

## ABSTRACT

Title of dissertation: **MULTI-SCALE SCHEDULING TECHNIQUES FOR SIGNAL PROCESSING SYSTEMS**

Zheng Zhou, Doctor of Philosophy, 2013

Dissertation directed by: **Shuvra S. Bhattacharyya (Chair/Advisor)**  
Professor  
Department of Electrical and Computer Engineering,  
and Institute for Advanced Computer Studies

**Gang Qu (Co-Advisor)**  
Associate Professor  
Department of Electrical and Computer Engineering

A variety of hardware platforms for signal processing have emerged, from distributed systems such as Wireless Sensor Networks (WSNs) to parallel systems such as Multi-core Programmable Digital Signal Processors (PDSPs), Multicore General Purpose Processors (GPPs), and Graphics Processing Units (GPUs) to heterogeneous combinations of parallel and distributed devices. When a signal processing application is implemented on one of those platforms, the performance critically depends on the scheduling techniques, which in general allocate computation and communication resources for competing processing tasks in the application to optimize performance metrics such as power consumption, throughput, latency, and accuracy.

Signal processing systems implemented on such platforms typically involve multiple levels of processing and communication hierarchy, such as network-level, chip-level, and processor-level in a structural context, and application-level, subsystem-level,

component-level, and operation- or instruction-level in a behavioral context. In this thesis, we target scheduling issues that carefully address and integrate scheduling considerations at different levels of these structural and behavioral hierarchies. The core contributions of the thesis include the following.

1. Considering both the network-level and chip-level, we have proposed an adaptive scheduling algorithm for wireless sensor networks (WSNs) designed for event detection. Our algorithm exploits discrepancies among the detection accuracies of individual sensors, which are derived from a collaborative training process, to allow each sensor to operate in a more energy efficient manner while the network satisfies given constraints on overall detection accuracy.
2. Considering the chip-level and processor-level, we incorporate both temperature and process variations to develop new scheduling methods for throughput maximization on multicore processors. In particular, we study how to process a large number of threads with high speed and without violating a given maximum temperature constraint. We target our methods to multicore processors in which the cores may operate at different frequencies and different levels of leakage. We develop speed selection and thread assignment schedulers based on the notion of a core's steady state temperature.
3. Considering the application-level, component-level and operation-level, we develop a new dataflow based design flow within the targeted dataflow interchange format (TDIF) design tool. Our new multiprocessor system-on-chip (MPSoC)-oriented design flow, called *TDIF-PPG*, is geared towards analysis and mapping of embedded

DSP applications on MPSoCs. An important feature of TDIF-PPG is its capability to integrate *graph level parallelism* and *actor level parallelism* into the application mapping process. Here, graph level parallelism is exposed by the dataflow graph application representation in TDIF, and actor level parallelism is modeled by a novel model for multiprocessor dataflow graph implementation that we call the *Parallel Processing Group (PPG)* model.

4. Building on Contribution 3, we formulate a new type of parallel task scheduling problem called Parallel Actor Scheduling (PAS) for chip-level MPSoC mapping of DSP systems that are represented as synchronous dataflow (SDF) graphs. In contrast to traditional SDF-based scheduling techniques, which focus on exploiting graph level (inter-actor) parallelism, the PAS problem targets the integrated exploitation of both intra- and inter-actor parallelism for platforms in which individual actors can be parallelized across multiple processing units. We address a special case of the PAS problem in which all of the actors in the DSP application or subsystem being optimized can be parallelized. For this special case, we develop and experimentally evaluate a two-phase scheduling framework with two work flows — particle swarm optimization with a mixed integer programming formulation, and particle swarm optimization with a fast heuristic based on list scheduling.

MULTI-SCALE SCHEDULING TECHNIQUES FOR SIGNAL  
PROCESSING SYSTEMS

by

Zheng Zhou

Dissertation submitted to the Faculty of the Graduate School of the  
University of Maryland, College Park in partial fulfillment  
of the requirements for the degree of  
Doctor of Philosophy  
2013

Advisory Committee:

Professor Shuvra S. Bhattacharyya, Chair/Advisor

Professor Gang Qu, Co-Advisor

Professor Steven Tretter

Professor Manoj Franklin

Professor Yang Tao, Dean's representative

© Copyright by  
Zheng Zhou  
2013

## Dedication

To my parents, my wife and my son.

## Acknowledgments

I express my sincere thanks to my advisor Prof. Shuvra Bhattacharyya for his mentoring, guidance, and inspiration on my research. I am particularly grateful for letting me become a member of the DSPCAD group where I worked on applicable projects and enjoyed great freedom for my research interests. Professor Bhattacharyya taught me in many ways how to become a good researcher. He gave me inspiring advice and insights to broaden the vision of my research. He used his expertise in the field of Digital Signal Processing to help me understand and solve the problems effectively and quickly in this field. He carefully reviewed every sentence of my paper to improve the quality of the paper. He generously shared with me the tips for the presentation. He introduced me to excellent opportunities to know and work with other great researchers. I will always feel thankful for the time we worked together.

I also want to express my deep gratitude to my co-advisor Prof. Gang Qu for his patient guidance, enthusiastic encouragement and useful critiques on my research of Wireless Sensor Network and Multicore Processor Low Power Design. I had not written any scientific paper before working with him. He expressed a great deal of patience to correct the mistakes I had made in my papers and carefully explained to me the good writing style. The work experience with Prof. Gang Qu helped to build a good foundation for my research and future career path. I will be always grateful to him.

I am also thankful to other members of my PhD dissertation committee Prof. Steven Tretter, Prof. Manoj Franklin, and Prof. Yang Tao for their comments, co-operation and valuable feedback.

I am very grateful to Karol Desnos at Institute of Electronics and Telecommunications Rennes (IETR). During his visit at DSPCAD group, he shared with me a lot about his research experiences. I have learned a lot from him. The discussions between us lead to a very good paper idea. His reviews on the paper was also greatly appreciated.

The memories at the Maryland DSPCAD group will be always cherished. I am so thankful to the members of DSPCAD group, my colleagues and my friends—Dr. William Plishker, Dr. Chung-Ching, Dr. George Zaki, Dr. Hsiang-Huang Wu, Inkeun Cho, Ilya Chukhman, Lai-Huei Wang, Scott Kim, Kishan Sudusinghe and Shuoxin Lin for the interesting discussions we had together. Especially, I want to thank Dr. Hsiang-Huang Wu and Lai-Huei Wang for sharing their opinions on many interesting topics with me.

Thank you to the entire ECE Department at University of Maryland. Thank you to all the staff and faculty that I worked with, especially that of Graduate Studies Office, Payroll Offices, UMIACS Business Offices for their constant help, advice, and support.

I want to say thank you to my close friends I met at University of Maryland—Changjiang Zhou, Xiaopeng Song, Kai Zhong, Dongquan Shen, Xinyi Chen, Qingtan Wu, Yi Niu, Hao Chen, Qingzun Wu, Shichun Wu, Di Wu, Yirui Pan, Xuan Liu, Xing Xu, Haoyu Wang, Tao Tao... I can not name everyone of them, but I will always remember everyone. You made my stay at University of Maryland so enjoyable.

I want to thank Youngho Cho, a PhD student supervised by Prof. Gang Qu for his valuable and constructive suggestions on my research works.

Finally and most importantly, my heartfelt appreciation goes to my parents Fulin Zhou and Jibin Sun, my wife Wei, my son BiuBiu and every member of my family for their unrequited love and continuous support. Mom and Dad, you are wonderful parents I

will try to imitate. I would like to thank my wonderful wife Wei for her support, encouragement, patience and unwavering love. I also want to thank my little son BiuBiu who is still behind mom's belly for bringing me so much pleasure in the last year of my PhD study.

*The research underlying this thesis was supported in part by the Laboratory for Telecommunications Sciences and Texas Instruments.*

# Table of Contents

List of Figures	ix
List of Tables	xi
1 Introduction	1
1.1 Overview	1
1.2 Contributions	3
1.2.1 Scheduling Event Detection on Wireless Sensor Networks	4
1.2.2 Scheduling Multithread Applications on Multicore GPPs	5
1.2.3 Exposing Intra- and Inter-Actor Parallelism for Implementation of DSP Applications	6
1.2.4 Scheduling Parallelized Synchronous DSP Systems on Multicore PDSPs	8
1.2.5 A Cross-platform Design Flow for DSP Applications	9
1.3 Dissertation Organization	10
2 Scheduling Event Detection on Wireless Sensor Networks	12
2.1 Introduction	12
2.2 Related Works	15
2.3 Proposed Adaptive Detection Scheme	17
2.3.1 Problem Statement	17
2.3.2 Overview	18
2.3.3 Training Process	21
2.3.4 Local Decision Rule	24
2.3.5 Final Decision Rule	29
2.3.6 The Extensible Work	30
2.4 Experiment Methodology	31
2.4.1 Experimental Setups	31
2.4.2 Energy Efficiency under One Set of Parameters	33
2.4.3 Impact of the Key Parameters	34
2.5 Summary	38
3 Scheduling Multithread Application on Multicore GPPs	40
3.1 Introduction	41
3.1.1 Related Work	42
3.1.2 Main Contributions	43
3.2 Motivation Example	44
3.3 Preliminary	45
3.3.1 Performance Analysis	45
3.3.2 Processor Model	46
3.3.3 Power and Thermal Model	47
3.3.4 Problem Formulation	48
3.4 Analytical Solutions	49

3.4.1	Steady State Throughput . . . . .	49
3.4.2	Local Optimal Frequency . . . . .	51
3.5	Scheduling Framework . . . . .	54
3.6	Experiment Results . . . . .	57
3.6.1	Experimental Setup . . . . .	57
3.6.2	The Throughput of Static and Dynamic Scheduling . . . . .	58
3.6.3	Completion Time Reduction . . . . .	60
3.7	Summary . . . . .	60
4	Exposing Intra- and Inter-Actor Parallelism for Implementation of DSP Applications . . . . .	63
4.1	Introduction . . . . .	64
4.2	Related Work and Background . . . . .	67
4.2.1	Core Functional Dataflow . . . . .	67
4.2.2	Targeted Dataflow Interchange Format . . . . .	68
4.2.3	Related Work . . . . .	68
4.3	TDIF-PPG Design Flow . . . . .	70
4.4	Parallel Processing Group . . . . .	74
4.4.1	Model Description . . . . .	74
4.4.2	Application Programming Interfaces . . . . .	76
4.4.3	PPG Static Execution Flow . . . . .	77
4.4.4	Examples . . . . .	79
4.4.4.1	FIR Filter . . . . .	79
4.4.4.2	FFT . . . . .	81
4.4.5	Discussions . . . . .	83
4.5	Experiments . . . . .	83
4.6	Summary . . . . .	89
5	Scheduling Parallelized Synchronous DSP Systems on Multicore PDSPs . . . . .	91
5.1	Introduction . . . . .	92
5.2	Background . . . . .	95
5.2.1	Dataflow Interchange Format . . . . .	95
5.3	Problem Statement . . . . .	96
5.4	Mixed Integer Programming Solution . . . . .	101
5.4.1	Computation Usage Graph . . . . .	102
5.4.2	Mixed Integer Programming Formulation . . . . .	103
5.5	List Scheduling Solution . . . . .	105
5.5.1	Story Scheduling Overview . . . . .	106
5.5.2	Algorithm Description . . . . .	107
5.5.3	Example . . . . .	108
5.6	Two Phase Scheduling Framework . . . . .	109
5.7	Related Work . . . . .	112
5.8	Experiments . . . . .	114
5.9	Summary . . . . .	120

6	A Cross-platform Design Flow for DSP Applications	121
6.1	Introduction	122
6.2	Background	124
6.2.1	Lightweight Dataflow	124
6.3	From simulation to implementation	126
6.3.1	Step 1: System Formulation	126
6.3.2	Step 2: System Validation and Profiling	129
6.3.3	Step 3: System Optimization	129
6.3.4	Step 4: System Verification and Instrumentation	132
6.3.5	Determining Buffer Sizes	133
6.3.6	Discussion	134
6.4	Case Study 1 - CPU/GPU	135
6.4.1	Simulation	137
6.4.2	Implementation	139
6.5	Case Study 2 - Multicore PDSP	144
6.5.1	Simulation	145
6.5.2	Implementation	147
6.5.2.1	Cross-Platform Implementation	148
6.5.2.2	Scheduling and Mapping	151
6.6	Summary	154
7	Conclusions and Future Work	156
7.1	Distributed Signal Processing Systems	156
7.2	Parallel Signal Processing Systems	157
7.2.1	Scheduling from Physical Aspects	157
7.2.2	Scheduling from Models of Computation	158
	Bibliography	161

## List of Figures

2.1	Wireless sensor network for event detection. . . . .	13
2.2	The hybrid and adaptive detection schemes (dashed steps is for adaptive scheme only). . . . .	20
2.3	Training process example. . . . .	23
2.4	Comparison of the average energy consumption per node per period over 100 independent trials. . . . .	34
2.5	Impact of node's detection accuracy on performance. . . . .	36
2.6	Energy savings on systems with different ratio of non-communication energy in total energy consumption. . . . .	38
3.1	A WID $V_{th}$ variation map for a 16-core processor in (a). The corresponding $f_{max}$ and $P_s$ map in (b) [43]. . . . .	47
3.2	High-level thermal model of a multicore processor [63]. . . . .	51
3.3	Dynamic scheduling framework . . . . .	55
4.1	TDIF-PPG design flow. . . . .	71
4.2	Group owner FSM and state description table for static execution flow. . .	78
4.3	Group member FSM and state description table for static execution flow. .	79
4.4	Mapping data parallelism from an FIR filter to PPG-based actor design. .	80
4.5	PPGs in an FFT actor. . . . .	82
4.6	The mp-sched benchmark for SDR. . . . .	84
5.1	An example of an assignment-constrained FP-PAS instance. . . . .	99

5.2	Solutions for the FP-PAS instance. . . . .	100
5.3	Result of applying the story scheduling algorithm on the example of Fig. 5.1. . . . .	109
5.4	Two-phase scheduling framework for the FP-PAS problem. . . . .	111
5.5	Generated AAF and $AAF'$ for actor 1. . . . .	116
6.1	Dataflow graph of an image processing application for Gaussian filtering. . . . .	136
6.2	A typical GPU architecture. . . . .	141
6.3	Block diagram of TI TMS320C6678 8-core PDSP device. . . . .	144
6.4	An illustration of the mp-sched benchmark. . . . .	145
6.5	Multithreaded FIR filter for PDSP implementation. . . . .	148
6.6	Parallel FFT actor implementation using TDIF. . . . .	151

## List of Tables

2.1	Key parameters in our adaptive scheme. . . . .	19
2.2	Simulation results for different number of nodes $K$ and observations $T$ . . .	37
2.3	Energy saving of different detection accuracy requirement. . . . .	37
3.1	Comparison of the three different strategies. . . . .	45
3.2	Optimal steady state throughput of 16-core processor for problems of different switching activity . . . . .	57
3.3	Dynamic scheduling average throughput for different problem size . . . .	57
3.4	Performance (completion time) gain with different $F$ and number of cores. .	62
4.1	Execution time comparison for sequential FIR filter and parallel FIR filter implementation on different input sizes. . . . .	85
4.2	Execution time and latency comparison among sequential FFT, parallel FFT using 2 cores and parallel FFT using 3 cores. The results are compared for different input sizes. . . . .	86
4.3	Execution time and latency comparison among the 4 schedules. . . . .	89
5.1	Performance of both work flows on randomly generated SDF graphs. . . .	118
5.2	Performance of both TPDF work flows on an image registration application. .	120
6.1	Execution time of the <code>gaussian_filter</code> (GF) actor and the Gaussian filtering application (App) during simulation. . . . .	140

6.2	Execution time of the <code>gaussian_filter</code> (GF) actor and the Gaussian filtering application (App) in simulation and GPU-accelerated implementation. . . . .	143
6.3	Execution time comparison between sequential FIR in simulation and parallel FIR implementations for different input sizes. . . . .	150
6.4	Execution time and latency comparisons between Seq-FFT in simulation, and the two parallel FFT implementations $I_a$ and $I_b$ . . . . .	152
6.5	Execution time and latency comparisons among 3 different scheduling and mapping schemes and simulation for the mp-sched benchmark. . . .	154

# Chapter 1

## Introduction

### 1.1 Overview

As the landscape of signal processing technologies becomes increasingly diverse, a wide variety of platforms, including General Purpose Processors (GPPs), Graphics Processing Units (GPUs), multicore Programmable Digital Signal Processors (PDSPs), Wireless Sensor Networks (WSNs), and heterogeneous combinations of these platforms have been emerging [4]. There are two major trends for the evolution of signal processing platforms. One is *distributed systems*. A distributed system consists of multiple autonomous computation nodes that communicate through a wired, wireless or hybrid wired/wireless network. The computation nodes interact with one other in order to achieve a common goal. The other one is *parallel systems*. In this context by a parallel system we mean a system that involves one more parallel processing devices (PPDs), where by a PPD, we mean a processor that includes multiple execution units (“cores”) on the same integrated circuit. Performance gains from parallel systems have been coming more from increasing the number of cores on a single die rather than from increasing the clock frequencies of individual cores. Distributed systems and parallel systems are typically very different in terms of hardware and operational structure, power consumption characteristics, and computational capability.

At the same time, signal processing applications, such as video and audio pro-

cessing, wireless communication, environmental monitoring, etc., have been evolving continually over the past decades: e.g., from H.264 [66] to H.265 [23], and from Global System for Mobile (GSM) [48] to Long Term Evolution (LTE) [34], to name a few. Implementation constraints for such applications, such as those involving power consumption, accuracy, latency and throughput, have been changing steadily as these applications have evolved.

The development of signal processing systems involves implementation of the appropriate algorithms using *programming models* that are dependent on (e.g., CUDA [65]) or independent of (e.g., MPI [20], OpenMP [13], Pthreads [49] and OpenCL [30]) of the target hardware platforms. The programming model provides guidelines for the developer to specify the computational tasks involved in the application along with the communication among tasks. In this context, a task can be viewed as a sequence of operations that utilizes the hardware resources on the target platform.

A schedule is a mapping and ordering of the tasks onto the hardware resources of the target platform. Scheduling has a strong influence on key implementation metrics, including latency, throughput, and buffer memory requirements, and is therefore one of the most critical parts in design flows for signal processing systems. The scheduling problem is NP hard in many contexts (e.g., see [18]). In practice, heuristics or exact algorithms are applied to derive scheduling solutions based on the the compilation time budget and the level of schedule quality desired.

There are many different formulations of scheduling problems. In this thesis, we first consider a general formulation, where an instance of the scheduling problem is described as four terms  $M$ ,  $P$ ,  $N$ , and  $O$ . Here,  $M$  is an acyclic dataflow graph model of

the application, which specifies the partially ordered tasks in the algorithm, along with the data dependencies among the tasks.  $P$  is the architecture model, which specifies the type, functionality, quantity and organization of the hardware resources in the target platform.  $N$  is a set of constraints that the signal processing system must be designed not to violate.  $O$  is a set of optimization objectives, such as latency minimization or throughput maximization, for the scheduling problem. Given an instance  $I = (M, P, N, O)$  of the scheduling problem, a solution to the scheduling problem is a mapping and ordering of tasks and dataflow graph edges in  $M$  onto the processing and communication resources of  $P$ . Such a solution is said to be *feasible* if it satisfies  $N$ , and intuitively, the quality of a feasible schedule is assessed based on how well it performance under the objectives specified by  $O$ .

## 1.2 Contributions

Signal processing systems implemented on hardware platforms typically involve multiple levels of processing and communication hierarchy, such as network-level, chip-level, and processor-level in a structural context, and application-level, subsystem-level, component-level, and operation- or instruction-level in a behavioral context. In this thesis, we target scheduling issues that carefully address and integrate implementation considerations at different levels of these structural and behavioral hierarchies.

### 1.2.1 Scheduling Event Detection on Wireless Sensor Networks

A wireless sensor network (WSN) consists of spatially distributed autonomous sensors to monitor physical or environmental conditions, such as temperature, sound, pressure, etc., and to cooperatively pass their data through the network to selected destinations. Among the numerous applications of WSNs, event detection is important and of wide interest. In a typical WSN for event detection, the sensor nodes are deployed in the field of interest to monitor the occurrence of an event or the presence of a target so that appropriate action can be taken.

In the class of event detection systems that we study in this thesis, there are three main operations for the sensor nodes to perform: data collection, data processing and data transmission. The sensor nodes collect data from the environment by utilizing their sensing devices and then process the raw data to extract meaningful information before transmitting it through the network to a fusion center (FC). At the FC, a final decision is made on whether or not the event has occurred. This procedure is normally repeated periodically, where the period is referred to as the *detection period*.

In our first contribution, we develop a detection scheme that schedules and maps event detection applications on WSNs. We consider the impact of environment noise, and other aspects, such as design variation and device ageing at the chip-level. We design a training process to measure the error probabilities of each sensor in the field. We propose an energy efficient adaptive scheme for event detection at the network-level, considering the discrepancies among sensor nodes' sensing error in a WSN. Specifically, we configure each sensor node in the most energy efficient way subject to the constraint that

the detection requirement must be satisfied. As technology improves, the complexity and computational capacity of sensor nodes are increasing. More and more intelligent systems are expected to emerge with such advances in technology. Our adaptive scheme can be viewed as an example of an intelligent system in a WSN for event detection. Simulation results show that our simple approach saves significant amounts of energy when system parameter settings vary.

### 1.2.2 Scheduling Multithread Applications on Multicore GPPs

Process variation becomes a major design challenge as technology scales. Manufactured dies exhibit large differences in *maximum operating frequency* ( $f_{max}$ ) and leakage power — both *die-to-die* (D2D) and *within die* (WID). It is also expected that WID process variations will manifest themselves as *core-to-core* (C2C)  $f_{max}$  and leakage power variations when cores within a single die become small enough [43]. When multicore processors are globally clocked,  $f_{max}$  is limited by the slowest core in the die and therefore fails to reach the throughput potential when each core operates at its individual maximal clock frequency.

Thermal issues are another limiting factor in the speed at which processor cores can run. When hundreds or thousands of cores are put into a single die, they result in large chip power density. High on-chip temperature not only increases the cost of packaging and cooling, it also has a strong impact on circuit reliability, and may cause permanent damage. Dynamic thermal management (DTM) techniques designed for single-core processors do not work well for multicore processors due to spatial temperature variations

and thermal influences among cores.

We consider the impact of both temperature and process variations on the scheduling problem for multicore processors, and we consider this detailed scheduling problem at both the chip-level and processor-level. We formulate the problem and derive analytical solutions on how to select each core's speed to minimize an application's completion time. We propose an efficient dynamic scheduling strategy that can achieve  $1.31X$  speedup against conventional globally-synchronized systems, and we show that this level of speedup —  $1.31X$  — coincides with an upper bound on the achievable throughput.

### 1.2.3 Exposing Intra- and Inter-Actor Parallelism for Implementation of DSP Applications

As multicore processor technology evolves, increasing numbers of processors are integrated into system-on-chip (SoC) devices for signal processing system implementation. The trend towards multiprocessor SoCs (MPSoCs) is motivated by the performance gain from efficient parallel execution of programs. This performance gain is determined in part by the amount of parallelism exposed from the program.

In the third contribution of this thesis, we have introduced a new dataflow based design flow, called TDIF-PPG, for integrating graph level parallelism and actor level parallelism in MPSoC software optimization for DSP applications. Our approach is based on a new model, called the parallel processing group (PPG), for actor design, and an associated new plug-in to the targeted dataflow interchange format (TDIF) environment. This plug-in allows designers to express parallelism within actor designs, and integrate such

intra-actor parallelism with the graph level parallelism that is already exposed in TDIF.

There are four layers in the TDIF-PPG design flow. Layer 1 is the system layer (application-level): the given DSP application is modeled as a core functional dataflow (CFDF) graph [54] using the DIF language. Layer 2 is the actor interface layer (component-level): actor interface specifications, including information about input and output ports, actor parameters, and CFDF modes, for individual actors are provided using the TDIF language. Layer 3 is the platform-independent mode specification (PIMS) layer (operation-level): the Parallel Processing Group (PPG) model [92] guides the programmer in exposing actor level parallelism within the functional specification for each mode of an actor. Layer 4 is the actor implementation layer (operation-level): generic actor specifications from the PIMS layer are integrated with optimized platform-specific PPG API and runtime implementations.

Additionally, we have motivated several directions for future work to help strengthen the utility of PPG-based actor design and integration. These include exploration of algorithms for automated scheduling of dataflow graphs that employ PPG-based parallel actor implementations; accurate and efficient functional simulation of PPG-based designs for early-stage DSP system validation; and experimentation on other kinds of state-of-the-art digital signal processing platforms.

## 1.2.4 Scheduling Parallelized Synchronous DSP Systems on Multicore PDSPs

Increases in computational power from clock frequency improvements have slowed. The trend toward MPSoC devices is motivated by the performance gain from simultaneous utilization of multiple processors for parallel execution of software systems. An application can be divided into tasks, each representing a piece of the computation performed by the overall software system. By a *parallel task* in this context, we mean a task that has some inner parallelism and whose execution can be performed by multiple processors. Many techniques have been created for recognizing parallel tasks and exploiting their inner parallelism (e.g., see [13, 50, 32]).

The performance of a software system composed of parallel tasks depends heavily on the scheduling of those tasks onto the targeted MPSoC. Each task requires different amounts of hardware resources for execution. The challenges of scheduling tasks efficiently include allocating hardware resources for competing tasks in such a way that the task demand is satisfied.

In our fourth contribution of this thesis, we formulate a new type of parallel task scheduling problem called Parallel Actor Scheduling (PAS) for MPSoC mapping of DSP systems that are represented as synchronous dataflow (SDF) graphs. This formulation is developed as a natural extension of the work in Chapter 1.2.3. In contrast to traditional SDF-based scheduling techniques, which focus on exploiting graph level (inter-actor) parallelism, the PAS problem targets the integrated exploitation of both intra- and inter-actor parallelism for platforms in which individual actors can be parallelized across multiple

processing units.

We first address a special case of the PAS problem in which all of the actors in the DSP application or subsystem being optimized can be parallelized. For this special case, we develop and experimentally evaluate a two-phase scheduling framework with two work flows — particle swarm optimization with a mixed integer programming formulation, and particle swarm optimization with a fast heuristic based on list scheduling. We demonstrate that our FP-PAS targeted scheduling framework provides a useful range of trade-offs between synthesis time requirements and the quality of the derived solutions.

### 1.2.5 A Cross-platform Design Flow for DSP Applications

As embedded processing platforms become increasingly diverse, designers must evaluate trade-offs among different kinds of devices such as GPPs, graphics processing units (GPUs), multicore programmable digital signal processors (PDSPs), and field programmable gate arrays (FPGAs). The diversity of relevant platforms is compounded by the trend towards integrating different kinds of processors onto heterogeneous multicore devices for DSP (e.g., see [4]). Such heterogeneous platforms help designers to simultaneously achieve manageable cost, high power efficiency, and high performance for critical operations. However, there is a large gap from the simulation phase to the final implementation. Simulation is used extensively in the early stage of system design for high-level exploration of design spaces and fast validation. In contrast, in the implementation phase, there is strong emphasis on platform dependent issues, performance centric optimization, and tuning low-level implementation trade-offs. A seamless design flow is needed to help

designers effectively bridge this gap.

Based on *Core Functional Dataflow (CFDF)* [56] semantics, two complementary tools have been developed in recent years. First, the *lightweight dataflow (LWDF)* programming methodology [69] provides a “minimalistic” approach for integrating coarse grain dataflow programming structures into DSP simulation for fast system formulation, validation and profiling with arbitrary languages. Second, the *targeted dataflow interchange format (TDIF)* framework [71] provides cross-platform actor design support, and the integration of (1) code generation for programming interfaces, and (2) low level customization for implementations targeted to homogeneous and heterogeneous platforms alike.

In the fifth contribution of this thesis, we present a novel dataflow-based design flow that builds upon on both LWDF and TDIF to allow rapid transition from simulation to optimized implementations on diverse platforms. We present this design flow, and provide case studies to demonstrate its application.

### 1.3 Dissertation Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we introduce our low power event detection scheme for WSNs. In Chapter 3, we demonstrate our scheduling framework for throughput optimization on multiprocessor system-on-chip (MPSoC) devices under temperature and process variations. Our novel design flow for exposing both graph and actor level parallelism in DSP application performance optimization is presented in Chapter 4. In Chapter 5, a new scheduling problem associated with the

design flow proposed in Chapter 4 is defined, and solutions to this problem are developed. In Chapter 6, multi-scale scheduling techniques are integrated in a new dataflow-based, cross-platform design flow for DSP applications. Finally, we discuss conclusions and directions for future work in Chapter 7.

## Chapter 2

### Scheduling Event Detection on Wireless Sensor Networks

In this chapter, we study scheduling issues at the network-level and chip-level in the structural context of signal processing system design. Particularly, we propose a scheduling technique for event detection on WSNs. Our proposed technique leverages chip-level discrepancies in detection accuracy across different sensors in a WSN. In our proposed design methodology, we characterize such discrepancies through a collaborative training process, and we apply the resulting characterizations to optimize communication among sensor nodes (network-level communication). Such optimization is performed to enhance energy efficiency while guaranteeing overall detection accuracy. Material in this chapter was published in preliminary form in [94].

#### 2.1 Introduction

Event detection is one of the compelling applications of wireless sensor networks. Figure. 2.1. Detection is of interest for habitat monitoring, security, surveillance and other defence application. The goal is determine whether a target is present or absent within a period in sensing field. Among the numerous applications of the wireless sensor network (WSN), event detection is very popular and important. In a typical WSN for event detection, the sensor nodes are deployed in the field of interest to monitor the occurrence of an event or the presence of a target so further action can be taken. In the event detection

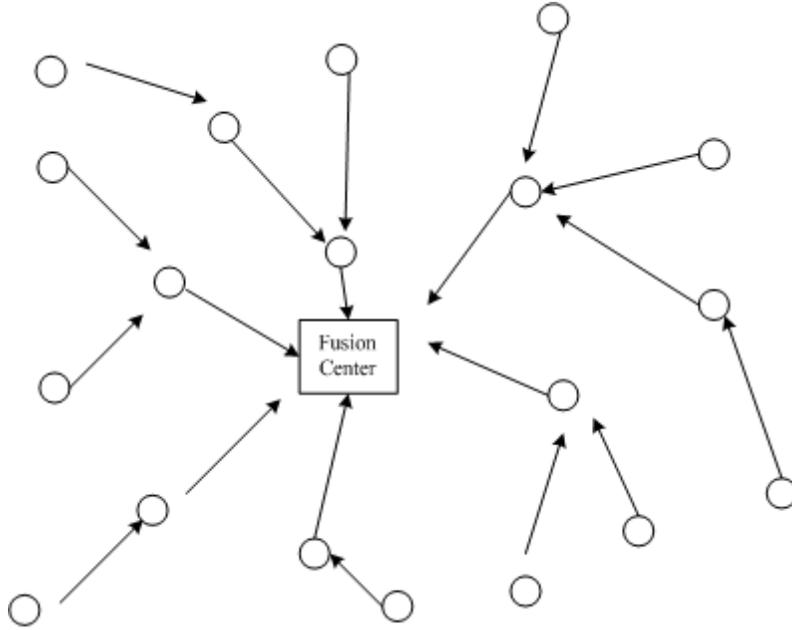


Figure 2.1: Wireless sensor network for event detection.

task, there are three main operations for the sensor node to perform: data collection, data process and data transmission. The sensor node collects data from the environment by utilizing its sensing device and then processes the raw data to extract meaningful information before transmitting it to a fusion center (FC) through the network. At the fusion center, a final decision is made on whether the event has happened or not. This procedure is normally repeated periodically, and the period is referred to as the detection period.

Normally, a detection scheme specifies (1) how each sensor node collects and processes data locally, (2) how the data is transmitted to the fusion center, and (3) how the fusion center makes the final decision. In the design of detection scheme for event detection on WSN, there are two essential aspects to be considered: one is the energy efficiency for the longer battery life, and the other one is the event detection accuracy which specifies a threshold that the error probability of the final result at the fusion center can not

exceed. Intuitively, these two aspects are against each other. Since a high detection accuracy requires more data to be collected, processed with and transmitted, resulting in a high energy consumption. Therefore, we need to consider the tradeoff between the two aspects in designing a good detection scheme.

In [87] two simple detection schemes are presented to explore this tradeoff in the scenario, where the data transmission energy dominates the sensor's total energy consumption which is a common situation in WSN. In a *centralized scheme*, at each sensor node, all the raw data without processing are transmitted back to the fusion center. This enables the fusion center to make the most accurate decision at the cost of large transmissions. In a *distributed scheme*, each sensor node is allowed to make a local decision first and then transmits its decision to the fusion center. The *distributed scheme* is energy efficient but loses much accuracy. Hence, how to reduce the data transmission energy while maintain the detection accuracy becomes an interesting problem.

Yu et al. [86] proposed a *hybrid scheme* that trades the detection accuracy for the data transmission energy saving. In their approach, each sensor node will send a local result (like the *distributed scheme*) if the sensor has a confidence on its local decision; otherwise the node will transmit all its collected and processed data to the fusion center (like the *centralized scheme*). Their simulation results showed that the hybrid scheme is the most energy efficient scheme to provide a required level of detection accuracy among three detection schemes. However, the *hybrid scheme* is based on the assumption that all individual sensor nodes in the system have the same sensing error probability, which could not be true in reality due to the environment noise [10, 27] and other aspects such as the imperfection (design variation) and the natural ageing process of the sensing device.

We investigate the same problem of how to design a detection scheme to minimize the energy consumption of WSN while maintain the system detection accuracy. But our work is based on a realistic model that sensor node may have variable false positive and false negative error probability. Our proposed *adaptive scheme* can leverage the discrepancy among individual sensor's detection accuracy, which is obtained from a collaborated training process, to allow each sensor to operate at its most energy efficient manner while guarantee the overall detection accuracy. A comprehensive set of simulations demonstrate that significant amount of energy (more than 60% on large scale WSN) can be saved compared with the best existing scheme.

The rest of chapter is organized as follows: Section 4.2 summaries existing works in wireless sensor network for event detection; Section 2.3 states the basic problem, describes our adaptive scheme, and compares it with the hybrid scheme; Section 2.4 discusses the experiment methodology adopted to prove the efficiency of our adaptive scheme; Section 5.9 concludes that our adaptive scheme is a simple intelligent system which is more energy efficient than the existing schemes.

## 2.2 Related Works

Before we elaborate our work, we briefly survey other works on the broad spectrum of research interests on WSN for event detection. In [73] multiple event detection problem is studied and solved by an collaborative network of binary sensor, which is capable of processing the information of multiple events simultaneously. In [80], the problem of coverage area extension has been investigated. The spatial diversity information from sen-

sensor reading is gathered. A collaborative signal processing method is proposed to reduce the environment noise, and improvements are achieved on both the reliability of detection and the reliable sensing region. A general coverage analysis model is built in [84], and applied on several simple decision fusion-based collaborative mechanisms, with the cons and pros being discussed. In [89], Zahedi et al. proposed a common modular analysis framework for sensor network from existing analysis models. Their method can speed up the process of design of a required sensor network. In [3], the problem of optimal node density analysis in detection region is inspected. A solution is explored on a sensor network in Gauss-Markov random field. In [45], Luo et al considers the sensor fault and environment noise in sensor network. An individual sensor node is allowed to communicate with its  $n$  neighbours and use their binary decisions to correct its own decision. The authors also propose a detection scheme with an optimal value of  $n$  to conserve power.

The above studies take advantages of collaboration among the sensor nodes to improve the detection reliability, coverage, and fault tolerance. However, they introduce, sometimes large, communication overhead. This will transfer to energy consumption overhead particularly in the scenario when communication energy dominates the sensor node's total energy consumption. In [87, 86], the authors assume that each sensor node performs event detection independently and only relies on each other during the multi-hop transmission. They formulate and investigate the energy related problems in WSNs deployed for event detection. In [87], they study the relationship between detection scheme's performance and its energy efficiency with focus on two representative detection schemes: the distributed scheme which consumes the least amount of communication energy but the most computation energy; and the centralized scheme that does

the opposite. In [86], they propose a hybrid scheme, which is the most energy efficient scheme to guarantee a given detection accuracy. However, their approach requires all the sensor nodes to have the same sensing error in the sensing field, which is unrealistic. In this chapter, we show not only how to remove this assumption, but also how further energy can be reduced when this assumption can be removed.

The advantages of collaboration among the sensor nodes have been demonstrated in [5,6,7,8,9,10] for improving the detection reliability, coverage, and fault tolerance, but it is not a energy efficient method due to the communication overhead. Therefore the sensor node is required to work independently in [1,2], and the energy saving comes from the tradeoff between the detection accuracy and the energy consumption. Our adaptive detection scheme is more related to [1,2], but our work is based on a realistic model that individual sensor has different sensing error in the field of interest.

## 2.3 Proposed Adaptive Detection Scheme

### 2.3.1 Problem Statement

We consider the field where an event of interest happens with a prior probability  $p$ . Let  $H = H_1$  and  $H = H_0$  denote the event happens and does not happen, respectively. A WSN of  $K$  sensor nodes is deployed in the field to detect periodically whether the event has happened or not. Each sensor node will independently collect data and convert each sensing sample to a 0 or a 1. For sensor  $i$ , there are two sensing error probabilities associated with this conversion: *false positive* error probability  $p_{1_i}$  is the probability that node  $i$  senses a 1 when the event has not happened; *false negative* error probability  $p_{0_i}$  is

the probability that node  $i$  senses a 0 when the event has happened. During each period, a sensor node can collect up to  $T$  samples, process them if necessary, and then transmit the sample data to the fusion center through the WSN. The fusion center will make a decision,  $H'$ , on whether the event has happened ( $H' = H_1$ ) or has not happened ( $H' = H_0$ ) during this period. The detection is accurate if  $H' = H$  and otherwise a detection error occurs.

A detection scheme includes two decision rules. At each sensor node, a *local decision rule* will guide the sensor node to determine the number of samples to be collected during each period and how to process the sensing data to information that will be transmitted to the fusion center. At the fusion center, a *final decision rule* will evaluate the information from all the sensor nodes and reach a decision on whether the event happened during each period. The WSN's detection accuracy is measured by its *detection error probability*, the probability that a final decision error occurs. We study the following problem: *how to design a detection scheme to minimize the WSN's total energy consumption while satisfying a given detection accuracy requirement.*

### 2.3.2 Overview

After the sensor deployment and the network setup, each sensor's sensing error probabilities  $p_{1_i}$  and  $p_{0_i}$  are measured by a training process in our adaptive detection scheme. Together with top four known parameters in Table I, the parameters  $N_{0_i}$  and  $N_{1_i}$  for each sensor node are calculated by a probability analysis framework at fusion center. It is designed to satisfy the system accuracy requirement.  $N_{0_i}$  and  $N_{1_i}$  are used to determine when sensor node  $i$  will stop collecting and processing data, and make a local decision,

Table 2.1: Key parameters in our adaptive scheme.

$K$	The total number of the sensor node in the field of interest.
$T$	The maximum samples collected in the field of interest for each sensor node.
$\delta$	The upper bond of final decision error probability.
$p$	The prior probability the event happens.
$p_{1i}, p_{0i}$	The error probabilities of a sensor node $i$ 's sensing.
$N_{1i}, N_{0i}$	The parameters for sensor node $i$ to decide its local decision.
$\xi_{1i}, \xi_{0i}$	The error probabilities of the local decision of sensor node $i$ .

and then transmit it back to the fusion center. For example, when sensor node  $i$  samples 1 for  $N_{1i}$  times, it will stop collecting and processing data and report one bit 1 as the local decision, indicating that the event has happened during the detection period, and when sensor node  $i$  samples 0 for  $N_{0i}$  times, it reports the event has not happened by sending back one bit 0 to the fusion center, Otherwise, it will report its total  $T$  observations in terms of the number of 1's ( $n_i$ ) it has observed by transmitting  $\lceil \log n_i \rceil$  bits to the fusion center. When the fusion center receives the data from all sensor nodes, it makes a final decision on whether the event has happened in the detection period.

Although the adaptive detection scheme is similar to the hybrid detection scheme [86] in the parameter names and the analysis methods, they are based on totally different models. As can be observed in Figure. 2.3.2, the hybrid detection scheme [86] does not consider the impact of environment noise, and other aspects such as design varia-

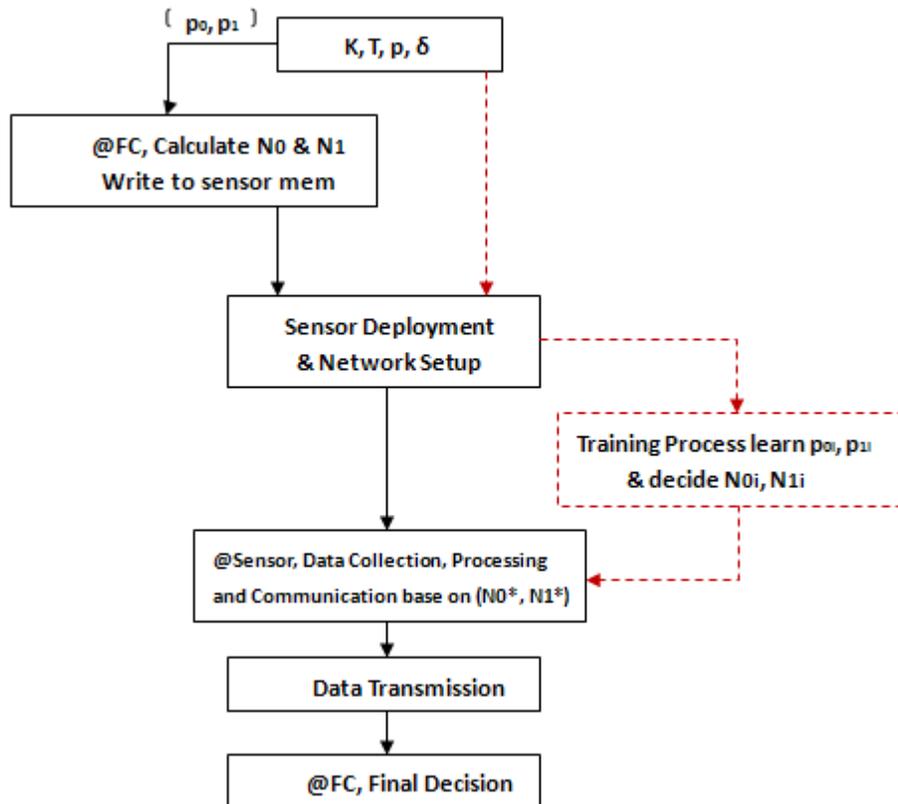


Figure 2.2: The hybrid and adaptive detection schemes (dashed steps is for adaptive scheme only).

tion and device ageing. It makes an unrealistic assumption that sensor's  $p_0$  and  $p_1$  are known before deployment, and have the same values for all sensor nodes. As a result the system detection accuracy can not be guaranteed or energy more than necessary will be consumed. In either case, it fails to minimize the WSN's energy consumption with a guaranteed detection accuracy.

However, the training process in our adaptive scheme consumes extra energy other than energy for event detection. So how to design a training process to obtain accurate  $p_{1_i}$  and  $p_{0_i}$  while consuming an acceptable amount of energy is critical. The other important part of our adaptive scheme is the probability analysis framework, which is essential to assure the required system detection accuracy under the condition that the sensor node has variable  $p_{1_i}$  and  $p_{0_i}$ . It is divided into the local decision rule of how the sensor node makes the local decision, and the final decision rule of how the fusion center makes the final decision on whether the event has happened or not. In the rest of this section, we will elaborate the training process, the local decision rule, the final decision rule, the extensible work and the probability theories behind them.

### 2.3.3 Training Process

The training process is one of the most important constituent parts in our adaptive detection scheme. It is used to obtain the sensing error probabilities  $p_{1_i}$  and  $p_{0_i}$  of sensor nodes, which is the essential information for developing the local decision rule and the final decision rule. We now elaborate it step by step.

In the first step, after being deployed, the sensor nodes are required to transmit all

collected  $T$  samples in a detection period to the fusion center. The fusion center makes its decision on each observation by comparing the number of 1 and the number of 0 in the columns as the Fig 2.3.3 shows. We make here an assumption that the individual sensor node has the higher probability of getting a correct sample ( $0 < p_{0_i}, p_{1_i} < 0.5$ ). It implies that the majority of the sensor nodes will make correct samples in the most cases if  $K$  is large.

In the second step, the result from the fusion center is considered as a reference (shadowed part in Fig 2.3.3), and the transmitted data of each sensor is compared to the corresponding element of the reference. The number of 1's in the transmitted data of sensor node  $i$  while in the reference a 0 at the same observation point is denoted as  $S_{1_i}$ , and the number of 0's in transmitted data of sensor node  $i$  while in the reference a 1 at the same observation point is denoted as  $S_{0_i}$ . The ratios  $S_{0_i}/T$  and  $S_{1_i}/T$  are considered as the measurement samples for  $p_{0_i}$  and  $p_{1_i}$ , respectively. The true values of  $p_{0_i}$  and  $p_{1_i}$  are the expected values of their measurements.

In the third step, we constructs confidence intervals [14] to estimate the true values of  $p_{0_i}$  and  $p_{1_i}$ . Equation 2.1 is the general method to construct confident intervals. In Equation 2.1,  $n$  is the population size,  $c$  is the half length of the interval,  $M_n(X)$  is the sample mean,  $E(X)$  is the expected value of random variable  $X$ , and  $Var[X]$  is the variance of random variable  $X$ .

$$P[M_n(X) - c < E(X) < M_n(X) + c] \geq 1 - \frac{Var[X]}{nc^2} \quad (2.1)$$

In our case, the distributions of measurements of  $p_{0_i}$  and  $p_{1_i}$  are assumed to be

normal. Totally 500 measurement samples are recorded. Large population size is chosen for squeezing the confidence intervals and mitigating any error introduced in previous steps. Then the 99% confidence intervals for  $p_{0_i}$  and  $p_{1_i}$  of each node are constructed. The upper bond values of the intervals are chosen to represent the actual  $p_{0_i}$  and  $p_{1_i}$  to assure system detection accuracy.

Observations	1	2	...	T	1	2	...	T	...	T
Fusion Center	1	1	...	1	0	0	...	1	...	0
Sensor Node 1	1	1	...	1	1	0	...	1	...	1
Sensor Node 2	1	0	...	0	0	1	...	1	...	0
Sensor Node 3	0	1	...	0	0	0	...	1	...	0
...	...	...	...	...	...	...	...	...	...	...
Sensor Node K-3	1	1	1	1	0	0	...	1	...	1
Sensor Node K-2	0	1	...	1	0	0	...	0	...	1
Sensor Node K-1	1	1	...	1	1	1	...	1	...	0
Sensor Node K	0	0	...	0	0	0	...	1	...	0

Figure 2.3: Training process example.

Since all the sensor nodes will transmit all the collected data to the fusion center during the training process, this may incur a large amount of energy consumption. It should be applied carefully and only when necessary. One common scenario that a training process is recommended is when the sensor nodes in the WSN may have very different operating environment. For example, when detecting wild fire, data from a temperature sensor exposed to the sunshine will be much higher than the temperature information collected by a sensor in the shade of a tree. Similarly, the data collected by the same sensor may vary as the time changes. Temperature around noon will be higher in general

than temperature at midnight. We focus on such spatial variation in this chapter. Other applicable scenarios are discussed in Section III-F. Our experimental results show that the energy overhead of the training process can be considered negligible compared with the energy saving from the proposed adaptive detection scheme.

### 2.3.4 Local Decision Rule

From the above training process, the information of each sensor node's  $p_{0i}$  and  $p_{1i}$  are obtained. Based on this, the local decision rule is to determine  $(N_{0i}, N_{1i})$  for each sensor  $i$ , such that when sensor node  $i$  samples 1 for  $N_{1i}$  times or 0 for  $N_{0i}$  times, it will send back 1-bit local decision, otherwise it will transmit all  $T$  observations to the fusion center. We elaborate the calculation of  $(N_{0i}, N_{1i})$  in the following.

First we define the error probabilities  $(\xi_{1i}, \xi_{0i})$  of the local decision  $b_i$  made by the sensor node  $i$ .  $\xi_{1i} = P[b_i = 1|H_0]$  is the probability that sensor node  $i$  makes the local decision of one bit 1 indicating the event has happened while the event has not happened. Based on the local decision rule, this occurs when the sensor collects  $N_{1i}$  error 1's, which should be 0 if correct. Therefore, we have Equation 2.2 for  $\xi_{1i}$ . Similarly, let  $\xi_{0i} = P[b_i = 0|H_1]$  be the probability that the sensor node makes the local decision of one bit 0 indicating the event has not happened while the event has happened. And we have the Equation 2.3 for  $\xi_{0i}$ .

$$\xi_{1i} = \sum_{n=N_{1i}}^T \binom{n-1}{N_{1i}-1} (1-p_{0i})^{n-N_{1i}} p_{0i}^{N_{1i}} \quad (2.2)$$

$$\xi_{0i} = \sum_{n=N_{0i}}^T \binom{n-1}{N_{0i}-1} (1-p_{1i})^{n-N_{0i}} p_{1i}^{N_{0i}} \quad (2.3)$$

**Theorem 1.** the  $\xi_{1i}$  increases as the  $N_{1i}$  decreases and the  $\xi_{0i}$  increases as the  $N_{0i}$  decreases.

**[Proof]** Let  $N$  and  $N + 1$  are two possible values of  $N_{1i}$ , then we have following equations:

$$\xi_{1i}(N + 1) = \sum_{n=N+1}^T \binom{n-1}{N-1} (1-p_{0i})^n \left(\frac{p_{0i}}{1-p_{0i}}\right)^{N+1} \quad (2.4)$$

$$\xi_{1i}(N) = \sum_{n=N}^T \binom{n-1}{N-1} (1-p_{0i})^n \left(\frac{p_{0i}}{1-p_{0i}}\right)^N \quad (2.5)$$

From Equation 3.14, we know that  $\xi_{1i}(N)$  has one more element in its sum than  $\xi_{1i}(N + 1)$ , so we can rewrite it as:

$$\xi_{1i}(N) = I_N + \sum_{n=N+1}^T \binom{n-1}{N-1} (1-p_{0i})^n \left(\frac{p_{0i}}{1-p_{0i}}\right)^N \quad (2.6)$$

where  $I_N$  is the first element in the sum of  $\xi_{1i}(N)$ . Next we compares  $\xi_{1i}(N)$  and  $\xi_{1i}(N + 1)$  by subtraction:

$$\xi_{1i}(N) - \xi_{1i}(N + 1) = I_N +$$

$$\sum_{n=N+1}^T \left[ \binom{n-1}{N-1} - \binom{n-1}{N} \left(\frac{p_{0i}}{1-p_{0i}}\right) \right] (1-p_{0i})^n \left(\frac{p_{0i}}{1-p_{0i}}\right)^N \quad (2.7)$$

In Equation 2.7, all coefficients are greater than zero except for  $[\binom{n-1}{N-1} - \binom{n-1}{N}(\frac{p_{0i}}{1-p_{0i}})]$ .

We can not decide its value right now, but we know that  $\binom{n-1}{N-1} = \frac{(n-1)!}{(N-1)!(n-N)!}$  and

$\binom{n-1}{N} = \frac{(n-1)!}{(N)!(n-N-1)!}$ , so we have the following equation:

$$\begin{aligned} & [\binom{n-1}{N-1} - \binom{n-1}{N}(\frac{p_{0i}}{1-p_{0i}})] = \\ & \frac{(n-1)!}{(N-1)!(n-N-1)!} \left[ \frac{1}{n-N} - \frac{1}{N} \frac{p_{0i}}{1-p_{0i}} \right] \end{aligned} \quad (2.8)$$

where  $0 < p_{0i} < 0.5$ , so  $\frac{p_{0i}}{1-p_{0i}} < 1$ , meanwhile  $n \leq T$ , so  $\frac{1}{n-N} \geq \frac{1}{T-N}$ . As a result, we have:

$$\left[ \frac{1}{n-N} - \frac{1}{N} \frac{p_{0i}}{1-p_{0i}} \right] > \frac{1}{T-N} - \frac{1}{N} \quad (2.9)$$

where as long as  $N \geq T/2$ ,  $\frac{1}{T-N} - \frac{1}{N} \geq 0$ , so  $\xi_{1i}(N) - \xi_{1i}(N+1) > 0$ . Apparently  $N \geq T/2$  because  $N$  and  $N+1$  must be the possible values of  $N_{1i}$  and  $N_{1i}$  is the threshold to make the local decision. Evidently at least  $T/2 + 1$  samples must be made before the sensor node can make its local decision. So we prove that the  $\xi_{1i}$  increases as the  $N_{1i}$  decreases. Similarly, we can prove that the  $\xi_{0i}$  increases as the  $N_{0i}$  decreases. So we prove **Theorem 1** is right.

Hence, the bigger  $(N_{0i}, N_{1i})$  is, the more accurate the final decision is. Here we are interested in finding the optimal  $(N_{0i}, N_{1i})$  in order to just meet the the detection accuracy requirement.

The system accuracy requirement requires  $P_e \leq \delta$ , where  $P_e$  is the system error probability or the final decision error probability. We know that the system error probability can be calculated by

$$P_e = (1 - p) * P[H' = H_1|H_0] + p * P[H' = H_0|H_1] \quad (2.10)$$

where  $P[H' = H_1|H_0]$  is the probability that the final decision considers the event has happened while actually the event has not happened,  $P[H' = H_0|H_1]$  is the probability that the final decision considers the event has not happened while actually the event has happened.  $p$  is the prior probability that the event happens. Since  $0 \leq p, 1 - p \leq 1$ , to meet the system accuracy requirement, it suffices to set  $P[H' = H_1|H_0] = \delta$  and  $P[H' = H_0|H_1] = \delta$ . In other words, our  $(N_{0i}, N_{1i})$  needs to satisfy the two requirements at the same time. In order to find the optimal  $(N_{0i}, N_{1i})$ , we first assume  $(N_{0i}, N_{1i})$  is known, such that the sensor node  $i$  can decide to make its local decision or send back all  $T$  observations. Generally, we assume that among all sensor nodes, some of them send their local decisions, and the rest send back their  $T$  observations, with the error probability of the final decision being  $P_e$ . If we reduce the  $(N_{0i}, N_{1i})$  to let every sensor node make its local decision, the error probability of the final decision will increase. In other words, if we can find  $(N_{0i}, N_{1i})$  to make sure  $P_e \leq \delta$  when every sensor node makes its local decision, the system accuracy requirement is satisfied in any other situations. The  $(N_{0i}, N_{1i})$  is what we need.

In the scenario where the final decision is made merely on the local decisions, the probabilities  $P[H' = H_0|H_1]$  and  $P[H' = H_1|H_0]$  are only related to the  $\xi_{1i}$  and  $\xi_{0i}$  of each sensor node. But if each sensor node is allowed to have different  $\xi_{1i}$  and  $\xi_{0i}$ , it would be too complicated to derive any analytical solution on the meaningful estimation. So we make a simplification here that all sensor nodes have the same  $(\xi_0, \xi_1)$ . In this case

we can use the binary hypothesis testing [60] to make the final decision by the following equations:

$$P[H' = H_1|H_0] = \sum_{k=\lceil \Gamma_D \rceil}^K \binom{K}{k} \xi_1^k * (1 - \xi_1)^{K-k} = \delta \quad (2.11)$$

$$P[H' = H_0|H_1] = \sum_{k=\lceil K-\Gamma_D \rceil}^K \binom{K}{k} \xi_0^k * (1 - \xi_0)^{K-k} = \delta \quad (2.12)$$

$$\Gamma_D = \frac{\ln \frac{1-p}{p} + K \ln \frac{1-\xi_1}{\xi_0}}{\ln \frac{(1-\xi_1)(1-\xi_0)}{\xi_1\xi_0}} \quad (2.13)$$

When the number of 1 among the local decisions exceeds  $\Gamma_D$ , the final decision is made that the event has happened ( $H' = H_1$ ), otherwise the event has not happened.

In three Equations 2.11-2.13, we have three unknown variables  $\{\xi_1, \xi_0, \Gamma_D\}$ . However, those equations of higher degree are hard to be solved directly. So the first two equations are changed to the inequalities  $P[H' = H_1|H_0] \leq \delta$  and  $P[H' = H_0|H_1] \leq \delta$ . And then the numerical method is adopted to find the optimal  $\xi_1, \xi_0$  which maximizes  $\xi_1 + \xi_0$  while satisfies the above inequalities, since that the larger error probability on the local result is allowed, the smaller  $N_{0i}$  and  $N_{1i}$  of sensor node  $i$  would be. Specifically, we try the all combinations of  $\xi_1, \xi_0$  in the range of  $(0, 0.5)$  with the precision of 0.01. Then from equations (1) and (2), we will be able to determine the  $\{N_{0i}, N_{1i}\}$  from the  $\{x_{0i}, x_{1i}\}$ . For the same reason, the numerical method is adopted to find the optimal  $\{N_{0i}, N_{1i}\}$  which minimizes  $N_{0i} + N_{1i}$ , since that the smaller  $N_{0i} + N_{1i}$  is, the larger probability sensor node  $i$  will make a local decision and save the energy on data transmission. All the combinations of  $\{N_{0i}, N_{1i}\}$  in the range of  $[1, T]$  with the precision of 1 are

testified.

### 2.3.5 Final Decision Rule

The final decision rule determines how the fusion center makes the final decision. In special cases when all sensor node makes its local decision, the binary hypothesis testing [60] is used to make the final decision. In general cases, we assume among  $K$  sensor nodes,  $s$  of them send local decisions of one bit 0,  $t$  send local decisions of one bit 1, the rest  $K - s - t$  sensor nodes send their  $n'_i$ 's, where  $s, t \geq 0$  and  $s + t < K$ . Let  $\Omega = n_1, \dots, n_{K-s-t}; 0, \dots, 1, \dots$  denote the message transmitted to the fusion center from the sensor nodes. For a given  $\Omega$ , the optimal final decision rule is to choose  $H' = H_1$  if

$$P[H_1|\Omega] \geq P[H_0|\Omega] \quad (2.14)$$

We know that  $P[H_1|\Omega] = pP[\Omega|H_1] \setminus P[\Omega]$  and  $P[H_0|\Omega] = (1-p)P[\Omega|H_0] \setminus P[\Omega]$  according to the Bayes' theorem. So the inequality (2.14) can be derived to

$$\frac{P[\Omega|H_1]}{P[\Omega|H_0]} \geq \frac{1-p}{p} \quad (2.15)$$

Where the probabilities  $P[\Omega|H_1]$  and  $P[\Omega|H_0]$  can be calculated by:

$$P[\Omega|H_1] = \prod_{i=1}^{K-s-t} P[n_i|H_1] * P[b=0|H_1]^s * P[b=1|H_1]^t \quad (2.16)$$

$$P[\Omega|H_0] = \prod_{i=1}^{K-s-t} P[n_i|H_0] * P[b=0|H_0]^s * P[b=1|H_0]^t \quad (2.17)$$

So we can compute  $P[\Omega|H_1] \setminus P[\Omega|H_0]$ :

$$\frac{P[\Omega|H_1]}{P[\Omega|H_0]} = \prod_{i=1}^{K-s-t} \frac{P[n_i|H_1]}{P[n_i|H_0]} * B_0^s * B_1^t \quad (2.18)$$

where  $B_0 = \frac{\xi_0}{1-\xi_1}$  and  $B_1 = \frac{1-\xi_0}{\xi_1}$ . And meanwhile we know that

$$P[n_i|H_0] = \binom{T}{n_i} p_{1i}^{n_i} (1 - p_{1i})^{T-n_i} \quad (2.19)$$

$$P[n_i|H_1] = \binom{T}{n_i} (1 - p_{0i})^{n_i} p_{1i}^{T-n_i} \quad (2.20)$$

From above explanations, the calculation of the inequality (2.14) is known, so is our final decision rule.

### 2.3.6 The Extensible Work

Although only the spatial variance in sensor node's  $p_{0i}$  and  $p_{1i}$  is considered in our adaptive scheme, our work can be easily extended to support temporal variance as well. For that we can design a special training process for monitoring the history of the transmitted data from sensor node to the fusion center. When we detect that some sensor node's data is consistently off the mark, an adjustment can be made on its  $N_0$  and  $N_1$  settings. For instance, if sensor node  $i$  keeps sending local decision of one bit 1 when the final decision is 0, an increase of  $N_{1i}$  will be made, and the new  $N_{1i}$  will be sent back to sensor node  $i$  for future detection; if sensor node  $i$  keeps sending local decision of one bit 0 when the final decision is 1, an increase of  $N_{0i}$  will be made and new  $N_{0i}$  will be sent back; if sensor node  $i$  keeps sending multiple bits of  $\lceil \log n_i \rceil$  when the final decision is 1, a decrease of  $N_{1i}$  will be made and new  $N_{1i}$  will be sent back; if sensor node  $i$  keeps

sending multiple bits of  $\lceil \log n_i \rceil$  when the final decision is 0, a decrease of  $N_{0i}$  will be made and new  $N_{0i}$  will be sent back. But for time limit, we have not implemented it in our current framework. We consider it as the work we shall explore in the future.

## 2.4 Experiment Methodology

We perform simulations to demonstrate the effective of the proposed adaptive detection approach in improving the energy efficiency of the hybrid scheme [86]. In this section, we describe the experiment setting and report the experimental results.

### 2.4.1 Experimental Setups

As we have mentioned, the WSN's total energy consumption of sensor node consists of energy for sensing (or data collection), data processing, and data communication in the wireless network. For comparison purpose, we adopt the parameters in [86] whenever possible.

For data collection energy, we assume the sensing device will be turn on and off periodically to sample the data. For one sampling, we denote the energy consumed is  $e_s = 10nJ$  as used in [86].

Data processing energy of sensor node depends on the amount of data to be processed and the processing time. Let the  $e_{cyc}$  be the energy consumption per cycle by the microprocessor in sensor node. And  $ne_{cyc}$  is the energy consumption when the data process takes  $n$  cycles. In many applications, sleep mode is used when the system is idle to save dynamic and static power. In that case, the energy overhead of waking up the system

also needs to be considered. We adopted the data processing energy for each observation  $e_p = 40nJ$  as used in [86].

Data communication energy is for transmitting and receiving data. It is proportional to the amount of the data to be transmitted and the transmitting energy is directly related to the distance between the sender and the receiver. We adopted  $e_r = 50nJ$  the energy for receiving 1 bit data from a neighbour and  $e_t = 400nJ$  the energy of transmitting one bit data over a unit distance [86]. A multi-hop communication is considered in our network model, where a greedy perimeter stateless routing (**GPSR**) algorithm in [31] is implemented.

More specifically, when sensor node  $i$  sends  $W_i$  bits of data to the fusion center, we can calculate the total transmitting and receiving energy consumed on this path as

$$E_{tx-i} = \sum_{j=1}^{l_i} e_t * d_j^2 * W_i, E_{rx-i} = \sum_{j=1}^{l_i} e_r * W_i$$

where  $l_i$  is the total length of the path, in number of hops, from node  $i$  to the fusion center, and  $d_j$  is the distance between the  $(j - 1)^{th}$  node and the  $j^{th}$  node on the path.

In our simulations, we consider the event of interest happening with probability  $p = 0.2$  in the field of a square shape, where a WSN of  $K$ , ranging from 8 to 128, sensor nodes are deployed randomly. Each sensor will collect up to  $T$  samples, which goes from 10 to 160, during each detection period. The sensor's Type I and Type II error probabilities  $p_0$  and  $p_1$  follow normal distributions with different parameters. And the WSN's detection accuracy requirement  $\delta$  varies from  $10^{-6}$  to  $10^{-1}$ .

## 2.4.2 Energy Efficiency under One Set of Parameters

We compare the average energy consumption per sensor node in one detection period of the same WSN using the hybrid detection scheme [86] and our proposed adaptive method, respectively. We first report the results on a representative parameter setting:  $p = 0.2$ ,  $\delta = 10^{-4}$ ,  $K = 8$ ,  $T = 40$ , and  $(p_0, p_1)$  follow normal distribution  $N_0(0.2, 0.1)$  and  $N_1(0.2, 0.1)$ , respectively. The impact of different parameters on the energy saving will be discussed later.

First, we numerically find  $\xi_0$  and  $\xi_1$  for the given value of  $K$ ,  $p$ , and  $\delta$  based on Equation 2.11-2.13. Then we randomly generate  $K = 8$  pairs of  $(p_0, p_1)$  based on their normal distribution models. For each  $(p_0, p_1)$  pair, we can obtain the value of  $N_0$  and  $N_1$  from Equation 2.2 and 2.3. In the hybrid scheme, the largest values of  $N_0$  and  $N_1$  will be used on all sensors to guide their local decisions. In the adaptive approach, each sensor will have a specific  $(N_0, N_1)$  pair computed by the fusion center based on its own  $(p_0, p_1)$ . Finally, we generate 4,000,000 observation points (or equivalently, 100,000 periods) where the event happens randomly with probability  $p = 0.2$ . For each observation point, a sensor node's collected data will be store as a 0 (event did not occur) or 1 (event occurred) based on its  $(p_0, p_1)$ .

Fig. 2.4 shows how the proposed adaptive detection scheme can improve the energy efficiency of the hybrid scheme from the data we obtained by repeating the above procedure 100 times. X-axis indicates these 100 trials and the Y-axis gives the average energy consumption per node per period in each trial. We see that on average, the hybrid scheme requires  $2.38 \times 10^4 nJ$  per node per period, while the adaptive approach need only

$0.84 \times 10^4 nJ$ , a reduction of **64.4%**. For the training process in the adaptive approach, we assume that all the sensor nodes will collect 2000 observation points for the first 500 periods and send all raw data to the fusion center. This energy overhead is  $1.06 \times 10^2 nJ$  per node per period and has been included in Fig. 2.4.

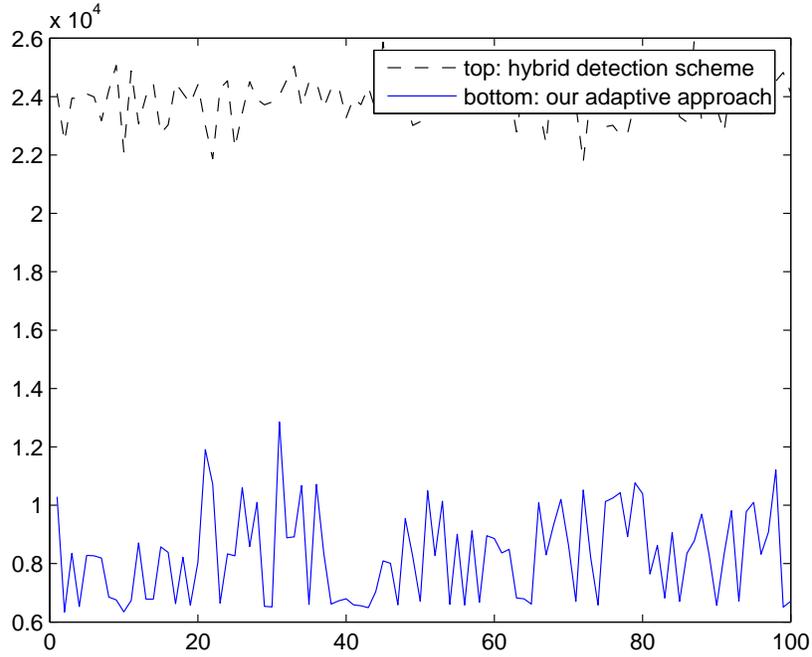


Figure 2.4: Comparison of the average energy consumption per node per period over 100 independent trials.

### 2.4.3 Impact of the Key Parameters

We now analyze the impact of various system parameters on the performance of our adaptive approach in terms of energy saving. More specifically, we will consider individual sensor’s detection accuracy  $(p_0, p_1)$ , scale of the WSN (number of sensors in the network  $K$ ), observation points in each period ( $T$ ), overall detection accuracy requirement

( $\delta$ ), and weight of communication energy in total energy consumption.

Firstly, we vary the values of the mean and standard deviation for Type I and Type II detection error at a sensor node  $(p_0, p_1)$ . Figure. 2.4.3 reports the energy saving of the adaptive approach over the hybrid detection scheme over 9 different settings, grouped by the values of the means of  $p_0$  and  $p_1$ . Let  $\mu_{p_1} = 0.1$ , for given  $\mu_{p_0}$  and  $\mu_{p_1}$ , the standard deviations are  $(\mu_{p_1} - \mu_{p_0})/10$ ,  $(\mu_{p_1} - \mu_{p_0})/5$ , and  $(\mu_{p_1} - \mu_{p_0})/3$ , from left to right. We see that as the mean of  $p_0$  decreases from 0.4 to 0.2, the energy saving decreases as well. This is because a smaller mean of  $p_0$  will result in a small  $N_0$  plus the event happens rarely  $p = 0.2$ , despite there is a deviation in  $N_0$ , all sensor node probably will make the local decision, leaving little advantage for the adaptive approach. Within each group, when we increase the standard deviation of  $p_0$  and  $p_1$ , the values of  $N_0$  and  $N_1$  for different sensors will have larger difference. Therefore, the adaptive approach will give more energy saving.

Secondly, for a fixed network setting, we increase the maximum number of observation points in each period, so the overall detection accuracy will increase at the cost of energy consumption as indicated in the top half of Table I. However, we see that the adaptive approach's energy increment is at a much slower pace. That is, the energy saving becomes more and more significant, from 30.4% to 73.7% as T increases from 10 to 160, as shown in the last column of Table II. Similar phenomenon happens when we increase the sensor nodes while keep the maximum number of observation points fixed for each period. When there are more sensor nodes, the discrepancy among  $(p_0, p_1)$  pairs and hence the  $(N_0, N_1)$  pairs becomes large, which will benefit the adaptive approach that takes advantage of such discrepancy.

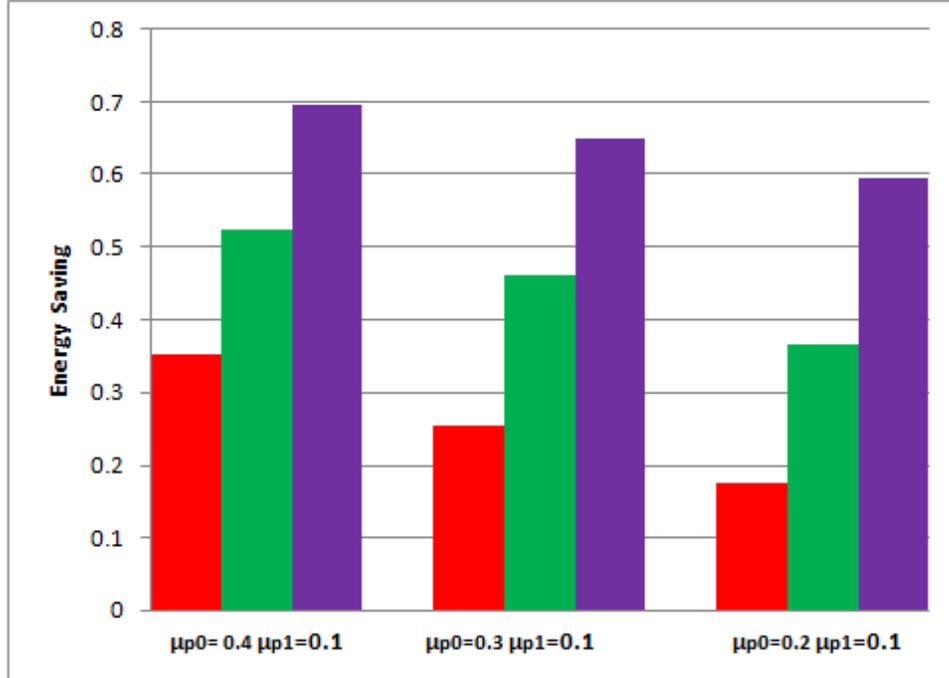


Figure 2.5: Impact of node's detection accuracy on performance.

Thirdly, Table III shows the energy savings when the overall detection accuracy requirement increases (or equivalently, when the overall detection error probability upper bound goes down from  $1.0 \times 10^{-1}$  to  $1.0 \times 10^{-6}$ ). The reason behind this is that to reach a higher detection accuracy, both  $N_0$  and  $N_1$  will increase for each node and become closer to  $T$ , leaving less space for the discrepancy among them and therefore the adaptive approach will be less effective. Nevertheless, we still have 53.1% energy saving in the case when the highest detection accuracy is required.

Finally, we consider the impact of the three main energy sources (data collection  $e_s$ , data processing  $e_p$ , and data communication  $e_t$  and  $e_r$ ). For each observation, the data collection and data processing energy (updating the counters that keep track of the number of 0's and 1's in the observed data, and comparing whether the threshold  $N_0$  and

Table 2.2: Simulation results for different number of nodes  $K$  and observations  $T$ .

$K$	$T$	$E_{adaptive}$ (nJ)	$E_{hybrid}$ (nJ)	saving
8	10	4.9e+03	7.00e+03	30.4%
8	20	5.64e+03	1.32e+04	40.2%
8	40	8.53e+03	2.39e+04	64.3%
8	80	1.25e+04	4.20e+04	70.1%
8	160	1.98e+04	7.57e+04	73.7%
8	40	8.53e+03	2.39e+04	64.3%
16	40	1.47e+04	4.73e+04	68.8%
32	40	9.46e+04	2.86e+05	69.1%
64	40	5.79e+05	1.93e+06	69.9%
128	40	1.16e+06	3.87e+06	70.7%

Table 2.3: Energy saving of different detection accuracy requirement.

$\delta$	$10^{-1}$	$10^{-2}$	$10^{-3}$	$10^{-4}$	$10^{-5}$	$10^{-6}$
saving	74%	71.7%	66.8%	64.3%	55.2%	53.1%

$N_1$  are reached) are roughly the same. So we consider the sum of  $e_s$  and  $e_p$  as the non-communication part. In Figure. 2.6, we vary the ratio of this non-communication energy from 1% to 99% of the total energy consumption and report the energy saving obtained from the adaptive approach. As a result, we see that we enjoy the most energy saving on the left when the ratio is small. This means when communication energy dominates the total energy consumption, our adaptive approach is more effective.

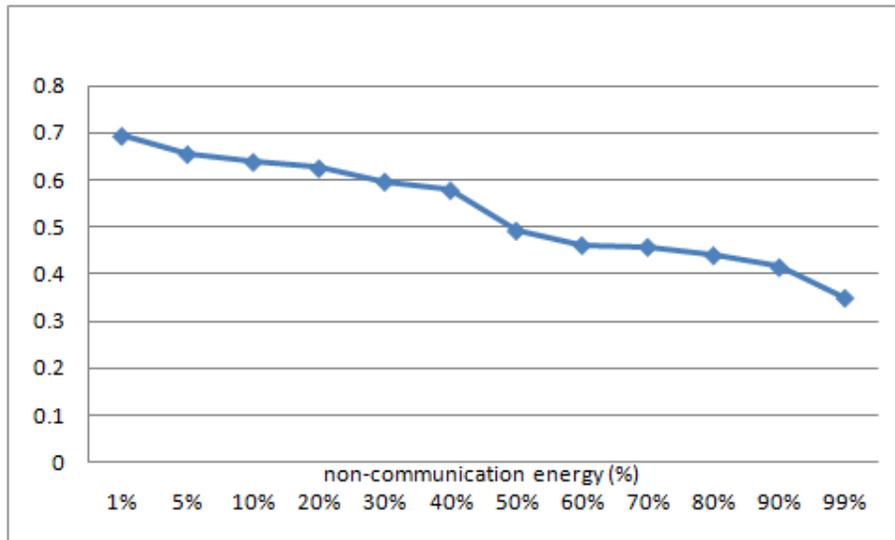


Figure 2.6: Energy savings on systems with different ratio of non-communication energy in total energy consumption.

## 2.5 Summary

In this chapter, we introduced a class of event detection application for WSNs, and for this class of applications, we defined a key scheduling problem whose objective is to minimize sensor node energy while maintaining detection accuracy. In this work, we focused on chip-level and network-level analysis of WSN structure. We developed a

novel detection scheme that efficiently maps the computation tasks and communication tasks to sensor nodes. Comprehensive simulation results show that our adaptive scheme can improve the energy efficiency of a state-of-the-art hybrid detection method by more than 60% on large scale WSNs.

## Chapter 3

### Scheduling Multithread Application on Multicore GPPs

In this chapter, we address scheduling issues at the chip-level and processor-level in a structural context. Particularly, we propose a novel scheduling framework for scientific computing applications on multicore GPPs. Our framework determines the speed at which each processor core operates and the number of threads assigned to each core. These determinations are made in an optimized manner to maximize throughput while taking into account chip temperature and process variations within the chip. Material in this chapter was published in preliminary form in [93].

Chip temperature has become an important constraint for achieving high performance on multicore processors with the popularity of dynamic thermal management techniques such as throttling that slow down the CPU speed. Meanwhile, the within-die process variation creates large discrepancy of maximum operating frequency and leakage power among different cores across the chip. In this chapter, we incorporate both temperature and process variations into the scheduling problem on multicore processor for throughput maximization. In particular, we study how to complete a large number of threads with the shortest time, without violating a given maximum temperature constraint, on the multicore processor where each core may have different frequency and leakage. We develop speed selection and thread assignment schedulers based on the notion of core's steady state temperature. Simulation results show that our approach is promising

to exploit the process and temperature variation for potential performance improvement on multicore processor. For a 16-core system, when the fully parallelized program's switching activity is less than 0.4 or the problem execution time is less than 6 seconds, we are able to achieve 31% speed-up over the system that uses the slowest core's speed as the frequency; when the switching activity is larger than 0.5 and half of the program can be parallelized our scheduler can reduce the total processing time by 4.3%.

### 3.1 Introduction

Technology scaling enables more and more processing cores accommodated within a single die area. Currently, CUDA architecture Fermi with 512 cores, has been put into market. The trend towards multicore processors is motivated by the performance gain compared to single-core processors when a budget on power or temperature or both is given. The performance is expected to further increase with the increasing number of cores if thread-level parallelism (TLP) can be fully exploited.

Process variation becomes a big design challenge as technology scales. Manufactured dies exhibit a large difference in *maximum operating frequency* ( $f_{max}$ ) and leakage power both *die-to-die* (D2D) and *within die* (WID). It is also expected that WID process variations will manifest themselves as *core-to-core* (C2C)  $f_{max}$  and leakage power variations when cores within a single die becomes small enough [43]. When multicore processors are globally clocked,  $f_{max}$  is limited by the slowest core in the die and therefore fails to reach the throughput potential when each core operates at its individual maximal clock frequency.

Thermal issue is another limiting factor for each core to run at its maximal speed. When hundreds or thousands of cores are put into a single die, they make the chip power density increasingly high. High on-chip temperature not only brings up the cost of packaging and cooling, it also affects severely reliability of the circuit and may cause permanent damage. Dynamic thermal management (DTM) techniques designed for single-core processors do not work well for multicore processors due to the spatial temperature variations and the thermal impact from core to core.

Applications that have high parallelism can benefit the most from multicore processors. Especially for computation bounded applications, massive small tasks (threads) are generated by the program with small data set and no data dependency among them, thus the memory bandwidth is not a issue. The threads carrying each task need to be assigned to cores for execution. The whole program is consider completed when the last thread/task is finished. We focus on such applications and more details will be given in Section 3.3.4.

### 3.1.1 Related Work

The impact of process variation on system throughput was first studied on throughput maximization of single-core processors [9, 77], then the study was extended to multicore processors [8, 43]. The impact of process variation on the throughput of several different multicore processors was analyzed and the sensitivities were compared in [8]. Lee et al. proposed a C2C variation model in multicore processor and studied how to choose the active cores when number of threads is smaller than that of cores for through-

put maximization[43]. However, none of the works considers the temperature variation at the same time.

On the other hand, thermal issue has also been widely studied on both single core and multicore processors. The relationship between chip temperature and leakage power on single-core processor was studied in [88]. Yeo et al. presented an analytical future temperature prediction model for multicore processors, and developed a migration policy based on the prediction model [85]. [12] formulated the multicore temperature-aware scheduling problem as an integer linear programming problem to minimize the hot spots and temperature gradients of MPSoC, and proposed an online heuristic to schedule real time tasks based on the core's current temperature and temperature history. Rao et al. investigated both steady state temperature and transient temperature's impact on the throughput of multicore processors [62]. Hanumaiah et al. proposed an optimal voltage and frequency control method for maximizing the throughput of multicore processors [25]. The throttling and thread migration techniques are incorporated in [24]. Similarly, none of the works mentioned above takes the process variation into consideration together with temperature variation.

### 3.1.2 Main Contributions

We consider the impact of both temperature and process variations on the scheduling problem on multicore processors. We formulate the problem and derive analytical solutions on how to select each core's speed to minimize the application's completion time. We propose an efficient dynamic scheduling strategy that can achieve 1.31 speedup

against the current globally synchronized system, which is the throughput upper bound.

### 3.2 Motivation Example

We demonstrate how a multicore processor performance, measured by throughput per core, can be affected by different strategies that control each core operating frequency and task assignment. We consider the simplest case of a 2-core processor model, where core 1 maximal speed is 1.0 and the faster core 2 is 1.12; and the leakage of core 1 and core 2 are 1.0 and 1.1, respectively. We assume that the highest temperature that the cores can operate is  $110^{\circ}C$ . We now compute the throughput of this processor while executing 200 units of workload, e.g. 200 identical threads each requires one unit of CPU time at the nominal speed 1.0.

First, strategy I runs both cores at the nominal speed 1.0 and assigns 100 threads on each core. Both core will complete their assignment after 100 units of time and the throughput is 1 thread per core per unit of time. Clearly we see that core 2 has the potential to run faster and produce more.

This leads us to strategy II which runs both cores at their respective maximal speed. 94 threads are assigned to core 1 and 106 to core 2 based on the ratio of their speeds. Core 1 will finish its assignment in 94 units of time. However, core 2 reaches  $110^{\circ}C$  after it completes 83 threads due to its higher dynamic and leakage power. The traditional "heat and run" method will then reduce core 2 speed by half to 0.56 and complete the remaining 23 threads. As a system, the 200 threads will be completed in  $\frac{83}{1.12} + \frac{23}{0.56} = 115$  units of time, resulting a throughput of  $\frac{200}{2 \times 115} = 0.87$ .

Table 3.1: Comparison of the three different strategies.

	Core 1 speed & workload	Core 2 speed & workload	Completion time	Throughput speed up
I	1.0 & 100	1.0 & 100	100	1.00
II	1.0 & 94	0.92 & 106	115	0.87
III	1.0 & 97	1.06 & 103	97	1.03

Strategy II fails to improve throughput because it aggressively and statically assigns more workload to the faster core. In strategy III, we propose to dynamically adjust the core speed based on temperature and assign workload accordingly. For this example, core 1 runs at its full speed, but core 2 will gradually reduce its speed. As a result, all the workload will be complete in 97 units of time with 97 and 103 threads assigned to core 1 and core 2, respectively. The throughput is improved to  $\frac{200}{2 \times 97} = 1.03$ .

Table 1 below summarizes the average speed and workload assignment to each core, the completion time and throughput of the system according to the three different strategies.

### 3.3 Preliminary

#### 3.3.1 Performance Analysis

A general program consists of serial part  $F$  and parallel part  $1 - F$ , which are executed separately by a single core and multicore processor respectively. So the program's

execution time can be expressed by

$$time = \frac{F}{S_{serial}} + \frac{1 - F}{S_{parallel}}$$

where  $S_{serial}$  is the speed at which serial part of program is being executed and  $S_{parallel}$  is the speed at which parallel part of program is executed.

With the single core speed (hence  $S_{serial}$ ) increase has already stopped due to power/thermal limit, the above formula suggests that the speedup relies on the decrease of  $F$  or the increase of  $S_{parallel}$ [79, 2]. In order to decrease  $F$ , the program needs to be rewritten by new parallel programming model to expose more parallelism inside the program. Meanwhile more and more simple cores are put on one die to increase  $S_{parallel}$ . Our chapter focuses on how to improve  $S_{parallel}$  with a fixed  $F$ , a problem that is equivalent to how to minimize the completion time of a fixed number of threads. The key idea is to leverage the process variation as described next.

### 3.3.2 Processor Model

Fig 3.1 depicts the process variation on a 16-core processor, where each core's maximum frequency  $f_{max}$  (normalized to the  $f_{max}$  of the slowest core) and leakage power  $P_s$  (normalized to the  $P_s$  of the least leaky core) are given [43]. We can see that both  $f_{max}$  and  $P_s$  have large variation from core to core. The fastest core is  $0.65X$  faster than the slowest core and the most leaky core is  $2.3X$  leakier than the least leaky one. Normally the WID process variation makes faster core also leakier.

We consider a processor with  $k$  cores, each core is independently clocked and can adjust its own operating frequency to  $f_j f_{max,j}$ , where  $f_j$  is the scaling factor in the range

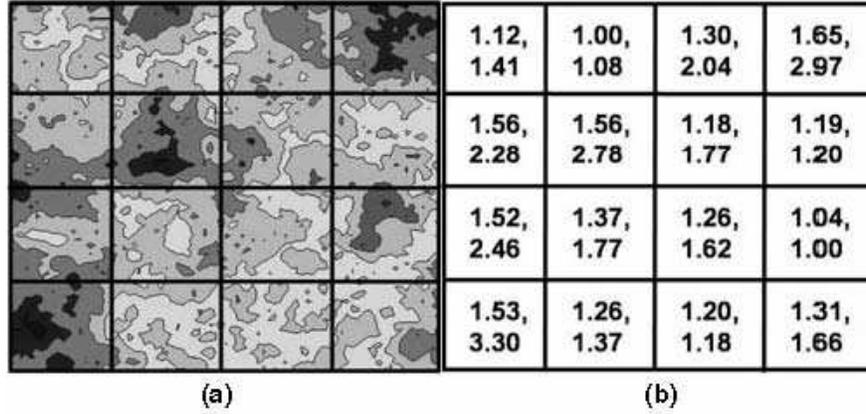


Figure 3.1: A WID  $V_{th}$  variation map for a 16-core processor in (a). The corresponding  $f_{max}$  and  $P_s$  map in (b) [43].

of  $[0, 1]$ . We use clock gating technique to control frequency rather than voltage scaling, because clock gating induces less activation overhead and has larger margin of adjustability. We assume that each thread will be assigned to one core and each core will run a single thread at a time.

### 3.3.3 Power and Thermal Model

The circuit equivalent thermal model in Hotspot [44] is adopted in our work. Assuming each of  $k$  cores has  $m$  function units, there are  $km$  blocks on the die and thermal interface material layer (TIM). Along with extra 14 blocks in the package, the total block number is  $M = km + 14$ . The multicore processor thermal model can be expressed as the following differential equation:

$$\frac{d\vec{T}(t)}{dt} = A\vec{T}(t) + B\vec{P}, \quad (3.1)$$

where  $\vec{P}$  and  $\vec{T}$  are  $M \times 1$  vectors denoting power and temperature of the blocks,  $A$  and  $B$  are  $M \times M$  matrices determined by thermal parameters of a given processor layout.

The power on each block consists of dynamic power, which has a linear dependency on operating frequency (recall that we don't use voltage scaling), and leakage power. We use a piecewise linear approximation model [63] to represent leakage of block  $i$ :  $P_{s,i} = P_{s0,i} + k_{i,s}T_i$ , where  $P_{s0,i}$  is the static power for block  $i$  under ambient temperature and  $k_{i,s}$  is the slope the static power changes with temperature at time step  $s$ . So equation (3.1) can be rewritten as:

$$\frac{d\vec{T}(t)}{dt} = \bar{A}\vec{T}(t) + B(\vec{P}_d(\vec{f}(t)) + \vec{P}_{s0}) \quad (3.2)$$

where  $\bar{A} = A + BK$ ,  $\vec{P}_d(\vec{f}(t))$  represents each core's dynamic power at its operating frequency. The package blocks does not generate power, so the package elements in vector  $P_d$ ,  $P_{s0}$  are all zero. More details and discussion about this can be found in the next section.

### 3.3.4 Problem Formulation

We first give a very restricted definition of the problem. After we solve the problem in the next section, we discuss how each of the constraints in the following problem formulation can be relaxed while our solution still remain valid.

*Given  $N$  identical and independent threads, and a  $k$ -core processor, each core has its leakage power  $P_{s,i}$ , maximal frequency  $f_{max,i}$  and can change its operate frequency between 0 and  $f_{max,i}$ ; determine the frequency for each core to minimize the completion*

time of the  $N$  threads while keeping the temperature of each core under a given threshold  $T_{max}$  all the time.

Let  $f_i(t)$  be the clock frequency of core  $i$  at time  $t$  normalized to its maximum frequency  $f_{max,i}$ ,  $t_f$  be the completion time of all the  $N$  threads. As describe in the thermal model, we partition each core into  $m$  blocks and let  $T_i(t)$  to be the temperature of block  $i$  at time  $t$ , then the problem becomes to decide  $f_i(t)$  such that  $t_f$  will be minimized under the following constraints:

$$\sum_1^k \int_0^{t_f} f_{max,i} f_i(t) dt = N \quad (3.3)$$

$$T_i(t) \leq T_{max} \forall i \in \{1, 2, \dots, km\}, 0 \leq t \leq t_f \quad (3.4)$$

$$\min t_f \quad (3.5)$$

This an optimal control problem that can be solved by dynamic programming [16], note that we will also need to include equation (2) to obtain the thermal information. However, due to the large state space ( $N$ ) and control space ( $k$ ), the computation is not affordable even for off-line scheduling. For the rest of the chapter, we focus on finding analytical solutions that can be computed efficiently.

## 3.4 Analytical Solutions

### 3.4.1 Steady State Throughput

When  $N \gg k$ , say when the completion time  $t_f$  is much longer than the time  $t_0$  needed for each core to reach its steady state temperature [62], the cores will execute

at their steady state temperatures for most of the time. Then the completion time of the program will be mainly determined by the steady state throughput of each core. The problem from Section 3.3.4 then becomes to find the speed vector  $\vec{F}_{ss}$  at the steady state that maximize the steady state throughput. More specifically,

$$\max_{\vec{F}_{ss}} \sum_1^k f_i f_{max,i} \quad (3.6)$$

$$A\vec{T}_{ss} + B(\vec{P}_d(\vec{f}_{ss}) + \vec{P}_{s0}) = \frac{d\vec{T}_{ss}}{dt} = 0 \quad (3.7)$$

$$T_{ss,i} \leq T_{max} \forall i \in \{1, 2, \dots, km\} \quad (3.8)$$

$$0 \leq f_i \leq 1 \quad (3.9)$$

where  $T_{ss,i}$  is the steady state temperature for core  $i$  and equation (8) indicates that temperature has reach steady state and will not change with time  $t$ . This is a linear system with  $k$  variables  $f_i$ ,  $km$  linear constraints in (9), and  $2k$  simple bound constraints in (10). It can be solved in reasonable time.

**Theorem 1.** (Static scheduler) Suppose that at time  $t_0$ , all the cores have reached their respective steady state and have completed  $n_i$  threads, then the minimum time to complete all the  $N$  threads is  $t_0 + \frac{N - \sum_1^k n_i}{\sum_1^k f_i f_{max,i}}$ , where  $f_i$  is the solution to the above linear system.

**[Proof]** We construct the scheduler that achieves this minimum completion time. On the arrival of the  $N$  threads, we keep on feeding each core thread until time  $t_0$ . By that time, the last core has reached its steady state temperature and core  $i$  has completed  $n_i$  threads. For the rest  $(N - \sum_1^k n_i)$  threads, we assign  $(N - \sum_1^k n_i) \frac{f_i f_{max,i}}{\sum_1^k f_i f_{max,i}}$  to core  $i$  and set its speed to be  $f_i f_{max,i}$ . Therefore, all the cores will complete their assigned threads

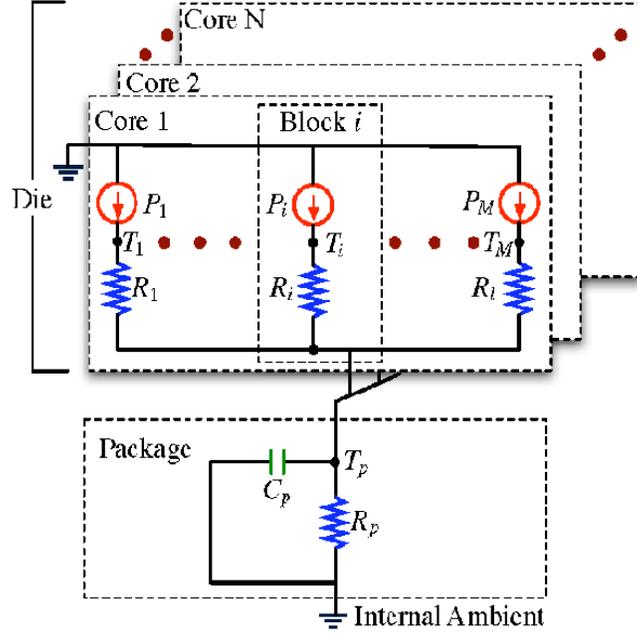


Figure 3.2: High-level thermal model of a multicore processor [63].

by time  $\frac{N - \sum_1^k n_i}{\sum_1^k f_i f_{max,i}}$ . From the fact that  $f_i$  is the solution to the above linear system, we know that there is no other scheduler that can finish more threads by this time. Therefore, this static scheduling strategy completes  $N \gg k$  threads with the minimum complete time  $t_0 + \frac{N - \sum_1^k n_i}{\sum_1^k f_i f_{max,i}}$ .

### 3.4.2 Local Optimal Frequency

When the completion time is not sufficiently larger than  $t_0$ , the steady state approach cannot be used and it becomes hard to find the optimal solution. We adopt the following thermal model (shown in Figure 3.2)[63] and use an approximation technique to find a local optimal solution.

From this equivalent thermal RC circuit, block  $i$ 's temperature satisfies the follow-

ing differential equations:

$$C_i \frac{T_i(t)}{dt} = -\frac{T_i(t) - T_p(t)}{R_i} + f_i P_{dmax,i} + [P_{s0,i} - k_i(T_{max} - T_i)] \quad (3.10)$$

$$\frac{T_p(t)}{dt} = -\frac{T_p(t)}{R_p C_p} + \frac{1}{C_p} \sum_{i=1}^M P_i(t) \quad (3.11)$$

Note that the time constant for the package and chip die block is of different order. For example, in the Alpha 21264, the largest time constant for chip block is 10 ms while the time constant of package is 60s [62]. Hence during  $t_s$ ,  $T_p$  can be considered as a constant. This allows us to solve the above first order linear differential equation and obtain

$$T_i(t) = T_i(0)e^{-\beta_i t} + (\alpha_i/\beta_i)(1 - e^{-\beta_i t}) \quad (3.12)$$

where  $\alpha_i = (1/C_i)(P_{s0,i} - k_i T_{max} + f_i P_{dmax,i} + T_p/R_i)$  and  $\beta = (1/C_i)(1/R_i - k_i)$ .

We assume that the chip is properly designed with  $k_i < 1/R_i$  to avoid thermal runaway. We now partition the execution interval  $[0, t_f]$  into small subintervals of length  $t_s$ .  $t_s$  is chosen to be of the order of  $10R_i C_i$  so the block temperature will be in its steady state [62, 25, 44]. The steady state temperature is given by

$$T_i = \alpha_i/\beta_i = \zeta_i T_p + (P'_{s,i} + f_i P'_{d,i})R_i \quad (3.13)$$

where  $\zeta_i = 1/(1 - k_i R_i)$ ,  $P'_{s,i} = \zeta_i(P_{s0,i} - k_i T_{max})$  and  $P'_{d,i} = \zeta_i P_{dmax,i}$ . We assume  $T_p$  can be obtained from the thermal sensor on the package, then the  $T_i$  is a linear function of  $f_i$ . The local optimal block frequency  $f_i$  can be calculated by

$$f_i = [(T_{max} - \zeta_i T_p) / R_i - P'_{s,i}] / P'_{d,i} \quad (3.14)$$

$$f_i = 1, \text{ if } (f_i > 1)$$

Note that the core  $j$  optimal frequency over the timestep  $t_s$  is subjected to the frequency of the hottest block belonging to the core and calculated by

$$f_j = \min(f_i) \quad (3.15)$$

So if we can identify those hottest blocks, the speed computation will have a considerable reduce. The equations can be reduced from  $mk$  to  $m$ . This job can be easily done by analyzing the coefficients in Equation 3.13.

**Theorem 2.** (Dynamic scheduler) With single sensor reading  $T_p$ , the Eq. (14) and (15) give the local optimal core frequency in timestep  $t_s$ .

**[Proof]** In each timestep  $t_s$  of dynamic scheduler, the core frequency is either maximum frequency  $f_{max}$  or the maximum frequency at which the core can leverage all thermal headroom to maintain its temperature at  $T_{max}$ . Any increase of the frequency results in a thermal violation.

Note that the Equation 3.14 and 3.15 involve only simple calculations, so they can be done very fast. For example, the local optimal core frequency selection of 16-core system for one time step and the frequency legalization in next section together use less than  $1ms$  on single core Pentium 4.

### 3.5 Scheduling Framework

Based on the solutions from previous section, we develop a scheduling framework combining the static scheduler and dynamic scheduler to select speed for each core and assign threads to them. The basic idea is to let the system user input the estimated completion time  $t_f^*$  of the program, if  $t_f^* \geq 10t_0$ , each core will execute at its optimal steady state speed, and the thread assignment is based on the ratio of their speeds, and else if  $t_f^* < 10t_0$ , feeding each core the right number of threads such that it can run at its local optimal speed over the period  $t_s$ . The static scheduling is very straightforward, so we mainly introduce our dynamic scheduler in the following.

The local optimal solution allows the individual core frequency change to arbitrary value in  $[0, f_{max}]$ , which is impractical in real multicore processor. So a frequency legalization approach in [72] is applied to legalize each core frequency  $f$  into  $r$  discrete levels. Basically, we choose highest lower bounding frequency in  $r$  to replace the  $f$  in  $t_s$  for each core. The frequency legalization also needs to be applied on optimal steady state speed.

Fig 3.3 depicts the idea of the proposed dynamic scheduler. At the start of each parallel processing of the program, the scheduler will fill up the thread queue for each core and each core will run at its  $f_{max,i}$ . At the end of each period  $t_s$ , the serial core, which is supposed to be idle during the parallel processing phase, will read the current package temperature from thermal sensor and the queue length of each parallel core's thread queue. Using the results from the previous section, the serial core will compute the local optimal solution and do frequency legalization, and then instruct each parallel core their respective speed during the next period of  $t_s$  and fill up the thread queue accordingly.

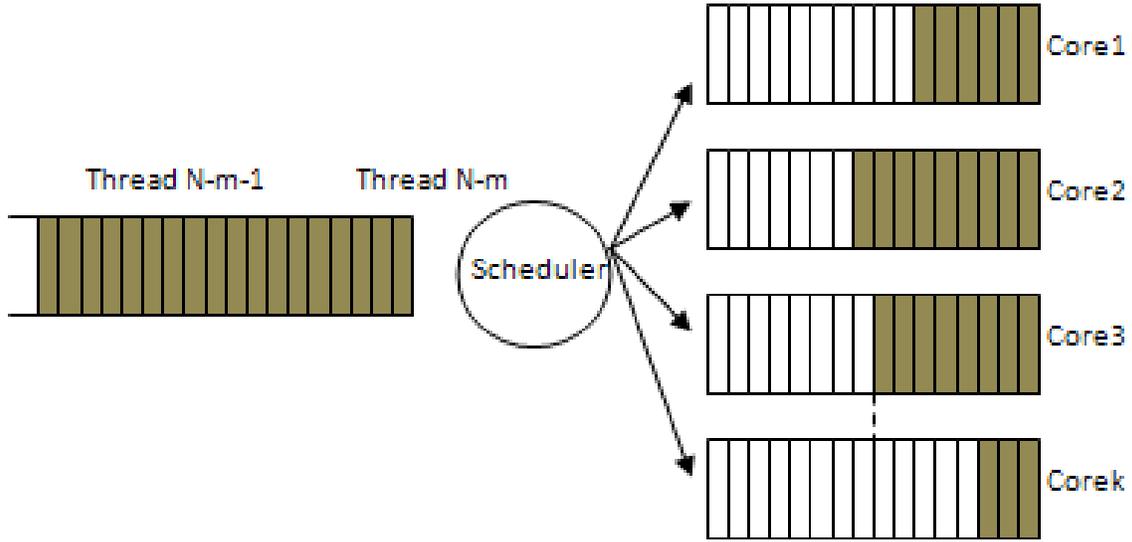


Figure 3.3: Dynamic scheduling framework

This procedure repeats until all the parallel cores reach their steady state temperature. At that time, the serial core will stop solving equations (14) and (15), it simply assigns threads to parallel cores without changing their speeds.

We use a timer to help the serial core to decide whether all the system has reached the steady state or not. The timer tracks the execution time of the current parallel processing and once it reaches a pre-set value, the serial core knows that the steady state is reached. The pre-set value can be chosen as the time for the slowest core to reach its steady state temperature. Stopping the serial core from finding local optimal solution will also save its energy consumption.

The run time overhead of this scheduler is hidden by two ways: first, we use the serial core to control the scheduler; second, we select the thread queue length such that

it can hold threads for the fastest parallel core to complete in  $2t_s$ . Therefore, while the serial core is computing, the parallel cores will not slow down or halt.

### **Discussion on the problem our scheduling framework targets:**

Each thread requires same amount of computation. This assumption can be relaxed. Indeed, in some of the experiment setting, we assume that thread's computation requirement has mean  $g$  and the variation across threads is uniformly distributed in  $(0, v)$  [15]. When we allow a thread to be executed on different cores, our results all hold. If one thread can only be run by one core, the completion time in Theorem 1 becomes a lower bound and may not be achievable. Independence of the threads. When the threads have data dependency, Theorem 1 provides a lower bound. The proposed dynamic scheduling framework can be easily modified so a thread will not be scheduled before the completion of threads it depends on. Computation bounded application. This enables us to ignore the memory problem, which is a major concern in multi-core system. Our solution can be extended to memory bounded applications by adding the notion of *ready* time that indicates the memory required for a thread to start is ready. However, the optimality of our result will be lost. Dynamic voltage scaling. We use clock gating to reduce frequency. Dynamic voltage scaling can be used in our framework, but it has time and energy overhead for circuit to become stable at the new voltage.

Table 3.2: Optimal steady state throughput of 16-core processor for problems of different switching activity

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	throughput	gain
WC	1.06	1.00	1.06	1.06	1.06	1.06	1.06	1.06	1.06	1.06	1.06	1.04	1.06	1.06	1.06	1.06	16.95	5.93%
NC	1.12	1.00	1.22	1.22	1.22	1.22	1.18	1.19	1.22	1.22	1.22	1.04	1.22	1.22	1.20	1.22	18.92	18.25%
BC	1.12	1.00	1.3	1.65	1.56	1.56	1.18	1.19	1.52	1.37	1.26	1.04	1.53	1.26	1.20	1.31	21.05	31.56%

Table 3.3: Dynamic scheduling average throughput for different problem size

Threads	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	throughput
1.0e6	0.9	1.00	0.77	0.51	0.54	0.54	0.85	0.84	0.66	0.73	0.8	0.97	0.66	0.8	0.84	0.77	15.59
1.0e5	1.09	1.00	1.22	1.38	1.33	1.33	1.07	1.11	1.30	1.24	1.18	1.01	1.34	1.16	1.09	1.27	19.12
5.0e3	1.12	1.00	1.3	1.65	1.56	1.56	1.18	1.19	1.52	1.37	1.26	1.04	1.53	1.26	1.20	1.31	21.05

## 3.6 Experiment Results

### 3.6.1 Experimental Setup

In this section, we describe how we simulate the power/thermal behavior of the multi-core processor executing parallel threads. First, our dynamic scheduler reads the package temperature  $T_p$  from the thermal sensor on the multi-core processor at the beginning of a period of length  $t_s$ . During simulation, we use Hotspot4 [44] with LDT pairwise liberalization model (PWL) [63] to estimate the transient package temperature  $T_p$ . The  $T_p$  information is passed to the scheduler and the serial core will use it to compute the local optimal speed for each parallel core for that period; then Hotspot uses this speed information to estimate the new  $T_p$  at the beginning of the next period.

We adopt a 16-core version of the Alpha 21264 floor plan in Hotspot and set the thermal threshold  $T_{max}$  to  $110^{\circ}C$ . The convection thermal resistance value  $R$  is configured to be  $0.4^{\circ}C/W$  and the maximum dynamic power of the whole chip  $chip_{dmax}$  is set to be  $120W$ . The maximum leakage power  $chip_{static}$  at  $T_{max}$  is set to be  $75W$ . The process variation  $f_{max}$  and  $P_s$  parameters from [43] are used to formulate different thermal/power equivalent RC circuit for each core in Hotspot. The single thread mean workload  $g$  and the variation  $v$  are defined by the execution time of the slowest core ( $g = 10ms$  and  $v = 5ms$ ) in the simulation. The length of the thread queue is set to be 32. We consider the scenario that different thread in same program has different switch activity in the simulation. The thread  $j$  power dissipation on core  $i$  is given by  $\rho_j P_{d,i}$  where  $\rho_j$  represents thread  $j$ 's switch activity. We set  $\rho_j$  as a random variable with three different uniform distributions  $U_w[1.0, 0.8]$ ,  $U_n[0.7, 0.5]$ , and  $U_b[0.4, 0.1]$  in the simulation to represent the worst case, the normal case, and the best case that a program can have.

### 3.6.2 The Throughput of Static and Dynamic Scheduling

The traditional synchronized multi-core system will use the frequency of the slowest core, core 2, as its speed and the 16-core system's throughput will be 16 normalized to the throughput of the core 2.

For different switch activity distributions, we can calculate the optimal steady state speed/frequency for each core from Theorem 1 using the upper bound of  $\rho_j$ , as presented in the Table 3.2. The sum of all core's steady state speed will be the throughput produced by the static scheduler, where we can see performance gain of 5.93%, 18.25%, and

31.56% in worst case, normal case and best case respectively. The throughput increases from the decrease of switch activity is due to the fact that dynamic power is proportional to switch activity where a low switch activity will allow the cores to run at a high speed, which is shown clearly in each column in Table 3.2. A quick comparison with the core information in Fig. 3.1 indicates that in best case, each core is able to run at its  $f_{max,i}$  without violating the  $T_{max}$  constraint.

For dynamic scheduler based on Theorem 2, we are more interested to know its performance on different problem sizes (thread number) under normal switch activity. So we use  $U_n[0.7, 0.5]$  distribution for thread switch activity and simulated three different problem size ( $N=5.0e3$ ,  $N=1.0e4$ ,  $N=1.0e5$ ) to represent small problem, middle-sized problem, and large problem. Table 3.3 reports the average throughput of each single core and the sum of all cores. All throughput is normalized to that of slowest core. The average throughput is 21.05, 19.12, and 15.59 in small problem, middle-sized problem and large problem respectively. We can see that as the problem size increase, the throughput produced by dynamic scheduler is decreasing. Spectacularly, for small problem, dynamic scheduler is capable of proving the upper bound throughput; for middle-sized problem, the dynamic scheduler beats the static scheduler with 1% more throughput; for large size problem, the dynamic scheduler's throughput is even lower than the traditional k-core system. From Eqn. (14), we know that if the dynamic scheduler runs long enough ( $t_f \geq t_0$ ), the package temperature  $T_p$  will eventually enter the steady state and no longer change as well as the local optimal solution, which is the steady state throughput of dynamic scheduler. It is less than the optimal steady state throughput from theorem 1. So when  $t_f \geq 10t_0$  the average throughput of the program mainly depends on the steady state

throughput, the static scheduler should be applied, as in our scheduling framework.

### 3.6.3 Completion Time Reduction

Table ?? reports the completion time reduction of our proposed scheduling framework over the traditional multi-core system that uses the slowest core's speed, under normal switch activity and on middle-sized problem.  $t_{serial}$ : serial execution time (s);  $N$ : number of threads;  $t_{k,parallel}$ : parallel execution time with traditional k-core (s);  $t'_{k,parallel}$ : parallel execution time with our scheduler (s);  $delta_{parallel}$ : parallel execution time reduction (%);  $delta_{total}$ : total execution time reduction (%). We consider programs where the serial part (the  $F$  values) counts for 10% to 50% of the total execution time. A low percentage means high parallel computation capability. The number of cores varies from 16 to 8 and to 4. We can see the clear trends of (1) larger completion time reduction with lower  $F$  value; (2) larger completion time reduction with more number of cores. For the 16-core system we take from Fig. 1, even with  $F = 0.5$ , which means that only half of the program can be paralleled, our scheduler can reduce the total processing time by 4.3%.

## 3.7 Summary

At the chip-level, process variations can cause significant differences in frequency and leakage characteristics across cores in a multicore processor. This trend is getting worse as chip technologies scale to smaller dimensions. To realize the full potential of multicore processors, such variations should be taken into account carefully throughout the design process. At the same time, dynamic thermal management (DTM) plays an

important role in the operation of parallel programs on multicore processors due to the large range of possible die temperatures. Considering process variations in DTM to maximize multicore processor throughput under thermal constraints is therefore an important problem.

In this chapter, we have formulated this problem as a scheduling problem, and demonstrated that this scheduling problem can be solved in reasonable time using dynamic programming techniques. We have also developed static and dynamic scheduling heuristics and proposed a policy to choose between their static and dynamic configurations to maximize the throughput. We have demonstrated that these static and dynamic heuristics are suitable for online implementation. Our hybrid static/dynamic scheduling method is the first to address multicore processor scheduling through an online solution that takes into account both process and temperature variations.

Table 3.4: Performance (completion time) gain with different  $F$  and number of cores.

F	10%	20%	30%	40%	50%
$t_{serial}$	33	66	99	132	165
$N$	90000	80000	70000	60000	50000
$t_{16,parallel}$	56.1	50.0	43.7	37.5	31.2
$t'_{16,parallel}$	43.5	37.8	32.2	27.2	22.6
$delta_{parallel}$	22.4%	24.4%	26.3%	27.5%	28.4%
$delta_{total}$	14.1%	10.5%	8.05%	6.08%	4.3%
$t_{8,parallel}$	112.2	100.0	87.4	75.0	62.4
$t'_{8,parallel}$	96.2	85.8	74.9	64.3	53.5
$delta_{parallel}$	14.2%	14.2%	14.2%	14.2%	14.2%
$delta_{total}$	10.9%	8.5%	6.6%	5.1%	3.9%
$t_{4,parallel}$	224.4	200	174.8	150	124.8
$t'_{4,parallel}$	205.3	183	159.9	137.2	114.2
$delta_{parallel}$	8.5%	8.5%	8.5%	8.5%	8.5%
$delta_{total}$	7.4%	6.3%	5.4%	4.5%	3.6%

## Chapter 4

### Exposing Intra- and Inter-Actor Parallelism for Implementation of DSP

#### Applications

In this chapter, we tackle scheduling issues at multiple levels in a behavioral context: application-level, component-level, and operation-level. In particular, we develop a dataflow based design method for DSP applications, in which the applications are described in layers to expose intra- and inter-actor parallelism. This exposed parallelism is then exploited systematically in our design method for performance optimization on multiprocessor system-on-chip (MPSoC) devices.

MPSoC technology is an important trend in the design and implementation of signal processing systems. However, the design of efficient DSP software for MPSoC platforms involves complex inter-related steps, including data decomposition, memory management, and inter-task and inter-thread synchronization. These design steps are challenging, especially under strict constraints on performance and power consumption, and tight time-to-market pressures. To facilitate these steps, we have developed a new dataflow based design flow within the targeted dataflow interchange format (TDIF) design tool [71].

Our new MPSoC-oriented design flow, called *TDIF-PPG*, is geared towards analysis and mapping of embedded DSP applications on MPSoCs. An important feature of TDIF-PPG is its capability to integrate *graph level parallelism* for DSP system flowgraphs and *actor level parallelism* for DSP functional modules into the application mapping pro-

cessing. Here, graph level parallelism is exposed by the dataflow graph application representation in TDIF, and actor level parallelism is modeled by a novel model for multiprocessor dataflow graph implementation that we call the *parallel processing group (PPG)* model. We demonstrate our approach through actor and subsystem design for software defined radio.

Material in this chapter was published in preliminary form in [92].

## 4.1 Introduction

As multicore processor technology evolves, increasing numbers of processors are integrated into system-on-chip (SoC) devices for signal processing system implementation. The trend towards multiprocessor SoCs (MPSoCs) is motivated by the performance gain from efficient parallel execution of programs. This performance gain is determined in part by the amount of parallelism exposed from the program.

Digital signal processing (DSP) applications are often specified in terms of dataflow graphs, which provide high level, model-based views of systems being designed (e.g., see [4]). The parallelism exposed from such high level dataflow representations includes the following three forms.

1. Data parallelism: an actor (dataflow graph functional component) performs the same computation on different units of data.
2. Control/task parallelism: multiple actors execute different tasks on the same or different data.

3. Temporal parallelism (pipeline parallelism): multiple instances of the same actor execute simultaneously, where the instances correspond to different iterations of the enclosing dataflow graph.

These forms of dataflow modeling parallelism can all be viewed as *graph level parallelism*. A significant body of work has been developed to help expose and exploit graph level parallelism from dataflow models [22, 75, 74]. Relatively less attention has been given to exploiting parallelism *within* actors (*actor level parallelism*) within an enclosing dataflow framework.

Actor level parallelism provides optimization opportunities for enhancing performance beyond graph level parallelism. What is more challenging is the effective integration of graph level parallelism and actor level parallelism within an overall system-level optimization framework. However, without suitable models and tools to facilitate this exploration, procedural language compiler techniques, such as data decomposition, memory management, and inter-task and inter-thread synchronization, need to be developed from scratch to effectively exploit both actor level parallelism and graph level parallelism. On the other hand, general purpose parallel programming models, such as OpenMP and MPI, are designed for use across arbitrary application domains. Such generality can enhance convenience, but does not allow designers to thoroughly exploit specialized properties of their targeted application areas, such as the coarse grain dataflow structure (i.e., the signal processing flowgraph structure) of DSP applications [4].

In the DSP domain, dataflow models are widely used to specify, analyze, and simulate DSP applications. A variety of dataflow techniques have been developed for DSP ap-

plications to target problems such as buffer size optimization, scheduling, and cross platform porting. In this chapter, we present a dataflow-based design flow, called *TDIF-PPG*, for design and implementation of parallel software targeted to MPSoC devices. TDIF-PPG extends the capabilities of the *targeted dataflow interchange format (TDIF)* [71] design environment with methods for expressing intra-actor parallelism, and associated capabilities for platform independent design, and early-stage performance evaluation. TDIF-PPG applies and systematically integrates both graph level parallelism and actor level parallelism. As a key component of TDIF-PPG, we propose a novel actor design technique called the *parallel processing group (PPG)*.

TDIF-PPG provides a flexible design environment without compromising the types of parallelism that can be exploited. TDIF-PPG achieves this by providing a breadth of formal models spanning graph and actor level parallelism. This approach to actor and system design allows the designer to exploit trade-offs among multiple factors, such as the number of utilized cores, buffer usage, throughput, and latency. The features of TDIF-PPG also provide schedulers more opportunities to achieve better system performance.

The remainder of this chapter is organized as follows. In Section 4.2, we review background and related work. We then introduce our new PPG plug-in to TDIF in Section 6.3. The PPG model, along with its application programming interfaces and execution flow, are discussed in Section 4.4. In Section 5.8, we discuss experiments in applying TDIF-PPG on a Texas Instruments (TI) multicore DSP platform. Here, a software define radio (SDR) application (MPSched) is used as a case study to demonstrate the features of the TDIF-PPG design flow. Conclusions and directions for future work are then discussed in Section 4.6.

## 4.2 Related Work and Background

### 4.2.1 Core Functional Dataflow

*Core functional dataflow (CFDF)* is a deterministic sub-class of enable-invoke dataflow [57], which is a dynamic dataflow model that can express both static and data-dependent dataflow behaviors. In CFDF, actors are specified as sets of modes, where each mode has a fixed production and consumption rate associated with each actor output and input port, respectively. On each CFDF firing (actor invocation), an actor operates based on a unique *current mode*, which is maintained as part of the actor state. During each firing, in addition to consuming input tokens and producing output tokens, the actor selects one mode from its set of modes as the *next mode*, which will be applied as the current mode in the next firing of the actor.

In CFDF, the separation of enable (firability checking) and invoke (firing) functionality is defined as a first class characteristic of the model. Each actor has an associated *enable function*. This function, which can be called at any time between firings (e.g., by a dynamic scheduler), returns a Boolean value indicating whether or not there is sufficient data available on the actor input ports to fire the actor in its current mode. Since such an isolated enable check is available, the invoke function of an actor assumes that sufficient data is present, and reads its input data without blocking reads. When an actor is invoked, it executes its current mode, produces and consumes data, and updates its current mode. Since different modes of an actor can have different production and consumption rates, dynamic dataflow can be modeled flexibly in CFDF.

## 4.2.2 Targeted Dataflow Interchange Format

TDIF [70] is an extension of the *dataflow interchange format (DIF)* [26] tool for design and analysis of DSP-oriented dataflow graphs. TDIF adds to DIF features that include dynamic dataflow software synthesis, cross-platform actor design support, and dataflow-integrated instrumentation and tuning of implementations [70]. TDIF leverages the power of dynamic dataflow models and provides automated code generation of actor programming interfaces and low level customizations for implementations targeted to heterogeneous platforms.

## 4.2.3 Related Work

Mapping an application to an MPSoC platform based on conventional methods is an error prone and time consuming process involving multiple steps. These steps include (1) decomposing a program into computational units and dividing these units into balanced threads; (2) managing the resulting inter-task and inter-thread communication and synchronization; and (3) handling resource management (memory, processors, interconnection bandwidth, etc.). A variety of layered models have been proposed to help hide hardware complexity from programmers, and allow advanced automation techniques to take over parts of the burden in the design process.

Jerraya et al. [29] discuss a design methodology based on a hardware/software layered interface and suggest a three-layered interface. This interface includes the parallel programming model, hardware dependent software, and hardware abstraction layer. Ceng [11] proposes a novel compiler technique using the tightly-coupled thread (TCT)

model. This approach uses sequential C code as input, and automatically generates a parallel executable with minimal user guidance for a specialized architecture that supports the TCT model. Mignolet et al. [47] develop an MPSoC mapping flow that takes sequential C code as input, and generates parallel code in terms of concurrent C threads. The user is required to designate the segments of code that are to be selected for parallelization, the number of threads that execute these segments, and a high level description of the memory hierarchy on the target platform. The resulting parallel C code is then compiled for the target platform. Kwon [38] introduces the Common Intermediate Code (CIC) layer to interface software and hardware. Software is first partitioned and then written in CIC using generic application programming interfaces (APIs) for inter-task communication and synchronization. Later, the CIC translator translates the CIC code to platform-dependent code with the required hardware information. After that, scheduling code for the different processors is generated.

In contrast to prior work, the primary contribution of this chapter is a novel framework for systematically integrating dataflow graph level parallelism and actor level parallelism into the design and implementation process for multiprocessor DSP systems. Key details on the novelty and utility of our contribution are as follows. (1) We provide a clear separation between graph- and actor-level parallelism, which enables the utilization of different forms of DSP parallelism at different levels of abstraction (e.g., parallelism across distinct filters versus parallelism across different taps of the same filter). At the graph level, dependencies between actors is loose and decoupled, while at the actor level, the synchronization and communication is more tightly coupled to exploit features for exploiting fine-grained parallelism in DSP-oriented processors. (2) Our proposed PPG model allows

DSP system designers to flexibly explore different combinations of data parallelism, task parallelism and temporal parallelism within individual actors. (3) When both graph-level parallelism and actor-level parallelism are exposed, our TDIF-PPG framework provides comprehensive APIs to implement schedules that efficiently manage the resulting inter-task and inter-thread communication and synchronization on the target platform. (4) Our TDIF-PPG framework provides a unified abstraction layer of the underlying hardware platform. This abstraction layer helps to hide hardware complexity from programmers, and automatically generate code to handle resource management. Our efficient support for such an abstraction layer is especially useful given the diversity of processor families that are relevant for DSP system design.

### 4.3 TDIF-PPG Design Flow

The PPG plug-in adds two extra layers to the previous two layer TDIF design flow, as shown in the four-layer design flow demonstrated in Fig. 4.1.

In layer 1 — the *system layer* — the given DSP application is modeled as a CFDF graph using the DIF language. The DIF parser takes the CFDF graph as input, and constructs a corresponding model in the DIF intermediate representation, which helps to expose graph level parallelism.

In layer 2 — the *actor interface layer* — actor interface specifications, including information about input and output ports, actor parameters, and CFDF modes, for individual actors are provided using the TDIF language. Then the TDIF compiler parses the TDIF specifications for the actors, and generates equivalent actor API code in the tar-

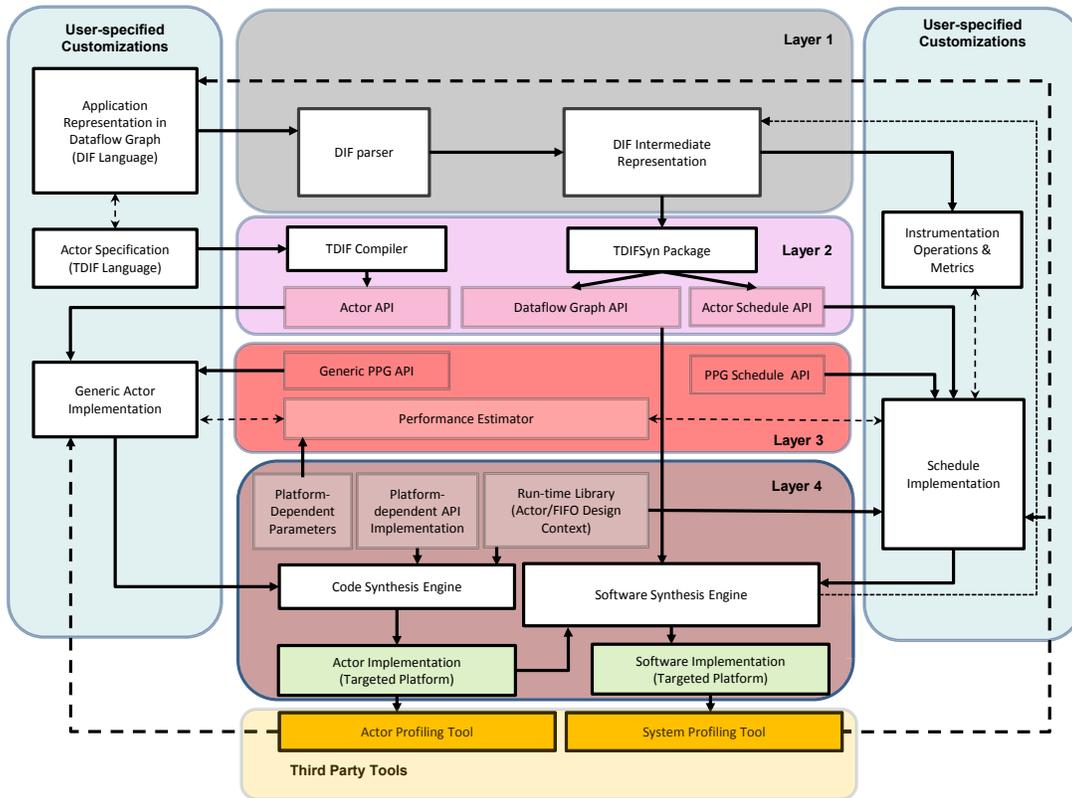


Figure 4.1: TDIF-PPG design flow.

geted actor implementation language (C with optional extensions in the current version of TDIF-PPG). The generated APIs provide prototypes for actor interface functions, including functions for accessing the ports, modes, and parameters of each actor, as well as invoking and testing firability of the actors.

In layer 3 — *the platform-independent mode specification (PIMS)* layer — the PPG model guides the programmer in exposing actor level parallelism within the functional specification for each mode of an actor. The programmer creates parallel threads for actor modes, and describes the corresponding inter-thread communication and synchronization using generic PPG APIs. This provides a generic implementation of the associated actor

that is not tied to any specific parallel platform, and thus, facilitates portability across platforms.

In layer 4 — *the actor implementation layer* — generic actor specifications from the PIMS layer are integrated with optimized platform-specific PPG API and run-time implementations. This integration is performed automatically by the *TDIF-PPG Code Synthesis Engine*. Third-party profiling tools are integrated in our design flow with layer 4 experimentation to provide measurements on final actor implementations, and associated feedback to help refine higher levels of the overall design flow. Note that multiple feedback loops (the dashed lines in Fig. 4.1) are provided in the design flow to allow for feedback at different levels with trade-offs between exploration speed and accuracy. The feedback loop in layer 3 is much faster but less accurate than the feedback loop in layer 4.

To produce a complete system implementation, the TDIFSyn package takes the DIF intermediate representation as input and generates the top-level C language implementation file and associated APIs for actor scheduling [71]. The automatically generated top-level C file initializes the operational contexts of actors and FIFOs, configures actor parameters, lays out the graph topology by instantiating connections between actor ports and their incident FIFOs and calls a user-defined scheduler that utilizes the APIs for actor scheduling and PPG scheduling. It is the responsibility of this user-defined scheduler, which can be provided by a programmer or constructed by a tool, to utilize the PPG scheduling APIs correctly, and ensure that the PPGs inside an actor are scheduled correctly. In our present implementation of the TDIF-PPG design flow, we develop the user-defined scheduler modules by hand (i.e., they are provided by a programmer). Inte-

grating tools into TDIF-PPG for automated schedule construction is a useful direction for future work.

The given user-defined schedule along with the TDIF-PPG run-time library and the actor implementations are integrated automatically through glue code that is synthesized by the *TDIF Software Synthesis Engine*. System profiling tools can then be applied on the generated implementations to validate whether or not system constraints are satisfied, and provide feedback for tuning of the schedule or higher levels of the design hierarchy, all the way up to the application model provided in layer 1.

In summary, the TDIF-PPG design flow enhances the retargetability of designs across different platforms by allowing designers to provide platform independent actor specifications, and automatically generating optimized implementations for different parallel platforms. Such retargetability is useful in efficiently exploring design options, and porting designs across platforms — e.g., to upgrade to newer hardware generations or provide alternative design versions that are targeted to different types of hardware, such as alternative versions for low cost, and high performance. Also, the provisions in the TDIF-PPG design flow for quick feedback across different design layers helps to reduce total software development time. Furthermore, the TDIF-PPG design flow uniquely takes both graph level parallelism and actor level parallelism into account for system optimization.

## 4.4 Parallel Processing Group

### 4.4.1 Model Description

In an abstract sense, a PPG is a set of threads associated with computations within a dataflow graph actor. A PPG either contains a single thread (*single thread PPG*) or contains multiple threads that can be executed in parallel. Each thread in a PPG contains a set of data objects and a single code segment (a task or function that implements the thread). In a PPG, inter-thread synchronization and communication are restricted to three basic methods: *broadcast*, *barrier* and *point to point (P2P)*, which are discussed in Section 4.4.2. We plan to extend support to other synchronization/communication methods in our future work.

An actor can contain any non-negative number of PPGs. The PPGs inside an actor are connected (in a logical sense) with FIFOs. SIMD and MIMD execution styles are both supported by our concept of PPGs. In addition to the *thread information* (the data objects and code segments), a PPG includes a *PPG execution context* for managing execution of the contained threads. APIs for PPG-based actor design are introduced in Section 4.4.2, and thread execution in PPGs is demonstrated in Section 4.4.3. In the remainder of this section, we elaborate on the components that make up a PPG.

#### *Thread information*

- **Readonly Shared Data Object:** A PPG *Data Object* is an abstract data type that specifies the start address of a data block and the byte length of the data block.

Each thread in a PPG can have a separate data object, or subsets of multiple threads can share common data objects. When implementing PPG threads that share data objects, it may be desirable (for enhanced performance) for each thread to maintain a local copy of the object. Such optimization is supported in our proposed PPG-based design flow.

- **Input Data Object:** Each thread in a PPG can have one or more input data objects. Threads do not share input data objects. Input data objects are used to access any data arriving from input ports of the enclosing dataflow actor, as well as any data arriving from other PPGs associated with the same actor.
- **Output Data Object:** (analogous to an input data object) Each thread in a PPG can have one or more output data objects. Threads do not share output data objects. Output data objects are used to access any data that is sent to output ports of the enclosing dataflow actor, as well as any data that is sent to other PPGs associated with the same actor.
- **Thread Function/Task:** Each thread in a PPG has an associated reference to a computational task (e.g., a function pointer in C-based actor implementation), which provides the program code associated with the thread.

#### *PPG Execution Context*

- **Group Input Manager:** Each PPG has a group input manager. If group  $B$  reads data from the output FIFO of group  $A$ , group  $A$  is called a *predecessor group* of

group  $B$ , while group  $B$  is called a *successor group* of group  $A$ . A group input manager handles reading of data from any actor input FIFOs, as well as any output FIFOs from predecessor groups that are referenced during group execution. The group input manager performs data transfers to ensure that such “group input data” is transferred into local buffers associated with the group before such data is operated on.

- **Group Output Manager:** Similarly, each PPG has a group output manager, which handles the writing of processed results from the group’s local buffer to actor output FIFOs, and input FIFOs of successor groups.
- **Group Member:** Each thread in a PPG has an associated group member, which is the identifier (ID) for the processor that is to execute the thread. The group member for a thread can in general be set and changed dynamically.
- **Group Owner:** Each PPG has a group owner (also a processor ID), which is invoked when the PPG is to be executed. Upon invocation, a group owner broadcasts the PPG to all of its associated group members, and then waits for the completion of the PPG. The group owner for a PPG can in general be set and changed dynamically.

#### 4.4.2 Application Programming Interfaces

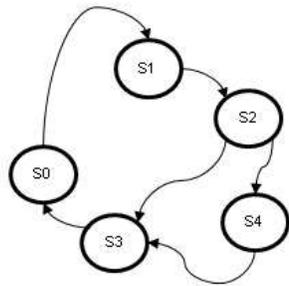
In our proposed design methodology, the PPG is the basic unit of functionality in an actor. Our development of the the PPG model includes interface APIs for various classes of key operations that are important for working with PPGs:

- Standardized interaction between PPGs in the actor and FIFOs of the actor;
- Standardized interaction between PPG threads and threads associated with data movement;
- Construction and configuration of new PPGs;
- Scheduling PPG threads onto processors.

These APIs are “abstract” in the sense that they are developed independently of any specific hardware platform, and can be retargeted across a variety of relevant platforms. Our design of these APIs helps to free the actor designer from tedious platform-specific details, and provides useful utilities for quick adoption of PPGs into his or her actor designs. The APIs also provide a consistent interface with which the TDIF-PPG design flow can be ported, fine tuned, and maintained on different platforms.

#### 4.4.3 PPG Static Execution Flow

We describe a *static execution flow* for PPGs in this section. In such an execution flow, the thread information and execution context for a PPG are specified when the PPG is created. As demonstrated in the group owner finite state machine (FSM), illustrated in Fig. 4.2, the group owner reads the PPG first and then reads the set of group members from the PPG. After that, it calls the `broadcast` API to notify the group members about the PPG. If the group owner is also a group member (for the same group), the group owner calls the `execute` API and the `barrier` API, and then waits for all of the PPG threads to complete; otherwise, it simply remains idle and waits for the other threads to complete.



State	Description	Transition state: condition
S0	Read the PPG	S1: PPG is read
S1	Read Group Member from PPG	S2: Group Member is read
S2	<b>Broadcast</b> the PPG to the group member	S3: The group owner is not the group member S4: The group owner is the group member
S3	Waite for the signal of completion of all threads	S0: The signal is received
S4	Invokes <b>Execute</b> and then <b>Barrier</b>	S3: Finishes <b>Barrier</b>

Figure 4.2: Group owner FSM and state description table for static execution flow.

Concurrently, when a group member receives the associated PPG handler, as depicted in Fig. 4.3, it calls the `execute` function and then uses the `barrier` function to synchronize with the other group members and with the group owner. If inter-thread communication is necessary, it is specified by the `P2P` function. The `barrier` function can be implemented by incrementing or decrementing the value of a shared variable called a *synchronizer variable*, and checking the value of this synchronizer variable upon completion of relevant threads. If the value of this variable is equal to an appropriate predefined value, then the group member interrupts the group owner to report that all of the group members have completed their tasks under the current PPG invocation. When the group owner receives the interrupt, it reorganizes the results (data reformation, data relocation, and data merging) for any successor groups as needed.



Figure 4.3: Group member FSM and state description table for static execution flow.

#### 4.4.4 Examples

In this section, we demonstrate the utility of our PPG model in expressing actor level parallelism with two important examples of signal processing actors. As with graph level parallelism, data parallelism (DP), control parallelism (CP), and temporal parallelism (TP) can all be relevant to actor level parallelism. The difference in our proposed design methodology is that at the actor level, all parallelism is described in terms of relationships among threads. Since CP and DP are similar, we focus only on DP and TP in the examples of this section. In particular, DP is utilized in designing a Finite Impulse Response (FIR) filter, and a combination of DP and TP is expressed in the design of a fast Fourier transform (FFT) actor.

##### 4.4.4.1 FIR Filter

The operation of the FIR filter is described by the following equation, which defines the output sequence  $y[n]$  in terms of its input sequence  $x[n]$ :

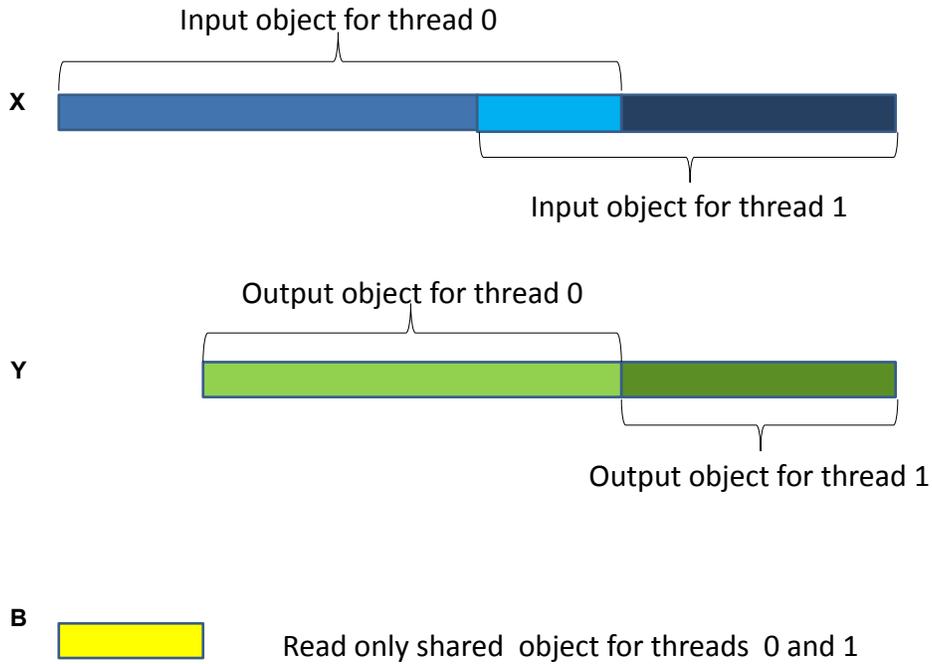


Figure 4.4: Mapping data parallelism from an FIR filter to PPG-based actor design.

$$\begin{aligned}
 y[n] &= b_0x[n] + b_1x[n - 1] + \dots + b_Mx[n - M] \\
 &= \sum_{i=0}^M b_ix[n - i]
 \end{aligned} \tag{4.1}$$

In our example of PPG-based FIR actor design, the input signal is first buffered so that blocks of samples (dataflow tokens) can be processed together. We assume that  $N$  input samples are buffered and then sent to an  $M$ th order FIR filter to produce  $(N - M)$  output samples. This example provides a significant amount of DP. For demonstration purposes, we map the DP onto two processor cores shown in Fig. 6.5. However, the retargetable TDIF-PPG design flow can easily adapt the implementation of this actor to utilize more cores if available.

In order to exploit DP using a PPG, the  $N$  input samples are divided into two

*input data objects* for two threads. One of these objects is derived from the samples  $[x[0], x[1], \dots, x[N/2+M-1]]$ , and the other object is derived from the samples  $[x[N/2], x[N/2+1], \dots, x[N-1]]$ . Similarly, the output samples are divided into two *output data objects*. The coefficient vector  $B$  is shared by two threads and placed into a *readonly shared data object*. Each thread executes the FIR calculation independently. This calculation is encapsulated in a function called `fir`, and the associated function address is set as the PPG function for each thread. This provides a PPG-based implementation of the FIR filter with two threads. Performance results from this implementation are examined in Section 5.8.

#### 4.4.4.2 FFT

The radix-2 decimation-in-time (DIT) FFT is the simplest and most common form of the Cooley-Tukey algorithm [19]. The radix-2 DIT FFT computes the DFTs of the even-indexed inputs  $x_{2m}$  ( $x_0, x_2, \dots, x_{N-2}$ ), and of the odd-indexed inputs  $x_{2m+1}$  ( $x_1, x_3, \dots, x_{N-1}$ ), and then combines the two results to produce the DFT of the whole sequence. Then the same procedure is performed recursively to reduce the overall runtime to  $O(N \log N)$ .

The recursive tree of the Radix-2 DIT FFT is illustrated in Fig. 6.6(a). The white nodes in the figure calculate DFT results, where the corresponding numbers of points are annotated next to these white nodes, and the black nodes merge pairs of smaller DFT results together.

Both DP and TP from the recursive tree are utilized in the derived PPGs shown in Fig. 4.5(b). With DP, each thread within  $G0$  calculates half of the overall  $DFT(N)$

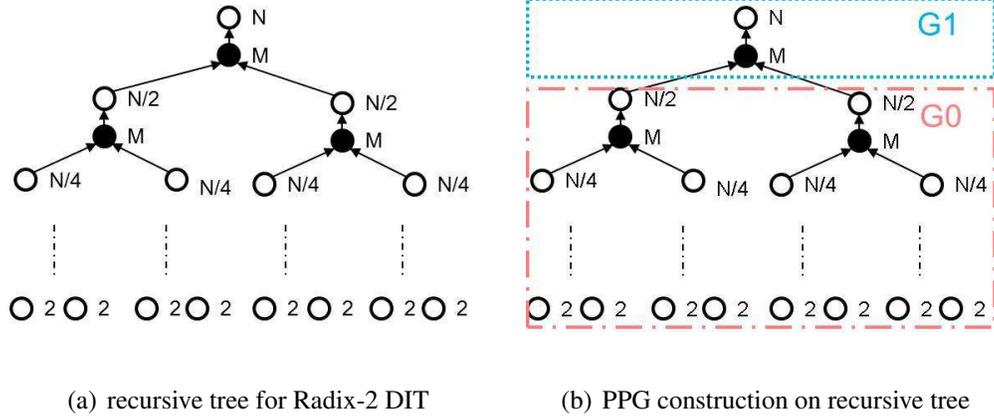


Figure 4.5: PPGs in an FFT actor.

( $N$  point DFT), which is a  $DFT(N/2)$ . This calculation is performed using the Cooley-Turkey algorithm. Then a single thread in  $G1$  merges the two results together to form a  $DFT(N)$ .

Two different implementations are provided here. In the first,  $G0$  is assigned to two cores, and  $G1$  executes on one of these two cores after  $G0$  completes. In the second implementation, which exploits both DP and TP,  $G0$  is assigned to two cores, and  $G1$  is assigned to another (third) core. The mapping is coordinated in such a way that  $G0$  and  $G1$  execute in a software-pipelined fashion to exploit TP. Experimental results from these two implementations are discussed in Section 5.8.

Through the FIR filter and FFT actor examples presented in this section, we have shown concretely that the PPG can support a variety of different forms and combinations of parallelism in actor design. Such actor level parallelism can be expressed naturally using PPGs in a manner that is not tied to any specific hardware platform. The PPG abstraction thus lets the programmer express actor level parallelism as part of the actor design, while deferring to lower (more specialized) levels of the design flow the optimized

mapping of such parallelism to specific hardware platforms.

#### 4.4.5 Discussions

The PPG is designed to facilitate efficient implementation of actors on a variety of MPSoC platforms. Since the threads within PPGs are tightly coupled, the PPG abstraction presented in this chapter is geared mainly towards shared memory architectures. Extension of our PPG design methodology to also accommodate distributed memory architectures is a useful direction for future work.

In our initial implementation of the TDIF-PPG design flow, P2P is employed for inter-thread communication in PPGs. This form of communication only supports Combined Blocking (CB) interfaces [78]. Extension of inter-thread communication in PPGs to allow other forms of interfacing, including Relative Blocking (RB), Relative Non-blocking (RN), Direct Blocking In-order (DBI), Direct Non-blocking In-order(DNI), Direct Blocking Out-of-order(DBO), and Direct Non-blocking Out-of-order(DNO) [78], is also a useful direction for future work.

#### 4.5 Experiments

In this section, we present experiments using TDIF-PPG. Our experiments involve the two actor design examples presented in Section 4.4.4 (FIR filtering and FFT computation), and a synthetic benchmark, called *mp-sched* (which stands for “multiprocessor scheduling”). The *mp-sched* benchmark is composed from FIR filtering and FFT computations, and is representative of a class of subsystems of software defined radio (SDR)

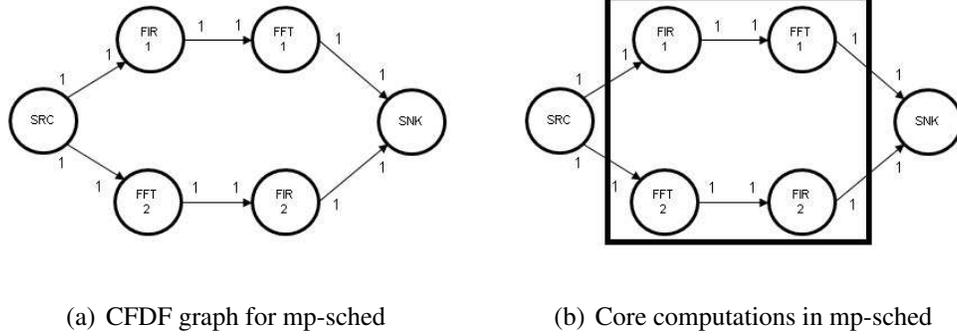


Figure 4.6: The mp-sched benchmark for SDR.

applications [59]. In our experiments, the mp-sched benchmark is modeled using CFDF semantics. The core functionality for each actor in the CFDF representation is encapsulated in a specific CFDF mode, called the *process* mode. In the process modes of the actors, generic PPGs are used to express actor level parallelism.

In our experiments, we employ a Texas Instruments (TI) 6678L multicore programmable digital signal processor (PDSP) as the target platform. The TI 6678L has 8 PDSP cores, where each core runs at 1.25 Ghz, and has 32KB L1 data cache, 32 KB L1 instruction cache, and 512KB L2 cache. The TI 6678L also has 4MB of shared S-DRAM and 512MB of DDR3 DRAM. Using TI’s Code Composer Studio IDE, we used the TI SYS/BIOS real-time operating system API and the TI Inter-Process Communication library API to implement the platform-dependent PPG APIs for these experiments. These APIs along with the generic, PPG-based actor implementations are fed into the TDIF-PPG Code Synthesis Engine to generate a complete parallel actor implementation.

To demonstrate the performance gain from PPG-based implementation, the actual execution time of a sequential  $79^{th}$  – order FIR filter (Seq-FIR) implementation is compared to that of our PPG-based parallel FIR filter (Par-FIR) implementation using the

Table 4.1: Execution time comparison for sequential FIR filter and parallel FIR filter implementation on different input sizes.

Input Size	1079	10079	100079	1000079
Seq-FIR (s)	0.0036	0.0336	0.334	3.34
Par-FIR (s)	0.0017	0.015	0.147	1.47
Speedup	2.11	2.24	2.27	2.27

same inputs on the targeted TI platform. The results for multiple input sizes are demonstrated in Table 4.1. The input size is in terms of the number of signal samples. The execution time is the processing time. This reported execution time excludes the time required for reading from the input FIFO and writing to the output FIFO. The FIFO reading and writing operations involve only pointer manipulations and no actual data movement, and thus have negligible impact on actor performance. The speedup is defined by the ratio between the execution time of Par-FIR and that of Seq-FIR.

The superlinear speedup is due to the VLIW feature of the employed PDSP cores. As the amount of data to process increases, so does the amount of available instruction level parallelism (ILP).

For the FFT, the execution time of the sequential FFT (Seq-FFT) implementation is compared to that of two different parallel FFT implementations. One of these parallel implementations (Par-FFT2) utilizes only data parallelism using 2 cores, while the other parallel implementation (Par-FFT3) utilizes both data parallelism and temporal parallelism using 3 cores. The execution time of Par-FFT3 is the time required to execute

Table 4.2: Execution time and latency comparison among sequential FFT, parallel FFT using 2 cores and parallel FFT using 3 cores. The results are compared for different input sizes.

Input Size	64	256	1024	4096
Seq-FFT (s)	0.00028	0.0016	0.0086	0.045
Par-FFT2 (s)	0.00021	0.001	0.005	0.255
Speedup	1.3	1.61	1.72	1.788
Par-FFT3 (s)	0.00023	0.00073	0.0039	0.021
Speedup	1.21	2.19	2.20	2.14
Latency (s)	0.00038	0.00118	0.00517	0.0257

its longest pipeline stage. The speedup is defined as the ratio between the execution time of the parallel FFT implementation (Par-FFT2 or Par-FFT3) and that of Seq-FFT. The latency is another important figure of merit. Here, the latency is defined by the elapsed time between when the FFT actor reads the first token from its input FIFO and when it writes the first result token to its output FIFO. The results are shown in Table 4.2. For Seq-FFT and Par-FFT2, the latency is equal to the execution time, so the latency is not shown separately.

From the results, we see that Par-FFT3 achieves more speedup than Par-FFT2 by introducing more latency.

Fig. 4.6(a) shows the CFDF graph for the mp-sched benchmark. In this graph, there are two paths from actor SRC to actor SNK, which represent two different signal

processing procedures on the incoming signal. In the upper path from SRC to SNK, the signal is first filtered in the time-domain and then transformed to the frequency-domain. In the lower path, the signal is first transformed to the frequency-domain and then filtered in the frequency-domain. There are both graph level parallelism and actor level parallelism in this application. We derive and experiment with four different schedules to demonstrate the performance gain and trade-offs associated with the two different forms of available parallelism.

The eight PDSP cores employed in these experiments are labeled as DSP0, DSP1, ..., DSP7. In the first schedule, all four actors are assigned to DSP0 and executed sequentially by the actor sequence FIR1, FFT2, FFT1, FIR2. Note that the two paths in the graph are independent so that control parallelism can be used. Additionally, actors in each path can be pipelined to make use of temporal parallelism in the graph. In the second schedule, we again use sequential implementations for the individual actors. The actors are manually scheduled onto 4 PDSP cores to take advantage of graph level parallelism. FIR1 is assigned to DSP0; FFT1 is assigned to DSP1; FFT2 is assigned to DSP2; and FIR2 is assigned to DSP3. FIR1 and FFT2 are fired simultaneously as pipeline stage 1, and FFT1 and FIR2 are fired simultaneously after execution of pipeline stage 1 completes. We could also schedule multiple graph iterations together to explore more temporal parallelism at the graph level. However, this approach is not chosen in our experiments because it increases the required buffer sizes.

By replacing the actor implementations with parallel versions, we derive two additional schedules, which provide the third and fourth schedules for our experiments. These schedules use two different combinations of actor level parallelism and graph level

parallelism based on the two different parallel FFT actor implementations discussed in Section 4.4.4.2. In the third schedule, FIR1 is assigned to DSP0 and DSP1 using a PPG. In similar ways, FFT1 is assigned to DSP2 and DSP3; FFT2 is assigned to DSP4 and DSP5; and FIR2 is assigned to DSP6 and DSP7. FIR1 and FFT2 are fired simultaneously as pipeline stage 1, and FFT1 and FIR2 are fired simultaneously after pipeline stage 1 completes execution.

In the fourth schedule, FIR1 is assigned to DSP0 using a PPG, and similarly, FFT1 is assigned to DSP1, DSP2, and DSP3; FFT2 is assigned to DSP4, DSP5, and DSP6; and FIR2 is assigned to DSP7. FIR1 and PPG G0 of FFT2 (see Section 4.4.4.2) are fired simultaneously as pipeline stage 1, PPG G1 of FFT2 and PPG G0 of FFT1 are fired simultaneously as pipeline stage 2, and PPG G1 of FFT1 and FIR2 are fired simultaneously as pipeline stage 3. The different schedules with the associated actor implementations are fed into the TDIF-PPG Software Synthesis Engine to generate the corresponding complete software implementations for targeted TI platform.

Using an input size of 1024, we experiment with the four different mp-sched implementations described above. The execution time, speedup and latency values for these implementations are compared on the core computation shown in Fig. 4.6(b). The remaining actors (SRC and SNK) take only approximately 1.2% of the computation time for sequential execution and thus do not have a significant impact on overall performance. The execution time is taken to be the processing time in the core computation region defined above. If pipelining is used, then the execution time is the time for the longest pipeline stage. The latency is defined as the elapsed time during the first iteration of graph execution between when the first input token enters the region and the time when

Table 4.3: Execution time and latency comparison among the 4 schedules.

Schedule	Execution Time (s)	Speedup	Latency (s)
1	0.0243	1	0.0243
2	0.00878	2.77	0.0122
3	0.00517	4.7	0.0089
4	0.0041	5.9	0.0092

the first output token leaves the region. The results are shown in Table 6.5.

From the results, we see that exploiting graph level parallelism by itself reduces the execution time by scheduling sequential actors on multiple PDSP cores, and combining graph level parallelism and actor level parallelism further reduces the execution time. For different design constraints, different combinations can be employed. For example, schedule 3 has higher execution time but lower latency compared to schedule 4. If the system has a tight latency constraint, then schedule 3 is preferable. Our approach provides the designer with more optimization opportunities from both the actor level and graph level to help satisfy the given system design requirements.

## 4.6 Summary

In this chapter, we have introduced a new dataflow based design flow for signal processing systems. This design flow, called TDIF-PPG, takes into account multiple levels in a behavioral context, and integrates graph level parallelism and actor level parallelism in MPSoC software optimization for DSP applications. Our approach is based on a new

model, called the parallel processing group (PPG), for actor design, and an associated new plug-in to the targeted dataflow interchange format (TDIF) environment. This plug-in allows designers to express parallelism within actor designs, and integrate such intra-actor parallelism with the graph level parallelism that is already exposed in TDIF.

TDIF-PPG provides useful new features in the TDIF environment, including retargetable parallel actor design, parallel actor scheduling, and early performance evaluation. To demonstrate the utility of the PPG model and the TDIF-PPG design flow, we have presented case studies involving FIR filter and FFT actor design and mp-sched application implementation on a practical multicore programmable digital signal processor platform. We have also examined useful trade-offs in the integration of graph level parallelism and actor level parallelism.

Additionally, we have motivated several directions for future work to help strengthen the utility of PPG-based actor design and integration. These include exploration of algorithms for automated scheduling of dataflow graphs that employ PPG-based parallel actor implementations; accurate and efficient functional simulation of PPG-based designs for early-stage DSP system validation; and experimentation on other kinds of state-of-the-art digital signal processing platforms.

## Chapter 5

### Scheduling Parallelized Synchronous DSP Systems on Multicore PDSPs

In this chapter, we formulate a new type of parallel task scheduling problem called Parallel Actor Scheduling (PAS). Our formulation of PAS is targeted to MPSoC mapping of DSP systems that are represented as synchronous dataflow (SDF) graphs. The developments of this chapter build directly on the techniques for actor design and modeling that we developed in Chapter 4.

In contrast to traditional SDF-based scheduling techniques, which focus on exploiting graph level (inter-actor) parallelism, the PAS problem targets the integrated exploitation of both intra- and inter-actor parallelism for platforms in which individual actors can be parallelized across multiple processing units. We address a special case of the PAS problem in which all of the actors in the DSP application or subsystem being optimized can be parallelized. For this special case, we develop and experimentally evaluate a two-phase scheduling framework with two work flows — particle swarm optimization with a mixed integer programming formulation, and particle swarm optimization with a fast heuristic based on list scheduling. We demonstrate that our PAS-targeted scheduling framework provides a useful range of trade-offs between synthesis time requirements and the quality of the derived solutions.

## 5.1 Introduction

Increases in computational power from clock frequency improvements have slowed. The trend toward multiprocessor system-on-chip (MPSoC) devices is motivated by the performance gain from simultaneous utilization of multiple processors for parallel execution of software systems. An application can be divided into tasks, each representing a piece of the computation performed by the overall software system. Tasks can have different levels of granularity, e.g., ranging from simple arithmetic operations to higher level digital signal processing (DSP) operations, such as FFT computations or adaptive filters.

By a *parallel task* in this context, we mean a task that has some inner parallelism and whose execution can be performed by multiple processors. Conversely, a *sequential task* is a task that does not have parallelism or otherwise cannot be executed on multiple processors in a given design context. Many techniques have been created for recognizing parallel tasks and exploiting their inner parallelism (e.g., see [13, 50, 32]). In addition to coming from individual software components, parallel tasks can also come from hierarchies of sequential tasks.

The performance of a software system composed of parallel tasks depends heavily on the scheduling of those tasks onto the targeted MPSoC. Each task requires different amounts of hardware resources for execution. The challenges of scheduling tasks efficiently include allocating hardware resources for competing tasks in such a way that the task demand is satisfied. The problem of deriving optimal schedules is known to be NP-hard even for highly restricted versions, such as when all tasks are independent (i.e., the

tasks do not have data dependencies among them) [18].

This chapter targets *DSP* applications, such as those associated with audio and video data stream processing, digital communications, and image processing (e.g., see [41, 67]). *DSP* applications usually require real-time processing capabilities and have critical performance constraints. Dataflow models of computation have been widely used in the design and implementation of *DSP* applications. A dataflow model is a directed graph, where vertices (*actors*) represent computational functions/tasks, and edges represent First-In-First-Out (FIFO) queues for storing data values (*tokens*) and imposing data dependencies between actors. Dataflow actors respectively consume and produce data tokens from their input edges and onto their output edges. Each actor executes as a sequence of discrete units of computation, called *firings*, where each firing depends on some well-defined amount of data from the input edges of the associated actor [42].

Traditionally, in the use of *DSP*-oriented dataflow models of computation, the firing of an actor is mapped to a unique processing element within the architecture running the application, and the dataflow model is used to exploit parallelism across actor firings [41, 74, 53]. Following the terminology in [92], we refer to an actor whose execution requires a mapping to a unique processing element as a Sequential Actor (SA). By contrast, a Parallel Actor (PA) is an actor that embeds inner (intra-firing) parallelism, and whose execution may be accelerated by use of multiple processing elements. In addition to data parallelism, pipeline parallelism and task parallelism [41, 74] at the dataflow graph level, parallel execution of PAs provides another way to optimize execution of dataflow models.

In this chapter, we consider two different forms of origin at design time for the parallelism embedded in a PA:

1. The computations within the actor are described using multiple threads. The main difference between running multiple actors (or actor firings) in parallel and running multiple threads of the same actor firing in parallel is that threads belonging to the same actor firing may flexibly share common variables, whereas executions of distinct actor firings are independent, unless dependencies are explicitly specified with First In, First Out queues (FIFOs) communication (including possible self-loop edges to implement actor state variables — i.e., dependencies across multiple firings of the same actor).

2. The actor computation is described by a dataflow subgraph (nested dataflow graph) rather than through a code module (“host language” module) that is based on a platform-oriented or otherwise non-graphical language, such as C, C++ or Java.

The Dataflow Interchange Format (DIF) language [26] is a textual language used to describe DSP applications as dataflow graphs. In DIF, a dataflow actor can be specified either as an SA or a PA to distinguish the different kinds of possibilities for mapping these kinds of actors onto target platforms.

We define the problem of scheduling an SDF graph with at least one PA onto an MPSoC as the *Parallel Actor Scheduling (PAS)* problem. The PAS problem takes into account intra-firing parallelism together with data dependencies and resource competition among distinct actor firings.

In this chapter, we focus on a special case of the PAS problem, called the fully parallelizable PAS (FP-PAS) problem, in which *all* actors in the given application are PAs. This special case is satisfied commonly, for example, in image and video processing applications. Furthermore, even if the entire application does not conform to this special case, major subsystems may satisfy the FP-PAS property, and can be optimized using the

techniques in this chapter.

We focus in this chapter on Symmetric Multi-Processing (SMP) platforms as the class of target platforms for addressing the FP-PAS problem. To address the FP-PAS problem for SMP targets, we develop and experimentally evaluate a novel two-phase scheduling framework with two alternative work flows — particle swarm optimization with a mixed integer programming formulation, and particle swarm optimization with a fast heuristic based on list scheduling. These alternative work flows provide the designer with a choice between fast synthesis time (e.g., during early stage prototyping) and high quality solutions (e.g., when deriving deployable implementations).

## 5.2 Background

### 5.2.1 Dataflow Interchange Format

The Dataflow Interchange Format (DIF) framework provides a standard approach for specifying mixed-grain dataflow-based semantics for signal processing system design [26]. The DIF Language (TDL), which is part of the DIF framework, provides a unified textual language for expressing different kinds of dataflow semantics, including graph topologies, hierarchical design structure, dataflow-related design properties, and actor-specific information.

TDL is therefore suitable for both programming and interchange (transfer of dataflow graphs across design tools). By using TDL, various kinds of signal processing systems can be represented as dataflow graphs at a high level of abstraction.

The DIF package (TDP) is a software tool that accompanies TDL, and provides a

variety of intermediate representations, analysis techniques, and graph transformations that are useful for working with dataflow graphs.

### 5.3 Problem Statement

In this section, we present a formal definition of the PAS problem, which we introduced intuitively in Section 6.1.

An instance of the PAS problem is represented as a 3-tuple  $I = (G, P, C)$ . Here,  $G = (V, E)$  represents an SDF graph with a set  $V$  of actors, and a set  $E$  of edges.  $P$  represents the set of available computational resources (e.g., a set of processors) in the target platform. Specifically, we target Symmetric Multi-Processing (SMP) platforms, which are widely used in the prototyping, design and implementation of embedded signal processing systems (e.g., see [76]).

Since the target platform is assumed to be an SMP, we assume that the set of processors in  $P$  is homogeneous. Extension of the techniques in this chapter to handle heterogeneous platforms is an interesting direction for further study.

In  $I$ , the third element  $C$  represents a function, called the *Actor Acceleration Function (AAF)*, which provides information about how much time it takes (actual or estimated) for a given actor to execute on a given number of processors.

The AAF ( $C$ ) can in general be determined by designers through profiling, although through follow-on work, one can also imagine development of static analysis techniques for automated derivation of this function. In more precise terms, the AAF  $C$  is a mapping

$$C : V \times \mathbb{N}_{>0} \rightarrow \mathbb{N}_{ext}, \quad (5.1)$$

where  $\mathbb{N}_{>0}$  represents the set of positive integers,  $\mathbb{N}_{ext}$  represents the set of extended positive integers (i.e.,  $\mathbb{N}_{ext} = \mathbb{N}_{>0} \cup \{\infty\}$ ), and  $C(a, n)$  provides the execution time of actor  $a$  when it is executed across  $n$  processors. The AAF may have “gaps” at arbitrary values of  $n$ . Such a gap means that the execution time value is not available for the associated value of  $n$  (e.g., because a corresponding library implementation does not exist or the maximum number of supported processors is exceeded). Such gaps are represented by the execution time value of  $\infty$ . Thus,  $C(a, m) = \infty$  effectively means that a useful solution to instance  $I$  cannot use exactly  $m$  processors for actor  $a$ .

A solution to the PAS problem is a mapping, called a *schedule*. Such a schedule  $L$  is a mapping from the set  $V$  of actors in  $G$  into  $(powerset(P) \times \mathbb{N}_0)$ . Here,  $powerset(P)$  represents the power set (or set of all subsets) of a given set  $P$ . In particular, if  $a \in V$  then  $L(a) = (\rho, \tau)$  indicates that actor  $a$  is scheduled to execute on all resources within  $\rho$  starting at time  $\tau$ .  $L$  determines the following characteristics associated with each PA: (1) how many processors the actor uses; (2) the processor assignment for the actor; and (3) the start time (the time when the actor begins execution).

A *feasible* schedule  $L$  must satisfy the following conditions.

1. *Data dependencies* in the given dataflow graph  $G$  must be adhered to.
2. *Resource constraints* in the targeted MPSoC must be adhered to — in other words, any given resource (i.e., any element of  $P$ ) in the targeted MPSoC can be reserved for at most one actor at any given time instant.

3. *Implementation availability.* For each actor  $a$ , we have that

$$L(a) = (\rho, \tau) \Rightarrow C(a, |\rho|) < \infty.$$

If  $L$  is a schedule and  $L(a) = (\rho, \tau)$ , then we define  $resource(L, a) = \rho$ , and  $stime(L, a) = \tau$ .

The objective of the PAS the problem is to find a feasible schedule  $L$  that minimizes the schedule length or “makespan” — i.e., to minimize the maximum completion time over all actors. For an actor  $a$  and a feasible schedule  $L$ , the completion time can be expressed as  $stime(L, a) + C(a, |resource(L, a)|)$

As discussed in Section 6.1, we address in this chapter a special case of the original PAS problem, called the FP-PAS (fully parallelizable PAS) problem. In the FP-PAS problem, we assume that all actors in  $G$  are PAs. This means that for every actor  $a$ , there exists at least one integer  $m$  such that  $m > 1$  and  $C(a, m) < \infty$ .

We assume that each actor in the dataflow graph provides the control to initialize the communication operations associated with its input and output edges, and we assume that communication between actors is carried out through shared memory. The communication time for each edge includes two parts: Shared Memory Write (SMW) time and Shared Memory Read (SMR) time. The shared memory model we assume is multiport shared memory (e.g., see [76]), which allows multiple processors to access the shared memory simultaneously. Thus, the SMWs on edges that connect to the same source actor are executed sequentially; however, the SMWs on edges that have different source actors can be executed in parallel. Similarly, the SMRs on edges that have different sink actors

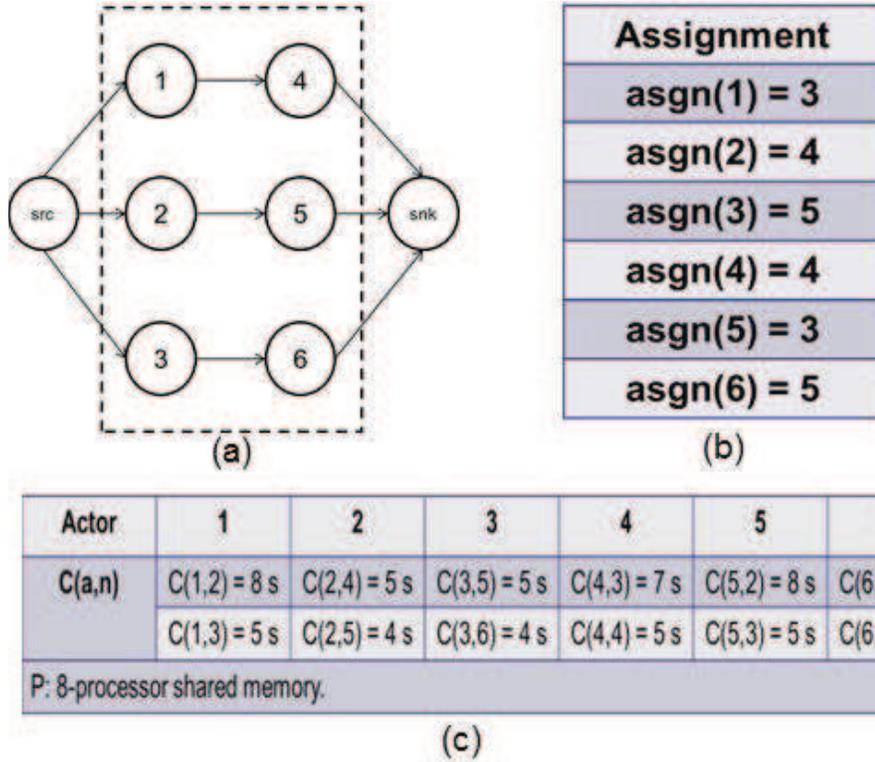


Figure 5.1: An example of an assignment-constrained FP-PAS instance.

can be executed in parallel.

Since the communication time for each edge is assumed to be fixed, we incorporate the communication cost on each edge into the AAF of its source actor and sink actor. The SMW time is incorporated into the AAF of the source actor, and the SMR time is incorporated into the AAF of the sink actor.

For SDF graphs that are multirate or contain cycles (or both), we first transform the graphs into acyclic, single rate SDF dataflow graphs using transformation techniques introduced in [41]. These transformation techniques are useful for exposing inter-firing parallelism that is embedded within multirate SDF graphs, thus providing a valuable complement to the intra-firing parallelization techniques that we introduce in this chapter.

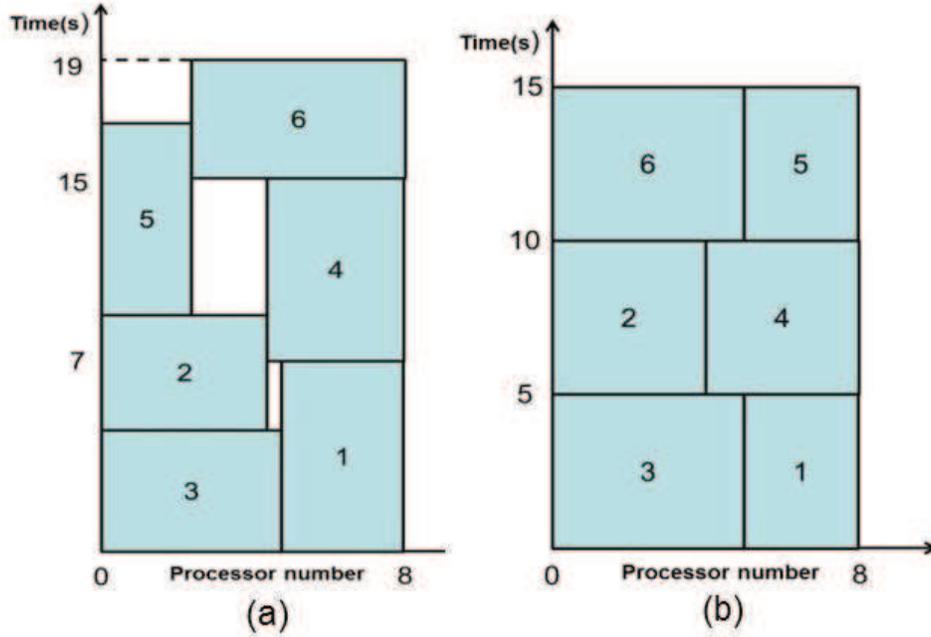


Figure 5.2: Solutions for the FP-PAS instance.

To help solve the FP-PAS problem, we define a general way to project an instance of the problem onto a specific processor count assignment. Here, by a “processor count assignment” we mean an assignment of a specific number of processors to each actor. We refer to such a projection as an assignment-constrained version of the associated instance. More precisely, suppose that we are given (1) an FP-PAS instance  $I = (G, P, C)$ , and (2) a processor count assignment  $asgn$  as a mapping  $asgn : V \rightarrow \mathbb{N}_{>0}$ . Then the assignment constrained version of  $I$  with assignment  $asgn$ , denoted  $constr(I, asgn)$ , is identical to  $I$  except that each feasible schedule  $L$  must adhere to the processor count assignment represented by  $asgn$ . This means that for each actor  $a$ , we must have that  $|resource(L, a)| == asgn(a)$ . An optimal schedule for  $I$  is a schedule that minimizes the schedule makespan over all feasible schedules for all assignment-constrained versions of  $I$ .

Fig. 5.1 gives an example of an assignment-constrained version of an FP-PAS instance. Here, the application dataflow graph  $G$  is shown in Fig. 5.1(a). We focus on scheduling the subgraph in the dashed rectangular region, which satisfies the condition that all actors are PAs.  $C$  and  $P$  are shown in Fig. 5.1(c). The processor count assignment  $asgn$  is shown in Fig. 5.1(b).

We observe that the schedule in Figure 5.2(a) is a feasible solution for the FP-PAS instance illustrated in Figure 5.1, but *not* a feasible solution for the assignment-constrained version of the FP-PAS problem. On the other hand, the schedule shown in Figure 5.2(b) is a feasible solution for the FP-PAS problem and also a feasible solution for the assignment-constrained FP-PAS problem. Moreover, the schedule in Figure 5.2(b) is an optimal schedule for the FP-PAS problem.

Intuitively, the concept of assignment-constrained versions of FP-PAS instances helps to decompose the FP-PAS problem into two distinct phases — (1) the generation of effective processor count assignments, and (2) the efficient assignment-constrained scheduling of the dataflow graph. We will apply this two-phase decomposition in later sections of this chapter.

## 5.4 Mixed Integer Programming Solution

In this section, we introduce a mixed integer programming (MIP) formulation for the assignment-constrained version of the FP-PAS problem. The formulation is based on an intermediate representation called the *computation usage graph* (CUG). The input to the assignment-constrained version of the FP-PAS problem is first transformed to a CUG.

Then, from the CUG, we formulate a corresponding instance  $\gamma$  of the MIP problem. Any existing MIP solver can then be used to solve the instance  $\gamma$ .

### 5.4.1 Computation Usage Graph

Given an instance  $I = (G, P, C)$  of the FP-PAS problem and the processor count assignment  $asgn$ , the corresponding CUG  $G'$  is a directed weighted graph  $G' = \langle V', E', \omega \rangle$ , where  $V'$  is the set of vertices,  $E'$  is the set of edges, and  $\omega : V' \rightarrow \mathbb{N}_{>0}$  assigns a positive integer weight to each vertex.

The vertices in  $V'$  are in one-to-one correspondence with the set of ordered pairs  $(x, y)$ , where  $x$  is a PA, and  $y \in \{1, 2, \dots, asgn(x)\}$ . Given a vertex  $v \in V'$ , the corresponding values of  $x$  and  $y$  are denoted as  $paractor(v)$ , and  $index(v)$ . The set of vertices in  $V'$  corresponding to a given PA  $a$  is denoted as  $S_a$ . In other words,  $S_a = \{v \mid paractor(v) = a\}$ .

For each edge  $e = (a, b)$  in  $G$ , there is an edge in  $G'$  connecting each member of  $S_a$  to each member of  $S_b$ . If we denote this set of edges by  $depend(e)$ , then we can write  $depend(e) = \{(p_1, p_2) \mid p_1 \in S_a \text{ and } p_2 \in S_b\}$ , where  $e = (a, b)$ . The edge set  $E'$  can then be expressed precisely as  $E' = \bigcup_{e \in E} depend(e)$ . Thus, intuitively, the edges in  $E'$  correspond to data dependencies between parallel actors.

The function  $\omega : V' \rightarrow \mathbb{N}_{>0}$  specifies weights for vertices in the CUG. These weights correspond to execution times, as determined by the given processor count assignments and the AAF. Thus, for each  $v \in V'$ ,  $\omega(v) = C(paractor(v), asgn(paractor(v)))$ .

For example, suppose that a dataflow graph  $G$  contains two PAs  $a$  and  $b$ , and sup-

pose that  $asgn(a) = 3$  and  $asgn(b) = 2$ . Suppose also that (1) PA  $a$  has AAF  $C(a, 3) = 5$  and  $C(a, n) = \infty$  for  $n \neq 3$ , and (2) PA  $b$  has a AAF  $C(b, 2) = 4$  and  $C(b, n) = \infty$  for  $n \neq 2$ . Finally, suppose that there is an edge  $e = (a, b)$  in  $G$ , and that we are given the processor count assignment  $asgn(a) = 3$  and  $asgn(b) = 2$ . Then in the corresponding CUG, there are 3 vertices  $A = \{a_1, a_2, a_3\}$  corresponding to PA  $a$ , and two vertices  $B = \{b_1, b_2\}$  corresponding to PA  $b$ . The vertex weights are given by  $\omega(a_i) = 5$  for  $i = 1, 2, 3$ , and  $\omega(b_i) = 4$  for  $i = 1, 2$ . Furthermore, the set of edges in the CUG can be expressed as  $(\alpha, \beta) \mid \alpha \in A$  and  $\beta \in B$ , which means that there are six edges in total.

## 5.4.2 Mixed Integer Programming Formulation

After an instance of the assignment constrained version of the FP-PAS problem has been transformed into a CUG, a corresponding instance of the MIP problem can be derived from the CUG. This derivation is outlined as follows.

### Parameters

- *Bin Assignment.*  $\forall v \in V'$  and  $\forall a \in V$ ,  $BA[v, a] = 1$  if  $a == paractor(v)$ , and  $BA[v, a] = 0$  otherwise.
- *Node Dependency.*  $\forall v_i, v_j \in V'$ ,  $ND[v_i, v_j] = 1$  if  $(v_i, v_j) \in E'$ , and  $ND[v_i, v_j] = 0$  otherwise.

### Variables

- *Mapping.*  $\forall v \in V'$  and  $\forall r \in P$ ,  $MP[v, r] = 1$  when the vertex  $v$  is assigned to processor  $r$ , and  $MP[v, r] = 0$  otherwise.

- *Non-dependent Vertex Ordering.*  $\forall$  vertices  $v_i$  and  $v_j$  such that  $v_i \neq v_j$ ,  $ND[v_i, v_j] = 0$ , and  $v_i, v_j$  are assigned to the same processor,  $NO[v_i, v_j] = 1$  when  $ST[v_i] \leq ST[v_j]$ , and  $NO[v_i, v_j] = 0$  when  $ST[v_i] > ST[v_j]$ . When  $v_i, v_j$  are assigned to different processors,  $NO[v_i, v_j] = 0$ .
- *Vertex Start Time.*  $\forall v \in V'$ ,  $ST[v]$  is the scheduled start time of vertex  $v$ .
- *Actor Start Time.*  $\forall a \in V$ ,  $SA[a]$  is the scheduled start time of actor  $a$ .
- *Pair Vertex Assignment.*  $\forall v_i, v_j \in V'$  and  $\forall r \in P$ ,  $NA[v_i, v_j, r] = 1$  when  $v_i$  and  $v_j$  are both assigned to processor  $r$ , and  $NA[v_i, v_j, r] = 0$  otherwise.
- *Vertex Collocation Relation.*  $\forall v_i, v_j \in V'$ ,  $CR[v_i, v_j] = 1$  when  $v_i$  and  $v_j$  are assigned to two different processors, and  $CR[v_i, v_j] = 0$  otherwise.

## Constraints

- *Vertex Mapping.* Every vertex is mapped to exactly one processor:

$$\forall v \in V', \sum_{r \in P} MP[v, r] = 1.$$

- *Actor Start Time.* Every vertex belonging to the same actor has the same start time:

$$\forall v \in V', ST[v] = \sum_{a \in V} BA[v, a] \times SA[a].$$

- *Dependent Vertex Start Time.*

$$\forall (x, y) \in E', ST[y] \geq ST[x] + \omega(x).$$

- *Non-dependent Vertex Start Time.*  $\forall$  vertices  $v_i, v_j \in V'$  such that  $v_i \neq v_j$  and  $ND[v_i, v_j] = 0$ :

$$ST[v_i] \geq ST[v_j] + \omega[v_j] - K(1 - NO[v_i, v_j] + CR[v_i, v_j]), \text{ and}$$

$$ST[v_j] \geq ST[v_i] + \omega[v_i] - K(NO[v_i, v_j] + CR[v_i, v_j]),$$

where  $K = \sum_{v \in V'} \omega(v)$ .

- *Other constraints.*  $\forall$  vertices  $v_i, v_j \in V'$  such that  $v_i \neq v_j$ , and  $\forall r \in P$ :

$$NA[v_i, v_j, r] \geq MP[v_i, r] + MP[v_j, r] - 1$$

$$NA[v_i, v_j, r] \leq MP[v_i, r]$$

$$NA[v_i, v_j, r] \leq MP[v_j, r]$$

$$CR[v_i, v_j] \geq \sum_{r \in P} NA[v_i, v_j, r]$$

The objective function is to minimize the maximum completion time over all vertices  $v \in V'$ , where the completion time  $completion(v)$  of a vertex  $v$  is defined by  $completion(v) = ST[v] + \omega[v]$ . The objective function is thus to minimize  $M$ , where

$$M = \max(\{completion(v) \mid v \in V'\}).$$

## 5.5 List Scheduling Solution

In this section, we develop a novel list scheduling algorithm for the assignment-constrained version of the FP-PAS problem. We call our algorithm the *story scheduling algorithm*.

We first clarify some notation that is used in our algorithm formulation. A *sink vertex* is a vertex in a directed graph that has no outgoing edges. The *level*  $l_i$  of a vertex  $a_i$  in a vertex-weighted dataflow graph  $G$  is defined as the length of the longest path (in terms of cumulative vertex weights) from vertex  $a_i$  to a sink vertex.

During the list scheduling process, the *dependence count*  $DC_i$  of a vertex  $a$  in a dataflow graph  $G = (V, E)$  is initially equal to the number of incoming edges of  $a$ . Whenever a vertex  $a$  is scheduled, the dependence counts of all vertices in the set  $successors(a)$  are decremented (decreased by 1), where  $successors(a) = \{b \in V \mid (a, b) \in E\}$ . At a given point during the scheduling process, a *free vertex* is a vertex that has dependence count equal to 0, but has not yet been scheduled.

We view the set  $P$  of processing resources and time  $T$  as a 2-dimensional *computation resource space*. We consider each SDF vertex  $a \in V$  as a rectangle for placement within this space. Given a processor count assignment  $asgn$ , the rectangle associated with a vertex  $a$  has width  $w(a) = asgn(a)$  and height  $h(a) = C(a, asgn(a))$ . The assignment-constrained version of the FP-PAS problem can be viewed as the problem of placing the rectangles associated with the SDF vertices in the computation resource space in such a way that no two rectangles overlap, and all of the dataflow graph dependencies are satisfied.

### 5.5.1 Story Scheduling Overview

Each time interval  $[t_1, t_2]$  specifies a region in the computation resource space that we called a *story*. We refer to  $t_1$  is the *floor* of the story and  $t_2$  as the *ceiling* of the story.

Two stories are *adjacent* if the floor of one story is equal to the ceiling of the other story. The proposed algorithm constructs stories one at a time for different dataflow graph actors, and conceptually lays these stories out in the computation resource space by adhering to resource and data dependence constraints. Intuitively, the algorithm repeatedly selects a free actor  $a$ , constructs and lays out a story for  $a$ , decrements any relevant dependence counts, and then repeats this iterative process.

The *available processor count (APC)* for a given story is initially equal to  $|P|$ , which means that a total of  $|P|$  processors can be assigned to actors scheduled in the story. When a story  $s = [t_1, t_2]$  is under construction, only its floor  $t_1$  is known; its ceiling  $t_2$  is determined as soon as construction of  $s$  is complete. When a free vertex  $a$  is evaluated for scheduling into the story  $s = [t_1, t_2]$  that is currently under construction, the actor's processor count assignment  $asgn(a)$  is compared to the APC. If  $asgn(a) \leq APC$ , then the start time of  $a$  is set to the floor  $t_1$  of  $s$ , and the APC is updated by  $APC - asgn(a)$ . On the other hand, if  $asgn(a) > APC$ , then  $a$  cannot be scheduled into  $s$ .

If no free vertex can be scheduled into  $s$ , then the story's construction is complete, and the story's ceiling  $t_2$  is determined as  $t_2 = \max(\{C(b, asgn(b)) + t_1 \mid b \in actors(s)\})$ , where  $actors(s)$  represents the set of actors that have been scheduled into story  $s$ .

## 5.5.2 Algorithm Description

Our story scheduling algorithm operates in two phases — the *preparation phase*, and the *core loop phase*. These phases are described as follows.

**Preparation phase.** The level for each vertex is calculated using Dijkstra’s algorithm, which has time complexity of  $O(|V|^2)$ . Then the dependence count for each graph vertex is initialized. This initialization process has complexity  $O(|E|)$ . Thus, the overall time complexity for the preparation phase is  $O(|V|^2)$ .

**Core loop phase.** At the beginning of each iteration in the core loop phase, a story and its APC are initialized. Then, the free vertices are sorted in descending order of their levels. If a tie occurs during this sorting process, then the vertex that requires the most processors (among the set of tied vertices) is selected first. The free vertices are selected one by one based on their positions in the sorted list, and evaluated for scheduling based on the process described in Section 5.5.1. A given core loop iteration completes when construction of the current story is completed, and then if there are one or more actors that remain to be scheduled, the construction process continues in the next iteration with the initialization and construction of a new story.

The core loop phase ends when all vertices in the input SDF graph  $G = (V, E)$  have been scheduled. The time complexity for the core loop phase is  $O(|E| + |V|\log|V|)$ . We omit details of this complexity derivation due to space limitations. We see that the time complexity of the story scheduling algorithm is dominated by that of the preparation phase, and can be expressed as  $O(|V|^2)$ .

### 5.5.3 Example

In this section, we present a simple example to illustrate the story scheduling algorithm. Consider the instance of the assignment-constrained version of the FP-PAS prob-

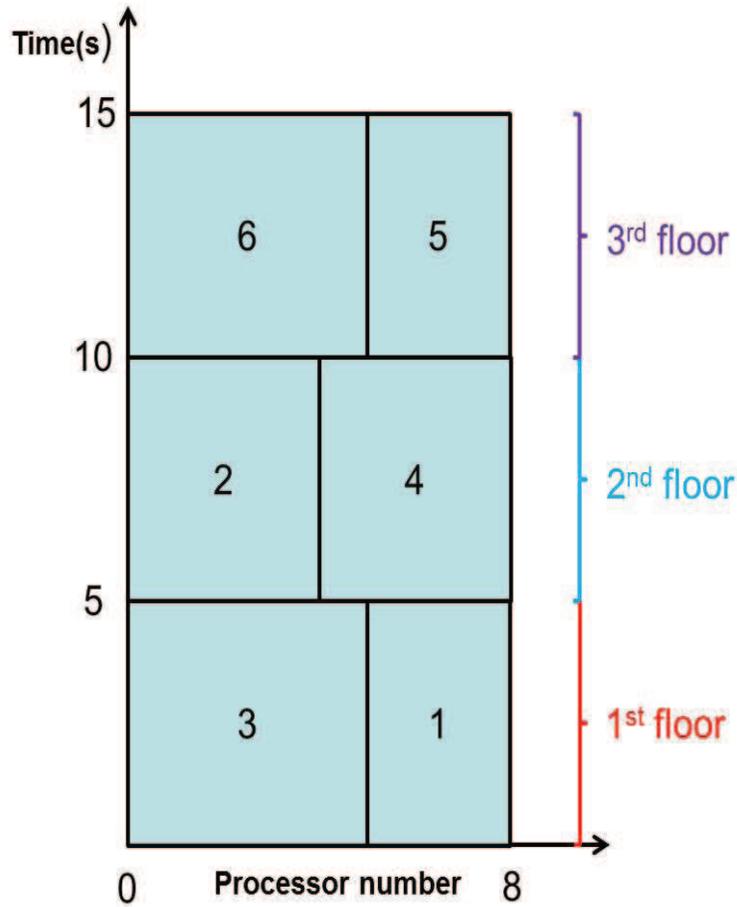


Figure 5.3: Result of applying the story scheduling algorithm on the example of Fig. 5.1. It can be verified that for this simple example, the solution shown in Fig. 5.3 is optimal. In general, however, the story scheduling algorithm is not guaranteed to produce optimal solutions, and can be viewed as a fast heuristic.

## 5.6 Two Phase Scheduling Framework

In Section 5.4 and Section 5.5, we introduced alternative algorithms for the assignment-constrained version of the FP-PAS problem. In this section, we build on these algorithms,

and introduce a two-phase scheduling framework, called the *Two-Phase FP-PAS Framework (TPFF)*, for solving the general FP-PAS problem (i.e., the FP-PAS problem without assignment constraints).

The TPFF is illustrated in Fig. 5.4. In the first phase, a candidate processor count assignment is generated using a *Particle Swarm Optimization (PSO)* [33] engine. Such candidate processor count assignments are used to project the given instance of the FP-PAS problem onto an assignment-constrained version of the FP-PAS problem. In the second phase of the TPFF, a solution to the assignment-constrained version of the FP-PAS problem is derived by using either the MIP problem formulation of Section 5.4 or the list scheduling heuristic of Section 5.5. The quality of the derived solution is then evaluated and used as feedback to the PSO engine to help generate new processor count assignments.

The TPFF framework iteratively optimizes the schedule until a given termination condition is satisfied. The set of supported termination conditions in our implementation of the framework currently includes a specific time limit for optimization or a maximum number of PSO iterations.

PSO [33] is a computational method that derives an optimized solution by iteratively trying to improve a candidate solution with regard to a given measure of quality. PSO performs optimization by maintaining a population of candidate solutions, where each candidate solution is referred to as a *particle*. Each particle's "movement" (trajectory through the underlying design space or problem space) is influenced by its local best known position, and is also guided toward the best known positions in the search-space, which are updated as better positions are found by other particles.

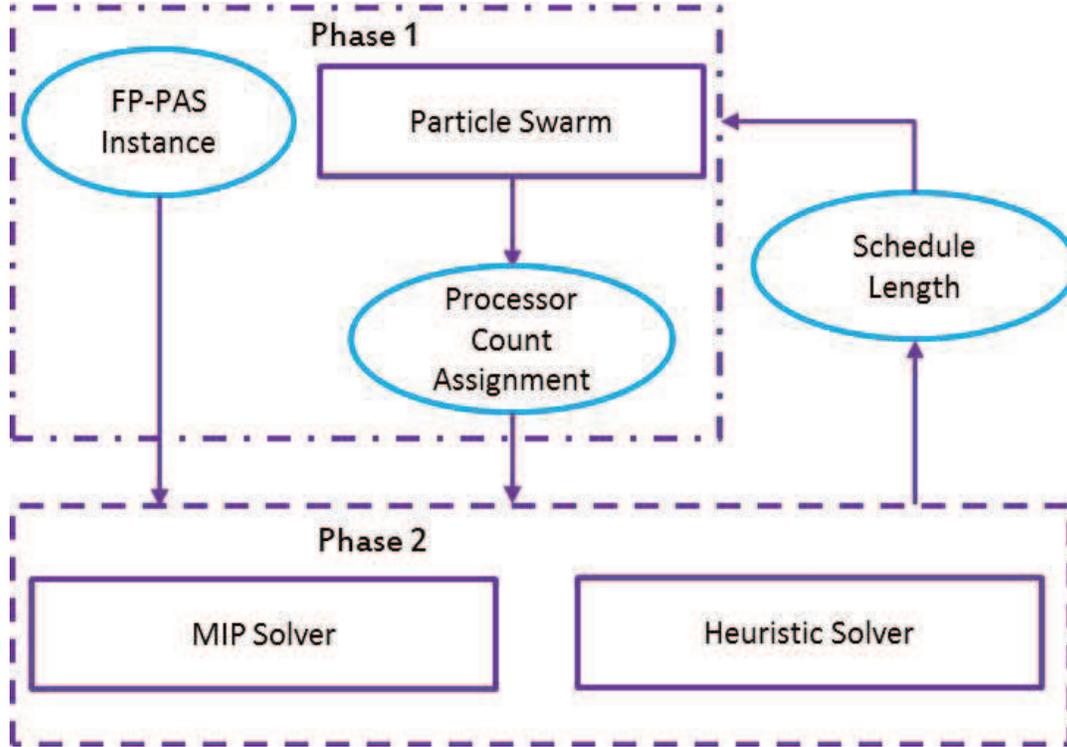


Figure 5.4: Two-phase scheduling framework for the FP-PAS problem.

Our PSO formulation is inspired by PSO techniques introduced in [90]. In our PSO formulation, a particle  $p$  in the PSO swarm encapsulates a processor count assignment  $asgn_p$  (using a minor abuse of notation).  $asgn_p(a)$  denotes the processor count assignment for a specific actor  $a$ . The PSO swarm  $S$  is a set of particles. For each particle  $p$ ,  $opt_p$  stores the best processor count assignment, in terms of schedule makespan, found so far for the particle. In case of ties for the best processor count assignment, the oldest (least recently encountered) solution is stored. For the swarm  $S$ ,  $opt_S$  stores the best processor count assignment found so far for the swarm, with ties handled in a similar way.

During initialization, a swarm is created by randomly generating its particles. Each particle in the current swarm along with the FP-PAS instance that has been input to the framework are passed to the second scheduling phase to derive a schedule. The schedule

makespan is then used to evaluate the schedule.

During optimization, each particle  $p$  in the current swarm is modified. The resulting set of particle modifications leads to a new swarm, which then becomes the “current swarm” for the next optimization iteration. The modification to a given particle is determined by two “force values”:  $lforce_p$  and  $gforce_p$ . Intuitively, the *local force*  $lforce_p$  tries to move the current particle towards  $opt_p$ , while the *global force*  $gforce_p$  tries to move the current particle towards  $opt_S$ . Key equations that govern the particle modification process are summarized as follows.

$$lforce_p(a) = W_1 \times rand() \times (opt_p(a) - asgn_p(a)) \quad (5.2)$$

$$gforce_p(a) = W_2 \times rand() \times (opt_S(a) - asgn_p(a)) \quad (5.3)$$

$$asgn_p(a) = asgn_p(a) + lforce_p(a) + gforce_p(a) \quad (5.4)$$

Here,  $W_1$  and  $W_2$  are weights to adjust the impact of local and global forces, respectively, on a particle’s movement, and  $rand()$  generates random integers. This use of randomization helps the optimization process to avoid “getting stuck” in local minima.

## 5.7 Related Work

Static MPSoC scheduling problems can be classified into four categories based on the task models: (1) independent sequential task scheduling (*ISTS*), (2) dependent sequential task scheduling (*DSTS*), (3) independent parallel task scheduling (*IPTS*), and (4)

dependent parallel task scheduling (*DPTS*).

For the ISTS problem, Dogramaci et al. introduce a dynamic programming approach to derive optimal schedules [15].

The DSTS problem is NP-hard, so heuristics and meta algorithms have been explored. Wu et al. [81] propose an algorithm called Modified Critical Path (MCP). MCP calculates the As-Late-As-Possible (*ALAP*) scheduling times of all tasks and ranks the tasks in order of ascending ALAP times. Tasks are selected one by one from the sorted list, and each task is assigned to the processor that provides the earliest start time. Omara et al. [51] introduce a genetic/evolutionary algorithm for the DSTS problem. An initial schedule is first formed as a chromosome. Mutations alter one or more genes in such “schedule chromosomes” to generate new chromosomes. The chromosomes are evaluated by a fitness function that assesses schedule performance. Genetic operations and fitness evaluation on schedule chromosomes are repeated until a given termination condition is reached.

The IPTS problem is also NP-hard. Depending on the targeted application domain, preemption of tasks may or may not be allowed. For preemptive IPTS with task migration, Klaus et al. [28] develop a linear programming algorithm and that is guaranteed to find an optimal solution in  $O(n) + poly(m)$  time, where  $n$  is the number of tasks,  $poly$  is a univariate polynomial, and  $m$  is an expression involving exponential variables.

Both preemptive and non-preemptive versions of the DPTS problem have been studied. The non-preemptive DPTS problem was investigated by Du et al. in [17] and shown to be strongly NP-hard. Algorithms for non-preemptive DPTS are discussed in [21, 46]. However, the solutions provided by these works are restricted to handling only 2 or 3 pro-

processors in the target (parallel processing) platform. The non-preemptive DPTS problem is similar to the FP-PAS problem that we target in this chapter; however, conventional approaches to non-preemptive DPTS do not consider interprocessor communication costs or multiple implementations of the same actor.

In [91], Zaki et al. formulate an MIP problem for scheduling SDF graphs on heterogeneous platforms that consist of graphics processing units and general-purpose processors. Our MIP formulation for CUG scheduling is similar in some ways to Zaki's; however, our MIP problem formulation is novel in that it considers alternative implementations of actors (actor acceleration functions) across different numbers of processors in addition to considering graph-level data dependencies.

## 5.8 Experiments

We have developed a prototype implementation of our FP-PAS scheduling framework, TPF, in the DIF environment [26]. In this section, we present experimental results obtained using this prototype implementation. We experiment with two different workflows within TPF: *PSO + MIP* and *PSO + Heuristic*, where *Heuristic* refers to the list scheduling heuristic presented in Section 5.5.

In our experiments, we use PREESM [53] to generate random SDF graphs in DIF format as one form of input for our scheduling framework. The SDF graphs are randomly generated from the following constraints: (1) each actor has between 1 and 3 input ports and between 1 and 5 output ports; (2) each actor has a load of  $rand10() \times 50$ , representing the execution time of the actor on a single processor, where  $rand10()$  generates a random

integer between 1 and 10; and (3) each edge has a communication cost of  $rand10() \times 6$ , representing data movement time through shared memory.

Due to the overhead of synchronization and communication, linear speedup in actor execution time on multiple processors is difficult to obtain in many realistic scenarios. Thus, we assume a logarithmic ( $\log_{1.5}$ ) speedup profile to generate AAFs. We assume that each actor can be sped up by 2 to 8 processors in the generated AAFs. These models of random SDF graph, and AAF generation provide a flexible mechanism for constructing a diverse set of synthetic benchmarks to test our framework under different conditions.

We also assume that the time for SMW and SWR on each edge is the same. Thus, half of the communication cost on an edge is attributed to its source actor, and the other half is attributed to its sink actor. The logarithmic speedup model for AAF generation results in general in fractional values; all such fractional values are rounded to their nearest integers.

For example, if the sequential execution time of actor 1 is 450 and the communication cost attributed to it is 30, then a representative “computation-only” AAF, and an extended AAF ( $AAF'$ ) that includes communication costs for actor 1 are shown in Fig. 5.5.

In our experiments, we empirically set the swarm size of the PSO to 2,  $W_1$  and  $W_2$  to 0.5, and  $rand()$  to generate integers between 0 and 4 in Equation 5.2 and Equation 5.3. We constrain the runtime of the two-phase scheduling framework by two termination criteria: one is the maximum number of optimization iterations  $N_o$  for the PSO, and the other is the maximum runtime  $N_r$  for optimization. We set  $N_o = 20$  and  $N_r = 24$ hours. We assume a  $|P| = 8$  multiport shared memory SMP platform.

Actor 1	AAF	AAF'
2 processor	$C(1,2) = 265$	$C(1,2) = 295$
3 processor	$C(1,3) = 167$	$C(1,3) = 197$
4 processor	$C(1,4) = 132$	$C(1,4) = 162$
5 processor	$C(1,5) = 113$	$C(1,5) = 143$
6 processor	$C(1,6) = 102$	$C(1,6) = 132$
7 processor	$C(1,7) = 94$	$C(1,7) = 124$
8 processor	$C(1,8) = 88$	$C(1,8) = 118$

Figure 5.5: Generated AAF and  $AAF'$  for actor 1.

The experimental results presented in this section are all obtained by executing the TPF on an Intel Core i7-2600K CPU (3.40GHz). Table 5.1 shows the results of the two evaluated work flows for solving the FP-PAS problem. Each entry presents results for a randomly generated SDF graph. For each work flow, the optimization run time and the graph schedule length (makespan) are given. The column labeled “Ratio” presents the ratio of the graph schedule length obtained by  $PSO + Heuristic$  compared to that of  $PSO + MIP$ . An entry labeled  $NF$  indicates that no feasible schedule was found.

As demonstrated in Table 5.1, the optimization runtime of  $PSO + MIP$  grows exponentially with the number of vertices and edges in the SDF graph. The reason is that the MIP solver utilizes a branch and bound method to find an optimal schedule for the assignment constrained version of the FP-PAS problem. Despite the optimality of the

solutions it provides, the *PSO + MIP* work flow fails to finish all optimization iterations for problems with more than 30 actors. Furthermore, the *PSO + MIP* work flow fails to find any feasible solutions for problems with more than 60 actors.

In contrast, the runtime of the *PSO + Heuristic* work flow is roughly proportional to  $|V|^2$ , which demonstrates good scalability. Solutions for all cases are calculated in less than 1 second. The results in Table 5.1 also show that the lengths of the schedules generated from the *PSO + Heuristic* work flow are shorter than those of the schedules generated by *PSO + MIP* for problems with more than 30 actors. Intuitively, this improvement arises because the heuristic-embedded workflow is able to explore more processor count assignments. When both work flows finish all of their optimization iterations, *PSO + Heuristic* generates schedules that are 11% longer on average compared to the schedules generated by *PSO + MIP*; however, these moderately under-performing schedules are generated much faster by the heuristic-embedded work flow.

We also experimented with the TPF on a practical application. Specifically, we incorporated the TPF into the TDIF-PPG framework [92] to implement an Image Registration (IR) application [82] on the Texas Instruments (TI) 6678L multicore programmable digital signal processor (PDSP) [76]. The TI 6678L has 8 PDSP cores, where each core runs at 1.0 Ghz. The TI 6678L also has 32KB L1 data cache, 32 KB L1 instruction cache, and 512KB L2 cache. Additionally, the device has 4MB of multiport shared SDRAM, which allows up to 4 “masters” to access it simultaneously.

DSP applications typically consist of different DSP algorithm kernels with data being streamed through the kernels sequentially. With little temporal locality in the data, data cache structures provide little benefit in terms of power and performance over soft-

Table 5.1: Performance of both work flows on randomly generated SDF graphs.

SDF Graphs		PSO+MIP		PSO+Heuristic		
$ V $	$ E $	Time	Length	Time	Length	Ratio
10	12	420 <i>s</i>	1300	0.192 <i>s</i>	1450	1.12
15	20	1.2 <i>hr</i>	1850	0.228 <i>s</i>	2150	1.16
20	31	3.8 <i>hr</i>	1800	0.276 <i>s</i>	1950	1.08
30	54	23.8 <i>hr</i>	4600	0.38 <i>s</i>	5050	1.1
40	57	24 <i>hr</i>	4850	0.46 <i>s</i>	4050	0.83
50	98	24 <i>hr</i>	7750	0.58 <i>s</i>	6700	0.86
60	107	24 <i>hr</i>	9300	0.61 <i>s</i>	7050	0.76
70	93	24 <i>hr</i>	NF	0.62 <i>s</i>	9100	NA
80	98	24 <i>hr</i>	NF	0.66 <i>s</i>	10150	NA
90	194	24 <i>hr</i>	NF	0.77 <i>s</i>	10800	NA
100	274	24 <i>hr</i>	NF	0.91 <i>s</i>	12050	NA

ware controlled scratchpad memories (*SPM*). Thus, for our experiments, we disable all levels of cache in the TI 6678L and configure them as *SPMs*.

Our targeted IR application [82] involves a significant amount of matrix operations, which have high levels of data parallelism. Under TI’s Code Composer Studio integrated development environment, we use the TI Inter-Process Communication (*IPC*) library to develop the multithreaded implementation for each actor in the dataflow graph, except for the file reading actor and file writing actor, which are used as input/output interfaces for the graph. *IRSUB* is a subgraph of the original dataflow graph that is composed entirely

of PAs, and is therefore a suitable input for the FP-PAS problem. In sequential execution on a single PDSP core, IRSUB uses up to 98% of the execution time of the whole dataflow graph. Thus, if we can accelerate IRSUB, we can provide significant acceleration for the overall IR application.

Using the TI multicore system analyzer, we derive the AAF for each actor by profiling its execution time on different numbers of TI 6678L cores. We also obtain the communication cost on each edge by measuring the associated data write time and data read time to and from shared memory. All of these measurements are in terms of PDSP cycles. We input the AAF and communication cost information to the TPFf, and apply the framework using both of our implemented work flows, *PSO + Heuristic* and *PSO + MIP*.

As shown in Table 5.2, both work flows derive the same schedule for this application. We refer to this common schedule as “Schedule 1”. The *PSO + Heuristic* work flow is much faster than the *PSO + MIP* work flow in deriving this schedule.

To further demonstrate the utility of the TPFf framework, we use Schedule 1 to develop a fully functional implementation of the IR application on the TI 6678L platform. The actual graph execution time is  $2.36E10$  PDSP cycles, which is 10% more than the estimated schedule length. We also manually create an optimal schedule that maps IRSUB to the target multi-core platform using only sequential actor implementations. We refer to this sequential-actor-only schedule as “Schedule 2”. We implement Schedule 2 on the TI 6678L platform and measure its performance. The result of this measurement indicates that when implemented on the TI 6678L platform, Schedule 1 provides a  $1.97X$  speedup compared to Schedule 2. This speedup can be viewed as a performance gain enabled by

Table 5.2: Performance of both TPFf work flows on an image registration application.

Graphs	PSO+MIP		PSO+Heuristic		
	Time	Length	Time	Length	Ratio
IRSUB	1.1 <i>hr</i>	2.14E10	0.21 <i>s</i>	2.14E10	1

exploiting both intra-actor and inter-actor parallelism compared to using only inter-actor parallelism.

## 5.9 Summary

In this chapter, we introduced a new scheduling problem, called the Parallel Actor Scheduling (PAS) problem, that is targeted to optimized MPSoC implementation of synchronous dataflow (SDF) graphs. We focused on an important special case of the PAS problem, called the fully parallelizable PAS (FP-PAS) problem, in which all actors in the given application are parallel actors. We utilized multiple levels in both structural and behavioral contexts of the targeted DSP system to derive our solution approach. We developed a two-phase scheduling framework, called TPFf, for the FP-PAS problem using (1) particle swarm optimization techniques in conjunction with (2) mixed integer programming and fast heuristic techniques that optimize FP-PAS solutions for fixed processor count assignments. We used a diverse set of randomly generated SDF graphs and a practical image processing application to demonstrate the effectiveness and utility of the TPFf framework in integrating both intra- and inter-actor parallelism for optimized MPSoC implementation.

## Chapter 6

### A Cross-platform Design Flow for DSP Applications

In the previous chapters of this thesis, we introduced several scheduling techniques that address different design levels and combinations of levels in the mapping of signal processing applications onto parallel and distributed platforms. In this chapter, we develop a design methodology for integrating and applying different scheduling techniques in a way that helps engineers to efficiently explore the diverse design spaces associated with state-of-the-art signal processing systems. In particular, we discuss a CFDF-based design flow and associated design methodology for efficient simulation and implementation of DSP applications. The design flow supports system formulation, simulation, validation, cross-platform software implementation, instrumentation, and system integration capabilities to derive optimized signal processing implementations on a variety of platforms. We provide a comprehensive specification of the design methodology using the lightweight dataflow (LWDF) and targeted dataflow interchange format (TDIF) tools, and demonstrate it with case studies on CPU/GPU and multicore PDSP designs that are geared towards fast simulation, rapid transition from simulation to deployment, high performance implementation, and power-efficient acceleration. Material in this chapter was published in preliminary form in [95].

## 6.1 Introduction

As embedded processing platforms become increasingly diverse, designers must evaluate trade-offs among different kinds of devices such as CPPs, graphics processing units (GPUs), multicore programmable digital signal processors (PDSPs), and field programmable gate arrays (FPGAs). The diversity of relevant platforms is compounded by the trend towards integrating different kinds of processors onto heterogeneous multicore devices for DSP (e.g., see [4]). Such heterogeneous platforms help designers to simultaneously achieve manageable cost, high power efficiency, and high performance for critical operations. However, there is a large gap from the simulation phase to the final implementation. Simulation is used extensively in the early stage of system design for high-level exploration of design spaces and fast validation. In contrast, in the implementation phase, there is strong emphasis platform dependent issues, performance centric optimization, and tuning low-level implementation trade-offs. A seamless design flow is needed to help designers to effectively bridge this gap.

Dataflow models of computation have been widely used in the design and implementation of DSP applications, such as applications for audio and video stream processing, digital communications, and image processing (e.g., see [4, 41]). These applications often require real-time processing capabilities and have critical performance constraints. Dataflow provides a formal mechanism for specifying DSP applications, imposes minimal data-dependency constraints in specifications, and is effective in exposing and exploiting task or data level parallelism for achieving high performance implementations.

A *dataflow graph* is a directed graph, where vertices (*actors*) represent computa-

tional functions (*actors*), and edges represent first-in-first-out (FIFO) channels for storing data values (*tokens*) and imposing data dependencies between actors. In DSP-oriented dataflow models of computation, actors can typically represent computations of arbitrary complexity as long as the interfaces of the computations conform to dataflow semantics. That is, dataflow actors produce and consume data from their input and output edges, respectively, and each actor executes as a sequence of discrete units of computation, called *firings*, where each firing depends on some well-defined amount of data from the input edges, and produces some well-defined amount of data onto the output edges of the associated actor [42].

Dataflow modeling has several advantages, which facilitate its use as a single, unified model across the simulation phase and implementation phases of the design flow. First, dataflow utilizes simple, standard, and loosely coupled interfaces (in the form of FIFOs), which can be easily implemented on individual platforms and across different platforms. Second, the structure of actor interfaces can be cleanly separated from the internal operation of actors. As long as the predefined amounts of data from the input edges are consumed and the predefined amounts of data to the output edges are produced on each firing, the implementation of an actor can be changed without violating key higher level properties of the enclosing dataflow model. Third, dataflow models enable a wide variety of powerful techniques for design analysis and optimization — e.g., for evaluating or optimizing performance, memory usage, throughput and latency [4].

Based on *Core Functional Dataflow (CFDF)* [56], two complementary tools have been developed in recent years. First, the *lightweight dataflow (LWDF)* programming methodology [69] provides a “minimalistic” approach for integrating coarse grain dataflow

programming structures into DSP simulation for fast system formulation, validation and profiling with arbitrary languages. Second, the *targeted dataflow interchange format (TDIF)* framework [71] provides cross-platform actor design support, and the integration of (1) code generation for programming interfaces, and (2) low level customization for implementations targeted to homogeneous and heterogeneous platforms alike. In this chapter, we present a novel dataflow-based design flow that builds upon on both LWDF and TDIF to allow rapid transition from simulation to optimized implementations on diverse platforms. In this chapter, we present this design flow, and provide case studies to demonstrate its application.

## 6.2 Background

In this section, we review background that is needed to introduce our design flow, which is introduced in the next section.

### 6.2.1 Lightweight Dataflow

*Lightweight dataflow (LWDF)* [69] is a programming approach that allows designers to integrate various dataflow modeling approaches relatively quickly and flexibly into existing design methodologies and processes. LWDF is designed to be minimally intrusive on existing design processes and requires minimal dependence on specialized tools or libraries. LWDF can be combined with a wide variety of dataflow models to yield a lightweight programming method for those models. In our design flow, the LWDF is used for system simulation.

In LWDF, an actor has an *operational context*, which encapsulates the following entities related to an actor design [68]:

- Actor parameters.
- Local actor variables — variables whose values store temporary data that does not persist across actor firings.
- Actor state variables — variables whose values do persist across firings.
- References to the FIFOs corresponding to the input and output ports (edge connections) of the actor as a component of the enclosing dataflow graph.
- Terminal modes: a (possibly empty) subset of actor modes in which the actor cannot be fired.

In LWDF, the operational context for an actor also contains a *mode variable* whose value stores the next CFDF mode of the actor and persists across firings. The LWDF operational context also includes references to the `invoke` function and `enable` function of the actor. The concept of terminal modes, defined above, can be used to model finite subsequences of execution that are “re-started” only through external control (e.g., by an enclosing scheduler). This is implemented in LWDF by extending the standard CFDF `enable` functionality such that it unconditionally returns `false` whenever the actor is in a terminal mode.

## 6.3 From simulation to implementation

In this section, we elaborate on the process of applying our proposed design flow based on LWDF and TDIF, and we discuss useful features of the design flow.

### 6.3.1 Step 1: System Formulation

In the first step of the design flow, the targeted DSP application is modeled in terms of CFDF semantics. Decisions are made on how to decompose the application specification into actors, along with the functionality of and parameterization associated with each actor.

Next, we design FIFOs and actors or select such design components from predefined libraries. Actor design in LWDF includes four interface functions — the *construct*, *enable*, *invoke*, and *terminate* functions. The *construct* function can be viewed as a form of object-oriented constructor, which connects an actor to its input and output edges (FIFO buffers), and performs any other pre-execution initialization associated with the actor. Similarly, the *terminate* function performs any operations that are required for “closing out” the actor after the enclosing graph has finished executing (e.g., deallocation of actor-specific memory or closing of associated files).

In the *enable* function for an LWDF actor  $a$ , a `true` value is returned if

$$population(e) \geq cons(a, m, e) \text{ for all } e \in inputs(a); \quad (6.1)$$

$$population(e) \leq (capacity(e) - prod(a, m, e)) \text{ for all } e \in outputs(a); \text{ and} \quad (6.2)$$

$$m \notin \tau(a), \quad (6.3)$$

where  $m$  is the current mode of  $a$ .

In other words, the enable function returns `true` if the given actor is not in a terminal mode, and has sufficient input data to execute the current mode, and the output edges of the actor have sufficient data to store the new tokens that are produced when the mode executes. An actor can be invoked at a given point of time if the enable function is `true`-valued at that time.

In the invoke function of an LWDF actor  $a$ , the operational sequence associated with a single invocation of  $a$  is implemented. Based on CFDF semantics, an actor proceeds deterministically to one particular mode of execution whenever it is enabled, and in any given mode, the invoke method of an actor should consume data from at least one input or produce data on at least one output (or both) [56]. Note that in case an actor includes state, then the state can be modeled as a self-loop edge (a dataflow edge whose source and sink actors are identical) with appropriate delay, and one or more modes can be defined that produce or consume data only from the self-loop edge. Thus, modes that affect only the actor state (and not the “true” inputs or outputs of the actor) do not fundamentally violate CFDF semantics, and are therefore permissible in LWDF.

After completing actor design and selection, we focus an analogous process for FIFOs. FIFO design for LWDF dataflow edge implementation is orthogonal to the design of dataflow actors in LWDF. That is, by using LWDF, application designers can focus on design of actors and mapping of edges to lower level communication protocols through

separate design processes (if desired) and integrate them later through well-defined interfaces. Such design flow separation is useful due to the orthogonal objectives, which center around computation and communication, respectively, associated with actor and FIFO implementation.

Standard FIFO operations in LWDF include operations that perform the following tasks:

- Create a new FIFO with a specified capacity;
- Read and write tokens from and to a FIFO;
- Check the capacity of a FIFO;
- Check the number of tokens that are currently in a FIFO;
- Deallocate the storage associated with a FIFO (e.g., for dynamically adapting graph topologies or, more commonly, as a termination phase of overall application execution).

The buffer sizes used during simulation and implementation are determined based on the scheduling strategies that are employed. Our proposed design flow facilitates experimentation with alternative scheduling strategies to help designers explore trade-offs involving buffer sizes and other relevant implementation metrics, such as latency and throughput. More details on scheduling are provided later in this section.

### 6.3.2 Step 2: System Validation and Profiling

After system formulation, we need a schedule to validate the correctness of the system and the behavior of each actor. A CFDF *canonical schedule* [58] can be employed for this purpose. A canonical schedule repeatedly traverses the list of actors in a CFDF graph according to some pre-defined sequence. When an actor is visited during this traversal process, its enable function is first called. If the enable function returns `true`, the invoke function for the actor is subsequently called; otherwise, firing of the actor is effectively “skipped”, and traversal continues with the next actor in the sequence.

During simulation, profiling information for each actor is obtained to extract relevant performance characterizations that will be employed to inform the implementation phase. For example, execution time statistics for each actor mode are extracted. Such profiling can help to identify modes that are bottlenecks of individual actors, and actors that are bottlenecks of the overall system.

### 6.3.3 Step 3: System Optimization

In this step, we enter the implementation phase, which is where TDIF comes into play. There are two main kinds of optimization techniques supported in the TDIF framework. One is *cross-platform implementation* for actor-level optimization, and the other is *scheduling and mapping* for system or subsystem optimization.

After we identify the bottleneck actors, cross-platform implementation of actors allows designers to efficiently experiment with alternative actor realizations on different kinds of platforms, such as GPUs, multicore PDSPs, and FPGAs, to help derive a platfor-

m or mix of platforms that will be strategic in terms of the given design constraints (e.g., constraints involving cost, performance, and energy consumption). During this process, much of the code from the simulation phase can be reused. Only the functionality associated with selected actor modes (e.g., bottleneck modes of bottleneck actors) needs to be rewritten or selected from available platform-specific libraries.

The TDIF environment currently supports C-like programming languages — i.e., languages that are targeted to CPU, GPU and multicore PDSP platforms. The GPU-based capabilities of TDIF are currently oriented towards NVIDIA GPUs, based on the CUDA programming language [1], which can be viewed as an extension of C. The multicore PDSP capabilities currently in TDIF are oriented towards Texas Instruments (TI) PDSP devices, and are interoperable with the multithreading libraries provided by TI [76].

TDIF also provides a library of FIFO implementations that are optimized for different platforms. These FIFOs all adhere to standard operations defined in LWDF so that they can be integrated in a manner that is consistent with the CFDF graph model from the simulation phase. After simulation-mode FIFOs are mapped into platform-specific FIFOs, optimized actors can communicate in a manner that is efficient, and consistent with the designer’s simulation model.

The scheduling strategy employed determines the execution order of the actors while the mapping process, which is typically coupled closely with scheduling, determines which resource each actor is executed on. TDIF provides the *generalized schedule tree (GST)* [36] representation to facilitate implementation of and experimentation with alternative scheduling and mapping schemes for system optimization. GSTs are ordered trees with leaf nodes and internal nodes. An internal node of a GST in TDIF represents

iteration control (e.g., a loop count) for an iteration construct that is to be applied when executing the associated subtree. On the other hand, a GST leaf node includes two pieces of information that are used to carry out individual actor firings — one is an actor of the associated dataflow graph, and the other is mapping information associated with the actor. The GST representation provides designers with a common interface through which topological information and algorithms for ordered trees can be applied to access and manipulate schedule elements.

Execution of a GST involves traversing the tree to iteratively enable (and then execute, if appropriate) actors that correspond to the schedule tree leaf nodes. Note that if actors are not enabled, the GST traversal simply skips their invocation. Subsequent schedule rounds (and thus subsequent traversals of the schedule tree) will generally revisit actors that were unable to execute in the current round.

For schedule construction in the implementation phase, the *CFDF graph decomposition* approach of [54] is integrated in the TDIF framework. This approach allows designers to decompose a CFDF graph into a set of SDF subgraphs. Each SDF subgraph can be scheduled by existing static scheduling algorithms, such as an APGAN-based scheduler [5]. The GST schedule trees that result from scheduling these SDF subgraphs are then systematically combined into a single, “execution-rate-balanced” GST using profiling and instrumentation techniques that are discussed in next section.

### 6.3.4 Step 4: System Verification and Instrumentation

Given a GST together with a set of actor and FIFO implementations, the TDIF framework automatically generates a complete CFDF graph implementation for the target platform. Performance metrics, such as latency and throughput, can then be evaluated using platform-specific simulation tools, or using execution on actual target hardware.

The TDIF framework also provides an instrumentation approach to help assess generated implementations. Such instrumentation facilitates experimentation with and tuning of alternative scheduling and mapping techniques. The instrumentation approach integrated in TDIF also facilitates trade-off assessment across different implementation metrics, and helps designers steer implementations towards effective solutions.

Our approach to instrumentation in TDIF is designed to support the following features: (a) no change in functionality (instrumentation directives should not change application functionality); (b) operations for adding and removing instrumentation points should be performed by designers in a way that is external to actors (i.e., does not interfere with or require modification of actor code); and (c) instrumentation operations should be *modular* so that they can be mixed, matched, and migrated with ease and flexibility.

In *schedule tuning mode*, TDIF allows designers to augment the GST representation with functional modules, encapsulated as *instrumentation nodes*, which are dedicated to instrumentation tasks. Like iteration nodes, instrumentation nodes are incorporated as internal nodes. We refer to GSTs that are augmented with instrumentation nodes as *instrumented GSTs (IGSTs)*. The instrumentation tasks associated with an instrumentation node are in general applied to the corresponding IGST sub-tree.

An IGST allows software synthesis for a schedule together with instrumentation functionality that is integrated in a precise and flexible format throughout the schedule. Upon execution, software that is synthesized from an IGST produces profiling data (e.g., related to memory usage, performance or power consumption) along with the output data that is generated by the source application. Modeling techniques, metrics, and measurements related to the coding efficiency of TDIF-based implementations are discussed in [83].

### 6.3.5 Determining Buffer Sizes

During simulation, estimated buffer size bounds are provided by the designer. If these bounds are not sufficient to keep the graph running without deadlock, then simulation is terminated with an appropriate diagnostic message. The designer can then increase selected buffer sizes, and retry the simulation. This process is repeated until the simulation completes for the desired number of iterations, or the buffer size constraints of the design are exhausted. In the latter case, the designer needs to re-examine the system for sample-rate inconsistencies (unbounded buffer sizes), and re-design the system to reduce buffer sizes. Such an iterative process is needed for general CFDF specifications, which are highly expressive, and therefore do not provide the kinds of guaranteed buffer size bounds that are available with less expressive models, such as synchronous dataflow (SDF) or cyclo-static dataflow (CSDF) [41, 6].

During the implementation phase, subgraphs corresponding to specialized models, such as SDF or CSDF, can be extracted from the CFDF specification and optimized

with appropriate guaranteed-buffer-bound algorithms. In particular, we apply the AP-GAN scheduling technique to optimize buffer sizes for SDF subgraphs [5]. Furthermore, quasi-static scheduling techniques, such as the CFDF *mode grouping* technique, can be applied to provide buffer optimization for CFDF specifications that contain certain kinds of dynamic dataflow structures (e.g., see [55]).

### 6.3.6 Discussion

Integration of the following four features distinguish the CFDF-based design flow developed in this chapter:

1. Support for dynamic dataflow behavior in the system;
2. Use of a common formal model for simulation and implementation, which provides rapid transition between the simulation and implementation phases, and promotes consistency between simulation and implementation versions of a design;
3. Support for diverse target platforms;
4. Support for significant code reuse across the simulation and implementation phases.

One limitation of TDIF is that due to the high expressive power of the underlying CFDF model of computation, key analysis and verification problems are undecidable for general TDIF-based applications (e.g., see [58]). However, the CFDF model helps to expose and exploit subsystems within an overall dataflow graph specification that adhere to more specialized (static) dataflow modeling techniques, so that decidable verification properties can be exploited at a “local” level on those subsystems. Some work has also

been developed on quasi-static scheduling of CFDF specifications, where fully-verified schedules for static dataflow subsystems are integrated systematically with global dynamic schedulers [55]. Efficient and reliable quasi-static and dynamic scheduling of CFDF specifications are useful directions for further investigation.

## 6.4 Case Study 1 - CPU/GPU

To demonstrate our design flow, we first experiment with an image processing application centered on Gaussian filtering. Two-dimensional Gaussian filtering is a common kernel in image processing that is used for preprocessing. Gaussian filtering can be used to denoise an image or to prepare for multiresolution processing. A Gaussian filter is a filter whose impulse response is a Gaussian curve, which in two dimensions resembles a bell.

For filtering in digital systems, the continuous Gaussian filter is sampled in a window and stored as coefficients in a matrix. The filter is convolved with the input image by centering the matrix on each pixel, multiplying the value of each entry in the matrix with the appropriate pixel, and then summing the results to produce the value of the new pixel. This operation is repeated until the entire output image has been created.

The size of the matrix and the width of the filter may be customized according to the application. A wide filter will remove noise more aggressively but will smoothen sharp features. A narrow filter will have less of an impact on the quality of the image, but will be correspondingly less effective against noise.

It should also be noted that the tiles indicated in Figure 6.1 do vary somewhat

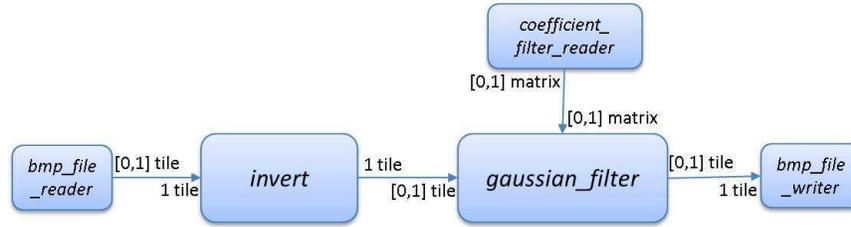


Figure 6.1: Dataflow graph of an image processing application for Gaussian filtering.

between edges. Gaussian filtering applied to tiles must consider a limited neighborhood around each tile (called a *halo*) for correct results. Therefore, tiles produced by `bmp_file_reader` overlap, while the halo is discarded after Gaussian filtering. As a result, non-overlapping tiles form the input to `bmp_file_writer`.

Figure 6.1 shows a simple application based on Gaussian filtering. It reads bitmap files in tile chunks, inverts the values of the pixels of each tile, runs Gaussian filtering on each inverted tile, and then writes the results to an output bitmap file. The main processing pipeline is single-rate in terms of tiles, and can be statically scheduled, but after initialization and end-of-file behavior is modeled, there is conditional dataflow behavior in the application graph, which is represented by square brackets in the figure.

Such conditional behavior arises, first, because the Gaussian filter coefficients are programmable to allow for different standard deviations. The coefficients are set once per image — `coefficient_filter_reader` produces a coefficient matrix for only the first firing. To correspond to this behavior, the `gaussian_filter` actor consumes the coefficient matrix only once, and each subsequent firing processes tiles. Such conditional firing also applies to `bmp_file_reader`, which produces tiles until the end of the associated file is reached.

As shown in Figure 6.1, our dataflow graph of the image processing application for Gaussian filtering is specified as a CFDF graph. The graph includes five actors: `bmp_file_reader`, `coefficient_filter_reader`, `invert`, `gaussian_filter`, and `bmp_file_writer`. We first use the LWDF programming methodology, integrated with the C language, to construct the system for simulation.

### 6.4.1 Simulation

In our design, the `bmp_file_reader` actor is specified using two CFDF modes, and one output FIFO. The two modes are the `process` mode and the `inactive` mode. It is the actor programmer's responsibility to implement the functionality of each mode. In the `process` mode, the `bmp_file_reader` reads image pixels of a given tile and the corresponding header information from a given bitmap file, and produces them to its output FIFOs. Then the actor returns the `process` mode as the mode for its next firing. This continues for each firing until all of the data has been read from the given bitmap file. After that, the actor returns the `inactive` mode, which is a *terminal mode*. Arrival at a terminal mode indicates that the actor cannot be fired anymore until its current mode is first reset externally (e.g., by the enclosing scheduler).

The `coefficient_filter_reader` actor is also specified in terms of two modes and one output FIFO. The two modes are again labeled as the `process` mode and the `inactive` mode, and again, the `inactive` mode is a terminal mode. On each firing when it is not in the `inactive` mode, the `coefficient_filter_reader` actor reads filter coefficients from a given file, stores them into a *filter coefficient vector*

(FCV) array, and produces the coefficients onto its output FIFO. The FCV  $V$  has the form

$$V = (\text{sizeX}, \text{sizeY}, c_0, c_1, \dots, c_{n-1}), \quad (6.4)$$

where `sizeX` and `sizeY` denote the size of the FCV represented in two dimensional format; each  $c_i$  represents a coefficient value; and  $n = \text{sizeX} \times \text{sizeY}$ . After firing, the actor returns the `process` mode if there is data remaining in the input file; otherwise, the actor returns the `inactive` mode.

The `bmp_file_writer` actor contains only a single mode and one input FIFO. The single mode is called the `process` mode. Thus, the actor behaves as an SDF actor. On each firing, the `bmp_file_writer` actor reads the processed image pixels of the given tile and the corresponding header information from its input FIFOs, and writes them to a bitmap file, which can later be used to display the processed results. The actor returns the `process` mode as the next mode for firing.

The `gaussian_filter` actor contains one input FIFO, one output FIFO and two modes: the `store coefficients (STC)` mode and the `process` mode. On each firing in the `STC` mode, the `gaussian_filter` actor consumes filter coefficients from its coefficient input FIFO, caches them inside the actor for further reference, and then returns the `process` mode as the next mode for firing. In the `process` mode, image pixels of a single tile will be consumed from the tile input FIFO of the actor, and the cached filter coefficients will be applied to these pixels. The results will be produced onto the tile output FIFO. The actor then returns the `process` mode as the next mode for firing. To activate a new set of coefficients, the actor must first be reset, through external control,

back to the `STC` mode.

The `invert` actor also contains a single mode called the `process` mode, and contains one input FIFO and one output FIFO. Because it has only one mode, it can also be viewed as an SDF actor. On each firing, the `invert` actor reads the image pixels of the given tile from its input FIFOs, inverts the color of the image pixels, and writes the processed result to its output FIFO. The actor always returns the `process` mode as the next mode for firing.

After designing the actors, as described above, we connect the actors with the appropriate FIFOs. For our simulation setup, we use 256x256 images decomposed into 128x128 tiles, and filtered with different sizes of matrices for Gaussian filter coefficients. The canonical scheduler (see Section 6.3.2) is used to run the simulation on 3GHz Intel Xeon processors. The profiling results are reported in Table 6.1. As can be observed from this table, increases in the matrix size lead to increases in the processing time for the Gaussian filter and the overall application. Furthermore, the Gaussian filter actor accounts for most of the processing time in the application in all cases. Thus, if we can optimize the Gaussian filter actor, the performance of the overall application will be enhanced.

## 6.4.2 Implementation

From the experiments discussed in the previous section, we identified the bottleneck actor to be the Gaussian filtering actor. To improve the performance of this actor, we apply the cross-platform implementation features of TDIF. In particular, we use TDIF to experiment with a new version of the implementation in which the Gaussian filtering

Table 6.1: Execution time of the `gaussian_filter` (GF) actor and the Gaussian filtering application (App) during simulation.

Filter size	5X5	11X11	21X21	25X25	37X37
GF. SIM (ms)	50	280	1080	1540	3310
App. SIM (ms)	70	295	1100	1550	3340
Percentage	<b>71.4%</b>	<b>95%</b>	<b>98.2%</b>	<b>99.3%</b>	<b>99.1%</b>

actor is executed on a graphics processing unit (GPU).

GPUs provide a class of high performance computing platforms that provide high peak throughput processing for certain kinds of regularly structured computations [52]. Typically, a GPU architecture is structured as an array of hierarchically connected cores as shown in Figure 6.2. Cores tend to be lightweight as the GPU will instantiate many of them to support massively parallel graphics computations. Some of the memories are small and scoped for access to small numbers of cores, but can be read or written in one or just a few cycles. Other memories are larger and accessible by more cores, but at the cost of longer read and write latencies.

Using TDIF, we explore the use of GPUs to accelerate the `gaussian_filter` actor. We employ an NVIDIA GTX 285 GPU and employ the CUDA programming environment to specify the internal functionality of the `gaussian_filter` actor for GPU acceleration. This CUDA based actor implementation is integrated systematically into the overall application-level CFDF graph through the TDIF design environment. We

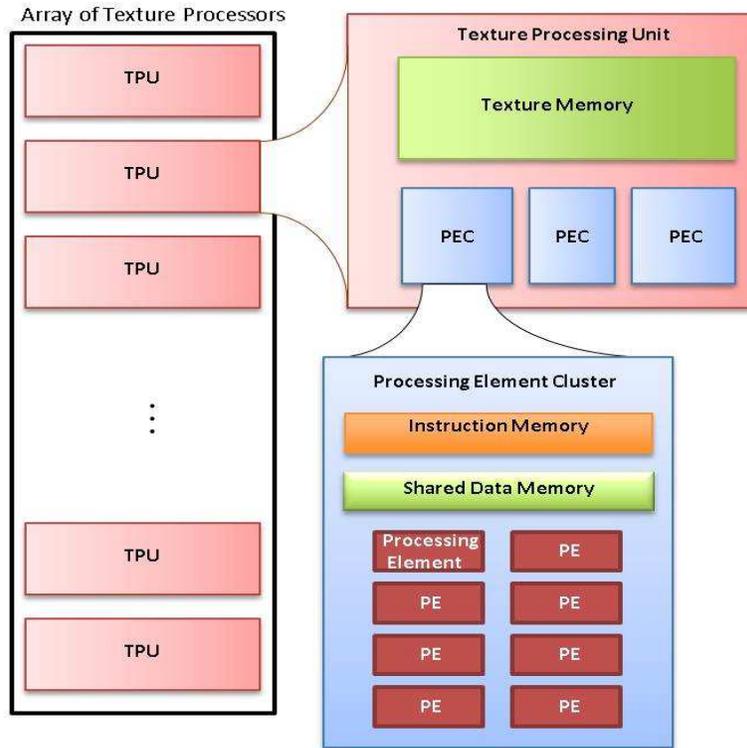


Figure 6.2: A typical GPU architecture.

apply actor-level vectorization to exploit data parallelism within the actor on the targeted GPU.

Fundamentals of vectorized execution for dataflow actors have been developed by Ritz [64], and explored further by Zivojnovic [96], Lalgudi [39], and Ko [35]. In such vectorization, multiple firings of the same actor are grouped together for execution to reduce the rate of context switching, enhance locality, and improve processor pipeline utilization. On GPUs, groups of vectorized firings can be executed concurrently to achieve parallel processing across different invocations of the associated actor. Each instance of a “vectorized actor” may be mapped to an individual thread or process, allowing the replicated instances to be executed in parallel.

An application developer may consider vectorization within and across actors while writing kernels for CUDA acceleration. In TDIF, the actor interface need not change as the vectorization degree changes, which makes it relatively easy for designers to start with the programming framework provided by CUDA and wrap the resulting vectorized kernel designs in individual modes of an actor for integration at the dataflow graph level.

In the GPU-targeted version of our Gaussian filtering application, a CUDA kernel is developed to accelerate the core Gaussian filtering computation (the `process` mode), and each thread is assigned to a single pixel, which leads to a set of parallel independent tasks. The threads are assembled into blocks to maximize data reuse. Each thread uses the same matrix for application to the local neighborhood, and there is significant overlap in the neighborhoods of the nearby pixels. To this end, the threads are grouped by tiles in the image. Once the kernel is launched, threads in a block cooperate to load the matrix, the tile to be processed, and a surrounding neighborhood of points. The image load itself is vectorized to ensure efficient bursting from memory. Because CUDA recognizes the contiguous accesses across threads, the subsequent image processing operations induce vectorized accesses to global memory.

We use the same canonical scheduler in the GPU implementation that we used in in the simulation phase. The performance of our Gaussian filtering application in simulation and GPU-accelerated implementation is compared to demonstrate the ability of our design flow to support cross-platform actor implementation exploration in a manner that is systematically coupled with the simulation-level application model. We use the same experimental setup — in terms of input and output images and overall dataflow graph structure — as used in the simulation. To accelerate the Gaussian Filtering actor,

Table 6.2: Execution time of the `gaussian_filter` (GF) actor and the Gaussian filtering application (App) in simulation and GPU-accelerated implementation.

Filter size	5X5	11X11	21X21	25X25	37X37
GF. SIM (ms)	50	280	1080	1540	3310
GF. GPU (ms)	4.228	4.874	10.257	12.759	21.72
<b>GF. Speedup</b>	<b>11.83</b>	<b>57.45</b>	<b>105.29</b>	<b>120.70</b>	<b>152.39</b>
App. SIM (ms)	70	295	1100	1550	3340
App. GPU (ms)	70	80	140	115	130
<b>App. Speedup</b>	<b>1</b>	<b>3.69</b>	<b>7.86</b>	<b>13.48</b>	<b>25.69</b>

we applied an NVIDIA GTX 285 running CUDA 3.1 and compared the associated implementation to the simulation system. The measurement results are reported in Table 6.2.

As shown in Table 6.2, our design flow provides flexible and efficient transition from the simulation system to a GPU-accelerated implementation that has superior performance compared to the corresponding simulation design for these experiments. The actor-level speedup realized by this acceleration process is in the range of  $10X$  to  $100X$ . However, the application-level speedup levels, while still significant (up to  $25X$  speedup), are consistently less than the corresponding actor-level speedup levels. This is due to factors such as context switch overhead and communication cost for memory movement, which are associated with overall schedule coordination in the application implementations.

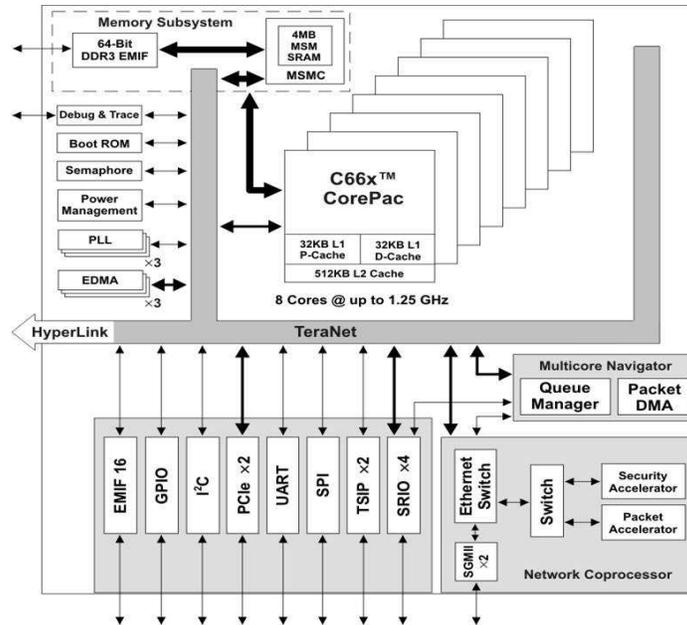


Figure 6.3: Block diagram of TI TMS320C6678 8-core PDSP device.

## 6.5 Case Study 2 - Multicore PDSP

A programmable digital signal processor (PDSP) is a specialized kind of microprocessor with an architecture optimized for the operational needs of real-time digital signal processing [40]. In multicore PDSPs, such as those available in the Texas Instruments TMS320C6678 family of fixed point and floating point processors [76], multiple PDSP cores are integrated on a single chip and connected with shared memory. A typical multicore PDSP is shown in Figure 6.3, where each core has private L1 cache as well as L2 cache. All cores share on-chip SRAM and DRAM. High peak throughput can be achieved if all PDSP cores can operate parallel.

Multicore PDSPs are increasingly employed in wireless communication systems, such as systems for software defined radio (SDR). Fig. 6.4 shows a CFDF graph model for the mp-sched benchmark [91], which is representative of an important class of digital

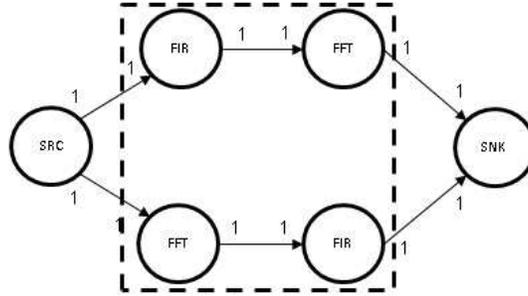


Figure 6.4: An illustration of the mp-sched benchmark.

processing subsystems for wireless communication, and is designed for use with the GNU Radio environment [7]. There are two paths between *SRC* and *SNK*, which represent two different signal processing procedures on the incoming signals. In the upper path, from *SRC* to *SNK*, the signal is first filtered in the time-domain and then transformed to the frequency-domain. In the lower path, from *SRC* to *SNK*, the signal is first transformed to the frequency-domain and then filtered in the frequency-domain. We illustrate the utility of our design flow to conduct simulation and implementation of the mp-sched benchmark on a TI TMS320C6678 8-core PDSP device.

### 6.5.1 Simulation

The mp-sched application involves mainly two actors: finite impulse response (FIR) filtering and fast Fourier transform (FFT) computation, which are fundamental operations in SDR applications.

Our FIR filter actor is specified using a single input FIFO, a single output FIFO, and one mode, the `process` mode. In this mode, the core operation of an FIR filter is developed in terms of the C language. The functionality is developed from the following

equation, which defines the output sequence  $y[n]$  in terms of the input sequence  $x[n]$ :

$$\begin{aligned} y[n] &= b_0x[n] + b_1x[n-1] + \dots + b_Nx[n-M+1] \\ &= \sum_{i=0}^{M-1} b_ix[n-i], \end{aligned} \quad (6.5)$$

where  $x[n]$  is the input signal,  $y[n]$  is the output signal, the  $b_i$ s are the filter coefficients, and  $M$  is the filter order, which is set to 79 in our design.

An FFT is an efficient algorithm to compute the discrete Fourier transform (DFT) and its inverse. FFTs are of great importance in a wide variety of applications, from digital signal processing and solving partial differential equations to algorithms for fast multiplication of large integers. Let  $x_0, x_1, \dots, x_{N-1}$  be complex numbers. The DFT is defined by the following formula:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k \frac{n}{N}}. \quad (6.6)$$

There are  $N$  outputs  $\{X_k\}$ . Each output requires a sum of  $N$  items, which leads to  $O(N^2)$  operations overall.

The most common form of FFT is the Cooley-Tukey algorithm, which calculates the DFT and its inverse using  $O(N \log N)$  operations. This is a divide and conquer algorithm that recursively breaks down a DFT of any composite size  $N = N_1 \times N_2$  into smaller DFTs of sizes  $N_1$  and  $N_2$ , along with  $O(N)$  multiplications by complex roots of unity. A radix-2 decimation-in-time (DIT) FFT is the simplest and most common form of the Cooley-Tukey algorithm. A Radix-2 DIT FFT computes the DFTs of the even-indexed inputs  $x_{2m}(x_0, x_2, \dots, x_{N-2})$  and of the odd-indexed inputs  $x_{2m+1}(x_1, x_3, \dots, x_{N-1})$ , and then

combines those two results to produce the DFT of the whole sequence. Then the same procedure is performed recursively to reduce the overall running time to  $O(N \log N)$ . The recursive tree of the Radix-2 DIT FFT is illustrated in Fig. 6.6(a). Each white node in the figure represents a DFT computation involving the number of points annotated next to the node, and each black node merges two smaller DFT results together.

The FFT actor connects to one output FIFO and one input FIFO, and has one mode, the `process` mode. In this mode, we carry out the steps of the Radix-2 DIT FFT algorithm.

On each firing, the *SRC* node generates 1024 samples to each of its two output FIFOs. These blocks of 1024 samples are packaged into single tokens — i.e., each token encapsulates a complete block of 1024 samples. Such “blocking” of the data does not affect overall system input/output functionality, but it influences the internal dataflow structure of the design model and the associated analysis and implementation steps. The canonical scheduler is used to simulate the dataflow graph on a single core of 8-core PDSP. The processing time for the core computation (as shown in the dashed rectangle in Fig. 6.4) is 98.8% of the computation time for the overall system. The overall computation time is 24.6ms. The FFT computation takes 8.6ms, while the FIR filter takes 3.55ms. To improve the overall system performance, we need to optimize the core computation.

## 6.5.2 Implementation

In this section, we demonstrate, through the core computation of `mp-sched`, the use of both cross-platform implementation and scheduling/mapping in our proposed LWDF-

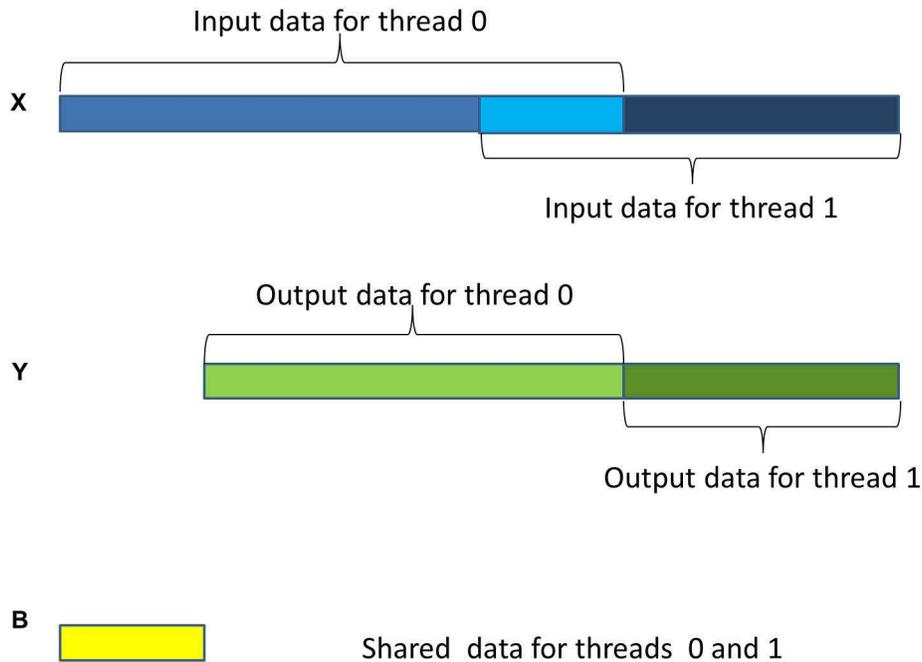


Figure 6.5: Multithreaded FIR filter for PDSP implementation.

and TDIF-based design flow.

### 6.5.2.1 Cross-Platform Implementation

To optimize the system, we first port the FIR filter and FFT actors to multiple PDSP cores by employing the multithreading libraries provided by TI. These libraries are provided with the TI PDSP platform to help DSP system designers express and exploit parallelism for efficient execution on the PDSP cores.

The FIR filtering operation provides a significant amount of data parallelism (DP). For demonstration purposes, this “intra-actor” DP is exploited across two cores on the targeted multicore PDSP using multithreading techniques. Our implementation can be readily adapted to utilize additional cores if desired.

Operation of our TDIF based FIR filter design is shown in Fig. 6.5, where  $X$  is single input token containing  $N$  input samples,  $Y$  is the output token, and  $B$  is the coefficient vector. In order to exploit DP, blocks of  $N$  input samples are divided into two groups, each for execution across two threads. Blocks of output samples are similarly divided into two groups each. The common coefficient vector  $B$  is shared by both threads, and each thread executes the `fir` calculation independently. This provides a multi-threaded FIR filter implementation, which exploits intra-actor parallelism in a manner that can be integrated systematically, through TDIF, with higher level parallelism exposed by the application dataflow graph. The overlap on the inputs of thread 0 and thread 1 arises because the current output depends on the previous  $(M - 1)$  input samples.

To demonstrate the performance gain from multithreading, the execution time of a sequential 79th order FIR filter (Seq-FIR) design in simulation is compared to that of our parallel FIR (Par-FIR) implementation using identical input streams on the targeted multicore PDSP platform. The results for multiple input sizes are demonstrated in Table 6.3. The input size is the number of signal samples. The reported execution time is the processing time, excluding the time for reading from the input FIFO and writing to the output FIFO. The FIFO reading and writing operations involve only pointer manipulations and no actual data movement, and thus have negligible impact on actor performance. The speedup is defined as the ratio between the execution time of Par-FIR and that of Seq-FIR. The super-linear speedup observed is due to the VLIW feature of the targeted kind of PDSP core. As more data is processed, more ILP is available to be exploited within each core.

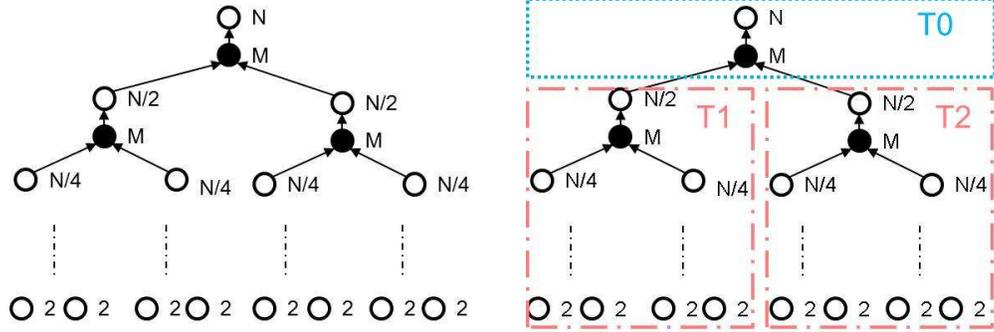
DP and temporal parallelism (TP) from the recursive tree of Fig. 6.6(a) are utilized

Table 6.3: Execution time comparison between sequential FIR in simulation and parallel FIR implementations for different input sizes.

Input Size	1079	10079	100079	1000079
Seq-FIR (s)	0.0036	0.0336	0.334	3.34
Par-FIR (s)	0.0017	0.015	0.147	1.47
Speedup	2.11	2.24	2.27	2.27

for multithreading in the FFT actor, as illustrated in Fig. 6.6(b). With DP, two threads  $T1$  and  $T2$  each calculate half of the overall  $DFT(N)$ , which is  $DFT(N/2)$  using the Cooley-Turkey algorithm. Another single thread  $T0$  merges the two results together. We experiment with two different implementations,  $I_a$  and  $I_b$ . In Implementation  $I_a$ ,  $T1$  and  $T2$  are assigned to two separate cores, and  $T0$  is assigned to one of these two cores. In this implementation, only DP is exploited. On the other hand, in Implementation  $I_b$ ,  $T1$  and  $T2$  are assigned to two separate cores, as in  $I_a$ , but  $T0$  is assigned to another (third) core. In this way,  $T1$  and  $T2$  can execute concurrently in a software pipelined manner with  $T0$  in a separate “pipeline stage”. Implementation  $I_b$  thus exploits both DP and TP.

The execution time of the sequential FFT (Seq-FFT) in simulation is compared to the two different parallel FFT implementations,  $I_a$  and  $I_b$ . The execution time of  $I_a$  is the time for its longest pipeline stage. The speedup is defined by the ratio between the execution time of the parallel implementation ( $I_a$  or  $I_b$ ) and that of Seq-FFT. The latency is another important figure of merit. The latency in these experiments is defined



(a) Recursive tree for Radix-2 DIT FFT.

(b) Thread construction on recursive tree.

Figure 6.6: Parallel FFT actor implementation using TDIF.

by the time difference  $(t_w - t_r)$ , where  $t_r$  denotes the time when the FFT actor reads the first token from its input FIFO, and  $t_w$  denotes the time when the actor writes the first result token to its output FIFO. The results are shown in Table 6.4. For Seq-FFT and  $I_a$ , the latency is the same as the execution time, so the latency values are not shown separately. We see from the results that  $I_b$  achieves more speedup than  $I_a$ , but introduces more latency.

### 6.5.2.2 Scheduling and Mapping

Using the GST representation for dataflow graph schedules, we have derived, by hand, three different scheduling and mapping schemes for the core computation of the mp-sched benchmark, and we have implemented these different schemes using the TDIF scheduling APIs. Due to the use of a common underlying CFDF-based modeling foundation, the actor design from LWDF can be scheduled with no change in the TDIF scheduling APIs. Our experiments with these different scheduling and mapping schemes demonstrate the utility of our design flow for assessing design trade-offs based on alter-

Table 6.4: Execution time and latency comparisons between Seq-FFT in simulation, and the two parallel FFT implementations  $I_a$  and  $I_b$ .

Input Size	64	256	1024	4096
Seq-FFT (s)	0.00028	0.0016	0.0086	0.045
$I_a$ (s)	0.00021	0.001	0.005	0.255
Speedup	1.3	1.61	1.72	1.788
$I_b$ (s)	0.00023	0.00073	0.0039	0.021
Speedup	1.21	2.19	2.20	2.14
Latency (s)	0.00038	0.00118	0.00517	0.0257

native implementation strategies for a given application dataflow graph.

Note that the two paths in the graph of Figure 6.4 are independent so that control parallelism can be used. Additionally, in each path, the individual actors can be pipelined to exploit more temporal parallelism. In the first scheme, we use a sequential implementation for each actor; however, we distribute different actors across different PDSP cores. In particular, the actors are mapped onto 4 DSP cores —  $FIR1$  is assigned to  $DSP0$ ;  $FFT1$  is assigned to  $DSP1$ ;  $FFT2$  is assigned to  $DSP2$ ; and  $FIR1$  is assigned to  $DSP3$ .

The second and third schemes in our experimentation are derived by replacing sequential actor implementations in the graph with corresponding parallel actor implementations. In the second scheme,  $FIR1$  is assigned to  $DSP0, DSP1$ ;  $FFT1$  is assigned to  $DSP2, DSP3$ ;  $FFT2$  is assigned to  $DSP4, DSP5$ ; and  $FIR1$  is assigned to

*DSP6, DSP7*. On the other hand, in the third schedule, *FIR1* is assigned to *DSP0*; *FFT1* is assigned to *DSP1, DSP2, DSP3*; *FFT2* is assigned to *DSP4, DSP5, DSP6*; and *FIR1* is assigned to *DSP7*. Thus, in the second scheme, intra-actor parallelism is exploited for all actors, while in the third scheme, intra-actor parallelism is exploited only for the FFT actors. The GSTs with the associated actor and FIFO implementations are processed to generate corresponding application implementations for the targeted PDSP platform.

We experiment with three different implementations, corresponding to the three different schemes described above. The implementations are evaluated using the same input stream used in simulation, which contains *1024* samples. The execution time, speedup and latency values for these implementations are compared on the core computation shown in Fig. 6.4. The remaining actors (*SRC* and *SNK*) take only approximately 1.2% of the computation time for sequential execution and thus do not have a significant impact on overall performance. The execution time is taken to be the processing time in the core computation region defined above. If pipelining is used, then the execution time is the time for the longest pipeline stage. The latency is defined as the elapsed time during the first iteration of graph execution between when the first input token enters the region and the time when the first output token leaves the region. The results of the three implementations are compared to the simulation, as shown in Table 6.5.

The results demonstrate the capabilities in our design flow for supporting flexible design exploration in terms of different implementation constraints. For example, Scheme 2 has higher execution time but less latency compared to Scheme 3. If the system has a tight latency constraint, then Scheme 2 should be chosen, whereas Scheme 3 is preferable

Table 6.5: Execution time and latency comparisons among 3 different scheduling and mapping schemes and simulation for the mp-sched benchmark.

Schedule	Execution Time (s)	Speedup	Latency (s)
simulation	0.0243	1	0.0243
1	0.00878	2.77	0.0089
2	0.00517	4.7	0.0052
3	0.0041	5.9	0.0092

if throughput is the most critical constraint. Using our design flow, the designer can experiment with such alternatives to efficiently arrive at an implementation whose trade-offs are well matched to the design requirements.

## 6.6 Summary

In this chapter, we have introduced dataflow-based methods that facilitate integrated software simulation and implementation for digital signal processing (DSP) applications. Specifically, we have demonstrated use of a design flow, based on core functional dataflow (CFDF) graphs and multi-scale scheduling techniques, for simulation and implementation of diverse DSP systems — (1) an image processing application on a CPU/GPU platform, and (2) a software defined radio benchmark on a multicore programmable digital signal processor (PDSP).

Through these case studies on diverse platforms, we show that our design flow

allows a designer to formulate and simulate DSP systems efficiently using lightweight dataflow (LWDF) programming. Then, from the profiling results in the simulation, the designer can identify bottlenecks in the system. To alleviate these bottlenecks, optimization techniques in the targeted dataflow interchange format (TDIF) are applied. While TDIF provides a framework where platform-specific optimizations can be applied and experimented with in a flexible way, our design methodology ensures that such experimentation adheres to the high level application structure defined by the simulation model. Such consistency is maintained as a natural by-product of the common CFDF modeling foundation that supports both LWDF and TDIF. The flexibility with which designers can implement different mapping and scheduling strategies using our design flow, and the efficiency with which such strategies can be integrated into complete implementations further facilitate exploration of optimization trade-offs.

## Chapter 7

### Conclusions and Future Work

With the fast evolution of hardware platforms and applications for signal processing systems, scheduling techniques play a critical role in satisfying performance requirements, and utilizing state-of-the art platforms effectively. In this thesis, we have developed multi-scale scheduling techniques simultaneously considering multiple levels of processing and communication hierarchies. We targeted two types of signal processing systems — 1) distributed signal processing systems, where applications are implemented on distributed processing platforms, and 2) parallel signal processing systems, where applications are implemented on parallel processing platforms.

#### 7.1 Distributed Signal Processing Systems

For distributed signal processing systems, we demonstrated a scheduling framework for event detection applications on wireless sensor networks [61].

In this framework, we systematically decompose the system into two layers: the chip layer and network layer. At the chip layer, a sensor node is considered to have variable false positive and false negative error probabilities due to process variations and environmental noise. We designed a training process to estimate the actual detection accuracy for an individual sensor. At the network layer, based on the information obtained from the lower (chip) layer, our scheduling framework efficiently maps the computational

tasks and communication tasks to the distributed sensor nodes across the network.

Only the spatial variance in sensor node detection accuracy is considered in our scheduling framework. Taking temporal variance into account in this framework is a useful direction for future work. One approach to addressing this problem is to design a special training process for monitoring the history of the transmitted data from each sensor node to the associated fusion center. When the system detects that a sensor node's data is consistently "off the mark", appropriate adjustments can be made.

## 7.2 Parallel Signal Processing Systems

For parallel signal processing systems, we developed scheduling techniques based on physical aspects of the platforms and abstract (high level) aspects of the application models. Both kind of aspects have strong influence on system performance. Through extensive experimentation, we showed that our scheduling techniques can effectively take these aspects into account to improve overall system performance.

### 7.2.1 Scheduling from Physical Aspects

We are the first to consider both process variations and thermal flow in task scheduling for multicore GPPs. We formulated the problem in terms of Partial Differential Equations (PDEs). We developed static and dynamic solutions to solve the PDEs. Based on the solutions, we built a scheduling framework that can statically and dynamically assign threads to cores to maximize the throughput of the threads. The proposed techniques are the first to address the problem with online solutions while considering both process and

temperature variations in multicore processors.

Useful directions that emerge from our work on scheduling from physical aspects include the following.

- Considering process variations on Network On Chip [37] devices, which are of increasing interest in future multicore processors and systems.
- Experimenting with our scheduling framework on real parallel platforms and other task models.

### 7.2.2 Scheduling from Models of Computation

We introduced a new design flow, called TDIF-PPG, that considers multiple levels in the behavioral hierarchy of DSP applications, and is formulated in terms of signal processing oriented dataflow models of computation. The TDIF-PPG is designed with a specific emphasis on integrating graph level parallelism and actor level parallelism in dataflow-based, MPSoC software optimization for DSP applications. Our approach is based on a new model, called the parallel processing group (PPG). Our new design flow applies this model to help designers express parallelism within actor designs, and integrate such intra-actor parallelism efficiently with graph level parallelism. Empirical results show that our TDIF-PPG design flow can provide significant speedup on multicore PDSP platforms.

To systematically exploit intra- and inter-actor parallelism in DSP applications, we formulated a new scheduling problem, called the Parallel Actor Scheduling (PAS) problem, that is targeted to optimized MPSoC implementation of DSP applications modeled

by synchronous dataflow (SDF) graphs. We first focused on an important special case of the PAS problem, called the fully parallelizable PAS (FP-PAS) problem, in which all actors in the given application are parallel actors. We utilized multiple levels in both structural and behavioral contexts to derive efficient solutions for the FP-PAS problem.

We developed a two-phase scheduling framework, called TPDF, for the FP-PAS problem using particle swarm optimization techniques in conjunction with (1) mixed integer programming (MIP), and (2) fast heuristic techniques that optimize FP-PAS solutions for fixed processor count assignments. We used a diverse set of randomly generated SDF graphs and a practical image processing application to demonstrate the effectiveness and utility of the TPDF framework in integrating both intra- and inter-actor parallelism for optimized MPSoC implementation.

In the process of developing signal processing systems, there is a large gap from the simulation phase to the implementation phase. To help migrate designs efficiently across this gap, we introduced a novel dataflow-based work flow. Our work flow is designed specifically to aid in the implementation of optimized systems on parallel signal processing platforms. Our work flow integrates use of a design flow, based on core functional dataflow (CFDF) graphs, and the multi-scale scheduling techniques described above. We have demonstrated this work flow in the design and implementation of diverse signal processing systems, including (1) an image processing application on a CPU/GPU platform, and (2) a software defined radio benchmark on a multicore PDSP.

Interesting directions for future work along these lines include the following.

- Accurate and efficient functional simulation of PPG-based designs for early-stage

DSP system validation.

- Experimentation on other kinds of state-of-the-art digital signal processing platforms.
- Investigation of dynamic and quasi-static scheduling techniques for DSP applications.
- Exploration of scheduling techniques for DSP applications in which performance requirements for the system can change significantly at runtime, such as LTE systems (e.g., see [34]).

## Bibliography

- [1] *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide, Version 1.0*, June 2007.
- [2] Whitepaper: NVIDIA's next generation CUDA computer architecture Fermi. Sep. 2009.
- [3] Anandkumar A., L. Tong, and A. Swami. Optimal node density for detection in energy-constrained random networks. *Signal Processing, IEEE Transactions on*, 56(10):5232–5245, oct. 2008.
- [4] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, editors. *Handbook of Signal Processing Systems*. Springer, second edition, 2013.
- [5] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations. *Journal of Design Automation for Embedded Systems*, 2(1):33–60, January 1997.
- [6] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, February 1996.
- [7] E. Blossom. GNU radio: tools for exploring the radio frequency spectrum. *Linux Journal*, June 2004.
- [8] K.A. Bowman, A.R. Alameldeen, S.T. Srinivasan, and C.B. Wilkerson. Impact of die-to-die and within-die parameter variations on the clock frequency and throughput of multi-core processors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(12):1679–1690, dec. 2009.
- [9] K.A. Bowman, S.G. Duvall, and J.D. Meindl. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *Solid-State Circuits, IEEE Journal of*, 37(2):183–190, feb 2002.
- [10] Vladimir Bychkovskiy, Seapahn Megerian, Deborah Estrin, and Miodrag Potkonjak. A collaborative approach to in-place sensor calibration. In *IPSN'03: Proceedings of the 2nd international conference on Information processing in sensor networks*, pages 301–316, Berlin, Heidelberg, 2003. Springer-Verlag.
- [11] J. Ceng et al. MAPS: An integrated framework for MPSoC application parallelization. In *Proceedings of the Design Automation Conference*, pages 754–759, 2008.
- [12] A.K. Coskun, T.T. Rosing, K.A. Whisnant, and K.C. Gross. Static and dynamic temperature-aware scheduling for multiprocessor socs. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(9):1127–1140, sept. 2008.

- [13] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, 1998.
- [14] Jay L. Devore. Probability and statistics for engineering and the sciences. 1995.
- [15] A. Dogramaci and J. Surkis. Evaluation of heuristic for scheduling independent jobs on parallel identical processors. *Management Science*, 25(12):1208–1216, 1979.
- [16] D.S.Naidu. Optimal control systems. *CRC Press*, 2002.
- [17] J. Du and J. Y.-T. Leung. Complexity of scheduling parallel task systems. *SIAM Journal of Discrete Mathematics*, 2(4):473–487, 1989.
- [18] H. El-Rewini, T. G. Lewis, and H. H. Ali. *Task Scheduling in Parallel and Distributed Systems*. Prentice Hall, 1994.
- [19] M. Frigo and S. G. Johnson. FFTW: an adaptive software architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 1998.
- [20] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open mpi: Goals, concept, and design of a next generation mpi implementation. In Dieter Kranzlmler, Pter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*, pages 97–104. Springer Berlin Heidelberg, 2004.
- [21] K. Giaro, M. Kubale, and P. Obszarski. A graph coloring approach to scheduling of multiprocessor tasks on dedicated machines with availability constraints. *Discrete Applied Mathematics*, 157(17):3625–3630, 2009.
- [22] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Symposium on Architectural Support for Programming Languages and Operating Systems*, 2006.
- [23] M.N. Haggag, M. El-Sharkawy, and G. Fahmy. Efficient fast multiplication-free integer transformation for the 2-D DCT H.265 standard. In *Image Processing (ICIP), 2010 17th IEEE International Conference on*, pages 3769–3772, 2010.
- [24] V. Hanumaiah, R. Rao, S. Vrudhula, and K.S. Chatha. Throughput optimal task allocation under thermal constraints for multi-core processors. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 776 –781, 26-31 2009.
- [25] V. Hanumaiah, S. Vrudhula, and K.S. Chatha. Maximizing performance of thermally constrained multi-core processors by dynamic voltage and frequency control. In *Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, pages 310 –313, 2-5 2009.

- [26] C. Hsu, M. Ko, and S. S. Bhattacharyya. Software synthesis from the dataflow interchange format. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, pages 37–49, Dallas, Texas, September 2005.
- [27] Alexander T. Ihler, John W. Fisher, III, Randolph L. Moses, and Alan S. Willsky. Nonparametric belief propagation for self-calibration in sensor networks. In *IPSN '04: Proceedings of the 3rd international symposium on Information processing in sensor networks*, pages 225–233, New York, NY, USA, 2004. ACM.
- [28] K. Jansen and L. Porkolab. Preemptive parallel task scheduling in  $O(n) + \text{Poly}(m)$  time. In G. Goos, J. Hartmanis, J. Leeuwen, D. T. Lee, and S. Teng, editors, *Algorithms and Computation*, Lecture Notes in Computer Science, pages 398–409. Springer Berlin Heidelberg, 2000.
- [29] A. A. Jerraya, A. Bouchhima, and F. Petrot. Programming models and HW-SW interfaces abstraction for multi-processor SoC. In *Proceedings of the Design Automation Conference*, pages 280–285, 2006.
- [30] P.O. Jskelinen, C.S. de La Lama, P. Huerta, and J.H. Takala. OpenCL-based design methodology for application-specific processors. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 223–230, 2010.
- [31] B. Karp and H. T. Kung. GPSR: Greedy perimeter stateless routing for wireless networks. *Proc. of ACM MOBICOM*, Aug. 2000.
- [32] H. Kasim, V. Marchu, R. Zhang, and S. See. Survey on parallel programming model. In J. Cao, M. Li, M. Wu, and J. Chen, editors, *Network and Parallel Computing*, volume 5245 of *Lecture Notes in Computer Science*, pages 266–275. Springer Berlin Heidelberg, 2008.
- [33] J. Kennedy and R. C. Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 1942–1948, November 1995.
- [34] Farooq Khan. *LTE for 4G Mobile Broadband: Air Interface Technologies and Performance*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.
- [35] M. Ko, C. Shen, and S. S. Bhattacharyya. Memory-constrained block processing for DSP software optimization. *Journal of Signal Processing Systems*, 50(2):163–177, February 2008.
- [36] M. Ko, C. Zissulescu, S. Puthenpurayil, S. S. Bhattacharyya, B. Kienhuis, and E. Deprettere. Parameterized looped schedules for compact representation of execution sequences in DSP hardware and software implementation. *IEEE Transactions on Signal Processing*, 55(6):3126–3138, June 2007.
- [37] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani. A network on chip architecture and design methodology. In *VLSI*,

2002. *Proceedings. IEEE Computer Society Annual Symposium on*, pages 105–112, 2002.
- [38] S. Kwon, Y. Kim, W.-C. Jeun, S. Ha, and Y. Paek. A retargetable parallel-programming framework for MPSoC. *ACM Transactions on Design Automation of Electronic Systems*, 13(3), July 2008.
- [39] K. N. Lalgudi, M. C. Papaefthymiou, and M. Potkonjak. Optimizing computations for effective block-processing. *ACM Transactions on Design Automation of Electronic Systems*, 5(3):604–630, July 2000.
- [40] P. Lapsley, J. Bier, A. Shoham, and E. A. Lee. *DSP Processor Fundamentals*. Berkeley Design Technology, Inc., 1994.
- [41] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.
- [42] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, pages 773–799, May 1995.
- [43] Jungseob Lee and Nam Sung Kim. Optimizing throughput of power- and thermal-constrained multicore processors using DVFS and per-core power-gating. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 47–50, 26-31 2009.
- [44] Weiping Liao, Lei He, and K.M. Lepak. Temperature and supply voltage aware performance and power modeling at microarchitecture level. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(7):1042 – 1053, July 2005.
- [45] X. Luo, M. Dong, and Y. Huang. On distributed fault-tolerant detection in wireless sensor networks. *Computers, IEEE Transactions on*, 55(1):58 – 70, Jan. 2006.
- [46] A. Manaa and C. Chu. Scheduling multiprocessor tasks to minimise the makespan on two dedicated processors. *European Journal of Industrial Engineering*, 4(3):265–279, 2010.
- [47] J.-Y. Mignolet and R. Wuyts. Embedded multiprocessor systems-on-chip programming. *IEEE Software*, 26(3):34–41, 2009.
- [48] Michel Mouly and Marie-Bernadette Pautet. *The GSM System for Mobile Communications*. Telecom Publishing, 1992.
- [49] Girija J. Narlikar and Guy E. Blelloch. Pthreads for dynamic and irregular parallelism. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, pages 1–16, Washington, DC, USA, 1998. IEEE Computer Society.

- [50] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly & Associates, Inc., 1996.
- [51] F. A. Omara and M. M. Arafa. Genetic algorithms for task scheduling problem. *Journal of Parallel and Distributed Computing*, 70(1):13–22, 2010.
- [52] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.
- [53] M. Pelcat, P. Menuet, S. Aridhi, and J.-F. Nezan. Scalable compile-time scheduler for multi-core architectures. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 1552–1555, 2009.
- [54] W. Plishker, N. Sane, and S. S. Bhattacharyya. A generalized scheduling approach for dynamic dataflow applications. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 111–116, Nice, France, April 2009.
- [55] W. Plishker, N. Sane, and S. S. Bhattacharyya. Mode grouping for more effective generalized scheduling of dynamic dataflow applications. In *Proceedings of the Design Automation Conference*, pages 923–926, San Francisco, July 2009.
- [56] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya. Functional DIF for rapid prototyping. In *Proceedings of the International Symposium on Rapid System Prototyping*, pages 17–23, Monterey, California, June 2008.
- [57] W. Plishker, N. Sane, M. Kiemb, and S. S. Bhattacharyya. Heterogeneous design in functional DIF. In *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 157–166, Samos, Greece, July 2008.
- [58] W. Plishker, N. Sane, M. Kiemb, and S. S. Bhattacharyya. Heterogeneous design in functional DIF. In P. Stenström, editor, *Transactions on High-Performance Embedded Architectures and Compilers IV*, volume 6760 of *Lecture Notes in Computer Science*, pages 391–408. Springer Berlin / Heidelberg, 2011.
- [59] W. Plishker, G. Zaki, S. S. Bhattacharyya, C. Clancy, and J. Kuykendall. Applying graphics processor acceleration in a software defined radio prototyping environment. In *Proceedings of the International Symposium on Rapid System Prototyping*, pages 67–73, Karlsruhe, Germany, May 2011.
- [60] H. Vincent Poor. *An introduction to signal detection and estimation (2nd ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1994.
- [61] G.J. Pottie. Wireless sensor networks. In *Information Theory Workshop, 1998*, pages 139–140, 1998.
- [62] R. Rao and S. Vrudhula. Efficient online computation of core speeds to maximize the throughput of thermally constrained multi-core processors. In *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, pages 537–542, 10-13 2008.

- [63] Ravishankar Rao, Sarma Vrudhula, and Chaitali Chakrabarti. Throughput of multi-core processors under thermal constraints. In *ISLPED '07: Proceedings of the 2007 International Symposium on Low Power Electronics and Design*, pages 201–206, New York, NY, USA, 2007. ACM.
- [64] S. Ritz, M. Pankert, and H. Meyr. Optimum vectorization of scalable synchronous dataflow graphs. In *Proceedings of the International Conference on Application Specific Array Processors*, October 1993.
- [65] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.
- [66] H. Schwarz, D. Marpe, and T. Wiegand. Overview of the scalable video coding extension of the h.264/avc standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 17(9):1103–1120, 2007.
- [67] M. Sen, Y. Hemaraj, W. Plishker, R. Shekhar, and S. S. Bhattacharyya. Model-based mapping of reconfigurable image registration on FPGA platforms. *Journal of Real-Time Image Processing*, 2008. 14 pages.
- [68] C. Shen, W. Plishker, and S. S. Bhattacharyya. Dataflow-based design and implementation of image processing applications. In L. Guan, Y. He, and S.-Y. Kung, editors, *Multimedia Image and Video Processing*, pages 609–629. CRC Press, second edition, 2012. Chapter 24.
- [69] C. Shen, W. Plishker, H. Wu, and S. S. Bhattacharyya. A lightweight dataflow approach for design and implementation of SDR systems. In *Proceedings of the Wireless Innovation Conference and Product Exposition*, pages 640–645, Washington DC, USA, November 2010.
- [70] C. Shen, H. Wu, N. Sane, W. Plishker, and S. S. Bhattacharyya. A design tool for efficient mapping of multimedia applications onto heterogeneous platforms. In *Proceedings of the IEEE International Conference on Multimedia and Expo*, Barcelona, Spain, July 2011. 6 pages in online proceedings.
- [71] C. Shen, S. Wu, N. Sane, H. Wu, W. Plishker, and S. S. Bhattacharyya. Design and synthesis for multimedia systems using the targeted dataflow interchange format. *IEEE Transactions on Multimedia*, 14(3):630–640, June 2012.
- [72] Bing Shi, Yufu Zhang, and Ankur Srivastava. Dynamic thermal management for single and multicore processors under soft thermal constraints. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*, pages 165–170, 2010.
- [73] J. Singh, U. Madhow, R. Kumar, S. Suri, and R. Cagley. Tracking multiple targets using binary proximity sensors. *IPSN'07*, pages 529–538, april 2007.

- [74] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, second edition, 2009.
- [75] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proceedings of the Design Automation Conference*, 2007.
- [76] Texas Instruments, Inc. *TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor Data Manual*, February 2012.
- [77] J.W. Tschanz, J.T. Kao, S.G. Narendra, R. Nair, D.A. Antoniadis, A.P. Chandrakasan, and V. De. Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage. *Solid-State Circuits, IEEE Journal of*, 37(11):1396 – 1402, nov 2002.
- [78] Pieter van der Wolf, Erwin de Kock, Tomas Henriksson, Wido Kruijtzter, and Gerben Essink. Design and programming of embedded multiprocessors: an interface-centric approach. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, CODES+ISSS '04*, pages 206–217, New York, NY, USA, 2004. ACM.
- [79] Uzi Vishkin, Shlomit Dascal, Efraim Berkovich, and Joseph Nuzman. Explicit multi-threading (XMT) bridging models for instruction parallelism (extended abstract). In *SPAA '98: Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 140–151, New York, NY, USA, 1998. ACM.
- [80] W. Wang, V. Srinivasan, K-C Chua, and B. Wang. Energy-efficient coverage for target detection in wireless sensor networks. *IPSN'07*, pages 313 –322, April 2007.
- [81] M.-Y. Wu and D.D. Gajski. Hypertool: a programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, 1990.
- [82] S. Wu. Representation and scheduling of scalable dataflow graph topologies. Master's thesis, Department of Electrical and Computer Engineering, University of Maryland, College Park, 2011.
- [83] S. Wu, C. Shen, N. Sane, K. Davis, and S. Bhattacharyya. Parameterized scheduling for signal processing systems using topological patterns. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 1561–1564, Kyoto, Japan, March 2012.
- [84] G. Yang, V. Shukla, and D. Qiao. Analytical study of collaborative information coverage for object detection in sensor networks. *Sensor, Mesh and Ad Hoc Communications and Networks*, pages 144 –152, june 2008.
- [85] Inchoon Yeo, Chih Chun Liu, and Eun Jung Kim. Predictive dynamic thermal management for multicore systems. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 734 –739, 8-13 2008.

- [86] L. Yu, L. Yuan, G. Qu, and A. Ephremides. Energy-driven detection scheme with guaranteed accuracy. *IPSN 2006*, pages 284–291, 2006.
- [87] Lige Yu. and A. Ephremides. Detection performance and energy efficiency trade-off in a sensor network. *Proc. of 2003 Allerton Conference, Alletion, IL*, Oct. 2003.
- [88] Lin Yuan, S. Leventhal, and Gang Qu. Temperature-aware leakage minimization technique for real-time systems. In *Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on*, pages 761–764, 5-9 2006.
- [89] S. Zahedi, M.B. Srivastava, and C. Bisdikian. A computational framework for quality of information analysis for detection-oriented sensor networks. *MILCOM'08*, pages 1–7, nov. 2008.
- [90] G. Zaki. *Scalable Techniques for Scheduling and Mapping DSP Applications on-to Embedded Multiprocessor Platforms*. PhD thesis, Department of Electrical and Computer Engineering, University of Maryland, College Park, 2013.
- [91] G. Zaki, W. Plishker, S. Bhattacharyya, C. Clancy, and J. Kuykendall. Vectorization and mapping of software defined radio applications on heterogeneous multiprocessor platforms. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, pages 31–36, Beirut, Lebanon, October 2011.
- [92] Z. Zhou, C. Shen, W. Plishker, H. Wu, and S. S. Bhattacharyya. Systematic integration of flowgraph- and module-level parallelism in implementation of DSP applications on multiprocessor systems-on-chip. In *Proceedings of the International Conference on Signal Processing*, pages 402–408, Beijing, China, October 2012.
- [93] Zheng Zhou, Junjun Gu, and Gang Qu. Scheduling for multi-core processor under process and temperature variation. In *Embedded Multicore Socs (MCSoc), 2012 IEEE 6th International Symposium on*, pages 113–120, 2012.
- [94] Zheng Zhou and Gang Qu. An energy efficient adaptive event detection scheme for wireless sensor network. In *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on*, pages 235–238, sept. 2011.
- [95] Zheng Zhou, Chung-Ching Shen, William Plishker, and Shuvra S. Bhattacharyya. Dataflow-based, cross-platform design flow for DSP applications. In *Embedded Systems Development — From Functional Models to Implementations*, volume 20, page 212. Springer, 2013.
- [96] V. Zivojnovic, S. Ritz, and H. Meyr. Retiming of DSP programs for optimum vectorization. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 492–496, April 1994.