



THE HONG KONG
POLYTECHNIC UNIVERSITY

香港理工大學

Pao Yue-kong Library

包玉剛圖書館

Copyright Undertaking

This thesis is protected by copyright, with all rights reserved.

By reading and using the thesis, the reader understands and agrees to the following terms:

1. The reader will abide by the rules and legal ordinances governing copyright regarding the use of the thesis.
2. The reader will use the thesis for the purpose of research or private study only and not for distribution or further reproduction or any other purpose.
3. The reader agrees to indemnify and hold the University harmless from and against any loss, damage, cost, liability or expenses arising from copyright infringement or unauthorized usage.

IMPORTANT

If you have reasons to believe that any materials in this thesis are deemed not suitable to be distributed in this form, or a copyright owner having difficulty with the material being included in our database, please contact lbsys@polyu.edu.hk providing details. The Library will look into your claim and consider taking remedial action upon receipt of the written requests.

THE HONG KONG POLYTECHNIC UNIVERSITY
DEPARTMENT OF COMPUTING

Overhead-Aware Real-Time Scheduling for
Streaming Applications on Multiprocessor
Systems-on-Chip

By
YI WANG

A Thesis Submitted in Partial Fulfillment of
the Requirements for the Degree of
Doctor of Philosophy

July 2011

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

_____(Signature)

_____(Name of Student)

ABSTRACT

With increasing demand for high-performance multimedia in battery-driven mobile devices, multicore architecture such as MPSoC (Multiprocessor System-on-Chip) is becoming widely adopted in embedded systems. When real-time streaming applications such as Internet video conferences and surveillance digital video recorders are executed on such chip multiprocessors, both time performance and energy consumption need to be considered. In order to fully take advantage of the multicore architecture of MPSoCs, various techniques have been proposed to explore and increase parallelism of streaming applications. These parallelization techniques usually impose a large amount of intercore communications with significant energy overhead and intercore communication overhead. By minimizing these overheads, a shorter period can be applied and system performance such as energy consumption and memory usage can be improved. In this thesis, we have attacked these problems from several aspects including the optimization of time performance, energy consumption, and memory usage for streaming applications on MPSoCs considering various overheads.

First, we focus on solving the energy optimization problem for real-time streaming applications on MPSoCs by combining task-level coarse-grained software pipelining with DPM (dynamic power management) and DVS (dynamic voltage scaling) considering transition overhead, intercore communication, and discrete voltage levels. We propose a two-phase approach to solve the problem. In the first phase, we propose a coarse-grained task parallelization algorithm to transform a periodic dependent task graph into a set of independent tasks by exploiting the periodic feature of streaming applications. In the second phase, we propose a genetic algorithm that can search and find the best schedule with the minimum energy consumption. Experimental results show that our approach can achieve a 24.4% reduction in energy consumption compared with previous work.

Second, we jointly optimize computation and communication task scheduling for streaming applications on MPSoCs with the objective of minimizing schedule length by totally removing intercore communication overhead. By minimizing schedule length, the system performance can be improved by adopting a smaller period or exploring the slacks generated for energy reduction with DVS. To guarantee the schedulability of communication tasks, we perform the schedulability analysis, and theoretically obtain the upper bound of the times needed to reschedule each computation task. Based on the analysis, we formulate the scheduling problem as an ILP (Integer Linear Programming) formulation and obtain an optimal solution. Experimental results show that our technique can achieve a 27.72% reduction in schedule length and a 14.98% reduction in energy consumption compared with previous work.

Third, we study the problem of removing intercore communication overhead for streaming applications on MPSoCs with the objective of minimizing the overall memory usage. The intercore communication overhead not only impacts time performance considerably but also influences the total memory usage of MPSoC architecture. Our basic idea is to let tasks with intra-period data dependencies transform to inter-period data dependencies so as to totally remove the intercore communication overhead. To solve the problem, we first perform analysis and obtain the bounds of the times needed to reschedule each task. Then we formulate the scheduling problem as an ILP model to obtain an optimal schedule. We also propose a heuristic approach to efficiently obtain a near optimal solution. Experimental results show that the proposed approach can significantly reduce the schedule length and improve the memory usage compared with previous work.

Keywords: Real-time, task scheduling, energy overhead, communication overhead, streaming applications, MPSoC.

PUBLICATIONS

Journal Papers

1. **Yi Wang**, Hui Liu, Duo Liu, Zhiwei Qin, Zili Shao, E. H.-M. Sha, “Overhead-Aware Energy Optimization for Real-Time Streaming Applications on Multiprocessor System-on-Chip”, *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, Volume 16, Issue 2, pages 14:1-14:32, March 2011.
2. **Yi Wang**, Duo Liu, Zhiwei Qin, Zili Shao, “Optimally Removing Inter-Core Communication Overhead for Streaming Applications on MPSoCs”, Accepted in *IEEE Transactions on Computers (TC)*, 2011.
3. Duo Liu, **Yi Wang**, Zili Shao, Minyi Guo, Jingling Xue, “Optimally Maximizing Iteration-Level Loop Parallelism”, Accepted in *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2011.
4. Duo Liu, **Yi Wang**, Zhiwei Qin, Zili Shao, Yong Guan, “A Space Reuse Strategy for Flash Translation Layers in SLC NAND Flash Memory Storage Systems”, Accepted in *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 2011.
5. Meng Wang, **Yi Wang**, Duo Liu, Zhiwei Qin, Zili Shao, “Compiler-Assisted Leakage-Aware Loop Scheduling for Embedded VLIW DSP Processors”, *Elsevier Journal of Systems and Software (JSS)*, Volume 83, Issue 5, pages 772-785, May 2010.
6. Miao Liu, Duo Liu, **Yi Wang**, Meng Wang, Zili Shao, “On Improving Real-Time Interrupt Latencies of Hybrid Operating Systems with Two-Level Hardware Interrupts”, *IEEE Transactions on Computers (TC)*, Volume 60, Number 7, pages 978-991, July 2011.

7. Hongxing Wei, Bin Wang, **Yi Wang**, Zili Shao, Keith C.C.Chan, “Staying-Alive Path Planning with Energy Optimization for Mobile Robots”, Accepted in *Elsevier Expert Systems With Applications (ESWA)*, 2011.

Conference Papers

1. **Yi Wang**, Luis Angel D. Bathen, Zili Shao, Nikil D. Dutt, “3D-FlashMap: A Physical-Location-Aware Block Mapping Strategy for 3D NAND Flash Memory”, in *Proceedings of the 15th Design, Automation and Test in Europe (DATE 2012)*, Dresden, Germany, March 12-16, 2012.
2. **Yi Wang**, Duo Liu, Zhiwei Qin, Zili Shao, “An Endurance-Enhanced Flash Translation Layer via Reuse for NAND Flash Memory Storage Systems”, in *Proceedings of the 14th Design, Automation and Test in Europe (DATE 2011)*, pages 14-20, Grenoble, France, March 14-18, 2011.
3. **Yi Wang**, Duo Liu, Zhiwei Qin, Zili Shao, “Memory-Aware Optimal Scheduling with Communication Overhead Minimization for Streaming Applications on Chip Multiprocessors”, in *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS 2010)*, pages 350-359, San Diego, CA, USA, November 30 - December 4, 2010.
4. **Yi Wang**, Duo Liu, Meng Wang, Zhiwei Qin, Zili Shao, “Optimal Task Scheduling by Removing Inter-core Communication Overhead for Streaming Applications on MP-SoC”, in *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2010)*, pages 195-204, Stockholm, Sweden, April 12-16, 2010.
5. **Yi Wang**, Duo Liu, Meng Wang, Zhiwei Qin, Zili Shao, Yong Guan, “RNFTL: A Reuse-Aware NAND Flash Translation Layer for Flash Memory”, in *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers and Tools for*

- Embedded Systems (LCTES 2010)*, pages 163-172, Stockholm, Sweden, April 12-16, 2010.
6. Zhiwei Qin, **Yi Wang**, Duo Liu, Zili Shao, Yong Guan, “MNFTL: An Efficient Flash Translation Layer for MLC NAND Flash Memory Storage Systems”, in *Proceedings of the 48th IEEE/ACM Design Automation Conference (DAC 2011)*, pages 17-22, San Diego, CA, USA, June 5-10, 2011.
 7. Zhiwei Qin, **Yi Wang**, Duo Liu, Zili Shao, “A Two-Level Caching Mechanism for Demand-Based Page-Level Address Mapping in NAND Flash Memory Storage Systems”, in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2011)*, pages 157-166, Chicago, IL, USA, April 11-14, 2011.
 8. Zhiwei Qin, **Yi Wang**, Duo Liu, Zili Shao, “Demand-Based Block-Level Address Mapping in Large-Scale NAND Flash Storage Systems”, in *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES +ISSS 2010)*, pages 173-182, Scottsdale, Arizona, USA, October 24-29, 2010.
 9. Meng Wang, **Yi Wang**, Duo Liu, Zili Shao, “Improving the Reliability of Embedded Systems with Cache and SPM”, in *Proceeding of the 2009 IEEE International Symposium on Trust, Security and Privacy for Pervasive Applications* in conjunction with *the 2009 IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS 2009)*, pages 825-830, Macau, China, October 12-15, 2009.
 10. Duo Liu, Tianzheng Wang, **Yi Wang**, Zhiwei Qin, Zili Shao, “A Block-Level Flash Memory Management Scheme for Reducing Write Activities in PCM-based Embedded Systems”, in *Proceedings of the 15th Design, Automation and Test in Europe (DATE 2012)*, Dresden, Germany, March 12-16, 2012.

11. Duo Liu, Tianzheng Wang, **Yi Wang**, Zhiwei Qin, Zili Shao, “PCM-FTL: A Write-Activity-Aware NAND Flash Memory Management Scheme for PCM-based Embedded Systems”, in *Proceeding of the 32nd IEEE Real-Time Systems Symposium (RTSS 2011)*, Vienna, Austria, November 29-December 2, 2011.
12. Meng Wang, Duo Liu, **Yi Wang**, Zili Shao, “Loop Scheduling with Memory Access Reduction under Register Constraints for DSP Applications”, in *Proceeding of the 2009 IEEE Workshop on Signal Processing Systems (SiPS 2009)*, pages 139-144, Tampere, Finland, October 7-9, 2009.

ACKNOWLEDGEMENTS

First and foremost, I want to express my gratitude to my supervisor, Prof. Zili Shao, whose expertise, understanding, and patience, added considerably to my graduate experience. I appreciate his vast knowledge and skill in many areas and his professional supervision. It is my great pleasure to be a student of Prof. Shao, and I want to thank him for supporting me over the years, and for giving me so much freedom to explore and discover new areas of research. Without his help and support, this body of work would not have been possible.

I want to thank my co-supervisor, Prof. Jiannong Cao, for his guidance, encouragement and advice. I also express my gratitude to the other members of Prof. Shao's research group - Duo Liu, Zhiwei Qin, Dr. Meng Wang, Tianzheng Wang, Guohui Wang, Chunjing Mao, and Luguang Wang - for the assistance they provided during my Ph.D. study. I also would like to thank all my teachers from whom I learned so much in my long journey of formal education. Specially thanks go to Prof. Zhijun Wang, Prof. Yan Liu, Prof. Bin Xiao, Prof. Qin Lu, Prof. Lei Zhang, and Dr. King Hong Cheung at the Hong Kong Polytechnic University. Furthermore, I acknowledge my gratitude to Dr. Yang Liu, Dr. Dongmin Guo, Dr. Xiaocui Sun, Kunfeng Lai, Dr. Guobin Liu, Weichao Li, Yi Yuan, and Dr. Hao Wang, who shared with me the pleasure of the Ph.D. study at the Hong Kong Polytechnic University.

I must acknowledge Prof. Nikil D. Dutt at University of California, Irvine, for offering me the opportunity to visit UCI. His truly scientist intuition and invaluable guidance inspires and enriches my intellectual maturity that I will benefit from, for a long time to come. I offer my regards and blessings to all of those who supported me in different respects during my visit at UCI. I would especially like to acknowledge Luis Bathen, Abbas Banaiyan, Kazuyuki Tanimura, and Jun Yong Shin, who have directly or indirectly collaborated on my

research.

I want to thank Prof. Henry C. B. Chan from the Hong Kong Polytechnic University for kindly being the Chairman of the Board of Examiners (BoE). I also thank Prof. Yi Pan from Georgia State University, and Prof. Cho-Li Wang from the University of Hong Kong, for kindly taking time out from their busy schedule to serve as my external examiners.

I recognize that this thesis would not have been possible without the financial assistance from the Hong Kong Polytechnic University. I appreciate Prof. Shao and the Department of Computing for offering me the travel grants to attend several international conferences. I acknowledge the grant for Research Student Attachment Program from the Hong Kong Polytechnic University for giving me the financial support to visit University of California.

Finally, I want to thank my family. They educated and guided me and have watched over me every step of way. I want to thank them for their endless love, support, and encouragement through my entire life, for letting me pursue my dream for so long and so far away from home, and for giving me the motivation to finish this thesis.

TABLE OF CONTENTS

CERTIFICATE OF ORIGINALITY	ii
ABSTRACT	iii
PUBLICATIONS	v
ACKNOWLEDGEMENTS	ix
LIST OF FIGURES	xiv
LIST OF TABLES	xvi
CHAPTER 1. INTRODUCTION.....	1
1.1 Related Work	3
1.1.1 Energy Optimization Techniques for Single and Multiple Processors	3
1.1.2 Communication-Aware Task Scheduling	5
1.1.3 Scheduling for Streaming Applications	6
1.2 The Unified Research Framework	7
1.3 Contributions	9
1.4 Thesis Organization	10
CHAPTER 2. OVERHEAD-AWARE ENERGY OPTIMIZATION FOR STREAMING APPLICATIONS ON MPSOCS	11
2.1 Overview	11
2.2 Models and Concepts	14
2.2.1 System Model	14
2.2.2 Task Model	14
2.2.3 Power Model	16
2.2.4 Retiming	18
2.2.5 Genetic Algorithm	19
2.2.6 Problem Statement	19
2.3 Motivational Example	20
2.4 Lower-bound Analysis.....	23

2.5	Task Parallelization and Scheduling	25
2.5.1	The RDAG Algorithm for Task Parallelization	27
2.5.2	The GeneS Algorithm for Energy Optimization	35
2.6	Experiments	42
2.6.1	Experimental Setup	42
2.6.2	Results and Discussion	44
2.7	Summary	54
 CHAPTER 3. OPTIMALLY REMOVING INTERCORE COMMUNICATION OVER- HEAD FOR STREAMING APPLICATIONS ON MPSOCS.....		56
3.1	Overview	56
3.2	Models and Concepts	58
3.2.1	System Model	59
3.2.2	Task Model	59
3.2.3	Static Schedule	60
3.2.4	Communication/Computation Overlapping and Retiming	61
3.2.5	Problem Analysis	64
3.2.6	Problem Statement	65
3.3	Schedulability Analysis	66
3.3.1	Bounds of Relative Retiming Values	66
3.3.2	Bounds of the Prologue Length	72
3.4	Optimal Joint Computation and Communication Task Scheduling	76
3.4.1	Joint Computation and Communication Task Scheduling (JCCTS)	76
3.4.2	The Extension for Minimizing Energy Consumption	81
3.5	Experiments	82
3.5.1	Experimental Setup	82
3.5.2	Results and Discussion	84
3.6	Summary	89
 CHAPTER 4. MEMORY-AWARE SCHEDULING WITH COMMUNICATION OVER- HEAD MINIMIZATION FOR STREAMING APPLICATIONS ON MPSOCS.....		90
4.1	Overview	90
4.2	Models and Concepts	92
4.2.1	System Model	92
4.2.2	Application Model and Communication Overhead	93

4.2.3	Static Schedule	94
4.2.4	Task Rescheduling and Retiming	94
4.3	Motivational Example and Problem Statement	96
4.4	Memory-Aware Task Scheduling for Minimizing the Intercore Communication Overhead	99
4.4.1	The Bounds Analysis of Retiming Value	100
4.4.2	The Analysis of Extra Memory Usage	103
4.4.3	Integer Linear Programming Formulation	105
4.4.4	A Heuristic Approach.....	109
4.5	Experiments	114
4.5.1	Experimental Setup	114
4.5.2	Results and Discussion	116
4.6	Summary	121
CHAPTER 5. CONCLUSION AND FUTURE WORK.....		122
5.1	Conclusion.....	122
5.2	Future Work	124
REFERENCES		125

LIST OF FIGURES

1.1	The Unified Research Framework.	8
2.1	A Multiprocessor SoC Architecture.	15
2.2	A Motivational Example. (a) The original DFG. (b) The retimed DFG using our RDAG algorithm. (c) The task information. (d) The schedule generated by the list scheduling in Landskov et al. [59] without power management (the energy is $137\mu J$). (e) The schedule generated by the DAG-based scheduling algorithm in Zhang et al. [115] with DVS and DPM (the energy is $73.4\mu J$). (f) The schedule generated by our technique (the energy is $25\mu J$). (g) The schedule generated by our technique with a tight timing constraint ($8\mu s$).	21
2.3	An Example of the RDAG Algorithm. (a) The original DFG G . (b) The retimed DFG G_r . (c) The static schedule generated from G . (d) The pipelined schedule generated from G_r	29
2.4	Latency and Memory Overhead of the RDAG Algorithm.	35
2.5	Chromosome Representation and Its Corresponding Task Schedule.	37
2.6	The Crossover Operator Generates New Chromosomes: Chromosomes 3 and Chromosome 4.	38
2.7	The Mutation Operator. Task T_D is selected to perform mutation, and its voltage level is changed from V_{dd_2} to V_{dd_1}	38
2.8	An Example of the GeneS Algorithm.	40
2.9	The Changing Tendency of Energy and Timing Constraint with Three Algorithms under Different Number of Processor Cores on Benchmark TGFF-1. ...	50
3.1	The MPSoC Architecture.	59
3.2	A DAG and Its Schedule.	61
3.3	Given the initial DAG and schedule in Figure 3.2, (a) a new schedule in which the intercore communication overhead caused by CT_3^1 and CT_4^1 is removed by overlapping communication and computation, and (b) the corresponding DAG with the node weight $R(T_1)$ of computation task T_1 changed to 1.	62
3.4	Given the initial DAG and schedule in Figure 3.2, (a) a new schedule in which the intercore communication overhead is totally removed, and (b) the corresponding DAG.	62
3.5	An Exemplary Task Schedule of Theorem 3.3.1.	68
3.6	An Exemplary Task Schedule of Property 3.3.1.	69
3.7	An Exemplary Task Schedule of Property 3.3.2.	71
3.8	An Exemplary Task Schedule of Property 3.3.3.	72

3.9	A Run Time Example of the Proposed Approach.	75
3.10	The Maximum Retiming Value R_{max} of Each Benchmark Running on 2, 3, and 4 Processor cores.	88
4.1	The MPSoC Architecture.	92
4.2	A Motivational Example. (a) A DAG. (b) The objective computation task schedule of the DAG. (c) The schedule considering intercore communication overhead. (d) and (e) two schedules in which intercore communication overheads are totally removed while they are with different memory usages...	97
4.3	A Task Schedule of Theorem 4.4.1.	102
4.4	Analysis of Memory Usage for Intercore Communication Task CT_j^i	104
4.5	A Runtime Example of MAOTS. (a) A DAG. (b) An initial schedule of the DAG. (c) Objective task schedule of all computation tasks that totally removes intercore communication overhead. (d) The execution time of each computation task and that of each communication task, and the release time of each computation task. (e) The bounds of relative retiming value of each pair of tasks. (f) The bounds of the retiming value of each task. (g) The optimal task schedule by our approach MAOTS.	107
4.6	Schedule Length by Task Schedules PEDF [115], Schedule2D [111], and the proposed approach (MAOTS) on 2, 3, and 4 processor cores.	115
4.7	Memory Usage of Schedule2D [111], JCCTS [106], and the Proposed Approach MAOTS on 2, 3, and 4 Processor Cores.	117
4.8	Memory Usage of Heuristic Approach (HMAOTS) and Memory-Aware Optimal Task Scheduling (MAOTS) on 2, 3, and 4 processor cores.	120

LIST OF TABLES

2.1	The Voltage Levels, Frequencies and Power Consumption based on the Power Model of the Mobile Athlon4 Processor [7].	42
2.2	Benchmark Descriptions and Characteristics.	44
2.3	The Energy of Each Benchmark under Various Timing Constraints on 2, 3, 4 Processor Cores.	46
2.4	The Energy of Each Benchmark under Various Timing Constraints on 6 and 8 Processor Cores.	47
2.5	The Comparison of Energy Consumption by the GeneS Algorithm and the Lower Bound Energy Consumption E_{LB}	49
2.6	The Comparison for the Schedules Generated by PEDF, SpringS, and GeneS on TGFF-2, a <i>fat</i> Task Graph.	51
2.7	The Prologue Latency of Our RDAG Algorithm with Different Timing Constraints.	52
2.8	Power Consumption and Transition Time for Memory.	53
2.9	The Memory Energy Consumption of the PEDF Algorithm and the RDAG Algorithm.	54
3.1	Comparison in Schedule Length of Our JCCTS Approach and the STC Algorithm in Chen et al. [24] on 2, 3, and 4 Processor Cores.	85
3.2	Comparison in Energy Consumption of Our JCCTS Approach and the FLSSR Algorithm in Zhu et al. [117] on 2, 3, and 4 Processor Cores.	87
4.1	Comparison in Time Cost of Heuristic Approach (HMAOTS) and Memory-Aware Optimal Task Scheduling (MAOTS) on 2, 3, and 4 processor cores.	119

CHAPTER 1

INTRODUCTION

Continuing advances in chip technology with the increasingly dense integration of intellectual property cores have created new opportunities in embedded applications. More and more embedded systems adopt the multiprocessor system-on-chip (MPSoC) to integrate multiple processor cores along with other hardware subsystems to implement a system. MPSoCs are not simply traditional multiprocessors shrunk to a single chip but have been designed to fulfill the unique requirements of embedded applications [109].

MPSoCs are often application specific, and they have very tight constraints in terms of computation power and memory space. In order to fully utilize the computation power of multiprocessor architecture, embedded applications can be customized to explore coarse-grained parallelism. Streaming applications are typically computationally-intensive with a lot of parallelism, and thus they are perfect candidates for being executed on MPSoCs. When streaming applications are running on MPSoCs, they are often required to provide real-time response with low power consumption. Task-level parallelism of streaming applications on MPSoCs is explored by executing multiple tasks on different processor cores concurrently. However, most of the existing optimal scheduling techniques on MPSoC architectures do not consider several overheads caused by task-level parallelism techniques.

In this thesis, we address the challenges in handling overheads in parallel processing of streaming applications on MPSoC architectures. Specifically, two major overheads (i.e., energy consumption overhead and intercore communication overhead) are considered in designing optimal task schedules. We present overhead-aware task scheduling schemes to optimize energy consumption, real-time performance, and memory usage for streaming applications on MPSoC architectures.

First, energy consumption becomes one of the important constraints for the design of multiprocessor system-on-chips, particularly for battery-operated embedded systems. With the increasing of operating frequency and transistor density of MPSoCs, energy consumption of these highly integrated and complex designs is becoming a major concern. DPM (dynamic power management) and DVS (dynamic voltage scaling) techniques are widely used to optimize energy consumption. When DPM and DVS are applied for energy optimization, several energy overheads (i.e., transition energy overhead associated with the sleep mode, transition energy overhead caused by voltage changes, and energy overhead by intercore communication) should be taken into account. In this thesis, we consider these energy overheads and provide a complete and energy efficient schedule.

Second, on-chip communication architectures have numerous sources of delay due to signal propagation along the wires, synchronization (e.g., handshaking), transfer modes (e.g., pipeline access), and arbitration mechanisms [82]. With the increasing of the number of system components and the processing gap between processor cores and memory, the delay caused by intercore communication will incur intercore communication overhead, which will significantly influence both time performance and power consumption of the system. In this thesis, we aim to totally remove the intercore communication overhead and to generate an energy-efficient optimal schedule with the minimum schedule length.

Third, memory usage is becoming an important factor for streaming applications on MPSoC architectures. Memory takes up a large chunk of on-chip area, as much as 70% in some cases [73]. Estimates indicate that this figure will go up to 90% in the coming years [5]. Parallel processing of streaming applications needs to store intermediate data streams across different processor cores, which directly accounts for the memory usage of the system. Intercore communication overhead directly influences the optimization of task schedule and significantly impacts the memory usage. In this thesis, we propose an optimal solution that can totally remove intercore communication overhead and generate a schedule with the minimum memory usage.

The rest of this chapter is organized as follows: Section 1.1 presents the related

work. Section 1.2 presents the unified research framework. Section 1.3 summarizes the contributions of this thesis. Section 1.4 gives the outlines of the thesis.

1.1 Related Work

In this section, we outline previous approaches related to task scheduling for streaming applications on multiprocessor system-on-chips. In the previous work, there has been work done in three main domains: (I) Energy optimization techniques for single and multiple processors, (II) Communication-aware task scheduling, and (III) Scheduling for streaming applications. We briefly describe these approaches, and detailed comparisons with representative techniques are presented in respective chapters.

1.1.1 Energy Optimization Techniques for Single and Multiple Processors

DVS is one of the most effective techniques for energy optimization. Therefore, a lot of DVS scheduling techniques have been proposed in previous work. For periodic independent tasks, the DVS scheduling has been extensively studied for single and multiple processors. For single processor, Aydin et al. [12] showed that for any periodic task, it is optimal for all of its task instances to run at the same processor speed on an ideal DVS processor. Jejurikar and Gupta [44] considered periodic tasks on a processor with discrete speed levels. Several studies have been conducted in the DVS scheduling on single processor based on dynamic priority [12, 23, 74] or fixed priority [16, 96, 100]. For multiple processors, several studies [4, 13, 22] focused on DVS scheduling on homogeneous multiple processors while other work [41, 69, 70, 114] focused on heterogeneous multiple processors. Recent work [11, 116] studied system-wide energy minimization for periodic and aperiodic tasks on a processor with continuous speed levels. They separate task execution into on-chip/off-chip cycles, which are applicable for both CPU and memory. Niu and Quan [77] proposed an approach for system-wide dynamic power management for multimedia portable devices. In all of the above work, the task model is based on *periodic independent tasks* at process or thread level.

In this thesis, we consider *periodic dependent tasks* which can better model stream-based applications such as MPEG-4 AVC decoder [107].

There have been a lot of studies of DVS scheduling for dependent tasks on multiprocessor systems with multiple voltage levels. Hua and Qu [40] studied the voltage setup problem and proposed an approach to select optimal voltage levels. Gruian and Kuchcinski [36] introduced a scheduling approach. In their approach, based on a given fixed task assignment, the delays of all tasks are scaled down by the ratio of the timing constraint over the critical path length. Luo and Jha [69] proposed an approach to evenly distribute slacks based on a fixed task scheduling. Zhang et al. [115] proposed a framework that integrates task scheduling and voltage selection together to minimize the energy consumption for dependent tasks on multiple processors. However, these works focus on *dependent* task model instead of *periodic dependent* task model. With the dependent task model, only intra-iteration data dependencies are considered. In our work, we further exploit inter-iteration data dependencies by utilizing the periodic characteristics of the periodic dependent task model.

Several recent studies have explored the periodic behavior of periodic tasks with pipelining and parallel processing [50,62,99]. In Kim et al. [50], a power reduction technique is proposed to optimize energy by exploring pipelining and parallel processing in uniprocessor systems. This uniprocessor-based technique cannot be directly applied to solve our multiprocessor-based problem. In Shao et al. [99], a loop scheduling technique is proposed to minimize energy by exploring inter-iteration dependencies for applications with loops on multi-core systems. The given technique, however, is based on loop optimization with instruction-level parallelism; thus, it is not applicable to the periodic task model. In Li and Martinez [62], an analytical model is developed to study the power-performance issues of running parallel applications on chip multiprocessors. The proposed technique shows that, parallel computing can bring significant power-performance benefits over uniprocessor systems.

1.1.2 Communication-Aware Task Scheduling

Since the early 1990s, several on-chip bus-based communication architecture standards have been proposed to handle the communication needs of emerging SoC designs [82]. These popular standards include ARM Microcontroller Bus Architecture (AMBA) versions 2.0 [8] and 3.0 [9], IBM CoreConnect [42], STMicroelectronics STBus [102], Sonics SMART Interconnect [101], OpenCores Wishbone [79], and Altera Avalon [6]. Our proposed approach is based on ARM-based architecture, but the proposed approach can be extended to all bus-based communication architectures.

There is a large body of work dealing with performance estimation models for communication architecture. The estimation-based models for communication architecture performance exploration can be roughly classified into three categories: the static estimation, the dynamic estimation, and the hybrid estimation [82].

Static estimation methods try to estimate the communication delay in applications statically. Some early work [86, 89, 113] focused on the estimation of communication delay for high-level synthesis in the context of distributed embedded real-time systems. Other work proposed communication delay model to ensure the performance constraints are satisfied during the design flow of hardware/software component integration [34, 55, 56]. Renner et al. [92, 93] proposed the communication model that considers the delay caused by the specific protocol. Cho et al. [25] proposed a delay model for AMBA AHB [8] single shared bus.

Dynamic or simulation-based performance estimation models provide more accurate estimation. There have been several approaches based on cycle accurate models [67], pin-accurate bus cycle accurate models [47, 97, 98], transaction-based bus cycle accurate models [18, 19, 54, 78, 83, 84]. Hybrid communication architecture performance estimation approaches attempts to combine the static estimation and dynamic simulation-based approaches to speed up communication architecture performance estimation while generating accurate performance exploration results. Some approaches are trace-based approach [57, 58], while other approaches are queuing theory-based approach [52, 53]. These communication performance estimation can provide accurate results that can be utilized in the

design of proposed overhead-aware scheduling schemes.

The intercore communication scheduling on multi-core architectures has been investigated in previous work. Several studies have been conducted in communication and task scheduling for mesh network [27,39,118], grid environment [120], cluster architecture [110], and multi-layer bus architecture [38]. Our work focuses on the shared bus architecture, so these techniques cannot be directly applied. Based on the shared bus architecture, which is the most widely used interconnection architecture, several techniques have been proposed in bus access policy and bus access scheduling [35,37,94]. In [87,88], real-time bus scheduling policies including event-triggered scheduling and time-triggered scheduling have been investigated. In Lehoczky and Sha [60], real-time bus scheduling algorithms considering the issues of task preemption, priority level granularity, and buffering are proposed. There have been also several studies in communication mechanisms such as task migration and data migration for real-time multimedia MPSoC architectures [2,49,80]. These techniques can provide good solutions for bus arbitration and communication synthesis. However, all the aforementioned work is not designed to reduce intercore communication overhead. Several communication-aware task allocation and scheduling frameworks for MPSoC architectures are proposed [29,68,95,105]. By increasing the parallelism, these techniques may cause more intercore communications. Our technique is a good supplement for these techniques by helping effectively reducing intercore communication overhead.

1.1.3 Scheduling for Streaming Applications

In this thesis, streaming applications are modeled as periodic dependent tasks. In previous work, a lot of techniques have been proposed to solve the scheduling problem for periodic tasks. For scheduling independent tasks, a number of studies have been conducted [15,22,41]. These techniques, however, cannot be directly applied to perform scheduling for periodic dependent tasks. Several techniques have been proposed to solve the scheduling problem for periodic dependent tasks on multicore architectures [1,24,26,64,117]. The above techniques can generate optimal or near-optimal task schedules. However, in these

techniques, intercore communication overhead is not considered. So they may not provide good solutions to our problem.

Several approaches have been conducted to improve the performance of streaming applications with different architectures [31, 43, 119]. In Foroozannejad et al. [31], a fine-grained analysis of temporal behavior of buffer allocation for streaming application is performed. In Issenin and Dutt [43], an energy-aware co-synthesis of both memory and TDMA bus-based communication architecture for streaming applications is proposed. In Zhu et al. [119], a scheduling technique for streaming applications on hybrid CPU/FPGA architectures is proposed to minimize the buffer requirement with throughput guarantees. For the above approaches, the data dependency relations for streaming applications are fixed inside each period. It may limit the optimization for performance (e.g., throughput, schedule length, and energy consumption). In Wang et al. [106], a task scheduling technique that changes the data dependency relations across different periods is proposed. However, it does not consider the extra memory usage to store the data among different periods. Our technique can combine with the above approaches to generate an optimal task schedule with the minimum memory usage.

1.2 The Unified Research Framework

In this section, we present the unified research framework for the proposed techniques. Figure 1.1 illustrates the sketch of our research framework.

In this thesis, streaming applications that process streams of data are modeled as periodic dependent tasks, in which streams of data are communicated from task to task. Periodic dependent tasks are represented by a Directed Acyclic Graph (DAG).

The system architecture adopted in this thesis is a typical MPSoC system, which consists of a set of processor cores, a shared bus, a bus arbiter, and a shared on-chip memory. A shared bus is adopted as it is one of the most widely used on-chip communication architectures.

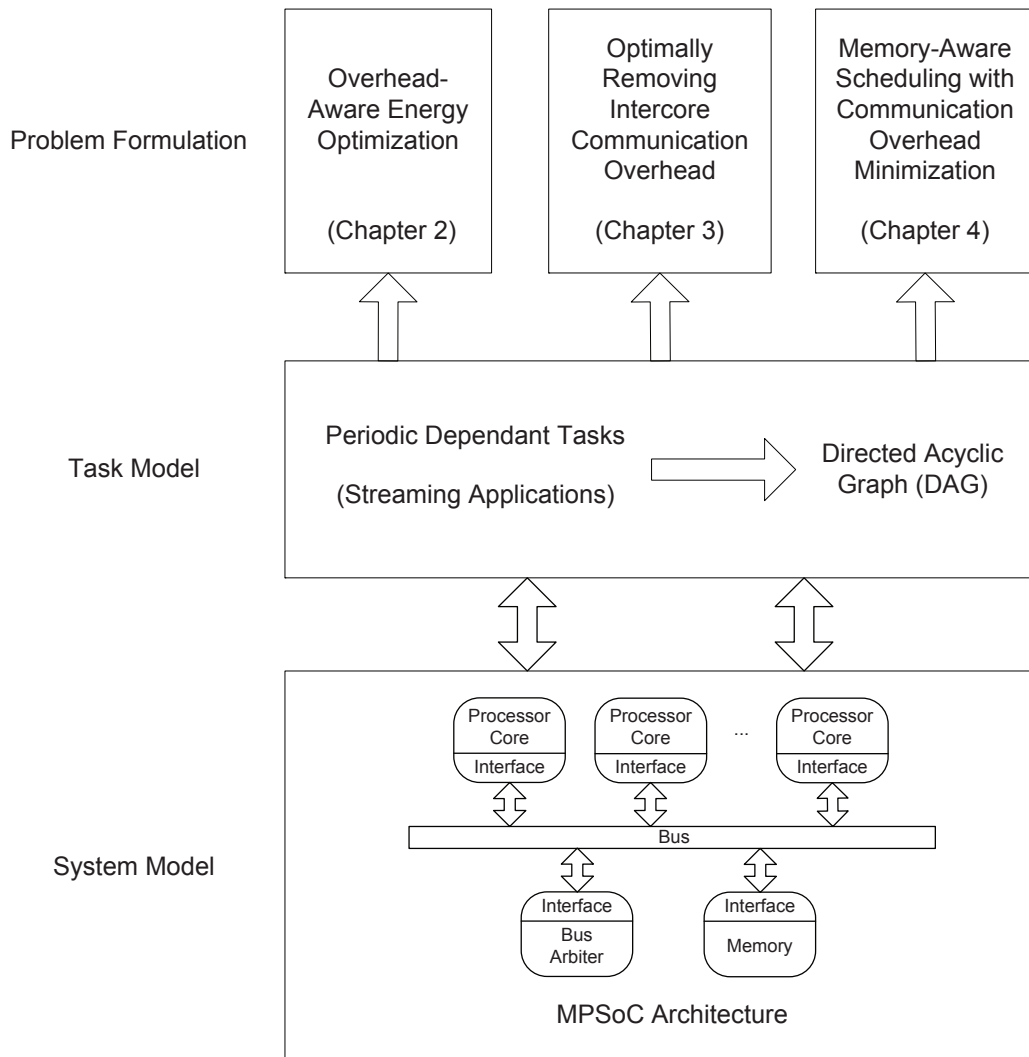


Figure 1.1. The Unified Research Framework.

Based on the system model and the application model, three task scheduling problems are formulated to optimize energy consumption, timing performance, and memory usage of streaming applications considering several overheads caused by parallel processing of streaming applications on MPSoC architectures.

For the first scheme, in Chapter 2, we studied the energy optimization problem for real-time streaming applications on multiprocessor system-on-chips considering several energy overheads. For the second scheme, in Chapter 3, we aimed to totally remove intercore communication overhead and to generate an optimal task schedule in which the schedule

length can be minimized with the minimum prologue length. For the third scheme, in Chapter 4, we studied the problem of totally removing intercore communication overhead with the objective of minimizing the overall memory usage.

1.3 Contributions

The contributions of this thesis are summarized as follows.

- The major contribution of this thesis is the idea of considering several overheads arising from the adoption of parallel processing of streaming applications on MPSoC architectures. To solve the overhead-aware optimization problem, this thesis introduces three scheduling schemes. These schemes are built on previous work on task-level software pipelining, which could provide good performance for computationally-intensive streaming applications on multiprocessor architectures. We extend the retiming technique and change the data dependency relationships across different periods. This could fully utilize the computation power of multiprocessor architectures, and get time slots to exploit DVS and DPM techniques to reduce energy consumption.
- To handle various energy overheads, we transform a periodic dependent tasks into independent tasks. By completely removing precedence relations, abundant idle task slacks incurred by precedence relations among tasks can be utilized. In this way, more opportunities are provided to do scheduling with energy optimization. Then we propose a genetic algorithm to search the best schedule considering several energy overheads.
- To handle intercore communication overheads, we reschedule several tasks into the previous execution process (or prologue) and overlap the execution of computation tasks with that of communication tasks. This can generate a task schedule by adopting a shorter period and with the maximum throughput. Different from the traditional pipelining technique, the number of pipeline stages of each task is determined by the allocation of computation tasks and the associated intercore communication

tasks. Two scheduling schemes are proposed with the objectives of minimizing the maximum retiming value, and minimizing the extra memory usage.

- The schedulability analysis in this thesis provides very tight bounds. For the problem of removing intercore communication overheads, the results of the analysis can be utilized in the ILP model to efficiently obtain the solution.
- We implement a simulator based on the ARM-based MPSoC system architecture to evaluate the proposed schemes. We conduct experiments and compare with representative schemes. Experimental results prove the effectiveness of the proposed schemes.

1.4 Thesis Organization

The rest of this thesis is organized as follows.

- In Chapter 2, we handle the energy overhead and propose a two-phase approach to solve problem. In the first phase, we present the RDAG algorithm to transform a set of periodic dependant tasks into a set of independent tasks. In the second phase, we propose a genetic algorithm GeneS to generate the schedule with the minimum energy consumption.
- In Chapter 3, we handle the intercore communication overhead and propose a scheme called JCCTS to obtain an optimal task schedule that totally removes intercore communication overhead with the minimum prologue length.
- In Chapter 4, we handle the intercore communication overhead and propose a scheme called MAOTS to obtain an optimal task schedule with the minimum extra memory usage. We also present a heuristic approach HMAOTS to efficiently obtain a near optimal schedule.
- In Chapter 5, we present conclusions and possible future directions of research arising from this work.

CHAPTER 2

OVERHEAD-AWARE ENERGY OPTIMIZATION FOR STREAMING APPLICATIONS ON MPSOCS

2.1 Overview

With increasing demand for high-performance multimedia in battery-driven mobile devices, multi-core architecture such as MPSoC (Multiprocessor System-on-Chip) is becoming widely adopted in embedded systems. The examples include TI TMS320DM6467 DaVinci processors, Freescale MSC8122 and MSC8126 multicore DSP processors, ARM ARM11 MPCore and Intel Atom processors. Some multicore processors such as ARM ARM11 MPCore and Intel Atom processors provide multiple voltage levels for low power optimization. When real-time streaming applications such as Internet video conferences and surveillance digital video recorders are executed on such chip multiprocessors, both time performance and energy consumption need to be considered as energy consumption is one of the most important performance metrics in embedded systems. Therefore, it becomes an important research problem to optimize energy consumption for streaming applications on MPSoCs.

To solve this problem, several issues need to be taken into account. First, a streaming application can be modeled as periodic dependent tasks in real-time systems, in which a stream of data is treated as a sequence of requests that are serviced by the streaming application when arrived [111]. In this chapter, the periodic behavior of dependent tasks is explored with task-level software pipelining [20, 21, 85]. Second, both dynamic power management (DPM) and dynamic voltage scaling (DVS) should be applied for energy optimization. DPM exploits idle times of a processor and turns off power supply so as to reduce static energy caused by leakage power [45]. DVS reduces energy consumption by adjusting supply volt-

ages of processors [45]. Here, we assume that DPM and DVS can be applied for each processor core independently. As a chip multiprocessor may contain many processor cores, we need to consider the trade-off between adjusting voltages by DVS and turning off by DPM in energy optimization. Third, various practical issues including transition overhead caused by mode/voltage changes in DPM/DVS, discrete voltage levels and intercore communication should be considered for a practical solution. Taking all issues into consideration, in the chapter, we focus on energy consumption optimization for streaming applications by combining task-level coarse-grained software pipelining with DVS and DPM techniques.

Our work is closely related to the previous work [1, 14, 48, 66, 106, 111, 115], in which periodic dependent tasks are modeled by task graphs (Directed Acyclic Graph (DAG)). In Xu et al. [111], an energy-aware scheduling technique is proposed to minimize energy consumption while satisfying both throughput and response time with pipelining. In Zhang et al. [115], an energy optimization framework is proposed to integrate task scheduling and voltage selection together. In Acharya and Mahapatra [1], a technique is developed to utilize slacks based on service rate and change in intervals for static and dynamic scheduling schemes, and a fault-tolerant scheme is incorporated into the slack management technique to implement reliable systems. In Kianzad et al. [48], an integrated framework combining task assignment, scheduling, and power management using genetic algorithm is proposed. In Bamabha and Bhattacharyya [14], a periodic graph model is explored to effectively select voltage levels of iterative applications on multiprocessor systems. In the above work, the intra-iteration precedence relations of a task graph are not changed, which limits the optimization for both performance and energy. In Liu et al. [66], coarse-grained software pipelining is applied to solve real-time streaming applications on MPSoC, and a DVS scheduling technique is proposed to optimize energy based on it. However, the technique is built on the assumption that there are no intercore communication overhead and transition overhead with mode/voltage changes in DVS. In Wang et al. [106], a task scheduling technique that changes the data dependency relations across different periods is proposed to effectively remove intercore communication overhead. However, it does not consider several overheads (i.e., transition overhead, sleep overhead). In this work, we consider various practical issues

and propose a genetic algorithm to solve the problem.

In this chapter, we propose a two-phase approach to solve the energy minimization problem for periodic dependent tasks on MPSoC architectures considering various practical issues. In our approach, we first completely remove the precedence relations of tasks based on task-level software pipelining, and then perform energy optimization. Our two-phase approach is summarized as follows.

- In the first phase, we propose a coarse-grained task-level software pipelining algorithm called RDAG to transform periodic dependent tasks into a set of independent tasks based on the retiming technique [61]. In RDAG, we regroup tasks and put tasks from different periods into one loop kernel so as to completely remove precedence relations. In this way, abundant idle slacks incurred by precedence relations among tasks can be utilized. In addition, after transforming a dependent task graph into a set of independent tasks, more opportunities are provided to do scheduling with energy optimization by simultaneously considering multiple factors such as dynamic/static power, sleep/voltage transition overheads, and intercore communication.
- In the second phase, we propose a scheduling algorithm, GeneS, to optimize energy consumption based on the results obtained in the first phase. GeneS is a genetic algorithm, and it can search and find the best schedule within the solution space generated by gene evolution.

To the best of our knowledge, this is the first work to solve the energy optimization problem for periodic dependent tasks on MPSoCs by combining task-level software pipelining with DVS and DPM considering various practical issues.

We conduct experiments on a set of benchmarks from Embedded Systems Synthesis Benchmarks Suite (E3S) [104] and TGFF [28]. The benchmarks from E3S consist of various multimedia applications such as JPEG compression/decompression, RGB to CYMK conversion, RGB to YIQ conversion, and FFT/IFFT. TGFF is used to generate several synthetic task graphs. We compare our technique with the approach in Zhang et al. [115] that

applied DVS and DPM but without software pipelining. The experimental results show that our technique can achieve better results compared with the previous work. On average, our GeneS algorithm can achieve a 24.4% reduction in energy consumption compared with the approach in Zhang et al. [115]. For systems with tight timing constraints, our approach can obtain a feasible solution while the approach in Zhang et al. [115] cannot.

The remainder of this chapter is organized as follows. Section 2.2 describes models and defines the problem. Section 2.3 gives a motivational example. Section 2.4 analyzes the lower bound of the energy consumption. Our two-phase approach is presented in Section 2.5. Experimental results are provided in Section 2.6. The conclusion is presented in Section 2.7.

2.2 Models and Concepts

In this section, we first introduce system architecture, application model and some basic concepts that will be used in the later sections and then define the problem.

2.2.1 System Model

In this chapter, we employ an MPSoC architecture shown in Figure 2.1. The architecture consists of M processor cores $\{PE_1, PE_2, \dots, PE_M\}$, and each processor core has its own data and program memory. The programmable bus controller implements a predefined bus protocol and assigns bus access rights to individual cores. If a task needs to read data that are not available in its local memory, intercore communication happens.

2.2.2 Task Model

Streaming applications are modeled as periodic dependent tasks. We use Directed Acyclic Graph (DAG) to represent periodic dependent tasks. DAG is a special case of Data Flow Graph (DFG). A DFG, $G = (V, E, \rho, C)$, is a node-weighted and edge-weighted directed graph. $V = \{T_1, T_2, \dots, T_n\}$ is the node set, and each node denotes a periodic task. $C(T_i)$ is

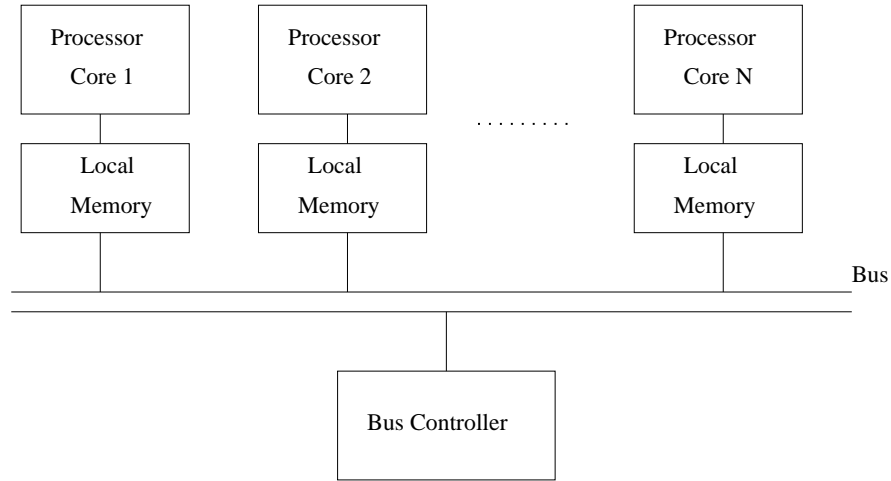


Figure 2.1. A Multiprocessor SoC Architecture.

the number of clock cycles to compute task T_i ($T_i \in V$). E is the edge set to represent data dependency among task nodes. An edge $(T_i, T_j) \in E$ represents that the data generated by T_i is needed in order to compute T_j , and $com(T_i, T_j)$ is used to represent the data volume associated with tasks T_i and T_j . $\rho(T_i, T_j)$ is a function to represent the number of delays for an edge $(T_i, T_j) \in E$. The edge without delay represents the intra-iteration data dependency, which means the dependencies inside one period, while the edge with delays represents the inter-iteration data dependency, which means the dependencies among different periods. The number of delays represents the number of periods involved. For an edge $(T_i, T_j) \in E$, initially $\rho(T_i, T_j) = 0$; later it may be changed as we group tasks from different periods into one period in our technique.

A *static* schedule of a given DFG is a repeated pattern of an execution of the corresponding periodic dependent tasks. In other words, a static schedule is used to represent the execution of **one period** of periodic dependent tasks. In this chapter, the execution of one period is called **one iteration** as well. A schedule implies both schedule step assignment and processor core allocation. A static schedule must obey the dependency relations of the DAG portion of the DFG. The DAG is obtained by removing all edges with delays in the DFG.

2.2.3 Power Model

In this chapter, a processor core in an MPSoC can support both DPM (Dynamic Power Management) and DVS (Dynamic Voltage Scaling). A processor core can operate at k different voltage/frequency levels, $\{(V_{dd_1}, f_1), (V_{dd_2}, f_2), \dots, (V_{dd_k}, f_k)\}$, in which it consumes both dynamic power and static power. Without loss of generality, we assume that the voltage levels from V_{dd_1} to V_{dd_k} are in ascending order, in which V_{dd_1} is the lowest voltage level and V_{dd_k} is the highest voltage level. The voltage level of a processor core can be changed independently by voltage-level-setting instructions without influencing other cores. A processor core has one sleep mode as well in which it is deactivated and dissipates reduced power.

Given a static schedule S , its total energy consumption, $E_{total}(S)$, can be represented as follows:

$$E_{total}(S) = E_{t_dynamic}(S) + E_{t_static}(S) + E_{t_sleep}(S) + E_{t_sleepOH}(S) + E_{t_tranOH}(S) + E_{t_comm}(S). \quad (2.1)$$

Here, $E_{t_dynamic}(S)$ is the total dynamic energy consumption, $E_{t_static}(S)$ is the total static energy consumption, $E_{t_sleep}(S)$ is the total energy consumption in the sleep mode, $E_{t_sleepOH}(S)$ is the total transition energy overhead associated with the sleep mode, $E_{t_tranOH}(S)$ is the total transition energy overhead caused by voltage changes, and $E_{t_comm}(S)$ is the total energy overhead by intercore communication. Next, we introduce how to calculate them one by one.

The dynamic power consumption of a processor core at a voltage level V_{dd} is calculated based on the power model in Rabaey et al. [91]:

$$P_{dynamic}(V_{dd}) = C_{SW} \cdot f_{op} \cdot V_{dd}^2, \quad (2.2)$$

where C_{SW} is the capacitance, and f_{op} is the frequency of a processor core at voltage level V_{dd} . Then, the dynamic energy consumption of a task T_i running at voltage level V_{dd} is

$$E_{dynamic}(T_i, V_{dd}) = P_{dynamic}(V_{dd}) \cdot \frac{C(T_i)}{f_{op}} = C(T_i) \cdot C_{SW} \cdot V_{dd}^2, \quad (2.3)$$

where $C(T_i)$ is the number of cycles of task T_i .

Different leakage sources contribute to the static power consumption in a processor core. The major contributors are the subthreshold leakage current and the reverse bias junction current. Based on the model in Martin et al. [72], the static power consumption, P_{static} , can be expressed as

$$P_{static}(V_{dd}) = I_{subn} \cdot V_{dd} + |V_{bs}| \cdot I_j, \quad (2.4)$$

where I_{subn} is the subthreshold current, V_{bs} is the body bias voltage, and I_j is the reverse bias junction current.

For the energy consumed in the sleep mode, let t_{sleep} be the time duration in which a processor core is in the sleep mode and let P_{sleep} be the corresponding power consumption. Then the energy consumed in the sleep mode, E_{sleep} , is calculated by

$$E_{sleep} = P_{sleep} \cdot t_{sleep}. \quad (2.5)$$

It takes both time and energy for a processor core to enter into and exit from the sleep mode. Let $t_{sleepOH}$ be the time transition overhead and $E_{sleepOH}$ be the energy transition overhead associated with one transition for entering into and exiting from the sleep mode. The total energy transition overhead can be obtained by the product of $P_{sleepOH}$ and the transition time.

Besides dynamic power and static power, we need to consider both time and energy overheads during voltage transitions. According to the power model in [17, 76], for a voltage change from V_{dd_i} to V_{dd_j} , the time transition t_{TRAN} can be calculated by

$$t_{TRAN} = \frac{2 \cdot C_{DD}}{I_{MAX}} \cdot |V_{dd_j} - V_{dd_i}|, \quad (2.6)$$

where C_{DD} is the capacitance of the voltage converter, and I_{MAX} is the maximum output current of the converter.

The energy transition overhead E_{tranOH} includes the energy consumed by the voltage converter, $E_{TRAN-DC}$, and the energy consumed by a processor core during the transition, $E_{TRAN-CPU}$, so

$$E_{tranOH} = E_{TRAN-DC} + E_{TRAN-CPU}. \quad (2.7)$$

For a voltage change from V_{dd_i} to V_{dd_j} , $E_{TRAN-DC}$ and $E_{TRAN-CPU}$ are calculated by

$$E_{TRAN-DC} = \alpha \cdot C_{DD} \cdot |V_{dd_i}^2 - V_{dd_j}^2| \quad (2.8)$$

$$E_{TRAN-CPU} = P_{TRAN} \cdot t_{TRAN}. \quad (2.9)$$

Here, α is the efficiency factor of the voltage converter, P_{TRAN} is the power consumption at the voltage level entered in the transition.

The energy consumed by intercore communication between task T_i and task T_j is calculated by

$$E_{comm}(T_i, T_j) = P_{comm} \cdot \frac{com(T_i, T_j)}{B}, \quad (2.10)$$

where P_{comm} is the power consumption of the shared bus in one clock cycle, $com(T_i, T_j)$ is the data volume transferred between tasks T_i and T_j , and B is the bus bandwidth. From the above, we can obtain the total energy consumption of a schedule by adding all components together.

2.2.4 Retiming

Retiming is originally proposed to minimize the cycle period of a synchronous circuit by evenly distributing registers [61]. It has been extended to schedule data flow graphs on parallel systems [20, 21, 85]. In this chapter, we generate a new loop kernel by regrouping tasks from different periods so as to remove intra-iteration dependencies. We use retiming to model this regrouping.

Given a DFG $G = (V, E, \rho, C)$, a *retiming* r of G is a function that maps each node T_i in V to an integer $r(T_i)$. Basically, by retiming a task node in a DFG once, a delay is drawn from *each* of its incoming edges, and then pushed to *each* of its outgoing edges. Every retiming operation corresponds to a software pipelining operation, and as shown in Section 2.5.1, retiming a node once means one copy of this task is moved into the prologue. From the program point of view, the retiming technique regroupes a loop body and attempts to remove intra-iteration dependencies among nodes. The transformed loop body after the

retiming can be obtained based on the retiming values of nodes [20]. The delay count of an edge (T_i, T_j) ($(T_i, T_j) \in E$) after retiming, $\rho_r(T_i, T_j)$, is named the retimed delay count, and can be calculated by $\rho_r(T_i, T_j) = r(T_i) - r(T_j)$ [61]. An edge $(T_i, T_j) \in E$ with delay count $\rho(T_i, T_j) > 0$ means that the computation of node T_j at the ℓ th iteration requires data produced by node T_i at the $\ell - \rho(T_i, T_j)$ th iteration. A retiming function r is *legal* if the retimed delay counts of all edges in the retimed graph G_r are nonnegative. An illegal retiming function occurs when the retimed delay count of one edge becomes negative, and this situation implies a reference to nonavailable data from a future period.

2.2.5 Genetic Algorithm

Genetic Algorithm (GA) is an iterative procedure to simulate evolution for a population of candidate solutions to the optimization problem. In this chapter, we adopt genetic algorithms to solve our energy optimization problem. In a genetic algorithm, each iteration step is called a *generation*, and each candidate solution is called a *chromosome* that consists of several pairs of *genes* [75]. A genetic algorithm begins with an initial population of chromosomes. Two genetic reproduction operators, *mutation* and *crossover* (recombination), are designed to create new chromosomes for the next generation. The number of chromosomes in each generation is constant. Thus, a fitness function is used to evaluate each chromosome, and only those chromosomes with higher fitness value will be selected to form the new population of the next generation. This evolution process is repeated until a termination condition has been reached. The chromosome with the highest fitness value in the last generation is selected as the final solution [3].

2.2.6 Problem Statement

For a DFG used to model given dependent periodic tasks, the overhead-aware energy optimization problem is defined as follows:

Given a DFG $G = (V, E, \rho, C)$, a timing constraint TC , an MPSoC with M proces-

processor cores, $\{PE_1, PE_2, \dots, PE_M\}$, and each processor core with k discrete voltage levels, $\{V_{dd_1}, V_{dd_2}, \dots, V_{dd_k}\}$, find voltage assignment for each task and a static schedule such that the schedule has the minimum energy consumption within the timing constraint TC , that is, for each task T_i ($T_i \in V$), find its assignment, its release time and its voltage level such that for the obtained static schedule S , the schedule length of S is less than or equal to TC and the total energy consumption of S , $E_{total}(S)$, is minimized.

2.3 Motivational Example

In this section, we motivate the energy optimization problem by showing how to schedule a DFG. We compare energy consumption of the schedules generated by the list scheduling in Landskov et al. [59], the algorithm in Zhang et al. [115], and our technique.

Figure 2.2(a) shows the DFG that is used to model periodic dependent tasks. In the DFG, each node represents a task, and the number beside each node represents the number of clock cycles needed to execute the node. The edge between two nodes represents data dependency, and if two nodes of an edge are assigned to different processor cores in a schedule, intercore communication will occur. For example, In the schedule shown in Figure 2.2(d), there are two intercore communications: $A - C$ and $C - E$.

We assume that there are two processor cores in this MPSoC, and each core has two voltage/frequency levels, the high level and the low level. Based on the dynamic power model in Equation (2.2), $P_{dynamic} = C_{SW} \cdot f_{op} \cdot V_{dd}^2$, without loss of generality, we assume that $C_{SW} = 1nF$; the voltage/frequency pair is $(2V, 1GHz)$ at the high level and $(1V, 0.5GHz)$ at the low level. Therefore, we get $P_H = 4W$, $P_L = 0.5W$, $CP_H = 1ns$, and $CP_L = 2ns$, where P_H and P_L are used to represent the high-level and low-level power consumptions, respectively, and CP_H and CP_L are the high-level and low-level clock periods, respectively. The number of clock cycles of a node is not changed with DVS. Thus, we get the execution time and energy consumption of each node in Figure 2.2(c), where the time unit is μs and the energy unit is μJ . For simplicity, we assume that the transition time overhead is $1\mu s$ for each voltage change, the transition energy overhead from the high to low voltage levels is ap-

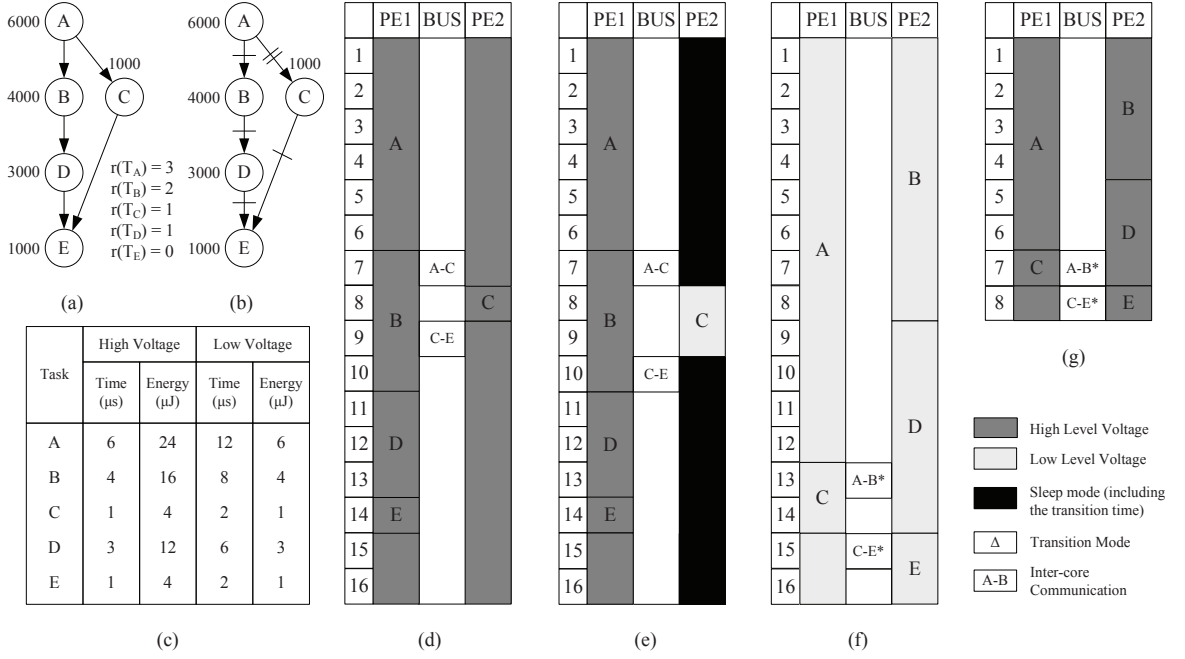


Figure 2.2. A Motivational Example. (a) The original DFG. (b) The retimed DFG using our RDAG algorithm. (c) The task information. (d) The schedule generated by the list scheduling in Landskov et al. [59] without power management (the energy is $137\mu J$). (e) The schedule generated by the DAG-based scheduling algorithm in Zhang et al. [115] with DVS and DPM (the energy is $73.4\mu J$). (f) The schedule generated by our technique (the energy is $25\mu J$). (g) The schedule generated by our technique with a tight timing constraint ($8\mu s$).

proximately $4\mu J$, and one from the low to high voltage levels $0.5\mu J$. The time overhead and energy overhead to enter into and exit from the sleep mode are $5\mu s$ and $2\mu J$, respectively, and the sleep state power is $0.1W$. The read/write communication power through the communication bus is $0.5W$, and the communication time between two tasks is $1\mu s$. The static power is $0.25W$. These assumptions are only for demonstration purpose. Our technique is general enough to deal with general cases, as discussed in later sections.

Assume that the timing constraint is $16\mu s$. The first schedule shown in Figure 2.2(d) is obtained by the traditional list scheduling algorithm in Landskov et al. [59] that focuses on optimizing time performance without power management. In Figure 2.2(d), we can see

that both processor cores operate at the high voltage level for the best time performance. There are some idle slacks in the schedule; however, they cannot be utilized because of the data dependencies. Based on Equation (2.1), we can obtain the total energy of the schedule:

$$E_{total}(S) = E_{t_dynamic}(S) + E_{t_static}(S) + E_{t_comm}(S) = P_H \times 32 + P_{static} \times 16 \times 2 + P_{comm} \times 2 \times 1 = 4 \times 32 + 0.25 \times 32 + 0.5 \times 2 = 137\mu J.$$

The second schedule shown in Figure 2.2(e) is obtained by the DAG-based scheduling algorithm in [115] that applies DPM and DVS to minimize the energy consumption. From the schedule, we can see that on the first core, tasks T_A , T_B , T_D , and T_E are assigned the high voltage level due to the timing constraint. On the second core, task T_C is assigned the low voltage level with DVS, and the idle slacks are turned into the sleep mode with DPM. Based on Equation 2.1, we can obtain the total energy of the schedule: $E_{total}(S) = E_{t_dynamic}(S) + E_{t_static}(S) + E_{t_comm}(S) + E_{t_sleep}(S) + E_{t_sleepOH}(S) = P_H \times 16 + P_L \times 2 + P_{static} \times (16 + 2) + P_{comm} \times 2 \times 1 + P_{sleep} \times (16 - 2 - 5) + E_{sleepOH} \times 1 = 4 \times 16 + 0.5 \times 2 + 0.25 \times 18 + 0.5 \times 2 + 0.1 \times 9 + 2 = 64 + 1 + 4.5 + 1 + 0.9 + 2 = 73.4\mu J.$

The schedule generated by our approach is shown in Figure 2.2(f). In our approach, we first use our RDAG algorithm (shown in Section 2.5.1) to transform all the intra-iteration data dependencies in Figure 2.2(a) into the inter-iteration data dependencies as shown in Figure 2.2(b). This step makes all tasks in one iteration be independent of each other. Next, we use our GeneS scheduling algorithm (shown in Section 2.5.2) to generate a task schedule shown in Figure 2.2(f). Because we adopt coarse-grained software pipelining, the slacks caused by the intra-iteration data dependencies are reclaimed. At the same time, because we can overlap communication and computation, the slacks caused by intercore communication can be reused as well (for the intercore communication in Figure 2.2(f)-(g), the symbol * represents the data dependence to the next period). In the schedule in Figure 2.2(f), as all idle slacks can be fully utilized, all the tasks can be executed at the low voltage level. Based on Equation (2.1), we can obtain the total energy of the schedule: $E_{total}(S) = E_{t_dynamic}(S) + E_{t_static}(S) + E_{t_comm}(S) = P_L \times 32 + P_{static} \times 16 \times 2 + P_{comm} \times 2 \times 1 = 0.5 \times 32 + 0.25 \times 32 + 0.5 \times 2 = 16 + 8 + 1 = 25\mu J.$

If given a tight timing constraint smaller than $14\mu s$, the two DAG-based scheduling algorithms cannot obtain feasible solutions while our approach can. Figure 2.2(g) shows the schedule obtained by our technique with the timing constraint $8\mu s$.

From these results, we can see that our technique can effectively reduce energy consumption. Based on the schedule obtained by our approach, the tasks can be scheduled with the insertion of voltage-setting instructions. The technique can be integrated into compilers or real-time O.S. to generate energy-efficient code. Next, we will present the details of our approach.

2.4 Lower-bound Analysis

In this section, we conduct lower-bound analysis for the energy optimization problem defined in Section 2.2.6. In our two-phase approach as shown in Section 2.5, in the first phase, we transform intra-iteration data dependencies into inter-iteration dependencies so as to make all tasks independent of each other inside a period (Section 2.2.6). With this transformation, the problem defined is changed to: *Given a set of independent tasks, a timing constraint TC , an MPSoC with M cores, $\{PE_1, PE_2, \dots, PE_M\}$, and each core with k discrete voltage levels, $\{V_{dd_1}, V_{dd_2}, \dots, V_{dd_k}\}$, find voltage assignment for each task and a static schedule such that the schedule has the minimum energy consumption within the timing constraint TC .* Next, we study the lower bound of the above problem.

The problem of finding an optimal task schedule with the minimum energy consumption for a single-core or multicore system is known to be NP-complete [30,39,66,90]. For our problem with independent tasks on multicore systems, efficient algorithms to obtain lower bounds do not exist from the previous work. Therefore, we simplify our problem to be a single-core scheduling problem, and use its solution as the approximate lower bound of our problem. The simplified problem is defined as follows: *Given a set of independent tasks, a single-core system with k discrete voltage levels, $\{V_{dd_1}, V_{dd_2}, \dots, V_{dd_k}\}$, a timing constraint $M \cdot TC$, assuming that there is no transition overhead for voltage changes, find voltage assignment for each task and a static schedule such that the schedule has the minimum energy*

consumption within the timing constraint $M \cdot TC$.

In the simplified problem, we put all available time slots from all cores to one core ($M \cdot TC$) and assume an ideal case without transition overhead for voltage changes. In our problem, a task cannot be divided into subtasks and assigned to different cores, and transition overhead associated with voltage changes is inevitable in practice; therefore, the result obtained by an optimal solution for this simplified problem must not be worse (should be better in most cases) than that by an optimal solution of our problem. On the other hand, in our approach, the dependencies among tasks inside one period are removed so tasks have more freedom to be moved around. For tasks assigned on the same processor core, transition overhead can be minimized by grouping tasks with the same voltage level together and scheduling groups following ascending order in terms of voltage levels. So the results obtained by our approach are close to those obtained by optimal solutions for this simplified problem. Therefore, optimal solutions of the simplified problem can serve as the theoretical lower bound of our problem.

In Liu et al. [66], this simplified problem is proved to be NP-complete, and a pseudo-polynomial algorithm based on dynamic programming is proposed to obtain optimal solutions. Although the proposed algorithm is pseudo-polynomial as its complexity is related to the timing constraint, the algorithm is efficient in practice as the execution time of each task is upper bounded by a constant. However, the proposed algorithm in Liu et al. [66] focuses on optimizing energy consumption with DVS only. Next, based on it, by applying both DVS and DPM, we propose a new algorithm called OLB to obtain the lower bound.

In Algorithm 2.4.1, we separate the total available time into two parts, $Time_DVS$ and $Time_DPM$. $Time_DVS$ is the time period managed by DVS to schedule all tasks; $Time_DPM$ is the idle time managed by DPM. Initially, $Time_DVS$ is set as $M \cdot TC$ to represent that all the available time is used for task execution and managed by DVS; $Time_DPM$ is set as zero to represent that there is no idle time to be managed by DPM. For $Time_DVS$, the time period for DVS, Algorithm DPVS in [66] is called to obtain an optimal voltage assignment for all tasks in V with $Time_DVS$ as the timing constraint. Based on

the voltage assignment, we obtain the total execution time of all tasks and compare it with $Time_DVS$. If there is idle slack, we then apply the lowest voltage on it to compute the idle energy as we can always move the idle slack next to the task that are executed with the lowest voltage level.

For $Time_DPM$, the time period for DPM, we attempt to apply DPM to save energy. If $Time_DPM$ is greater than TC , considering that a schedule will be repeatedly executed, it means that one processor core is completely idle; thus, we can turn it off outside the loop so its energy is $TC \cdot P_{sleep}$. We calculate how many idle cores by $Time_DPM/TC$ and put these cores into the sleep mode. Then we attempt to apply DPM on the remaining time ($Time_DPM \% TC$) in $Time_DPM$, and the time period will be put into the sleep mode if we can save more energy by doing that. Finally, we calculate the total energy based on the power model in Section 2.2.3, and record the minimum energy accordingly. At the end of each iteration, $Time_DVS$ is decreased by one, $Time_DPM$ is increased by one, and the above procedure is repeated so all combinations with DVS and DPM can be obtained. The algorithm stops when $Time_DVS$ becomes $\sum_{T_i \in V} \frac{C(T_i)}{f_k}$ that is the minimum time we need to execute all tasks (the processor core is operating at the highest voltage at that time).

As shown in Liu et al. [66], the complexity of Algorithm DPVS is $O(TC \cdot n)$ where TC is the timing constraint, and n is the number of tasks. So the complexity of Algorithm 2.4.1 is $O(TC^2 \cdot n)$. Usually, the execution time of each task is upper bounded by a constant. So TC is equal to $O(n^c)$ (c is a constant). In this case, Algorithm 2.4.1 is polynomial.

2.5 Task Parallelization and Scheduling

In this section, we propose our two-phase approach for task parallelization and energy optimization. Because intra-iteration data dependencies of a DFG not only impede parallelism but also cause abundant idle slacks on processor cores, it plays a negative role on energy minimization. Hence, in the first phase, we propose an algorithm called RDAG to remove intra-iteration data dependencies in Section 2.5.1. Our RDAG algorithm transforms a depen-

Algorithm 2.4.1 Algorithm OLB (Obtain the Lower Bound of the Energy Consumption)

Input: A task set V with n independent tasks, $V = \{T_1, T_2, \dots, T_n\}$, timing constraint $M \cdot TC$, a processor core with k different voltage/frequency levels $\{(V_{dd_1}, f_1), (V_{dd_2}, f_2), \dots, (V_{dd_k}, f_k)\}$ ($V_{dd_1} < V_{dd_2} < \dots < V_{dd_k}$).

Output: E_{LB} , the lower bound of the energy consumption.

```
1:  $E_{LB} \leftarrow \infty$ ;  $Min\_Time \leftarrow \sum_{T_i \in V} \frac{C(T_i)}{f_k}$ .
2: if  $M \cdot TC < Min\_Time$  then
3:   No feasible solution and exit.
4: end if
5:  $Time\_DVS \leftarrow M \cdot TC$ ;  $Time\_DPM \leftarrow 0$ .
6: while  $Time\_DVS \geq Min\_Time$  do
7:   Using  $Time\_DVS$  as the timing constraint, call Algorithm DPVS in Liu et al. [66] to obtain
   an optimal voltage assignment for all tasks in  $V$ , and let  $E_{t\_dynamic}$  be the total dynamic
   energy with the voltage assignment.
8:   For the obtained voltage assignment, let  $V_{dd}(T_i)/f(T_i)$  be the corresponding voltage
   level/frequency of  $T_i$ .  $Total\_Time\_DVS \leftarrow \sum_{T_i \in V} C(T_i)/f(T_i)$ , and  $min\_voltage \leftarrow$ 
    $\min\{V_{dd}(T_i)\}, T_i \in V$ .
9:   if  $(Time\_DVS - Total\_Time\_DVS) > 0$  then
10:     $E_{idle} \leftarrow (Time\_DVS - Total\_Time\_DVS) \cdot P_{dynamic}(min\_voltage)$ .
11:   else
12:     $E_{idle} \leftarrow 0$ .
13:   end if
14:    $E_{DVS} \leftarrow E_{t\_dynamic} + P_{static} \cdot Time\_DVS + E_{idle}$ .
15:    $NumSleepCore \leftarrow Time\_DPM/TC$ ;  $E_{sleepCore} \leftarrow NumSleepCore \cdot TC \cdot P_{sleep}$ .
16:    $Time\_DPM \leftarrow Time\_DPM \% TC$ ;  $E_{DPM} \leftarrow P_{static} \cdot Time\_DPM$ .
17:   if  $Time\_DPM > t_{sleepOH}$  then
18:     if  $(P_{sleep} \cdot (Time\_DPM - t_{sleepOH}) + E_{sleepOH}) < E_{DPM}$  then
19:       Put the core into the sleep mode for the time period  $Time\_DPM$ .
20:        $E_{DPM} \leftarrow P_{sleep} \cdot (Time\_DPM - t_{sleepOH}) + E_{sleepOH}$ .
21:     end if
22:   end if
23:   if  $E_{LB} > (E_{DVS} + E_{DPM} + E_{sleepCore})$  then
24:      $E_{LB} \leftarrow E_{DVS} + E_{DPM} + E_{sleepCore}$ .
25:   end if
26:    $Time\_DVS \leftarrow Time\_DVS - 1$ ;  $Time\_DPM \leftarrow Time\_DPM + 1$ .
27: end while
```

Algorithm 2.5.1 The RDAG Algorithm

Input: A DFG $G = (V, E, \rho, C)$.

Output: The retiming value $r(T_i)$ of each task T_i .

```
1: for each task  $T_i \in V$  do
2:    $r(T_i) \leftarrow 0$ 
3: end for
4: for each  $T_i \in V$  do
5:   if  $T_i$  is a leaf node then
6:      $ENQUEUE(Q, T_i)$ 
7:      $tail \leftarrow T_i$ 
8:   end if
9: end for
10: while  $Q \neq \emptyset$  do
11:    $T_i \leftarrow DEQUEUE(Q)$ 
12:   for each parent node  $T_j$  of  $T_i$  do
13:      $r(T_j) \leftarrow \max\{r(T_j), r(T_i) + 1\}$ 
14:     if  $tail \neq T_j$  then
15:        $ENQUEUE(Q, T_j)$ 
16:        $tail \leftarrow T_j$ 
17:     end if
18:   end for
19: end while
```

dent task graph into a set of independent tasks. We will also analyze the prologue latency and the extra memory overhead caused by the RDAG algorithm. Then in the second phase, we propose a scheduling algorithm called GeneS that adopts a genetic approach to perform energy optimization considering DVS, DPM and various transition overheads in Section 2.5.2.

2.5.1 The RDAG Algorithm for Task Parallelization

(1) *The RDAG Algorithm.* Intra-iteration data dependency in task graphs may impede parallelism and cause idle slacks on processor cores. For example, due to the intra-iteration dependencies of the DFG, in Figure 2.2(d) and Figure 2.2(e), the schedules can utilize at most two processor cores, in other words, there is no more gain with three or more processor

cores. The idle slacks in Figure 2.2(d) and Figure 2.2(e) play a negative role in energy consumption. Hence, if we get rid of intra-iteration data dependencies, we can obtain more design space to reduce idle slacks or achieve better parallelism. In this way, more opportunities are provided for energy optimization. Motivated by this, we propose the RDAG algorithm for task parallelization by transforming a DFG into a new DFG with only inter-iteration data dependencies.

According to the definition of retiming, in order to transform periodic dependent tasks into a set of periodic independent tasks, we need to add at least one delay onto each edge of the original DFG. At the same time, we need to find out the minimum retiming value for each node because a big retiming value may cause a big prologue and epilogue. To achieve this, we use Equation (2.11) to calculate the retiming value of each node as follows:

$$r(T_i) = \begin{cases} \max\{r(T_i), r(T_j) + 1\}, & \text{if } T_i \text{ is } T_j \text{'s parent} \\ 0, & \text{if } T_i \text{ is a leaf node} \end{cases} \quad (2.11)$$

In Equation (2.11), basically, for each leaf node, we set its retiming value as 0; the retiming value of each non-leaf node is calculated from bottom to top. Based on Equation (2.11), we design the RDAG algorithm that is shown in Algorithm 2.5.1.

In the RDAG algorithm, two procedures, *ENQUEUE* and *DEQUEUE*, are used for the INSERT and DELETE operations on a queue, respectively. In Lines 1-3, we assign the initial retiming value of each node to 0. In Lines 4-9, we find out all the leaf nodes and put them into a queue named Q , and store the current tail element of Q into a variable named *tail*. Next, in Lines 10-19, we calculate the retiming value of each node based on Equation (2.11) in a breadth-first manner. Especially, in Lines 14-17, we judge if the parent node T_j of node T_i is the tail element of the current queue. If T_j happens to be the tail element, then we do not need to put node T_j into the queue again. In such a way, we can avoid putting extra adjacent nodes into the queue which will cause unnecessary redundant calculations.

An example is given in Figure 2.3 to show the potential impact that the RDAG algorithm can provide for scheduling. Figure 2.3(a) is used to model a streaming application with periodic dependent tasks and the execution time of each task is listed beside each node.

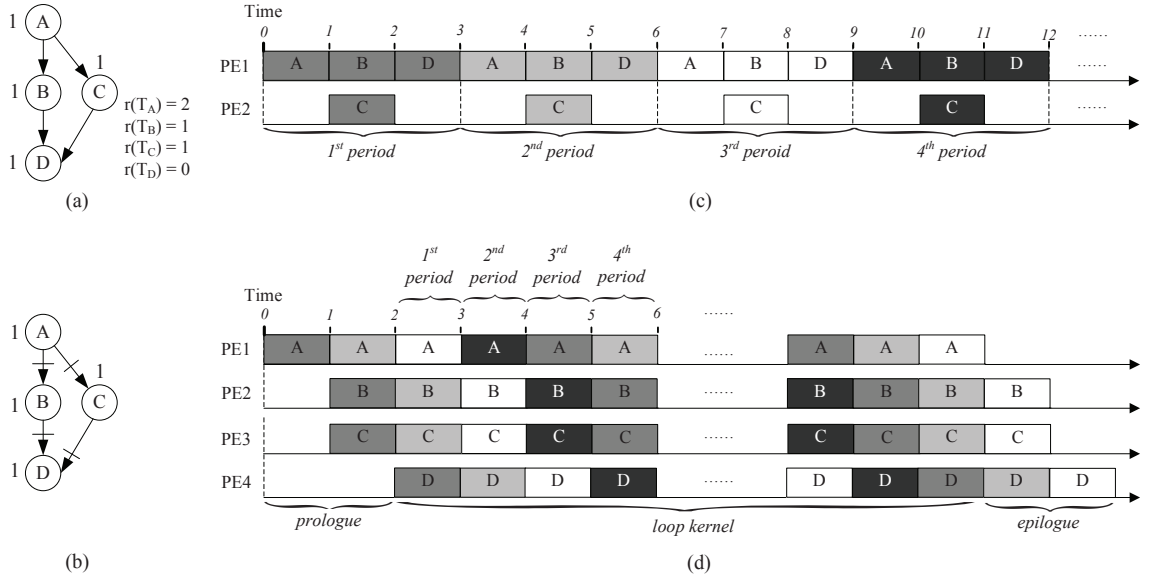


Figure 2.3. An Example of the RDAG Algorithm. (a) The original DFG G . (b) The retimed DFG G_r . (c) The static schedule generated from G . (d) The pipelined schedule generated from G_r .

Based on our RDAG algorithm, we obtain the retiming value of each node, and the corresponding retiming values are listed as $r(T_i)$ for each task T_i . Figure 2.3(b) is the retimed DFG obtained by the RDAG algorithm, in which the count of delays of an edge is represented by the number of bars. In Figure 2.3(b), there is at least one delay on each edge; therefore, all tasks are independent of each other in one iteration. Suppose that we have a four-core MPSoC, according to Figure 2.3(a) and Figure 2.3(b), we generate different schedules as shown in Figure 2.3(c) and Figure 2.3(d), respectively. In Figure 2.3(c), we can see that, due to the inherent data dependency of the application, the schedule can only use 2 cores and the schedule length in each period is 3 time units. After adopting the RDAG algorithm, in Figure 2.3(d), the scheduler can effectively take advantage of 4 processor cores. In Figure 2.3(d), each period of the loop kernel consists of 4 tasks coming from 3 different iterations, and the schedule length in each period in the loop kernel is reduced to 1 time unit.

With the RDAG algorithm, by transforming intra-iteration dependencies into inter-iteration dependencies, we can utilize more processor cores to increase parallelism. How-

ever, as shown in Figure 2.3(d), an extra prologue is added in order to make all tasks independent of each other inside one period. Although the prologue is only executed once, it causes the extra latency in the beginning. Also, we may need more memory to hold data caused by regrouping different periods of tasks into one iteration. Next, we analyze the prologue latency of the RDAG algorithm, and discuss the memory overhead caused by the RDAG algorithm.

(2) *Prologue Latency*. The prologue latency of the RDAG algorithm is caused by rescheduling tasks to previous periods with earlier release time. In the RDAG algorithm, the prologue latency is equal to the time duration of the prologue, and it is determined by the maximum retiming values among all tasks. Given a DFG, let *Prologue_Latency* represent the prologue latency, and it can be calculated by

$$Prologue_Latency = r_{max} \cdot I = \max\{r(T_i)\} \cdot I, (T_i \in V), \quad (2.12)$$

where r_{max} is the maximum retiming value among all tasks in V , and I is the period.

The maximum retiming value obtained by RDAG is determined by the dependency relations of a DFG. Note that the prologue is only executed once so the overhead it introduces is one-time delay. Typical streaming applications such as high-performance multimedia belong to soft real-time applications in which deadline misses are not desirable but allowed. Therefore, as long as the prologue latency is not too large, our technique can be applied to optimize streaming applications on MPSoC. On the other hand, a streaming application is repeatedly executed for many times. After waiting for the execution of the prologue, tasks can be periodically executed in the new loop kernel. As our approach can greatly reduce the schedule length of each period, we can either apply a shorter period or apply DVS and DPM for energy optimization. So we can benefit from each period in the loop kernel after waiting for the execution of the prologue that only executes once.

(3) *Memory Overhead*. As the RDAG algorithm regroupes tasks from different periods into one period, extra memory space is needed to hold data across different periods. In this section, we analyze the memory space required by our method. To make comprehensive

analysis, the memory space needed by both intracore communication and intercore communication is included. Given a DFG that models a streaming application, a retiming function obtained by the RDAG algorithm in Section 2.5.1, and a static schedule obtained in Section 2.5.2, our objective is to obtain the maximum memory space needed by our approach. Based on this, we can analyze the extra energy caused by the extra memory space introduced by our approach and perform comparison with the previous work.

In order to achieve this, we need to analyze all data transfer in a static schedule across the prologue and all periods. However, as a static schedule will be repeatedly executed starting from the first period as shown in Figure 2.3, all the memory space can be obtained by analyzing the data transfer in the prologue and the first period. In a schedule, data transfer is associated with two tasks of an edge of a DFG. For an edge (T_i, T_j) in a DFG, we need a data buffer to hold the data from the time when they are generated by T_i to the time when T_j is finished, and this time period is called *the lifetime segment* of a data buffer. The lifetime segment of a data buffer is represented as $\langle start, end, volume \rangle$ in which “*start*” represents the start time of the segment, “*end*” represents the end time of the segment, and “*volume*” represents the data volume that needs to be transferred through the data buffer. Given a lifetime segment, li , we use $li.start$, $li.end$ and $li.volume$ to represent its start time, end time, and data volume, respectively. To analyze all data transfer in the prologue and the first period, next, we first obtain the lifetime segments of all data buffers associated with all intracore communication and intercore communication, and we then conduct lifetime analysis so as to obtain the total data volume of each time unit by putting all lifetime segments together.

Algorithm Obtain_Lifetime() in Algorithm 2.5.2 is used to collect the lifetime segments of all data buffers needed for data transfer in the prologue and the first period. In Algorithm 2.5.2, we first obtain the maximum retiming value, r_{max} , among all nodes. As the prologue with $r_{max} \cdot I$ (I is the period) is added in the schedule in our method, this should be counted for calculating the abstract release and end times of a task. Given a static schedule S , for a task $T_i \in V$, let R_{T_i} be its release time in S , then its abstract release time in the first period is $r_{max} \cdot I + R_{T_i}$. In the algorithm, we add the lifetime segment of each

Algorithm 2.5.2 Algorithm Obtain_Lifetime()

Input: A DFG $G = (V, E, \rho, C)$, a retiming function r , period I , a static schedule S in which for task T_i , R_{T_i} is its release time and ET_{T_i} is its execution time in S .

Output: A lifetime segment set, $DB_Lifetime_Set$, that contains all lifetime segments of all data buffers in the prologue and the first period.

- 1: Sort all nodes in V in topological ordering.
 - 2: $r_{max} \leftarrow \max\{r(T_i)\}, T_i \in V$.
 - 3: **for** each $T_i \in V$ following the topological order **do**
 - 4: **for** each of T_i 's adjacent node T_j in G **do**
 - 5: $\rho_r(T_i, T_j) \leftarrow r(T_i) - r(T_j)$.
 - 6: **for** $rt=0; rt \leq r(T_i); rt++$ **do**
 - 7: Add li , the lifetime segment of the data buffer that holds the data transferred from T_i to T_j , into $DB_Lifetime_Set$, in which
 - 8: $li.start \leftarrow (r_{max} - rt) \cdot I + R_{T_i} + ET_{T_i}$;
 - 9: $li.end \leftarrow (r_{max} - rt + \rho_r(T_i, T_j)) \cdot I + R_{T_j} + ET_{T_j}$;
 - 10: $li.volume \leftarrow com(T_i, T_j)$.
 - 11: **if** $li.end > (r_{max} + 1) \cdot I$ **then**
 - 12: $li.end \leftarrow (r_{max} + 1) \cdot I$.
 - 13: **end if**
 - 14: **end for**
 - 15: **end for**
 - 16: **end for**
-

data buffer associated with each edge into a set following the topological order. For an edge $(T_i, T_j) \in E$, after retiming, its delay count is $\rho_r(T_i, T_j)$. So in the first period, the lifetime segment of the data buffer associated with (T_i, T_j) should begin with $r_{max} \cdot I + R_{T_i} + ET_{T_i}$ and end with $(r_{max} + \rho_r(T_i, T_j)) \cdot I + R_{T_j} + ET_{T_j}$, and its data volume is $com(T_i, T_j)$. As defined in Section 2.2.4, by retiming a node once, one of its copy is moved into the prologue. So for (T_i, T_j) , after obtaining its lifetime segment in the first period, correspondingly, we add $r(T_i)$ (the retiming value of T_i) lifetime segments with one period time difference into the set for these data buffers in the prologue. For each segment, if its end time is over the first period whose the abstract time is $(r_{max} + 1) \cdot I$, we change it to be $(r_{max} + 1) \cdot I$ as we

only need to calculate up to the first period.

Algorithm *Lifetime_Analysis()* in Algorithm 2.5.3 is to perform lifetime analysis so we can obtain the total data volume of each time unit. The input of Algorithm 2.5.3 is the lifetime segment set, *DB_Lifetime_Set*, that contains all lifetime segments of all data buffers in the prologue and the first period obtained in Algorithm 2.5.2. In Algorithm 2.5.3, in each iteration, we first sort all lifetime segments in *DB_Lifetime_Set* in ascending order in terms of their start times as some new segments may be added into the set. Then following the order, we remove the first two segments from *DB_Lifetime_Set* and compare their start and end times. Basically, when their start times are equal, we combine them together and put the new segments back into *DB_Lifetime_Set*. Otherwise, we output a new lifetime segment that is generated by the start times of the two segments as the data volume of this time period is fixed; then we combine other parts of the two segments and put them back into *DB_Lifetime_Set*. The above procedure is repeated until *DB_Lifetime_Set* is empty or there is only one element in *DB_Lifetime_Set* when we can directly output that element. The output of Algorithm 2.5.3 is a set that contains disjoint lifetime segments and each segment represents the total data volume we need to store in the time period from its start time to its end time. Based on it, therefore, we can find the maximum memory space needed by our method.

Using the DFG and schedule in Figure 2.2 as an example, Figure 2.4(a) shows the given DFG, the retiming values obtained by our RDAG algorithm, and the schedule obtained by our GeneS algorithm in Section 2.5.2. Based on the DFG, retiming function and schedule in Figure 2.4(a), by applying Algorithm 2.5.2, we can obtain *DB_Lifetime_Set*, which is the set that contains all lifetime segments of all data buffers in the prologue and the first period, shown in Figure 2.4(b). Figure 2.4(c) shows *Total_Lifetime_Set*, the output of Algorithm 2.5.3 based on *DB_Lifetime_Set* in Figure 2.4(b). Because of limited space, we only list the first three items of *Total_Lifetime_Set* in Figure 2.4(c). The corresponding schedule with the prologue and the first period is shown in Figure 2.4(d), in which the lifetime segments of all data transfer are provided. From it, we can see that it is not easy to obtain the maximum space needed directly from a schedule.

Algorithm 2.5.3 Algorithm Lifetime_Analysis()

Input: *DB_Lifetime_Set* obtained from Algorithm Obtain_Lifetime().

Output: A lifetime segment set, *Total_Lifetime_Set*, by which we can obtain the total data volume of each time unit in the prologue and the first period.

```
1: while DB_Lifetime_Set is not empty do
2:   if There is only one segment in DB_Lifetime_Set then
3:     Remove the segment from DB_Lifetime_Set, and add it into Total_Lifetime_Set.
4:     Exit.
5:   end if
6:   Sort all lifetime segments in DB_Lifetime_Set in ascending order in terms of start times of
   all lifetime segments.
7:   Remove the first two lifetime segments,  $lt_1$  and  $lt_2$ , from DB_Lifetime_Set.
8:   if  $lt_1.start = lt_2.start$  then
9:     if  $lt_1.end = lt_2.end$  then
10:      Add  $\langle lt_1.start, lt_1.end, lt_1.volume + lt_2.volume \rangle$  into DB_Lifetime_Set.
11:    else
12:      if  $lt_1.end > lt_2.end$  then
13:        Add two lifetime segments:  $\langle lt_1.start, lt_2.end, lt_1.volume + lt_2.volume \rangle$  and
         $\langle lt_2.end, lt_1.end, lt_1.volume \rangle$ , into DB_Lifetime_Set.
14:      else
15:        Add two lifetime segments:  $\langle lt_1.start, lt_1.end, lt_1.volume + lt_2.volume \rangle$  and
         $\langle lt_1.end, lt_2.end, lt_2.volume \rangle$ , into DB_Lifetime_Set.
16:      end if
17:    end if
18:    else
19:      if  $lt_1.end \leq lt_2.start$  then
20:        Add  $\langle lt_1.start, lt_1.end, lt_1.volume \rangle$  into Total_Lifetime_Set.
21:        Add  $\langle lt_2.start, lt_2.end, lt_2.volume \rangle$  into DB_Lifetime_Set.
22:      else
23:        Add  $\langle lt_1.start, lt_2.start, lt_1.volume \rangle$  into Total_Lifetime_Set.
24:        if  $lt_1.end = lt_2.end$  then
25:          Add  $\langle lt_2.start, lt_1.end, lt_1.volume + lt_2.volume \rangle$  into DB_Lifetime_Set.
26:        else
27:          if  $lt_1.end > lt_2.end$  then
28:            Add two lifetime segments:  $\langle lt_2.start, lt_2.end, lt_1.volume + lt_2.volume \rangle$  and
             $\langle lt_2.end, lt_1.end, lt_1.volume \rangle$ , into DB_Lifetime_Set.
29:          else
30:            Add two lifetime segments:  $\langle lt_2.start, lt_1.end, lt_1.volume + lt_2.volume \rangle$  and
             $\langle lt_1.end, lt_2.end, lt_2.volume \rangle$ , into DB_Lifetime_Set.
31:          end if
32:        end if
33:      end if
34:    end if
35: end while
```

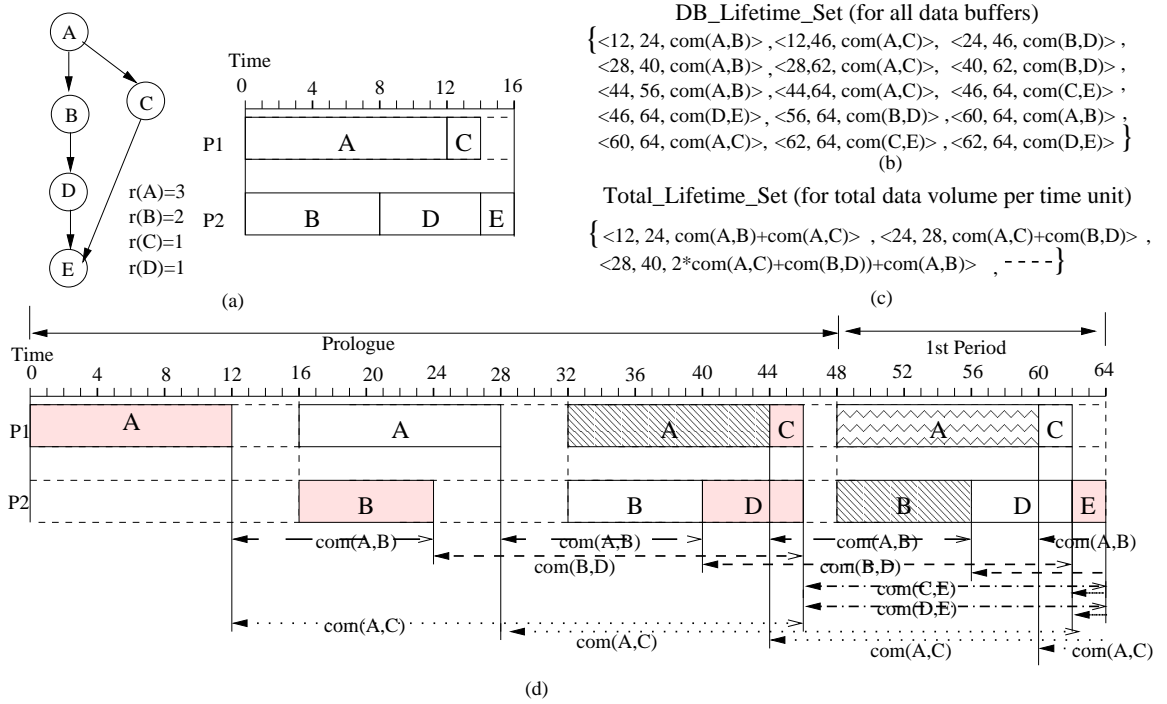


Figure 2.4. Latency and Memory Overhead of the RDAG Algorithm.

The complexity of Algorithm 2.5.2 is $O(|E|)$ in which E is the total number of edges in a DFG, as we need to traverse each edge in order to obtain the data transfer associated with it. So for $DB_Lifetime_Set$, the output of Algorithm 2.5.2, its total segment number is bounded by $O(|E|)$. In Algorithm 2.5.3, we try to find all disjoint sets of all the lifetime segments in $DB_Lifetime_Set$ in terms of their start and end times, and the maximum number of all the disjoint sets is $2 \cdot |DB_Lifetime_Set|$. Therefore, the complexity of Algorithm 2.5.3 is $O(|E|)$.

2.5.2 The GeneS Algorithm for Energy Optimization

In this section, we propose our genetic scheduling algorithm, GeneS, to perform energy optimization with DVS and DPM. GeneS can search and find the best schedule within the solution space generated by gene evolution. Next, we first introduce three key components of our GeneS algorithm, chromosome representation, crossover and mutation (the two basic

genetic operators), and the fitness function. We then present our GeneS algorithm.

(1) *Chromosome and Schedule*. In our approach, each chromosome, ξ_i , consists of two lists of genes, Π and Ω . The content in list Π represents the task assignment of each task, and the content in list Ω represents the voltage level of each task. For each task, its task assignment and voltage selection form a pair of genes. Given a chromosome, we generate a schedule with energy optimization as follows.

- Step 1. *Construct task groups*. Following the genes of tasks, put the tasks that are assigned into the same processor core and have the same voltage level into the same group.
- Step 2. *Generate a schedule with DPM*. For each processor core, if there are some task groups assigned on it, sort the groups in ascending order in terms of their voltage levels and schedule them following the order. If there are idle slacks after all the task groups have been scheduled on a core, move all the idle slacks to the location that is immediately next to the first task group (with the lowest voltage level in the core). Put the idle slacks into the sleep mode if we can save energy by doing it. For a processor core, if there is no any task scheduled on it, put it into the sleep mode.

As just shown, when generating a schedule based on a chromosome, we first group all tasks based on their voltage levels on the same core and then schedule the groups following ascending order in terms of the voltage levels. In this way, we can minimize the total energy/time transition overhead on the core based on Equations (2.6), (2.7), (2.8), and (2.9). We also move all the idle slacks next to the group with the lowest voltage level and attempt to put them into the sleep mode if we can save energy. For a processor core that is completely idle, as we can turned it off outside the loop so its power is P_{sleep} .

Figure 2.5 shows a chromosome and its corresponding task schedule. In Figure 2.5(a), from each pair of genes of a task, we can obtain its task assignment and voltage selection. For example, task T_B is assigned to processor PE_1 with the voltage level V_{dd_2} . Therefore, PE_1 and V_{dd_2} forms a pair of genes for task T_B . Given the chromosome in Figure 2.5(a), the

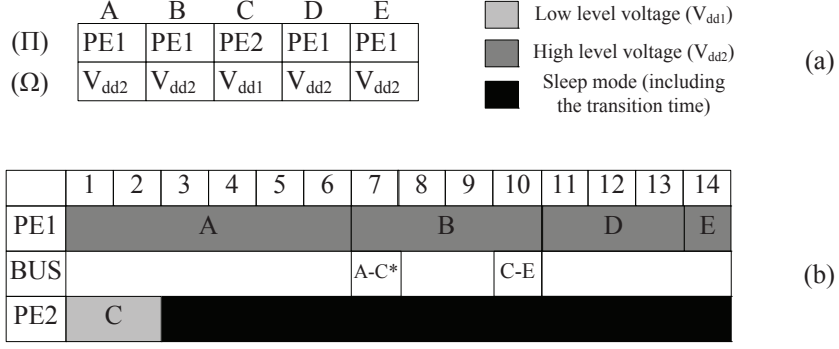


Figure 2.5. Chromosome Representation and Its Corresponding Task Schedule.

task schedule can be generated as shown in Figure 2.5(b). For the tasks that map to processor core PE_1 , their voltage levels are the same (V_{dd2}) so there is only one task group. Similarly we can obtain the schedule on PE_2 in which we put the idle slacks into the sleep mode with DPM.

(2) *Crossover and Mutation.* We use two genetic operators, *crossover* and *mutation*, to create new generations of chromosomes. The crossover operator selects genes from parent chromosomes and creates new pairs of offspring. For each pair of chromosomes, we randomly select the crossover point of two chromosomes to swap their genes, and create a new pair of chromosomes. Figure 2.6 shows an example of the crossover operator.

In this example, we perform crossover to create chromosome 3 and chromosome 4 from chromosome 1 and chromosome 2. We use a vertical bar to represent the crossover point. To generate a new pair of offspring, the genes before crossover point (the genes of tasks T_A and T_B) of chromosome 1 and the genes after the crossover point (the genes of tasks T_C , T_D and T_E) of chromosome 2 form chromosome 3. Similarly, the genes of tasks T_A and T_B of chromosome 2 and the genes of tasks T_C , T_D and T_E of chromosome 1 form the chromosome 4.

The mutation operator is used to maintain the genetic diversity from one generation to another. In our approach, for each chromosome, we perform mutation by randomly selecting one task and decreasing its voltage for one voltage level. Figure 2.7 shows an example of

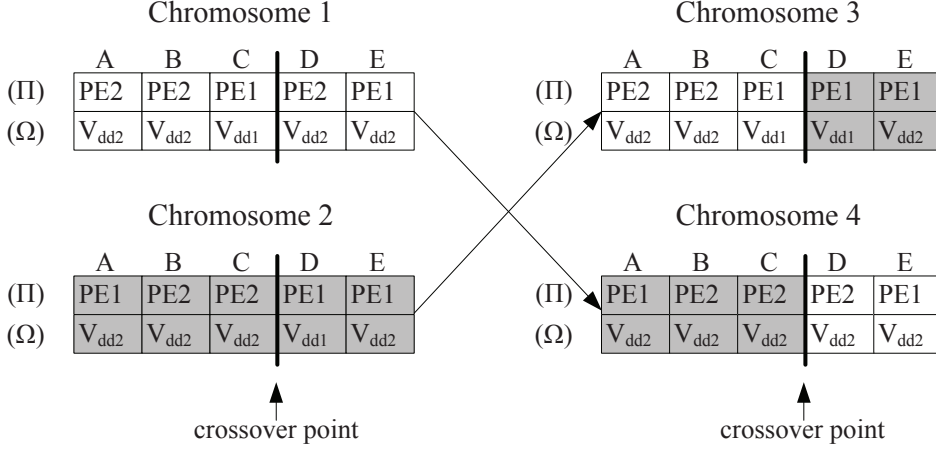


Figure 2.6. The Crossover Operator Generates New Chromosomes: Chromosomes 3 and Chromosome 4.

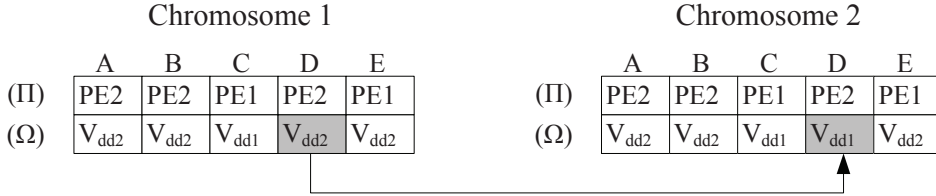


Figure 2.7. The Mutation Operator. Task T_D is selected to perform mutation, and its voltage level is changed from V_{dd_2} to V_{dd_1} .

the mutation operator. In this example, after mutation, the voltage level of task T_D changes from V_{dd_2} to V_{dd_1} .

(3) *Fitness Function*. The fitness function is designed to evaluate each chromosome in order to find a schedule with the minimum energy within the solution space generated by gene evolution. Given the timing constraint TC , a chromosome ξ_i whose schedule length is $L(\xi_i)$ and whose energy consumption is $E_{total}(\xi_i)$ that is calculated based on the schedule associated with it, the fitness value, $fitness(\xi_i)$, is defined by

$$fitness(\xi_i) = \begin{cases} \frac{1}{E_{total}(\xi_i)}, & TC \geq L(\xi_i) \\ 0, & TC < L(\xi_i). \end{cases} \quad (2.13)$$

In the fitness function, both the timing constraint TC and the energy consumption $E_{total}(\xi_i)$ are taken into account. We compare the timing constraint TC with the schedule

Algorithm 2.5.4 The GeneS Algorithm

Input: A set of independent tasks, the timing constraint TC , M processor cores, and each processor core with k different voltage levels $\{V_{dd_1}, V_{dd_2}, \dots, V_{dd_k}\}$.

Output: An objective task schedule with the minimum energy consumption.

- 1: Generate the initial generation with N_C chromosomes. In each chromosome, the voltage level of each task is set as the highest voltage V_{dd_k} , and the task assignment is randomly selected.
 - 2: Calculate the fitness value of each chromosome based on Equation 2.13.
 - 3: Sort chromosomes in the ascending order of the fitness value.
 - 4: Remove $\frac{1}{2}N_C$ chromosomes whose fitness values are smaller.
 - 5: Perform crossover on the preserved $\frac{1}{2}N_C$ chromosomes to create another $\frac{1}{2}N_C$ chromosomes.
 - 6: Calculate the fitness value of each newly generated chromosome based on Equation 2.13.
 - 7: Sort the preserved $\frac{1}{2}N_C$ chromosomes and the newly generated $\frac{1}{2}N_C$ chromosomes in ascending order in terms of the fitness values, and remove $\frac{1}{4}N_C$ chromosomes whose fitness values are smaller.
 - 8: Randomly select $\frac{1}{4}N_C$ chromosomes from the preserved $\frac{3}{4}N_C$ chromosomes to perform mutation and generate $\frac{1}{4}N_C$ chromosomes.
 - 9: Let the current N_C chromosomes be chromosomes in the new generation.
 - 10: **if** the termination condition is satisfied **then**
 - 11: Let the chromosome with the biggest fitness value be the best solution.
 - 12: **else**
 - 13: Go to Step 4.
 - 14: **end if**
-

length $L(\xi_i)$ of the chromosome ξ_i . If the schedule length $L(\xi_i)$ is greater than the timing constraint TC , the fitness value of the chromosome ξ_i , $fitness(\xi_i)$, is equal to zero since the schedule is not feasible; otherwise, its fitness value increases when its total energy consumption $E_{total}(\xi_i)$ decreases. Using this fitness function, we can select the chromosome with the highest fitness value to be the solution of our energy minimization problem.

(4) *The GeneS Algorithm.* In this section, we present our genetic algorithm, GeneS, to generate the objective task schedule with the minimum energy consumption. Our GeneS algorithm is shown in Algorithm 2.5.4.

In GeneS, we start from an initial population that is randomly generated by a number

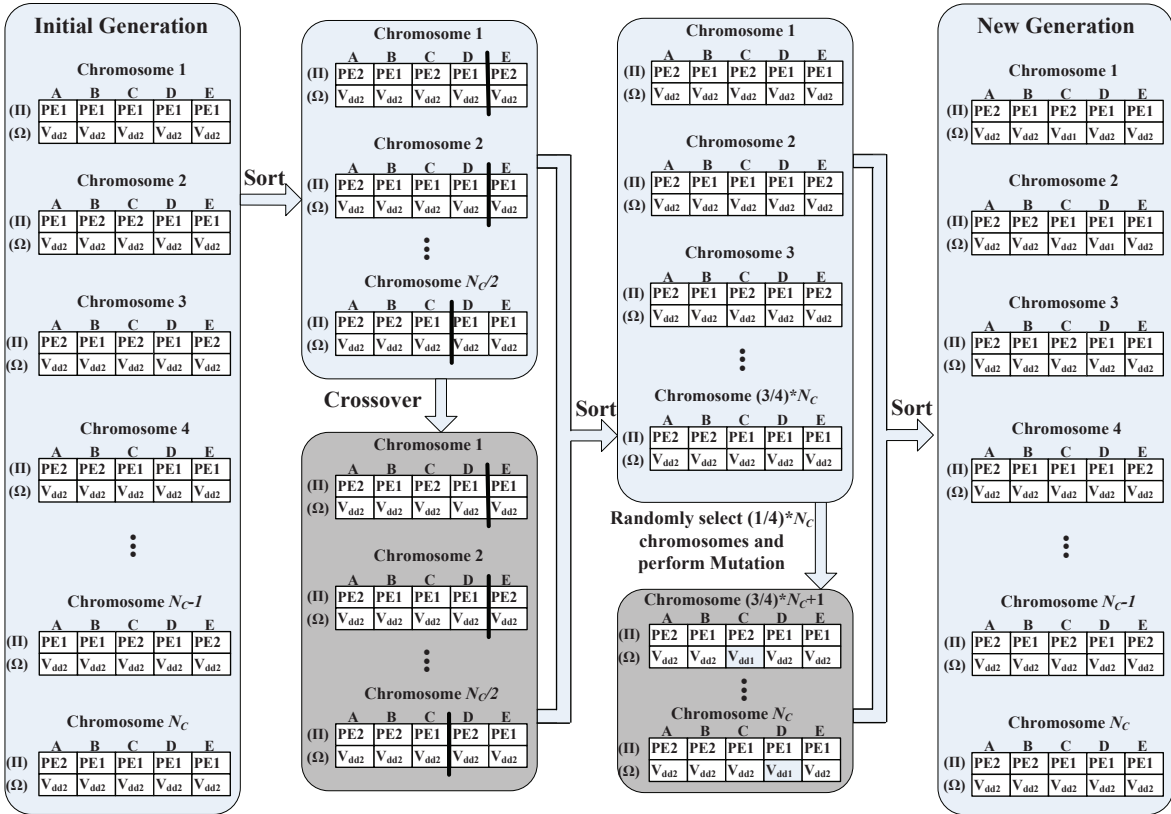


Figure 2.8. An Example of the GeneS Algorithm.

of chromosomes. The number of chromosomes in each generation is denoted by N_C . For each chromosome ξ_i in the initial generation, the task assignments are randomly selected, and the voltage level of each task is assigned with the highest voltage level, V_{dd_k} .

Starting from the initial population, we iteratively perform crossover and mutation operators over the chromosomes to create new generations. Based on the fitness function, we calculate the fitness value of each chromosome. The average fitness value of each generation will be increased as we only keep the chromosomes with higher fitness value through evolution. The algorithm terminates when the predefined maximum number of generations is reached.

We give an example in Figure 2.8 to illustrate our GeneS algorithm. Given the DFG in Figure 2.2(a), the power model in Figure 2.2(c), two processor core $\{PE_1, PE_2\}$, and the timing constraint $TC = 16$, after applying our RDAG algorithm on the DFG, we can obtain a

set of independent tasks. As they are independent of each other, they can be scheduled in any orders in one period. Following GeneS, we first generate the initial generation with N_C chromosomes, in which all tasks in each chromosome are assigned with the highest voltage level V_{dd2} . We can calculate the fitness value of each chromosome, sort them and remove $\frac{1}{2}N_C$ chromosomes that have the smaller fitness values. The preserved $\frac{1}{2}N_C$ chromosomes form $\frac{1}{4}N_C$ pairs of chromosomes. For each pair of chromosomes, we perform crossover and generate two new chromosomes. For the preserved $\frac{1}{2}N_C$ chromosomes and the newly generated $\frac{1}{2}N_C$ chromosomes, we sort them by the fitness values and remove the $\frac{1}{4}N_C$ chromosomes with the smaller fitness values. From the current $\frac{3}{4}N_C$ chromosomes, we randomly select $\frac{1}{4}N_C$ chromosomes and perform mutation operation. Then we sort the $\frac{3}{4}N_C$ chromosomes and the $\frac{1}{4}N_C$ chromosomes newly generated by the fitness values. These N_C chromosomes form the new generation for gene evolution.

The complexity of our GeneS algorithm is governed by sorting chromosomes and constructing schedules from chromosomes. Let N_G be the number of generations in the GeneS algorithm, let N_C be the number of chromosomes in each generation, and let n be the number of tasks. To generate a new generation, the GeneS algorithm will sort the N_C chromosomes two times, so the corresponding complexity is $O(N_C \log N_C)$. To construct a schedule from a chromosome, in which we can obtain its energy consumption at the same time, we need to group tasks based on their voltage level, sort task groups based on their voltage levels and then schedule task groups. It takes $O(n)$ to group each task, takes at most $O(n \log n)$ to sort all groups in one core, and takes $O(n)$ to perform scheduling. Therefore, the complexity is $O(n \log n)$ for constructing a schedule from one chromosome. So for each generation, it takes $O(N_C \cdot n \log n)$ to construct schedules from chromosomes as we totally need to generate $1\frac{3}{4}N_C$ schedules (the times to calculate the fitness values). Thus, the complexity of our GeneS algorithm is $O(N_G \cdot N_C \log N_C + N_G \cdot N_C \cdot n \log n)$.

Voltage	Freq.	Power	Voltage	Freq.	Power
$V_{dd}(V)$	$f_{op}(MHz)$	$P(W)$	$V_{dd}(V)$	$f_{op}(MHz)$	$P(W)$
1.2	500	9.2	1.25	600	12.0
1.3	700	15.1	1.35	800	18.6
1.4	1000	25.0			

Table 2.1. The Voltage Levels, Frequencies and Power Consumption based on the Power Model of the Mobile Athlon4 Processor [7].

2.6 Experiments

In this section, we evaluate and compare our approach with the PEDF algorithm [115] and the SpringS algorithm [65] in terms of three performance metrics: (1) the energy consumption, (2) the minimum valid timing constraint, and (3) the extent of parallelism. We will also present the results of prologue latency and memory overhead of our approach.

2.6.1 Experimental Setup

(1) *Power Model.* The experiments are conducted based on the power model of the AMD Mobile Athlon4 DVS processor [7]. The AMD Mobile Athlon4 processor can operate at various voltage levels in the range of 1.2 – 1.4V with 50mV steps, and the corresponding frequencies vary from 500MHz to 1GHz with 100MHz steps [76]. The power is calculated by $P_{dynamic} = C_{SW} \cdot f_{op} \cdot V_{dd}^2$ [91], where C_{SW} is 12.75nF from the data sheet of the AMD Mobile Athlon4 processor [7]. The five voltage levels, and their corresponding frequencies and power consumption are shown in Table 2.1.

The time overhead during a voltage transition among five voltage levels is calculated based on Equation 2.6, $t_{TRAN} = \frac{2 \cdot C_{DD}}{I_{MAX}} \cdot |V_{dd_j} - V_{dd_i}|$, in which C_{DD} and I_{MAX} are set as 12pF and 16mA [7]. The energy transition overhead is calculated based on Equations (2.8) and (2.9). For the energy consumed by the converter, $E_{TRAN-DC} = \alpha \cdot C_{DD} \cdot |V_{dd_i}^2 - V_{dd_j}^2|$, α is set as 0.9 [17].

AMD Mobile Athlon4 DVS processors have low power sleep states that can be uti-

lized when systems are idle. The power consumed in the sleep state is $2.4W$ [76]. For the transition involving sleep states, considering the synchronization delay with off-chip components such as memory, the transition time is quite large. It takes at least $5ms$ to synchronize with the main memory entering in or exiting from the sleep state [7]. The power consumption for transition overhead associated with one transition for entering into and exiting from the sleep mode is assumed to be the power consumption at the voltage level entering from the sleep mode. The power of the bus is assumed to be $147mW$. For static energy consumption, $P_{static} = I_{subn} \cdot V_{dd} + |V_{bs}| \cdot I_j$, the subthreshold current I_{subn} is set as $250\mu A$ [7], the body bias voltage V_{bs} is set as $0.4V$, and the reverse bias junction current I_j is set as $4.8 \times 10^{-10} A$ [72].

(2) *Benchmarks*. We conduct experiments on 12 benchmarks as shown in Table 2.2. Among them, the first 9 benchmarks are obtained from Embedded Systems Synthesis Benchmarks (E3S) [104]. E3S is largely based on data from the Embedded Microprocessor Benchmark Consortium (EEMBC). Consumer-1 and consumer-2 are embedded consumer electronic applications including tasks like JPEG compression, JPEG decompression, high pass gray-scale filter, RGB to CYMK conversion and RGB to YIQ conversion, etc. Auto-industry-1, auto-industry-2 and auto-industry-3 come from embedded auto-industry applications including major tasks like FFT (Fast Fourier Transform), finite/infinite impulse response filter, IDCT (Inverse Discrete Cosine Transform), IFFT (Inverse Fast Fourier Transform), matrix arithmetic, road speed calculation, table lookup and interpolation, etc. telecom-1 and telecom-2 represent embedded telecom applications consisting of tasks like autocorrelation-data (pulse, sine, and speech), convolution encoder-data (xk5r2dt, xk4r2dt, and xk3r2dt), viterbi GSM decoder-data, etc. Office-1 describes an embedded office application which consists of tasks like dithering, image rotation and text processing. Network-1 is an embedded network application including tasks like OSPF/Dijkstra, route lookup/patricia, and packet flow, etc.

Besides the 9 benchmarks, we use TGFF [28] to generate 3 periodic task graphs, TGFF-1, TGFF-2 and TGFF-3. Among them, TGFF-1 is a *slim* graph while TGFF-2 is a *fat* graph. Basically, in TGFF-1, it has very long critical path length and there are not many independent nodes; in TGFF-2, its critical path is relatively short and there are many

Benchmarks	No. of tasks	No. of cycles	Execution time of critical path (μs)
consumer-1	7	11050	8
consumer-2	5	16520	10
auto-industry-1	6	13607	9
auto-industry-2	4	351120	348
auto-industry-3	9	1397567	1392
telecom-1	4	53900	51
telecom-2	6	438900	395
office-1	5	3311000	2584
network-1	4	264127	263
TGFF-1	6	90000	83
TGFF-2	8	170000	58
TGFF-3	24	924000	563

Table 2.2. Benchmark Descriptions and Characteristics.

independent nodes. Table 2.2 illustrates the detailed information of each benchmark. In Table 2.2, “No. of tasks” represents the number of tasks in each benchmark; “No. of cycles” represents the total number of clock cycles in each benchmark; “Execution time of critical path” represents the total execution time of tasks along the critical path, and each task is with the highest voltage/frequency level (V_{dd_k}, f_k).

2.6.2 Results and Discussion

In this section, we evaluate the energy consumption of *one iteration* of each benchmark under different timing constraints and different number of processor cores using three algorithms, GeneS, SpringS [65], and PEDF [115].

GeneS is our scheduling algorithm that uses genetic approach to solve energy minimization problem for *periodic dependent tasks* on multicore architecture. In the experiments, the maximum number of generations is set as 5000, and the number of chromosomes in each

generation is set as 64. SpringS is a scheduling algorithm proposed in Liu et al. [65]. PEDF is a DVS scheduling algorithm used to solve the energy minimization problem for dependent tasks on multiple variable voltage processors. Instead of fixing the task assignment like Gruian and Kuchcinski [36] or task scheduling like Luo and Jha [69], a two-phase framework has been proposed in the PEDF algorithm that integrates task assignment, ordering and voltage selection together to iteratively generate a schedule. So the PEDF algorithm performs better in energy consumption compared with the list scheduling algorithm and the algorithms in Gruian and Kuchcinski [36] and Luo and Jha [69]. Therefore, PEDF is selected for comparison in the chapter.

(1) *Energy Consumption.* Table 2.3 and Table 2.4 show the experimental results for all the 12 benchmarks running on 2, 3, 4, 6, and 8 processor cores. In Table 2.3 and Table 2.4, column “TC Range (μs)” represents timing constraints we used that start from the minimum execution time and increase by $2\mu s$ each step. Columns “PEDF (μJ),” “SpringS (μJ),” and “GeneS (μJ)” represent the energy consumption obtained by corresponding algorithms. Column “PEDF over GeneS(%)” represents the percentage of how much extra energy is saved by GeneS compared to PEDF. “-” means for a timing constraint, PEDF or SpringS cannot find a solution. Note that for each benchmark, experimental results are separated into two parts based on different ranges of timing constraints. We call the timing constraint at the partition point *the critical timing constraint*. On the left part, the timing constraint is smaller than the critical timing constraint, and the PEDF algorithm has no solution under these small timing constraints. On the contrary, on the right part, the timing constraints are bigger than the critical timing constraint, and PEDF can find feasible schedules. In the experiments, we test each benchmark with different timing constraints. Starting from the minimum timing constraint to perform task scheduling, we gradually increase the timing constraint by $2\mu s$ each step. The experiment results list the average energy consumption of each benchmark for different ranges of timing constraints.

In Table 2.3 and Table 2.4, the results show that with tight timing constraints, the PEDF algorithm cannot achieve a feasible solution while ours can. By extending the timing constraint, PEDF may obtain feasible solutions. On average, GeneS achieves a 24.4% reduc-

Benchmark	TC Range (μs)	PEDF (μJ)	Springs (μJ)	GeneS (μJ)	TC Range (μs)	PEDF (μJ)	Springs (μJ)	GeneS (μJ)	PEDF over GeneS (%)
2-core									
consumer-1	8–13	–	249	197	14–19	362	264	264	27.1
consumer-2	10–15	–	391	330	16–22	471	376	311	34.0
auto-1	7–13	–	296	235	14–26	367	315	315	14.2
auto-2	335–348	–	11739	8596	349–705	7828	6476	6449	17.6
auto-3	830–1392	–	31622	25326	1393–2792	30716	30437	24136	21.4
telecomm-1	41–51	–	1600	1176	52–110	1473	1457	1261	14.4
telecomm-2	330–395	–	12445	9107	396–900	13590	12910	11087	18.4
office-1	1925–2584	–	80913	79695	2585–5725	79710	74893	74893	6.0
network-1	253–263	–	8849	6484	264–606	9366	9211	8422	10.1
TGFF-1	48–65	–	2142	2113	66–110	2627	1875	1833	30.2
TGFF-2	87–97	–	–	4183	98–180	3671	3607	3288	10.4
TGFF-3	464–591	–	22263	22263	592–1000	21513	19216	19216	10.7
average									17.9
3-core									
consumer-1	5–13	–	271	269	14–19	491	413	413	15.9
consumer-2	6–11	–	421	375	12–22	708	452	452	36.2
auto-1	5–13	–	340	316	14–26	542	469	469	13.5
auto-2	335–348	–	14880	8758	349–705	15229	15205	13695	10.1
auto-3	550–1392	–	32907	30489	1393–2792	30850	30502	30502	1.1
telecomm-1	41–51	–	2022	1176	52–110	2040	1934	1643	19.5
telecomm-2	330–395	–	15779	9108	396–900	18302	17768	14124	22.8
office-1	1925–2584	–	89819	71274	2585–5725	121537	101309	77556	36.2
network-1	253–263	–	11223	6484	264–606	14020	12752	10327	26.3
TGFF-1	33–65	–	2017	1992	66–110	4005	2429	2429	39.4
TGFF-2	57–61	–	4236	4236	62–180	4124	3923	2778	32.6
TGFF-3	309–504	–	21085	21085	505–700	23329	18046	18046	22.6
average									23.0
4-core									
consumer-1	5–13	–	344	319	14–19	666	565	565	15.2
consumer-2	5–10	–	441	375	11–22	998	590	590	40.9
auto-1	5–13	–	422	376	14–26	542	515	515	5.0
auto-2	335–348	–	18021	8783	349–705	22722	21596	16781	26.1
auto-3	696–1392	–	32591	29799	1393–2792	30970	26014	26014	16.0
telecomm-1	41–51	–	2444	1176	52–110	3083	2742	2024	34.3
telecomm-2	330–395	–	19114	9108	396–900	24374	22625	17160	29.6
office-1	1925–2584	–	110561	78493	2585–5725	167057	130911	95268	43.0
network-1	253–263	–	11223	6484	264–606	14020	12752	10327	26.3
TGFF-1	33–65	–	2173	1966	66–110	5198	3238	3238	37.7
TGFF-2	44–59	–	4199	4172	60–180	5181	4625	3702	28.5
TGFF-3	233–465	–	20362	14862	466–700	23899	17655	17655	26.1
average									27.4

Table 2.3. The Energy of Each Benchmark under Various Timing Constraints on 2, 3, 4 Processor Cores.

Benchmark	TC Range (μs)	PEDF (μJ)	SpringS (μJ)	GeneS (μJ)	TC Range (μs)	PEDF (μJ)	SpringS (μJ)	GeneS (μJ)	PEDF over GeneS (%)
6-core									
consumer-1	5–13	–	489	419	14–19	1063	867	867	18.4
consumer-2	5–10	–	509	375	11–22	1186	742	742	37.4
auto-1	5–13	–	587	496	14–26	827	771	771	6.8
auto-2	335–348	–	18021	8783	349–705	22722	21596	16781	26.1
auto-3	696–1392	–	49354	47501	1393–2792	42449	40632	40632	4.3
telecomm-1	41–51	–	2444	1176	52–110	3083	2742	2024	34.3
telecomm-2	330–395	–	25783	9108	396–900	36517	32339	23232	36.4
office-1	1925–2584	–	131302	78493	2585–5725	225593	160513	112979	49.9
network-1	253–263	–	11223	6484	264–606	14020	12752	10327	26.3
TGFF-1	33–65	–	2173	1966	66–110	5198	3238	3238	37.7
TGFF-2	44–59	–	4199	4172	60–180	5181	4625	3702	28.5
TGFF-3	161–321	–	20429	13256	322–600	26214	22168	22168	15.4
average									26.8
8-core									
consumer-1	5–13	–	561	469	14–19	1201	1018	1018	15.2
consumer-2	5–10	–	509	375	11–22	1186	742	742	37.4
auto-1	5–13	–	587	496	14–26	827	771	771	6.8
auto-2	335–348	–	18021	8783	349–705	22722	21596	16781	26.1
auto-3	696–1392	–	60890	57622	1393–2792	77374	67009	67009	13.4
telecomm-1	41–51	–	2444	1176	52–110	3083	2742	2024	34.3
telecomm-2	330–395	–	25783	9108	396–900	36517	32339	23232	36.4
office-1	1925–2584	–	131302	78493	2585–5725	225593	160513	112979	49.9
network-1	253–263	–	11223	6484	264–606	14020	12752	10327	26.3
TGFF-1	33–65	–	2173	1966	66–110	5198	3238	3238	37.7
TGFF-2	44–59	–	4199	4172	60–180	5181	4625	3702	28.5
TGFF-3	122–243	–	20597	12785	244–550	29413	26373	26373	10.3
average									26.9

Table 2.4. The Energy of Each Benchmark under Various Timing Constraints on 6 and 8 Processor Cores.

tion in energy consumption compared with PEDF. For two groups of timing constraints, our algorithm can save extra energy consumption compared with SpringS. From these experimental results, we can see that (1) our scheduling algorithm does not have a strict requirement on timing constraint and can be applied for those embedded systems with tight timing constraint, and (2) our scheduling algorithm can perform better compared with PEDF and SpringS for different number of processor cores.

In order to evaluate our proposed approach with larger number of processor cores, we also present the experiment results on 6 and 8 processor cores. In Table 2.4, we can see that our approach can also achieve significant energy reduction compared to the PEDF algorithm and the SpringS algorithm. For the MPSoC with 6 and 8 processor cores, our GeneS algorithm can save average 26.8% and 26.9% of energy consumption, respectively, compared to the PEDF algorithm.

At this point, we have analyzed the data in one table horizontally. Now we fix the benchmark and analyze the data in the table vertically. This analysis helps us identify whether or not the scheduling algorithm can effectively take advantage of the potential computation power of multiple processor cores. For simplicity, the last benchmark, TGFF-3, is used as an example. The experimental results show that when the number of processor cores is increased from 2 to 8, the minimum valid timing constraint for GeneS (by which we can obtain a feasible solution) is reduced from $464\mu s$ to $122\mu s$ while it is not changed for PEDF. This shows that our GeneS algorithm can effectively exploit the potential of multicore architectures to minimize the energy consumption.

In Section 2.4, we have analyzed the lower bound energy consumption E_{LB} . As we mentioned in Section 2.4, E_{LB} is the lower bound energy consumption that is close to the optimal solution, and an optimal solution for multicore processors may not achieve it. Here we compare the energy consumption obtained by the GeneS algorithm and the lower bound energy consumption E_{LB} . Table 2.5 shows the experimental results. From the results, we can see that, for all 12 benchmarks, the GeneS algorithm can generate the task schedule with energy consumption close to the lower bound energy consumption E_{LB} . For benchmark

Benchmark	TC Range (μs)	2-core		4-core		6-core		8-core	
		GeneS (μJ)	E_{LB} (μJ)	GeneS (μJ)	E_{LB} (μJ)	GeneS (μJ)	E_{LB} (μJ)	GeneS (μJ)	E_{LB} (μJ)
consumer-1	5–19	232	216	414	381	592	523	680	666
consumer-2	5–22	319	309	459	440	627	590	627	590
auto-1	5–26	290	257	429	401	643	610	643	610
auto-2	335–705	7049	6499	14660	14660	14660	14660	14660	14660
auto-3	696–2792	24659	20017	27374	24518	42450	39448	62374	58963
telecomm-1	41–110	1189	1125	1640	1122	1640	1122	1640	1122
telecomm-2	330–900	10267	9842	15625	13416	17241	14890	17241	14890
office-1	1925–5725	77290	67835	85901	72794	100241	79493	100241	79493
network-1	253–606	7705	6546	8850	7308	8850	7308	8850	7308
TGFF-1	33–110	1909	1717	2710	1935	2710	1935	2710	1935
TGFF-2	44–180	3381	3135	3756	3534	3756	3534	3756	3534
TGFF-3	122–1000	20220	18147	16502	16191	17293	16962	18147	17075

Table 2.5. The Comparison of Energy Consumption by the GeneS Algorithm and the Lower Bound Energy Consumption E_{LB} .

auto-2 running on 4, 6, or 8 processor cores, the energy consumption of the GeneS algorithm is the same as the lower bound energy consumption E_{LB} .

(2) *Optimization Trade-Off on Energy, Number of Processor Cores and Timing Constraint.*

We have compared energy consumption of all benchmarks with different algorithms. In this section, we will list detailed data of two benchmarks and analyze other gains and trade-offs in terms of energy, number of processor cores and timing constraint. We use TGFF-1 and TGFF-2 as examples. TGFF-1 is a *slim* DFG while TGFF-2 is a *fat* DFG. Basically, in the slim DFG, it has very long critical path length and there are not a lot of independent nodes; in the fat DFG, its critical path is relatively short and there are many independent nodes. We attempt to use the two extreme cases to compare GeneS with SpringS and PEDF.

Figure 2.9 shows the trend of TGFF-1 with three algorithms in terms of the minimum valid timing constraint and energy consumption on 2, 3, and 4 processor cores, respectively.

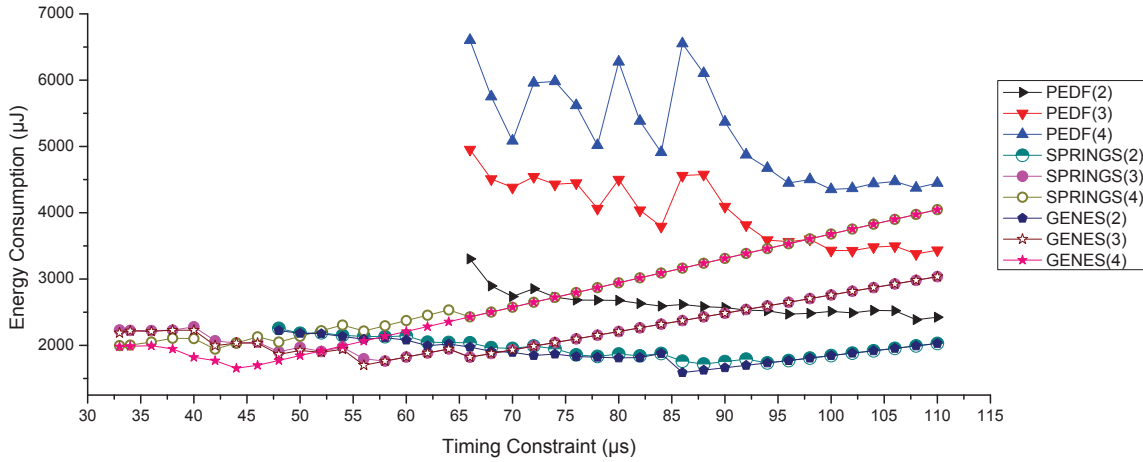


Figure 2.9. The Changing Tendency of Energy and Timing Constraint with Three Algorithms under Different Number of Processor Cores on Benchmark TGFF-1.

In this figure, the notation GeneS(x), SpringS(x) and PEDF(x) represent the execution trace of corresponding algorithm on x processor cores.

From this figure, we can see that: (1) When the timing constraints are in the scope $[33\mu s, 65\mu s]$, PEDF cannot find a feasible solution while SpringS and GeneS can, which means that GeneS does not put a lot of limitation on timing constraints. This is important for embedded systems, especially for those with tight timing constraints. (2) When the timing constraints are more than $65\mu s$, all of three algorithms can find feasible solutions. However, PEDF and SpringS consumes more energy than GeneS. (3) When the number of processor cores is increased, for GeneS, the minimum feasible timing constraint is reduced from $48\mu s$ to $33\mu s$. However, for PEDF, it is always $65\mu s$ no matter on 2, 3 or 4 processor cores. This comparison shows that GeneS can take advantage of the benefit of multiple processor cores to generate a feasible schedule with more tight timing constraint. (4) Although SpringS can also get feasible solutions in timing constraints $[33\mu s, 65\mu s]$, GeneS can save more energy compared to SpringS.

From this analysis, we can see that GeneS can achieve better energy consumption compared with the PEDF algorithm [115] and the SpringS algorithm [65] for the *slim* DFG.

Table 2.6 shows the result of TGFF-2, the *fat* task graph. For simplicity, we list the

TC (μs)	2-core			3-core			4-core		
	PEDF (μJ)	SpringS (μJ)	GeneS (μs)	PEDF (μJ)	SpringS (μJ)	GeneS (μJ)	PEDF (μJ)	SpringS (μJ)	GeneS (μJ)
44	–	–	–	–	–	–	–	4398	4398
58	–	–	–	–	4283	4283	–	3861	3861
60	–	–	–	–	4188	4188	6001	3952	3912
62	–	–	–	4651	4120	4118	5923	3845	3833
88	–	–	4233	4150	3684	3658	4587	3237	2123
98	4191	4172	4074	3897	3413	3399	4345	3604	2493
100	4129	4065	4065	3784	3467	3413	4197	3678	2567
120	3874	3779	3750	3605	3315	1411	4422	4414	3307
140	3687	3638	3464	3870	3867	1951	5158	5150	4047
160	3310	3300	3300	4423	4419	2491	5894	5886	4787
180	3326	3324	1955	4975	4971	3031	6630	6622	5527

Table 2.6. The Comparison for the Schedules Generated by PEDF, SpringS, and GeneS on TGFF-2, a *fat* Task Graph.

part of the data derived from the selected timing constraints to show the trend. From the results, similar conclusion can be obtained as for the *slim* DFG. Some little differences are that, for the *fat* DFG, PEDF behaves better than it does for the *slim* DFG. PEDF can use more processor cores compared to the case of the *slim* DFG, while its parallelism is still not as good as that of GeneS. The energy gap between GeneS and PEDF is reduced. However, GeneS still performs better than PEDF. On average, GeneS can achieve 23.8% improvement over PEDF in energy consumption for the *fat* DFG. And also, GeneS still performs better than SpringS regarding energy consumption.

(3) *Prologue Latency*. In Section 2.5.1, we have analyzed the prologue latency of the RDAG algorithm. The prologue latency *Prologue_Latency* is the time duration in the prologue. Table 2.7 shows the prologue latency of each benchmark with different timing constraints. From the results, we can see that, our approach causes several periods of prologue latency. As discussed before, however, the prologue is only executed once, so the overhead it introduces

Benchmark	TC Range (μs)	<i>Prologue_Latency</i> (μs)	TC Range (μs)	<i>Prologue_Latency</i> (μs)
consumer-1	5–12	38	13–19	65
consumer-2	5–15	34	16–22	64
auto-1	5–13	56	14–26	103
auto-2	335–348	1021	349–705	1319
auto-3	696–1392	6936	1393–2792	14640
telecomm-1	41–51	137	52–110	228
telecomm-2	330–395	1468	396–900	2568
office-1	1925–2584	6751	2585–5725	10782
network-1	253–263	772	264–606	1242
TGFF-1	33–65	260	66–110	389
TGFF-2	44–59	135	60–180	221
TGFF-3	122–500	3970	501–700	7565

Table 2.7. The Prologue Latency of Our RDAG Algorithm with Different Timing Constraints.

is one-time delay. After waiting for the execution of the prologue, tasks can be periodically executed in the new loop kernel as a streaming application is usually repeatedly executed for many times. As our approach can greatly reduce the schedule length of each period as shown above, we can either apply a shorter period or apply DVS and DPM for energy optimization. As we can benefit from each period in the loop kernel, it is usually worth waiting for the execution of the prologue.

(4) *Memory Overhead.* In this section, we give a case study to illustrate the energy consumption caused by the memory subsystem with different methods. We have analyzed the memory overhead caused by the RDAG algorithm in Section 2.5.1. Using our approach, extra memory space is needed to hold data across different periods as the RDAG algorithm regroups tasks from different periods into one period. Therefore, we should conduct analysis based on the energy overhead caused by the extra memory space from our method. However, we cannot find a consistent SRAM energy model that can clearly show the relation between

Power State/Transition	Power	Time
Active	300mW	-
Standby	180mW	-
Nap	30mW	-
Powerdown	3mW	-
Active → Standby	240mW	1 memory cycle
Active → Nap	160mW	8 memory cycle
Active → powerdown	15mW	8 memory cycle
Standby → Active	240mW	+6ns
Nap → Active	160mW	+60ns
Powerdown → Active	15mW	+6000ns

Table 2.8. Power Consumption and Transition Time for Memory.

memory size and energy consumption. For most SRAM chips, only one power consumption parameter is provided, so we cannot effectively evaluate the energy consumption caused by the extra memory space. On the other hand, based on the memory lifetime analysis in Algorithm 2.5.3, we can apply DPM to turn off memory subsystem when it is idle. So in this section, we use Rambus DRAM (RDRAM) as an exemplary memory subsystem to compare the energy consumption between our approach and PEDF by applying DPM.

The power model of RDRAM is listed in Table 2.8. RDRAM offers four power modes: active, standby, nap, and powerdown. The energy consumption in each mode and transition times between different modes are listed in Table 2.8. Rambus RDRAM chip runs at the frequency of $1600MHz$ and provides a peak transfer rate of $3.2GB/s$ [81].

Table 2.9 shows the energy consumption of memory subsystem by the PEDF algorithm and the RDAG algorithm for processing one iteration of each benchmark. As our approach changes intra-iteration data dependencies into inter-iteration data dependencies, the memory chip has to be in active mode during the whole period. For the PEDF algorithm, memory chip can be put in the standby or power down mode to save energy consumption.

Benchmark	TC Range (μs)	GeneS (μJ)	PEDF (μJ)			
			2-core	4-core	6-core	8-core
consumer-1	14–19	5.0	3.3	3.1	3.1	3.1
consumer-2	16–32	7.2	5.7	5.6	5.6	5.6
auto-1	14–26	6.0	4.2	4.1	4.1	4.1
auto-2	349–705	158.1	135.7	114.9	114.9	114.9
auto-3	1393–2792	627.8	525.2	517.1	517.1	517.1
telecomm-1	52–110	24.3	10.0	9.3	9.3	9.3
telecomm-2	396–900	194.4	156.6	143.1	137.7	137.7
office-1	2585–5725	1246.5	1027.8	963.0	938.7	938.7
network-1	264–606	130.5	100.8	95.4	95.4	95.4
TGFF-1	85–110	29.3	24.7	23.9	22.0	22.0
TGFF-2	121–180	45.2	40.8	39.5	39.5	39.5
TGFF-3	564–1000	234.6	224.1	216.3	213.1	210.1

Table 2.9. The Memory Energy Consumption of the PEDF Algorithm and the RDAG Algorithm.

Compared to the energy consumption by the PEDF algorithm, our approach causes extra memory consumption overhead. Compared to the energy consumption caused by processor cores, however, the extra energy consumption of memory subsystem caused by our approach is relatively small. If taking the energy consumption caused by memory as one of the components of the overall energy consumption E_{total} , our approach can still significantly reduce the overall energy consumption compared to the PEDF algorithm.

2.7 Summary

In this chapter, we proposed a two-phase approach to solve the energy optimization problem for periodic dependent tasks on MPSoCs considering various overheads. In the first phase, we proposed a coarse-grained task-level software pipelining algorithm called RDAG to transform periodic dependent tasks into a set of independent tasks based on the retiming

technique [61]. In the second phase, we proposed a genetic algorithm called GeneS for energy optimization. We conducted experiments with a set of benchmarks from E3S [104] and TGFF [28]. The experimental results show that through the combination of software pipelining with DVS and DPM, our approach can fully exploit the potential of MPSoC architectures and the periodic characteristic of streaming applications to reduce energy consumption.

CHAPTER 3

OPTIMALLY REMOVING INTERCORE COMMUNICATION OVERHEAD FOR STREAMING APPLICATIONS ON MPSOCS

3.1 Overview

In real-time systems, a streaming application can be modeled as periodic dependent tasks, in which a stream of data are treated as a sequence of requests that are serviced by the streaming application when arrived [111]. To process continuous stream of data, streaming applications are usually computation-intensive and highly parallelizable [111]; therefore, they are very suitable to be executed on MPSoC (Multiprocessor System-on-Chip) architectures. In order to fully take advantage of the multi-core architecture of MPSoCs, various techniques have been proposed to explore and increase parallelisms of streaming applications. These parallelization techniques usually impose a large amount of intercore communications with significant communication overhead [63]. By removing intercore communication overhead, system performance such as time performance and energy consumption can be improved. Therefore, it becomes an important research problem to effectively remove intercore communication overhead for streaming applications on MPSoC architectures.

An MPSoC is usually designed for a specific streaming application with special hardware and software. Therefore, to optimize a specific streaming application on an MPSoC, one of the key challenges is to generate a good task schedule so that it can satisfy all real-time requests by fully utilizing the computation power of the MPSoC. Then the schedule can be generated by compilers and statically loaded into processor cores or integrated into real-time operating systems. So this chapter aims to develop a scheduling technique for streaming applications on MPSoC architectures.

In the previous work, a lot of techniques have been proposed to solve the scheduling problem for periodic tasks. For scheduling independent tasks, a number of studies have been conducted [15, 22, 41] and several techniques have been proposed. These techniques, however, cannot be directly applied to perform scheduling for periodic dependent tasks. Several studies have been conducted for scheduling periodic dependent tasks on multi-core architectures [24, 64, 115, 117]. However, intercore communication overhead is not considered. So they may not provide good solutions to our problem. In [29, 68, 95, 105], several communication-aware task allocation and scheduling frameworks for MPSoC architectures are proposed. By increasing the parallelism, these techniques may cause more intercore communications. Our technique is a good supplement for these techniques by helping effectively reducing intercore communication overhead.

In this chapter, we propose an approach to totally remove intercore communication overhead by jointly optimizing computation and communication task scheduling for streaming applications on MPSoCs. In particular, in our technique, we reschedule both computation and intercore communication tasks such that the execution of computation and communication tasks of one period can be totally overlapped and the intercore communication overhead is totally removed. Our basic idea is to let some computation and intercore communication tasks be executed in earlier periods (the newly-added periods are called *the prologue*) such that intercore data transfer can be finished before the execution of the tasks that need the data to start. So one problem arises, how to do rescheduling such that the schedule length can be minimized with the minimum prologue length while the intercore communication overhead can be totally removed in each period?

To solve this problem, we first perform schedulability analysis for communication tasks, and theoretically obtain the upper bound of the times needed to reschedule each computation task. Based on this analysis, we formulate the problem as an ILP (Integer Linear Programming) formulation and obtain an optimal solution with joint computation and communication task scheduling. The solution ensures that the intercore communication overhead is totally removed while the schedule length is minimized with the minimum prologue length. As our schedulability analysis produces very tight bounds, the ILP formulation can

be solved efficiently in practice. To the best of our knowledge, this is the first work to optimally remove intercore communication overhead with joint computation and communication task scheduling for streaming applications on MPSoCs.

We have implemented our technique and conducted experiments based on the processor model of ARM11 MPCore processors [10]. Our technique is evaluated with a set of benchmarks from both real-life streaming applications and synthetic task graphs including Embedded Systems Synthesis Benchmarks (E3S) [104], TGFF [28], Automatic Target Recognition (ATR) [71], Computerized Numerical Control (CNC) [51], and an image enhancement application [103]. We compare our technique with the algorithms from Chen et al. [24] and Zhu et al. [117], respectively. The algorithm in Chen et al. [24] is a performance-oriented task scheduling algorithm that can generate near-optimal solutions for periodic dependent tasks on multi-core architectures; the algorithm in Zhu et al. [117] is a power-aware DVS scheduling algorithm that can effectively optimize energy consumption of streaming applications on MPSoCs. The experimental results show that our technique can obtain better performance compared to these algorithms by effectively removing intercore communication overhead. Our technique can achieve a 27.72% reduction in schedule length compared with the algorithm in Chen et al. [24], and a 14.98% reduction in energy consumption compared with the one in Zhu et al. [117] on average.

The remainder of this chapter is organized as follows. In Section 3.2, we present our system models and formally define the problem. In Section 3.3, we perform schedulability analysis, and get constraints that will be integrated into the ILP formulation in Section 3.4. Our optimal joint computation and communication task scheduling technique is presented in Section 3.4. Section 3.5 presents experimental results. Finally, we conclude the chapter and discuss the future work in Section 3.6.

3.2 Models and Concepts

In this section, we introduce several models that will be used in later sections, and then provide the problem formulation.

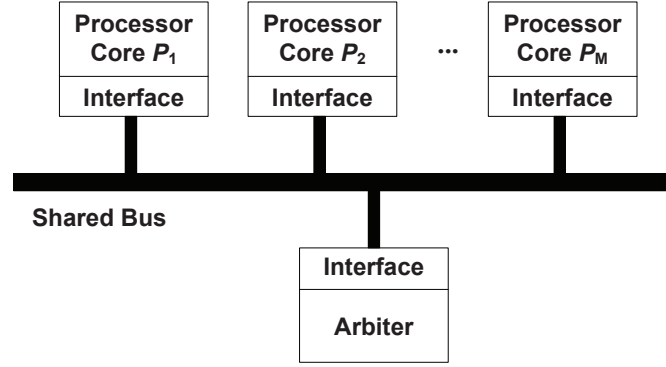


Figure 3.1. The MPSoC Architecture.

3.2.1 System Model

In this chapter, we employ a typical MPSoC architecture shown in Figure 3.1. The MPSoC architecture consists of M processor cores $\{P_1, P_2, \dots, P_M\}$, a shared bus, and a bus arbiter. The M processor cores communicate through a shared bus. The shared bus is adopted as it is the most widely used interconnection architecture. Bus access requests from processor cores are managed by the bus arbiter.

3.2.2 Task Model

In this chapter, streaming applications are modeled as periodic dependent tasks, and periodic dependent tasks are represented by Directed Acyclic Graph (DAG). A DAG $G = (V, E, CT, R)$ is a node-weighted directed acyclic graph. $V = \{T_1, T_2, \dots, T_n\}$ is the node set, and each node represents one periodic task. $E \subseteq V \times V$ is the edge set that defines the data dependency relations for all nodes in V . Each directed edge, $(T_i, T_j) \in E$ ($T_i, T_j \in V$), represents the data dependency between tasks T_i and T_j , i.e., the execution of T_j needs the results generated by the execution of T_i in the same period. $CT : E \mapsto \mathbb{Z}$ is a function that associates every directed edge $(T_i, T_j) \in E$ with a communication task CT_j^i to denote the corresponding data transfer from task T_i to task T_j .

$R : V \mapsto \mathbb{Z}$ is a function that associates every task $T_i \in V$ with a non-negative weight

$R(T_i)$. Node weight represents extended data dependency relations. Initially, all nodes are used to represent data dependency relations inside one period (**intra-period dependency**), so each node weight is zero. As shown later, in our technique, tasks from different periods are regrouped into one period in order to overlap the execution of intercore communication and computation tasks. Therefore, a task node with positive weight is introduced to describe the data dependency relation across multiple periods with other tasks (**inter-period dependency**), and the node weight represents the number of periods involved. We will discuss this in detail in Section 3.2.4.

3.2.3 Static Schedule

A static schedule of a DAG is a repeated pattern for the execution of *one period* of the corresponding periodic dependent tasks. Static scheduling approaches offer a set of benefits to embedded applications including the predictability of worst case schedules and the ability to use complex heuristics [112]. In our work, a static schedule contains both control step assignment (when to start) and processor-core allocation (where to be executed). A schedule must obey all data dependency relations of a DAG. Assuming that I is the period of the given computation tasks and intercore communication tasks, I implies the deadline of the schedule, and the schedule must be finished in I . Given a static schedule, we use $T_{i,\ell}$ and $CT_{j,\ell}^i$ ($\ell \geq 1$) to represent computation task T_i and communication task CT_j^i in period ℓ , respectively. As a schedule is repeatedly executed in each period, T_i and $T_{i,\ell}$ (CT_j^i and $CT_{j,\ell}^i$) are used interchangeably if it is clear from the context.

Given a schedule, let r_i be the release time (schedule step) of task T_i in the first period and let I be the period. Then the release time of T_i in the ℓ th period is $r_{i,\ell} = r_i + (\ell - 1)I$, $\ell \geq 1$. Similarly, for a communication task CT_j^i between tasks T_i and T_j , let r_j^i be the release time of CT_j^i in the first period, then its release time in the ℓ th period is $r_{j,\ell}^i = r_j^i + (\ell - 1)I$, $\ell \geq 1$. e_i is used to represent the execution time of computation task T_i , and e_j^i is used to represent the execution time of communication task CT_j^i . e_j^i is determined by the assignment of tasks T_i and T_j : if T_i and T_j are assigned to the same processor core, it is an intracore communication

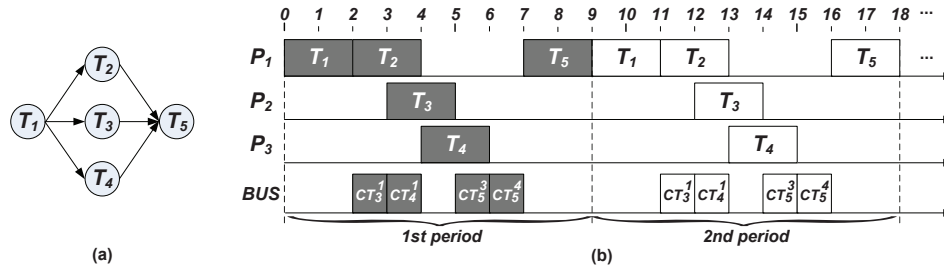


Figure 3.2. A DAG and Its Schedule.

and we assume e_j^i is equal to zero; otherwise, it is an intercore communication and e_j^i is equal to $c(T_i, T_j)/B$, in which $c(T_i, T_j)$ is the data volume transferred and B is the bus bandwidth.

An example is given in Figure 3.2. In Figure 3.2(a), an exemplary DAG is given to model periodic dependent tasks. In the DAG, there are five tasks and the execution time of each task is 2 time units. There are six edges. For each edge, there is one communication task associated. The execution time of an intercore communication task is 1 time unit and that of an intracore communication task is zero. Let the period I be 9 time units. In Figure 3.2(b), a schedule for the DAG is given. The schedule is repeatedly executed in each period and the schedule length of one period is 9 time units. From this example, we can see that the intercore communication introduces big overhead (3 time units) such that the schedule length is increased and a big period has to be used correspondingly.

3.2.4 Communication/Computation Overlapping and Retiming

Figure 3.2 shows that if intercore communication overhead can be removed, then the schedule length in each period can be reduced. We found this can be achieved by regrouping tasks from different periods with joint computation and communication task rescheduling. This is illustrated in Figure 3.3. Given the initial schedule shown in Figure 3.2(b), as each task is periodic, in Figure 3.3(a), we reschedule periodic task T_1 to make it execute one period ahead of T_3 and T_4 . The newly-added period is called *prologue*. As a result, communication tasks CT_3^1 and CT_4^1 that transfer data from task T_1 to T_3 and T_4 , respectively, can be

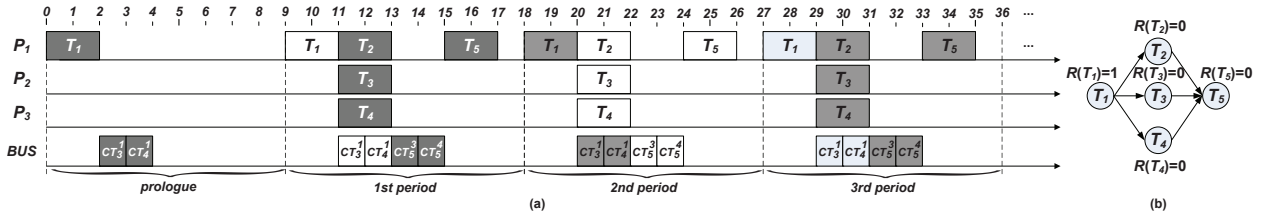


Figure 3.3. Given the initial DAG and schedule in Figure 3.2, (a) a new schedule in which the intercore communication overhead caused by CT_3^1 and CT_4^1 is removed by overlapping communication and computation, and (b) the corresponding DAG with the node weight $R(T_1)$ of computation task T_1 changed to 1.

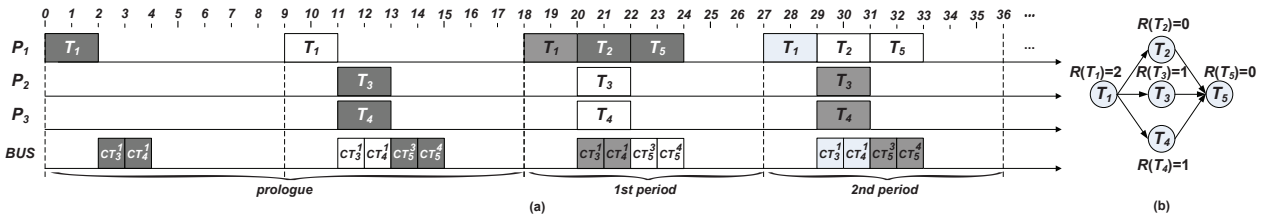


Figure 3.4. Given the initial DAG and schedule in Figure 3.2, (a) a new schedule in which the intercore communication overhead is totally removed, and (b) the corresponding DAG.

rescheduled to be executed one period ahead of T_4 and T_5 (in the prologue) as well. In such a way, the data required by T_3 and T_4 from T_1 are always available in a period; thus, the intercore communication overhead (caused by the communication tasks CT_3^1 and CT_4^1) can be removed inside the schedule, and the schedule length is reduced from 9 to 8.

By adding a prologue and rescheduling some tasks into it, we can effectively remove intercore communication overhead in a schedule by overlapping the execution of communication and computation tasks as shown in Figure 3.3. In this chapter, we apply retiming [61] to describe how many periods of a task node are rescheduled into the prologue and how this influences data dependency relations in a given DAG. The retiming technique is originally proposed to minimize the cycle period of a synchronous circuit by evenly distributing registers [61]. We extend it and it is defined as follows:

Definition 3.2.1. (Retiming) Given a DAG $G = (V, E, CT, R)$, retiming R of G is a func-

tion that maps each node T_i , $T_i \in V$, to an integer $R(T_i)$. $R(T_i)$ is the retiming value of T_i . $R(T_i) = 0$ initially; by retiming T_i once, if it is legal, $R(T_i) = R(T_i) + 1$, and one period of task T_i is rescheduled into the prologue.

As defined above, we use retiming to model task nodes that are rescheduled into the prologue for overlapping communication and computation. Basically, $R(T_i)$ represents how many periods of task T_i are rescheduled into the prologue, and its initial value is zero. Similarly, for communication task CT_j^i , its retiming value $R(CT_j^i)$ represents how many periods of communication task CT_j^i are rescheduled into the prologue. Let **the prologue length** be the number of periods in the prologue, and the prologue length can be calculated by $R_{max} \times I$ where R_{max} is the maximum retiming value among all tasks ($R_{max} = \{max\{R(T_i)\}, T_i \in V\}$) and I is the period. For example, in Figure 3.3(a), one period of task T_1 is rescheduled into the prologue; so its retiming value $R(T_1)$ becomes 1, and the retiming values of other nodes are zero. The prologue length is one.

Based on the data dependency relations in a DAG and its retiming function, we can construct the retimed graph. A retimed graph is used to represent new data dependencies generated by rescheduling task nodes into the prologue. In a retimed graph, a node weight $R(T_i)$ of a task node T_i may be greater than zero as inter-period data dependencies may be introduced by the rescheduling. In general, given a retimed graph G_R , for an edge (T_i, T_j) of G_R , if $R(T_i) - R(T_j) > 0$, it represents that the data generated by T_i at $R(T_i) - R(T_j)$ period(s) before are needed in order to execute T_j . For example, in Figure 3.3(b), for an edge (T_1, T_3) , $R(T_1) - R(T_3) = 1$. It represents that T_3 is dependent on T_1 one period before as the data generated by T_1 one period before are needed by T_3 . This can be seen from Figure 3.3(a) as T_1 is rescheduled one period ahead of T_3 for overlapping communication and computation.

A retiming function must be legal in order to preserve the semantic correctness. **For each edge $(T_i, T_j) \in E$, a retiming function R of G is legal if the relative retiming value $R(T_i) - R(T_j)$ is non-negative.** If $R(T_i) - R(T_j)$ is negative, it implies that the data generated

by T_i from a period in the future are needed in order to execute T_j in the current period. This cannot occur in a correct program. If G_R is a retimed graph of G derived by a legal retiming function R , then G_R is functionally equivalent to G [61].

3.2.5 Problem Analysis

Given a schedule, our objective is to remove intercore communication overhead and at the same time the schedule length is minimized with the minimum prologue length. By minimizing schedule length, the system performance can be improved by adopting a smaller period or exploring the slacks generated for energy reduction with DVS. On the other hand, a longer prologue not only introduces more delays in the beginning but also requires more data buffers to hold intercore communication data. So we want to minimize the prologue length as well. Figure 3.4(a) shows the objective schedule that we want to achieve for the initial DAG and schedule in Figure 3.2. From this example, we can see that we can easily obtain the objective schedule of all computation tasks of one period by removing all the intercore communication overhead and rescheduling each computation task as early as possible. However, it is not trivial to determine how many periods of one task should be rescheduled into the prologue and how to reschedule communication tasks such that the schedule length is minimized with the minimum prologue length. The problem is difficult as we cannot achieve this by rescheduling task nodes into the prologue freely with the constraint of the minimum prologue length. Another difficulty is that removing intercore communication overhead itself cannot directly reduce the schedule length. We still need to reschedule communication tasks such that they can finish before the schedule length in each period. For example, in Figure 3.4(a), in each period, communication tasks CT_5^3 and CT_5^4 are scheduled to start at 4 and 5, respectively, while they start at 5 and 6, respectively, in the initial schedule in Figure 3.2(b).

Let **the objective task schedule** Obj_Sch be the schedule that we want to achieve. The objective task schedule includes the objective computation task schedule Obj_Comp_Sch and the objective communication task schedule Obj_Commu_Sch . Let **the objective computation task schedule** Obj_Comp_Sch be the objective static schedule of all computa-

tion tasks, which totally removes the intercore communication overhead with the minimum schedule length S_{length} . Given a DAG and an initial schedule $Init_Sch$, an objective computation task schedule can be generated by rescheduling each computation task as early as possible from initial schedule $Init_Sch$. **The objective communication task schedule Obj_Commu_Sch** is the objective static schedule of all intercore communication tasks with schedule length S_{length} .

Then the problem of removing intercore communication overhead is transformed to the problem of minimizing the prologue length of the objective computation task schedule with the minimum schedule length. Specifically, given the objective computation task schedule, we want to get, how many periods of each computation task T_i are rescheduled into the prologue (the retiming value $R(T_i)$); how many periods of each communication task CT_j^i are rescheduled into the prologue (the retiming values $R(CT_j^i)$); and the release time r_j^i of each communication task CT_j^i in the objective communication task schedule Obj_Commu_Sch .

3.2.6 Problem Statement

Based on the above problem analysis, we further clarify the problem as follows:

Given a DAG $G = (V, E, CT, R)$, an MPSoC platform with M processor cores, an initial schedule $Init_Sch$ and an objective computation task schedule Obj_Comp_Sch with schedule length S_{length} , we will study the following two problems:

- *What is the upper bound of the prologue length of the objective computation task schedule to guarantee all intercore communication tasks are schedulable in each period with the schedule length S_{length} ?*
- *How to perform joint computation and communication task scheduling such that an objective task schedule Obj_Sch in which intercore communication overhead is totally removed can be generated with the minimum prologue length?*

3.3 Schedulability Analysis

In this section, we perform schedulability analysis for jointly optimizing computation and communication task scheduling for streaming applications on MPSoCs. We first analyze the schedulability of each communication task, and theoretically get the minimum and maximum *relative retiming values of each pair of computation tasks*. Then, based on this analysis and data dependency relations in the DAG, we iteratively get the minimum and maximum *retiming values of each task*, and get the upper bound of *the prologue length* to guarantee computation tasks can totally overlap with communication tasks. The bounds on the relative retiming value of each pair of computation tasks, the bounds on the retiming value of each task, and the bounds on the prologue length will become the constraints of our integer linear programming formulation in Section 3.4. These constraints significantly reduce the search space and greatly improve the efficiency for finding the optimal solution.

3.3.1 Bounds of Relative Retiming Values

In this section, we analyze the bounds of the relative retiming value. The relative retiming value of each pair of computation tasks represents the number of periods involved to guarantee the schedulability of the associated intercore communication task. Specifically, for a pair of computation tasks T_i and T_j , $(T_i, T_j) \in E$, we use $\hat{R}_{min}(T_i, T_j)$ and $\hat{R}_{max}(T_i, T_j)$ to denote that, relative to the retimed task T_j , the minimum and maximum extra numbers of times to perform retiming for task T_i to ensure the schedulability of the associated intercore communication task CT_j^i .

Based on the objective computation task schedule generated from the initial schedule, we propose the following theorem to analyze the schedulability of each communication task and get the minimum and maximum relative retiming values of each pair of computation tasks. The derived relative retiming values can be used to get the upper bound of the prologue length.

Theorem 3.3.1. *For a directed edge $(T_i, T_j) \in E$ ($T_i, T_j \in V$), computation tasks T_i and T_j*

associate with a communication task CT_j^i in the ℓ th period of the objective computation task schedule. After retiming task T_i for $R(T_i)$ times and retiming task T_j for $R(T_j)$ times, as long as the retimed task $T_{i,\ell-R(T_i)}$ is retimed at most two more times relative to the retimed task $T_{j,\ell-R(T_j)}$, the associated intercore communication task CT_j^i is always schedulable on the bus during the time span between the finishing time of the retimed task $T_{i,\ell-R(T_i)}$ and the release time of the retimed task $T_{j,\ell-R(T_j)}$.

Proof. After retiming task T_i for two times relative to the retimed task $T_{j,\ell-R(T_j)}$, task T_i will be scheduled in period $\ell - R(T_j) - 2$, which is two periods ahead of the retimed task $T_{j,\ell-R(T_j)}$. Then the time span between the finishing time of the retimed task $T_{i,\ell-R(T_i)}$ and the release time of the retimed task $T_{j,\ell-R(T_j)}$ is always greater than or equal to period I . As all intercore communication tasks periodically execute in each period, in one period of time, we can find one and exactly one intercore communication task CT_j^i that has data dependency with task T_i and task T_j . Let the intercore communication task CT_j^i in period $\ell - R(T_j) - 1$ be the retimed communication task that associates with tasks $T_{i,\ell-R(T_i)}$ and $T_{j,\ell-R(T_j)}$. Its release time is no earlier than the finishing time of $T_{i,\ell-R(T_i)}$, and its finishing time is no later than the release time of $T_{j,\ell-R(T_j)}$. Therefore, the associated intercore communication task CT_j^i is always schedulable on the bus during that time span. An example is given in Figure 3.5. □

Theorem 3.3.1 gives an upper bound of the maximum relative retiming value of each pair of computation tasks, $\hat{R}_{max}(T_i, T_j) \leq 2$, $(T_i, T_j) \in E$ ($T_i, T_j \in V$). In order to preserve legal retiming, $R(T_i) - R(T_j) \geq 0$. Then, $0 \leq \hat{R}_{min}(T_i, T_j) \leq \hat{R}_{max}(T_i, T_j) \leq 2$.

For a communication task CT_j^i associated with tasks T_i and T_j , if tasks T_i and T_j are assigned to the same processor core, it is an intracore communication and the execution time of the intracore communication task is equal to zero. Therefore, $\hat{R}_{min}(T_i, T_j) =$

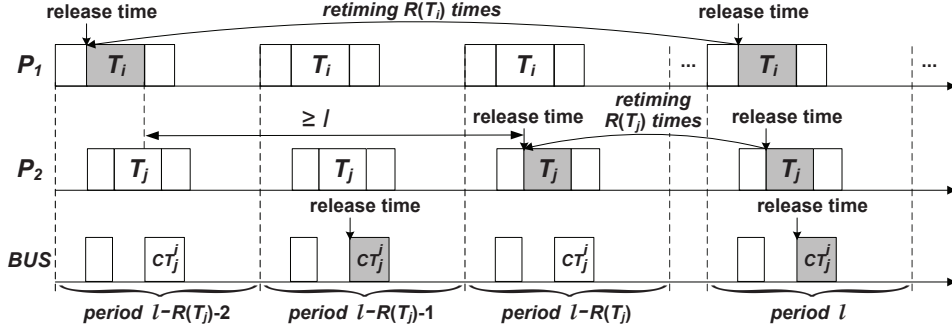


Figure 3.5. An Exemplary Task Schedule of Theorem 3.3.1.

$\hat{R}_{max}(T_i, T_j) = 0$. For tasks T_i and T_j with intercore communication task CT_j^i , $(T_i, T_j) \in E$ ($T_i, T_j \in V$), by comparing the earliest finishing time $r_i + e_i + e_j^i$ of CT_j^i with the release time r_j of task T_j and the schedule length S_{length} in the objective computation task schedule, we can further narrow down the range of the minimum and maximum relative retiming values of each pair of tasks. Based on one period of the objective computation task schedule, we propose the following properties to further analyze the minimum and maximum relative retiming values of each pair of tasks T_i and T_j , $(T_i, T_j) \in E$ ($T_i, T_j \in V$).

Property 3.3.1. For computation tasks T_i and T_j , $(T_i, T_j) \in E$ ($T_i, T_j \in V$), in one period of the objective computation task schedule, if $r_i + e_i + e_j^i \leq r_j$, then by retiming task T_i for at most once relative to the retimed task T_j , the associated intercore communication task CT_j^i is schedulable on the bus. That is, $\hat{R}_{min}(T_i, T_j) = 0$, $\hat{R}_{max}(T_i, T_j) = 1$.

Proof. After retiming task T_j for $R(T_j)$ times, task T_j is rescheduled in the period $\ell - R(T_j)$. In period $\ell - R(T_j)$, the finishing time of the retimed task T_i is $r_i + e_i - (\ell - R(T_j)) \cdot I$, and the release time of the retimed task T_j is $r_j - (\ell - R(T_j)) \cdot I$. According to the condition, in period $\ell - R(T_j)$, the time span between the finishing time of task T_i and the release time of task T_j is greater than or equal to the execution time e_j^i of communication task CT_j^i . An example is shown in Figure 3.6.

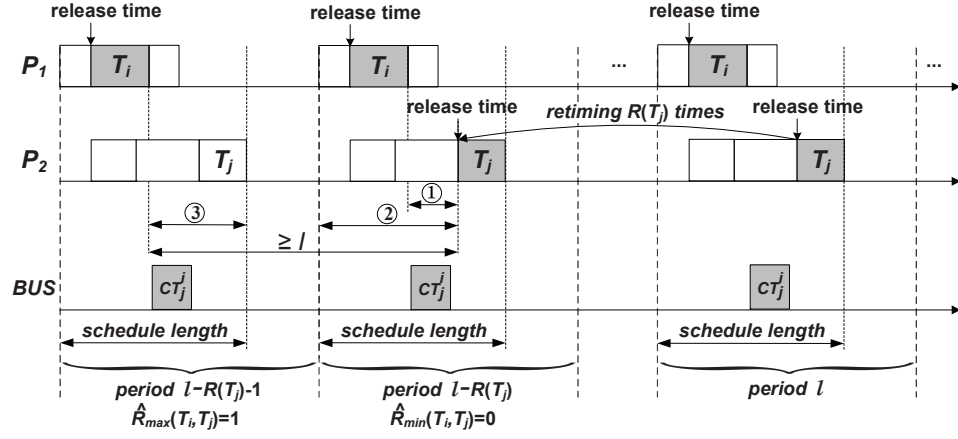


Figure 3.6. An Exemplary Task Schedule of Property 3.3.1.

As each communication task is periodically executed in each period, if communication task CT_j^i is scheduled in this time span (time span 1 in Figure 3.6), there is no need to perform extra retiming for task T_i relative to task T_j . Therefore, $\hat{R}_{min}(T_i, T_j) = 0$. If communication task CT_j^i is not scheduled in this time span, one more extra retiming have to be performed for task T_i relative to task T_j . Then the time span between retimed tasks T_i and T_j is greater than period I . Since both time span 2 and time span 3 are greater than e_j^i ; and the sum of time span 2 and time span 3 are greater than or equal to $I + e_j^i$, either scheduling communication task CT_j^i in time span 2 or in time span 3 can guarantee that its release time is no earlier than the finishing time of the retimed task T_i in period $\ell - R(T_j) - 1$, and its finishing time is no later than the release time of $T_{j, \ell - R(T_j)}$. Therefore, by retiming task T_i for at most once relative to the retiming of task T_j , the associated intercore communication task CT_j^i is schedulable on the bus. \square

Property 3.3.2. For computation tasks T_i and T_j , $(T_i, T_j) \in E$ ($T_i, T_j \in V$), in one period of the objective computation task schedule with schedule length S_{length} , if $r_j < r_i + e_i + e_j^i \leq$

S_{length} or $e_j^i \leq r_j < r_i + e_i + e_j^i$, then by retiming task T_i for once or at most twice relative to that of task T_j , the associated intercore communication task CT_j^i is schedulable on the bus.

That is, $\hat{R}_{min}(T_i, T_j) = 1$, $\hat{R}_{max}(T_i, T_j) = 2$.

Proof. According to the condition $r_j < r_i + e_i + e_j^i$, in period $\ell - R(T_j)$, the time span between the finishing time of task T_i and the release time of task T_j is less than the execution time of communication task CT_j^i . Then at least one more extra retiming have to be performed for task T_i relative to task T_j . An example is shown in Figure 3.7. After retiming T_i one more period ahead of T_j , either time span 4 in period $\ell - R(T_j) - 1$ or time span 5 in period $\ell - R(T_j)$ is greater than e_j^i . If communication task CT_j^i is scheduled in time span 4 or scheduled in time span 5, there is no need to perform extra retiming for task T_i relative to task T_j . Therefore, $\hat{R}_{min}(T_i, T_j) = 1$. If communication task CT_j^i is neither scheduled in time span 4 nor scheduled in time span 5, one more extra retiming for task T_i is needed. Then task T_i is scheduled two periods ahead of the retimed task $T_{j, \ell - R(T_j)}$. According to Theorem 3.3.1, the upper bound of the relative retiming value of each pair of computation tasks T_i and T_j is 2. Therefore, by retiming task T_i for once or at most twice relative to the retiming of task T_j , the associated intercore communication task CT_j^i is schedulable on the bus. □

Property 3.3.3. For computation tasks T_i and T_j , $(T_i, T_j) \in E$ ($T_i, T_j \in V$), in one period of the objective computation task schedule with schedule length S_{length} , if $r_i + e_i + e_j^i > S_{length}$ and $r_j < e_j^i$, then by retiming task T_i for exactly twice relative to that of task T_j , the associated intercore communication task CT_j^i is schedulable on the bus. That is, $\hat{R}_{min}(T_i, T_j) = \hat{R}_{max}(T_i, T_j) = 2$.

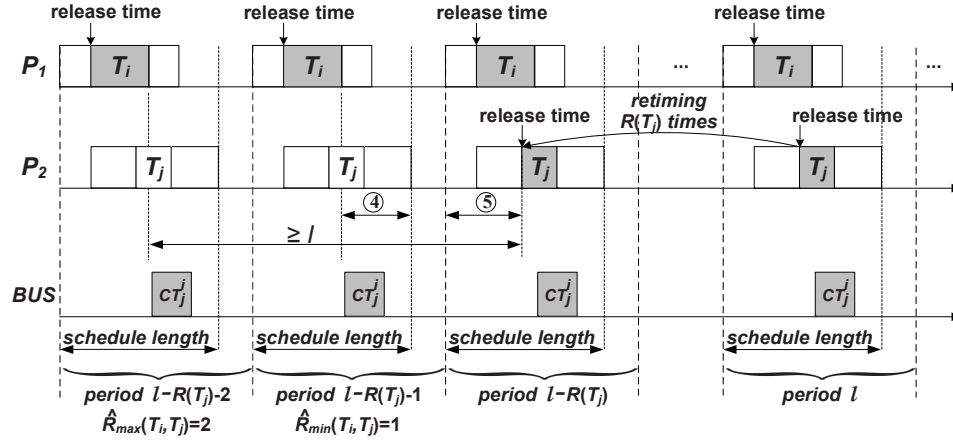


Figure 3.7. An Exemplary Task Schedule of Property 3.3.2.

Proof. According to the condition $r_i + e_i + e_j^i > S_{length}$ and $r_j < e_j^i$, in the objective computation task schedule, the time span between the finishing time of task T_i and the release time of task T_j is less than the execution time e_j^i of communication task CT_j^i . Therefore, at least one more extra retiming have to be performed for task T_i relative to task T_j . An example is shown in Figure 3.8. After retiming T_i one more period ahead of T_j , both time span 6 and time span 7 are less than e_j^i . Task T_i has to perform retiming twice to guarantee the schedulability of communication task CT_j^i . Therefore, $\hat{R}_{min}(T_i, T_j) = 2$. According to Theorem 3.3.1, the upper bound of the relative retiming value of each pair of computation tasks T_i and T_j is 2. Therefore, by retiming task T_i for exactly twice relative to the retiming of task T_j , the associated intercore communication task CT_j^i is schedulable on the bus. \square

Property 3.3.4. *If task T_i and task T_j are assigned to the same processor core, it is an intracore communication. Then, $\hat{R}_{min}(T_i, T_j) = \hat{R}_{max}(T_i, T_j) = 0$.*

The above four properties classify the pair of the minimum and maximum relative retiming values $(\hat{R}_{min}(T_i, T_j), \hat{R}_{max}(T_i, T_j))$ of computation tasks $(T_i, T_j) \in E$ ($T_i, T_j \in V$) into four cases: $[0, 0]$, $[0, 1]$, $[1, 2]$, and $[2, 2]$. For computation tasks T_i and T_j associated

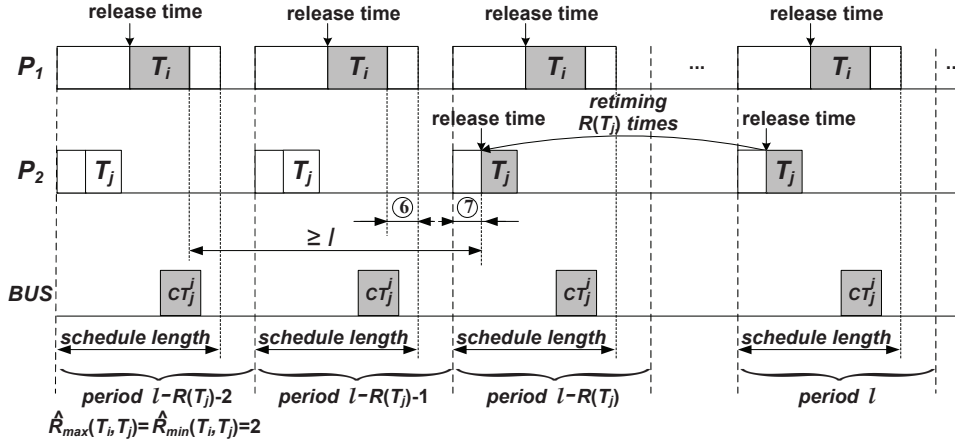


Figure 3.8. An Exemplary Task Schedule of Property 3.3.3.

with communication task CT_j^i , by checking the above four properties, we can get the minimum and maximum relative retiming values of each pair of computation tasks. The derived relative retiming values can be used to obtain the upper bound of the prologue length.

3.3.2 Bounds of the Prologue Length

Based on the definition of prologue length, to minimize the prologue length is equivalent to the problem of reducing the maximum retiming value of all task nodes in the DAG. We observe that the retiming value of each task node also holds certain bounds. In this section, using the minimum and maximum relative retiming values ($\hat{R}_{min}(T_i, T_j), \hat{R}_{max}(T_i, T_j)$) of each pair of computation tasks $(T_i, T_j) \in E$ ($T_i, T_j \in V$) derived from Section 3.3.1 as the input, by checking the data dependency in the DAG, we propose Algorithm 3.3.1 to iteratively get the pair of **the minimum and maximum retiming values** ($R_{min}(T_i), R_{max}(T_i)$) of each task T_i , and get **the upper bound of the prologue length** $\sup P_{length}$. The derived upper bound of the prologue length holds the constraints of the integer linear programming formulation.

Algorithm 3.3.1 presents a bottom-up approach to obtain the pair of the minimum and maximum retiming values of each task T_i . Starting from tasks that do not have any successor

Algorithm 3.3.1 Get the Upper Bound of the Prologue Length for the Objective Computation**Task Schedule**

Input: A DAG $G = (V, E, CT, R)$, the minimum and maximum relative retiming values $(\hat{R}_{min}(T_i, T_j), \hat{R}_{max}(T_i, T_j))$ of each pair of computation tasks (T_i, T_j) , $(T_i, T_j) \in E$ ($T_i, T_j \in V$).

Output: The minimum and maximum retiming values $(R_{min}(T_i), R_{max}(T_i))$ of each task T_i , the upper bound of the maximum retiming value of all task nodes $\sup R(T)$, the upper bound of the prologue length $\sup P_{length}$.

```
1: for each  $T_i \in V$  do
2:    $R_{min}(T_i) \leftarrow 0, R_{max}(T_i) \leftarrow 0$ 
3: end for
4: if  $T_i \in V$  has successor task(s) then
5:   for each successor task  $T_j$  of task  $T_i$  do
6:     if  $\hat{R}_{min}(T_i, T_j) + R_{min}(T_j) > R_{min}(T_i)$  then
7:        $R_{min}(T_i) \leftarrow \hat{R}_{min}(T_i, T_j) + R_{min}(T_j)$ 
8:     end if
9:     if  $\hat{R}_{max}(T_i, T_j) + R_{max}(T_j) > R_{max}(T_i)$  then
10:       $R_{max}(T_i) \leftarrow \hat{R}_{max}(T_i, T_j) + R_{max}(T_j)$ 
11:    end if
12:  end for
13: end if
14:  $\sup R(T) \leftarrow \{max\{R_{max}(T_i)\}, T_i \in V\}$ 
15:  $\sup P_{length} \leftarrow \sup R(T) \times I$ 
```

tasks, Algorithm 3.3.1 iteratively checks the data dependency relations in the DAG and gets the minimum and maximum retiming values of each predecessor task. For a task T_i that does not have any successor tasks, both its minimum and maximum retiming values are equal to zero, $R_{min}(T_i) = R_{max}(T_i) = 0$. For a task T_i with at least one successor tasks, this corresponds to at least one edge pointing from task T_i to other task nodes in the task graph. For each successor task T_j of task T_i , $(T_i, T_j) \in E$ ($T_i, T_j \in V$), the value of $\hat{R}_{min}(T_i, T_j) + R_{min}(T_j)$ can be obtained. Among these values, the maximum value of $\hat{R}_{min}(T_i, T_j) + R_{min}(T_j)$ will be assigned to $R_{min}(T_i)$. Similarly, the maximum of $\hat{R}_{max}(T_i, T_j) + R_{max}(T_j)$

will be assigned to $R_{max}(T_i)$. After got the upper bound of the maximum retiming value of all tasks, $\sup R(T) = \{max\{R_{max}(T_i)\}, T_i \in V\}$, we can derive the upper bound of the prologue length, $\sup P_{length} = \sup R(T) \times I$.

We use a run time example in Figure 3.9 to illustrate the proposed approach. Given a DAG in Figure 3.9(a) and the initial task schedule in Figure 3.9(b), the execution time of each computation task and each communication task are derived, shown in Figure 3.9(c). By totally removing the intercore communication overhead and getting the earlier release time of each computation task, an objective computation schedule of one period is generated in Figure 3.9(d). In Figure 3.9(e), each pair of the numbers represents the minimum and maximum relative retiming values of each pair of tasks, respectively, which can be obtained by checking the four properties in Section 3.3.1. For example, task T_A and task T_C associate with intercore communication task CT_C^A . From Figure 3.9(c), we can get the execution time of task T_A , $e_A = 2$, and get the execution time of communication task CT_C^A , $e_C^A = 1$. In the objective computation task schedule in Figure 3.9(d) with schedule length $S_{length} = 13$, we can obtain the finishing time of task T_A , $r_A + e_A = 2$, and the release time of task T_C , $r_C = 2$. As $r_C < r_A + e_A + e_C^A < S_{length}$, it satisfies the condition of Property 3.3.2. Therefore, $\hat{R}_{min}(T_A, T_C) = 1$, $\hat{R}_{max}(T_A, T_C) = 2$.

To get the pair of the minimum and maximum retiming values of each task, we iteratively check the data dependency relations in the DAG using a breadth-first manner. In Figure 3.9(f), we put each pair of the minimum and maximum retiming values of each task above each task node. For a task that does not have any successor tasks, (e.g., task T_I), both its minimum and maximum retiming values are equal to zero. For a task with at least one successor tasks (e.g., task T_A), the maximum value of $\hat{R}_{min}(T_i, T_j) + R_{min}(T_j)$ will be assigned to $R_{min}(T_i)$. For example, task T_A has two successor tasks, T_B and T_C . The maximum value between $\hat{R}_{min}(T_A, T_B) + R_{min}(T_B)$ and $\hat{R}_{min}(T_A, T_C) + R_{min}(T_C)$ is 2. Thus, $R_{min}(T_A)$ is 2. Similarly, the maximum of $\hat{R}_{max}(T_i, T_j) + R_{max}(T_j)$ will be assigned to $R_{max}(T_i)$. Therefore, $\hat{R}_{max}(T_A, T_B) + R_{max}(T_B) = 4$ will be assigned to $R_{max}(T_A)$. Then the minimum and maximum retiming values of task T_A are 2 and 4, respectively. Follow the data dependency in the DAG (from task T_I to task T_A), we can iteratively get the minimum

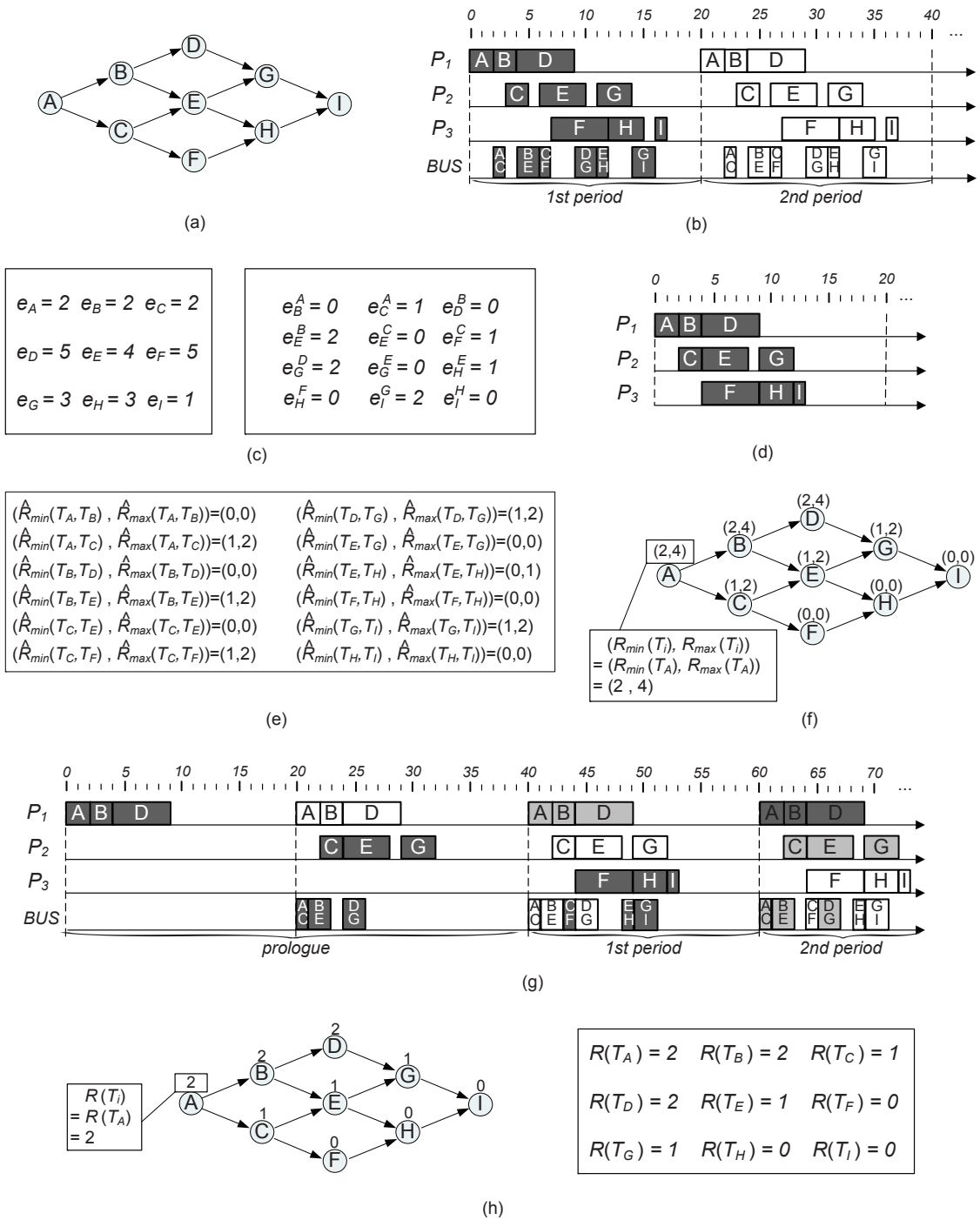


Figure 3.9. A Run Time Example of the Proposed Approach.

and maximum retiming values of each task. Among them, the maximum retiming value of all tasks $\sup R(T)$ is 4, which is equal to the maximum retiming value of task T_A . Therefore, the upper bound of the prologue length $\sup P_{length} = \sup R(T) \times I$, is 4 periods of time.

3.4 Optimal Joint Computation and Communication Task Scheduling

In this section, we first propose our approach that jointly optimizes computation and communication task scheduling. An ILP model is presented to solve the intercore communication overhead minimization problem and obtain an optimal task schedule with the minimum prologue length. Then, as an extension, in Section 3.4.2, we combine our approach with DVS techniques to save energy consumption.

3.4.1 Joint Computation and Communication Task Scheduling (JCCTS)

Our joint computation and communication task scheduling approach is shown in Algorithm 3.4.1. Given a DAG and an initial task schedule $Init_Sch$, an objective computation task schedule of one period can be generated, in which each computation task is rescheduled as early as possible and intercore communication overhead is totally removed. Then the release time and the execution time of each computation task in one period of the objective computation task schedule can be obtained. Based on the objective computation task schedule, we perform schedulability analysis in Section 3.3.1 and get the bounds of the relative retiming value of each pair of computation tasks. Then we use Algorithm 3.3.1 to get the bounds on the retiming value of each task and the bounds on the prologue length. The derived constraints are integrated into our integer linear programming formulation.

In the ILP model, the objective function seeks to minimize the maximum retiming value of all task nodes. In order to generate the optimal solution, the ILP model considers the following constraints:

(1) *System Architecture*. Constraint 1 represents that, for a shared bus, when a communication task CT_j^i has been scheduled to start execution at time r_j^i , no other communication tasks

Algorithm 3.4.1 Joint Computation and Communication Task Scheduling (JCCTS)

Input: A DAG $G = (V, E, CT, R)$ and an initial task schedule $Init_Sch$.

Output: An optimal objective task schedule Obj_Sch .

- 1: Perform schedulability analysis in Section 3.3 and obtain the constraints of the ILP formulation.
- 2: Add the following constraints into the ILP formulation:

$$\left\{ \begin{array}{l} \forall CT_j^i, CT_{j'}^{i'} \in CT, r_j^i \leq r_{j'}^{i'} : \\ \quad r_{j'}^{i'} - r_j^i \geq e_j^i \quad (1) \\ \forall CT_j^i \in CT, (T_i, T_j) \in E, (T_i, T_j \in V) : \\ \quad r_j^i \geq 0 \quad (2) \\ \quad r_j^i \leq I - e_j^i \quad (3) \\ \quad r_j^i + R(T_i) \cdot I - R(CT_j^i) \cdot I \geq r_i + e_i \quad (4) \\ \quad r_j^i + R(T_j) \cdot I - R(CT_j^i) \cdot I \leq r_j - e_j^i \quad (5) \\ \quad R(CT_j^i) - R(T_i) \leq 0 \quad (6) \\ \quad R(CT_j^i) - R(T_j) \geq 0 \quad (7) \\ \quad R(T_i) - R(T_j) \geq \hat{R}_{min}(T_i, T_j) \quad (8) \\ \quad R(T_i) \geq R_{min}(T_i) \quad (9) \\ \quad R(T_i) \leq R_{max}(T_i) \quad (10) \\ \quad R(T_i) - R_{max} \leq 0 \quad (11) \\ \quad R_{max} \leq \mathbf{sup}R(T) \quad (12) \end{array} \right.$$

- 3: Set the objective function: Minimize R_{max} .
 - 4: Find the feasible release time of each communication task, the retiming value of each computation task, and the retiming value of each communication task that satisfy the ILP formulation.
 - 5: Generate objective task schedule Obj_Sch based on the release time, the execution time, and the retiming value of each computation task and each communication task.
-

$CT_j^{i'}$ can start execution until the communication task CT_j^i has completed its execution.

(2) *Characteristics of Periodic Dependent Tasks.* Considering the ℓ th period of the objective computation task schedule, $\ell \geq 1$, Constraint 2 and Constraint 3 illustrate that, each communication task $CT_{j,\ell}^i$ should be schedulable in the time span $[(\ell - 1) \cdot I, \ell I]$. That is, the release time r_j^i of intercore communication task CT_j^i should be no earlier than the start of the period, and the finishing time $r_j^i + e_j^i$ of communication task CT_j^i should be no later than the end of the period.

$$\begin{cases} (\ell - 1) \cdot I \leq r_j^i + (\ell - 1) \cdot I \\ r_j^i + (\ell - 1) \cdot I \leq \ell \cdot I - e_j^i \end{cases}$$

By simplifying the above inequalities and moving variables to the left-hand side of the inequalities, we can obtain the following constraints.

$$\begin{cases} r_j^i \geq 0 \\ r_j^i \leq I - e_j^i \end{cases}$$

Constraint 4 and Constraint 5 satisfy the constraints of data dependency to ensure that intercore communication task CT_j^i is schedulable during the time span between the finishing time of retimed task $T_{i,\ell-R(T_i)}$ and the release time of retimed task $T_{j,\ell-R(T_j)}$.

$$\begin{cases} r_i + e_i - R(T_i) \cdot I \leq r_j^i - R(CT_j^i) \cdot I \\ r_j^i + e_j^i - R(CT_j^i) \cdot I \leq r_j - R(T_j) \cdot I \end{cases}$$

We can simplify the above expressions and move variables to the left-hand side of the inequalities. Then we get the formulations of Constraint 4 and Constraint 5.

$$\begin{cases} r_j^i + R(T_i) \cdot I - R(CT_j^i) \cdot I \geq r_i + e_i \\ r_j^i + R(T_j) \cdot I - R(CT_j^i) \cdot I \leq r_j - e_j^i \end{cases}$$

(3) *Legal Retiming.* Constraint 6 and Constraint 7 preserve the semantic correctness to keep the retiming function legal. For computation tasks T_i and T_j , $(T_i, T_j) \in E$ ($T_i, T_j \in V$), the retiming value of task T_j is always less than or equal to the retiming value of task T_i .

$$R(T_j) \leq R(T_i)$$

Similarly, the retiming value of the associated intercore communication task CT_j^i is no more than the retiming value of task T_i , and it is no less than the retiming value of task T_j .

$$R(T_j) \leq R(CT_j^i) \leq R(T_i)$$

Therefore, we can obtain the formulation of Constraint 6 and Constraint 7.

$$\begin{cases} R(CT_j^i) - R(T_i) \leq 0 \\ R(CT_j^i) - R(T_j) \geq 0 \end{cases}$$

(4) *The Bound of the Relative Retiming Value.* In Section 3.3.1, we proposed four properties to analyze the relative retiming value of each pair of computation tasks. By checking the conditions of these four properties, the bound of the relative retiming value can be obtained. Constraint 8 gives the lower bound of the relative retiming value of each pair of tasks. To guarantee the schedulability of the intercore communication task CT_j^i , the retimed computation task T_i should be at least $\hat{R}_{min}(T_i, T_j)$ periods ahead of the retimed computation task T_j .

$$R(T_i) - R(T_j) \geq \hat{R}_{min}(T_i, T_j)$$

(5) *The Bound of the Prologue Length.* In Section 3.3.2, we proposed Algorithm 3.3.1 to get the bounds on the retiming value of each task and the bounds on the prologue length. For each task T_i , its minimum retiming value $R_{min}(T_i)$ and its maximum retiming value $R_{max}(T_i)$ can be obtained. Then the retiming value of each task T_i , $R(T_i)$, is always greater than or equal to its minimum retiming value $R_{min}(T_i)$, and it is always less than or equal to its maximum retiming value $R_{max}(T_i)$. Therefore, $R(T_i)$ is in the range $[R_{min}(T_i), R_{max}(T_i)]$.

$$R_{min}(T_i) \leq R(T_i) \leq R_{max}(T_i)$$

Then we get the formulations of Constraint 9 and Constraint 10.

$$\begin{cases} R(T_i) \geq R_{min}(T_i) \\ R(T_i) \leq R_{max}(T_i) \end{cases}$$

Constraint 11 finds the maximum retiming value $R_{max} = \{max\{R(T_i)\}, T_i \in V\}$ among all computation tasks.

$$R(T_i) - R_{max} \leq 0$$

Constraint 12 restricts that the maximum retiming value R_{max} should be less than or equal to the upper bound of the retiming value $\sup R(T)$, which is the number of periods of time

for the upper bound of the prologue length $\sup P_{length}$. Then, we get the formulation of Constraint 12.

$$R_{max} \leq \sup R(T)$$

The objective function of the ILP formulation tries to minimize the maximum retiming value R_{max} , which is the number of periods of tasks in the prologue. Using this ILP model, it can derive the release time r_j^i of each intercore communication task CT_j^i in the first period of the objective task schedule. It can also get the retiming value $R(CT_j^i)$ of each intercore communication task CT_j^i and the retiming value $R(T_i)$ of each computation task T_i . Then an optimal objective communication task schedule can be generated based on the derived release time and the retiming value of each communication task. Communication task CT_j^i is first released at $r_j^i - R(CT_j^i) \cdot I$, and it is periodically executed in each of the following periods. Similarly, computation task T_i is first released at $r_i - R(T_i) \cdot I$, and it is periodically executed in each of the following periods. As the objective function of the ILP formulation seeks to minimize R_{max} , the computation task schedule with the minimum prologue length is guaranteed.

As the ILP model integrates several constraints (e.g., the bounds of the relative retiming value, and the bound of the prologue length), these constraints greatly reduce search space for finding the optimal solution. The proposed approach can efficiently solve the problem and provide the optimal communication task schedule to ensure that the corresponding prologue length of the objective computation task schedule is minimized.

We continue to use Figure 3.9 as an example. Figure 3.9(g) shows the objective task schedule generated by the proposed approach, in which the intercore communication overhead is totally removed and the schedule length in each period is greatly reduced. Compared with the initial task schedule in Figure 3.9(b), the schedule length is reduced from 17 to 13. The objective task schedule in Figure 3.9(g) is generated based on the results derived from the ILP formulation. Each computation task or each communication task can get its corresponding release time. For example, the ILP formulation gets the retiming value of task T_A , $R(T_A) = 2$. So the first release of task T_A will be rescheduled two periods ahead of the first

period of objective task schedule. Then task T_A is periodically executed in each of the following periods. For intercore communication tasks, the ILP formulation can get the release time r_j^i and the retiming value $R(CT_j^i)$ of each communication task CT_j^i . For example, the ILP formulation gets the retiming value of communication task CT_C^A , $R(CT_C^A) = 1$, and it obtains the release time of communication task CT_C^A , $r_C^A = 0$. We can see that the first release of communication task CT_C^A is one period ahead of the first period of the objective communication task schedule. After that, communication task CT_C^A is periodically executed in each of the following periods. In Figure 3.9(h), the number above each task node represents the retiming value of each computation task. The maximum retiming value of all tasks is 2, which is equal to the retiming value of computation task T_A (i.e., $R(T_A) = 2$). Therefore, the prologue length is in 2 periods of time.

3.4.2 The Extension for Minimizing Energy Consumption

After removing intercore communication overhead, the proposed technique can reduce schedule length in each period. By utilizing the slacks generated, it is possible to reduce energy consumption by exploring existing power-aware techniques, such as DVS. Here, we present how to extend our technique to save energy consumption.

Given an initial task schedule and the objective schedule length, we can first explore a DVS technique to generate an objective computation task schedule with the minimum energy consumption. In this way, the scheduling slacks caused by the intercore communication overhead can be fully utilized. For the second step, based on the objective computation task schedule, the proposed technique can be used to generate an optimal objective schedule. As DVS techniques can generate energy efficient objective computation task schedule, these techniques can be served as the input task schedule of our technique. Note that our technique is general enough to cope with various DVS techniques. Therefore, various DVS techniques can be explored to generate an objective computation task schedule. Combined with DVS techniques, our technique can be extended to reduce energy consumption and improve the system performance.

3.5 Experiments

To evaluate the effectiveness of our approach, we conduct experiments on various benchmarks from both real-life streaming applications and synthetic task graphs. The objective of the evaluation is to quantify the gains of our approach over the previous work in terms of two performance metrics: schedule length and energy consumption. Our approach is compared with the STC algorithm [24] and the FLSSR algorithm [117]. The STC algorithm is a performance-oriented task scheduling algorithm that can generate near-optimal solutions for periodic dependent tasks on multi-core architectures; the FLSSR algorithm is a power-aware DVS scheduling algorithm that can effectively optimize energy consumption of streaming applications on MPSoCs. So they are selected to do comparison. In this section, we first introduce the experimental environment and performance metrics. Then we present the experimental results and discussion.

3.5.1 Experimental Setup

Our experiments are conducted based on the processor model of ARM11 MPCore processor [10]. The ARM11 MPCore processor implements ARM11 microarchitecture and can be configured to contain between one and four processors [10]. Therefore, we implement a simulator based on the processor model of ARM11 MPCore and tests various benchmarks under 2, 3, and 4 processor cores. The ARM11 MPCore utilizes a single 64-bit AMBA AXI system bus to interconnect different cores and provides throughput of 1.3Gbytes/sec. In our experiments, the ARM11 MPCore operates at 3 typical frequencies: 333MHz, 400MHz and 532MHz.

We use the following metrics to evaluate the performance of our approach: (1) *Schedule length*. By removing intercore communication overhead, our approach can reduce the schedule length in each period such that the system performance can be improved by adopting a shorter period. For each benchmark, the initial task schedule is generated by the Starting Time Controller (STC) algorithm in [24] because of its reasonably good performance for task mapping and task ordering for multi-core platform. We test the schedule length of each

benchmark with the frequency of 333MHz. We also present the corresponding experimental results on the maximum retiming value R_{max} among all tasks for each benchmark. (2) *Energy consumption*. By fully utilizing the slacks obtained from the proposed communication overhead minimization approach, the energy consumption can be reduced by combining it with DVS (Dynamic Voltage Scaling) techniques. The initial power-aware task schedule is generated by the DVS algorithm, Fixed-order List Scheduling with Shared Slack Reclamation (FLSSR) in Zhu et al. [117], which provides relatively good scheduling for periodic dependent tasks running on MPSoC architectures. We present the experimental results on energy consumption under different periods with different numbers of processor cores.

In this chapter, a processor core in an MPSoC can support DVS. The dynamic power consumption of a processor core at a voltage level V_{dd} is calculated based on the power model in Rabaey et al. [91]: $P_{dynamic}(V_{dd}) = C_{SW} \cdot f_{op} \cdot V_{dd}^2$, where C_{SW} is the capacitance, and f_{op} is the frequency of a processor core at voltage level V_{dd} . The operating voltage V_{dd} of each ARM11 MPCore is set as 1.22V, 1.33V, and 1.47V for its corresponding clock frequency, and the capacitance C_{SW} is set as 40pF from the data sheet of Freescale Semiconductor i.MX35 multimedia application processor [33]. The i.MX35 processor implements an ARM11 microprocessor core, and it is designed for automotive entertainment and navigation applications [33].

We conduct experiments on various benchmarks from E3S [104], ATR [71], CNC [51], Image Enhancement [103], and TGFF [28]. Among them, *consumer* from Embedded Systems Synthesis Benchmarks (E3S) represents an embedded consumer electronic application consisting of tasks like JPEG compression, JPEG decompression, high pass gray-scale filter, RGB to CYMK conversion, and RGB to YIQ conversion. *Telecom* from E3S represents an embedded telecom application. ATR is a streaming application that does pattern matching of targets in images. CNC controller is an automatic machining tool which is used to produce real-time user-designed work pieces. Image enhancement application uses Sobel gradient, histogram, and Laplacian to improve the image quality. *kseries_parallel* and *kseries_parallel_xover* from TGFF are several synthetic task graphs generated by TGFF using the sample input files that come with the software package.

We implement the simulator in C. Based on this simulator, we can generate the objective computation task schedule and obtain the constraints for integer linear programming (ILP) formulation. The ILP formulation is solved by the open source program linear programming solver, LP_solve_5.5 [32]. Both the simulator and LP_solve_5.5 are running on a 2.83GHz Intel Core2 Quad processor with 4GB memory.

3.5.2 Results and Discussion

In this section, we present and discuss the experimental results. We first compare our approach with the STC algorithm [24] in terms of schedule length. Then we present the results in energy consumption obtained by our approach and the FLSSR algorithm [117]. Finally, we present the prologue length introduced by our approach.

(1) *Schedule Length.* Table 3.1 presents the experimental results in schedule length obtained by our approach (“JCCTS”) and the STC algorithm in [24] (“STC”) under 2, 3, and 4 processor cores. Columns “# of task”, “# of edge”, “STC (μs)”, “JCCTS (μs)”, and “Reduction (%)” represent the number of tasks, the number of edges, the schedule length from the STC algorithm, the schedule length from our approach, and the average reduction in schedule length in one period by comparing our approach with the STC algorithm, respectively. From the experimental results, our approach can achieve an average 27.72% reduction in schedule length compared with the STC algorithm.

With the same period, for both our approach and the STC algorithm, the schedule length can be reduced when the number of processor cores increases. This can be observed in Table 3.1 in which for each benchmark, less schedule length can be obtained with 3 or 4 cores compared with that with 2 cores. However, for our approach, it can be observed that the schedule length does not increase when the number of processor cores increases from 3 to 4 for some benchmarks such as *consumer* and *telecom*. This is because the number of tasks in these benchmarks is relatively small. The parallelism of the benchmark has been fully exploited by our approach when the number of processor cores increases from 2 to 3. Therefore, the additional processor cores cannot provide more benefit as there are not more

Table 3.1. Comparison in Schedule Length of Our JCCTS Approach and the STC Algorithm in Chen et al. [24] on 2, 3, and 4 Processor Cores.

Benchmarks	# of task	# of edge	STC (μs)	JCCTS (μs)	Reduction (%)
2 processor cores					
consumer1	7	8	75543	72132	4.52
consumer2	5	4	52798	46704	11.54
telecom	6	6	947	813	14.15
ATR	14	15	11988	10694	10.79
CNC	8	9	1866	1091	41.53
image	8	11	272	210	22.79
kseries_parallel1	30	33	2644	1316	50.23
kseries_parallel2	20	19	1391	905	34.94
kseries_parallel3	62	61	5276	2769	47.52
kseries_parallel4	47	46	3876	2118	45.36
kseries_parallel_xover1	30	37	2788	1342	51.87
kseries_parallel_xover2	21	24	1380	905	34.42
kseries_parallel_xover3	38	41	2490	1694	31.97
kseries_parallel_xover4	27	30	1732	1152	33.49
3 processor cores					
consumer1	7	8	76145	63681	16.37
consumer2	5	4	52758	46704	11.48
telecom	6	6	939	813	13.42
ATR	14	15	7799	7403	5.08
CNC	8	9	1417	884	37.61
image	8	11	225	189	16
kseries_parallel1	30	33	2069	997	51.81
kseries_parallel2	20	19	1083	665	38.6
kseries_parallel3	62	61	4209	1974	53.1
kseries_parallel4	47	46	2878	1578	45.17
kseries_parallel_xover1	30	37	2017	1016	49.63
kseries_parallel_xover2	21	24	994	658	33.8
kseries_parallel_xover3	38	41	1774	1152	35.06
kseries_parallel_xover4	27	30	1185	987	16.71
4 processor cores					
consumer1	7	8	72621	63681	12.31
consumer2	5	4	52734	46704	11.43
telecom	6	6	930	813	12.58
ATR	14	15	7671	7403	3.49
CNC	8	9	959	868	9.49
image	8	11	208	189	9.13
kseries_parallel1	30	33	1674	815	51.31
kseries_parallel2	20	19	666	570	14.41
kseries_parallel3	62	61	3636	1661	54.32
kseries_parallel4	47	46	2196	1316	40.07
kseries_parallel_xover1	30	37	1485	844	43.16
kseries_parallel_xover2	21	24	823	650	21.02
kseries_parallel_xover3	38	41	1225	1110	9.39
kseries_parallel_xover4	27	30	1069	931	12.91

tasks can be executed in parallel.

(2) *Energy Consumption.* As an extension of the proposed approach, energy consumption can be reduced by combining our approach with the DVS technique. We evaluate and compare our approach with the FLSSR algorithm in [117]. Table 3.2 shows the results for all 14 benchmarks with 2, 3 and 4 processor cores, in which column “Timing Constraints (μs)” refers to the range of the timing constraints, and columns “FLSSR (μJ)”, “JCCTS (μJ)”, and “Reduction (%)” represent the energy consumption obtained by the FLSSR algorithm, the energy consumption obtained by our approach, and the reduction in energy consumption by comparing our approach with the FLSSR algorithm, respectively. The range of the timing constraints is determined by the minimum timing constraint and the maximum timing constraint. The minimum timing constraint refers to the tightest timing constraint that can generate a feasible schedule by the FLSSR algorithm, while the maximum timing constraint refers to the smallest timing constraint of a feasible schedule in which each task operates at its lowest frequency. In the experiments, we test each benchmark with different timing constraints. Starting from the minimum timing constraint to perform task scheduling, we gradually increase the timing constraint by $20\mu s$ each time. The experimental results list the average energy consumption of each benchmark for different ranges of timing constraints.

From the experimental results, we observe that, with the increasing of the number of processor cores, the improvement steadily increases. This is because increasing the parallelism may cause more intercore communications, which leads to more intercore communication overheads. Our approach can further effectively reduce the energy consumption by utilizing the slacks generated from the removal of intercore communication overhead. We also observe that, for real-life streaming applications (benchmarks *consumer*, *telecom*, *ATR*, *CNC*, and *image*), our approach can achieve a reduction between 5.90% and 19.52% compared with the FLSSR algorithm. For synthetic task graphs with higher numbers of tasks (benchmarks *kseries_parallel*, and *kseries_parallel_xover*), our approach can obtain up to 28.51% reduction in energy consumption (*kseries_parallel_xover3* for 4 processor cores). This observation also shows that the proposed technique can take advantage of multiple processor cores to reduce energy consumption. Although the FLSSR algorithm can get feasible

Table 3.2. Comparison in Energy Consumption of Our JCCTS Approach and the FLSSR Algorithm in Zhu et al. [117] on 2, 3, and 4 Processor Cores.

	Timing Constraints (μs)	FLSSR (μJ)	JCCTS (μJ)	Reduction (%)
2-core				
consumer1	77806 – 112237	2588.53	2430.57	6.10
consumer2	55416 – 81112	1869.30	1638.26	12.36
telecom	539 – 1018	30.10	26.15	13.12
ATR	11869 – 17459	504.52	474.76	5.90
CNC	1081 – 1541	40.89	36.99	9.53
image	234 – 330	8.80	8.23	6.48
kseries_parallel1	1341 – 1999	52.20	49.90	4.42
kseries_parallel2	949 – 1388	35.97	33.61	6.57
kseries_parallel3	3708 – 5333	142.16	133.62	6.01
kseries_parallel4	2688 – 3994	102.58	96.80	5.63
kseries_parallel_xover1	2008 – 2858	77.44	69.98	9.63
kseries_parallel_xover2	860 – 1246	33.86	30.35	10.35
kseries_parallel_xover3	1650 – 2462	67.98	60.82	10.54
kseries_parallel_xover4	1175 – 1718	47.70	41.03	13.98
Average				8.62
3-core				
consumer1	66327 – 95023	2424.69	2255.60	6.97
consumer2	55416 – 81112	1869.30	1638.26	12.36
telecom	539 – 1018	30.10	26.15	13.12
ATR	9474 – 14264	502.13	405.52	19.24
CNC	884 – 1277	43.00	34.61	19.52
image	223 – 308	11.12	9.31	16.24
kseries_parallel1	1153 – 1621	48.37	43.71	9.64
kseries_parallel2	823 – 1137	33.98	28.97	14.74
kseries_parallel3	2896 – 4249	124.84	113.12	9.39
kseries_parallel4	2315 – 3247	97.12	82.49	15.06
kseries_parallel_xover1	843 – 2493	79.75	66.67	16.40
kseries_parallel_xover2	750 – 1025	32.94	27.58	16.27
kseries_parallel_xover3	1548 – 2157	71.41	54.63	23.50
kseries_parallel_xover4	1040 – 1514	53.13	40.88	23.06
Average				15.39
4-core				
consumer1	66327 – 95023	2424.69	2255.60	6.97
consumer2	55416 – 81112	1869.30	1638.26	12.36
telecom	539 – 1018	30.10	26.15	13.12
ATR	9474 – 14264	502.13	405.52	19.24
CNC	884 – 1277	43.00	34.61	19.52
image	223 – 308	11.12	9.31	16.24
kseries_parallel1	1059 – 1434	56.53	41.24	27.05
kseries_parallel2	561 – 1074	38.90	28.21	27.48
kseries_parallel3	1625 – 3707	148.94	111.64	25.04
kseries_parallel4	1228 – 3060	111.97	81.38	27.32
kseries_parallel_xover1	843 – 2493	83.07	64.74	22.07
kseries_parallel_xover2	695 – 970	36.37	27.45	24.53
kseries_parallel_xover3	1147 – 2056	76.96	55.02	28.51
kseries_parallel_xover4	1040 – 1514	53.09	40.49	23.73
Average				20.94

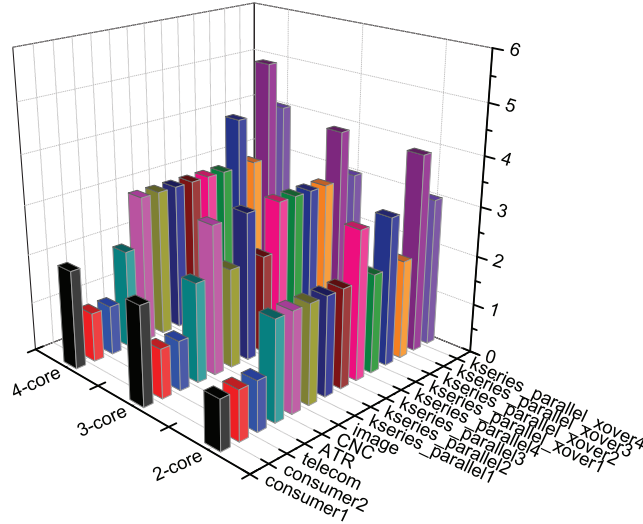


Figure 3.10. The Maximum Retiming Value R_{max} of Each Benchmark Running on 2, 3, and 4 Processor cores.

solutions under the same timing constraint as those for our approach, our approach can save extra energy consumption compared with FLSSR by helping it fully utilizing the empty time slots caused by intercore communication overheads. For each benchmark under different timing constraints, with the decreasing of the timing constraint, the energy reduction of our approach over FLSSR is accordingly increased. For each benchmark running on different processor cores with the same timing constraint, the energy consumption on 2 or 3 processor cores is greater than or equal to the energy consumption running on 4 processor cores. This result also shows that our approach can utilize multiple processor cores to minimize the energy consumption. From the above analysis, we can conclude that the proposed approach can achieve better energy consumption compared with the energy-efficient task scheduling algorithm FLSSR.

(3) *Prologue Length*. The observations in Section 3.3 point to the maximum retiming value among all tasks as a direct factor that influences the prologue length. In this section, we study the impact of prologue length (i.e., the maximum retiming value). Figure 3.10 shows the maximum retiming value R_{max} of each benchmark with 2, 3, and 4 processor cores, respectively. From the results, we can see that, our approach causes several periods of prologue latency. As discussed before, however, the prologue is only executed once. So the overhead

introduced is one-time delay. After waiting for the execution of the prologue, tasks can be periodically executed in the new loop kernel as a streaming application is usually repeatedly executed for many times. Since our approach can effectively reduce the schedule length of each period as shown above, we can either apply a shorter period or apply DVS for energy optimization. Considering the benefit obtained from each period after the prologue, it is usually worth waiting for the execution of the prologue.

3.6 Summary

This chapter studied the problem of minimizing intercore communication overhead for streaming applications running on MPSoC architectures. We jointly optimize computation and intercore communication task scheduling such that intercore communication overhead can be totally removed and the schedule length can be minimized. Specifically, we first performed schedulability analysis and theoretically obtained the upper bound on the prologue length of the computation task schedule. Then we presented an ILP (Integer Linear Programming) formulation to generate an optimal objective task schedule. The experimental results show that our technique can significantly reduce schedule length and energy consumption compared with representative techniques.

CHAPTER 4

MEMORY-AWARE SCHEDULING WITH COMMUNICATION OVERHEAD MINIMIZATION FOR STREAMING APPLICATIONS ON MPSOCS

4.1 Overview

Streaming applications that process streams of data are often modeled as periodic dependent tasks, in which streams of data are communicated from task to task [108, 111]. Streaming applications are data intensive and highly parallelizable; therefore, they are very suitable to be executed on Multiprocessor System-on-Chips (MPSoCs). To fully utilize the computation capacity of MPSoCs, various techniques have been explored to increase parallelism of streaming applications. However, this may cause a large amount of intercore communications with considerable communication overhead. Streaming applications often have firm real-time requirements. The communication overhead poses a challenge for multicore hard real-time systems, since most of the existing theoretically optimal scheduling techniques on multicore architectures assume zero cost for intercore communications. By removing intercore communication overhead, a shorter period can be applied and system performance (e.g., throughput) can be improved. Therefore, it becomes an important research problem to effectively reduce intercore communication overhead for streaming applications on MPSoCs.

In this chapter, we effectively remove intercore communication overhead and generate an optimal task schedule with the minimum memory usage for streaming applications on MPSoC architectures. Specifically, in the proposed technique, we jointly reschedule both computation and intercore communication tasks and let a limited number of tasks reschedule into earlier periods (the newly-added preprocessing step is called *prologue*). After transforming intra-period data dependencies into inter-period data dependencies, the execution of

computation tasks and that of intercore communication tasks in each period can be totally overlapped and the intercore communication overhead can be effectively removed.

Since streaming applications are data intensive, fairly large shared buffers would be required to store the processing results between tasks. As a result, total size of the buffer arrays usually accounts for a significant portion of the application binary memory footprint [31]. Our approach will generate an optimal task schedule with the maximum application throughput while minimizing the overall memory usage, which would be of great value in the resource constrained embedded multiprocessor systems. To the best of our knowledge, this is the first work that totally removes intercore communication overhead considering the memory usage with joint computation and communication task scheduling for streaming applications on MPSoC architectures.

To solve the problem, we first perform schedulability analysis and theoretically obtain the upper bound of the times needed to reschedule each computation task. Based on this analysis, we formulate the problem as an ILP (Integer Linear Programming) model and obtain an optimal solution with the objective of minimizing the overall memory usage. As the schedulability analysis produces very tight bounds, they can significantly reduce the search space of our ILP formulation and greatly improve the efficiency for finding the optimal solution. We also propose a heuristic approach Heuristic Memory-Aware Optimal Task Scheduling (HMAOTS) to efficiently obtain a near optimal solution.

We evaluate the proposed technique with a set of benchmarks from both real-life streaming applications and synthetic task graphs, including E3S (Embedded Systems Synthesis Benchmarks) [104], CNC (Computerized Numerical Control) [51], ATR (Automatic Target Recognition) [71], an image enhancement application [103], and TGFF [28]. We implement a simulator based on the processor model of ARM11 MPCore processor [10]. We compare the proposed technique with the approaches in [111, 115] and [106] in terms of schedule length and memory usage. Experimental results show that the proposed technique can achieve 14.71% and 12.25% reduction in schedule length compared with the approaches in [111] and [115], respectively; and also 15.98% and 32.45% reduction in memory usage

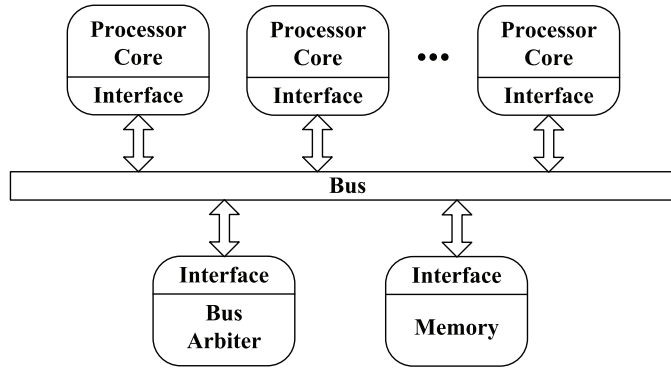


Figure 4.1. The MPSoC Architecture.

compared with the approaches in [106] and [111], respectively.

The rest of this chapter is organized as follows. Section 4.2 introduces models and concepts used in this chapter. Section 4.3 gives a motivational example and formally defines the problem. Section 4.4 proposes our optimal joint computation and communication task scheduling technique. Section 4.5 presents experimental results. Finally, we conclude the chapter in Section 4.6.

4.2 Models and Concepts

4.2.1 System Model

In this chapter, we consider a typical MPSoC system architecture shown in Figure 4.1. The MPSoC architecture consists of M homogenous processor cores $\{P_1, P_2, \dots, P_M\}$, a shared bus, a bus arbiter, and a shared on-chip memory. Every processor core is connected via the bus to a shared memory. A shared bus is adopted as it is one of the most widely used on-chip communication architectures. A shared on-chip memory is used for intercore communication. The accesses from a processor core to the shared memory are not cached. Bus access requests from processor cores are managed by the bus arbiter. This target architecture is a generic and typical infrastructure for new generation MPSoC architecture [94].

4.2.2 Application Model and Communication Overhead

In this chapter, streaming applications are modeled as periodic dependent tasks and represented by a Directed Acyclic Graph (DAG). For a DAG $G = (V, E, CT)$, $V = \{T_1, T_2, \dots, T_n\}$ is the node set, and each node represents one periodic task. $E \subseteq V \times V$ is the edge set, and each directed edge, $(T_i, T_j) \in E$ ($T_i, T_j \in V$), represents the data dependency between tasks T_i and T_j . That is, the execution of T_j needs the results generated by the execution of T_i . $CT : E \mapsto \mathbb{Z}$ is a function that associates every directed edge $(T_i, T_j) \in E$ with a communication task CT_j^i to denote the corresponding data transfer from task T_i to task T_j .

The data transfer between two dependent tasks involves different components in system architecture: processor cores, bus, and the shared on-chip memory. If two tasks with the data dependency are assigned to different processor cores, an intercore communication is issued and the shared on-chip memory is used to store the intermediate communication. The processor core is able to initiate write operations to shared on-chip memory by providing an address and control information, which normally takes one bus clock cycle. The communication latency or *communication overhead* is the length of time incurred in communicating a message containing a number of words from a source processor core to a target processor core. This is the time overhead for a message to cross through the bandwidth bottleneck in the bus system.

According to the property of the shared bus, only one component in system architecture (i.e., processor core, on-chip memory) is allowed to actively use the bus at any one time. The shared bus has a finite capacity or communication bandwidth B . Therefore, to transmit $\mathcal{D}(T_i, T_j)$ amount of data volume from task T_i to task T_j , the communication overhead is $\lceil \mathcal{D}(T_i, T_j) / B \rceil$. On-chip memory will allocate the memory space to hold the intermediate data. The required memory space is released until the target processor core signals back to the bus arbiter the success of the data transfer.

4.2.3 Static Schedule

For a task schedule, let p be the period of each computation task and that of each intercore communication task. p implies the deadline of the schedule, and the schedule must be finished in p . Let S_i be the release time (start time) of task T_i in the first period. For task T_i in the ℓ th period (i.e., $T_{i,\ell}$), the release time of $T_{i,\ell}$ is $S_{i,\ell} = S_i + (\ell - 1) \cdot p$, $\ell \geq 1$. Similarly, for a communication task CT_j^i associated with tasks T_i and T_j , let S_j^i be the release time of task CT_j^i in the first period, then its release time in the ℓ th period is $S_{j,\ell}^i = S_j^i + (\ell - 1) \cdot p$, $\ell \geq 1$.

Let c_i be the execution time of computation task T_i , and c_j^i denotes the execution time of communication task CT_j^i . c_j^i is determined by the assignment of tasks T_i and T_j : if T_i and T_j are assigned to the same processor core, it is an intracore communication and we assume c_j^i is equal to zero; otherwise, it is an intercore communication and c_j^i is equal to $\lceil \mathcal{D}(T_i, T_j) / B \rceil$, where $\mathcal{D}(T_i, T_j)$ is the data volume transferred and B is the bus bandwidth.

A schedule must obey all data dependency relations of a DAG. For each directed edge $(T_i, T_j) \in E$ in the DAG, task T_j has data dependency with task T_i . In each period of the schedule, the execution of computation task T_j has to wait the completion of computation task T_i and the corresponding communication task CT_j^i .

4.2.4 Task Rescheduling and Retiming

In this chapter, we utilize the feature of periodic dependent tasks and reschedule several tasks into previous periods so as to overlap the execution of computation tasks and that of intercore communication tasks. After rescheduling tasks into previous periods, data dependency relations are changed across different periods, and the newly-added preprocessing step is called *prologue*. In this chapter, we apply retiming technique [61] to describe how many periods of a task node are rescheduled into the prologue and how this influences data dependency relations for a given DAG. The retiming technique is originally proposed to minimize the cycle period of a synchronous circuit by evenly distributing registers [61]. We extend it and

redefine it as follows:

Definition 4.2.1. (Retiming) Given a DAG $G = (V, E, CT)$, retiming $\mathcal{R} : V \mapsto \mathbb{Z}$ is a function that maps each node $T_i, T_i \in V$, to an integer $\mathcal{R}(T_i)$. Retiming function represents extended data dependency relations. By retiming T_i once, if it is legal, one period of task T_i is rescheduled into the prologue.

Definition 4.2.2. (Retiming value) Given a DAG $G = (V, E, CT)$, the retiming value $\mathcal{R}(T_i)$ of a computation task $T_i, T_i \in V$, denotes the number of periods of task T_i that are rescheduled into the prologue; similarly, the retiming value $\mathcal{R}(CT_j^i)$ of an intercore communication task $CT_j^i, (T_i, T_j) \in E (T_i, T_j \in V)$, denotes the number of periods of task CT_j^i that are rescheduled into the prologue.

Definition 4.2.3. (Legal retiming) Given a DAG $G = (V, E, CT)$, for each edge $(T_i, T_j) \in E (T_i, T_j \in V)$, a retiming function \mathcal{R} is legal if the relative retiming value $\mathcal{R}(T_i) - \mathcal{R}(T_j)$ is non-negative.

A retiming function must be legal in order to preserve the semantic correctness. If $\mathcal{R}(T_i) - \mathcal{R}(T_j)$ is negative, it implies that the data generated by T_i in current period is needed in order to execute T_j in previous periods. This cannot be feasible in a task schedule.

Initially, all tasks are scheduled inside one period and it represents an *intra-period dependency*, and the retiming value is zero. After performing retiming, tasks from different periods are regrouped into one period in order to overlap the execution of computation and intercore communication tasks. Then, a non-negative retiming value represents the data dependency relations across multiple periods (*inter-period dependency*).

4.3 Motivational Example and Problem Statement

Given an initial schedule with intercore communication overhead, the objective is to totally remove the intercore communication overhead and generate a schedule with the minimum memory usage. By removing the intercore communication overhead, the system performance can be improved by adopting a smaller period and the throughput is improved. On the other hand, a longer prologue not only introduces more delays in the beginning, but also it requires more data buffers to hold intercore communication data. So we want to minimize the memory usage as well.

A DAG is shown in Figure 4.2(a). In the DAG, there are four tasks and the execution time of each task is 2 time units. There are four edges, and each edge associates with one communication task. The execution time of each intercore communication task is 1 time unit and that of each intracore communication task is zero.

The intercore communication overhead poses a challenge for multicore real-time systems since theoretically optimal scheduling techniques on multicore architectures mostly assume zero cost for intercore communications. Based on this assumption, the optimal computation task schedule can be generated as the one shown in Figure 4.2(b). Let the period p be 6 time units. The schedule is repeatedly executed in each period and the schedule length (the total execution time of a schedule) of one period is 6.

Although the task schedule in Figure 4.2(b) is the optimal computation task schedule in terms of the minimum schedule length, this does not hold true in context of intercore communication overheads. Since intercore communication overheads may not be negligible, the occurrence of intercore communication overheads will lead to a longer schedule length. This may compromise the predictability of existing theoretically optimal scheduling techniques.

Figure 4.2(c) shows the schedule that considers the intercore communication overhead. In this task schedule, task T_3 can start execution only after the completion of task T_1 and intercore communication task CT_3^1 . As a result, intercore communication task CT_3^1 introduces the overhead (the latency of execution) for one time unit. From this example, we can see that the intercore communication (communication tasks CT_3^1 and CT_4^3) introduces

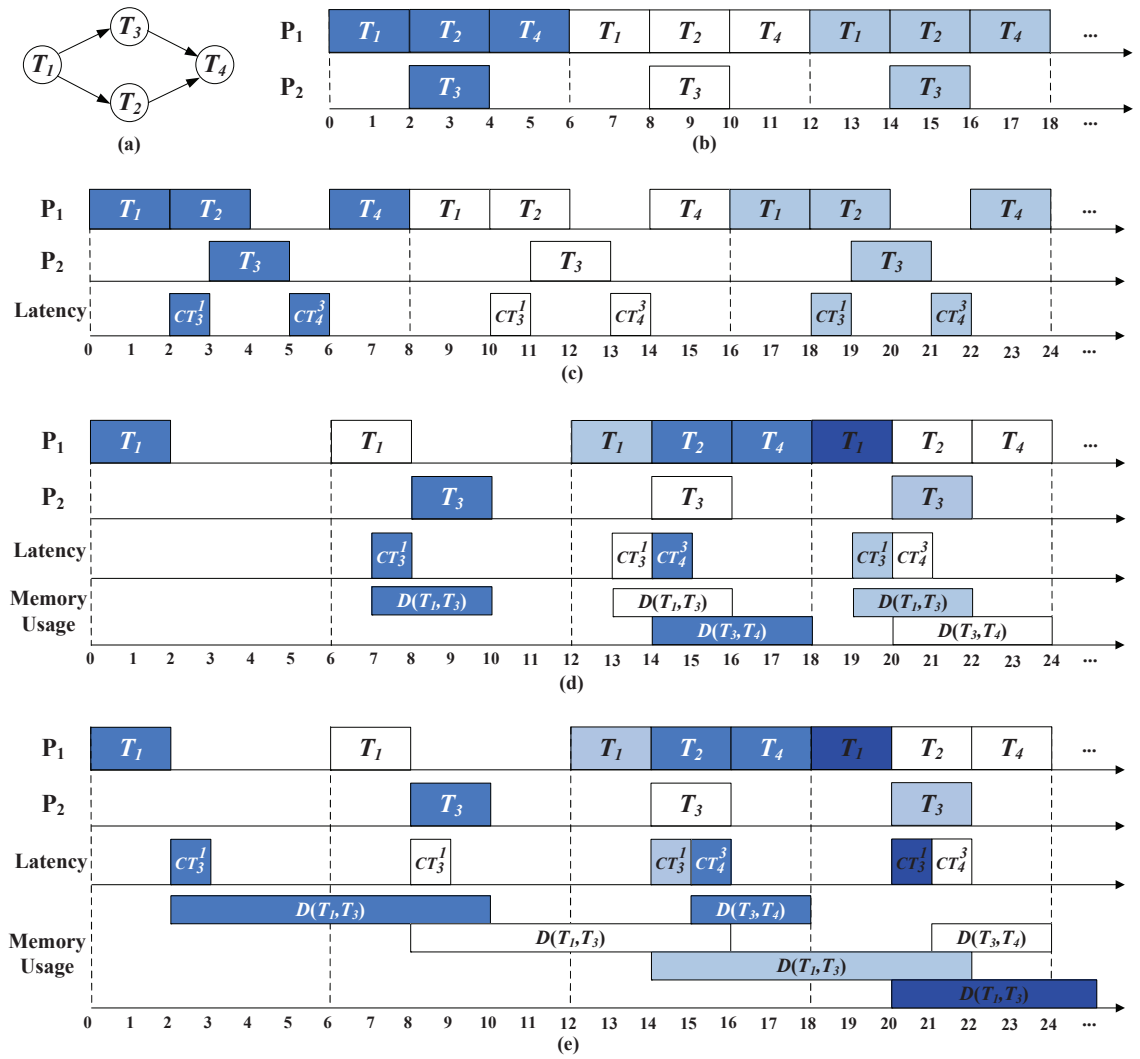


Figure 4.2. A Motivational Example. (a) A DAG. (b) The objective computation task schedule of the DAG. (c) The schedule considering intercore communication overhead. (d) and (e) two schedules in which intercore communication overheads are totally removed while they are with different memory usages.

big overhead (2 time units) such that the schedule length is increased and a long period (8 time units) has to be used correspondingly.

The computation task schedule that totally removes the effects of intercore communication overhead is denoted as *the objective computation task schedule*. Given a DAG and an initial schedule with intercore communication overheads, the objective computation task

schedule can be generated by rescheduling each computation task with earlier release time. The generation of the objective computation task schedule considers the data dependency in the DAG while assuming that the execution time of each communication task is equal to zero. For example, given a DAG and its initial schedule in Figure 4.2(a) and (c), the objective computation task schedule can be generated like the task schedule in Figure 4.2(b).

In order to ensure the objective computation task schedule as the one in Figure 4.2(b), intercore communication tasks have to be rescheduled. We found this can be achieved by overlapping the execution of intercore communication and computation tasks from different periods with joint communication and computation task rescheduling. This is illustrated in Figure 4.2(d).

Given the DAG in Figure 4.2(a) and the objective computation task schedule in Figure 4.2(b), as each task is periodic, we reschedule two periods of task T_1 and one period of task T_3 into the prologue. In Figure 4.2(d), by adopting the retiming technique, the data required by task T_3 from task T_1 are always available inside a period, and the intercore communication overhead caused by communication task CT_3^1 can be removed inside the schedule. After adopting the processing step, the schedule length is reduced from 8 time units to 6 time units, and the objective computation task schedule in Figure 4.2(b) is guaranteed.

From this example, we can also see that the objective computation task schedule of one period can be obtained by removing all the intercore communication overhead and rescheduling each computation task as early as possible. However, it is not trivial to determine the retiming value (how many periods of one task rescheduled into the prologue) of each computation task. The problem is difficult as we cannot achieve this by rescheduling tasks into the prologue freely without considering the constraint of memory usage. And it is not easy to reschedule intercore communication tasks to ensure the schedule length is minimized with the minimum memory usage. For example, although task schedules in Figure 4.2(d) and (e) have the same schedule for computation tasks, the memory usage of these two schedules are different.

For the schedule in Figure 4.2(d), in one period of time, it needs to store one period of

data volume for communication task CT_3^1 and one period of data volume for communication task CT_4^3 . Compared with the schedule in Figure 4.2(d), the schedule in Figure 4.2(e) needs an extra communication buffer to store one more period of data volume for communication task CT_3^1 in the schedule. This example illustrates that, it is necessary to obtain a task schedule that totally removes intercore communication overheads while minimizing the memory usage to store the data transfers of streaming applications.

Based on the models and problem analysis, we further formulate the problem as follows:

Given a DAG $G = (V, E, CT)$, an MPSoC architecture with M processor cores, and an initial task schedule $Init_Sch$ of the DAG G , how to jointly reschedule computation tasks and intercore communication tasks such that the objective computation task schedule with the minimum schedule length can be generated with the minimum extra memory usage?

4.4 Memory-Aware Task Scheduling for Minimizing the Intercore Communication Overhead

In this section, we present a Memory-Aware Optimal Task Scheduling (MAOTS) approach that jointly optimizes computation and communication task schedule. We will first analyze the bounds of the relative retiming value in Section 4.4.1, and we analyze the extra memory usage caused by retiming operations in Section 4.4.2. The results of the analysis will be integrated into the constraints of the integer linear programming (ILP) model in Section 4.4.3. The ILP model is provided to obtain the optimal solution for the problem of removing intercore communication overheads with the objective of minimizing extra memory usage caused by retiming. As these constraints hold very tight bounds, they can greatly reduce the search space of ILP model and significantly improve the efficiency for finding the optimal solution. Based on the analysis, in Section 4.4.4, we also formulate the target problem into several subproblems and propose a heuristic approach (HMAOTS) to generate a near optimal solution.

4.4.1 The Bounds Analysis of Retiming Value

The relative retiming value of each pair of computation tasks represents the number of periods involved to guarantee the schedulability of the associated intercore communication task. Specifically, for a pair of computation tasks T_i and T_j with data dependency relations in the DAG, $(T_i, T_j) \in E$ ($T_i, T_j \in V$), we use $\hat{\mathcal{R}}_{min}(T_i, T_j)$ and $\hat{\mathcal{R}}_{max}(T_i, T_j)$ to denote that, relative to the retimed task T_j , the minimum and the maximum extra number of times to perform retiming for task T_i to ensure the schedulability of the associated intercore communication task CT_j^i .

Based on the objective computation task schedule generated from the initial schedule, we propose the following theorem and properties to analyze the schedulability of each communication task and get the minimum and the maximum relative retiming value of each pair of computation tasks. The derived relative retiming values can be used to get the bound of the retiming value of each task, and they will become the constraints of the integer linear programming formulation.

Property 4.4.1. *If computation tasks T_i and T_j , $(T_i, T_j) \in E$ ($T_i, T_j \in V$), are assigned to the same processor core, it is an intracore communication. Then, $\hat{\mathcal{R}}_{min}(T_i, T_j) = \hat{\mathcal{R}}_{max}(T_i, T_j) = 0$.*

Property 4.4.1 gives the case for the pair of computation tasks T_i and T_j that are assigned to the same processor core. In this case, there is no need to perform retiming operations to reschedule computation task T_i relative to computation task T_j . For computation tasks that are mapped to different processor cores, the following theorem gives the upper bound of the relative retiming value of each pair of tasks.

Theorem 4.4.1. *For a pair of computation tasks T_i and T_j , $(T_i, T_j) \in E$ ($T_i, T_j \in V$), assigned to different processor cores, computation tasks T_i and T_j associate with an intercore communication task CT_j^i in the l th period of the objective computation task schedule. After retiming task T_i for $\mathcal{R}(T_i)$ times and retiming task T_j for $\mathcal{R}(T_j)$ times, as long as the retimed*

task $T_{i,\ell-\mathcal{R}(T_i)}$ is rescheduled at most one more period relative to the retimed task $T_{j,\ell-\mathcal{R}(T_j)}$, the associated intercore communication task CT_j^i is schedulable on bus during the time span between the finishing time of the retimed task $T_{i,\ell-\mathcal{R}(T_i)}$ and the release time of the retimed task $T_{j,\ell-\mathcal{R}(T_j)}$.

Proof. The objective computation task schedule obeys the data dependency relations in the DAG. In the ℓ th period of the objective computation task schedule, the finishing time of task T_i is no later than the release time of task T_j . After retiming task T_i for one time relative to the retimed task $T_{j,\ell-\mathcal{R}(T_j)}$, task T_i will be scheduled in period $\ell - \mathcal{R}(T_j) - 1$, which is one period ahead of the retimed task $T_{j,\ell-\mathcal{R}(T_j)}$. Then the time span between the finishing time of the retimed task $T_{i,\ell-\mathcal{R}(T_i)}$ and the release time of the retimed task $T_{j,\ell-\mathcal{R}(T_j)}$ is always greater than or equal to period p . As all intercore communication tasks periodically execute in each period, in one period of time, there is one and exactly one intercore communication task CT_j^i that has data dependency with task T_i and task T_j . Let this intercore communication task CT_j^i be the retimed communication task that associates with tasks $T_{i,\ell-\mathcal{R}(T_i)}$ and $T_{j,\ell-\mathcal{R}(T_j)}$. Its release time is no earlier than the finishing time of $T_{i,\ell-\mathcal{R}(T_i)}$, and its finishing time is no later than the release time of $T_{j,\ell-\mathcal{R}(T_j)}$. Therefore, the associated intercore communication task CT_j^i is always schedulable on bus during that time span. An example is illustrated in Figure 4.3. □

Theorem 4.4.1 gives the upper bound of the maximum relative retiming value of each pair of computation tasks, $\hat{\mathcal{R}}_{max}(T_i, T_j) \leq 1$, $(T_i, T_j) \in E$ ($T_i, T_j \in V$). This tight constraint provides a good property for rescheduling each computation task. This upper bound also implies that the maximum latency introduced by the proposed retiming technique is in one period of time, which can ensure the basic functionality requirements for most of commercial available streaming applications. For the lower bound of the relative retiming value, the

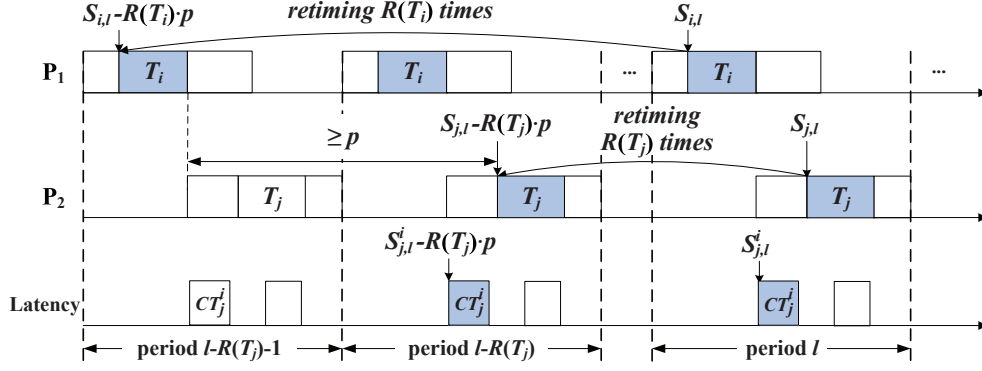


Figure 4.3. A Task Schedule of Theorem 4.4.1.

property of legal retiming constraints that the relative retiming value of each pair of computation tasks is non-negative, $\mathcal{R}(T_i) - \mathcal{R}(T_j) \geq 0$. Then, the lower bound of the relative retiming value of each pair of computation tasks is zero. Based on these analysis, the bounds of the relative retiming value is $[0, 1]$. That is, $0 \leq \hat{\mathcal{R}}_{min}(T_i, T_j) \leq \hat{\mathcal{R}}_{max}(T_i, T_j) \leq 1$.

Property 4.4.2. For a pair of computation tasks T_i and T_j , $(T_i, T_j) \in E$ ($T_i, T_j \in V$), assigned to different processor cores, computation tasks T_i and T_j associate with an intercore communication task CT_j^i . If $S_i + c_i + c_j^i > S_j$, task T_i needs to retiming once and exactly once relative to task T_j to ensure the schedulability of communication task CT_j^i . That is, $\hat{\mathcal{R}}_{min}(T_i, T_j) = \hat{\mathcal{R}}_{max}(T_i, T_j) = 1$.

Property 4.4.2 illustrates the case that the time span between the finishing time of task T_i and the release time of task T_j is not sufficient to hold the execution of communication task CT_j^i . In this case, the retimed task $T_{i, \ell - \mathcal{R}(T_i)}$ has to rescheduled at least one more period ahead of the retimed task $T_{j, \ell - \mathcal{R}(T_j)}$. According to Theorem 4.4.1, the maximum relative retiming value is bounded by one. Therefore, the minimum and the maximum relative retiming value of the case in Property 4.4.2 are both equal to one.

The above theorem and properties classify the minimum and the maximum relative retiming value $(\hat{\mathcal{R}}_{min}(T_i, T_j), \hat{\mathcal{R}}_{max}(T_i, T_j))$ of computation tasks T_i and T_j , $(T_i, T_j) \in$

$E (T_i, T_j \in V)$, into three cases: $[0, 0]$, $[0, 1]$, and $[1, 1]$. For computation tasks T_i and T_j associated with communication task CT_j^i , the minimum and the maximum relative retiming value can be obtained by checking the above theorem and properties. The derived relative retiming values can be used to obtain the bounds of the retiming value of each computation task.

We adopt a breadth-first manner that follows the data dependency relations in the DAG and obtain the minimum and the maximum retiming value. For a task that does not have any successor tasks, there is no need to reschedule these tasks into the prologue. Therefore, both its minimum and its maximum retiming value are equal to zero.

$$\mathcal{R}_{min}(T_i) = \mathcal{R}_{max}(T_i) = 0, \quad T_i \in E$$

For a task with at least one successor tasks, the retiming value of this task is bounded by the maximum retiming value of its successor tasks. The following equation is used to obtain the minimum and the maximum retiming value of a task that has at least one successor tasks.

$$\begin{cases} \mathcal{R}_{min}(T_i) = \max\{\hat{\mathcal{R}}_{min}(T_i, T_j) + \mathcal{R}_{min}(T_j)\} \\ \mathcal{R}_{max}(T_i) = \max\{\hat{\mathcal{R}}_{max}(T_i, T_j) + \mathcal{R}_{max}(T_j)\} \end{cases} \quad \forall (T_i, T_j) \in E (T_i, T_j \in V)$$

4.4.2 The Analysis of Extra Memory Usage

After obtained the bounds of retiming value of each computation task, we further analyze how the retiming value of each task influences the memory usage to store the associated intercore communication tasks. For the target multiprocessor architecture adopted in this chapter, the shared memory is accessed by multiple processor cores with intent to provide intermediate data storage for the input/output data streams among different processor cores. Data streams that are processed by streaming applications are stored in the dedicated memory space in the shared memory, such that the streaming application can fetch the data for processing.

Data streams transferred between streaming applications are modeled as an intercore communication task CT_j^i , and the data volume $\mathcal{D}(T_i, T_j)$ of intercore communication task

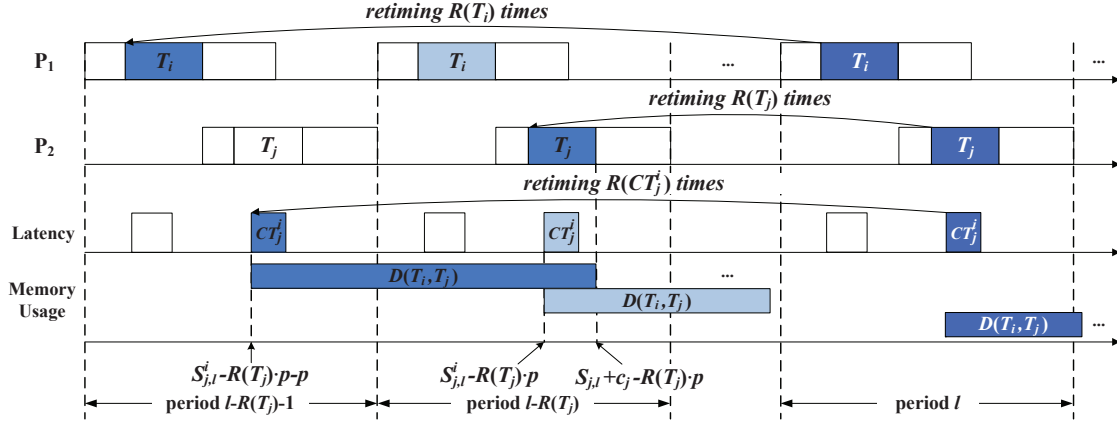


Figure 4.4. Analysis of Memory Usage for Intercore Communication Task CT_j^i .

CT_j^i will directly account for a portion of the memory usage. We use the following property to analyze data volume transferred between a pair of retimed computation tasks.

Property 4.4.3. For a pair of computation tasks T_i and T_j , $(T_i, T_j) \in E$ ($T_i, T_j \in V$), associated with an intercore communication task CT_j^i , the data volume for transferring one period of intercore communication task CT_j^i is $\mathcal{D}(T_i, T_j)$. After retiming task T_i for $\mathcal{R}(T_i)$ times and retiming task T_j for $\mathcal{R}(T_j)$ times, the total data volume transferred between the retimed tasks T_i and T_j is $\lceil \frac{1}{p} \cdot (S_j + c_j - S_j^i) + \mathcal{R}(CT_j^i) - \mathcal{R}(T_j) \rceil \cdot \mathcal{D}(T_i, T_j)$.

The buffer to store data volume $\mathcal{D}(T_i, T_j)$ is allocated from the starting time of intercore communication task CT_j^i , and it is released when task T_j finishes its execution. If the lifetime of $\mathcal{D}(T_i, T_j)$ crosses different periods, it will cause extra memory usage to store the data volume for intercore communication tasks. If communication task CT_j^i is rescheduled to period $\ell - \mathcal{R}(CT_j^i)$ and task T_j is rescheduled to period $\ell - \mathcal{R}(T_j)$, the data volume will across at most $\mathcal{R}(CT_j^i) - \mathcal{R}(T_j)$ periods.

In order to minimize the extra memory space, communication task CT_j^i should be rescheduled as close as possible to computation task T_j . More specifically, as illustrated in Figure 4.4, if the finishing time of the retimed task $T_{j,\ell}$ is later than the starting time

of the retimed communication task $CT_{j,\ell}^i$ of the next period, i.e., $S_{j,\ell}^i - \mathcal{R}(T_j) \cdot p + p < S_{j,\ell} + c_j - \mathcal{R}(T_j) \cdot p$, an extra memory space is necessary. In this case, $S_{j,\ell} + c_j - S_{j,\ell}^i > p$. This property implies that, as long as communication task CT_j^i is released within the period $[S_{j,\ell} + c_j - p, S_{j,\ell} + c_j]$, no extra memory usage is consumed.

4.4.3 Integer Linear Programming Formulation

Integer linear programming (ILP) provides a mechanism to get the optimal solution of a problem in which all of its constraints can be formulated as linear constraints of integer variables. In this section, an integer linear programming model is provided to obtain the optimal solution for the problem of removing the intercore communication overhead with the objective of minimizing memory usage.

Given a DAG and its initial schedule, we can formulate an integer linear programming model as follows.

$$\text{Minimize } \sum_{(T_i, T_j) \in E} \left\lceil \frac{1}{p'} \cdot (S_j + c_j - S_j^i) + \mathcal{R}(CT_j^i) - \mathcal{R}(T_j) \right\rceil \cdot \mathcal{D}(T_i, T_j)$$

Subject to

$$\left\{ \begin{array}{l} \forall CT_j^i, CT_{j'}^{i'} \in CT, S_j^i \leq S_{j'}^{i'} : \\ \quad S_j^i \geq 0 \quad (1) \\ \quad S_j^i \leq p' - c_j^i \quad (2) \\ \quad S_{j'}^{i'} - S_j^i \geq c_j^i \quad (3) \\ \forall CT_j^i \in CT, (T_i, T_j) \in E, (T_i, T_j \in V) : \\ \quad \mathcal{R}(T_i) - \mathcal{R}(T_j) \geq \hat{\mathcal{R}}_{min}(T_i, T_j) \quad (4) \\ \quad \mathcal{R}(T_i) \geq \mathcal{R}_{min}(T_i) \quad (5) \\ \quad \mathcal{R}(T_i) \leq \mathcal{R}_{max}(T_i) \quad (6) \\ \quad \mathcal{R}(T_i) - \mathcal{R}(CT_j^i) \geq 0 \quad (7) \\ \quad \mathcal{R}(T_j) - \mathcal{R}(CT_j^i) \leq 0 \quad (8) \\ \quad \mathcal{R}(T_i) - \mathcal{R}(CT_j^i) \leq 1 \quad (9) \\ \quad S_j^i + \mathcal{R}(T_i) \cdot p' - \mathcal{R}(CT_j^i) \cdot p' \geq S_j + c_j \quad (10) \\ \quad S_j^i + \mathcal{R}(T_j) \cdot p' - \mathcal{R}(CT_j^i) \cdot p' \leq S_j - c_j^i \quad (11) \end{array} \right.$$

Due to the data dependency relations in the DAG, the retiming value $\mathcal{R}(CT_j^i)$ of each communication task CT_j^i is bounded by the retiming value of tasks T_i and T_j , $\mathcal{R}(T_i) \geq \mathcal{R}(CT_j^i) \geq \mathcal{R}(T_j)$. The retiming value of each computation task is further bounded by its

minimum and the maximum retiming value ($\mathcal{R}_{min}(T_i), \mathcal{R}_{max}(T_i)$). These linear constraints can be used to formulate the ILP model to solve the problem.

In the ILP model, the objective function seeks to minimize the overall memory usage to store intercore communications. Let \mathcal{L} be the schedule length of the objective computation task schedule that is derived from the initial task schedule, and $\sum c_j^i, (T_i, T_j) \in E$, denotes the sum of the execution time of all communication tasks. Then the new period p' for the optimal task schedule can be obtained, $p' = \max\{\mathcal{L}, \sum c_j^i\}$.

Given a set of intercore communication tasks, Constraint 1 and Constraint 2 illustrate that, in each period p' , the release time S_j^i of a communication task CT_j^i should be in the time span $[0, p' - c_j^i]$. Constraint 3 restricts that every two intercore communication tasks scheduled on a shared bus are allocated without any conflict. That is, when a communication task CT_j^i has been scheduled to release at S_j^i , no other communication task $CT_{j'}^{i'}$ can start its execution only after the completion of communication task CT_j^i . Constraint 4 gives the lower bound of the relative retiming value of each pair of tasks, which can be obtained by the analysis in Section 4.4.1. Constraint 5 and Constraint 6 ensure that the retiming value of each task is bounded by the minimum and the maximum retiming value, which can be obtained by the analysis in Section 4.4.1. Constraint 7 and Constraint 8 preserve the semantic correctness to keep the retiming function legal. Constraint 9 illustrates that communication task CT_j^i has to be allocated to shared memory before the finishing of the next period's retimed task T_j . Constraint 10 and Constraint 11 follow the data dependencies to ensure that communication task CT_j^i is schedulable between the finishing time of the retimed task T_i and the release time of retimed task T_j .

Using this ILP formulation, we can obtain the retiming value $\mathcal{R}(T_i)$ of each computation task T_i , the retiming value $\mathcal{R}(CT_j^i)$ of each intercore communication task CT_j^i , and the release time S_j^i of each intercore communication task CT_j^i . Based on these results derived from the ILP model, an optimal task schedule can be generated.

We use a runtime example in Figure 4.5 to illustrate our approach. Based on the initial schedule in Figure 4.5(b) and the objective computation task schedule in Figure 4.5(c), we

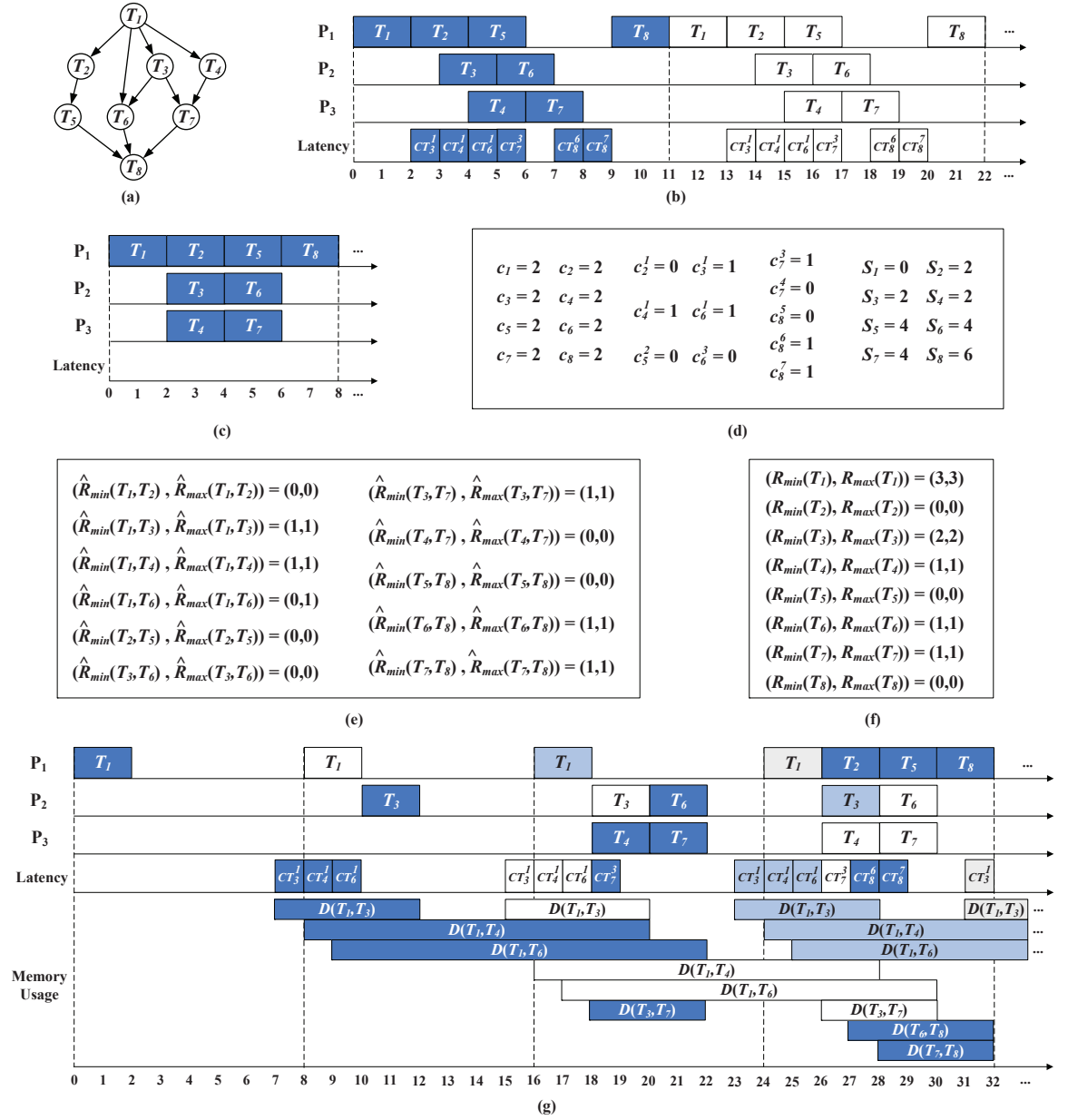


Figure 4.5. A Runtime Example of MAOTS. (a) A DAG. (b) An initial schedule of the DAG. (c) Objective task schedule of all computation tasks that totally removes intercore communication overhead. (d) The execution time of each computation task and that of each communication task, and the release time of each computation task. (e) The bounds of relative retiming value of each pair of tasks. (f) The bounds of the retiming value of each task. (g) The optimal task schedule by our approach MAOTS.

can obtain the execution time c_i of each computation task T_i , the execution time c_j^i of each communication task CT_j^i , and the release time S_i of each computation task T_i , which are shown in Figure 4.5(d). Then, by checking Theorem 4.4.1, Property 4.4.1 and Property 4.4.2, the minimum and the maximum relative retiming value of each pair of computation tasks can be obtained. For example, task T_1 and task T_6 are assigned to different processor cores (processor core P_1 and P_2 , respectively). They associate with an intercore communication task. Then, $(\hat{\mathcal{R}}_{min}(T_1, T_6), \hat{\mathcal{R}}_{max}(T_1, T_6))=(0, 1)$. For tasks with data dependency relations assigned to the same processor core (i.e., tasks T_1 and T_2), $(\hat{\mathcal{R}}_{min}(T_i, T_j), \hat{\mathcal{R}}_{max}(T_i, T_j))=(0, 0)$.

Using the minimum and the maximum relative retiming value derived in Figure 4.5(e), the bounds of the retiming value of each task can be obtained. For a task that does not have any successor tasks, (i.e., task T_8), both its minimum and its maximum retiming value are equal to zero. For a task with at least one successor tasks, the maximum value of $\hat{\mathcal{R}}_{min}(T_i, T_j) + \mathcal{R}_{min}(T_j)$ will be assigned to $\mathcal{R}_{min}(T_i)$. For example, task T_3 has two successor tasks, T_6 and T_7 . The maximum value between $\hat{\mathcal{R}}_{max}(T_3, T_6) + \mathcal{R}_{max}(T_6)$ and $\hat{\mathcal{R}}_{max}(T_3, T_7) + \mathcal{R}_{max}(T_7)$ is 2. Thus, $\mathcal{R}_{max}(T_3)$ is 2. Similarly, the maximum of $\hat{\mathcal{R}}_{min}(T_i, T_j) + \mathcal{R}_{min}(T_j)$ will be assigned to $\mathcal{R}_{min}(T_i)$. Therefore, $\mathcal{R}_{min}(T_3)$ is 2. Then the minimum and the maximum retiming value of task T_3 are both equal to 2. Follow the data dependency relations in the DAG (from task T_8 to task T_1), we can iteratively get the minimum and the maximum retiming value of each task. The results are listed in Figure 4.5(f). The results of the bound analysis will become the constraints of the ILP formulation in the next section. As the schedulability analysis produces very tight bounds, the ILP formulation can be solved efficiently in practice.

To illustrate how to generate the optimal task schedule, we continue to use Figure 4.5 as an example. It includes the generation of task schedule for computation tasks and the generation of task schedule for intercore communication tasks. The task schedule of computation tasks can be generated based on the objective computation task schedule in Figure 4.5(c) and the derived retiming value $\mathcal{R}(T_i)$ of each computation task T_i . For example, the ILP formulation gets the retiming value of task T_1 , $\mathcal{R}(T_1) = 3$. Then the first release of task T_1 will be rescheduled three periods ahead of the first period of the task schedule, and

three periods of task T_1 will be allocated to the prologue. For intercore communication tasks, the ILP formulation can obtain the release time S_j^i and retiming value $\mathcal{R}(CT_j^i)$ of each intercore communication task CT_j^i . Similar to the generation of task schedule of computation tasks, $\mathcal{R}(CT_j^i)$ periods of an intercore communication task is rescheduled into the prologue. For example, the ILP formulation obtains the release time of intercore communication task CT_3^1 , $S_3^1 = 0$, and gets the retiming value of CT_3^1 , $\mathcal{R}(CT_3^1) = 2$. In the task schedule in Figure 4.5(g), we can see that the first release of communication task CT_3^1 is two periods ahead of the first period of the objective schedule. After that, communication task CT_3^1 is periodically executed in each of the following periods. In Figure 4.5(g), intercore communication overhead is totally removed and a shorter period (8 time units) is applied with the minimum memory usage.

4.4.4 A Heuristic Approach

In this section, we present a heuristic approach (HMAOTS) to solve the memory-aware task scheduling problem. HMAOTS consists of three steps. In the first step, in Algorithm 4.4.1, communication tasks are classified to two different groups and inserted into two queues Q_1 and Q_2 , respectively. Tasks in Q_1 will influence the maximum retiming value and the memory usage of the task schedule, while tasks in Q_2 will not influence the memory usage. Communication tasks in each queue are sorted into the monotonically increasing order by deadline. Tasks in Q_1 have higher priority than tasks in Q_2 , and tasks in Q_1 can preempt tasks in Q_2 . In the second step, in Algorithm 4.4.2, empty time slots created by allocating tasks in Q_1 could be used to schedule tasks in Q_2 , and tasks in Q_2 will be scheduled in each time slot based on Best Fit Decreasing algorithm [46]. In the last step, based on the release time and execution time of each task, Algorithm 4.4.3 obtains the retiming value of each task and generate the final task schedule.

For the first step, in lines 1-12 of Algorithm 4.4.1, the weight $W(CT_j^i)$ of each task CT_j^i is calculated and the deadline of task is assigned based on the weight. For tasks with the zero weight, i.e., $\mathcal{R}_{max}(T_i) = \mathcal{R}_{min}(T_i)$ and $\mathcal{R}_{max}(T_j) = \mathcal{R}_{min}(T_j)$, retiming values

Algorithm 4.4.1 A Heuristic Approach for Memory-Aware Optimal Task Scheduling (HMAOTS)

Input: A set of communication tasks in Q_1 and Q_2 , period p .

Output: The task schedule $Schedule_Q$.

```

1: for each communication task  $CT_j^i$  do
2:   if  $\mathcal{R}_{max}(T_i) > \mathcal{R}_{max}(T_j) > \mathcal{R}_{min}(T_i) > \mathcal{R}_{min}(T_j)$  then
3:      $W(CT_j^i) \leftarrow (\mathcal{R}_{max}(T_i) - \mathcal{R}_{min}(T_j)) \cdot \mathcal{D}(T_i, T_j)$ ,  $d_j^i \leftarrow S_j$ ,  $S_j^i \leftarrow S_i + c_i$ ,
       INSERT( $Q_1$ ,  $CT_j^i$ ).
4:   else
5:      $W(CT_j^i) \leftarrow (\mathcal{R}_{max}(T_i) - \mathcal{R}_{min}(T_i) + \mathcal{R}_{max}(T_j) - \mathcal{R}_{min}(T_j)) \cdot \mathcal{D}(T_i, T_j)$ .
6:     if  $\mathcal{R}_{max}(T_i) = \mathcal{R}_{min}(T_i)$  and  $\mathcal{R}_{max}(T_j) = \mathcal{R}_{min}(T_j)$  then
7:        $d_j^i \leftarrow p$ ,  $S_j^i \leftarrow S_i + c_i$ , INSERT( $Q_2$ ,  $CT_j^i$ ).
8:     else
9:        $d_j^i \leftarrow S_j$ ,  $S_j^i \leftarrow S_i + c_i$ , INSERT( $Q_1$ ,  $CT_j^i$ ).
10:    end if
11:  end if
12: end for
13: while  $Q_1 \neq \emptyset$  do
14:   ReturnValue  $\leftarrow$  Schedule( $t$ ,  $CT_j^i$ )
15:   if ReturnValue = Success then
16:     Continue.
17:   else
18:     if  $W(CT_j^i) > W(CT_{j'}^{i'})$  then
19:        $t \leftarrow t - c_{j'}^{i'}$ , ReturnValue  $\leftarrow$  Schedule( $t$ ,  $CT_j^i$ )
20:       if ReturnValue = Success then
21:         INSERT( $Q_2$ ,  $CT_{j'}^{i'}$ ), Continue.
22:       else
23:          $d_j^i \leftarrow S_j + c_j$ , ReturnValue  $\leftarrow$  Schedule( $t$ ,  $CT_j^i$ )
24:         if ReturnValue = Success then
25:           INSERT( $Q_2$ ,  $CT_{j'}^{i'}$ ), Continue.
26:         else
27:            $t \leftarrow t + c_{j'}^{i'}$ , DELETE( $Q_1$ ,  $CT_j^i$ ), INSERT( $Q_2$ ,  $CT_j^i$ ), Continue.
28:         end if
29:       end if
30:     end if
31:   end if
32: end while
33: for each communication task  $CT_j^i$  in  $Q_2$  do
34:    $S_j^i \leftarrow t$ ,  $t \leftarrow t + c_j^i$ , DELETE( $Q_2$ ,  $CT_j^i$ ).
35: end for
36: GetRetimingValue().

```

Algorithm 4.4.2 $Schedule(t, CT_j^i)$.

Input: Earliest available time t , a communication task CT_j^i in Q_1 to be scheduled on bus, communication tasks in Q_2 .

Output: Release time of communication task CT_j^i if it returns *Success*.

```
1: if  $t \leq S_j^i$  then
2:   Schedule tasks in  $Q_2$  to time slot  $[t, S_j^i]$  using Best Fit Decreasing.
3:   for each task  $CT_{j'}^{i'}$  scheduled in time slot  $[t, S_j^i]$  do
4:      $S_{j'}^{i'} \leftarrow t + c_{j'}^{i'}$ , DELETE( $Q_2, CT_{j'}^{i'}$ ).
5:   end for
6:    $t \leftarrow S_j^i + c_j^i$ , DELETE( $Q_1, CT_j^i$ ), Return Success.
7: end if
8: if  $t \leq d_j^i - c_j^i$  then
9:    $S_j^i \leftarrow t$ ,  $t \leftarrow S_j^i + c_j^i$ , DELETE( $Q_1, CT_j^i$ ), Return Success.
10: end if
11: Return Failure.
```

of T_i and T_j are fixed. Therefore, the associated communication task will not influence the memory usage of the schedule. These tasks will be inserted into Q_2 . Other communication tasks with non-zero weight will be inserted into Q_1 . Lines 13-32 schedule each task in Q_1 . The task CT_j^i with larger weight will have higher priority, and it can preempt the previous task $CT_{j'}^{i'}$ with smaller weight.

For the second step, Algorithm 4.4.1 will call function $Schedule(t, CT_j^i)$ to schedule task CT_j^i at time t . Function $Schedule(t, CT_j^i)$, which is shown in Algorithm 4.4.2, compares the release time S_j^i with the earliest available time t . If t is less than or equal to S_j^i , a time slot between t and S_j^i is created, and this time slot can be used to allocate tasks in Q_2 . Function $Schedule(t, CT_j^i)$ adopts Best Fit Decreasing algorithm to schedule tasks to time slot $[t, S_j^i]$. If t is greater than S_j^i , function $Schedule(t, CT_j^i)$ will check if t is less than or equal to $d_j^i - c_j^i$. If it satisfies the condition, in line 10 of Algorithm 4.4.2, communication task CT_j^i will be scheduled at time t . If time t satisfies neither the condition in line 1 nor that in line 9, function $Schedule(t, CT_j^i)$ will return *Failure*.

For the last step, Algorithm 4.4.3 will get the retiming value of each task. Lines 1-6 will initialize the retiming value of each task as 0, and each leaf node in the DAG G will be inserted into queue Q . In lines 7-19, Algorithm 4.4.3 iteratively gets the retiming value of each computation task T_i . If communication task CT_j^i can be scheduled between the finishing time of task T_i and the release time of task T_j inside the same period, the larger value between $\mathcal{R}(T_i)$ and $\mathcal{R}(T_j)$ will be assigned to $\mathcal{R}(T_i)$. Otherwise, the larger value between $\mathcal{R}(T_i)$ and $\mathcal{R}(T_j) + 1$ will be assigned to $\mathcal{R}(T_i)$. In line 16, task T_i will be inserted into queue Q , and it will be used to generate the retiming value of its predecessor tasks. Lines 20-26 get the retiming value of each intercore communication task CT_j^i . If the finishing time of task CT_j^i is no later than the finishing time of task T_i and the retiming value of task T_i is greater than the retiming value of task T_j , $\mathcal{R}(T_i) - 1$ will be assigned to $\mathcal{R}(CT_j^i)$. Otherwise, $\mathcal{R}(T_i)$ will be assigned to $\mathcal{R}(CT_j^i)$ to guarantee that intercore communication task CT_j^i is finished before the completion of task T_i in the next period.

Here we analyze the time complexity of the proposed heuristic algorithm HMAOTS. Given a set of n computation tasks and a set of m intercore communication tasks, in Algorithm 4.4.1, the first “for” loop contains at most m iterations and each iteration takes linear time. Therefore, the time complexity of lines 1-12 is $O(m)$. For the “while” loop in Algorithm 4.4.1 (lines 13-32), it calls function $\text{Schedule}(t, CT_j^i)$ in line 14. The time complexity of the function is bounded by Best Fit Decreasing algorithm in line 2 of Algorithm 4.4.2. The time complexity of Best Fit Decreasing algorithm is $O(m \log m)$, so the time complexity of the “while” loop is $O(m^2 \log m)$. The time complexity of the last “while” loop is $O(m)$. For function $\text{GetRetimingValue}()$ in Algorithm 4.4.3, the time complexity of the first “for” loop (lines 1-6) is $O(n)$; the “while” loop (lines 7-19) takes $O(mn)$; and the time complexity of the last “for” loop (lines 20-26) is $O(m)$. Based on this analysis, the time complexity of Algorithm HMAOTS is $O(m^2 \log m) + O(mn)$.

Algorithm 4.4.3 GetRetimingValue().

Input: A DAG $G = (V, E, CT)$, release time and execution time of each task.

Output: Retiming value of each task.

```
1: for each task  $T_i \in V$  do
2:    $\mathcal{R}(T_i) \leftarrow 0$ .
3:   if  $T_i$  is a leaf node then
4:     ENQUEUE( $Q, T_i$ ).
5:   end if
6: end for
7: while  $Q \neq \emptyset$  do
8:    $T_j \leftarrow$  DEQUEUE( $Q$ ).
9:   for each communication task  $CT_j^i$  associated with tasks  $T_i$  and  $T_j$  do
10:    if  $S_i + c_i \leq S_j^i$  and  $S_j^i + c_j^i \leq S_j$  then
11:       $\mathcal{R}(T_i) \leftarrow \max\{\mathcal{R}(T_i), \mathcal{R}(T_j)\}$ .
12:    else
13:       $\mathcal{R}(T_i) \leftarrow \max\{\mathcal{R}(T_i), \mathcal{R}(T_j) + 1\}$ .
14:    end if
15:    if  $tail \neq T_i$  then
16:      ENQUEUE( $Q, T_i$ ),  $tail \leftarrow T_i$ .
17:    end if
18:  end for
19: end while
20: for each communication task  $CT_j^i$  associated with tasks  $T_i$  and  $T_j$  do
21:  if  $\mathcal{R}(T_i) > \mathcal{R}(T_j)$  and  $S_i + c_i \geq S_j^i + c_j^i$  then
22:     $\mathcal{R}(CT_j^i) \leftarrow \mathcal{R}(T_i) - 1$ .
23:  else
24:     $\mathcal{R}(CT_j^i) \leftarrow \mathcal{R}(T_i)$ .
25:  end if
26: end for
```

4.5 Experiments

To evaluate the effectiveness of the proposed approach, we conduct a series of experiments on various benchmarks from both real-life streaming applications and synthetic task graphs. We compare and evaluate the proposed approach over the representative schemes in Xu et al. [111], Zhang et al. [115], and Wang et al. [106], in terms of two performance metrics: schedule length and memory usage. In this section, we first introduce the experimental setup and performance metrics. Then, we present the experimental results and discussion.

4.5.1 Experimental Setup

(1) *Experimental Setup.* We conduct experiments on various benchmarks from both real-life streaming applications and synthetic task graphs. Several real-life task graphs are obtained from benchmarks E3S [104], CNC [51], ATR [111], and Image enhancement [103]. Among them, *consumer* from Embedded Systems Synthesis Benchmarks (E3S) represents an embedded consumer electronic application. *Auto* from E3S is an embedded auto-industry application. *Telecom* from E3S represents an embedded telecom application. CNC controller is an automatic machining tool which is used to produce real-time user-designed work pieces. ATR is a streaming application that does pattern matching of targets in images. Image enhancement application uses Sobel gradient, histogram, and Laplacian to improve the image quality. The synthetic task graphs were generated by TGFF v3.5 [28]. Benchmarks *kseries_parallel* and *kseries_parallel_xover* are generated by TGFF using the sample input files that come with the software package.

We implement a simulator based on the processor model of ARM11 MPCore multicore processor microarchitecture [10]. The ARM11 MPCore processor implements the ARM11 microarchitecture and can be configured to contain up to four ARM11 processors [10]. The ARM11 MPCore adopts a single 64-bit AMBA AXI system bus to interconnect different processor cores and provides the maximum throughput of 1.3Gbytes/sec. In the experiments, each ARM11 MPCore is configured to operate at the frequency of 620MHz. The simulator can generate the objective computation task schedule and obtain the con-

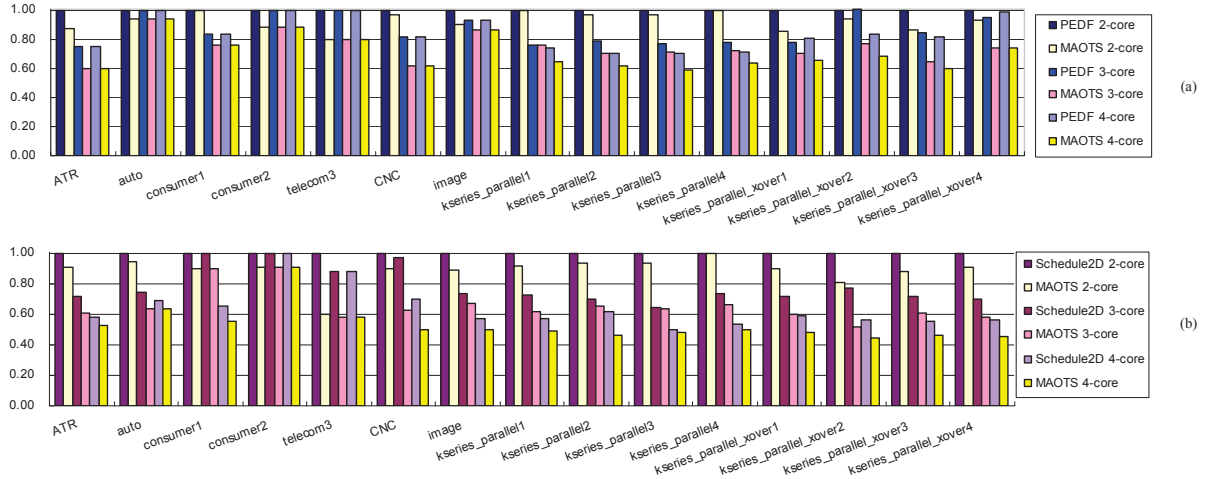


Figure 4.6. Schedule Length by Task Schedules PEDF [115], Schedule2D [111], and the proposed approach (MAOTS) on 2, 3, and 4 processor cores.

straints for the integer linear programming formulation. The ILP model is solved by the open source program linear programming solver, LP_solve_5.5 [32]. Both the simulator and LP_solve_5.5 are running on a 2.83GHz Intel Core2 Quad processor with 4GB memory.

(2) *Performance Metrics*. Two performance metrics are used to evaluate the effectiveness of the proposed approach:

Schedule length. By removing intercore communication overhead, our approach can reduce the schedule length in each period such that the system performance can be improved by adopting a shorter period. Applying a shorter period implies the improvement for application throughput, which is the primary concern for streaming applications. For each benchmark, we compare with the task schedules generated by the algorithm `Schedule2D` in [111] and the algorithm `PEDF` in [115]. `Schedule2D` [111] is an algorithm that jointly performs processor allocation and task scheduling for streaming applications on multiprocessor architectures. The generated schedule consists of multiple pipeline stages and it is proved to be an optimal allocation for the amount of time in each pipeline stage. `PEDF` [115] is selected for comparison because of its reasonably good performance for task mapping and task ordering for multicore platform. We test each benchmark and obtain the schedule length

in each period.

Memory usage. By rescheduling tasks with inter-period dependency, more data buffers are required to hold intercore communication data. Our approach can generate the task schedule with the objective of minimizing the memory usage. We compare the proposed approach with the algorithm `Schedule2D` in [111] and the algorithm `JCCTS` in [106]. Both of these schemes allocate tasks into different pipeline stages and change data dependency relations across different periods. `JCCTS` [106] is a task scheduling technique considering intercore communication overhead. However, in this technique, the memory usage is not considered. We present experimental results on memory usage for each benchmark under different numbers of processor cores. In this thesis, we also propose a heuristic approach `HMAOTS` to efficiently obtain a near optimal solution. We compare the memory usage and time cost of the heuristic approach and the proposed ILP-based optimal solution.

4.5.2 Results and Discussion

In this section, we present the experimental results of the proposed approach and the previous work in terms of schedule length and memory usage. We use `PEDF`, `Schedule2D`, `JCCTS`, and `MAOTS` to represent the experimental results generated by task scheduling in Zhang et al. [115], Xu et al. [111], Wang et al. [106], and the proposed approach, respectively.

Figure 4.6 shows the experimental results for schedule length of task schedules `PEDF`, `Schedule2D`, and the proposed approach under 2, 3, and 4 processor cores. The task schedules `PEDF` and `Schedule2D` are used to generate the initial task schedule. In Figure 4.6(a), we normalize the schedule length of `PEDF` and the proposed approach running on different number of processor cores by the schedule length of `PEDF` running on 2 processor cores. In Figure 4.6(b), we normalize the schedule length of `Schedule2D` and the proposed approach running on different number of processor cores by the schedule length of `Schedule2D` running on 2 processor cores. For task schedule `PEDF`, which does not change intra-period data dependency relation, the proposed technique can get more performance gains when the number of processor cores is increased. This is because, with more processor cores, the par-

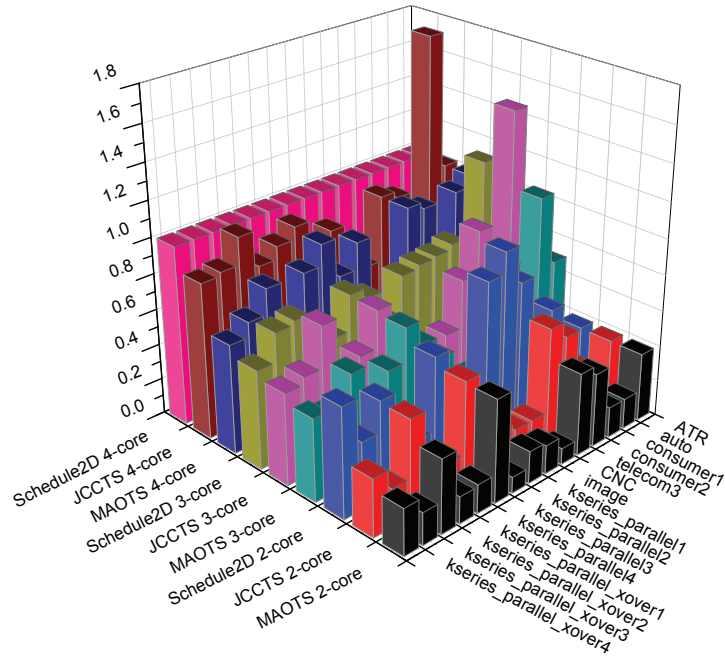


Figure 4.7. Memory Usage of Schedule2D [111], JCCTS [106], and the Proposed Approach MAOTS on 2, 3, and 4 Processor Cores.

allelism is accordingly increased with more intercore communications. For benchmarks with relatively small number of tasks (i.e., auto, consumer2), the reduction of schedule length on 2 processor cores are the same as the one on 3 processor cores or 4 processor cores. This is due to the parallelism of these benchmarks has been exploited with the smaller number of processor cores. The additional processor cores may not provide more performance gains. For task schedule Schedule2D, which changes data dependency relations across different periods, using more numbers of processor cores will directly increase the number of pipeline stages in the schedule. From the results, we can see that, with the increasing of the number of processor cores, the schedule length accordingly decreased. On average, compared with PEDF and Schedule2D, our approach can achieve 12.25% and 14.71% reduction in schedule length.

In the second set of experiments, we test the memory usage for each benchmark under different numbers of processor cores. Figure 4.7 shows the results for memory usage of task schedules Schedule2D, JCCTS, and the proposed approach under 2, 3, and 4 processor

cores. In Figure 4.7, we normalize the memory usage by the memory requirement for Schedule2D running on 4 processor cores. It can be seen from the results that for each benchmark, the proposed approach gives better memory usage than other two approaches. For memory usage, our approach can achieve a 32.45% reduction on average compared with Schedule2D, and a 15.98% reduction on average compared with JCCTS. The task schedule Schedule2D adopts different pipeline stages and each stage is expected to be mapped on the processor cores. Therefore, tasks in the same pipeline stage will have the same retiming value. This may cause unnecessary rescheduling for intercore communication tasks and take extra memory usage to store the intercore communication data. JCCTS is the task schedule that has the objective of minimizing the prologue. By minimizing the prologue, less pipeline stages are adopted. However, more intercore communication tasks may be rescheduled into the prologue, which causes extra memory usage. From the results, our approach can effectively reduce the intercore communication overhead while minimizing the extra memory usage to store the intercore communications.

In the third set of experiments, we test the memory usage and the time cost of the proposed heuristic approach (HMAOTS) and the proposed ILP-based approach Memory-Aware Optimal Task Scheduling (MAOTS). Although the ILP model can obtain optimal solution for the memory-aware task scheduling problem, it is an NP-hard problem to solve the ILP model. Therefore, the ILP model may take very long time to get the results. Table 4.1 presents the time cost for the heuristic approach and the ILP-based optimal solution. From the experimental results, we can see that ILP-based MAOTS approach takes much longer time to get results compared with the one from the heuristic approach HMAOTS. For benchmark *sp_rand3* generated by TGFF [28], the ILP-based approach MAOTS cannot get the results after 3 days, while the heuristic approach HMAOTS still can get the results for less than one minute.

Figure 4.8 presents the experiential results for the memory usage of heuristic approach HMAOTS and the proposed ILP-based optimal task scheduling MAOTS. From the experimental results, we can see that, for most benchmarks, the heuristic approach can obtain near-optimal solutions. To generate an optimal solution for a general problem, the ILP-based

Table 4.1. Comparison in Time Cost of Heuristic Approach (HMAOTS) and Memory-Aware Optimal Task Scheduling (MAOTS) on 2, 3, and 4 processor cores.

Benchmarks	# of task	# of edge	HMAOTS (s)	MAOTS (s)	HMAOTS (s)	MAOTS (s)	HMAOTS (s)	MAOTS (s)
ATR	14	15	0.23	0.92	0.53	13.25	0.53	13.73
auto	6	9	0.23	0.93	0.53	12.15	0.43	7.89
consumer1	7	8	0.43	6.88	0.53	13.40	0.75	36.74
consumer2	5	4	0.23	0.91	0.43	6.88	0.43	6.88
telecom3	6	6	0.53	13.25	0.47	7.52	0.49	12.26
CNC	8	9	0.33	2.97	0.53	13.26	0.53	13.29
image	8	11	0.53	11.77	0.83	53.13	0.83	54.36
kseries_parallel1	30	33	0.54	15.11	0.84	92.22	1.34	216.33
kseries_parallel2	20	19	0.43	6.88	0.63	22.69	0.74	36.26
kseries_parallel3	62	61	0.60	14.57	0.89	56.97	1.01	81.82
kseries_parallel4	47	46	0.44	7.06	0.84	53.78	1.14	137.95
kseries_parallel_xover1	30	37	0.54	13.24	1.04	104.22	1.34	226.46
kseries_parallel_xover2	21	24	0.74	36.26	0.84	54.77	1.04	105.66
kseries_parallel_xover3	38	41	0.58	14.33	1.08	107.88	1.47	248.51
kseries_parallel_xover4	27	30	0.58	12.11	0.92	51.14	1.38	213.53
sp_rand1	199	279	9.99	2705.56	11.96	4569.76	16.82	13314.68
sp_rand2	498	698	25.95	44702.27	32.83	96347.07	34.63	115773.52
sp_rand3	998	1452	52.67	–	71.03	–	82.24	–

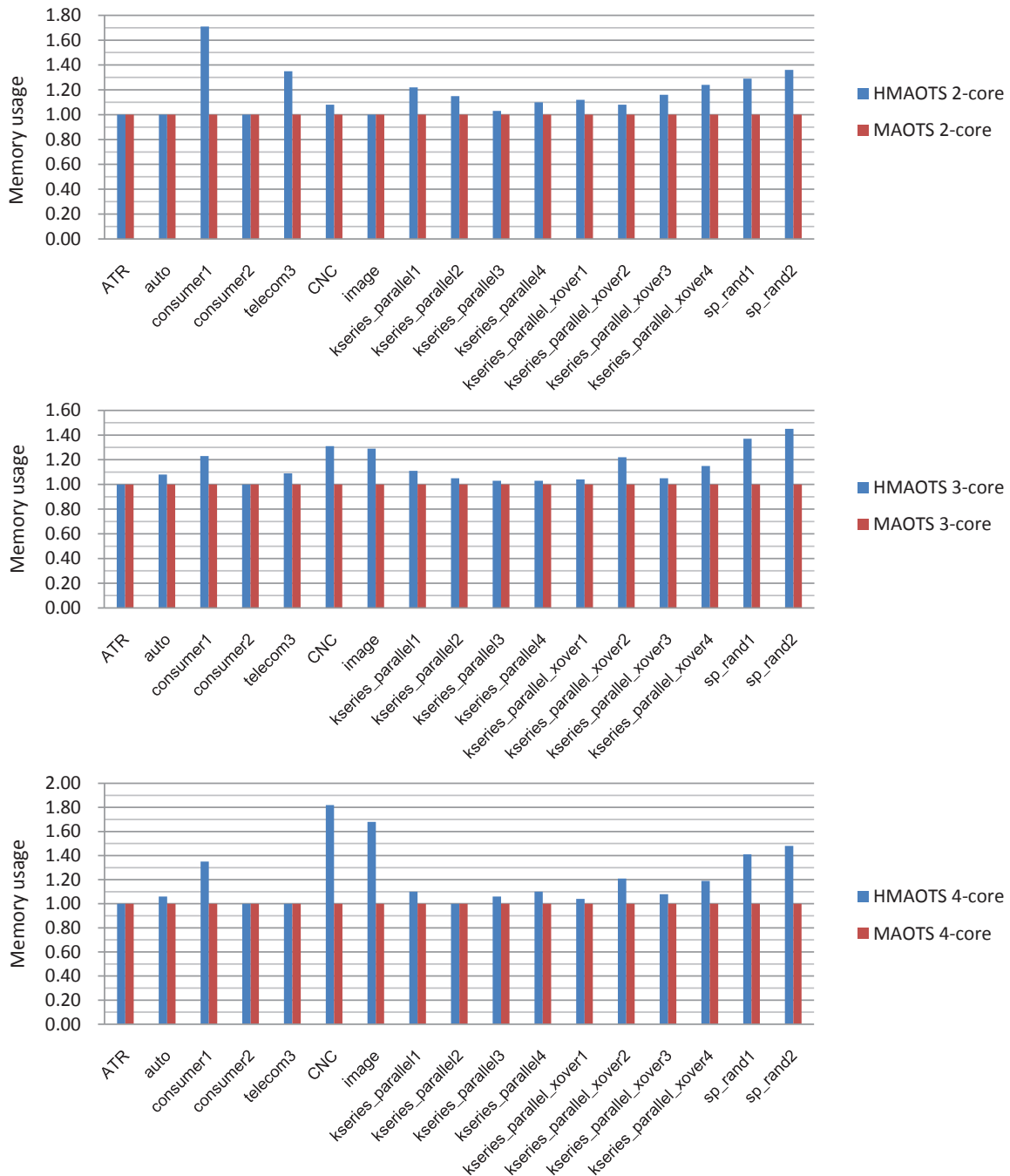


Figure 4.8. Memory Usage of Heuristic Approach (HMAOTS) and Memory-Aware Optimal Task Scheduling (MAOTS) on 2, 3, and 4 processor cores.

approach MAOTS is recommended to be used, which can obtain the optimal solution with the minimum memory usage. When the task graph becomes too big for the ILP model to solve, the proposed heuristic approach is recommended to be used, which gives near-optimal results with less time compared with the ILP-based optimal solution.

4.6 Summary

In this chapter, we have considered the task scheduling problem of removing intercore communication overhead for streaming applications running on MPSoC architectures. We totally removed intercore communication overhead by rescheduling tasks with intra-period data dependencies into inter-period data dependencies, such that the execution of computation and that of intercore communication tasks can be overlapped and a shorter period can be applied. We performed analysis and presented an ILP model to obtain an optimal schedule with the minimum memory usage. We also proposed a heuristic algorithm to efficiently obtain a near optimal solution. Experimental results show that the proposed approach can significantly reduce the schedule length and improve the memory usage compared with representative techniques.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

Streaming applications are data intensive and highly parallelizable; therefore, they are very suitable to be executed on Multiprocessor System-on-Chips (MPSoCs). To fully utilize the computation capacity of MPSoCs, various techniques have been explored to increase parallelism of streaming applications. However, this may cause a large amount of intercore communications with considerable energy overhead and intercore communication overhead. In this thesis, we investigated overhead-aware task scheduling schemes for streaming applications on MPSoC architectures, which can provide comprehensive solutions and generate optimal task schedules for resource-constrained MPSoC architectures. Specifically, we proposed three scheduling schemes to optimize energy consumption, time performance, and memory usage of streaming applications on MPSoCs.

- For the first scheme, we proposed a two-phase approach to solve the energy optimization problem for streaming applications on MPSoCs considering various energy overheads. In the first phase, we proposed a coarse-grained task-level software pipelining algorithm RDAG to transform a set of periodic dependent tasks into a set of periodic independent tasks based on the retiming technique. In the second phase, we proposed a genetic algorithm GeneS for energy optimization considering various overheads. We conducted experiments on a set of benchmarks. Experimental results show that through the combination of software pipelining with DVS and DPM considering several energy overheads, our approach can fully exploit the potential of MPSoC ar-

chitectures and the periodic characteristic of streaming applications to reduce energy consumption.

- For the second scheme, we studied the problem of minimizing intercore communication overhead for streaming applications running on MPSoC architectures. We jointly optimized computation task schedule and intercore communication task schedule such that intercore communication overheads can be totally removed and the schedule length can be minimized. We first performed schedulability analysis and theoretically obtained the upper bound on the prologue length of the computation task schedule. Then we presented an ILP formulation to generate an optimal objective task schedule. Experimental results show that our technique can significantly reduce schedule length and energy consumption compared with representative techniques.
- For the third scheme, we considered the task scheduling problem of removing intercore communication overhead for streaming applications running on MPSoC architectures to minimize the extra memory usage caused by retiming. We totally removed intercore communication overhead by rescheduling tasks with intra-period data dependencies into inter-period data dependencies, such that the execution of computation and that of intercore communication tasks can be overlapped and a shorter period can be applied. We performed analysis and presented an ILP model to obtain an optimal schedule with the minimum memory usage. We also proposed a heuristic algorithm to efficiently obtain a near optimal solution. Experimental results show that the proposed approach can significantly reduce schedule length and improve memory usage compared with representative techniques.

The proposed schemes are integrated into the overhead-aware task scheduling framework, which can be a good supplement to the previous work on modeling for MPSoC architectures, communication protocols and standards, as well as active research on communication-centric design and exploration for MPSoC architectures. The exploration of our design methodology can create a comprehensive study that improves upon the state-of-the-art. Re-

sults of applying our schemes on several benchmarks for embedded systems have shown the effectiveness of the proposed schemes for resource-constrained MPSoC designs.

5.2 Future Work

The work presented in this thesis can be extended in different directions in the future.

- First, task splitting and task migration are not allowed in this work. How to combine our approach with task splitting and task migration can be a future direction for us to explore.
- Second, our approach is based on the shared bus architecture. We will extend our approach to other intercore communication infrastructures (such as crossbar and mesh) and propose a general model that can be applied to different system architectures.
- Third, for the minimization of the intercore communication overhead, task mapping of computation tasks is predefined. It is an interesting problem to exploit the trade-off between intercore communication overhead and energy consumption of the tasks.
- Finally, a possible research direction is to integrate our technique into a compiler or real-time operating systems to leverage system-wide energy consumption.

REFERENCES

- [1] Subrata Acharya and Rabi Mahapatra. A dynamic slack management technique for real-time distributed embedded systems. *IEEE Transactions on Computers*, 57(2):215–230, 2008.
- [2] Andrea Acquaviva, Andrea Alimonda, Salvatore Carta, and Michele Pittau. Assessing task migration impact on embedded soft real-time streaming multimedia applications. *EURASIP Journal on Embedded Systems*, 2008:1–15, 2008.
- [3] Enrique Alba and José M. Troya. A survey of parallel distributed genetic algorithms. *Complexity*, 4(4):31–52, 1999.
- [4] Tarek A. AlEnawy and Hakan Aydin. Energy-aware task allocation for rate monotonic scheduling. In *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium (RTAS '05)*, pages 213–223, 2005.
- [5] Alan Allan, Don Edenfeld, William H. Joyner, Jr., Andrew B. Kahng, Mike Rodgers, and Yervant Zorian. 2001 technology roadmap for semiconductors. *Computer*, 35:42–53, January 2002.
- [6] Altera Corporation. Altera Avalon Interface Specifications. http://www.altera.com/literature/manual/mnl_avalon_spec.pdf, 2010.
- [7] AMD. Mobile AMD Athlon 4 processor model 6 CPGA data sheet. *Advanced Micro Devices, Technical Report 24319*, November 2001.
- [8] ARM. ARM AMBA specification and multilayer AHB specification (rev 2.0). <http://www.arm.com/>, 2001.

- [9] ARM. ARM AMBA 3.0 AXI specification. <http://www.arm.com/armtech/AXI>, 2011.
- [10] ARM. ARM11 MPCore multicore processor microarchitecture. <http://www.arm.com/products/processors/classic/arm11/arm11-mpcore.php>, June 2011.
- [11] Hakan Aydin, Vinay Devadas, and Dakai Zhu. System-level energy management for periodic real-time tasks. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS '06)*, pages 313–322, 2006.
- [12] Hakan Aydin, Rami Melhem, Daniel Mossé, and Pedro Mejía-Alvarez. Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS '01)*, pages 225–232, 2001.
- [13] Hakan Aydin and Qi Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS '03)*, pages 113–121, 2003.
- [14] Neal K. Bambha and Shuvra S. Bhattacharyya. A joint power/performance optimization algorithm for multiprocessor systems using a period graph construct. In *Proceedings of the 13th international symposium on System synthesis (ISSS '00)*, pages 91–97, 2000.
- [15] Sanjoy K. Baruah, Louis E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2(4):301–324, 1990.
- [16] Enrico Bini, Giorgio Buttazzo, and Giuseppe Lipari. Speed modulation in energy-aware real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS '05)*, pages 3–10, 2005.

- [17] Thomas David Burd. *Energy-Efficient processor system design*. PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 2001.
- [18] Lukai Cai and Daniel Gajski. Transaction level modeling: an overview. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '03)*, pages 19–24, 2003.
- [19] M. Caldari, M. Conti, M. Coppola, S. Curaba, L. Pieralisi, and C. Turchetti. Transaction-level models for AMBA bus architecture using SystemC 2.0. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE '03)*, pages 26–31, 2003.
- [20] Liang-Fang Chao and Andrea LaPaugh. Rotation scheduling: a loop pipelining algorithm. In *Proceedings of the 30th international Design Automation Conference (DAC '93)*, pages 566–572, 1993.
- [21] Liang-Fang Chao and Edwin Hsing-Mean Sha. Static scheduling of uniform nested loops. In *Proceedings of 7th International Parallel Processing Symposium (IPPS '93)*, pages 254–258, 1993.
- [22] Jian-Jia Chen and Tei-Wei Kuo. Energy-efficient scheduling of periodic real-time tasks over homogeneous multiprocessors. In *Proceedings of the 2nd International Workshop on Power-Aware Real-Time Computing (PARC'05)*, pages 30–35, 2005.
- [23] Jian-Jia Chen, Tei-Wei Kuo, and Chi-Sheng Shih. $1 + \epsilon$ approximation clock rate assignment for periodic real-time tasks on a voltage-scaling processor. In *Proceedings of the 5th ACM international conference on Embedded software (EMSOFT '05)*, pages 247–250, 2005.
- [24] Ya-Shu Chen, Chi-Sheng Shih, and Tei-Wei Kuo. Dynamic task scheduling and processing element allocation for multi-function SoCs. In *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS '07)*, pages 81–90, 2007.

- [25] Young-Sin Cho, Eun-Ju Choi, and Kyoung-Rok Cho. Modeling and analysis of the system bus latency on the SoC platform. In *Proceedings of the 2006 international workshop on System-level interconnect prediction (SLIP '06)*, pages 67–74, 2006.
- [26] Chen-Ling Chou and Radu Marculescu. User-aware dynamic task allocation in networks-on-chip. In *Proceedings of the conference on Design, automation and test in Europe (DATE '08)*, pages 1232–1237, 2008.
- [27] Seo DaeHo and Mithuna Thottethodi. Disjoint-path routing: Efficient communication for streaming applications. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing (IPDPS '09)*, pages 1–12, 2009.
- [28] Robert P. Dick, David L. Rhodes, and Wayne Wolf. TGFF: task graphs for free. In *Proceedings of the Sixth International Workshop on Hardware/Software Codesign (CODES '98)*, pages 97–101, 1998.
- [29] Emiliano Dolif, Michele Lombardi, Martino Ruggiero, Michela Milano, and Luca Benini. Communication-aware stochastic allocation and scheduling framework for conditional task graphs in multi-processor systems-on-chip. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software (EMSOFT '07)*, pages 47–56, 2007.
- [30] Hesham El-Rewini, Hesham H. Ali, and Ted Lewis. Task scheduling in multiprocessing systems. *Computer*, 28(12):27–37, 1995.
- [31] Mohammad H. Foroozannejad, Matin Hashemi, Trevor L. Hodges, and Soheil Ghiasi. Look into details: the benefits of fine-grain streaming buffer analysis. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems (LCTES '10)*, pages 27–36, 2010.
- [32] Free Software Foundation, Inc. Lp_solve 5.5. <http://lpsolve.sourceforge.net/5.5/>, 2010.

- [33] Freescale Semiconductor, Inc. i.MX35 Multimedia Applications Processors. http://www.freescale.com/webapp/sps/site/taxonomy.jsp?code=IMX35_FAMILY, 2010.
- [34] Guy Gogniat, Michel Auguin, Luc Bianco, and Alain Pegatoquet. Communication synthesis and HW/SW integration for embedded system design. In *Proceedings of the 6th international workshop on Hardware/software codesign (CODES/CASHE '98)*, pages 49–53, 1998.
- [35] Sathish Gopalakrishnan, Lui Sha, and Marco Caccamo. Hard real-time communication in bus-based networks. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS '04)*, pages 405–414, 2004.
- [36] Flavius Gruian and Krzysztof Kuchcinski. LEneS: task scheduling for low-energy systems using variable supply voltage processors. In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference (ASP-DAC '01)*, pages 449–455, 2001.
- [37] Zonghua Gu, Xiuqiang He, and Mingxuan Yuan. Optimization of static task and bus access schedules for time-triggered distributed embedded systems with model-checking. In *Proceedings of the 44th annual Design Automation Conference (DAC '07)*, pages 294–299, 2007.
- [38] Pi-Cheng Hsiu, Der-Nien Lee, and Tei-Wei Kuo. Multi-layer bus optimization for real-time task scheduling with chain-based precedence constraints. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS '09)*, pages 479–488, 2009.
- [39] Jingcao Hu and Radu Marculescu. Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints. In *Proceedings of the conference on Design, automation and test in Europe (DATE '04)*, pages 234–239, 2004.

- [40] Shaoxiong Hua and Gang Qu. Voltage setup problem for embedded systems with multiple voltages. *IEEE Transactions on Very Large Scale Integration Systems*, 13(7):869–872, 2005.
- [41] Chia-Mei Hung, Jian-Jia Chen, and Tei-Wei Kuo. Energy-efficient real-time task scheduling for a DVS system with a non-DVS processing element. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS '06)*, pages 303–312, 2006.
- [42] IBM. IBM CoreConnect bus architecture. https://www-01.ibm.com/chips/techlib/techlib.nsf/products/CoreConnect_Bus_Architecture, 2011.
- [43] Ilya Issenin and Nikil Dutt. Data reuse driven energy-aware MPSoC co-synthesis of memory and communication architecture for streaming applications. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis (CODES+ISSS '06)*, pages 294–299, 2006.
- [44] Ravindra Jejurikar and Rajesh Gupta. Dynamic voltage scaling for systemwide energy minimization in real-time embedded systems. In *Proceedings of the 2004 international symposium on Low power electronics and design (ISLPED '04)*, pages 78–81, 2004.
- [45] Niraj K. Jha. Low power system scheduling and synthesis. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design (ICCAD '01)*, pages 259–263, 2001.
- [46] David S. Johnson. *Near-optimal bin packing algorithms*. Massachusetts Institute of Technology Press, Cambridge, MA, USA, 1973.
- [47] Praveen Kalla, X. Sharon Hu, and Jörg Henkel. A flexible framework for communication evaluation in SoC design. *International Journal of Parallel Programming*, 36:457–477, October 2008.

- [48] Vida Kianzad, Shuvra S. Bhattacharyya, and Gang Qu. CASPER: An integrated energy-driven approach for task graph scheduling on distributed embedded systems. In *Proceedings of the 2005 IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP '05)*, pages 191–197, 2005.
- [49] Minyoung Kim, Sudarshan Banerjee, Nikil Dutt, and Nalini Venkatasubramanian. Design space exploration of real-time multi-media MPSoCs with heterogeneous scheduling policies. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis (CODES+ISSS '06)*, pages 16–21, 2006.
- [50] Nam Sung Kim, Taeho Kgil, K. Bowman, V. De, and T. Mudge. Total power-optimal pipelining and parallel processing under process variations in nanometer technology. In *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design (ICCAD '05)*, pages 535–540, 2005.
- [51] Namyun Kim, Minsoo Ryu, Seongsoo Hong, M. Saksena, Chong-Ho Choi, and Heonshik Shin. Visual assessment of a real-time system design: a case study on a CNC controller. In *Proceedings of the 17th IEEE International Real-Time Systems Symposium (RTSS '96)*, pages 300–310, 1996.
- [52] Sungchan Kim, Chaeseok Im, and Soonhoi Ha. Schedule-aware performance estimation of communication architecture for efficient design space exploration. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '03)*, pages 195–200, 2003.
- [53] Sungchan Kim, Chaeseok Im, and Soonhoi Ha. Schedule-aware performance estimation of communication architecture for efficient design space exploration. *IEEE Transactions on Very Large Scale Integration Systems*, 13:539–552, May 2005.
- [54] Young-Taek Kim, Taehun Kim, Youngduk Kim, Chulho Shin, Eui-Young Chung, Kyu-Myung Choi, Jeong-Taek Kong, and Soo-Kwan Eo. Fast and accurate transaction level modeling of an extended AMBA 2.0 bus architecture. In *Proceedings of the*

- conference on Design, Automation and Test in Europe (DATE '05), pages 138–139, 2005.
- [55] Peter Voigt Knudsen and Jan Madsen. Communication estimation for hardware/software codesign. In *Proceedings of the 6th international workshop on Hardware/software codesign (CODES/CASHE '98)*, pages 55–59, 1998.
- [56] Peter Voigt Knudsen and Jan Madsen. Integrating communication protocol selection with partitioning in hardware/software codesign. In *Proceedings of the 11th international symposium on System synthesis (ISSS '98)*, pages 111–116, 1998.
- [57] Kanishka Lahiri, Sujit Dey, and Anand Raghunathan. Performance analysis of systems with multi-channel communication architectures. In *Proceedings of the 13th International Conference on VLSI Design (VLSID '00)*, pages 530–537, 2000.
- [58] Kanishka Lahiri, Anand Raghunathan, and Sujit Dey. System-level performance analysis for designing on-chip communication architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(6):768–783, 2001.
- [59] David Landskov, Scott Davidson, Bruce Shriver, and Patrick W. Mallett. Local microcode compaction techniques. *ACM Computing Surveys*, 12(3):261–294, September 1980.
- [60] John P. Lehoczky and Lui Sha. Performance of real-time bus scheduling algorithms. In *Proceedings of the 1986 ACM SIGMETRICS joint international conference on Computer performance modelling, measurement and evaluation (SIGMETRICS '86/PERFORMANCE '86)*, pages 44–53, 1986.
- [61] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.
- [62] Jian Li and José F. Martínez. Power-performance considerations of parallel computing on chip multiprocessors. *ACM Transactions on Architecture and Code Optimization*, 2(4):397–422, 2005.

- [63] Yu-Hsien Lin, Chiaheng Tu, Chi-Sheng Shih, and Shih-Hao Hung. Zero-buffer inter-core process communication protocol for heterogeneous multi-core platforms. In *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '09)*, pages 69–78, 2009.
- [64] Cong Liu and James H. Anderson. Scheduling suspendable, pipelined tasks with non-preemptive sections in soft real-time multiprocessor systems. In *Proceedings of the 16th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS '10)*, pages 23–32, 2010.
- [65] Hui Liu, Zili Shao, Meng Wang, and Ping Chen. Overhead-aware system-level joint energy and performance optimization for streaming applications on multiprocessor systems-on-chip. In *Proceedings of the 2008 Euromicro Conference on Real-Time Systems (ECRTS '08)*, pages 92–101, 2008.
- [66] Hui Liu, Zili Shao, Meng Wang, Junzhao Du, Chun Jason Xue, and Zhiping Jia. Combining coarse-grained software pipelining with DVS for scheduling real-time periodic dependent tasks on multi-core embedded systems. *Journal of Signal Processing Systems*, 57(2):249–262, 2009.
- [67] Mirko Loghi, Federico Angiolini, Davide Bertozzi, Luca Benini, and Roberto Zafalon. Analyzing on-chip communication in a MPSoC environment. In *Proceedings of the conference on Design, automation and test in Europe (DATE '04)*, pages 752–757, 2004.
- [68] Alessio Guerri Luca Benini, Davide Bertozzi and Michela Milano. Allocation, scheduling and voltage scaling on energy aware MPSoCs. In *Lecture Notes in Computer Science, Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 44–58. Springer, 2006.
- [69] Jiong Luo and Niraj K. Jha. Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems. In *Proceedings of the*

2000 *IEEE/ACM international conference on Computer-aided design (ICCAD '00)*, pages 357–364, 2000.

- [70] Jiong Luo and Niraj K. Jha. Power-efficient scheduling for heterogeneous distributed real-time embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(6):1161–1170, June 2007.
- [71] A. Mahalanobis, B. V. K. Vijaya Kumar, and S. R. F. Sims. Distance-classifier correlation filters for multiclass target recognition. In *Applied Optics*, volume 35, pages 3127–3133, 1996.
- [72] Steven M. Martin, Krisztian Flautner, Trevor Mudge, and David Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design (ICCAD '02)*, pages 721–725, 2002.
- [73] Samy Meftali, Ferid Gharsalli, Frederic Rousseau, and Ahmed A. Jerraya. An optimal memory allocation for application-specific multiprocessor system-on-chip. In *Proceedings of the 14th international symposium on Systems synthesis (ISSS '01)*, pages 19–24, 2001.
- [74] Pedro Mejia-Alvarez, Eugene Levner, and Daniel Mossé. Adaptive scheduling server for power-aware real-time tasks. *ACM Transactions in Embedded Computing Systems*, 3(2):284–306, 2004.
- [75] Melanie Mitchell. *An introduction to genetic algorithms*. The MIT Press, Cambridge, MA, USA, 1996.
- [76] Bren Mochocki, Xiaobo Sharon Hu, and Gang Quan. A unified approach to variable voltage scheduling for nonideal DVS processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(9):1370–1377, 2004.

- [77] Linwei Niu and Gang Quan. System-wide dynamic power management for portable multimedia devices. In *Proceedings of the Eighth IEEE International Symposium on Multimedia (ISM '06)*, pages 97–104, 2006.
- [78] Osamu Ogawa, Sylvain Bayon de Noyer, Pascal Chauvet, Katsuya Shinohara, Yoshiharu Watanabe, Hiroshi Niizuma, Takayuki Sasaki, and Yuji Takai. A practical approach for bus architecture optimization at transaction level. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE '03)*, pages 176–181, 2003.
- [79] OpenCores. WISHBONE System-on-Chip (SoC) Interconnect Architecture. http://cdn.opencores.org/downloads/wbspec_b4.pdf, 2010.
- [80] O. Ozturk, M. Kandemir, S. W. Son, and M. Karakoy. Selective code/data migration for reducing communication energy in embedded MPSoC architectures. In *Proceedings of the 16th ACM Great Lakes symposium on VLSI (GLSVLSI '06)*, pages 386–391, 2006.
- [81] Vivek Pandey, Weihang Jiang, Yuanyuan Zhou, and Ricardo Bianchini. DMA-aware memory energy management. In *Proceedings of the twelfth international symposium on High-Performance Computer Architecture (HPCA '06)*, pages 133–144, 2006.
- [82] Sudeep Pasricha and Nikil Dutt. *On-Chip Communication Architectures: System on chip interconnect*. Morgan Kaufmann Publishers, Burlington, MA, USA, 2008.
- [83] Sudeep Pasricha, Nikil Dutt, and Mohamed Ben-Romdhane. Extending the transaction level modeling approach for fast communication architecture exploration. In *Proceedings of the 41st annual Design Automation Conference (DAC '04)*, pages 113–118, 2004.
- [84] Sudeep Pasricha, Nikil Dutt, and Mohamed Ben-Romdhane. Fast exploration of bus-based on-chip communication architectures. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '04)*, pages 242–247, 2004.

- [85] Nelson Luiz Passos and Edwin Hsing-Mean Sha. Achieving full parallelism using multidimensional retiming. *IEEE Transactions on Parallel and Distributed Systems*, 7(11):1150–1163, 1996.
- [86] D.-T. Peng and K.G. Shin. Static allocation of periodic tasks with precedence constraints in distributed real-time systems. In *Proceedings of the 9th International Conference on Distributed Computing Systems (ICDCS '89)*, pages 190–198, 1989.
- [87] Paul Pop, Petru Eles, Zebo Peng, and Traian Pop. Analysis and optimization of distributed real-time embedded systems. In *Proceedings of the 41st annual conference on Design automation (DAC '04)*, pages 593–625, 2004.
- [88] Traian Pop, Petru Eles, and Zebo Peng. Design optimization of mixed time/event-triggered distributed embedded systems. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '03)*, pages 83–89, New York, NY, USA, 2003. ACM.
- [89] Shiv Prakash and Alice C. Parker. Readings in hardware/software co-design. chapter SOS: synthesis of application-specific heterogeneous multiprocessor systems, pages 324–337. Kluwer Academic Publishers, Norwell, MA, USA, 2002.
- [90] Gang Quan and X. Sharon Hu. Minimum energy fixed-priority scheduling for variable voltage processor. In *Proceedings of the conference on Design, automation and test in Europe (DATE '02)*, pages 782–787, 2002.
- [91] Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. *Digital Integrated Circuits, 2nd edition*. Prentice Hall, Englewood Cliffs, N.J, 2002.
- [92] Frank-Michael Renner, Juergen Becker, and Manfred Glesner. Communication performance models for architecture-precise prototyping of real-time embedded systems. In *Proceedings of the 10th IEEE International Workshop on Rapid System Prototyping (RSP '99)*, pages 108–113, 1999.

- [93] Frank-Michael Renner, Juergen Becker, and Manfred Glesner. Automated communication synthesis for architecture-precise rapid prototyping of real-time embedded system. In *Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping (RSP 2000)*, pages 154–159, 2000.
- [94] Jakob Rosen, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS '07)*, pages 49–60, 2007.
- [95] Martino Ruggiero, Alessio Guerri, Davide Bertozzi, Francesco Poletti, and Michela Milano. Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip. In *Proceedings of the conference on Design, automation and test in Europe (DATE '06)*, pages 3–8, 2006.
- [96] Saowanee Saewong and Ragunathan (Raj) Rajkumar. Practical voltage-scaling for fixed-priority RT-systems. In *Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '03)*, pages 106–114, 2003.
- [97] Gunar Schirner and Rainer Dömer. Quantitative analysis of transaction level models for the AMBA bus. In *Proceedings of the conference on Design, automation and test in Europe (DATE '06)*, pages 230–235, 2006.
- [98] Luc Séméria and Abhijit Ghosh. Methodology for hardware/software co-verification in C/C++. In *Proceedings of the 2000 Asia and South Pacific Design Automation Conference (ASP-DAC '00)*, pages 405–408, 2000.
- [99] Zili Shao, Meng Wang, Ying Chen, Chun Xue, Meikang Qiu, and Laurence T. Yang. Real-time dynamic voltage loop scheduling for multi-core embedded systems. *IEEE Transactions on Circuits and Systems II (TCAS-II)*, 54(5):445–449, May 2007.
- [100] Dongkun Shin, Jihong Kim, and Seongsoo Lee. Low-energy intra-task voltage scheduling using static timing analysis. In *Proceedings of the 38th annual Design Automation Conference (DAC '01)*, pages 438–443, 2001.

- [101] Sonics Inc. SonicsSX SMART Interconnect solution. <http://www.sonicsinc.com/sonicsSX.htm>, 2010.
- [102] STMicroelectronics. STBus Interconnect. http://www.st.com/stonline/products/technologies_11302010/soc/stbus.htm, 2010.
- [103] Fei Sun, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. Synthesis of application-specific heterogeneous multiprocessor architectures using extensible processors. In *Proceedings of the 18th International Conference on VLSI Design (VLSID '05)*, pages 551–556, 2005.
- [104] Keith S. Vallerio and Niraj K. Jha. Task graph extraction for embedded system synthesis. In *Proceedings of the 16th International Conference on VLSI Design (VLSID '03)*, pages 480–486, 2003.
- [105] Girish Varatkar and Radu Marculescu. Communication-aware task scheduling and voltage selection for total systems energy minimization. In *Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design (ICCAD '03)*, pages 510–517, 2003.
- [106] Yi Wang, Duo Liu, Meng Wang, Zhiwei Qin, and Zili Shao. Optimal task scheduling by removing inter-core communication overhead for streaming applications on MPSoC. In *Proceedings of the 16th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS '10)*, pages 195–204, 2010.
- [107] Thomas Wiegand, Gary J. Sullivan, Gisle Bjontegaard, and Ajay Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, 2003.
- [108] Maarten H. Wiggers, Marco J. G. Bekooij, and Gerard J. M. Smit. Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication. In *Proceedings of the 14th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS '08)*, pages 183–194, 2008.

- [109] Wayne Wolf, A.A. Jerraya, and Grant Martin. Multiprocessor System-on-Chip (MP-SoC) Technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1701–1713, oct. 2008.
- [110] Cathy Qun Xu, Chun Jason Xue, Bessie C. Hu, and Edwin H. M. Sha. Computation and data transfer co-scheduling for interconnection bus minimization. In *Proceedings of the 2009 Asia and South Pacific Design Automation Conference (ASP-DAC '09)*, pages 311–316, 2009.
- [111] Ruibin Xu, Rami Melhem, and Daniel Mosse. Energy-aware scheduling for streaming applications on chip multiprocessors. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS '07)*, pages 25–38, 2007.
- [112] Chengmo Yang and Alex Orailoglu. Towards no-cost adaptive MPSoC static schedules through exploitation of logical-to-physical core mapping latitude. In *Proceedings of the conference on Design, automation and test in Europe (DATE '09)*, pages 63–69, 2009.
- [113] Ti-Yen Yen and Wayne Wolf. Communication synthesis for distributed embedded systems. In *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design (ICCAD '95)*, pages 288–294, 1995.
- [114] Yang Yu and Viktor K. Prasanna. Power-aware resource allocation for independent tasks in heterogeneous real-time systems. In *Proceedings of the 9th International Conference on Parallel and Distributed Systems (ICPADS '02)*, pages 341–348, 2002.
- [115] Yumin Zhang, Xiaobo Sharon Hu, and Danny Z. Chen. Task scheduling and voltage selection for energy minimization. In *Proceedings of the 39th annual Design Automation Conference (DAC '02)*, pages 183–188, 2002.
- [116] Xiliang Zhong and Cheng-Zhong Xu. Frequency-aware energy optimization for real-time periodic and aperiodic tasks. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES '07)*, pages 21–30, 2007.

- [117] Dakai Zhu, Rami Melhem, and Bruce R. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multiprocessor real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(7):686–700, 2003.
- [118] Jun Zhu, Ingo Sander, and Axel Jantsch. Energy efficient streaming applications with guaranteed throughput on MPSoCs. In *Proceedings of the 8th ACM international conference on Embedded software (EMSOFT '08)*, pages 119–128, 2008.
- [119] Jun Zhu, Ingo Sander, and Axel Jantsch. Buffer minimization of real-time streaming applications scheduling on hybrid CPU/FPGA architectures. In *Proceedings of the conference on Design, automation and test in Europe (DATE '09)*, pages 1506 – 1511, 2009.
- [120] Qian Zhu and Gagan Agrawal. Resource allocation for distributed streaming applications. In *Proceedings of the 2008 37th International Conference on Parallel Processing (ICPP '08)*, pages 414–421, 2008.