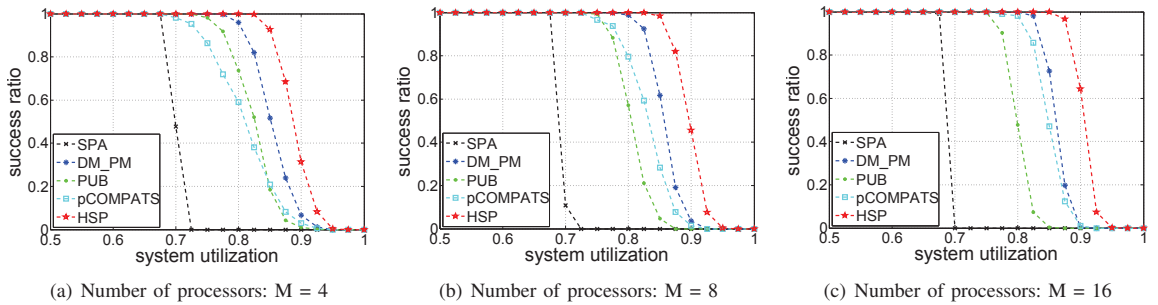


Figure 4.5: Experimental results for light task sets, $u \in [0, 0.5]$.

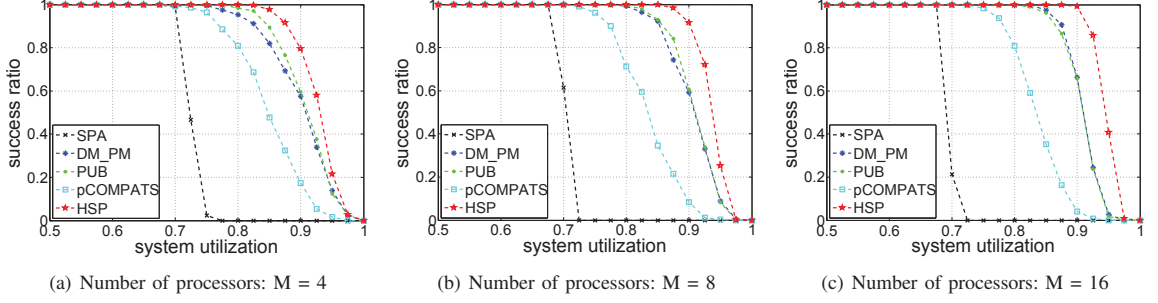


Figure 4.6: Experimental results for general task sets, $u \in [0, 1]$.

Figure 4.5 shows our experimental results for task sets containing only light tasks. From Figure 4.5, we can observe that *HSP* can achieve success ratios significantly better than other four approaches. Compared with *SPA*, all other four approaches, i.e. *DM_PM*, *PUB*, *pCOMPATS* and *HSP* can guarantee the schedulability of any task set with utilization below *Liu&Layland's bound*, the same as *SPA*. The success ratio by *SPA* drops sharply when system utilization around 0.7. This is because that while *SPA* can guarantee any task sets with utilizations no more than the *Liu&Layland's bound*, it rejects any task set with system utilization exceeding the *Liu&Layland's bound*. While *DM_PM*, *PUB* and *pCOMPATS* may potentially schedule task sets with utilization higher than the *Liu&Layland's bound*, *HSP* can achieve a much higher performance, especially when the system utilization is high. For example, in Figure 4.5(a), when the system utilization is around 0.9, *HSP* can still achieve a success ratio up to 30%, while that of *DM_PM* is 10%, and that of *PUB* and *pCOMPATS* are no more than 5%. Similar to our first group of experiments, we can see that the performance improvement by *HSP* tends to increase as the number of processors increases. Under the system utilization of 0.9, *HSP* can achieve a success ratio of 30% with 4 processors, 40% with 8 processors, and increased up to 60% with 16 processors.

Figure 4.6 shows our experimental results for general task sets containing both

heavy and light tasks. From Figure 4.6, we can also observe that *HSP* performs significantly better than other four approaches. In Figure 4.6(c), *HSP* can achieve a success ratio four times of that by *DM-PM* and *PUB* when the system utilization is around 0.925.

Our experimental results clearly show, by exploiting the harmonic relationship among tasks more aggressively, *HSP* can significantly improve the schedulability of semi-partitioned scheduling compared with the existing algorithms.

4.6 Summary

In this chapter, we have presented a new semi-partitioned approach for scheduling real-time sporadic tasks on multi-core platform under RMS. Our approach can take advantage of the harmonic relations among task periods and improve the schedulability. To achieve this goal, we introduced a metric to quantify how close a task set is to a harmonic task set. Two algorithms, i.e. *HSP-light* and *HSP*, were presented to schedule light and general task sets, respectively. We have formally analyzed the schedulability for both algorithms, and presented a simple schedulability test method for each one. Specifically, we formally proved that our scheduling algorithms can successfully schedule any task set with a system utilization bounded by the *Liu&Layland's bound*. The experimental results demonstrated that the proposed algorithm can significantly improve the scheduling performance compared with previous work.

CHAPTER 5

TEMPERATURE-CONSTRAINED FEASIBILITY ANALYSIS FOR MULTI-CORE REAL-TIME SCHEDULING

In previous chapters, we present our multi-core scheduling algorithms to ensure the timing constraints for real-time embedded applications. In this chapter, we consider not only timing constraints but also temperature constraints as well. Specifically, we focus our interest on the problem that if a given real-time periodic dynamic voltage frequency scaling (DVFS) schedule for a multi-core platform can satisfy a pre-defined maximum temperature constraint. Many processors today have a built-in digital thermal sensor integrated with each core with predefined temperature limit [21]. When temperature rises above the limit, the processor can automatically shut down. Thus real-time tasks may miss their deadlines. Therefore, it is important to develop appropriate feasibility checking methods such that a periodic schedule can ensure the peak temperature constraint during its life time.

5.1 Related Work

There are a few researches targeted on temperature-constrained real-time scheduling. Zhang *et al.* [125] proposed a scheduling algorithm that guarantees the maximum temperature constraint for a periodic schedule by requiring the starting temperature is no more than that at the end of its first scheduling period. Quan *et al.* [97] took the temperature/leakage dependency into consideration and proposed several feasibility conditions. All these techniques are developed based on single core platforms. The problem becomes significantly more challenging for multi-core systems, since the temperature of each core varies depending on not only its instant temperature and power consumption, but also the temperatures of other cores as well.

There are a few work published on thermal/temperature aware multi-core scheduling (e.g. [125, 97, 104, 116, 105, 63]). Zhang *et al.* [125] proposed a single-core scheduling algorithm that guarantees the maximum temperature constraint for a periodic schedule by requiring the starting temperature is no more than that at the end of its first scheduling period. Quan *et al.* [97] took the temperature/leakage dependency into consideration and proposed several feasibility conditions for periodic scheduling on single-core platforms. Lars *et al.* [104] proposed an approach to estimate the worst-case temperature on a multi-core platform by searching for the worst-case task/workload allocations among different cores. They assumed each core can only have two running modes (i.e. on or off), and also assumed a complete knowledge of the temperature “impulse response function” (i.e. the temperature variation on a core due to the heat transfer from another core, which can be very challenging in practice). However, how to employ the proposed method for periodic tasks is still a problem. Ukhov *et al.* [116] presented a method to keep track of temperature for a multi-core platforms under steady state under the assumption that the power consumption is a constant.

In our research, we adopt a power model that accounts for the interdependency between leakage power and temperature, and a thermal model that takes the heat transfer among different processing cores into consideration. In what follows, we first introduce some preliminary background closely related to this chapter.

5.2 Preliminary

5.2.1 System Models

The multi-core platform considered in this work consists of N_c identical cores, denoted as $\mathbb{C} = \{\mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_{N_c}\}$. Each core can run in r different *processing modes*, each of which is characterized by a pair of parameters (v_k, f_k) ($1 \leq k \leq r$), where v_k is the supply voltage and f_k is the working frequency in mode k , respectively. Let f_{max} be the largest frequency among different processing modes.

We assume that a static, periodic *speed schedule* \mathbb{S} , which dictates how to vary the supply voltage and working frequency for each core at different intervals, is given. For a speed schedule \mathbb{S} , we define the concept of *state interval* below:

Definition 5.2.1. *Given a multi-core system and a speed schedule \mathbb{S} , an interval $[t_{q-1}, t_q]$ is called a state interval if each core runs only at one processing mode during that interval.*

Consider a periodic task set, and let L denote the length of one scheduling period (which equals to the *least common multiple* (LCM) of periods of all tasks). According to Definition 5.2.1, a periodic speed schedule \mathbb{S} essentially consists of a number of non-overlapped state intervals, i.e. N_s state intervals, such that

1. $\bigcup_{q=1}^{N_s} [t_{q-1}, t_q] = [0, L]$, and
2. $[t_{q-1}, t_q] \cap [t_{p-1}, t_p] = \emptyset$, if $q \neq p$.

For a state interval $[t_{q-1}, t_q]$, we use κ_q to denote the *interval processing mode*, which consists of the processing mode of each core, i.e. $\kappa_q = \{k_1, \dots, k_{N_c}\}$ where k_i is the processing mode of core \mathcal{C}_i .

5.2.2 Power Model

The overall power consumption (in *Watt*) of each core is composed of two parts: dynamic power P_{dyn} and leakage power P_{leak} . In our power model, P_{dyn} is independent of the temperature, while P_{leak} is sensitive to both temperature and supply voltage. The dynamic power consumption of the i^{th} core \mathcal{C}_i , denoted as $P_{dyn,i}$, can be formulated as [102]

$$P_{dyn,i} = \gamma_{k_i} \cdot v_{k_i}^3 \quad (5.1)$$

where v_{k_i} is the supply voltage of core \mathcal{C}_i and γ_{k_i} is a constant, both of which depend on the processing mode of core \mathcal{C}_i , i.e. mode k_i . While the circuit level study reveals a very complicated relation between leakage power and temperature, Liu *et al.* [122] found that a linear approximation of the leakage temperature dependency is fairly accurate. As such, similar to the work in [122, 97], we model the leakage power of the i^{th} core \mathcal{C}_i , denoted as $P_{leak,i}$, as follows

$$P_{leak,i} = (\alpha_{k_i} + \beta_{k_i} \cdot T_i(t)) \cdot v_{k_i} \quad (5.2)$$

where α_{k_i} and β_{k_i} are constants only depending on the core's processing mode, i.e. mode k_i . Consequently, the total power consumption of core \mathcal{C}_i at time t , denoted as $P_i(t)$, can be formulated as:

$$P_i(t) = (\alpha_{k_i} + \beta_{k_i} \cdot T_i(t)) \cdot v_{k_i} + \gamma_{k_i} \cdot v_{k_i}^3 \quad (5.3)$$

According to equation (5.3), we thus have

$$P_i(t) = \theta_i + \phi_i \cdot T_i(t) \quad (5.4)$$

where $\theta_i = \alpha_{k_i} \cdot v_{k_i} + \gamma_{k_i} \cdot v_{k_i}^3$ and $\phi_i = \beta_{k_i} \cdot v_{k_i}$. For the entire multi-core system, we have

$$\mathbf{P}(t) = \mathbf{\Theta} + \mathbf{\Phi} \mathbf{T}(t) \quad (5.5)$$

where $\mathbf{P}(t)$ is the power vector, $\mathbf{T}(t)$ is the temperature vector, $\mathbf{\Theta}$ is the vector of temperature independent power (i.e. $\mathbf{\Theta} = \{\theta_1, \theta_2, \dots, \theta_{N_c}\}$, and $\mathbf{\Phi}$ is a diagonal matrix with the i^{th} diagonal element as ϕ_i . Note that, to distinguish a vector/matrix with a value, we use bold text and normal text respectively, e.g. \mathbf{T} represents a temperature vector and T represents a temperature value.

5.2.3 Thermal Model

The thermal behavior for a multi-core platform is modeled using an equivalent RC thermal circuit, similar to that used in [105, 116]. A multi-core platform may contains multiple die layers and thermal packaging (e.g. the thermal interface material, heat spreader, and heat sink). Thermal nodes on die layers are called active nodes, since they represent the actual processing cores of the system and consume non-zero power. In contrast, thermal nodes on the thermal packaging layers are called inactive nodes, since their power dissipation is assumed to be zero regardless of the system processing modes. Specifically, we assume that a multi-core system \mathcal{C} consists of N_t thermal nodes, denoted as $\diamond = \{\Pi_1, \Pi_2, \dots, \Pi_{N_t}\}$, with the first N_c nodes as the active nodes and others as the inactive nodes. Then the thermal phenomena of Π_i (the i^{th} thermal node) can be formulated as

$$C_i \cdot \frac{dT_i(t)}{dt} + \frac{T_i(t) - T_{amb}}{R_{ii}} + \sum_{j \neq i} \frac{T_i(t) - T_j(t)}{R_{ij}} = P_i(t) \quad (5.6)$$

where C_i is the thermal capacitance (in $Watt/^\circ C$) of Π_i , R_{ij} is the thermal resistance (in $J/^\circ C$) between Π_i and Π_j , T_i is the temperature of Π_i , T_{amb} is the ambient temperature, and P_i is the power consumption of Π_i . Equivalently, we have

$$\mathbf{C} \frac{d\mathbf{T}(t)}{dt} + \mathbf{G}(\mathbf{T}(t) - \mathbf{T}_{amb}) = \mathbf{P}(t) \quad (5.7)$$

where \mathbf{C} , \mathbf{G} , \mathbf{T} and \mathbf{P} are thermal capacitance matrix, thermal conductance matrix, temperature vector and power vector, respectively. From equation (5.5) and further normalize the temperature with respect to T_{amb} (i.e. $\mathbf{T}(t) \equiv \mathbf{T}(t) - \mathbf{T}_{amb}$), we have

$$\mathbf{C} \frac{d\mathbf{T}(t)}{dt} + \mathbf{g}\mathbf{T}(t) = \mathbf{\Psi} \quad (5.8)$$

where $\mathbf{g} = \mathbf{G} - \mathbf{\Phi}$ and $\mathbf{\Psi} = \mathbf{\Theta} + \mathbf{\Phi}\mathbf{T}_{amb}$.

Consider a state interval $[t_{q-1}, t_q]$, according to equation (5.8), we have

$$\left. \frac{d\mathbf{T}(t)}{dt} \right|_{t \in [t_{q-1}, t_q]} = \mathbf{A}_{\kappa_q} \mathbf{T}(t) + \mathbf{B}_{\kappa_q} \quad (5.9)$$

where $\mathbf{A}_{\kappa_q} = -\mathbf{C}^{-1}\mathbf{g}_{\kappa_q}$ and $\mathbf{B}_{\kappa_q} = \mathbf{C}^{-1}\mathbf{\Psi}_{\kappa_q}$. Note that within $[t_{q-1}, t_q]$, both \mathbf{A}_{κ_q} and \mathbf{B}_{κ_q} are constant. The thermal model given by equation (5.11) is a model of ordinary differential equations (ODE), specifically a first-order constant coefficient ODE model, with the following solution:

$$\mathbf{T}(t_q) = e^{\mathbf{A}_{\kappa_q} \Delta t_q} \mathbf{T}(t_{q-1}) + \mathbf{A}_{\kappa_q}^{-1} (e^{\mathbf{A}_{\kappa_q} \Delta t_q} - \mathbf{I}) \mathbf{B}_{\kappa_q} \quad (5.10)$$

where $\Delta t_q = t_q - t_{q-1}$, and \mathbf{I} is an $m \times m$ identity matrix.

5.2.4 Problem Description

With the models introduced above, our research problem can be formally formulated as follows:

Problem 5.2.2. *Given:*

- a multi-core system $\mathbb{C} = \{\mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_m\}$ with r different processing modes;
- a static, periodic speed schedule \mathbb{S} consisting of s state intervals within each scheduling period;
- a maximum temperature constraint T_{max} ;

determine if, starting from the ambient temperature T_{amb} and repeating schedule \mathbb{S} until system reaches the stable status, the temperature on each core will exceed T_{max} .

Note that checking temperature constraint within the first scheduling period cannot ensure the constraint in the future. To exhaustively check temperature for all scheduling periods would also be impossible. In what follows, we first strive to determine the peak temperature for a schedule. We then present our solutions for this problem.

5.3 Temperature Calculation For Multi-core Scheduling

In this section, we present new methods to rapidly calculate temperatures for a periodic multi-core scheduling.

We first introduce the temperature formulation within a state interval. Then we present a method to rapidly calculate temperatures when repeating a periodic speed schedule based on temperature information of the first scheduling period. Finally, we present an efficient method to calculate the steady-state temperature for a periodic speed schedule.

5.3.1 Temperature Formulation Within A State Interval

Consider a state interval $[t_{q-1}, t_q]$, according to equation (5.8), the thermal variation can be rewritten as

$$\left. \frac{d\mathbf{T}(t)}{dt} \right|_{t \in [t_{q-1}, t_q]} = \mathbf{A}_{\kappa_q} \mathbf{T}(t) + \mathbf{B}_{\kappa_q} \quad (5.11)$$

where $\mathbf{A}_{\kappa_q} = -\mathbf{C}^{-1}\mathbf{g}_{\kappa_q}$ and $\mathbf{B}_{\kappa_q} = \mathbf{C}^{-1}\Psi_{\kappa_q}$. Note that within $[t_{q-1}, t_q]$, both \mathbf{A}_{κ_q} and \mathbf{B}_{κ_q} are constant.

The thermal model given by equation (5.11) is a model of ordinary differential equations (ODE), specifically a first-order constant coefficient ODE model, with the following solution.

$$\mathbf{T}(t_q) = e^{\mathbf{A}_{\kappa_q} \Delta t_q} \mathbf{T}(t_{q-1}) + \mathbf{A}_{\kappa_q}^{-1} (e^{\mathbf{A}_{\kappa_q} \Delta t_q} - \mathbf{I}) \mathbf{B}_{\kappa_q} \quad (5.12)$$

where $\Delta t_q = t_q - t_{q-1}$, and \mathbf{I} is an $m \times m$ identity matrix.

Note that the above analytical solution for temperature calculation includes a computationally expensive operation, i.e. the matrix exponential operation of $e^{\mathbf{A}_{\kappa_q}}$. Generally, there are two alternatives for solving this kind of operation: mathematical solvers (e.g. MatLab tool) and auxiliary transformation (e.g. work [116]). Any available auxiliary transformation method used for matrix exponential operation can be apply to optimize the computations of the proposed analytical temperature solutions in this work.

From equation (5.12), we can see that given a state interval, its ending temperature can be determined by the starting temperature $\mathbf{T}(t_{q-1})$ and the corresponding interval processing mode κ_q . In the next subsection, we will develop some further analytical solutions to capture the characteristic of temperature variation for a periodic schedule.

5.3.2 Temperature Formulation For A Periodic Schedule

With the method introduced above, given a periodic schedule \mathbb{S} and the initial temperature $\mathbf{T}(0)$, we can calculate the temperature at any time instant by tracking temperature from one state interval to another. However, when $t \gg 0$, the computational cost can be extremely large. In what follows, we present a method to rapidly calculate temperatures when repeating a periodic speed schedule \mathbb{S} based on temperature information of the first scheduling period.

For ease of presentation, we first introduce a new notation, i.e. \mathbf{K}_q , where $q = 1, 2, \dots, s$, such that

$$\mathbf{K}_q = e^{\mathbf{A}_{\kappa_q} \Delta t_q} \cdot e^{\mathbf{A}_{\kappa_{q-1}} \Delta t_{q-1}} \dots e^{\mathbf{A}_{\kappa_1} \Delta t_1} \quad (5.13)$$

In particular, let \mathbf{K} be a special case of \mathbf{K}_q when $q = s$, such that

$$\mathbf{K} = e^{\mathbf{A}_{\kappa_s} \Delta t_s} \cdot e^{\mathbf{A}_{\kappa_{s-1}} \Delta t_{s-1}} \dots e^{\mathbf{A}_{\kappa_1} \Delta t_1} \quad (5.14)$$

Then we consider an arbitrary time instance t_q within the first scheduling period, i.e. $t_q \in [0, L]$. If $\mathbf{T}(t_q)$ is determined, with the help of the above notifications, i.e. \mathbf{K}_q and \mathbf{K} , we then provide a method that can quickly calculate the temperature at $t = nL + t_q$, where $n \geq 1$.

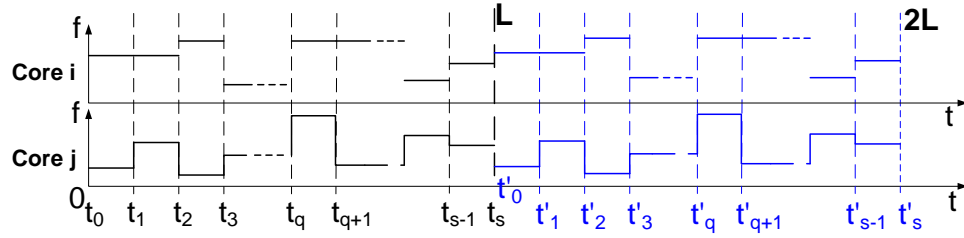


Figure 5.1: A speed schedule within 2 scheduling periods.

Before presenting our temperature calculation method for an arbitrary time point, we first introduce a method that can quickly calculate the temperature at the end of each scheduling period, i.e. $t = nL$, where $n \geq 1$.

Lemma 5.3.1. *Given a periodic speed schedule \mathbb{S} , let $\mathbf{T}(L)$ be the temperature at time L . When repeating \mathbb{S} later, if $(\mathbf{I} - \mathbf{K})$ is invertible, then the temperature at time $t = nL$, where n is an integer and $n \geq 1$, can be formulated as*

$$\mathbf{T}(nL) = \mathbf{T}(0) + (\mathbf{I} - \mathbf{K})^{-1}(\mathbf{I} - \mathbf{K}^n)(\mathbf{T}(L) - \mathbf{T}(0)) \quad (5.15)$$

Proof: We first consider the temperature dynamics at the end of each scheduling period, i.e. $t = nL$. Let the scheduling points of the state intervals in \mathbb{S} be

t_0, t_1, \dots, t_{s-1} , respectively. After repeating \mathbb{S} , let the corresponding points in the second scheduling period be $t'_0, t'_1, \dots, t'_{s-1}$, respectively (see Fig. 5.1). Note that $t_0 = 0, t'_0 = t_s = L$ and $t'_s = 2L$. According to equation (5.12), at time t_1 and t'_1 , we have

$$\mathbf{T}(t_1) = e^{\mathbf{A}_{\kappa_1} \Delta t_1} \mathbf{T}(t_0) + \mathbf{A}_{\kappa_1}^{-1} (e^{\mathbf{A}_{\kappa_1} \Delta t_1} - \mathbf{I}) \mathbf{B}_{\kappa_1} \quad (5.16)$$

$$\mathbf{T}(t'_1) = e^{\mathbf{A}_{\kappa_1} \Delta t'_1} \mathbf{T}(t'_0) + \mathbf{A}_{\kappa_1}^{-1} (e^{\mathbf{A}_{\kappa_1} \Delta t'_1} - \mathbf{I}) \mathbf{B}_{\kappa_1} \quad (5.17)$$

Subtract equation (5.16) from (5.17), and simply the result by applying $\Delta t'_1 = \Delta t_1$, $t_0 = 0$ and $t'_0 = L$, we get

$$\mathbf{T}(t'_1) - \mathbf{T}(t_1) = e^{\mathbf{A}_{\kappa_1} \Delta t_1} (\mathbf{T}(L) - \mathbf{T}(0))$$

Similarly, we can derive that

$$\begin{aligned} \mathbf{T}(t'_2) - \mathbf{T}(t_2) &= e^{\mathbf{A}_{\kappa_2} \Delta t_2} e^{\mathbf{A}_{\kappa_1} \Delta t_1} (\mathbf{T}(L) - \mathbf{T}(0)) \\ &\dots \\ \mathbf{T}(t'_s) - \mathbf{T}(t_s) &= e^{\mathbf{A}_{\kappa_s} \Delta t_s} \dots e^{\mathbf{A}_{\kappa_1} \Delta t_1} (\mathbf{T}(L) - \mathbf{T}(0)) \end{aligned} \quad (5.18)$$

Since $t_s = L$, $t'_s = 2L$, and $e^{\mathbf{A}_{\kappa_s} \Delta t_s} \dots e^{\mathbf{A}_{\kappa_1} \Delta t_1} = \mathbf{K}$, equation (5.18) can be rewritten as

$$\mathbf{T}(2L) - \mathbf{T}(L) = \mathbf{K}(\mathbf{T}(L) - \mathbf{T}(0)) \quad (5.19)$$

In the same way, we can see that

$$\begin{aligned} \mathbf{T}(3L) - \mathbf{T}(2L) &= \mathbf{K}(\mathbf{T}(2L) - \mathbf{T}(L)) \\ \mathbf{T}(4L) - \mathbf{T}(3L) &= \mathbf{K}(\mathbf{T}(3L) - \mathbf{T}(2L)) \\ &\dots \\ \mathbf{T}(nL) - \mathbf{T}((n-1)L) &= \mathbf{K}(\mathbf{T}((n-1)L) - \mathbf{T}((n-2)L)) \end{aligned}$$

Thus, we can construct that

$$\mathbf{T}(xL) - \mathbf{T}((x-1)L) = \mathbf{K}^{x-1} (\mathbf{T}(L) - \mathbf{T}(0)) \quad (5.20)$$

where $x = 1, 2, \dots, n$. Sum up these n equations based on the above, and simplify the result, then we can derive that

$$\mathbf{T}(nL) = \mathbf{T}(0) + \left(\sum_{x=1}^n \mathbf{K}^{x-1}\right)(\mathbf{T}(L) - \mathbf{T}(0)) \quad (5.21)$$

In the above, $\{\mathbf{K}^{x-1} | x = 1, 2, \dots, n\}$ forms a matrix geometric sequence. If $(\mathbf{I} - \mathbf{K})$ is invertible, then we have

$$\mathbf{T}(nL) = \mathbf{T}(0) + (\mathbf{I} - \mathbf{K})^{-1}(\mathbf{I} - \mathbf{K}^n)(\mathbf{T}(L) - \mathbf{T}(0)) \quad (5.22)$$

□

With the help of Lemma 5.3.1, we can directly obtain the temperature trace of a periodic speed schedule at the ending points of all scheduling periods, i.e. $t = nL$ where $n \geq 1$. Further, for any arbitrary time point, i.e. $t = nL + t_q$ where $n \geq 1$ and $0 \leq t_q \leq L$, we develop a similar approach that can efficiently calculate the corresponding temperature. We formally conclude our method in Theorem 5.3.2.

Theorem 5.3.2. *Given a periodic speed schedule \mathbb{S} , let $\mathbf{T}(L)$ and $\mathbf{T}(t_q)$ be the temperatures at time L and t_q , $t_q \in [0, L]$, respectively. When repeating \mathbb{S} later, if $(\mathbf{I} - \mathbf{K})$ is invertible, then the temperature at time $t = nL + t_q$, where n is an integer and $n \geq 1$, can be formulated as*

$$\mathbf{T}(nL + t_q) = \mathbf{T}(t_q) + \mathbf{K}_q(\mathbf{I} - \mathbf{K})^{-1}(\mathbf{I} - \mathbf{K}^n)(\mathbf{T}(L) - \mathbf{T}(0)) \quad (5.23)$$

Proof: Follow the same procedure of Lemma 5.3.1's proof, we have that

$$\mathbf{T}(L + t_q) - \mathbf{T}(t_q) = e^{\mathbf{A}_{\kappa_q} \Delta t_q} \dots e^{\mathbf{A}_{\kappa_1} \Delta t_1} (\mathbf{T}(L) - \mathbf{T}(0))$$

According to equation (5.13), we can replace $e^{\mathbf{A}_{\kappa_q} \Delta t_q} \dots e^{\mathbf{A}_{\kappa_1} \Delta t_1}$ with \mathbf{K}_q , then the above equation can be rewritten as

$$\mathbf{T}(L + t_q) - \mathbf{T}(t_q) = \mathbf{K}_q \cdot (\mathbf{T}(L) - \mathbf{T}(0))$$

Similarly, for all temperatures at time instances of $2L + t_q, 3L + t_q, \dots, nL + t_q$, we have

$$\begin{aligned} \mathbf{T}(2L + t_q) - \mathbf{T}(L + t_q) &= \mathbf{K}_q(\mathbf{T}(2L) - \mathbf{T}(L)) \\ &\dots \\ \mathbf{T}(nL + t_q) - \mathbf{T}((n-1)L + t_q) &= \mathbf{K}_q(\mathbf{T}(nL) - \mathbf{T}((n-1)L)) \end{aligned}$$

Add all above n equations together, we get

$$\mathbf{T}(nL + t_q) = \mathbf{T}(t_q) + \mathbf{K}_q(\mathbf{T}(nL) - \mathbf{T}(0))$$

According to Lemma 5.3.1, we know that $\mathbf{T}(nL)$ can be explicitly represented by equation (5.15). Thus, if $(\mathbf{I} - \mathbf{K})$ is invertible, by replacing $\mathbf{T}(nL)$ with equation (5.15) in the above, we can get

$$\mathbf{T}(nL + t_q) = \mathbf{T}(t_q) + \mathbf{K}_q(\mathbf{I} - \mathbf{K})^{-1}(\mathbf{I} - \mathbf{K}^n)(\mathbf{T}(L) - \mathbf{T}(0)) \quad (5.24)$$

□

From Theorem 5.3.2, we can see that once the temperature information in the first scheduling period is determined, the corresponding temperature in any future scheduling period can be quickly calculated.

5.3.3 Steady-State Temperature Formulation

Now we introduce an efficient method to calculate the steady-state temperature for a periodic speed schedule based on the temperature information in the first scheduling period.

Theorem 5.3.3. *Given a periodic speed schedule \mathbb{S} , let $\mathbf{T}(L)$ and $\mathbf{T}(t_q)$ be the temperatures at time L and t_q , $t_q \in [0, L]$, respectively. If for each eigenvalue λ_i of*

\mathbf{K} , we have $|\lambda_i| < 1$, then the steady-state temperature corresponding to t_q can be formulated as

$$\mathbf{T}_{ss}(t_q) = \mathbf{T}(t_q) + \mathbf{K}_q(\mathbf{I} - \mathbf{K})^{-1}(\mathbf{T}(L) - \mathbf{T}(0)) \quad (5.25)$$

Proof: As $n \mapsto \infty$, no matter with or without temperature constraint, $\mathbf{T}(nL + t_q)$ will achieve its stable status while the system achieves its thermal steady-state. Thus, the temperature in the system stable status corresponding to the end of the q^{th} state interval, denoted as $\mathbf{T}_{ss}(t_q)$, can be formulated as

$$\mathbf{T}_{ss}(t_q) = \lim_{n \mapsto \infty} \mathbf{T}(nL + t_q) \quad (5.26)$$

According to Theorem 5.3.2, if $(\mathbf{I} - \mathbf{K})$ is invertible, the temperature at time $nL + t_q$ can be formulated as

$$\mathbf{T}(nL + t_q) = \mathbf{T}(t_q) + \mathbf{K}_q(\mathbf{I} - \mathbf{K})^{-1}(\mathbf{I} - \mathbf{K}^n)(\mathbf{T}(L) - \mathbf{T}(0)) \quad (5.27)$$

Apply equation (5.27) into (5.26), $\mathbf{T}_{ss}(t_q)$ can be further represented as

$$\begin{aligned} \mathbf{T}_{ss}(t_q) &= \mathbf{T}(t_q) + \\ &\quad \mathbf{K}_q(\mathbf{I} - \mathbf{K})^{-1}(\mathbf{I} - \lim_{n \mapsto \infty} \mathbf{K}^n)(\mathbf{T}(L) - \mathbf{T}(0)) \end{aligned} \quad (5.28)$$

When $n \mapsto \infty$, the matrix sequence \mathbf{K}^n converges if and only if $|\lambda_i| < 1$, for each eigenvalue λ_i of \mathbf{K} [64]. Under this condition, we have $\lim_{n \mapsto \infty} \mathbf{K}^n = 0$. Moreover, if $\forall \lambda_i, |\lambda_i| < 1$ holds, $(\mathbf{I} - \mathbf{K})$ is invertible. Thus, the steady-state temperature formulated by equation (5.28) can be further represented as

$$\mathbf{T}_{ss}(t_q) = \mathbf{T}(t_q) + \mathbf{K}_q(\mathbf{I} - \mathbf{K})^{-1}(\mathbf{T}(L) - \mathbf{T}(0)) \quad (5.29)$$

□

It is important to point out that the condition assumed in Theorem 5.3.3, i.e. for each eigenvalue λ_i of \mathbf{K} , we have $|\lambda_i| < 1$, makes sense in the physical world.

As well known, without any peak temperature constraint, the system can always achieve its thermal stable status by running a long enough time. Thus, the matrix sequence \mathbf{K}^n , as shown in equation (5.23), must converge when $n \mapsto \infty$; otherwise $\mathbf{T}(nL + t_q)$ will go to infinity, which conflicts with the practical scenario. Therefore, the condition given in Theorem 5.3.3, i.e. $\forall \lambda_i, |\lambda_i| < 1$, is reasonable and feasible.

From Theorem 5.3.3 we can see that, given a periodic speed schedule \mathbb{S} , the system steady-state temperature can be formulated with the temperature information of the first scheduling period directly. This is much more efficient than to keep track of temperature variations based on equation (5.12). In the following section, we discuss the problem of how to determine the peak temperature for a given periodic speed schedule.

5.4 Identifying The Peak Temperature

To get the peak temperature for a periodic schedule is essential for solving Problem 5.2.2. While solutions for single core platforms has been presented in previous work (e.g. [97]), this problem becomes substantially more difficult since the temperature of each core changes not only with its instant temperature and power consumption, but also with the temperatures of other cores as well. In what follows, we first study how to find the peak temperature within a state interval. We then discuss how to find the peak temperature for a periodic schedule.

5.4.1 Challenging Problem In Peak Temperature Detection

Intuitively, if all cores use constant speeds throughout an interval, we would expect that the peak temperature occurs at one of the ending points of the interval. This

is true on a single-core case and can be easily proved. Unfortunately, it is not true anymore for the multi-core platforms.

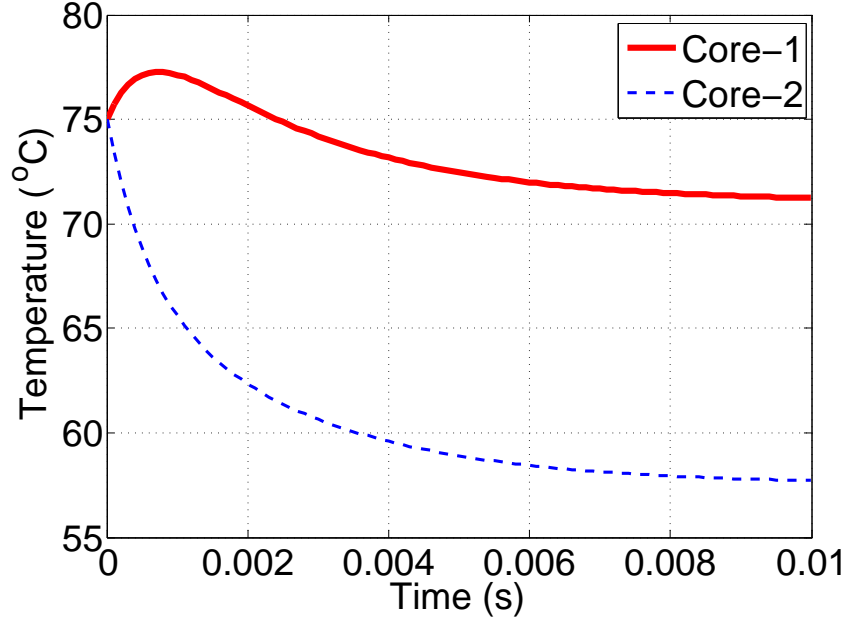


Figure 5.2: Negative interaction on temperature variation between two cores. $C_1 = C_2 = 0.00035$, $G_{11} = G_{22} = 0.4$, $G_{12} = G_{21} = -0.1$, $v_1 = 0.8V$, $v_2 = 0V$, $T_1(0) = T_2(0) = 75^\circ C$.

Figure 5.2 shows the temperature traces of two cores when running with constant speeds within an interval. As shown in Figure 5.2, while the temperature of Core-2 decreases monotonically, Core-1's temperature first rises and then drops. As a result, the peak temperature of the system does not occur at either of the ending points of the interval at all. Then, to solve Problem 5.2.2, we have to ask the question: where and when the processor achieves its peak temperature? In what follows, we first present some characteristics about the peak temperature variation within a state interval, then propose an efficient approach to detect the peak temperature for any state interval.

5.4.2 Important Properties For Multi-core Temperature Variation

To study the temperature variation on a multi-core platform within a state interval, we have made a number of interesting and important findings.

Given a state interval (i.e. $[t_0, t_1]$), if temperatures of all cores are simultaneously either increasing or decreasing at the starting point (i.e. $t = t_0$), then the temperatures of all cores must monotonically change, i.e. either all increase or decrease, within that interval. We formally conclude this property in Lemma 5.4.1

Lemma 5.4.1. *Given a multi-core platform \mathbb{C} and a state interval $[t_0, t_1]$, if it holds*

$$\forall \mathcal{C}_i \in \mathbb{C} \quad , \quad \left. \frac{dT_i(t)}{dt} \right|_{t=t_0} > 0 \quad (5.30)$$

or

$$\forall \mathcal{C}_i \in \mathbb{C} \quad , \quad \left. \frac{dT_i(t)}{dt} \right|_{t=t_0} < 0 \quad (5.31)$$

then the temperature of all cores must monotonically either increase or decrease within $[t_0, t_1]$.

Proof: Assume that $\forall \mathcal{C}_i \in \mathbb{C}$, $\left. \frac{dT_i}{dt} \right|_{t=t_0} > 0$. Then for $\forall t_x \in [t_0, t_1]$, according to equation (5.11), we have

$$\left. \frac{d\mathbf{T}(t)}{dt} \right|_{t=t_x} = \mathbf{A}\mathbf{T}(t_x) + \mathbf{B} \quad (5.32)$$

Replace $\mathbf{T}(t_x)$ based on equation (5.12) into the above, and after simplification, we can derive

$$\left. \frac{d\mathbf{T}(t)}{dt} \right|_{t=t_x} = e^{\mathbf{A}\Delta t} \left. \frac{d\mathbf{T}(t)}{dt} \right|_{t=t_0} \quad (5.33)$$

where $\Delta t = t_x - t_0$. For each core \mathcal{C}_i , from the above equation, we can directly get that

$$\left. \frac{dT_i(t)}{dt} \right|_{t=t_x} = \sum_{j=1}^{N_c} e_{ij} \cdot \left. \frac{dT_j(t)}{dt} \right|_{t=t_0} \quad (5.34)$$

where e_{ij} is the item in the i^{th} row and j^{th} column of matrix $e^{\mathbf{A}\Delta t}$. On one hand, we know that $\forall e_{ij} \geq 0$ ([95]). On the other hand, we have that $\forall \frac{dT_i}{dt}|_{t=t_0} > 0$. Thus, we can derive

$$\forall \mathcal{C}_i \in \mathbb{C} \quad , \quad \frac{dT_i(t)}{dt}|_{t=t_x} > 0 \quad (5.35)$$

Since t_x is an arbitrary time instance within $[t_0, t_1]$, equation (5.35) shows that the temperature of each core will monotonically increase within the interval $[t_0, t_1]$. Similarly, if $\forall \mathcal{C}_i \in \mathbb{C} \quad , \quad \frac{dT_i}{dt}|_{t=t_0} \leq 0$, then we can prove that the temperature of each core will monotonically decrease within $[t_0, t_1]$. Therefore, if inequality (5.30) or (5.31) holds, the temperature of all cores will monotonically change. \square

Based on Lemma 5.4.1, we can easily derive the following property.

Corollary 5.4.2. *Given a multi-core platform \mathbb{C} and a state interval $[t_0, t_1]$, if equation (5.30) or (5.31) holds, then the peak temperature of that interval can be detected at time t_0 or t_1 .*

Proof: According to Lemma 5.4.1, if equation (5.30) or (5.31) holds, then the temperatures of all cores will either monotonically increase or decrease within $[t_0, t_1]$. Thus, the peak temperature must occur at one of the interval boundaries, i.e. t_0 or t_1 . \square

From Corollary 5.4.2, we can see that if the temperature of all cores are simultaneously increasing (decreasing) at t_0 , then the temperature keep monotonically changing, thus we can directly detect the peak temperature at one the interval boundaries, i.e. t_0 or t_1 . However, when the problems is, as shown in Figure 5.2, the temperatures for some cores increase and some others decrease at $t = t_0$, the temperature variation of each core may become more complicated.

Lemma 5.4.3. *Given a multi-core platform \mathbb{C} and a state interval $[t_0, t_1]$, for any core \mathcal{C}_i , $\mathcal{C}_i \in \mathbb{C}$, if it holds*

$$\frac{dT_i(t)}{dt}\bigg|_{t=t_0} \cdot \frac{dT_i(t)}{dt}\bigg|_{t=t_1} < 0 \quad (5.36)$$

then there is one and only one time instance t_x , $t_x \in [t_0, t_1]$, such that

$$\frac{dT_i(t)}{dt}\bigg|_{t=t_x} = 0 \quad (5.37)$$

Proof: We first prove this property under a two-core platform scenario, then extend to any multi-core platform scenario. Consider a two-core platform $\mathbb{C} = \{\mathcal{C}_1, \mathcal{C}_2\}$. Without loss of generality, let $\frac{dT_1(t)}{dt}\big|_{t=t_0} > 0$ and $\frac{dT_1(t)}{dt}\big|_{t=t_1} < 0$. According to Lemma 5.4.1, we must have that $\frac{dT_1(t)}{dt}\big|_{t=t_0} < 0$ (Otherwise, if $\frac{dT_1(t)}{dt}\big|_{t=t_0} > 0$, then both \mathcal{C}_1 and \mathcal{C}_2 will monotonically increase, which contradicts with $\frac{dT_1(t)}{dt}\big|_{t=t_1} < 0$). Then we know there must exist at least one time instance satisfies $\frac{dT_1(t)}{dt} = 0$. Let t_x represent the first time instance within $[t_0, t_1]$ such that

$$t_x = \min\{t \mid \frac{dT_1(t)}{dt} = 0, t \in [t_0, t_1]\} \quad (5.38)$$

If $\frac{dT_2(t)}{dt}\big|_{t=t_x} < 0$, based on Lemma 5.4.1, we know that $T_1(t)$ and $T_2(t)$ will monotonically decrease from t_x to t_1 . Otherwise, if $\frac{dT_2(t)}{dt}\big|_{t=t_x} > 0$, based on Lemma 5.4.1, we would have that $T_1(t)$ and $T_2(t)$ will monotonically increase within $[t_x, t_1]$, which contradicts with our assumption since if equation (5.38) holds, then $\exists \epsilon > 0$ such that $\frac{dT_1(t)}{dt}\big|_{t=t_x+\epsilon} < 0$. Based on the above analysis, we can see that t_x is the unique time instance within $[t_0, t_1]$ such that $\frac{dT_1(t)}{dt} = 0$.

For any multi-core platform with more than 2 cores, from the circuit perspective, when analyzing the temperature of the i^{th} core that satisfies equation (5.36), we can always combine all the other cores as an equivalent core, and then apply the above procedure as what we did for a two-core platform to get the same conclusion. Therefore, if the condition given by equation (5.37) holds, there must exist one and only one time instance t_x within $[t_0, t_1]$, such that $\frac{dT_i(t)}{dt}\big|_{t=t_x} = 0$. \square

5.4.3 Peak Temperature Detection Within A State Interval

Now we discuss how to effectively detect the peak temperature within a state interval on a multi-core platform. In what follows, we first introduce three lemmas to help us to detect the peak temperature under three different cases, respectively. Then we introduce our proposed peak temperature detection algorithm to detect the peak temperature for any state interval on a multi-core platform.

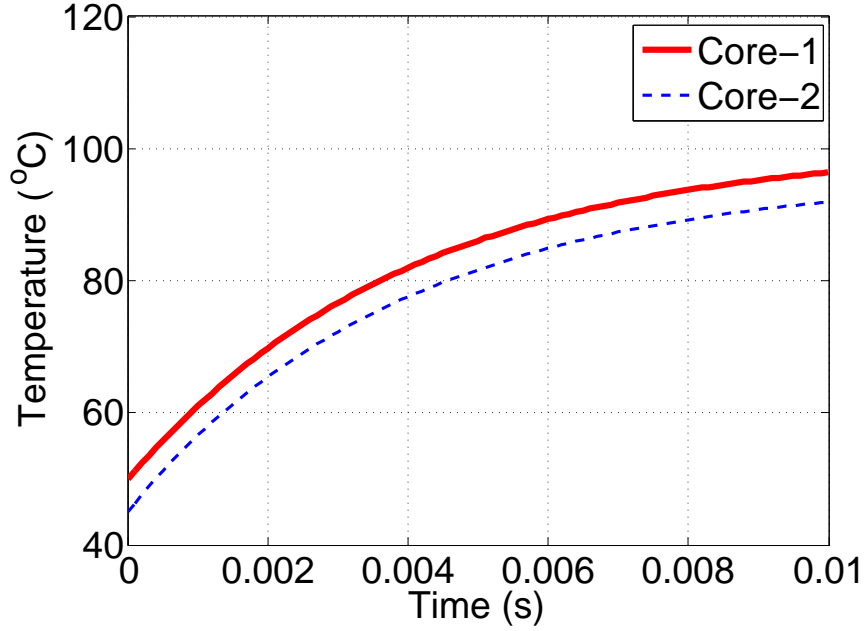


Figure 5.3: $T_i(t)$ increases at both time t_0 and t_1 .

Lemma 5.4.4. *Given a multi-core platform \mathbb{C} and a state interval $[t_0, t_1]$, for any core \mathcal{C}_i , $\mathcal{C}_i \in \mathbb{C}$, if it holds*

$$\frac{dT_i}{dt}\bigg|_{t=t_0} \cdot \frac{dT_i}{dt}\bigg|_{t=t_1} > 0 \quad (5.39)$$

then the peak temperature of core \mathcal{C}_i within that interval can be detected at time t_0 or t_1 .

Proof: Assume that $\frac{dT_i}{dt}\big|_{t=t_0} > 0$ and $\frac{dT_i}{dt}\big|_{t=t_1} > 0$ (see Figure 5.3). Then we prove that $T_i(t)$ will monotonically increase within $[t_0, t_1]$ by contradiction. Assume

there exist a time instance t_x , where $t_x \in [t_0, t_1]$ such that $\frac{dT_i}{dt}|_{t=t_x} < 0$. Then there at least exist two time points t_{y1} and t_{y2} , where $t_{y1} < t_x < t_{y2}$ such that $\frac{dT_i}{dt}|_{t=t_{y1}} = \frac{dT_i}{dt}|_{t=t_{y2}} = 0$, which contradict with Lemma 5.4.3. Thus $T_i(t)$ must monotonically increase within $[t_0, t_1]$. Similarly, if $\frac{dT_i}{dt}|_{t=t_0} < 0$ and $\frac{dT_i}{dt}|_{t=t_1} < 0$, we can get that $T_i(t)$ must monotonically decrease within $[t_0, t_1]$. Therefore, if equation (5.39) holds, the the peak temperature of core \mathcal{C}_i can be directly detected at t_0 or t_1 . \square

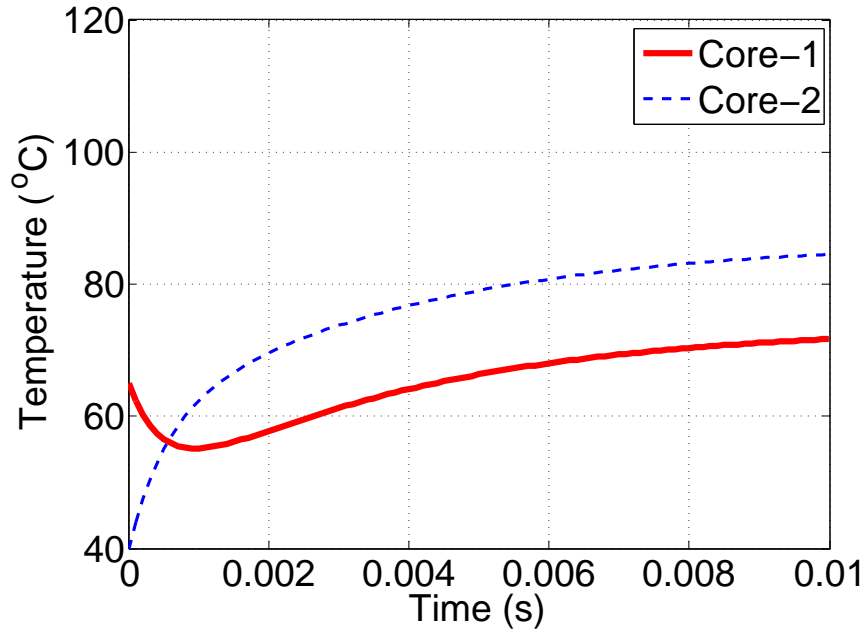


Figure 5.4: $T_i(t)$ decreases at time t_0 and increases at time t_1 .

Lemma 5.4.5. *Given a multi-core platform \mathbb{C} and a state interval $[t_0, t_1]$, for any core \mathcal{C}_i , $\mathcal{C}_i \in \mathbb{C}$, if it holds*

$$\frac{dT_i(t)}{dt}|_{t=t_0} < 0 \quad \text{and} \quad \frac{dT_i(t)}{dt}|_{t=t_1} > 0 \quad (5.40)$$

then the peak temperature of core \mathcal{C}_i within that interval can be detected at time t_0 or t_1 .

Proof: According to Lemma 5.4.3, we know that there exist one and only one time instance $t_x \in [t_0, t_1]$, such that $\frac{dT_i}{dt}|_{t=t_x} = 0$ (see Figure 5.4). Thus, we have

$$\frac{dT_i(t)}{dt}|_{t \in [t_0, t_x)} < 0 \text{ and } \frac{dT_i(t)}{dt}|_{t \in (t_x, t_1]} > 0 \quad (5.41)$$

That means $T_i(t)$ monotonically decreases within $[t_0, t_x]$ and monotonically increases within $(t_x, t_1]$. Thus, the peak temperature of core \mathcal{C}_i within $[t_0, t_1]$ can be detected at time t_0 or t_1 . \square

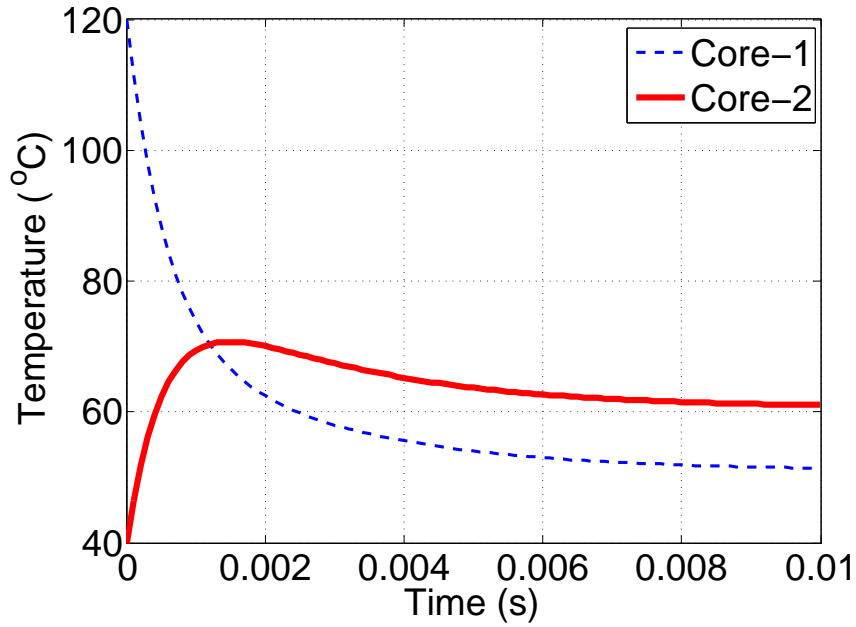


Figure 5.5: $T_i(t)$ increases at time t_0 and decreases at time t_1 .

Lemma 5.4.6. *Given a multi-core platform \mathbb{C} and a state interval $[t_0, t_1]$, for any core \mathcal{C}_i , $\mathcal{C}_i \in \mathbb{C}$, if it holds*

$$\frac{dT_i}{dt}|_{t=t_0} > 0 \quad \text{and} \quad \frac{dT_i}{dt}|_{t=t_1} < 0 \quad (5.42)$$

then the peak temperature of core \mathcal{C}_i within that interval can be determined by using the traditional binary search method.

Proof: According to Lemma 5.4.3, we know that there exist one and only one time instance $t_x \in [t_0, t_1]$, such that $\frac{dT_i}{dt}|_{t=t_x} = 0$ (see Figure 5.5). Thus, peak temperature of core \mathcal{C}_i within that interval must occur at time t_0 , t_c or t_1 . By applying the traditional binary search method [32], we can easily get the peak temperature of core \mathcal{C}_i within that interval. \square

Based on Lemma 5.4.4- 5.42, we introduce our peak temperature detection algorithm, denoted as *TPeak_Detection*, to obtain the peak temperature among all cores within a state interval.

Algorithm 6 TPeak_Detection

Require:

- 1) $[t_0, t_1]$: a constant speed interval;
 - 2) \mathbf{T}_{t_0} : temperature vector at time t_0 ;
 - 3) κ : $\kappa = \{k_1, k_2, \dots, k_{N_c}\}$, where k_i is the i^{th} core's processing mode;
 - 1: $\mathbf{T}_{t_1} = e^{\mathbf{A}_\kappa \Delta t} \mathbf{T}_0 + \mathbf{A}_\kappa^{-1}(e^{\mathbf{A}_\kappa \Delta t} - \mathbf{I})\mathbf{B}_\kappa$, where $\Delta t = t_1 - t_0$;
 - 2: $\frac{d\mathbf{T}(t)}{dt}|_{t=t_0} = \mathbf{A}_\kappa \mathbf{T}_{t_0} + \mathbf{B}_\kappa$ and $\frac{d\mathbf{T}(t)}{dt}|_{t=t_1} = \mathbf{A}_\kappa \mathbf{T}_{t_1} + \mathbf{B}_\kappa$;
 - 3: $T_{max} = \max\{T_i(t) \mid t = t_0, t_1; i = 1, 2, \dots, N_c\}$;
 - 4: **for** $i = 1$ to N_c **do**
 - 5: **if** $\frac{dT_i(t)}{dt}|_{t=t_0} > 0, \frac{dT_i(t)}{dt}|_{t=t_0} < 0$ **then**
 - 6: T_i^{peak} = calculate the peak temperature of core \mathcal{C}_i according to Lemma 5.4.6;
 - 7: $T_{max} = \max\{T_{max}, T_i^{peak}\}$;
 - 8: **end if**
 - 9: **end for**
 - 10: **return** T_{max}
-

Based on Algorithm 6, we can derive the following theorem.

Theorem 5.4.7. *Given a multi-core platform \mathbb{C} and a state interval $[t_0, t_1]$ with processing mode as κ , where $\kappa = \{k_1, k_2, \dots, k_{N_c}\}$. If the starting temperature \mathbf{T}_{t_0} is known, then the maximum temperature (T_{max}) among all cores within $[t_0, t_1]$ can be obtained by*

$$T_{max} = TPeak_Detection(t_0, t_1, \mathbf{T}_{t_0}, \kappa); \quad (5.43)$$

Proof: For each core \mathcal{C}_i , there are four different cases with respect of $(\frac{dT_i}{dt}|_{t=t_0}, \frac{dT_i}{dt}|_{t=t_1})$: 1) $(> 0, > 0)$; 2) $(< 0, < 0)$; 3) $(< 0, > 0)$; 4) $(> 0, < 0)$. For the case 1, 2 and 3, according to Lemma 5.4.4 and Lemma 5.4.5, we can always detect the peak temperature of core \mathcal{C}_i at t_0 or t_1 . For the last case, i.e. case 4, the corresponding peak temperature can be detected by Lemma 5.4.6. By applying Algorithm 6, all the above four cases will be considered (see line 3,6 and 7). Thus, the temperature returned by Algorithm 6 must be the maximum temperature among all cores within $[t_0, t_1]$. \square

Theorem 5.4.7 proves a way to effectively calculate the maximum temperature within any constant speed interval (i.e. a state interval). In what follows, we further discuss how to detect the peak temperature for a periodic speed schedule.

5.4.4 Peak Temperature Detection For A Periodic Schedule

Our goal is to ensure the peak temperature of the entire processor does not exceed T_{max} during its life time. Since the peak temperature of a state interval does not necessary occur within the first scheduling period, where the system-wide peak temperature can be? To sample temperatures using very short sampling period, from the start to the time when system reaches the stable status, does not seem to be a promising solution, given its prohibitive complexity. Fortunately, we have the following theorem help us to limit the peak temperature at the system steady state.

Theorem 5.4.8. *Given a multi-core platform \mathbb{C} and a periodic speed schedule \mathcal{S} , if all cores start from the ambient temperature, then when repeating \mathcal{S} , the peak temperature occurs when temperature of the platform reaches its steady state, i.e.*

$$\max\{\mathbf{T}(t) | t \in [0, nL], n \mapsto +\infty\} = \max\{\mathbf{T}_{ss}(t) | t \in [0, L]\} \quad (5.44)$$

Proof: Similarly with the proof of Theorem 5.3.2, for any counterpart time point t_q in two successive scheduling periods, i.e. the $(n-1)^{th}$ period and the n^{th} period, where $n \geq 2$, we can derive that

$$\mathbf{T}(nL + t_q) - \mathbf{T}((n-1)L + t_q) = \mathbf{K}_q^n \cdot (\mathbf{T}(L) - \mathbf{T}(0))$$

If every core starts from the ambient temperature, i.e. $\mathbf{T}(0) = (T)_{amb}$, then we have $T_i(L) - T_i(0) \geq 0, \forall \mathcal{C}_i \in \mathbb{C}$. Moreover, we also know that every item in matrix \mathbf{K}_q^n is non-negative, i.e. $e_{ij} \geq 0$ holds for $\forall e_{ij} \in \mathbf{K}_q^n$. Thus, $T_i(nL + t_q) - T_i((n-1)L + t_q) \geq 0$. That means the temperature of a relative time point within any two successive periods will monotonically increase until the system achieves the steady state. Thus we can always determine the system peak temperature within the steady state. According to our steady-state temperature formation shown in subsection 5.3.3, we can easily derive the result as given by equation (5.44). \square

Note that when a processor runs with a periodic schedule and reaches the temperature steady status, it does not mean that the temperature for each core remains constant. Instead, it means that the temperature at starting points for all scheduling periods are the same. To ease our effort in determining the feasibility under peak temperature constraint, we next introduce three effective methods to test the feasibility for a multi-core periodic schedule.

5.5 Feasibility Analysis For Multi-Core Scheduling With Temperature Constraint

We are now ready to present our feasibility checking methods. In this section, we present three methods to check if a periodic speed schedule can satisfy a given maximum temperature constraint.

5.5.1 TmaxCheck: Feasibility Checking With Initial Temperature As T_{max}

As mentioned before, a schedule that is feasible for the first scheduling period under T_{max} cannot ensure that the maximum temperature constraint will not be violated later. This is because that, after the first scheduling period, the cores need to run at a higher initial temperatures, and thus potentially reaches even higher temperatures. However, if for all the cores, the ending temperature of the first scheduling period is no greater than the initial temperature, a feasible schedule within the first scheduling period can ensure that the system feasibility in the future. We call this feasibility checking method as **TmaxCheck**, and formally formulate it in the following theorem.

Theorem 5.5.1. *Given a periodic speed schedule \mathbb{S} , if starting with initial temperature as T_{max} for all cores, \mathbb{S} is feasible under T_{max} for each state interval within $[0, L]$ by applying Algorithm 6, then when repeating \mathbb{S} later, the temperature will never exceed T_{max} .*

Theorem 5.5.1 can be proved by simply noting that, since $T_i(L) \leq T_{max}$ for any core \mathcal{C}_i , thus $T_i(L) \leq T(0)$, according to Theorem 5.3.1, the second scheduling period always starts at the same or a more favorable situation for core \mathcal{C}_i . Thus, the temperature constraint will not be violated in the future. In addition, the feasibility under T_{max} within the first scheduling period, i.e. $[0, L]$, can be easily determined by applying Algorithm 6 to each state interval within $[0, L]$. One problem for this approach, however, is that it is very pessimistic to assume all cores have to start with T_{max} simultaneously. Therefore, the efficiency of Theorem 5.5.1 is limited.

5.5.2 ModeCheck: Feasibility Checking With Temperature

Safe Modes

According to our system models given in Section 5.2, each core can run in different modes. For an arbitrary processing core \mathcal{C}_i , there may exist certain processing modes such that if all other cores do not exceed the temperature constraint (i.e. T_{max}), then \mathcal{C}_i will never exceed T_{max} . We call such modes as the *safe modes* of core \mathcal{C}_i . Once we have identified the safe modes of all cores, we can predict the feasibility of a speed schedule by checking whether all processing modes of each core belong to its safe mode set. For the rest of this subsection, we first discuss how to identify the safe modes of each core, then we present a new feasibility checking method based on the identified safe modes.

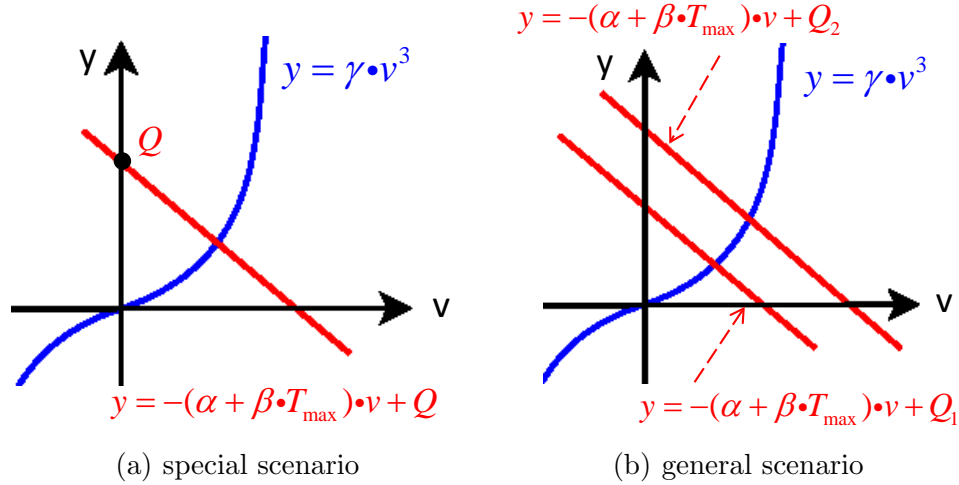


Figure 5.6: Equilibrium voltage of core \mathcal{C}_i under processing mode k_i . a) all other cores except \mathcal{C}_i are under fixed constant processing modes; b) all other cores except \mathcal{C}_i are under any arbitrary available processing modes.

Without loss of generality, we consider an arbitrary core \mathcal{C}_i and one of its processing modes k_i . In order to determine mode k_i is a safe mode of core \mathcal{C}_i , we need to guarantee that the temperature of \mathcal{C}_i always stays below T_{max} under that specific

mode. Recall that the thermal model of core \mathcal{C}_i is given by equation (5.6). According to equation (5.6), the power constant consumption needed for core \mathcal{C}_i 's temperature to saturate at T_{max} can be obtained by

$$\left. \frac{dT_i(t)}{dt} \right|_{t=\tilde{t}} = 0 \quad (5.45)$$

where \tilde{t} is a time point such that $T_i(\tilde{t}) = T_{max}$. By applying the above into equation (5.6), we can derive

$$\gamma_{k_i} \cdot v_{k_i}^3 = -(\alpha_{k_i} + \beta_{k_i} \cdot T_{max}) \cdot v_{k_i} + Q_i \quad (5.46)$$

where

$$Q_i = \frac{T_{max} - T_{amb}}{R_{ii}} + \sum_{j \neq i} \frac{T_{max} - T_j}{R_{ij}} \quad (5.47)$$

Note that equation (5.46) is the classic *depressed cubic equation* [94] with respect of v_{k_i} . Since $\gamma_{k_i} > 0$ and $-(\alpha_{k_i} + \beta_{k_i} \cdot T_{max}) < 0$, equation (5.47) has only one single real root for v_{k_i} under a fixed constant speed configuration for all other processors, which can be solved analytically (see Figure 5.6(a)). We call the solution of equation (5.47) under a fixed speed configuration as an *equilibrium voltage* of core \mathcal{C}_i , denoted by \tilde{v}_{k_i} . Thus, we see if $v_i \leq \tilde{v}_{k_i}$ and all other cores are under that corresponding speed configuration, then the temperature of core \mathcal{C}_i will never exceed T_{max} .

Moreover, from our thermal model of core \mathcal{C}_i (given by equation (5.6)), we can observe that the higher the temperatures of all other cores, the higher the rate that core \mathcal{C}_i can increase its own temperature. Thus, the worst-case equilibrium voltage of core \mathcal{C}_i under processing mode k_i must occur when the temperatures for the rest of the cores are all T_{max} , i.e. $T_j(t) = T_{max}$ for $\forall j, j \neq i$. From Figure 5.6(b), we can also see that when Q_i achieves the minimum value (which means all other cores except \mathcal{C}_i achieve T_{max}), the equilibrium voltage of \mathcal{C}_i will achieve its minimum (worst-case) value.

$$Q_i^{WC} = \frac{T_{max} - T_{amb}}{R_{ii}} \quad (5.48)$$

We call the solution of equation (5.46) under the above condition, i.e. equation (5.48), as a *worst-case equilibrium voltage* of core \mathcal{C}_i under processing mode k_i , and denote it as \tilde{v}_{k_i} . In fact, \tilde{v}_{k_i} is the maximal voltage that can keep the temperature of \mathcal{C}_i always staying below T_{max} under the coefficients of that specific mode. Then we formally define the *safe mode* for a core as follows.

Definition 5.5.2. *Let \tilde{v}_{k_i} be the worst-case equilibrium voltage of core \mathcal{C}_i under processing mode k_i . Then the processing mode k_i is a safe mode of core \mathcal{C}_i if $v_{k_i} \leq \tilde{v}_{k_i}$.*

With the help of Definition 5.5.2, we can establish another feasibility checking method, called **ModeCheck**, in the following theorem.

Theorem 5.5.3. *Given a periodic speed schedule \mathbb{S} , if the highest speed in \mathbb{S} of each core belongs to one of its safe modes, then temperature will never exceed T_{max} .*

Proof: The proof is done by contradiction. Consider an arbitrary core \mathcal{C}_i , and assume that at time $t = t_0$ we have that $T_i(t) = T_{max}$ but at time $t = t_0 + \Delta t$, we have $T_i(t) > T_{max}$, while temperatures of the rest of the cores are all below T_{max} . Based on our assumption, we must have that

$$\left. \frac{dT_i(t)}{dt} \right|_{t=t_0} = \left. \frac{dT_i(t)}{dt} \right|_{T_i(t)=T_{max}} > 0 \quad (5.49)$$

On the other hand, from equation (5.6), we can derive that

$$C_i \cdot \left. \frac{dT_i(t)}{dt} \right|_{t=t_0} = \gamma_{k_i} \cdot v_{k_i}^3 + (\alpha_{k_i} + \beta_{k_i} \cdot T_{max}) \cdot v_{k_i} - Q_i \quad (5.50)$$

where $Q_i = \frac{T_{max}-T_{amb}}{R_{ii}} + \sum_{j \neq i} \frac{T_{max}-T_j}{R_{ij}}$. Since core \mathcal{C}_i runs at a safe mode and all other cores are below T_{max} , we have $v_{k_i} \leq \tilde{v}_{k_i}$ and $Q_i \geq \frac{T_{max}-T_{amb}}{R_{ii}} = Q_i^{WC}$. Apply these to the above, we get

$$C_i \cdot \left. \frac{dT_i(t)}{dt} \right|_{t=t_0} \leq \gamma_{k_i} \cdot \tilde{v}_{k_i}^3 + (\alpha_{k_i} + \beta_{k_i} \cdot T_{max}) \cdot \tilde{v}_{k_i} - Q_i^{WC} \quad (5.51)$$

Replace the right side of the above with equation (5.46), we can derive that $\frac{dT_i(t)}{dt}\big|_{t=t_0} \leq 0$, which contradicts equation (5.49). \square

Note that, as long as the maximum temperature T_{max} and the multi-core platform are given, the safe modes for each core can be readily determined. This method is particular useful when it is much less costly to get the maximum speed of core C_i in a schedule (such as those generated by the approach in [120]) rather than to get the entire speed schedule for periodic tasks on each core. At the same time, this feasibility condition is still only a sufficient condition. In other word, this method cannot be applied to the scenarios when the maximum core speed is higher than the maximum safe speed. In what follows, we introduce another much stronger feasibility checking method.

5.5.3 TssCheck: Feasibility Checking With Steady-State Temperature Formula

Before introduce our third method, we define an operator, denoted by \preceq , such that $\mathbf{T}(t_q) \preceq \mathbf{T}(t_p)$ is equivalent to $T_i(t_q) \leq T_i(t_p)$ for $i = 1, 2, \dots, m$. Then we present a new feasibility checking method, called **TssCheck**, which provides a much strong condition for checking the feasibility of a periodic speed schedule with maximum temperature constraint.

Theorem 5.5.4. *Given a periodic speed schedule \mathbb{S} , when repeating \mathbb{S} later, the temperature will never exceed T_{max} if for any state interval within $[0, L]$, i.e. $\forall [t_{q-1}, t_q] \subseteq [0, L]$, the following conditions hold:*

- $|\lambda_i| < 1$, for each eigenvalue λ_i of K ;
- $\mathbf{T}(t_q^*) \preceq \mathbf{T}_{max} - \mathbf{K}_{q*} \cdot (\mathbf{I} - \mathbf{K})^{-1} \cdot (\mathbf{T}(L) - \mathbf{T}(0))$

where $\mathbf{T}(t_q^*)$ is the maximum temperature within $[t_{q-1}, t_q]$ and t_q^* is the corresponding time instance, both of which can be obtained by applying Algorithm 6.

Proof: The idea is to make sure T_{max} is not violated within any state interval during the system stable status. Consider an arbitrary state interval $[t_{q-1}, t_q]$, let $\mathbf{T}(t_q^*)$ and t_q^* be maximum temperature and the corresponding time instance, both of which can be obtained by applying Algorithm 6. Then according to Theorem 5.3.3, the system temperatures can achieve a stable state iff $|\lambda_i| < 1$, for each eigenvalue λ_i of K . In this condition, $(I - K)$ is invertible, then we have

$$\mathbf{T}_{ss}(t_q^*) = \mathbf{T}(t_q^*) + \mathbf{K}_{q^*} \cdot (\mathbf{I} - \mathbf{K})^{-1} \cdot (\mathbf{T}(L) - \mathbf{T}(0))$$

Thus, $\mathbf{T}_{ss}(t_q^*) \preceq \mathbf{T}_{max}$, if and only if

$$\mathbf{T}(t_q^*) \preceq \mathbf{T}_{max} - \mathbf{K}_{q^*} \cdot (\mathbf{I} - \mathbf{K})^{-1} \cdot (\mathbf{T}(L) - \mathbf{T}(0))$$

□

From Theorem 5.5.4, we can determine the feasibility of a given speed schedule by checking the temperatures only at the end of all state intervals within the first scheduling period. In the next section, we conduct experiments to evaluate our proposed temperature calculation technique and feasibility checking methods.

5.6 Experimental Evaluations

In this section, we present a detailed discussion for our experimental evaluation of the proposed techniques and feasibility checking methods.

Our multi-core platform consists of total nine homogenous processing cores placed as a 3×3 mesh. As the goal of this work is to study system level thermal behavior of each core on a multi-core platform, we simplified the granularity of the

floorplan to core level. Based on the processor model proposed in [79], we adopted 65nm technology node and applied the corresponding chip parameters. Specifically, we used HotSpot-5.02 [108] as the reference thermal model, to compare the accuracy and computational effectiveness of our temperature calculation method. We showed the configuration parameters of HotSpot and our thermal model in Table 5.1 and Table 5.2, respectively.

Table 5.1: HotSpot parameters and floorplan

Parameter	Value
Total Cores	9 (3x3)
Area per Core	4 mm^2
Die Thickness	0.15 mm
Heat Spreader Side	20 mm
Heat Sink Side	30 mm
Convection Resistance	0.1 K/W
Convection Capacitance	140 J/K
Ambient Temperature	35°C
Sampling Interval	10 ms

In the rest of this section, we first present an accuracy analysis for our temperature calculation approach. Then we evaluate the performance of our proposed three feasibility checking methods.

5.6.1 Accuracy Analysis Of Our Analytical Temperature Calculation Method

We first examined the accuracy of the proposed system level analytical temperature calculation method with HotSpot simulator. For each core, we selected a single constant speed within [0.6 : 0.05 : 1.3]. Then we let all cores run under the selected speed(s) by 10 seconds. To calculate and compare the temperature traces obtained by our method and HotSpot, we assumed a sampling interval as 10ms, and further

Table 5.2: Power/thermal parameters

$V_{dd}(V)$	α	β	γ
0.00	0	0	0
0.60	0.0012	0.0098	7.2564
0.65	0.0224	0.0103	7.2564
0.70	0.0493	0.0109	7.2564
0.75	0.0838	0.0114	7.2564
0.80	0.1282	0.0120	7.2564
0.85	0.1857	0.0126	7.2564
0.90	0.2607	0.0133	7.2564
0.95	0.3591	0.0140	7.2564
1.00	0.4890	0.0147	7.2564
1.05	0.6611	0.0154	7.2564
1.10	0.8904	0.0162	7.2564
1.15	1.1972	0.0170	7.2564
1.20	1.6091	0.0178	7.2564
1.25	2.1640	0.0187	7.2564
1.30	2.9135	0.0197	7.2564

assumed one LCM contained 10 sampling intervals (note that each sampling interval was a state interval since each core run under a constant speed). On one hand, we calculated one temperature trace (which contained $10s/(10 * 10ms) = 100$ LCMs) by our proposed analytical method shown in equation (5.23). On the other hand, we generated the corresponding power trace under our sampling interval length (i.e. $10ms$), and then used HotSpot to obtain another temperature trace. We compared two temperature traces with respect of each core, and based on different physical locations on a 3x3 mesh (i.e. corner core, boundary core and the central core), we plotted three groups of results in Figure 5.7.

Figure 5.7 shows the experimental result of the accuracy of our proposed temperature calculation method (given by equation (5.23)) under voltage $1.3V$. From Figure 5.7(a), 5.7(b) and 5.7(c), we can see that the temperature calculated by our analytical method matches closely with HotSpot's simulation result. For example, the maximum temperature error between HotSpot and our method is $2.27^{\circ}C$ in

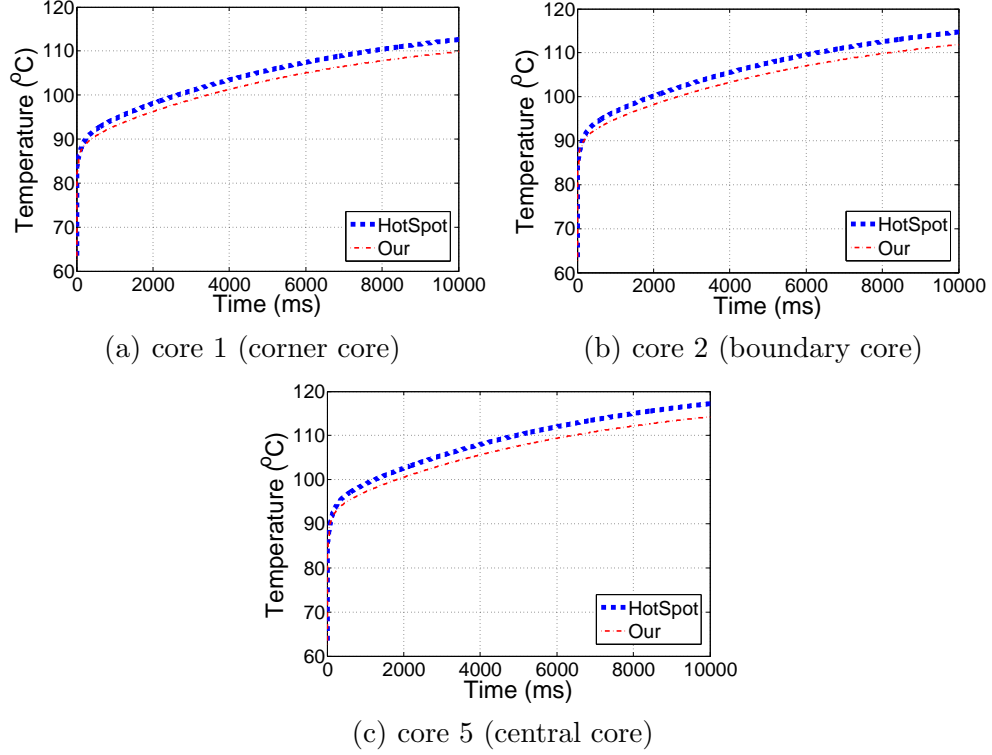


Figure 5.7: Accuracy analysis of our proposed temperature calculation method.

Figure 5.7(a), $2.37^{\circ}C$ in Figure 5.7(b) and $2.5^{\circ}C$ in Figure 5.7(c).

Moreover, our proposed method outperforms HotSpot simulation on computational time for calculating temperature traces with the same length. In this experiment, we observed that our proposed method was at least 100 times faster than HotSpot, 0.169s versus 18.825s for the the experiment conducted in Figure 5.7. This is because the proposed method determines temperature based on temperature within the first scheduling period only, while HotSpot requires to repeat periodic iterations until the end time of the entire schedule.

5.6.2 Steady-State Peak Temperature Variation Under Different Constant Speeds

In this experiment, we studied the peak temperature variation in the thermal stable state (also called system steady state) under different constant speeds. We let all cores run at the same speed/voltage from $0.6V$ to $1.3V$, and used our closed-form steady-state temperature computation method, given by equation (5.25), to quickly calculate the steady-state temperature of all cores. In order to apply our steady-state temperature formula given by equation (5.25), we assumed one LCM contained 10 state intervals, each of which was with a length of $10ms$. And we assumed all cores started at the ambient temperature, i.e. $35^{\circ}C$. After computing the steady-state temperature under each constant speed, we recorded the maximal peak temperature among all cores, and plotted the corresponding result in Figure 5.8.

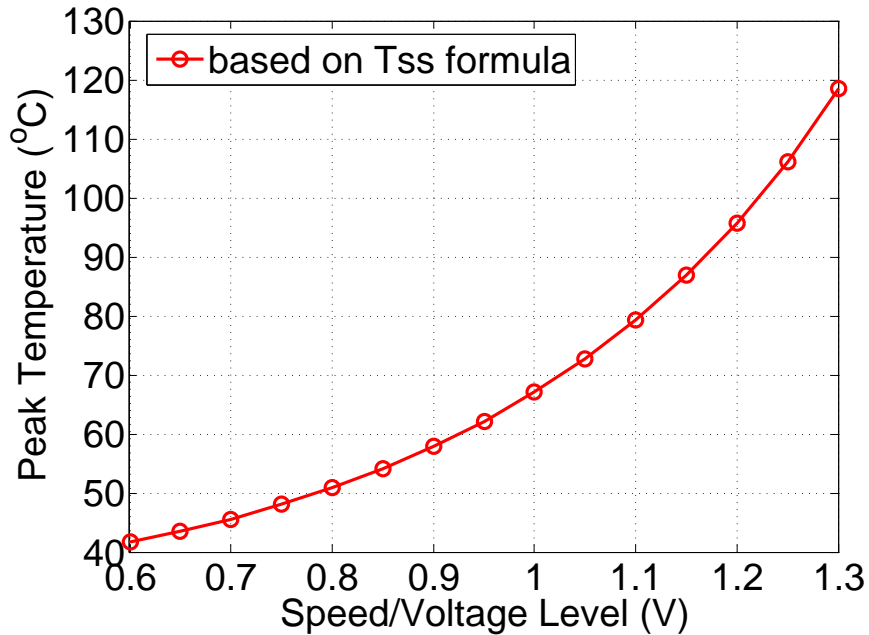


Figure 5.8: Steady-state temperature under different constant speeds with our analytical steady-state temperature formula.

Figure 5.8 shows the trend of steady-state peak temperature for a 3×3 multi-core system under different single speeds/voltages. From Figure 5.8, we can clearly observe the value of steady-state peak temperature under each speed/voltage level. For example, in Figure 5.8, when the speed/voltage level was set to 1.3V, the peak temperature of the entire system could reach almost up to $120^{\circ}C$. Figure 5.8 can help us to quickly get an intuition of the peak temperature variation under different speed/voltage levels.

5.6.3 Threshold Temperature Determined By TmaxCheck

Recall that in Section 5.5.1, we proposed a feasibility checking method, called **TmaxCheck**, to predict the schedulability of a given schedule by initializing all cores with certain pre-defined temperature constraint, i.e. T_{max} . In steady of giving a T_{max} to determine the schedulability of a schedule, given a single speed schedule, we can predict a feasible T_{max} , called *threshold temperature*, which is the lowest temperature that can be used as the temperature constraint without any violation for that speed schedule. In the rest of this part, we studied the property of threshold temperature by varying the speed/voltage level from 0.6V to 1.3V with a step width of 0.05V. Under each speed/voltage level, we applied **TmaxCheck** method to find the threshold temperature by enumerating T_{max} within $[40^{\circ}C, 130^{\circ}C]$. The corresponding result was collected and plotted in Figure 5.9.

Figure 5.9, similar to Figure 5.8 also gives an intuition of a boundary of the maximal temperature that can be safely used as the temperature constraint for a single speed schedule.

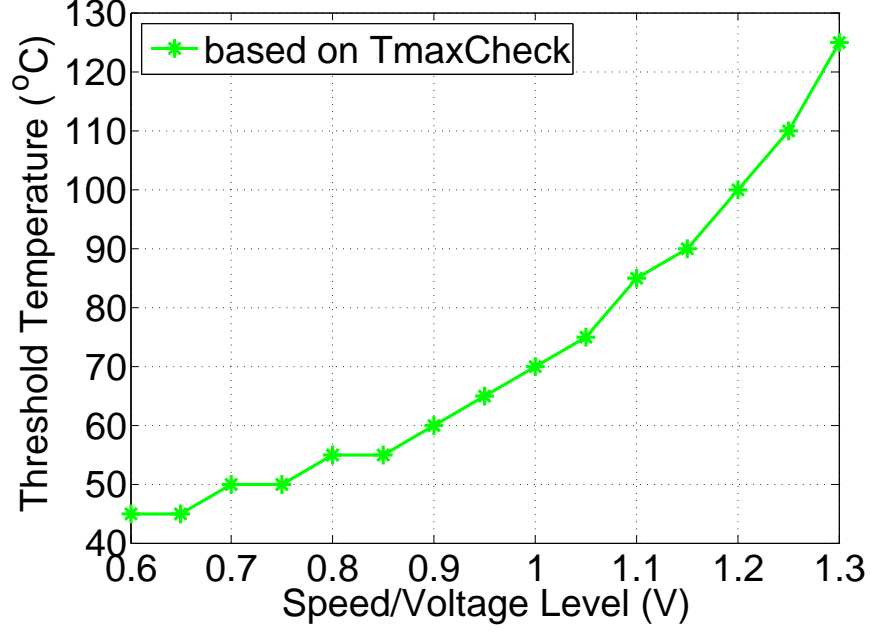


Figure 5.9: Threshold temperature determined by TmaxCheck under different voltages.

5.6.4 Worst-Case Equilibrium Voltage Determined By ModeCheck

The **ModeCheck** method, as one of our proposed feasibility checking methods, is one and the only one method that determines the feasibility of a schedule without any computation of temperature calculation. Regardless of what speed schedule is given, once the temperature constraint (T_{max}) is determined, the **ModeCheck** method can compute a *worst-case equilibrium voltage* (see Section 5.5.2), which represents the maximum safety voltage that can be used under T_{max} . This worst-case equilibrium voltage can be further used to decide whether an available processing mode is safe or not. In this experiment, we will analyze the relationship between the temperature constraint and worst-case equilibrium voltage.

We varied the maximum temperature constraint (T_{max}) from $40^{\circ}C$ to $130^{\circ}C$ with an increment of $5^{\circ}C$. Under each temperature constraint, we applied the

approach introduced in **ModeCheck** method (see Section 5.5.2) to calculate the corresponding worst-case equilibrium voltage.

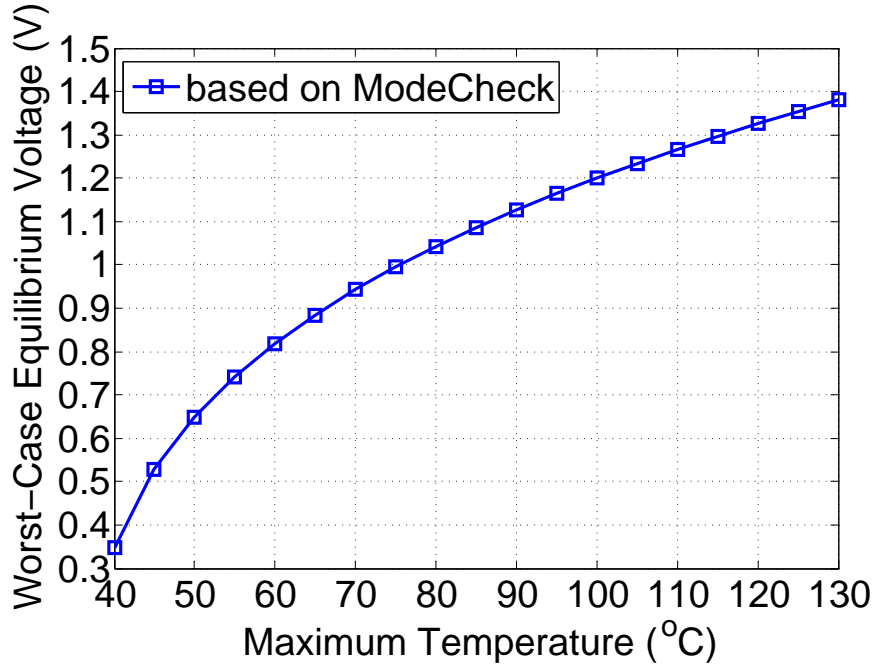


Figure 5.10: Worst-case equilibrium voltage determined by ModeCheck under different maximum temperature constraints.

Figure 5.10 shows the experimental result of the worst-case equilibrium voltage with respect of the maximum temperature constraint. From Figure 5.10, we can directly find the threshold voltage under each temperature constraint, and thus further determine the safe processing mode for the entire multi-core system under that specific temperature constraint. For example, in Figure 5.10, when the maximum temperature was set to 90°C , the corresponding worst-case equilibrium voltage was 1.13V , thus processing modes with voltage as 0.6V , 0.65V , 0.7V , ..., 1.10V are all safe modes.

5.6.5 Performance Comparison For Different Feasibility Checking Methods

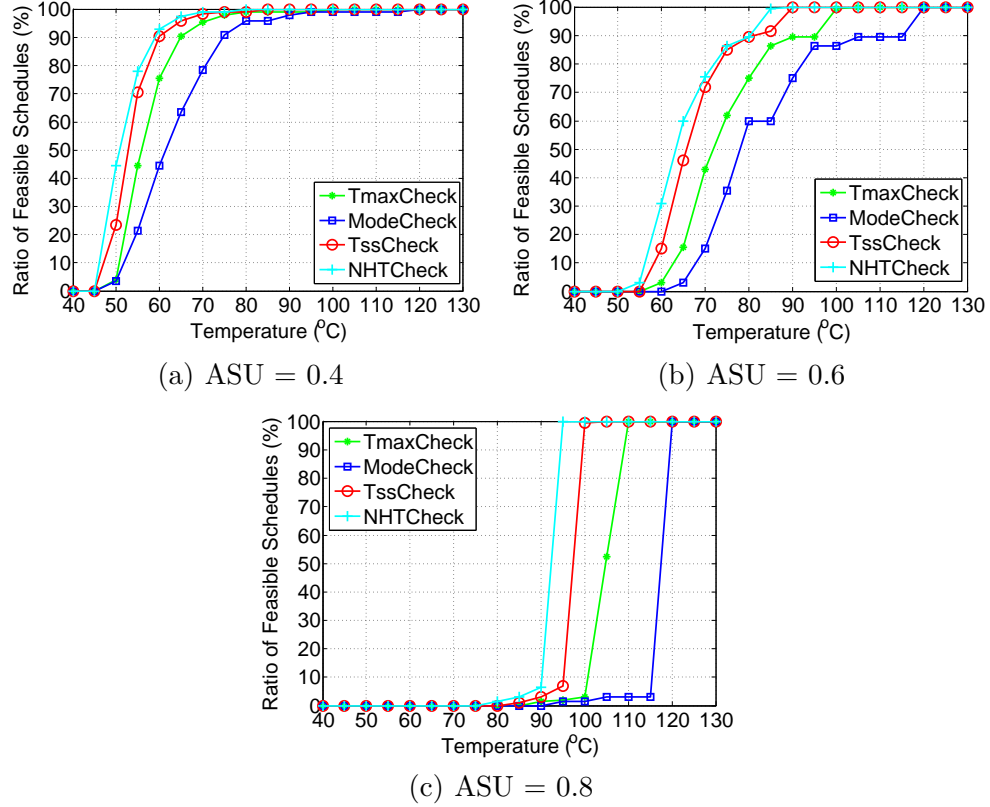


Figure 5.11: Feasibility ratios under different maximum temperatures.

We then evaluated the performance of our proposed feasibility checking methods in this part. Our proposed three feasibility checking methods were studied: **TmaxCheck** (see Theorem 5.5.1), **ModeCheck** (see Theorem 5.5.3) and **TssCheck** (see Theorem 5.5.4). Moreover, we conducted another *no heat transfer* feasibility checking method, denoted as **NHTCheck**, which ignored any heating transfer among any two cores and thus tested the feasibility of each core isolatedly as a single-core scenario. **NHTCheck** was used as a cross-reference for our proposed three feasibility checking methods. Three sub-experiments were conducted under different

average system utilizations (ASU), i.e. ASU is 0.4, 0.6 and 0.8. Within each sub-experiment, we generated 200 periodic speed schedules, each of which was composed of a group of state intervals with lengths evenly distributed within $[50ms, 200ms]$ and total utilization was equal to the ASU of that sub-experiment. The ambient temperature (T_{amb}) was set to $35^{\circ}C$, and the maximum temperature constraint was varied from $40^{\circ}C$ to $130^{\circ}C$ with an increment of $5^{\circ}C$. We collected the numbers of feasible task sets by different methods and computed the ratio of feasible task sets over the entire test cases. The corresponding result was plotted in Figure 5.11.

Figure 5.11 shows the feasibility checking results of different approaches under three configurations of average systems utilization, i.e. ASU is 0.4, 0.6 and 0.8. From Figure 5.11, we can see that **NHTCheck** would incorrectly predict some infeasible schedules as feasible compared with **TssCheck**. For example, in Figure 5.11(a), when the temperature constraint was set to $50^{\circ}C$, **TssCheck** gave a feasible ratio no more than 25%, while **NHTCheck** predicted feasible schedules up to 45%. This is because **NHTCheck** ignores the heat transfer among different cores, which lead to a lower temperature result during the temperature calculation, and finally results to let some infeasible schedules pass the feasibility test.

From Figure 5.11, we can also observe that **TssCheck** performs better than **TmaxCheck** and **ModeCheck**. For example, in Figure 5.11(b), when the maximum temperature was set to $70^{\circ}C$, **TssCheck** presented a feasible checking result up to 72% of the original schedules, while **TmaxCheck** and **ModeCheck** only predicted 43% and 15% feasible schedules, respectively. Moreover, from Figure 5.11(a), 5.11(b) and 5.11(c), we can further observe that with the increment of average system utilization, the feasibility under each temperature constraint had a dramatic drop for all three approaches. For example, when the maximum temperature was set to $70^{\circ}C$, the ratio of feasible schedules by **TssCheck** was 98% in Figure 5.11(a), 72%

in Figure 5.11(b) and 0% in Figure 5.11(c). This is because under DVFS scheduling, the average processor running speed for light loaded system (e.g. ASU is 0.4) will be lower than that of heavy loaded system (e.g. ASU is 0.6 or 0.8). As we know, the overall power consumption and the peak temperature under higher system utilization will be larger than that under lower system utilization. Thus, under the same maximum temperature constraint, the system feasibility could significantly decrease with the increment of system utilization.

Through this experimental result, we can also conclude that temperature factor does have an important impact on system feasibility for multi-core real-time scheduling.

5.7 Summary

Feasibility checking problem is one of the most fundamental problems in real-time embedded system design. This problem becomes much more difficulty and challenging when shifting from single-core platforms to multi-core platforms, particularly under maximum temperature constraint.

This work studies the feasibility checking problem for real-time periodic speed schedule on multi-core platforms under maximum temperature constraint. We present a number of novel analytical solutions for temperature calculation, i.e. the temperatures within an arbitrary LCM and the thermal steady-state. Then we propose a complete solution to effectively and efficiently detect the peak temperature for a periodic speed schedule on a multi-core platform. We further introduce three temperature-constrained feasibility checking methods, i.e. **TmaxCheck**, **ModeCheck** and **TssCheck**. Our proposed techniques form the basis of more advanced thermal aware real-time embedded system design on multi-core platforms.

CHAPTER 6

LEAKAGE-AWARE ENERGY ESTIMATION FOR MULTI-CORE REAL-TIME SCHEDULING

In this chapter, we further extend our research to energy minimization problem with thermal impacts taken into considerations. One of the fundamentals in energy efficiency design is to calculate the energy consumption effectively and efficiently for a design alternative. To accurately and also quickly estimate the energy consumption for a voltage/frequency scheduling on multi-core platforms, there are two major challenges: 1) how to address the interdependency of leakage and temperature appropriately, and 2) how to deal with the heat transfer among different processing cores. First, by considering the leakage/temperature dependency, the leakage power consumption (and thus the overall power consumption) varies with the temperature, and temperature changes with the power consumption as well. This interdependency between leakage and temperature makes the power calculation, thus the energy calculation, much complicated and difficult. Second, by further considering the heat transfer among different cores, the solution of power consumption becomes even more challenging, i.e. leading to the problems of matrix exponential operation and its corresponding integration, which may not always have explicit analytical solutions. In this chapter, we present a fast and accurate method to calculate the overall energy consumption for a given voltage schedule on multi-core platforms. Different from the traditional numerical method for energy calculation, we develop a closed-form analytical solution for the overall energy consumption under a given multi-core schedule.

6.1 Related Work

A key problem in energy efficiency design is to calculate the energy consumption for a design alternative. Earlier research, e.g. [119, 76], has been exclusively focused on dynamic energy consumption. Some later research such as that in [62] takes the leakage power into consideration, but assumes that leakage power is constant. Under this assumption, the calculation of energy consumption for a given voltage schedule is trivial, since the overall power consumption remains the same as long as a system keeps the same running voltage and frequency. However, when considering the leakage/temperature dependency, the problem substantially becomes more challenging since the leakage power consumption (and thus the overall power consumption) varies with the temperature, and temperature changes with the power consumption as well. The energy calculation problem becomes even more complicated for multi-core platforms when the leakage power of one core depends not only on its own temperature, but also temperatures from other cores as well.

To calculate the overall energy consumption with leakage/temperature dependency taken into consideration, one intuitive and commonly adopted approach is to use the numerical method. According to this method, the entire voltage schedule is split into a set of small time intervals such that within each interval the voltage/frequency and temperature of all cores can be regarded as invariant. The temperature and power trace, and thus the energy consumption, for a schedule can be obtained accordingly. For example, Liu *et al.* [86] formulated the energy minimization under a peak temperature as a non-linear programming problem, and then employed the above mentioned method to calculate the energy consumption. Bao *et al.* [17] also used the similar approach to keep track of temperature variations, and proposed an energy minimization method by dynamically selecting the supply

voltage. One major problem of this approach is that the accuracy significantly depends on the variation rate of power and temperature. To achieve high accuracy, the length of the interval needs to be kept very small and thus the computation cost can be very high. Huang *et al.* [57] proposed a different approach to calculate the energy consumption. Based on leakage/temperature dependency model proposed in [100], they developed an analytical closed-form energy estimation method for a schedule. However, their work can only be applied for single core platforms, since when extending to multi-core platforms, the heat transfer among different cores makes the existed energy calculation formula unsolvable. Sharifi *et al.* [105] proposed a thermal aware power estimation method for heterogeneous MPSoCs with an objective to optimize p-state per core (not energy estimation), but energy calculation is not a straight forward integration of the power over time due to the temperature variation and the unknown solution of certain matrix operation, i.e. how to solve exponential matrix integration. We are not aware of any other technique published to analytically calculate the multi-core energy consumption with temperature/leakage dependency taken into consideration.

6.2 Preliminary

6.2.1 System Models

The real-time system considered in this work consists of M cores, denoted as $\mathbb{P} = \{\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_M\}$. Each core has N *running modes*, each of which is characterized by a pair of parameters (v_k, f_k) , where v_k and f_k are the supply voltage and working frequency under mode k , respectively. Assume that a *speed schedule* \mathbb{S} (see Chapter 5) is given.

Recall that, in Chapter 5, we have already defined our power model and thermal model. To ease our presentation, we rewrite the formulas of power and thermal models below

$$\mathbf{P}(t) = \mathbf{\Theta} + \mathbf{\Phi}\mathbf{T}(t) \quad (6.1)$$

$$\mathbf{C}\frac{d\mathbf{T}(t)}{dt} + \mathbf{g}\mathbf{T}(t) = \mathbf{P}(t) + \mathbf{\delta} \quad (6.2)$$

where $\mathbf{\Theta}$ and $\mathbf{\Phi}$ are power related coefficient matrices, and \mathbf{C} , \mathbf{g} and $\mathbf{\delta}$ are thermal related coefficient matrices.

6.2.2 Temperature Calculation

Recall that, in Chapter 5 we have proposed an analytical solution to calculate the temperature dynamics at any time instant for a given speed schedule. To keep the integrity of this chapter itself, we briefly present the solution of temperature calculation below.

By applying the power model (see equation (6.1)) into the thermal model (see equation (6.2)), we can directly obtain that

$$\mathbf{C}\frac{d\mathbf{T}(t)}{dt} + \mathbf{g}\mathbf{T}(t) = \mathbf{\Theta} + \mathbf{\Phi}\mathbf{T}(t) + \mathbf{\delta} \quad (6.3)$$

Let $\mathbf{G} = \mathbf{g} - \mathbf{\Phi}$, then the above equation can be rewritten as

$$\mathbf{C}\frac{d\mathbf{T}(t)}{dt} + \mathbf{G}\mathbf{T}(t) = \mathbf{\Theta} + \mathbf{\delta} \quad (6.4)$$

Since \mathbf{C} is the capacitance matrix with none zero values only on the diagonal, we know \mathbf{C} is nonsingular. Thus, the inverse of \mathbf{C} , i.e. \mathbf{C}^{-1} exists. Then equation (6.4) can be further represented as

$$\frac{d\mathbf{T}(t)}{dt} = \mathbf{A}\mathbf{T}(t) + \mathbf{B} \quad (6.5)$$

where $\mathbf{A} = -\mathbf{C}^{-1}\mathbf{G}$ and $\mathbf{B} = \mathbf{C}^{-1}(\boldsymbol{\Theta} + \boldsymbol{\delta})$. The system thermal model shown in equation (6.5) has a form of first order *Ordinary Differential Equations* (ODE), which has the following solution under constant coefficients:

$$\mathbf{T}(t) = e^{t\mathbf{A}}\mathbf{T}_0 + \mathbf{A}^{-1}(e^{t\mathbf{A}} - \mathbf{I})\mathbf{B} \quad (6.6)$$

where \mathbf{T}_0 is the initial temperature.

Specifically, for a state interval $[t_{q-1}, t_q]$, and let κ_q be the corresponding interval mode, once the temperatures at the starting point, i.e. $\mathbf{T}(t_{q-1})$, are given, according to equation (6.6), the ending temperatures of that interval, i.e. $\mathbf{T}(t_q)$, can be directly formulated as

$$\mathbf{T}(t_q) = e^{\Delta t_q \mathbf{A}_{\kappa_q}} \mathbf{T}(t_{q-1}) + \mathbf{A}_{\kappa_q}^{-1}(e^{\Delta t_q \mathbf{A}_{\kappa_q}} - \mathbf{I})\mathbf{B}_{\kappa_q} \quad (6.7)$$

where $\mathbf{A}_{\kappa_q} = -\mathbf{C}^{-1}\mathbf{G}_{\kappa_q}$, $\mathbf{B}_{\kappa_q} = \mathbf{C}^{-1}(\boldsymbol{\Theta}_{\kappa_q} + \boldsymbol{\delta})$, and $\Delta t_q = t_q - t_{q-1}$. Note that since \mathbf{A}_{κ_q} and \mathbf{B}_{κ_q} are only dependent on the core running modes, i.e. κ_q , within a state interval $[t_{q-1}, t_q]$, both \mathbf{A}_{κ_q} and \mathbf{B}_{κ_q} are constant.

Consequently, given a speed schedule \mathbb{S} and the corresponding initial temperature $\mathbf{T}(0)$, with the method introduced above, we can obtain the temperature traces of \mathbb{S} by successively calculating the temperature from one state interval to another.

6.3 Energy Calculation For Multi-Core Scheduling With Thermal Awareness

With the temperature formulation introduced as above, we are now ready to discuss our method to formulate the energy consumption on multi-core systems considering the interdependence of leakage power and temperature. In what follows, we first present an analytical solution to calculate the energy consumption for one state

interval. Then we formulate the total energy consumption for the entire speed schedule.

Consider a state interval, i.e. $[t_{q-1}, t_q]$ with initial temperature of $\mathbf{T}(t_{q-1})$. The energy consumption of all cores within that interval can be simply formulated as

$$\mathbf{E}(t_{q-1}, t_q) = \int_{t_{q-1}}^{t_q} \mathbf{P}(t) dt \quad (6.8)$$

Based on our system power model, given by equation (6.1), we have

$$\mathbf{E}(t_{q-1}, t_q) = \Delta t_q \mathbf{\Theta} + \mathbf{\Phi} \int_{t_{q-1}}^{t_q} \mathbf{T}(t) dt \quad (6.9)$$

For a given state interval and multi-core platform, note that, $\theta_i = \alpha_{k_i} \cdot v_{k_i} + \gamma_{k_i} \cdot v_i^3$ and $\phi_i = \beta_{k_i} \cdot v_{k_i}$, thus $\mathbf{\Theta}$ is a constant. Therefore, to calculate $\mathbf{E}(t_{q-1}, t_q)$, we only need to get $\int_{t_{q-1}}^{t_q} \mathbf{T}(t) dt$.

Recall that the analytical solution for $\mathbf{T}(t)$ is given by equation (6.6). One intuitive approach is therefore to find $\int_{t_{q-1}}^{t_q} \mathbf{T}(t) dt$ as follows:

$$\begin{aligned} & \int_{t_{q-1}}^{t_q} \mathbf{T}(t) dt \\ &= \int_{t_{q-1}}^{t_q} (e^{t\mathbf{A}} \mathbf{T}(t_{q-1}) + \mathbf{A}^{-1}(e^{t\mathbf{A}} - \mathbf{I})\mathbf{B}) dt \end{aligned} \quad (6.10)$$

$$= \int_{t_{q-1}}^{t_q} e^{t\mathbf{A}} dt \mathbf{T}(t_{q-1}) + \mathbf{A}^{-1} \left(\int_{t_{q-1}}^{t_q} e^{t\mathbf{A}} dt - t\mathbf{I} \right) \mathbf{B} \quad (6.11)$$

The problem of this approach is that we need to find $\int_{t_{q-1}}^{t_q} e^{t\mathbf{A}} dt$, but unfortunately, we are not aware of any existing method or mathematical tools that can be used to solve the problem of exponential matrix integration. Therefore, to replace $\mathbf{T}(t)$ in equation (6.9) with equation (6.6) does not seem to be a promising approach.

Note that, as long as we can get $\int_{t_{q-1}}^{t_q} \mathbf{T}(t) dt$, we find the solution to the overall energy consumption for state interval $[t_{q-1}, t_q]$. If we let $\mathbf{X} = \int_{t_{q-1}}^{t_q} \mathbf{T}(t) dt$, then the above can be simplified as

$$\mathbf{E}(t_{q-1}, t_q) = \Delta t_q \mathbf{\Theta} + \mathbf{\Phi} \mathbf{X} \quad (6.12)$$

In what follows, we introduce a novel method to calculate \mathbf{X} . Recall that the system thermal model can be formulated as (see equation (6.4)):

$$\mathbf{C} \frac{d\mathbf{T}(t)}{dt} + \mathbf{G}\mathbf{T}(t) = \mathbf{\Theta} + \boldsymbol{\delta}$$

Since $\mathbf{C}, \mathbf{G}, \mathbf{\Theta}$ and $\boldsymbol{\delta}$ are all constants within interval $[t_{q-1}, t_q]$, if we integrate on both sides of the above equation with respect to time t , where $t \in [t_{q-1}, t_q]$, we have

$$\mathbf{C}\Delta\mathbf{T}_q + \mathbf{G} \int_{t_{q-1}}^{t_q} \mathbf{T}(t)dt = \Delta t_q(\mathbf{\Theta} + \boldsymbol{\delta}) \quad (6.13)$$

where $\Delta\mathbf{T}_q = \mathbf{T}(t_q) - \mathbf{T}(t_{q-1})$ and $\Delta t_q = t_q - t_{q-1}$. If we further replace $\int_{t_{q-1}}^{t_q} \mathbf{T}(t)dt$ with \mathbf{X} , we have

$$\mathbf{C}\Delta\mathbf{T}_q + \mathbf{G}\mathbf{X} = \Delta t_q(\mathbf{\Theta} + \boldsymbol{\delta}) \quad (6.14)$$

Now let \mathbf{H} be that

$$\mathbf{H} = \Delta t_q(\mathbf{\Theta} + \boldsymbol{\delta}) - \mathbf{C}\Delta\mathbf{T}_q \quad (6.15)$$

Note that, based on equation (6.7), $\Delta\mathbf{T}_q$ can be easily calculated as

$$\Delta\mathbf{T}_q = \mathbf{T}(t_q) - \mathbf{T}(t_{q-1}). \quad (6.16)$$

Therefore, \mathbf{H} can be easily obtained once the state interval $[t_{q-1}, t_q]$ is defined. Accordingly, from equation (6.14), we can get

$$\mathbf{G}\mathbf{X} = \mathbf{H} \quad (6.17)$$

Assuming \mathbf{G} is nonsingular, \mathbf{X} can thus be solved as

$$\mathbf{X} = \mathbf{G}^{-1}\mathbf{H} \quad (6.18)$$

By applying equation (6.18) into (6.12), we can get that

$$\mathbf{E}(t_{q-1}, t_q) = \Delta t_q \mathbf{\Theta} + \Phi \mathbf{G}^{-1} \mathbf{H} \quad (6.19)$$

As such, given a multi-core platform and a state interval, the energy consumption within the interval can be calculated using equation 6.19 analytically. We formally present our energy calculation method for a state interval in Theorem 6.3.1.

Theorem 6.3.1. *Given a state interval $[t_{q-1}, t_q] \in \mathbb{S}$ with \mathbf{T}_{q-1} the temperature at time t_{q-1} , the overall system energy consumption within interval $[t_{q-1}, t_q]$ can be formulated as*

$$\mathbf{E}(t_{q-1}, t_q) = \Delta t_q \mathbf{\Theta}_{\kappa_q} + \mathbf{\Phi}_{\kappa_q} \mathbf{G}_{\kappa_q}^{-1} \mathbf{H}_{\kappa_q} \quad (6.20)$$

Note that given a speed schedule and initial temperature, the temperature at the ends of each state interval can be readily determined using equation (6.7). For a speed schedule \mathbb{S} consisting of Q state intervals, the total system energy consumption under \mathbb{S} can be obtained by summing up the energy consumptions of all state intervals. We conclude this energy calculation method in Theorem 6.3.2.

Theorem 6.3.2. *Given an initial temperature T_0 and a speed schedule \mathbb{S} consisting of Q state intervals, the total system energy consumption under \mathbb{S} , denoted as $E_{total}(\mathbb{S})$, can be calculated as*

$$E_{total}(\mathbb{S}) = \sum_{q=1}^Q \sum_{i=1}^M E_i(t_{q-1}, t_q) \quad (6.21)$$

where $E_i(t_{q-1}, t_q)$ can be calculated from equation (6.20).

The computational complexity for our energy calculation of each state interval mainly comes from the matrix multiplications and inversions, with a complexity of $O(M^3)$. To calculate the overall energy consumption for a schedule with Q state intervals, the complexity is thus $O(Q \times M^3)$. In what follows, we use experiments to evaluate the performance of our proposed method.

6.4 Experiments And Results

In this section, we validated the proposed energy calculation method with simulations. We compared our proposed method with the traditional *numerical method* to obtain some insights with regard to the effectiveness and efficiency of an energy estimation approach. In what follows, we first introduce the settings for our experiments. We then present and discuss the experimental results.

6.4.1 Experimental Setup

Table 6.1: HotSpot parameters and floorplan

<i>Parameter</i>	<i>Value</i>
Total Cores	9 (3x3)
Area per Core	4 mm^2
Die Thickness	0.15 mm
Heat Spreader Side	20 mm
Heat Sink Side	30 mm
Convection Resistance	0.1 K/W
Convection Capacitance	140 J/K
Ambient Temperature	30°C

Table 6.2: Power/thermal parameters

$V_{dd}(V)$	α	β	γ
0.0	0.0	0.0	0.0
0.8	1.4533	0.0760	6.0531
0.9	2.4173	0.0844	5.8008
1.0	4.0533	0.0936	5.8906

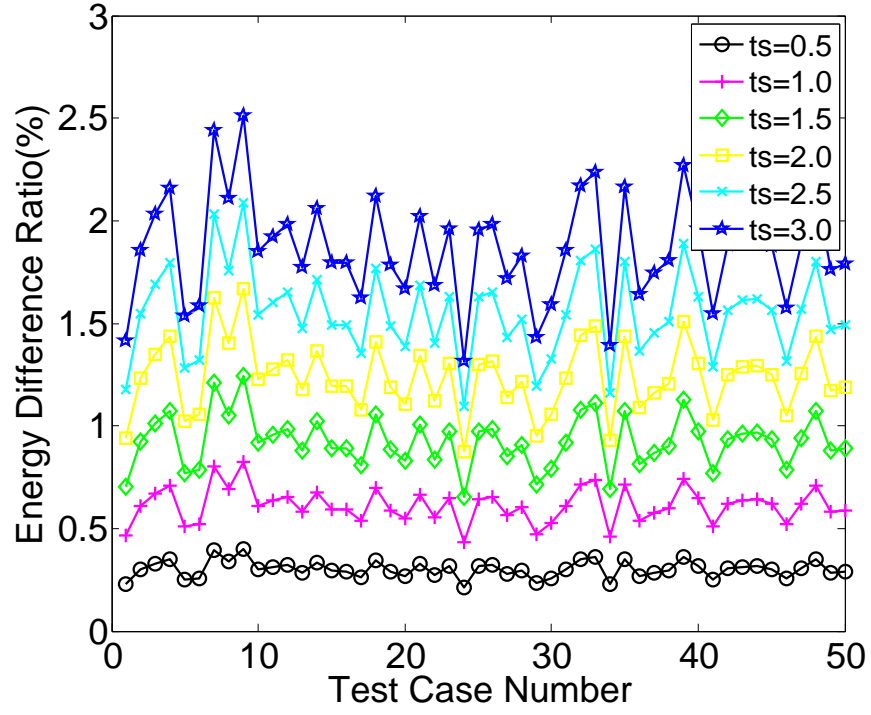
We performed our experimental simulations based on a 3×3 multi-core system. The granularity of the floorplan was restricted to core-level. Our core model was based on 65nm technology as presented in [80]. We assumed that each core supports 3 active modes with supply voltage ranging from 0.8V to 1.0V and a step size of 0.1V. We also set one inactive/sleep mode with supply voltage equal to 0V.

We adopted the same thermal parameters as used in work [97] (see Table 6.4.1). We set the power consumption under the peak temperature constraint of $110^{\circ}C$. The thermal parameters, including thermal conductance, capacitance etc. were taken from HotSpot-4.02 [1]. The thermal nodes in our thermal model included active layer, interface layer, heat spreader and heat sink. The relevant useful parameters were shown in Table 6.4.1. We set the ambient temperature T_{amb} as well as the initial temperature T_0 as $30^{\circ}C$.

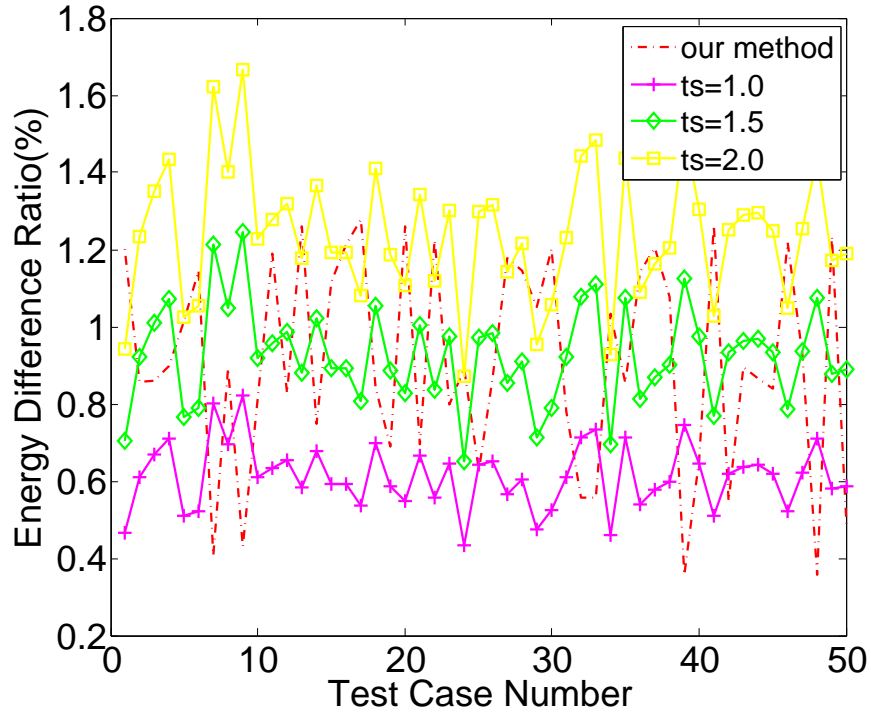
We randomly generated 50 multi-core speed schedules as our test cases. The running mode for each scheduling interval was randomly chosen from $[0, 0.8, 0.9, 1.0]V$ (see Table 6.4.1). The total length of the scheduling interval was evenly distributed within $[100, 200]$, and the length of each scheduling interval was evenly distributed within $[30, 50]$. For each test case, our proposed method as well as the traditional numerical method with sampling interval length varied from 0.5 second to 3.0 second were used to calculate the energy consumption. The baseline was obtained by setting the length of sampling interval to 0.01. When applying the numerical method, we calculated the leakage power consumption based on the accurate circuit level leakage temperature model [80], i.e.

$$I_{leak} = I_s \cdot (\mathcal{A} \cdot T^2 \cdot e^{((a \cdot V_{dd} + b)/T)} + \mathcal{B} \cdot e^{(c \cdot V_{dd} + d)}) \quad (6.22)$$

where I_s is the leakage current at certain reference temperature and supply voltage, T is the core temperature, and $\mathcal{A}, \mathcal{B}, a, b, c, d$ are physically determined constants (i.e. fitting parameters). All simulations were conducted on a *Dell Precision T1500 Desktop Workstation* with CPU type of *Intel i5 750 Quad Core* and *4GB* memory capacity.



(a) Numerical method



(b) Our proposed method

Figure 6.1: Accuracy analysis, compared with the numerical method under $t_s = 0.01$

6.4.2 Accuracy Analysis

In this subsection, we investigate the performance of our proposed method in terms of accuracy. To compare the accuracy of different energy estimation approaches, we need to identify the accurate energy consumption for a given speed schedule. We resorted to the numerical method with a very short sampling interval to achieve this goal. The question is how short the sampling interval should be.

In our experiments, we set the length of sampling interval t_s from 0.5 second to 3.0 second with a step width of 0.5 second and calculated the energy consumption for different schedules. Particularly, we set $t_s = 0.01$ second as the baseline since we found that the largest relative energy difference between $t_s = 0.01$ second and $t_s = 0.5$ second was smaller than 0.4%. We then normalized the energy consumption by other approaches to the baseline results. Figure 6.1(a) shows the relative differences of energy consumption estimation results using numerical approach with different sampling intervals, i.e. from $t_s = 0.5$ second to $t_s = 3.0$ second. The relative differences of energy consumption based on our proposed approach and comparable numerical results are presented in Figure 6.1(b).

From Figure 6.1(a), it is not surprising to see that the smaller the sampling interval, the smaller the energy difference ratio becomes. For example, when t_s is decreased from 3.0 to 0.5, the average energy difference ratio is reduced from 1.7% to 0.4%. This is because that the smaller the sampling interval is, the less the temperature can change. Since the numerical method estimates the leakage consumption within an interval assuming temperature within a sampling interval does not change, the error of the estimated leakage energy can be kept small if the sampling interval is small enough.

On the other hand, we can see from Figure 6.1(b) that our proposed method performed well from the aspect of accuracy. For example, the largest relative error

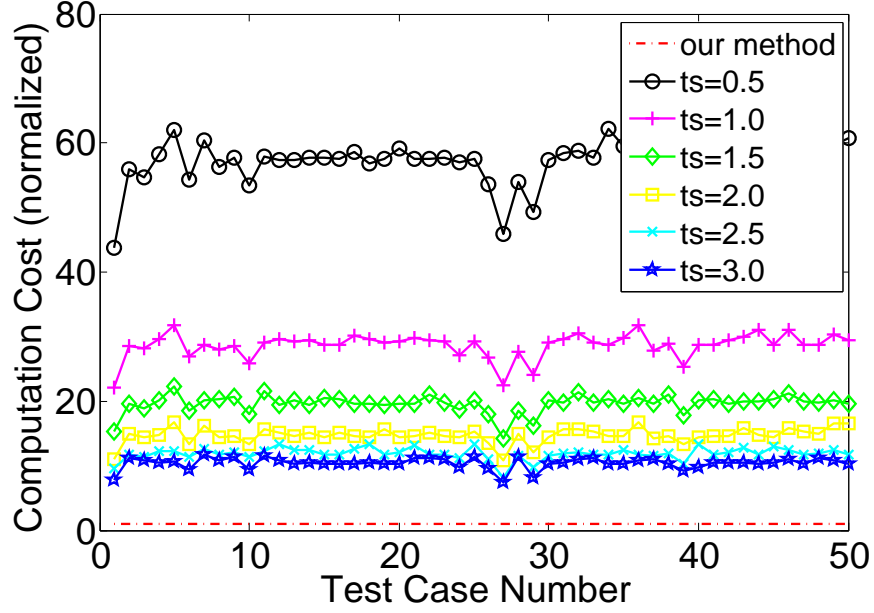


Figure 6.2: Time efficiency analysis, normalized with our method

observed in Figure 6.1(b) is no more than 1.5%. As shown in Figure 6.1(b), we can see that our method outperforms the numerical method with $t_s = 2.0$ second for most test cases, and compatible with the method with $t_s = 1.5$ second. The experimental results clearly show that our proposed approach can achieve very good accuracy in estimating the overall energy consumption for a given speed schedule.

6.4.3 Time Efficiency Analysis

We next want to evaluate the computational efficiency of our proposed method. We collected the CPU times for different approaches for all test cases. We then use the CPU times of our method as the baseline results. The normalized results are shown in Figure 6.2.

From Figure 6.2, we can see that the numerical method with a small sampling interval can have a substantially large computational overhead than our approach. For example, as shown in Figure 6.2, our method is more than 50 times (on average)

faster than the numerical approach with $t_s = 0.5$, and 10 times (on average) faster than that with $t_s = 3.0$. Compared with the numerical method with $t_s = 1.5$, which is compatible with our method from the perspective of accuracy, our method can achieve an average speedup of 15 times. From Figure 6.2, we can conclude that the proposed method is much more time efficient than the numerical approach.

6.5 Summary

Energy consumption optimization is a critical design issue in design of multi-core computing systems. It becomes more challenging in deep submicron domain when leakage consumption becomes more and more significant and the interdependency of leakage and temperature becomes substantial. A key to solve this problem is to calculate the energy consumption efficiently and effectively.

In this chapter, we present a fast and accurate solution for energy calculation on multi-core systems that takes the interdependency of leakage, temperature and supply voltage into consideration. Different from the traditional numerical approach, we develop an analytical formulation for the energy consumption, and based on which, to calculate the overall energy consumption rapidly and accurately. Our system models are rather general and can be easily extended for different platforms and applications. Our experiments show that the proposed method can achieve a speedup of 15 times compared with the numerical method, with a relative error no more than 1.5%.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

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

7.1 Summary

Today, real-time embedded applications and systems have been pervasive in our daily life. Designers from both industry and academia have made great efforts on increasing the computing performance of such systems. For a more sustainable improvement of computing system performance, designers rely more on multi-core platform rather than building more complicated single core architecture and raising its working frequency. Multi-core is becoming mainstream.

In this dissertation, we presented our research work on real-time multi-core scheduling at the operating system level. Specifically, we presented several novel strategies to schedule real-time tasks effectively and efficiently on multi-core platforms, and optimize design criteria such as system utilization, peak temperature and energy consumption.

First, we started our research work on partitioned scheduling problem, in which each task is allocated to a dedicated processor and run only on that specific processor. presented two new partitioned approaches (i.e. *PSER* and *HAPS*) for scheduling real-time sporadic tasks on multi-core platform under *RMS*. The *PSER* algorithm first transformed a given task set with respect to each task's period, and then assigned tasks based on their scaled periods under the traditional RBound. The *HAPS* algorithm took the harmonic advantage by transforming the entire task set into a harmonic set, and based on made the partitioning decision according to a efficient utilization bound, i.e. CBound. We formally proved that our scheduling

algorithms could guarantee the feasibility of any task set successfully passed the partitioned procedures. The experimental results demonstrated that our proposed algorithms could significantly improve the scheduling performance compared with previous work. For example, when the system average utilization is around 0.8, our proposed algorithms could improve the scheduling performance by 1.25 times compared with existing work.

Then we moved our research concentration from the partitioned scheduling to the semi-partitioned scheduling, in which some tasks can be split and executed on different processing cores. We developed two semi-partitioned scheduling algorithms, i.e. HSP-light and HSP, to schedule light and general task sets, respectively. Our approaches exploited the well known fact that tasks with closer harmonic relationship have better schedulability than non-harmonic ones on a single processing core. We formally prove that our proposed algorithms could successfully schedule any task set with a utilization bounded by *Liu&Layland's bound*. The experimental results clearly showed that, by exploiting the harmonic relationship among tasks more aggressively, our algorithms could significantly improve the schedulability of semi-partitioned scheduling compared with the existing algorithms, e.g. our HSP algorithm could achieve a success ratio up to four times over the existing work when the average system utilization was around 0.9.

Next, we took the temperature constraint into consideration in our research problem of multi-core real-time scheduling. Specifically, we studied the feasibility testing problem in the design of multi-core systems. We presented a number of novel analytical solutions for temperature calculation, i.e. the temperatures within an arbitrary LCM and the steady-state. Then we proposed a complete solution to effectively and efficiently detect the peak temperature for a periodic speed schedule on a multi-core platform. We further proposed three temperature-constrained feasi-

bility checking methods, i.e. TmaxCheck, ModeCheck and TssCheck. Our proposed techniques formed the basis of more advanced thermal aware real-time embedded system design on multi-core platforms.

Finally, we diverted our focus to energy estimation problem on multi-core platforms. We studied the multi-core energy calculation problem, which is one of the most fundamental, as well as critical, problems in design of multi-core computing systems. We presented a fast and accurate solution for energy calculation on multi-core systems that took the interdependency of leakage, temperature and supply voltage into consideration. Different from the traditional numerical approach, we developed an analytical formulation for the energy consumption, and based on which, to calculate the overall energy consumption rapidly and accurately. We adopted a rather general system model and thus the proposed technique could be easily extended to different platforms and applications. Our experiments showed that the proposed method could achieve a speedup of 15 times compared with the numerical method, with a relative error no more than 1.5%.

7.2 Future Work

As transistor size has become smaller and smaller, the reliability of IC chips is increasingly becoming a serious concern. First, the rapidly decreased feature size of the transistors has dramatically increased the rate of radiation-induced faults, up to several orders of magnitude [75]. Second, the ever-increasing on-chip power consumption and temperature have imposed serious threats for the lifetime reliability of IC chips[59]. Due to the nature of safety-critical real-time systems, e.g. automobiles and industrial controls, catastrophic consequences may occur if system faults can not be handled timely or properly.

Processor faults can be largely classified as *transient* or *permanent* [111]. The transient fault refers to the temporary malfunction of a processor, usually caused by electromagnetic interferences or cosmic ray radiations, that can lead to temporary errors in computation and corruptions in data [110]. The permanent fault is caused by hardware failures [23], and once this kind of fault occurs, it disables a processor permanently. In our research, we only consider transient faults because they occur more frequently than permanent faults (with a ratio of 100:1 or higher) [96].

We intend to extend our research to address the problem of how to develop real-time scheduling algorithms under reliability constraint. Specifically, we seek to develop a approach to minimize the energy consumption for scheduling real-time tasks, meanwhile to guarantee pre-defined reliability requirement. In what follows, we present some preliminary results of our research on reliability aware real-time scheduling design.

7.2.1 System Models And Underlying Scheduling Problem

The real-time system applications considered in our work are described by DAG-based intra-task model. A task is represented by a direct acyclic graph, denoted as $G = (V, L)$. V is the node (vertex) set of G , represented as $V = \{v_1, v_2, \dots, v_N\}$, and contains the execution requirement for each node. A node v_i is represented by a tuple (c_i, d_i) , where c_i is the workload on v_i and d_i is the corresponding deadline. L is the edge (link) set of G , and represents the precedence constraints. Each edge is denoted as a tuple $\langle v_i, v_j \rangle$ which indicates that the directed edge emerges from source node v_i and incidents on the destination node v_j . Nodes which have no predecessors are called *entry nodes*, and those which have no successors are called *exit nodes*. Moreover, let $P(G)$ represent the path set of G , denoted as $P(G) = (P_1, \dots, P_K)$. A

path is a directed series of nodes that starts from an entry node and terminate at an exit node. Further, for the k_{th} path of $P(G)$, it is represented by $P_k = \{v_{k,1}, \dots, v_{k,N_k}\}$. We assume that the *communication cost* on each edge is zero.

We adopt the transient fault model with a Poisson distribution in our research [128].

$$\lambda(f) = \lambda_0 \cdot 10^{\frac{d \cdot (1-f)}{1-f_{min}}} \quad (7.1)$$

where the exponent $d(> 0)$ is a constant, indicating the sensitivity of fault rates to voltage scaling. λ_0 is the average fault rate corresponding to the maximum frequency f_{max} . That is, reducing the frequency for energy savings will result in exponentially increased fault rates. The maximum average fault rate is assumed to be $\lambda_{max} = \lambda_0 \cdot 10^d$, which corresponds to the lowest frequency f_{min} .

First, the reliability of a node v_i under frequency f_i , denoted as $R_i(f_i)$, can be computed by

$$R_i(f_i) = e^{-\lambda(f_i) \cdot \frac{c_i}{f_i}} \quad (7.2)$$

where $\lambda(f_i)$ is given by Equation (7.1). $R_i(f_i)$ represents the probability of completing the workload c_i on node v_i successfully under frequency f_i with no fault occurrence.

Next, we consider the reliability on one path. Given a path P_k consisting of N_k tasks, i.e. $P_k = \{v_1, v_2, \dots, v_{N_k}\}$, and the corresponding frequency assignment $\{f_1, f_2, \dots, f_{N_k}\}$, then the *path reliability* of P_k , denoted as R_{P_k} , is defined as the probability of completing all nodes on P_k successfully under that frequency assignment.

$$\begin{aligned} R_{P_k} &= R_1(f_1) \cdot R(f_2, \dots, f_{N_k}) + \\ &\quad (1 - R_1(f_1)) \cdot R_1(f_{max}) \cdot R(f_2, \dots, f_{N_k}) \end{aligned} \quad (7.3)$$

The first term in equation (7.3) represents the path reliability of P_k while node v_1 completes successfully. Similarly, the second term represents the path reliability of P_k while node v_1 fails and subsequently is recovered with one recovery block.

Then, we consider the reliability among all paths. Given a task G , there may be more than one path in $P(G)$, in order to describe system reliability under certain frequency assignment, we quantify the *system reliability* by choosing the minimum path reliability under that frequency assignment among all paths. Thus, the system reliability of G , denoted as $R(G)$, under intra-task DVFS management can be calculated by

$$R(G) = \min_{\forall P_k \in P(G)} R_{P_k} \quad (7.4)$$

Further, to clearly and explicitly describe the original task reliability requirement, we define the *reliability threshold* of G in below.

Given a task G , the *reliability threshold* of G , denoted as R_{thr} , is defined as the minimum path reliability among all paths under the system maximum frequency, which can be calculated by

$$R_{thr} = \min_{\forall P_k \in P(G)} \left(\prod_{v_i \in P_k} R_i(f_{max}) \right) \quad (7.5)$$

R_{thr} represents the minimum probability for completing any path in G successfully at the maximum frequency f_{max} with no fault occurrence. Note that R_{thr} is usually referred as the reliability requirement that need be guaranteed in design of intra-task scheduling.

Now we are ready to formulate our research problem for reliability aware energy efficient scheduling.

Problem 7.2.1. *Given a task Γ , our research problem can be formulated as follows:*

$$\text{Minimize: } E(\Gamma) = \sum_{P_k \in P(\Gamma)} w_k \cdot E_{P_k} \quad (7.6)$$

$$s.t. \forall P_k \in P(G), \sum_{i=1}^{N_k} \frac{c_i}{f_i} \leq d_{N_k} \quad (7.7)$$

$$\forall P_k \in P(G), R_{P_k} \geq R_{thr} \quad (7.8)$$

The objective is to minimize the energy consumption while maintaining the system feasibility and desired reliability. The feasibility of the the real-time task is guaranteed by satisfying equation (7.7), where c_i is the execution time of node v_i under maximum frequency f_{max} and f_i is the actual frequency assigned to it. Similarly, the system reliability threshold is guaranteed by making sure reliability on each path is no less than R_{thr} as in equation (7.8). This optimization problem is well known to be NP-hard and an optimal solution is computationally intractable and impractical. In the following section, we introduce our heuristic to deal with this problem.

7.2.2 Preliminary Results

In what follows, we first discuss the available slacks for each node in a DAG-based task. Then based on the available slacks, we present a way to calculate the potential energy saving of each node with respect to the available slacks.

Given a task G , let X be a backup indicator vector for all nodes in G , i.e. $X = \{x_1, x_2, \dots, x_N\}$. For $\forall x_i \in X$, we have that:

$$\begin{cases} x_i = 1 & \text{if } v_i \text{ has backup block,} \\ x_i = 0 & \text{otherwise.} \end{cases} \quad (7.9)$$

We first calculate the *earliest starting time* of each node. We assume that the earliest starting time of v_1 is 0, then the earliest starting time of v_i , for $i = 2, 3, \dots, N$

can be calculated as the latest completion time among all its predecessors. Note that, if the frequency of a node v_j is scaled down, a recovery block of size c_j has to be reserved in order to maintain its original reliability under f_{max} . The earliest starting time t_i^E is computed in equation (7.10).

$$t_i^E = \begin{cases} 0, & i = 1 \\ \max\{t_j^E + \frac{c_j}{f_j} + x_j \cdot c_j \mid v_j, v_i \in L\}, & i = 2, \dots, N \end{cases} \quad (7.10)$$

Then we calculate the *latest starting time* of each node, which is the latest time when a node has to start execution without compromising the feasibility of a schedule. If v_i is a terminal node, it needs to begin execution c_i/f_i (its actual execution requirement) amount of time before its deadline, otherwise, it should start c_i/f_i ahead the latest starting time of its closet successor. Same as above, a node v_i needs to start additional c_i time units ahead to leave space for its recovery in case fault occurs to maintain its reliability. The calculation of latest starting time for v_i is given in equation (7.11).

$$t_i^L = \begin{cases} d_i - \frac{c_i}{f_i} - x_i \cdot c_i, & \text{if } v_i \text{ is terminal node.} \\ \min\{t_j^L - \frac{c_i}{f_i} - x_i \cdot c_i \mid v_i, v_j \in L\}, & \text{otherwise.} \end{cases} \quad (7.11)$$

After obtaining the earliest and latest starting time respectively, we can now calculate the maximum amount of slacks which potentially could be used for DVFS to reduce energy consumption. For any node v_i , its maximum available slack, denoted as s_i , is calculated by

$$s_i = \begin{cases} 0, & \text{if } t_i^L - t_i^E \leq c_i. \\ t_i^L - t_i^E, & \text{otherwise.} \end{cases} \quad (7.12)$$

Next, based on the above result, we discuss the corresponding frequency assignment and the potential energy saving. First, the frequency assignment under

maximum available slacks can be represented by equation (7.13).

$$f'_i = \begin{cases} f_{max}, & \text{if } s_i = 0. \\ \max\{\frac{c_i}{s_i}, f_{opt}\}, & \text{otherwise.} \end{cases} \quad (7.13)$$

Secondly, the potential energy saving for each task after DVFS can be presented by equation (7.14).

$$\Delta E_i = E_i(f_i) - E_i(f'_i) \quad (7.14)$$

ΔE_i represents the energy saving of v_i by efficiently utilizing its available slack s_i . In the following, we apply this ΔE_i to make the frequency assignment decision in our proposed algorithm.

Scheduling algorithm Now we introduce our proposed *Reliability-Aware Intra-Task Scheduling* (RA-ITS) algorithm, which statically determines the frequencies for all nodes in a DAG task such that system energy consumption can be minimized and meanwhile the timing and reliability constraints can be guaranteed.

Algorithm 7 *Reliability Aware Intra-Task Scheduling* (RA-ITS) algorithm

Require:

- 1) Task : $G = \langle V, L \rangle, V = \{v_1, v_2, \dots, v_N\}$;
 - 1: $f_i = f_{max}$, for $i = 1, 2, \dots, N$;
 - 2: **for** $i = 1$ to N **do**
 - 3: for $\forall i \in [1, N]$, t_i^E = earliest start time of v_i under $\{f_1, \dots, f_N\}$ (see equation (7.10));
 - 4: for $\forall i \in [1, N]$, t_i^L = latest start time of v_i under $\{f_1, \dots, f_N\}$ (see equation (7.11));
 - 5: for $\forall i \in [1, N]$, s_i = maximum available slack of v_i (see equation (7.12));
 - 6: **if** $\max\{s_i | i = 1, \dots, N\} = 0$, **then** Break, **end if**
 - 7: for $\forall i \in [1, N]$, f'_i = frequency of v_i by using slack s_i (see equation (7.13));
 - 8: for $\forall i \in [1, N]$, $\Delta E_i = E_i(f_i) - E_i(f'_i)$;
 - 9: **if** $\max\{\Delta E_i | i = 1, \dots, N\} \leq 0$, **then** Break, **end if**
 - 10: find v_{i^*} , such that $\Delta E_{i^*} = \max\{\Delta E_i | i = 1, \dots, N\}$;
 - 11: assign f'_{i^*} to f_{i^*} ;
 - 12: **end for**
 - 13: **return** $\{f_1, f_2, \dots, f_N\}$
-

Experimental setup and results In our experiment, transient faults are assumed to follow Poisson distribution with an average fault rate of $\lambda_0 = 10^{-6}$ at f_{max} , which is a realistic fault rate as reported in [134]. Moreover, the fault rate exponent d is set to 2. We use a cubic frequency-dependent power component P_d which is equal to unity at $f_{max} = 1.0$. The frequency-independent power component P_{ind} for each task is normalized with respect to P_d and is generated according to the uniform distribution in the range of $[0, 0.25]$.

The task graph is generated by TGFF benchmark, which is configured as below: 1) node number for each task graph is set between $[10, 20]$; 2) fanin and fanout of each node are both set to 3; 3) execution requirement of each node is set between $[20, 80]$. Each point in the presented figures is obtained by averaging the results obtained through 100 different task graphs. Moreover, we assume that all paths have the same probability for occurring, that is $w_k = w_h$, for any two different P_k and P_h in $P(G)$. All energy consumption results are normalized with respect to the no power management (NPM) scheme that executes all nodes on any path at the maximum frequency f_{max} .

We compare the energy consumption of our *RA-ITS* approach with other two baseline approaches, i.e the *No Power Management* (NPM) approach (which assigns maximum frequency to all nodes) and the *GREEDY* approach (which assigns frequencies by letting the current executing node using the system available slacks as much as needed under the reliability threshold).

First, we evaluated the impact of available slack on energy savings. In this part, P_{ind} is set to 0.05 as a constant. We use S/C to represent the maximum slack-execution-ratio among all paths in each task graph. We vary this slack to execution time ratio from 0.4 to 1.4, and show the result in Figure 7.1. As illustrated in 7.1, all three methods can save achieve energy savings compared to *NPM*. Clearly,

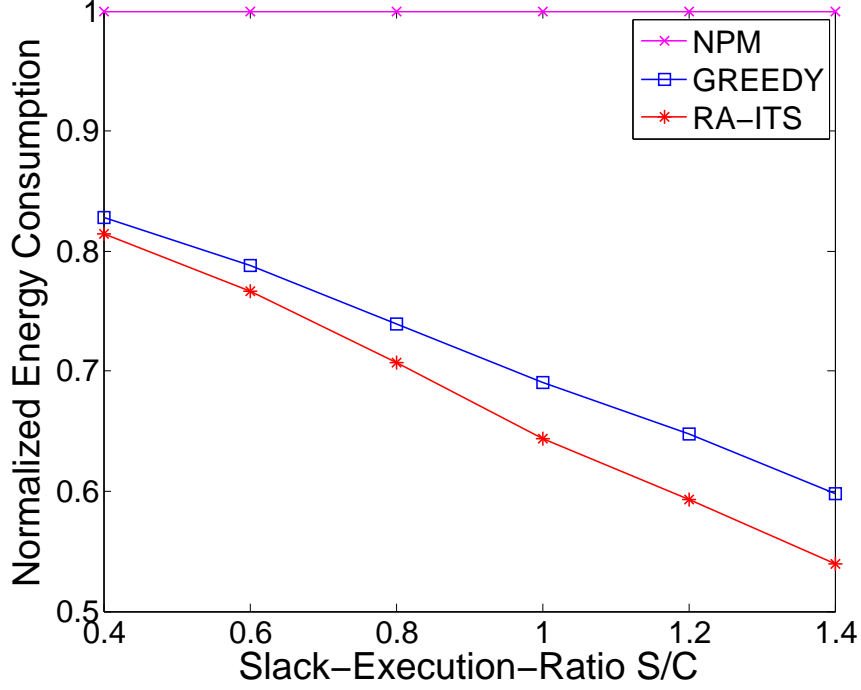


Figure 7.1: Impact of slack on energy consumption

our *RA-ITS* outperforms the other two techniques. For instance, when the slack to execution ratio is 1, our approach attains approximately 34% $((1.0 - 0.66)/1.0)$ more energy saving than *NPM* and 8% $((0.72 - 0.66)/0.72)$ more than *GREEDY*. Moreover, compared with *NPM* and *GREEDY*, we see that the larger the relative slack time, the more energy saving of our *RA-ITS* algorithm.

Then we study the impact of P_{ind} on energy savings. Same as [128, 131], the P_{ind} is varied between $[0.05, 0.35]$ for each node v_i and the maximum slack-execution-ratio is fixed at $S/C = 1.4$. According to 7.2, the larger the P_{ind} , the higher the energy consumption. The reason is that as the P_{ind} increases, the contribution of frequency independent energy consumption becomes more dominant, the energy-efficient frequency is therefore increased according to [132] and results in fewer opportunities for DVFS. Even under this situation, *RA-ITS* still has the better performance in terms of energy consumption.

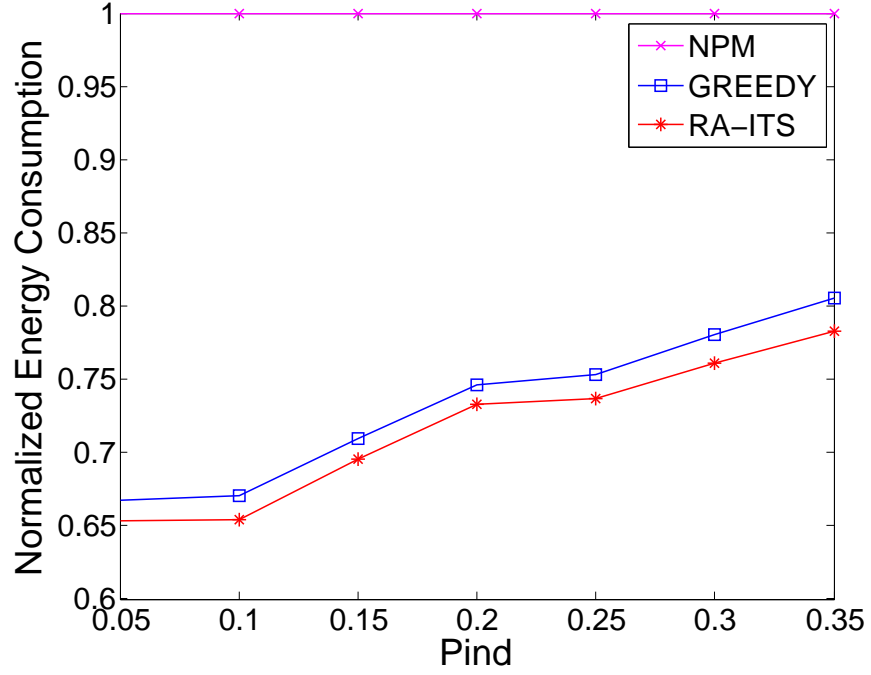


Figure 7.2: Impact of P_{ind} on energy consumption

Overall, the experimental results show clearly the effectiveness of our proposed algorithm. We can see that: 1) with the increment of slack time, more energy savings can be achieved by our *RA-ITS*; 2) under different configurations of P_{ind} , the proposed algorithm can get an efficient energy saving.

How to take leakage/temperature dependency into consideration for reliability-constrained scheduling is an interesting problem and needs further study.

BIBLIOGRAPHY

- [1] Hotspot 4.2 temperature modeling tool. *University of Virginia*, page <http://lava.cs.virginia.edu/HotSpot>, 2009.
- [2] Behind the birth of m3. *IHS iSuppli*, 2012.
- [3] Embedded system market - global industry analysis, size, share, growth, trends and forecast. *Transparency Market Research*, 2013.
- [4] T. F. Abdelzaher, V. Sharma, and C. Lu. A utilization bound for aperiodic tasks and priority driven scheduling. *IEEE Transactions on Computers*, 53(3):334–350, Mar. 2004.
- [5] B. Ackland, A. Anesko, D. Brinthaup, S. Daubert, A. Kalavade, J. Knobloch, E. Micca, M. Moturi, C. Nicol, J. O’Neill, J. Othmer, E. Sackinger, K. Singh, J. Sweet, C. Terman, and J. Williams. A single-chip, 1.6-billion, 16-b mac/s multiprocessor dsp. *Solid-State Circuits, IEEE Journal of*, 35(3):412–424, 2000.
- [6] V. Agarwal, M. S. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus ipc: the end of the road for conventional microarchitectures. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 248–259, Jun. 2000.
- [7] AMD. Amd server processor series. pages <http://www.amd.com/US/PRODUCTS/SERVER/PROCESSORS/6000-SERIES-PLATFORM/6300/Pages/6300-series-processors.aspx>, 2013.
- [8] J. Anderson, V. Bud, and U. Devi. An EDF-Based Scheduling Algorithm for Multiprocessor Soft Real-Time Systems. In *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, Jul. 2005.
- [9] B. Andersson. Global Static-Priority Preemptive Multiprocessor Scheduling with Utilization Bound 38% . In *Proc. ACM International Conference on Principles of Distributed Systems (OPODIS)*, volume 5401, pages 73–88, 2008.
- [10] B. Andersson, S. Baruah, and J. Jonsson. Static-Priority Scheduling on Multiprocessors. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, Dec. 2001.

- [11] B. Andersson, K. Bletsas, and S. Baruah. Scheduling Arbitrary-Deadline Sporadic Task Systems on Multiprocessors. In *IEEE Real-Time Systems Symposium (RTSS)*, Dec. 2008.
- [12] B. Andersson and J. Jonsson. Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition. In *RTCSA '00: Proceedings of the Seventh International Conference on Real-Time Systems and Applications (RTCSA '00)*, page 337, 2000.
- [13] B. Andersson and J. Jonsson. The Utilization Bounds of Partitioned and Pfair Static-Priority Scheduling on Multiprocessors Are 50%. In *Euromicro Conference on Real-Time Systems (ECRTS)*, Jul. 2003.
- [14] B. Andersson and E. Tovar. Competitive analysis of static-priority partitioned scheduling on uniform multiprocessors. In *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pages 111–119, Aug.
- [15] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.
- [16] N. Bansal, T. Kimbrel, and K. Pruhs. Speed scaling to manage energy and temperature. *Journal of the ACM*, 54(1):1–39, 2007.
- [17] M. Bao, A. Andrei, P. Eles, and Z. Peng. Temperature-aware voltage selection for energy optimization. In *Design Automation Conference, 2008. DAC '08. 45th ACM/IEEE*, pages 1083 –1086, 2008.
- [18] M. Bao, A. Andrei, P. Eles, and Z. Peng. On-line thermal aware dynamic voltage scaling for energy optimization with frequency/temperature dependency consideration. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 490 –495, Jul. 2009.
- [19] M. Bao, A. Andrei, P. Eles, and Z. Peng. Temperature-aware idle time distribution for energy optimization with dynamic voltage scaling. In *DATE*, pages 21 – 27, 2010.

- [20] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. Is Semi-Partitioned Scheduling Practical? In *Euromicro Conference on Real-Time Systems (ECRTS)*, Jul. 2011.
- [21] M. Berktdorf and T. Tian. *CPU Monitoring With DTS/PECI*. Intel Corporation, 2010.
- [22] E. Bini, G. C. Buttazzo, and G. M. Buttazzo. Rate Monotonic Analysis: The Hyperbolic Bound. *IEEE Transactions on Computers*, 52(7):933–942, Jul. 2003.
- [23] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10 – 16, Nov.-Dec. 2005.
- [24] S. Borkar. Thousand core chips: a technology perspective. In *DAC*, pages 746–749, 2007.
- [25] W. J. Bouknight, S. Denenberg, D. McIntyre, J. M. Randall, A. Sameh, and D. Slotnick. The illiac iv system. *Proceedings of the IEEE*, 60(4):369–388, 1972.
- [26] A. Burchard, J. Liebeherr, Y. Oh, and S. Son. New strategies for assigning real-time tasks to multiprocessor systems. *Computers, IEEE Transactions on*, 44(12):1429–1442, Dec.
- [27] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.
- [28] T. Chantem, R. P. Dick, and X. S. Hu. Temperature-aware scheduling and assignment for hard real-time applications on mpsoes. In *Proceedings of the conference on Design, automation and test in Europe, DATE '08*, pages 288–293, 2008.
- [29] T. Chantem, X. S. Hu, and R. Dick. Online work maximization under a peak temperature constraint. In *ISLPED*, pages 105–110, 2009.
- [30] V. Chaturvedi, H. Huang, and G. Quan. Leakage aware scheduling on maximal temperature minimization for periodic hard real-time systems. In *ICISS*, pages 1802–1809, 2010.

- [31] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. *Approximation algorithms for bin packing: a survey*, pages 46–93. PWS Publishing Co., Boston, MA, USA, 1997.
- [32] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms (1st ed.)*. MIT Press and McGraw-Hill, 1990.
- [33] A. Coskun, J. Ayala, D. Atienza, T. Rosing, and Y. Leblebici. Dynamic thermal management in 3d multicore architectures. In *Design, Automation, and Test in Europe (DATE)*, pages 1410–1415, 2009.
- [34] V. Darera and L. Jenkins. Utilization bounds for rm scheduling on uniform multiprocessors. In *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on*, pages 315–321, 0-0.
- [35] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, Oct. 2011.
- [36] S. K. Dhall and C. L. Liu. On a Real-Time Scheduling Problem. *Operations Research*, 26(1):127–140, 1978.
- [37] M. Fan, V. Chaturvedi, S. Sha, and G. Quan. An analytical solution for multi-core energy calculation with consideration of leakage and temperature dependency. In *Low Power Electronics and Design (ISLPED), 2013 IEEE International Symposium on*, pages 353–358, Sep. 2013.
- [38] M. Fan and G. Quan. Harmonic-fit partitioned scheduling for fixed-priority real-time tasks on the multiprocessor platform. In *Embedded and Ubiquitous Computing (EUC), IFIP 9th International Conference on*, pages 27–32, Oct. 2011.
- [39] M. Fan and G. Quan. Harmonic semi-partitioned scheduling for fixed-priority real-time tasks on multi-core platform. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 503–508, Mar. 2012.
- [40] N. Fisher, J.-J. Chen, S. Wang, and L. Thiele. Thermal-aware global real-time scheduling on multicore systems. In *Proceedings of the 2009 15th IEEE Symposium on Real-Time and Embedded Technology and Applications, RTAS '09*, pages 131–140, 2009.

- [41] Y. Ge, P. Malani, and Q. Qiu. Distributed task migration for thermal management in many-core systems. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 579–584, Jun. 2010.
- [42] N. Guan, M. Stigge, W. Yi, and G. Yu. Fixed-Priority Multiprocessor Scheduling: Beyond Liu and Layland’s Utilization Bound. In *WiP Real-Time Systems Symposium (RTSS)*, Dec. 2010.
- [43] N. Guan, M. Stigge, W. Yi, and G. Yu. Fixed-Priority Multiprocessor Scheduling with Liu and Layland’s Utilization Bound. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2010.
- [44] N. Guan, M. Stigge, W. Yi, and G. Yu. Parametric Utilization Bounds for Fixed-Priority Multiprocessor Scheduling. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, May 2012.
- [45] R. Ha and J. W. S. Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *Distributed Computing Systems, 1994., Proceedings of the 14th International Conference on*, pages 162–171, Jun. 1994.
- [46] L. Hammond, B. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997.
- [47] C.-C. Han, K.-J. Lin, and C.-J. Hou. Distance-Constrained Scheduling and Its Applications to Real-Time Systems. *IEEE Transactions on Computers*, 45(7):814–826, Jul. 1996.
- [48] C.-C. Han and H.-Y. Tyan. A Better Polynomial-Time Schedulability Test for Real-Time Fixed-Priority Scheduling Algorithms. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, Dec. 1997.
- [49] Q. Han, M. Fan, and G. Quan. Energy minimization for fault tolerant real-time applications on multiprocessor platforms using checkpointing. In *Low Power Electronics and Design (ISLPED), 2013 IEEE International Symposium on*, pages 76–81, Sep. 2013.
- [50] V. Hanumaiah, R. Rao, S. Vrudhula, and K. S. Chatha. Throughput optimal task allocation under thermal constraints for multi-core processors. In *Proceedings of the 46th Annual Design Automation Conference, DAC ’09*, pages 776–781, New York, NY, USA, 2009. ACM.

- [51] V. Hanumaiah and S. Vrudhula. Energy-efficient operation of multi-core processors by dvfs, task migration and active cooling. *Computers(TC), IEEE Transactions on*, pages 1–14, 2012.
- [52] V. Hanumaiah, S. Vrudhula, and K. Chatha. Maximizing performance of thermally constrained multi-core processors by dynamic voltage and frequency control. pages 310–313, 2009.
- [53] V. Hanumaiah, S. Vrudhula, and K. Chatha. Performance optimal online dvfs and task migration techniques for thermally constrained multi-core processors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(11):1677–1690, Nov. 2011.
- [54] S. Heath. *Embedded Systems Design, 2nd edition*. Newnes, 2003.
- [55] Henry. A comparison of intels 32nm and 22nm core i5 cpus: Power,voltage, temperature, and frequency. *Intel Cor.*, pages <http://blog.stuffedcow.net/2012/10/intel32nm-22nm-core-i5-comparison/>, 2012.
- [56] H. Huang, M. Fan, and G. Quan. On-line leakage-aware energy minimization scheduling for hard real-time systems. In *ASP-DAC*, pages 677–682, 2012.
- [57] H. Huang and G. Quan. Leakage aware energy minimization for real-time systems under the maximum temperature constraint. In *DATE*, pages 1–6, 2011.
- [58] H. Huang, G. Quan, J. Fan, and M. Qiu. Throughput maximization for periodic real-time systems under the maximal temperature constraint. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 363–368, Jun. 2011.
- [59] L. Huang, F. Yuan, and Q. Xu. On task allocation and scheduling for lifetime extension of platform-based mp soc designs. *Parallel and Distributed Systems, IEEE Transactions on*, 22(12):2088–2099, Dec. 2011.
- [60] ITRS. *International Technology Roadmap for Semiconductors (2011 Edition)*. International SEMATECH, Austin, TX., <http://public.itrs.net/>.
- [61] R. Jayaseelan and T. Mitra. Temperature aware task sequencing and voltage scaling. *ICCAD*, pages 618–623, 2008.

- [62] R. Jejurikar, C. Pereira, and R. Gupta. Dynamic slack reclamation with procrastination scheduling in real-time embedded systems. *DAC*, pages 111 – 116, 2005.
- [63] D.-C. Juan, H. Zhou, D. Marculescu, and X. Li. A learning-based autoregressive model for fast transient thermal analysis of chip-multiprocessors. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 597 –602, Feb. 2012.
- [64] S. B. Judah Rosenblatt. Mathematical analysis for modeling. Dec. 1998.
- [65] M.-J. Jung, Y. R. Seong, and C.-H. Lee. Optimal RM Scheduling for Simply Periodic Tasks on Uniform Multiprocessors. In *Proc. ACM International Conference on Hybrid Information Technology (ICHIT)*, volume 321, pages 383–389, Aug. 2009.
- [66] A. Kandhalu, K. Lakshmanan, J. Kim, and R. Rajkumar. pCOMPATS: Period-Compatible Task Allocation and Splitting on Multi-core Processors. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2012.
- [67] S. Kato and N. Yamasaki. Real-Time Scheduling with Task Splitting on Multiprocessors. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Aug. 2007.
- [68] S. Kato and N. Yamasaki. Portioned Static-Priority Scheduling on Multiprocessors. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, Apr. 2008.
- [69] S. Kato and N. Yamasaki. Semi-Partitioned Fixed-Priority Scheduling on Multiprocessors. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2009.
- [70] P. Kumar and L. Thiele. Thermally optimal stop-go scheduling of task graphs with real-time constraints. In *ASP-DAC*, pages 123–128, 2011.
- [71] T.-W. Kuo and A. Mok. Load Adjustment in Adaptive Real-Time Systems. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, Dec. 1991.
- [72] K. Lakshmanan, R. Rajkumar, and J. Lehoczky. Partitioned Fixed-Priority Preemptive Scheduling for Multi-core Processors. In *Euromicro Conference on Real-Time Systems (ECRTS)*, Jul. 2009.

- [73] S. Lauzac, R. Melhem, and D. Mossé. An Efficient RMS Admission Control and Its Application to Multiprocessor Scheduling. In *IPPS/SPDP Parallel Processing Symposium*, Mar. 1998.
- [74] S. Lauzac, R. Melhem, and D. Mossé. An Improved Rate-Monotonic Admission Control and Its Applications. *IEEE Transactions on Computers*, 52(3):337–350, Mar. 2003.
- [75] R. Lawrence. Radiation characterization of 512mb sdrams. In *Radiation Effects Data Workshop, 2007 IEEE*, volume 0, pages 204–207, Jul. 2007.
- [76] C.-H. Lee and K. Shin. On-line dynamic voltage scaling for hard real-time systems using the edf algorithm. In *Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International*, pages 319 – 335, Dec. 2004.
- [77] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proc. Real Time Systems Symposium (RTSS)*, Dec. 1989.
- [78] J. Li, M. Qiu, J. Niu, Y. Zhu, and T. Chen. Real-time constrained task scheduling in 3d chip multi-processor to reduce peak temperature. In *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, pages 170–176, 2010.
- [79] W. Liao, L. He, and K. Lepak. Temperature and supply voltage aware performance and power modeling at microarchitecture level. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(7):1042 – 1053, 2005.
- [80] W. Liao, L. He, and K. Lepak. Temperature and supply voltage aware performance and power modeling at microarchitecture level. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(7):1042 – 1053, Jul. 2005.
- [81] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20:46–61, Jan. 1973.
- [82] G. Liu, M. Fan, and G. Quan. Neighbor-aware dynamic thermal management for multi-core platform. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 187 –192, Mar. 2012.

- [83] J. Liu. *Real-Time Systems*. Prentice Hall, NJ, 2000.
- [84] S. Liu, M. Qiu, W. Gao, X.-J. Tang, and B. Guo. Hybrid of job sequencing and DVFS for peak temperature reduction with nondeterministic applications. In *ICESS*, pages 1780–1787, 2010.
- [85] S. Liu, J. Zhang, Q. Wu, and Q. Qiu. Thermal-aware job allocation and scheduling for three dimensional chip multiprocessor. In *International Symposium on Quality Electronic Design (ISQED)*, 2010, pages 390–398, 2010.
- [86] Y. Liu, H. Yang, R. Dick, H. Wang, and L. Shang. Thermal vs energy optimization for dvfs-enabled processors in embedded systems. In *Quality Electronic Design, 2007. ISQED '07. 8th International Symposium on*, pages 204–209, Mar. 2007.
- [87] J. Lopez, J. Diaz, and D. Garcia. Minimum and maximum utilization bounds for multiprocessor rm scheduling. In *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pages 67–75, 2001.
- [88] J. Lopez, J. Diaz, and D. Garcia. Minimum and maximum utilization bounds for multiprocessor rate monotonic scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 15(7):642–653, July.
- [89] W.-C. Lu, H.-W. Wei, and K.-J. Lin. Rate Monotonic Schedulability Conditions Using Relative Period Ratios. In *Proc. IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2006.
- [90] C. Lung, Y. Ho, D. Kwai, and S. Chang. Thermal-aware online task allocation for 3d multi-core processor throughput optimization. In *Design, Automation, and Test in Europe (DATE)*, pages 1–6, Grenoble, France, 2011.
- [91] J. Markoff. Intel’s big shift after hitting technical wall. *New York Times*, 2004.
- [92] G. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8):114–117, May 1965.
- [93] D. Müller. Accelerated Simply Periodic Task Sets for RM Scheduling. In *Embedded Real Time Software and Systems*, May 2010.
- [94] P. Nahin. The story of $\sqrt{-1}$. In *Boston:Princeton University Press*, 1998.

- [95] M. Neumann, 1946, and R. J. Stern. *Nonnegative matrices in dynamic systems*. Wiley, New York, 1989.
- [96] P. Pop, V. Izosimov, P. Eles, and Z. Peng. Design optimization of time- and cost-constrained fault-tolerant embedded systems with checkpointing and replication. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(3):389–402, Mar. 2009.
- [97] G. Quan and V. Chaturvedi. Feasibility analysis for temperature-constraint hard real-time periodic tasks. *Industrial Informatics, IEEE Transactions on*, 6(3):329–339, Aug. 2010.
- [98] G. Quan and X. Hu. Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 828–833, 2001.
- [99] G. Quan, L. Niu, B. Mochocki, and X. Hu. Fixed priority scheduling for reducing overall energy on variable voltage processors. *RTSS'04*, pages 309–318, Dec. 2004.
- [100] G. Quan and Y. Zhang. Leakage aware feasibility analysis for temperature-constrained hard real-time periodic tasks. In *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, pages 207–216, Jul. 2009.
- [101] G. Quan, Y. Zhang, W. Wiles, and P. Pei. Guaranteed scheduling for repetitive hard real-time tasks under the maximal temperature constraint. *ISSS+CODES*, 2008.
- [102] J. Rabaey, A. Chandrakasan, and B. Nikolic. Digital integrated circuits: A design perspective. In *Englewood Cliffs, NJ: Prentice-Hall*, 2003.
- [103] C. A. B. REALPE. Programming languages towards multicore architectures crisis. In *TECHNOLOGY, PROGRAMMING, THOUGHTS AND OTHER ESSENTIALS*, 2013.
- [104] L. Schor, I. Bacivarov, H. Yang, and L. Thiele. Worst-case temperature guarantees for real-time applications on multi-core systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 87–96, Apr. 2012.
- [105] S. Sharifi, R. Ayoub, and T. Rosing. Tempomp: Integrated prediction and management of temperature in heterogeneous mpsoes. In *Design, Automation*

Test in Europe Conference Exhibition (DATE), 2012, pages 593–598, Mar. 2012.

- [106] K. Shin and P. Ramanathan. Real-Time Computing: A New Discipline of Computer Science and Engineering. *Proc. IEEE*, 82(1):6–24, Jan. 1994.
- [107] A. Shye, B. Scholbrock, and G. Memik. Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures. pages 168–178, 2009.
- [108] K. Skadron, M. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware computer systems: opportunities and challenges. *IEEE Micro*, 23(6):52–61, 2003.
- [109] K. Skadron, M. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. *ICSA*, pages 2–13, 2003.
- [110] J. Srinivasan, S. Adve, P. Bose, and J. Rivers. The impact of technology scaling on lifetime reliability. In *Dependable Systems and Networks, 2004 International Conference on*, pages 177 – 186, Jun.-Jul. 2004.
- [111] J. Srinivasan, S. V. Adve, P. Bose, J. Rivers, and C.-K. Hu. Ramp: A model for reliability aware microprocessor design. *IBM Research Report, RC23048 Computer Science*, 0, Dec. 2003.
- [112] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3):202–210, 2005.
- [113] A. Tanenbaum. *Modern operating systems*. Prentice Hall, 2001.
- [114] U. Tech. 2013 embedded market study. *UBM Tech Electronics’s Annual Survey of The Embedded Markets Worldwide*, 2013.
- [115] T.-H. Tsai and Y.-S. Chen. Thermal-aware real-time task scheduling for three dimensional multicore chip. In *ACM Symposium on Applied Computing (SAC-2012)*, pages 1618–1624, 2012.
- [116] I. Ukhov, M. Bao, P. Eles, and Z. Peng. Steady-state dynamic temperature analysis and reliability optimization for embedded multiprocessor systems. In *Design Automation Conference, 2012. DAC ’12.*, Jun. 2012.

- [117] W. Wolf. Multiprocessor system-on-chip technology. *Signal Processing Magazine, IEEE*, 26(6):50–54, 2009.
- [118] C.-Y. Yang, J.-J. Chen, L. Thiele, and T.-W. Kuo. Energy-efficient real-time task scheduling with temperature-dependent leakage. In *DATE*, pages 9–14, 2010.
- [119] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *FOCS*, pages 374–382, 1995.
- [120] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pages 374 –382, Oct. 1995.
- [121] D. Yeh, L.-S. Peh, S. Borkar, J. Darringer, A. Agarwal, and W. Hwu. Thousand-core chips [roundtable]. *Design Test of Computers, IEEE*, 25(3):272–278, May-Jun. 2008.
- [122] L. Yongpan and Y. Huazhong. Temperature-aware leakage estimation using piecewise linear power models. *IEICE transactions on electronics*, 93(12):1679–1691, 2010.
- [123] L. Yuan, S. Leventhal, and G. Qu. Temperature-aware leakage minimization technique for real-time systems. In *ICCAD*, pages 761–764, 2006.
- [124] B. Yun, K. Shin, and S. Wang. Predicting thermal behavior for temperature management in time-critical multicore systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 185–194, 2013.
- [125] S. Zhang and K. Chatha. Approximation algorithm for the temperature-aware scheduling problem. In *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on*, pages 281–288, 2007.
- [126] S. Zhang and K. S. Chatha. Thermal aware task sequencing on embedded processors. In *DAC*, pages 585 – 590, 2010.
- [127] S. Zhang, K. S. Chatha, and G. Konjevod. Near optimal battery-aware energy management. In *ISLPED*, pages 249–254, 2009.

- [128] B. Zhao, H. Aydin, and D. Zhu. Generalized reliability-oriented energy management for real-time embedded applications. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 381–386, Jun. 2011.
- [129] X. Zhou, Y. Xu, Y. Du, Y. Zhang, and J. Yang. Thermal management for 3d processors via task scheduling. In *International conference on Parallel processing*, pages 115–122, 2008.
- [130] C. Zhu, Z. Gu, L. Shang, R. Dick, and R. Joseph. Three-dimensional chip-multiprocessor run-time thermal management. *IEEE Transactions on Computer-Aided Design of Integrated circuits and SystCems*, 8(27):1479–1492, 2008.
- [131] D. Zhu and H. Aydin. Energy management for real-time embedded systems with reliability requirements. In *Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on*, pages 528–534, Nov. 2006.
- [132] D. Zhu, R. Melhem, and D. Mosse. The effects of energy management on reliability in real-time embedded systems. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design, ICCAD '04*, pages 35–40, Washington, DC, USA, 2004. IEEE Computer Society.
- [133] Y. Zhu and F. Mueller. Dvsleak: combining leakage reduction and voltage scaling in feedback edf scheduling. *SIGPLAN Not.*, 42(7):31–40, 2007.
- [134] J. Ziegler. Trends in electronic reliability: Effects of terrestrial cosmic rays. 2004.

VITA

MING FAN

2006	B.S., Software Engineering Beihang University Beijing, China
2009	M.S., Software Engineering Beihang University Beijing, China
2014	Ph.D., Electrical Engineering Florida International University Florida, USA

PUBLICATIONS

Ming Fan, Gang Quan, (2014). *Harmonic-Aware Multi-Core Scheduling For Fixed-Priority Real-Time Systems*, IEEE Transactions on Parallel and Distributed Systems (TPDS). (accepted)

Ming Fan, Qiushi Han, Gang Quan, Shangping Ren, (2014). *Multi-Core Partitioned Scheduling For Fixed-Priority Periodic Real-Time Tasks With Enhanced RBound*, Quality Electronic Design, International Symposium on (ISQED), 284–291.

Tianyi Wang, Ming Fan, Gang Quan, Shangping Ren, (2014). *Heterogeneity Exploration for Peak Temperature Reduction on Multi-Core Platforms*, Quality Electronic Design, International Symposium on (ISQED), 107–114.

Huang Huang, Ming Fan, Gang Quan, (2013). *Thermal Aware Overall Energy Minimization Scheduling for Hard Real-Time Systems*, Sustainable Computing: Informatics and Systems (SUSCOM), 3(4):274–285.

Ming Fan, Vivek Chaturvedi, Shi Sha, Gang Quan, (2013). *An Analytical Solution For Multi-Core Energy Calculation With Consideration Of Leakage And Temperature Dependency*, IEEE International Symposium on Low Power Electronics and Design (ISLPED), 353–358.

Qiushi Han, Ming Fan, Gang Quan, (2013). *Fault Tolerance with Shared-Recovery Checking Points*, IEEE International Symposium on Low Power Electronics and Design (ISLPED), 76–81. (best paper nomination)

Ming Fan, Vivek Chaturvedi, Shi Sha, Gang Quan, (2013). *Feasibility Analysis for Temperature Constrained Real-Time Scheduling on Multi-Core Platforms*, IEEE/ACM Design Automation Conference (DAC) Work-in-progress.

Ming Fan, Vivek Chaturvedi, Shi Sha, Gang Quan, (2013). *Thermal-Aware Energy Minimization for Real-Time Scheduling on Multi-core Systems*, ACM SIGBED Review - Special Issue on the Work-in-Progress (WiP) session of the 33rd IEEE Real-Time Systems Symposium (RTSS), 10(2):27–27.

Ming Fan, Gang Quan, (2012). *Harmonic Semi-Partitioned Scheduling For Fixed-Priority Real-Time Tasks On Multi-Core Platform*, Design, Automation & Test in Europe (DATE), 503–508.

Guanglei Liu, Ming Fan, Gang Quan, (2012). *Neighbor-Aware Dynamic Thermal Management for Multi-core Platform*, Design, Automation & Test in Europe (DATE), 187–192.

Huang Huang, Ming Fan, Gang Quan, (2012). *On-Line Leakage-Aware Energy Minimization Scheduling for Hard Real-Time Systems*, Asia and South Pacific Design Automation Conference (ASP-DAC), 677–682.

Guanglei Liu, Ming Fan, Gang Quan, Meikang Qiu, (2012). *On-Line Predictive Thermal Management under Peak Temperature Constraints for Practical Multi-core Platforms*, 2012, *Journal of Low Power Electronics*, 8(5):565–578.

Ming Fan, Gang Quan, (2011). *Harmonic-Fit Partitioned Scheduling For Fixed-Priority Real-Time Tasks On the Multiprocessor Platform*, IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC), 27–32.