

## ABSTRACT

Title of dissertation: DESIGN SPACE EXPLORATION FOR  
SIGNAL PROCESSING SYSTEMS USING  
LIGHTWEIGHT DATAFLOW GRAPHS

Lin Li  
Doctor of Philosophy, 2018

Dissertation directed by: Professor Shuvra S. Bhattacharyya  
Dept. of Electrical and Computer Engineering,  
and Institute for Advanced Computer Studies

Digital signal processing (DSP) is widely used in many types of devices, including mobile phones, tablets, personal computers, and numerous forms of embedded systems. Implementation of modern DSP applications is very challenging in part due to the complex design spaces that are involved. These design spaces involve many kinds of configurable parameters associated with the signal processing algorithms that are used, as well as different ways of mapping the algorithms onto the targeted platforms.

In this thesis, we develop new algorithms, software tools and design methodologies to systematically explore the complex design spaces that are involved in design and implementation of signal processing systems. To improve the efficiency of design space exploration, we develop and apply compact system level models, which are carefully formulated to concisely capture key properties of signal processing algorithms, target platforms, and algorithm-platform interactions.

Throughout the thesis, we develop design methodologies and tools for integrating new compact system level models and design space exploration methods with lightweight dataflow (LWDF) techniques for design and implementation of signal processing systems. LWDF is a previously-introduced approach for integrating new forms of design space exploration and system-level optimization into design processes for DSP systems. LWDF provides a compact set of retargetable application programming interfaces (APIs) that facilitates the integration of dataflow-based models and methods. Dataflow provides an important formal foundation for advanced DSP system design, and the flexible support for dataflow in LWDF facilitates experimentation with and application of novel design methods that are founded in dataflow concepts. Our developed methodologies apply LWDF programming to facilitate their application to different types of platforms and their efficient integration with platform-based tools for hardware/software implementation. Additionally, we introduce novel extensions to LWDF to improve its utility for digital hardware design and adaptive signal processing implementation.

To address the aforementioned challenges of design space exploration and system optimization, we present a systematic multiobjective optimization framework for dataflow-based architectures. This framework builds on the methodology of multiobjective evolutionary algorithms and derives key system parameters subject to time-varying and multidimensional constraints on system performance. We demonstrate the framework by applying LWDF techniques to develop a dataflow-based architecture that can be dynamically reconfigured to realize strategic configurations in the underlying parameter space based on changing operational requirements.

Secondly, we apply Markov decision processes (MDPs) for design space exploration in adaptive embedded signal processing systems. We propose a framework, known as the Hierarchical MDP framework for Compact System-level Modeling (HMCSM), which embraces MDPs to enable autonomous adaptation of embedded signal processing under multidimensional constraints and optimization objectives. The framework integrates automated, MDP-based generation of optimal reconfiguration policies, dataflow-based application modeling, and implementation of embedded control software that carries out the generated reconfiguration policies.

Third, we present a new methodology for design and implementation of signal processing systems that are targeted to system-on-chip (SoC) platforms. The methodology is centered on the use of LWDF concepts and methods for applying principles of dataflow design at different layers of abstraction. The development processes integrated in our approach are software implementation, hardware implementation, hardware-software co-design, and optimized application mapping. The proposed methodology facilitates development and integration of signal processing hardware and software modules that involve heterogeneous programming languages and platforms.

Through three case studies involving complex applications, we demonstrate the effectiveness of the proposed contributions for compact system level design and design space exploration: a digital predistortion (DPD) system, a reconfigurable channelizer for wireless communication, and a deep neural network (DNN) for vehicle classification.

DESIGN SPACE EXPLORATION  
FOR SIGNAL PROCESSING SYSTEMS  
USING LIGHTWEIGHT DATAFLOW GRAPHS

by

Lin Li

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  
2018

Advisory Committee:

Professor Shuvra S. Bhattacharyya, Chair/Advisor

Professor Rajeev Barua

Professor Charalampos Papamantou

Professor Marilyn Wolf

Professor Ramani Duraiswami

Dedication

To my family

## Acknowledgments

I would like to thank my advisor, Professor Shuvra S. Bhattacharyya, for his kind support, shepherd, and encouragement during the past few years. His intelligence, patience, and his pursuit of excellence on academics and teaching have inspired me significantly in both my Ph.D. study and my life. It is my great honor to have him as my advisor.

I also want to thank my committee members, Professor Rajeev Barua, Professor Charalampos Papamanthou, Professor Marilyn Wolf, and Professor Ramani Duraiswami for providing insightful and valuable suggestions and comments.

I am grateful to all members of DSPCAD research group and other colleagues and collaborators, especially Professor Marilyn Wolf, Professor Francesca Palumbo, Ms. Tiziana Fanni, and Dr. Timo Viitanen. I would also like to thank Marshall Plan Foundation and Fachhochschule Salzburg for providing a great opportunity for collaborative research.

Finally I give my special thanks to my parents and my boyfriend Chen Chen for their continuous and unconditional love and support.

*This research was sponsored in part by the U.S. National Science Foundation, the Austrian Marshall Plan Foundation, and the Laboratory for Telecommunications Sciences.*

# Table of Contents

List of Figures	vi
1 Introduction	1
1.1 Compact System Level Models . . . . .	2
1.2 Application Areas . . . . .	3
1.3 Outline of Thesis . . . . .	5
2 Constrained Optimization for Digital Predistortion (DPD) Systems	7
2.1 Introduction . . . . .	8
2.2 Design Space Exploration for DPD Systems . . . . .	10
2.3 Experimental Setup and Simulation Results . . . . .	16
2.4 Summary . . . . .	20
3 Evolutionary Optimization for DPD Architectures	22
3.1 Introduction . . . . .	23
3.2 Related Work . . . . .	25
3.3 Adaptive Dataflow-based DPD Architecture . . . . .	27
3.4 Optimization Metrics and Design Space . . . . .	28
3.5 Multiobjective Optimization Using Evolutionary Algorithm . . . . .	33
3.6 Experimental Setup and Simulation Results . . . . .	39
3.7 Summary . . . . .	45
4 Design of Adaptive Signal Processing Systems Using Markov Decision Processes	47
4.1 Introduction . . . . .	48
4.2 Background . . . . .	51
4.3 Hierarchical MDP Approach for Compact System-level Modeling . . . . .	55
4.4 Case Study of Channelizer/Receiver Application . . . . .	60
4.5 Experiments . . . . .	67
4.6 Summary . . . . .	74
5 An Integrated Hardware/Software Design Methodology for Signal Processing Systems	76
5.1 Introduction . . . . .	77
5.2 Related Work . . . . .	80
5.3 Proposed Design Methodology . . . . .	82
5.4 Case Study: A Deep Neural Network for Vehicle Classification . . . . .	112
5.5 Results . . . . .	135
5.6 Conclusions . . . . .	151
6 Conclusions and Future Work	153
6.1 Conclusions . . . . .	153
6.2 Future Work . . . . .	156





## List of Figures

2.1	Predistorter structure for the joint predistortion of PA and I/Q modulator impairments. . . . .	12
2.2	Dataflow graph model of the predistortion filter. . . . .	13
2.3	BAA distribution of the derived Pareto-optimal settings. . . . .	18
2.4	Real-time power and performance comparison involving the OAD system. ACPR and EVM measurements are presented using absolute values — thus, higher values indicate better performance. . . . .	20
3.1	Dataflow graph model of the predistortion filter. . . . .	28
3.2	Multiobjective optimization model for DPD system. . . . .	35
3.3	Pareto-optimized solutions obtained from the EADI Framework and PS for (a) QPSK, (b) 16-QAM, (c) 64-QAM. . . . .	41
3.4	Output spectra of the ideal linear PA, the Wiener PA model without DPD, with DPD under configuration obtained from PS and EF. . . . .	43
3.5	Output constellation of the ideal linear PA, the Wiener PA model without DPD, with DPD under configuration obtained from PS and EF. . . . .	44
4.1	An illustration of the HMCSM framework for design and implementation of adaptive signal processing systems. . . . .	56
4.2	Block diagram of receiver signal processing and two MDP schemes. . . . .	63
4.3	PSDF specification of channelizer subsystem. . . . .	65
4.4	Simulation results for MDP-I. . . . .	71
4.5	Comparison among MDP-generated policies and fixed-configuration designs. . . . .	73
5.1	An illustration of STMCM in the context of DNN system design. . . . .	83
5.2	Switch actor in CFDF. (a) Switch Actor, (b) Dataflow Table, (c) Mode Transition Diagram between CFDF Modes. . . . .	86
5.3	Illustration of an LWDF-V-based actor. . . . .	92
5.4	Example of an AIM FSM for a CFDF actor with three modes. . . . .	95
5.5	Illustration of LWDF-V-based actors communication. . . . .	97
5.6	Example of a AEM with three different firing condition for three possible modes. . . . .	97
5.7	An example of an FSM that controls an ASM. . . . .	100
5.8	Examples of signal waveforms during execution of an LWDF-V actor. . . . .	101
5.9	Synchronous FIFO design . . . . .	102
5.10	Illustration of an LWDF-V-based implementation of a CFDF graph that consists of three actors. . . . .	105
5.11	Asynchronous FIFO design in LIDE-V. . . . .	106
5.12	Clock gating in a LIDE-V actor. . . . .	107
5.13	Signal waveforms in the clock gating module. . . . .	107
5.14	Pseudo-CDC FIFO design in LIDE-V. . . . .	110

5.15	DNN for automatic discrimination of four types of vehicles. . . . .	113
5.16	The code segment that implements loop tiling within the LIDE-C actor for convolution. . . . .	116
5.17	LIDE-V implementation for the accelerated SFM. . . . .	122
5.18	An illustration of the hierarchical actor associated with Design $SFM_h$ . . . . .	127
5.19	Reference configuration for hardware/software co-design exploration in our experiments. . . . .	131
5.20	Performance evaluation of convolution actors with different image dimensions: (a) $48 \times 48$ , (b) $96 \times 96$ , (c) $750 \times 750$ , (d) $1500 \times 1500$ . . . . .	137
5.21	Buffer memory and communication requirements in the DNN architecture. . . . .	139
5.22	Buffer memory allocation for the DNN application. . . . .	139

# Chapter 1

## Introduction

In recent years, increased requirements on functionality and flexibility in digital signal processing (DSP) applications have imposed significant new challenges on the design and implementation of embedded signal processing systems. Many state-of-the-art signal processing applications involve challenging requirements on flexibility and reconfigurability so that the systems can be adaptive to time-varying constraints or changes in the operational environment. Additionally, implementation of these applications is subject to multidimensional design criteria, such as competing objectives or constraints that involve power consumption, system accuracy, hardware cost, and processing speed. Thus, flexible adaptation across diverse system configurations, and efficient design space exploration are key aspects in advanced design methodologies for DSP systems.

Due to the requirements of DSP system design described above, the design space exploration for signal processing systems is both important and challenging. Effective design space exploration in this context includes experimenting with different sets of system configuration parameters, exploring alternative designs with different optimization methods, and assessing trade-offs across the relevant optimization objectives. In this work, we propose software tools, algorithms, and methodologies to efficiently and systematically explore the design space of DSP systems across

multiple levels of abstraction, including algorithm, software, and hardware.

## 1.1 Compact System Level Models

In this thesis, we develop new algorithms, software tools and design methodologies to systematically explore the complex design spaces that are involved in deploying signal processing systems. To improve the efficiency of design space exploration, we develop and apply compact system level models, which are carefully formulated to concisely capture key properties of signal processing algorithms, target platforms, and algorithm-platform interactions.

Throughout the thesis, we develop design methodologies and tools for integrating new compact system level models and design space exploration methods with lightweight dataflow (LWDF) techniques for design and implementation of signal processing systems [1]. LWDF is a previously-introduced approach for integrating new forms of design space exploration and system-level optimization into design processes for DSP systems. LWDF provides a compact set of retargetable application programming interfaces (APIs) that facilitates the integration of dataflow-based models and methods with different hardware platforms and their associated platform-based design tools.

Dataflow provides an important formal foundation for advanced DSP system design [2, 3], and the flexible support for dataflow in LWDF facilitates experimentation with and application of novel design methods that are founded in dataflow concepts [4]. Our developed methodologies apply LWDF programming to facilitate

their application to different types of platforms and their efficient integration with platform-based tools for hardware/software implementation. Additionally, we introduce novel extensions to LWDF to improve its utility for digital hardware design and adaptive signal processing implementation.

## 1.2 Application Areas

We investigate three major application areas of signal processing systems to demonstrate our contributions in design space exploration using compact system level models and lightweight dataflow techniques. These are the areas of digital predistortion (DPD) and channelizer design for wireless communications, and deep neural networks (DNNs) for computer vision.

### 1.2.1 Digital Predistortion

Digital predistortion (DPD) is an application area in wireless communications that we have applied our proposed design space exploration methods to. DPD is a technique to counteract impairments, such as I/Q mismatch, power amplifier (PA) nonlinearities, and local oscillator (LO) leakage, that compromise the linearity of radio transmitter amplifiers [5]. Using our proposed design space exploration techniques, we have developed a novel reconfigurable DPD architecture. The architecture provides flexible configuration across different combinations of polynomial orders, bit-widths, and filter orders that are relevant to key trade-offs in DPD system operation.

In Chapter 2 and Chapter 3, we present a dataflow-based implementation of our DPD architecture as a case study of our design optimization framework. Through this case study, we concretely introduce new methods for multiobjective design optimization on which the framework is based. These methods are applied to efficiently navigate the complex design spaces associated with DPD implementation. Through extensive experiments, we demonstrate the effectiveness of our configurable DPD architecture and design optimization framework in enabling efficient, adaptive DPD system operation.

### 1.2.2 Channelizer Design

In wireless communication, the channelizer is a key part of a receiver that extracts one or more radio channels of distinct bandwidths from a digitized wideband input signal. By adapting the configuration of the channelizer based on the communication scenario, we seek to optimize its energy efficiency while ensuring that it extracts the number of channels that is required by the communication scenario at any given time.

In Chapter 4, we develop a novel adaptive signal processing architecture to achieve these objectives, and we apply lightweight dataflow techniques to implement the architecture efficiently on a state-of-art embedded processing platform.

### 1.2.3 Deep Neural Networks

Deep learning is another application area in which efficient design space exploration is of great importance. Deep learning is a sub-field of machine learning that has attracted a large amount of attention from academia and industry in recent years. The core component of a deep learning system is the deep neural network (DNN), a neural network with hidden layers to optimize an objective function for the targeted learning purpose. Deep learning has been applied in numerous fields, such as speech recognition, computer vision, machine translation, and autonomous driving, with huge success in cases where a sufficient volume of labeled data is available for training (e.g., see [6]).

Presently, there is a growing trend of deploying deep learning algorithms on mobile devices, such as smart phones and tablets, in real-time applications. For example, Facebook has presented a real-time video style transfer on mobile devices using Caffe2Go, a lightweight deep learning framework [7]. The migration of DNNs from high-end machines with huge amounts of computational resources to low-end devices with limited resources is challenging. For example, a DNN with many layers may not fit into the random access memory (RAM) of a mobile device. In Chapter 5, as a case study of our proposed new hardware/software design methodologies, we explore in depth the design space of a DNN application for vehicle classification.

## 1.3 Outline of Thesis

The remainder of this dissertation is organized as follows.

The emphasis of Chapter 2 and Chapter 3 is on design space optimization for DPD systems. In particular, Chapter 2 presents models and methods for efficient search of the complex, multidimensional design spaces associated with DPD systems, and Chapter 3 extends the work in Chapter 2 by proposing a novel evolutionary algorithm framework for multiobjective optimization of DPD systems. In Chapter 3, we demonstrate the proposed framework by applying it to develop an adaptive DPD architecture.

In Chapter 4, we propose the Hierarchical MDP framework for Compact System-level Modeling (HMCSM), which applies Markov decision processes (MDPs) to enable autonomous adaptation of embedded signal processing under multidimensional constraints and optimization objectives. The framework integrates automated, MDP-based generation of optimal reconfiguration policies, dataflow-based application modeling, and implementation of embedded control software that carries out the generated reconfiguration policies. The effectiveness of HMCSM is demonstrated through experiments with an adaptive channelizer for wireless communications.

In Chapter 5, we develop a dataflow-based methodology, along with supporting software tools and libraries, for integrated hardware/software co-design and design optimization of signal processing systems. As outlined above in Section 1.2.3, we develop a DNN implementation for vehicle classification as a demonstration of the proposed design methodology.

We conclude in Chapter 6 with a summary of the thesis followed by a discussion on useful directions for future work.



## Chapter 2

### Constrained Optimization for Digital Predistortion (DPD) Systems

As a starting point of design space exploration for DPD systems, in this chapter, we develop new models and methods for exploring multidimensional design spaces associated with digital predistortion (DPD) systems. DPD systems are important components for power amplifier linearization in wireless communication transceivers. In contrast to conventional DPD implementation methods, which are focused on optimizing a single objective — most commonly, the adjacent channel power ratio (ACPR) — without systematically taking into account other relevant metrics, we consider DPD system implementation in a multiobjective optimization context. In our targeted multiobjective context, trade-offs among power consumption and multiple DPD performance metrics are jointly optimized subject to performance constraints imposed by the given modulation scheme. Through synthesis and simulation results, we demonstrate that DPD systems derived through our design space exploration techniques exhibit significantly improved trade-offs among multidimensional implementation criteria, including energy consumption, ACPR, and symbol error-rate. Additionally, we perform experiments using three different Long-Term Evolution (LTE) modulation schemes, and we demonstrate that our multiobjective optimization approach significantly enhances system adaptivity in response to changes in the employed modulation scheme.

Material in this chapter was published in [8].

## 2.1 Introduction

In wireless communication systems, *power amplifier (PA)* nonlinearities, I/Q mismatch, and leakage in the *local oscillator (LO)* introduce power leakage into adjacent bands, interference between I and Q baseband signals, and *direct current (DC)* offset, respectively. In the frequency domain of the transmitted signal, the effects of these impairments are translated as power leakage into adjacent channels. *Digital predistortion (DPD)* is a widely investigated technique (e.g., see [9, 10, 11, 12, 13]) to counteract such impairments by applying carefully-calculated distortion to the signal prior to transmission. In this chapter, we develop new methods for design optimization of field-programmable gate array (FPGA)–based DPD systems that are novel in their adaptability to different modulation constraints, and their support for multidimensional design space exploration. Specifically, we develop methods to formulate and perform multiobjective optimization across the key DPD operational metrics of *power consumption, adjacent channel power ratio, and error vector magnitude*, which we abbreviate as *PAE*.

Our work builds on the DPD algorithm introduced in [5]. In this chapter, we go beyond the work in [5] through a deep investigation into implementation aspects of this DPD algorithm. In contrast, the study in [5] is primarily at the levels of theoretical analysis and algorithm-level (MATLAB) simulation evaluation.

Unlike earlier DPD architectures (e.g., see [10, 14]), the DPD algorithm pro-

posed in [5] is one of the first DPD techniques that jointly compensates for PA nonlinearities and I/Q modulator impairments. Conventional digital predistorters are constructed using serial configurations. For example, the work in [10] is focused on modeling and compensation of frequency-dependent gain/phase imbalance and DC offset. For more details on this serial digital predistorter structure, we refer the reader to [10]. Instead of using a serial structure, the DPD architecture in [5] employs an extended parallel Hammerstein structure, which decomposes DPD operation into direct and conjugate predistortion subsystems. Such a decomposed structure provides additional degrees of freedom in the predistorter design compared to the serial structures used in traditional DPD systems.

The problem of parameter optimization for DPD systems has been studied extensively in prior work. These works can be distinguished in terms of three key (vector-valued) DPD parameters — the filter orders, polynomial orders, and filter coefficients. For example, Çiflikli and Yapıcı and Sperlich et al. apply genetic algorithms to optimize the filter coefficients while assuming fixed filter and polynomial orders [9, 15]. In contrast, Abdelhafiz et al. jointly optimize all three parameters [16]; however, this optimization is performed with respect to only a single objective — the adjacent channel power ratio (ACPR). Freiberger et al. determine DPD filter coefficients based on constrained multiobjective optimization [17]; however, the work in this chapter also assumes fixed polynomial and filter orders, and is specialized to a specific modulation scheme (64-QAM).

To the best of our knowledge, the work of this chapter is the first to go beyond the aforementioned works by simultaneously (1) considering joint optimization of

all three DPD implementation parameters; (2) optimizing the derived DPD architectures in a multiobjective context (PAE); (3) and demonstrating applicability to multiple modulation schemes, for example, Quadrature Phase Shift Keying (QPSK), 16-Quadrature Amplitude Modulation (QAM), and 64-QAM in LTE) under their respective, modulation-specific constraints.

## 2.2 Design Space Exploration for DPD Systems

Given a multidimensional design evaluation space (MDES), we view a *multiobjective optimization* (MOO) process as a process that searches the associated design space in an effort to derive a set of Pareto-optimal solutions. A solution in an MDES is Pareto-optimal if no solution in the space is superior to it when all objectives are considered. Our design optimization problem for DPD system implementation can be viewed as an MOO problem where the MDES is defined by the PAE metrics defined in Section 2.1. In this section, we provide details on these metrics and the DPD parameters — the filter coefficients, filter orders and polynomial orders — that define the design space that is investigated in our work.

To implement the DPD algorithm in [5], we develop a dataflow representation for part of the overall DPD system. This DPD algorithm operates in two stages. In the *coefficient estimation* stage, the DPD filtering coefficients are estimated. The estimated coefficients are then employed in the *DPD filtering* stage for actual predistortion of the input signal. Since the coefficient estimation stage of this system is performed off-line [5], and our objective is to investigate FPGA implementation

of the on-line (real-time) part, we implement only the second stage — predistortion filtering — in the dataflow graph.

The structure of the predistortion filtering system is shown in Fig. 2.1. The DPD system is split into two branches, namely direct and conjugate predistortions. The output of the predistortion filter can be expressed as

$$z_n = \sum_{p \in I_P} f_{p,n} \star \psi_p(x_n) + \sum_{q \in I_Q} \bar{f}_{q,n} \star \psi_q(x_n^*) + c' , \quad (2.1)$$

where  $\star$  denotes convolution;  $x_n$  and  $x_n^*$  are the direct and conjugate input samples, respectively;  $I_P$  and  $I_Q$  are the employed sets of direct and conjugate term orders, respectively;  $\psi_p$  and  $\psi_q$  are *polynomial basis functions* for the direct and conjugate branches, respectively;  $f_{p,n}$  and  $\bar{f}_{q,n}$  are the Finite-Impulse-Response (FIR) filter coefficients for the direct and conjugate polynomials, respectively; and  $c'$  is the LO leakage compensation component. The maximum polynomial order used can be different for the direct and conjugate branches of the predistorter [5].

Given  $r \in \{p, q\}$ , the polynomial basis function  $\psi_r$  can be expressed as

$$\psi_r(x_n) = \sum_{k \in I_r} u_{k,r} |x_n|^{k-1} x_n, \quad r \in I_R , \quad (2.2)$$

where  $I_R$  denotes the set of term orders employed in the given DPD configuration ( $I_R = I_P$  if  $r = p$ , and  $I_R = I_Q$  if  $r = q$ );  $I_r$  denotes the subset of  $I_R$  that contains only of term orders up to  $r$  in  $I_R$ ; and  $\{u_{k,r}\}$  denotes the polynomial weights. Here, given a polynomial  $\rho = a_0 + a_1x + \dots + a_nx^n$ , we define each monomial  $a_i x^i$  to be a *term* of  $\rho$ , and we define  $i$  to be the associated *term order*. According to [5], only

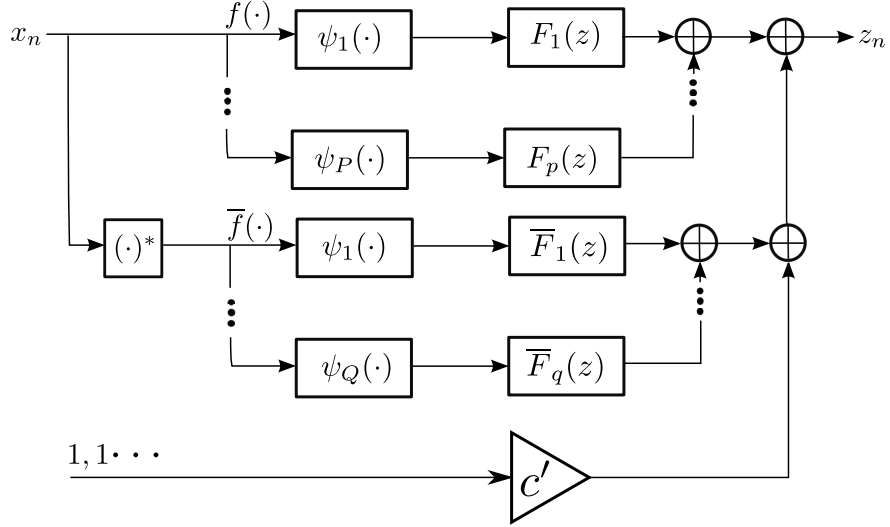


Figure 2.1: Predistorter structure for the joint predistortion of PA and I/Q modulator impairments.

odd-order polynomials are used to avoid the computation of the square-root within  $|x_n|^{k-1}$ , which is a computation-saving option that has been applied in the proposed implementation.

The dataflow model for this second stage is illustrated in Fig. 2.2. The actor labeled *Poly Compt.* in Fig. 2.2 represents a polynomial computation module that produces a polynomial basis function  $\psi_p$ , which is used for both the direct and conjugate branches.

### 2.2.1 Optimization Metrics

The power consumption of the DPD system is an important metric that we carefully take into account in our design optimization approach. We implement the FIR filters in Fig. 2.2 in hardware using the Altera EP2C35F672C6 FPGA from the Cyclone II family, and we estimate the total power consumed by all of the instantiated filters by modeling the power consumption as a function of the DPD

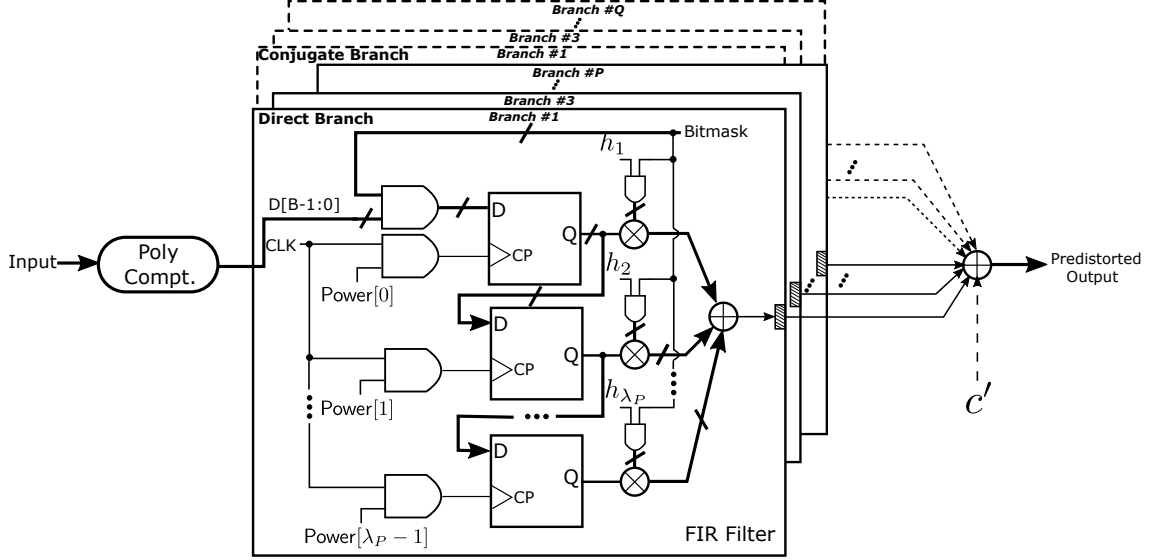


Figure 2.2: Dataflow graph model of the predistortion filter.

parameters. We employ such a model-based estimation approach for assessing power consumption because detailed measurement of total power consumption would be time-consuming when exploring the large multidimensional DPD design space, which is elaborated on in the following section.

Our approach to system-level DPD power estimation starts by first measuring the total power consumption of a single branch under all valid filter order and bit-width values using Altera PowerPlay Analyzer. The power consumption for a specific DPD configuration is then estimated as

$$Power_{est} = \sum_{p \in I_P} Power_p(bw_p, fo_p) + \sum_{q \in I_Q} Power_q(bw_q, fo_q), \quad (2.3)$$

where  $bw_x$  and  $fo_x$  are the bit-width and filter order for branch  $x$ , respectively, and  $Power_x(bw_x, fo_x)$ , the power consumed by branch  $x$  with bit-width  $bw_x$  and filter order  $fo_x$ , is obtained from the aforementioned power measurement process.

During MOO, we are interested in the power *comparison* result of two configurations instead of their actual power consumption levels. This is because, as we explore different pairs of design points during the search process, we are interested in determining which configuration in any given pair is “better” than the other. Thus, we can validate the utility of the above power estimator in our estimation context using the *estimation fidelity*, which is defined by (e.g., see [18]):

$$Fidelity = \frac{2}{M(M-1)} \left( \sum_{i=1}^{M-1} \sum_{j=i+1}^M f_{ij} \right), \quad (2.4)$$

where  $M$  is the number of configurations that we generate to calculate the fidelity. Here,  $f_{ij} = 1$  if  $sign(S_i - S_j) = sign(F_i - F_j)$ , and  $f_{ij} = 0$  otherwise. The terms  $S_i$  and  $S_j$  denote the simulated average power consumption levels of configurations  $i$  and  $j$ , respectively;  $F_i$  and  $F_j$  are the corresponding estimates from the power estimation function  $F$ ; and  $sign(x)$  equals  $-1$  if  $x < 0$ ,  $0$  if  $x = 0$ , and  $1$  if  $x > 0$ .

We generate 100 uniformly distributed system configurations to calculate the fidelity of the power estimators used in our work for three LTE modulation schemes — QPSK, 16-QAM, and 64-QAM. The respective fidelity values resulting from these experiments are 0.79, 0.78, and 0.81. The proposed power estimation method and corresponding fidelity calculation method are not restricted to FPGA implementation, and can be adapted readily to implementations on other types of platforms.

In addition to imposing constraints on system power consumption, the LTE standard requires the ACPR and error vector magnitude (EVM) levels of the transmitted signal to stay below certain values. Different modulation schemes impose dif-



ferent constraints on the transmitter; thus, we incorporate ACPR and EVM as two distinct optimization objectives in our MOO framework. ACPR is commonly used to quantitatively assess the extent of out-of-band energy leakage [12], while EVM measures the distortion of the original signal under the influence of non-linearities introduced by the PA and DPD subsystems. More details of ACPR metric for DPD systems are discussed in Section 3.4.1.2.

## 2.2.2 Design Space for DPD Implementation

### 2.2.2.1 Polynomial Orders

Since the DPD algorithm proposed in [5] splits its signal processing into a direct part and a conjugate part, which enables use of different polynomial orders for direct and conjugate signal terms. For example, a DPD system can be realized with fifth-order for the direct signal and only third-order for the conjugate signal. We denote the polynomial order for the direct signal by  $P$ , and that for the conjugate signal by  $Q$ . The ordered pair  $(P, Q)$  is referred to as the *polynomial order parameter* in the DPD design space. Following [5], we allow only odd-valued term orders. For example, if  $P = 5$  and  $Q = 3$ , then  $I_P = \{1, 3, 5\}$ , and  $I_Q = \{1, 3\}$ . If  $x$  denotes the input signal to be filtered, then each term order  $i \in I_P$  corresponds to the processing of  $|x|^{i-1}x$  by an FIR filter, and similarly, each term order  $j \in I_Q$  corresponds to FIR filtering of  $|x * |^{j-1}x*$ . Thus, the number of instantiated FIR filters for this example in Fig. 2.2 is 5.

### 2.2.2.2 FIR Filter Orders

We define the *filter order parameter* in our DPD design space to be the ordered pair  $(\lambda_P, \lambda_Q)$ , where  $\lambda_P$  represents the FIR filter order for the direct branch, and similarly,  $\lambda_Q$  represents the common FIR filter order for the conjugate branch. Thus, we allow heterogeneous filter orders across branches, but all filters in the same branch have the same order.

### 2.2.2.3 Bit-width

We introduce the bit-width  $B$  of the input data samples and FIR filter coefficients as another DPD design space parameter. We assume that all of the employed filters have the same bit-width, which is defined by the design space parameter  $B$ .

## 2.3 Experimental Setup and Simulation Results

As described in Section 2.2, the design space for our DPD system is based on the following parameters:  $(P, Q)$ ,  $(\lambda_P, \lambda_Q)$ , and  $B$ . We explore the design space defined by these parameters across the following ranges of admissible values for the different parameter components —  $P$  and  $Q$ :  $\{1, 3, 5\}$ ;  $\lambda_P$  and  $\lambda_Q$ :  $\{1, 2, 3, 4, 5\}$ ; and  $B$ :  $\{5, 6, \dots, 15\}$ . This results in a total of  $3^2 \times 5^2 \times 11 = 2475$  distinct configurations, which we evaluate exhaustively with the aid of our proposed power estimator.

The constraint on ACPR used in this chapter for all three modulation schemes is  $-45.0$  dBc. The constraints on EVM are  $-15.1$  dB,  $-18.1$  dB, and  $-21.1$  dB for

QPSK, 16-QAM, and 64-QAM, respectively. We also impose a constraint on the *symbol error rate (SER)*, which is measured as the average rate of erroneous symbol transmissions. We filter out the configurations with non-zero SER levels to further increase the accuracy of the DPD system. To measure the ACPR, EVM, and SER for a given DPD configuration under a specific modulation scheme, we generate 10,240 OFDM symbols with a baseband sampling rate of 5 MHz and an upsampling rate of 8. This procedure is repeated 100 times and averages are calculated over these 100 trials to derive values for the three performance metrics.

After the aforementioned DPD design space exploration, we obtain 26, 27, and 8 Pareto-optimal configurations for QPSK, 16-QAM, and 64-QAM, respectively. We find that for most of the derived Pareto-optimal settings,  $P \geq Q$ , which validates the argument in [5] that the higher orders of the conjugate predistorters are weak, and a smaller  $Q$  value is therefore preferred. To further explore the utility of this kind of asymmetry between the direct and conjugate branches, we define the *branch asymmetry attribute (BAA)* of a DPD configuration as the associated value of  $(P - Q)$ , and we plot the BAA distribution of the obtained Pareto-optimal settings in Fig. 2.3.

From Fig. 2.3, we see that there is a high concentration of design points with  $(P - Q) = 2$  for all three modulations. This concentration involving asymmetric direct and conjugate branch processing validates the utility of decomposing DPD signal processing into these two separate parts. Our validation here, which is implementation-oriented, is complementary to the algorithm-level validation in [5] of such decomposed processing.

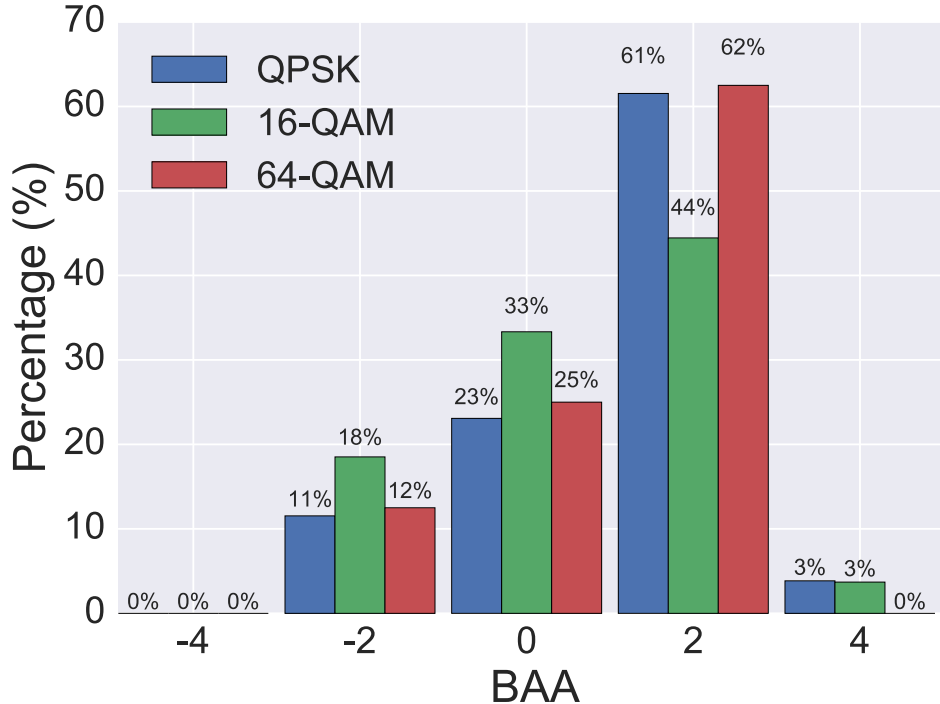


Figure 2.3: BAA distribution of the derived Pareto-optimal settings.

For each LTE modulation scheme, we select three representative DPD configurations among the Pareto-optimal design points derived from our multiobjective optimization approach. These configurations are selected from diverse regions of the design evaluation space. The selected configurations along with their associated figures of merit are listed in Table 2.1.

To further evaluate the effectiveness of the selected DPD structures, we also provide — in Table 2.1 — a comparison of transmitter performance with (a) a randomly-selected, fixed-configuration DPD system, and also with (b) a transceiver setup that does not use DPD. Values in boldface violate the aforementioned constraints. From the results in Table 2.1, we see that the fixed-configuration DPD system does not satisfy the ACPR constraint of any of the three modulation schemes, the EVM constraint of 64-QAM, and the SER constraint of 16-QAM and 64-QAM.

Table 2.1: Pareto-optimal configurations for three LTE modulation schemes. The configurations are shown in the format  $(P, Q, \lambda_P, \lambda_Q, B)$ , and the measurements in the format (Power, ACPR, EVM, SER) with units (mW, dBc, dB, error per symbol). The configuration of the fixed DPD system is (5, 1, 5, 2, 14).

Mod.	Configuration	Pareto-optimized DPD	Config.-fixed DPD	Without DPD
QPSK	(3,1,1,1,10)	(349.98,-45.61,-27.97,0)	(387.50,- <b>44.53</b> ,-37.98,0)	(NA,- <b>40.85</b> ,-21.21,0)
	(3,1,1,2,12)	(356.02,-46.35,-28.07,0)		
	(3,3,1,3,15)	(376.61,-46.68,-28.07,0)		
16-QAM	(3,1,2,1,10)	(352.86,-45.79,-27.88,0)	(383.95,- <b>40.98</b> ,-25.41,+)	(NA,- <b>40.82</b> ,- <b>11.24</b> ,0)
	(3,1,4,4,14)	(376.74,-46.49,-28.34,0)		
	(3,3,5,3,15)	(392.67,-46.55,-28.66,0)		
64-QAM	(3,1,2,2,10)	(354.98,-45.15,-21.54,0)	(388.59,- <b>42.58</b> ,- <b>20.39</b> ,+)	(NA,- <b>40.84</b> ,- <b>4.98</b> ,+)
	(3,3,3,1,10)	(360.32-46.07,-22.17,0)		
	(3,1,2,4,14)	(369.79,-46.34,-22.27,0)		

Furthermore, we see that most of the constraints are violated when no DPD is used. In contrast, the DPD system with our selected set of optimized configurations can meet the modulation-specific requirements in all cases, and with relatively low power consumption.

Next, we consider an optimized adaptive DPD (OAD) system that can switch itself to the most power-efficient configuration (as obtained from Table 2.1) associated with the current modulation scheme so that the DPD system can always satisfy the relevant real-time constraints with optimized power consumption. In Fig. 2.4, we compare predistortion figures of merit among transmitters with our OAD system; the non-optimized, fixed-configuration DPD system represented in Table 2.1; and a transmitter that does not use DPD. We perform this comparison using a time-varying sequence  $M$  of LTE modulation schemes, where  $M$  is derived from simulations using the ns-3 network simulator. In these simulations, we consider a single-user scenario where the COST231 path loss channel model from ns-3 is selected, base station position is constant, and user equipment mobility is modeled using a Gaussian Markov model. This comparison neglects the overhead (expected

to be relatively small) of switching between alternative configurations in the OAD system. A more comprehensive evaluation that accounts for this overhead is a useful direction for further work.

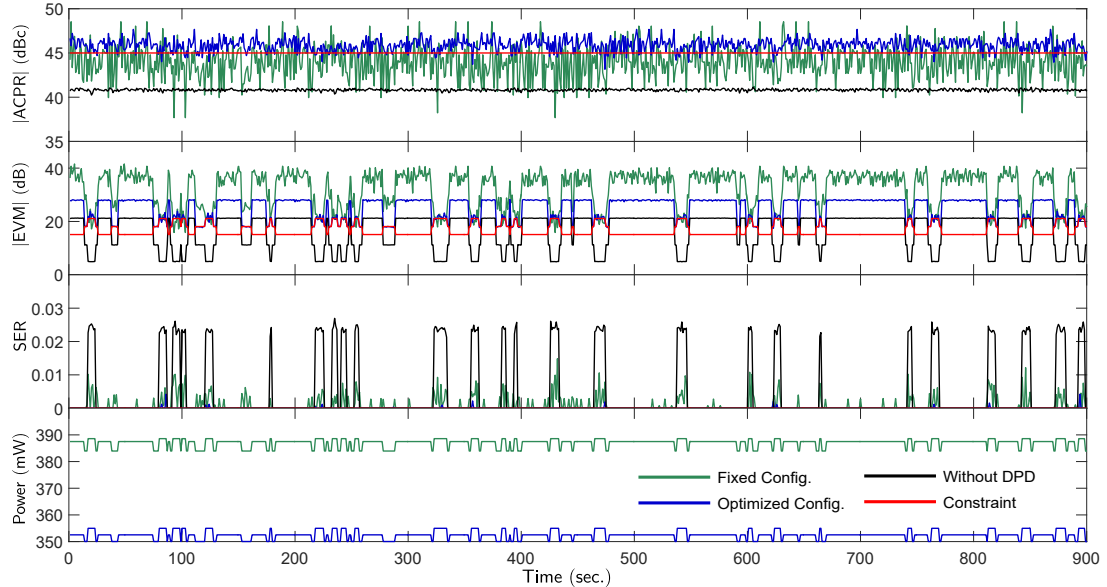


Figure 2.4: Real-time power and performance comparison involving the OAD system. ACPR and EVM measurements are presented using absolute values — thus, higher values indicate better performance.

The results in Fig. 2.4 demonstrate the potential of the OAD system, which is derived using our proposed DPD design optimization approach as a foundation, in achieving significantly improved trade-offs compared to the fixed-configuration and no-DPD alternatives.

## 2.4 Summary

In this chapter, we have developed new methods for exploring the design space for digital predistortion (DPD) system implementation. Our methods are developed to jointly optimize power consumption and multiple DPD performance objectives

subject to constraints imposed by the given modulation scheme. We have also demonstrated the utility of an adaptive DPD technique that switches among optimized configurations that are derived using our multidimensional design optimization methods. As part of the design space exploration process, we have applied an approach to efficiently estimate DPD system power, and we have validated the fidelity of this estimation approach. Simulation results demonstrate the capability of our proposed DPD design optimization techniques to support diverse signal processing trade-offs, and to significantly outperform a fixed-configuration DPD design under time-varying operational scenarios.

## Chapter 3

### Evolutionary Optimization for DPD Architectures

In Chapter 2, we introduced a design space exploration method for DPD that performs exhaustive search over a reduced subset of the design space. This process can be time-consuming and may not sufficiently explore the overall (original) design space. It is therefore useful to investigate more automated and thorough methods to search large DPD design spaces.

Additionally, to help maximize effectiveness, such design space exploration should be performed based on multidimensional operational criteria. With this motivation, we develop in this chapter a novel evolutionary algorithm framework for multiobjective optimization of DPD systems. We demonstrate our framework by applying it to develop an adaptive DPD architecture, called the *adaptive, dataflow-based DPD architecture (ADDA)*, where Pareto-optimized DPD parameters are derived subject to multidimensional constraints to support efficient predistortion across time-varying operational requirements and modulation schemes. Through extensive simulation results, we demonstrate the effectiveness of our proposed multiobjective optimization framework in deriving efficient DPD configurations for runtime adaptation.

Material in this chapter has been published in [19] and [20].



### 3.1 Introduction

A major challenge in deploying DPD architectures for cognitive radio systems is the dynamic optimization of key DPD parameters subject to time-varying and multidimensional constraints on system performance. A general approach to such optimization is to perform efficient search at design time (i.e., off-line) across alternative DPD configurations, and to then select from the search results a set of configurations that are Pareto-optimal, and that effectively cover the targeted range of operational scenarios and their trade-offs. These selected, “Pareto-optimized” configurations can then be stored in memory, and switched across during system operation based on time-varying changes in communication system requirements. Here, “Pareto-optimized” configurations refer to configurations that are Pareto-optimal with respect to the applied search process, while “Pareto-optimal” configurations refer to configurations that are globally optimal in a Pareto sense.

In this chapter, we develop a novel framework for systematic derivation of Pareto-optimized DPD system configurations that can be applied to adaptive DPD implementations. Our framework builds on the methodology of multiobjective evolutionary algorithms (e.g., see [21]), and incorporates adaptations of this methodology to efficiently handle distinguishing characteristics of DPD system optimization. We refer to our framework for DPD system optimization as the framework for *Evolutionary Adaptive DPD Implementation (EADI)* or (“EADI Framework”).

We demonstrate the EADI Framework in this chapter by applying it to develop an adaptive DPD architecture, called the *adaptive, dataflow-based DPD ar-*

*chitecture* (*ADDA*), where Pareto-optimized DPD parameters are derived subject to multidimensional constraints to support efficient predistortion across time-varying operational requirements and modulation schemes. While the *ADDA* architecture is used to concretely demonstrate the capabilities of the EADI Framework, the EADI Framework is not specific to any particular DPD architecture, and can readily be adapted to work across a variety of parameterized DPD architectures. Exploring such adaptations is a useful direction for future work that emerges from the developments of this chapter.

Similar to the work in 3, the design evaluation metrics (optimization objectives) targeted in our development of the EADI Framework and *ADDA* architecture in this chapter are system energy consumption, adjacent channel power ratio (ACPR), and system accuracy. We abbreviate this set of metrics as *EAA*. The *ADDA* is a parameterized architecture that can be configured dynamically to achieve a range of *EAA* trade-offs. The DPD design space that we consider consists of three design parameters: the polynomial order, bit-width, and filter order. This design space is modeled in the EADI Framework, and optimization results from the framework are used to extract a subset of generated Pareto-optimized configurations (settings of the DPD parameter values). This subset of configurations provides the set of DPD system modes that will be implemented in the *ADDA* architecture. The set of DPD modes provided in the *ADDA* configuration set is made available during operation such that predistortion trade-offs can be reconfigured among the different options in the configuration set based on dynamically changing operational requirements.

To demonstrate and experiment with the ADDA, we apply the *lightweight dataflow environment (LIDE)*, which is a design tool for dataflow-based design and implementation of signal processing systems [22]. Dataflow graphs provide a useful form of model-based design in many areas of signal processing, and wireless communications (e.g., see [23]). We map the signal flow structure of the ADDA into actors (dataflow-based signal processing components) in LIDE, and implement the internal functionality of these actors using the Verilog hardware description language (HDL).

We demonstrate the effectiveness of the EADI Framework through extensive simulations, and validate the capabilities of the ADDA through hardware synthesis.

## 3.2 Related Work

In this chapter, we exploit the decomposed, parallel structure of the DPD method introduced in [5], and we present new methods to search the design space, and derive Pareto-optimized realizations for this form of DPD architecture.

In architectures for cognitive radios, adaptive DPD systems that operate under Pareto-optimized configurations are highly desirable due to the multidimensional space of relevant implementation metrics. However, prior work on system-level DPD optimization has emphasized single-objective optimization of ACPR [13, 9]. These works employ a form of search technique called genetic algorithms, which are closely related to evolutionary algorithms, to optimize DPD ACPR performance. However, the resulting solutions may not be efficient in terms of energy consumption or accuracy. Furthermore, the underlying design methodology does not produce multiple

alternative configurations that may be employed for dynamic reconfiguration based on time-varying changes in operational requirements. The methods that we develop in this chapter address these limitations, respectively, through development of the (1) EADI Framework for multidimensional, Pareto-optimized DPD configuration, and (2) ADDA for reconfigurable DPD architecture implementation based on configurations that are derived by the EADI Framework.

The DPD design optimization problem addressed in our work can be viewed as a multiobjective optimization problem, where the multiple objectives are generally conflicting, preventing simultaneous optimization of all objectives. One approach to such a problem is to transform all of the objective functions into a single composite function — a common method for such an approach is to use a weighted sum of the objective functions. In this case, small changes to the weights may lead to large differences in the solution set, and proper selection of the weights can be a major problem. Also, the optimization method generally returns a solution set that is preferred by the applied weights, and thus has less diversity [24]. Another general approach is to attempt to compute a representative subset of the entire Pareto set of design points. The EADI framework developed in this chapter adopts this second approach, and therefore, does not suffer from the aforementioned limitations of the weighted sum approach.

A preliminary version of this chapter has been presented in [19]. This chapter goes beyond the previous optimization framework presented in [19] by employing fidelity-based validation of our employed power estimation approach, and applying an improved system accuracy measurement for DPD design space exploration. More

specifically, in Section 3.4, computation of estimation fidelity is integrated to verify the accuracy of the proposed power estimator, and the EVM measurement is modified to better represent the accuracy of the system. In Section 3.6, the simulation results are updated based on this new EVM measurement approach.

### 3.3 Adaptive Dataflow-based DPD Architecture

The ADDA architecture developed in this chapter is based on the algorithm presented in [5]. Since the first stage is intended for off-line computation, the ADDA architecture and EADI optimization process are focused only on the second (filtering) stage.

The structure of the predistortion filtering system is shown in Fig. 2.1. Details of the DPD structure have been shown in Section 2.2 of Chapter 2.

Fig. 3.1 illustrates the dataflow model of the DPD filtering subsystem that is employed in the ADDA. Here, the mode selection actor dynamically selects the DPD operational mode based on the current application scenario (i.e., based on the current modulation and requirements on EAA) and finds the corresponding parameter settings for that mode in its local memory, and distributes these DPD parameter values to the polynomial computation actor and all of the filter actors. Following [5], we decompose the signal processing for the applied DPD algorithm into separate direct and conjugate parts.

With the parameters obtained from the mode selection actor, the polynomial computation actor computes the polynomial basis function defined in Equ. 2.2 for

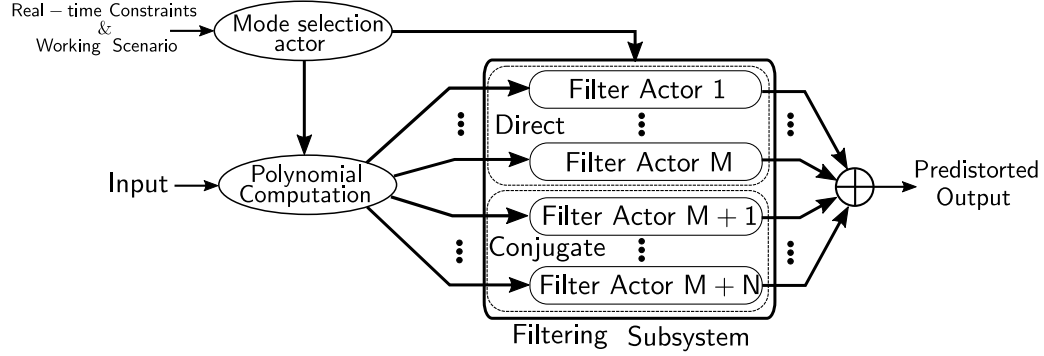


Figure 3.1: Dataflow graph model of the predistortion filter.

both the direct and conjugate branches. The computed polynomials are then sent to their corresponding branches and filtered by the filter actors in those branches. These filter actors are implemented with integrated use of LIDE and Verilog, as described in Section 3.1. As shown in Fig. 3.1, according to Equ. 2.1, the filtered samples (one output sample from each filter) are summed to produce a single sample as the final predistorted output.

Based on the analysis in [11], where a similar dataflow model is constructed for the DPD algorithm in [5], most of the computation and energy consumption is concentrated in the filter actors. Thus, in this chapter, we map only the filter actors to hardware, and focus our design optimization processes on the filter actors.

## 3.4 Optimization Metrics and Design Space

### 3.4.1 Optimization Metrics

In this subsection, we elaborate on the three objectives in our targeted design optimization problem. As defined in Section 3.1, we refer to these metrics collectively as *EAA*.

### 3.4.1.1 Energy Measurement

As explained in Section 3.3, we focus our energy measurement on the energy consumed by the filtering subsystem, and the figure of merit that we employ is the filtering energy expended to producing a single output sample, which is denoted by the *energy per sample* (*eps*). To calculate *eps*, we use the total power consumption of all FIR filters used in the predistortion subsystem, which we denote as  $P_{FIR}$ . The *eps* metric is then defined as  $\text{eps} = P_{FIR} \times C/F$ , where  $C$  represents the average number of clock cycles required by the filter actors to process a single new input sample, and  $F$  represents the clock frequency. In our design, both  $F$  and  $C$  are fixed for each configuration. Thus, *eps* is proportional to  $P_{FIR}$ , and we can therefore use  $P_{FIR}$  as optimization objective for our evolutionary algorithm process. Also, we report results for  $P_{FIR}$  in Section 3.6 (instead of *eps*) as our assessment of the energy efficiency of each configuration.

We implement the DPD filtering subsystem using the Altera EP2C35F672C6 field-programmable gate array (FPGA) from the Cyclone II family. To facilitate efficient design space exploration within the EADI optimization process, we model the power consumption as a function of the design vector  $[P \ Q \ \mathbf{BW}^T \ \mathbf{FO}^T]^T$ . The definitions of the quantities  $P$ ,  $Q$ ,  $\mathbf{BW}$  and  $\mathbf{FO}$  are given in Section 3.5.

We apply the same system-level DPD power estimation as described in 2.2.1.

### 3.4.1.2 ACPR Measurement

ACPR is a metric that is commonly used to assess the extent of out-of-band energy leakage [12]. ACPR is defined as the ratio of the mean power centered on the adjacent channel to the mean power centered on the desired channel, as shown in (3.1).

$$\text{ACPR} = 10 \log_{10} \frac{\int_{\omega_A} S(\omega) d\omega}{\int_{\omega_D} S(\omega) d\omega}. \quad (3.1)$$

Here,  $S(\omega)$  denotes the power spectral density of the postdistorter input signal  $s_n$ , and  $\omega_A$  and  $\omega_D$  denote the frequency bands of the adjacent channel and desired channel, respectively.

### 3.4.1.3 Accuracy Measurement

We measure the accuracy of candidate DPD designs by the *error vector magnitude (EVM)* and *symbol error rate (SER)*. The former is considered as an optimization objective and the latter as a constraint on the derived configurations. The EVM measures the distortion of original symbols under the influence of nonlinearities introduced by the PA and DPD. This distortion is calculated as

$$\text{EVM(Pf)} = \left( \frac{\sum_{k=1}^K |X_0(k) - \hat{X}^{\text{Pf}}(k)|^2}{\sum_{k=1}^K |X_0(k)|^2} \right)^{\frac{1}{2}}, \quad (3.2)$$

where Pf represents a certain profile (finite sequence)  $X_0(1), X_0(2), \dots, X_0(K)$  of symbols to be transmitted, and  $\hat{X}^{\text{Pf}}(k)$  is the  $k$ th actual transmitted symbol under



Pf.

SER is measured as the average rate of erroneous symbol transmissions. This rate is determined as

$$\text{SER(Pf)} = \frac{1}{K} \sum_{k=1}^K I(X_0(k) - \hat{X}^{\text{Pf}}(k)) , \quad (3.3)$$

where  $I(x)$  (the *indicator function*), has value 1 if  $x \neq 0$  and 0 otherwise. We require that all of the configurations extracted for mapping into the ADDA must have zero SER.

### 3.4.2 Design Space

In this section, we elaborate on the selected DPD parameters that define the predistorter design space associated with the ADDA.

#### 3.4.2.1 Polynomial Orders

In this chapter, polynomial orders are defined in the same way as Section 2.2.2.1. Thus, the number of branches (or filter actors) that is employed in a specific DPD configuration is given by  $N_{branch} = (P + 1)/2 + (Q + 1)/2$ . In our experiments, we set the domain  $D$  of valid values for both  $P$  and  $Q$  as  $D = \{1, 3, 5, 7, 9\}$ . Thus, there are in total 25  $P - Q$  combinations in our targeted design space.

### 3.4.2.2 Bit-widths

Intuitively, smaller bit-widths for data storage and computation lead to less energy consumption. However, signal processing accuracy may be traded off as a consequence. To incorporate this trade-off between energy efficiency and accuracy, we incorporate bit-width as a parameter of ADDA, and as a design space component of EADI. Considering requirements on system accuracy and constraints on hardware resources, we set the range of allowable bit-widths in our experiments as  $\{5, 6, \dots, 15\}$ . Additionally, we allow different branches to be configured with different bit-widths in the same design. This leads to great flexibility in design optimization, and a correspondingly large design space — if there are  $m$  branches used in a specific design, then the total number of valid bit-width combinations is  $11^m$ .

### 3.4.2.3 Filter Orders

Similar to the bit-width design, the filter used in each branch may also have different number of coefficients. We denote this parameter as *filter order*. The filter order parameters would also significantly affect the trade-offs among EAA. The range of filter order in this chapter is set to be  $\{1, 2, 3, 4, 5\}$ .

According to the above description, our design space is too huge for exhaustive search. As a numerical example, given the aforementioned ranges for the system parameters, the design space would contain more than  $55^{10}$  configurations.

## 3.5 Multiobjective Optimization Using Evolutionary Algorithm

As motivated in Section 3.4, the DPD design space addressed in this work is a complex multidimensional space that is too large to be evaluated using exhaustive search techniques. Therefore, we apply a heuristic search strategy called *evolutionary algorithms (EAs)*, including a particular form of EA, called *strength Pareto EA (SPEA)*, that is suited for multiobjective optimization [21]. We select the SPEA approach due to its efficiency and scalability in addressing complex optimization problems, and its customizability to different kinds of design spaces and optimization criteria. This latter feature makes the EADI Framework readily adaptable across different kinds of DPD architectures and communication system constraints.

### 3.5.1 Problem Encoding

The parameters involved in the DPD design optimization problem are polynomial orders, bit-widths, and filter orders. Each configuration can be represented throughout the EA process by a vector, specified as  $[P \ Q \ \mathbf{BW}^T \ \mathbf{FO}^T]^T$ . Here,  $P$  and  $Q$  are the direct and conjugate polynomial order, respectively. As described in Section 3.4, the maximum number of branches considered in the design space is 10 (at most 5 branches for both the direct signals and the conjugate signals). Thus,  $\mathbf{BW}$  is a vector with 10 dimensions representing bit-width settings for up to 10 branches, where each dimension represents the bit-width associated with the corresponding branch. For the branches that are not used, the corresponding vector elements are set to zero. Similar conventions are applied to generate the 10-dimensional vector

**FO** of filter order settings.

As discussed in Section 3.1, the objective space of the EADI Framework encompasses average power consumption, ACPR and EVM. Thus, the objective vector can be formulated as  $[P_{FIR} \text{ ACPR EVM}]$  with units (mW, dBc, %). Here,  $P_{FIR}$  is the power consumption, as estimated by the method discussed in in Section 3.4, and ACPR and EVM are calculated according to (3.1) and (3.2), respectively.

### 3.5.2 Optimization Process

The EADI optimization process is executed separately for each modulation type that is to be supported in the targeted ADDA platform. The resulting Pareto-optimized configurations for the different modulation types are then collected and stored in the ADDA memory. This enables the ADDA to dynamically to select among different modulation types, and among different operational trade-offs for each modulation type.

As mentioned previously, the work flow of the EADI optimization process is based on the SPEA methodology for multidimensional search. For details on SPEA, we refer the reader to [21].

The SPEA-based optimization workflow used in our work is illustrated in Fig. 3.2.

According to SPEA, the population set (set of candidate solutions or *individuals*)  $\rho$  contains the individuals generated during each SPEA iteration, and the external set  $\bar{\rho}$  maintains selected non-dominated individuals among all individuals

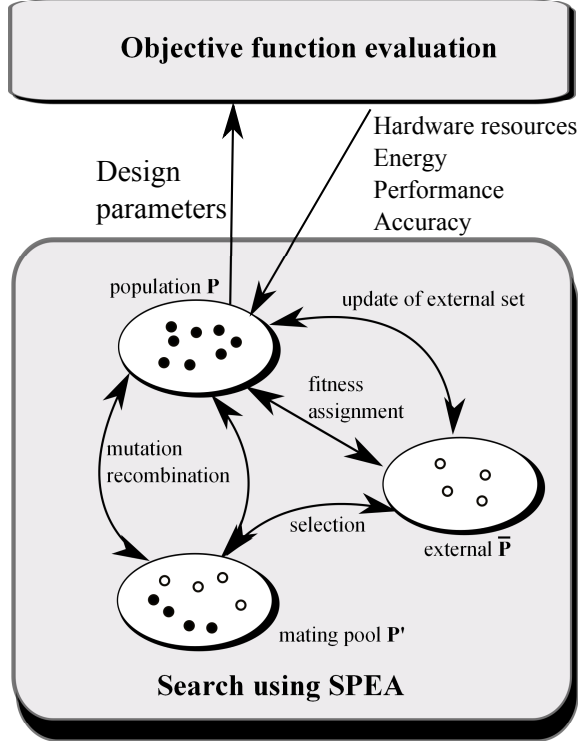


Figure 3.2: Multiobjective optimization model for DPD system.

generated so far up through the current iteration. Here, we say that an individual  $x$  dominates another individual  $y$  if  $x$  is superior to  $y$  in terms of at least one design evaluation metric, and  $x$  is not inferior to  $y$  in terms of any metric. A *non-dominated* individual is one that is not dominated by any individual.

We initialize  $\rho$  with a well-distributed population across the design space. For each possible  $P - Q$  combination, we generate two design vectors by selecting the corresponding bit-width and filter order values randomly from their valid ranges. Thus, the size of  $\rho$ , denoted by  $\mathbf{N}$ , is 50 individuals.

During each iteration, each individual in  $\rho$  is evaluated to generate the objective vector  $[P_{FIR} \text{ ACPR} \text{ EVM}]$ . The individuals that do not satisfy certain modulation-specific constraints (defined in Section 3.6) are ignored. Only the re-

maintaining non-dominated individuals are copied to  $\bar{\rho}$ . If the size of  $\bar{\rho}$  exceeds a pre-defined maximum population size  $\bar{N}_{max}$ , a k-means clustering algorithm is used to classify the members in  $\bar{\rho}$  into  $\bar{N}_{max}$  groups. This allows us to limit the size of  $\bar{\rho}$  while maintaining a diverse population in  $\bar{\rho}$  by retaining a “representative” individual of each group in  $\bar{\rho}$  [21].

After updating of  $\bar{\rho}$  during an optimization iteration (*generation*), individuals from both  $\rho$  and  $\bar{\rho}$  are selected to generate a “mating pool”  $\rho'$ . This selection process is performed randomly in a manner such that the probability of an individual’s selection for the mating pool is larger for individuals with smaller fitness values. Here, “fitness” is a measure of the quality of an individual; smaller fitness values imply higher quality solutions. The *recombination operator* selects pairs of individuals (“parents”) in  $\rho'$ , and for each selected pair, two new individuals (“children”) are generated with probability  $p_r$ .

Each generated child (from recombination) undergoes a process of random modification by a *mutation operator* with probability  $p_m$ . After all recombination and mutation operations are completed on the mating pool  $\rho'$ , the resulting new population is assigned as the current population  $\rho$  for the next generation. The individuals that comprise the set  $\bar{\rho}$  after  $\mathbf{T}$  generations are the Pareto-Optimized solutions obtained by the EADI Framework. Here,  $\mathbf{T}$  is a pre-defined number of optimization iterations that is to be executed by the SPEA.

The values  $p_r$ ,  $p_m$ , and  $\mathbf{T}$  are design parameters of the optimization process that can be set through experimentation or by selecting commonly-used values from the literature.

These general concepts of fitness measures, recombination operators, and mutation operators are standard components of EAs. They are applied to form an optimization process that has analogies to processes by which living species evolve. However, these three operators need to be designed specifically for each optimization context. In the remainder of this section, we discuss how these operators have been designed in the EADI Framework.

### 3.5.3 Fitness Measure

Based on the SPEA approach, each individual  $\mathbf{i} \in \bar{\rho}$  is assigned a real value  $S(\mathbf{i}) \in [0, 1)$ , which is referred to as the *strength* of  $\mathbf{i}$ . If  $N$  represents the number of individuals in the set  $\rho$ , then  $S(\mathbf{i})$  is calculated as the ratio of (a) the number of individuals in  $\rho$  that are dominated by  $\mathbf{i}$  to (b)  $(N + 1)$ . The fitness of  $\mathbf{i}$  is equal to  $S(\mathbf{i})$ . The fitness of an individual  $\mathbf{i} \in \rho$  is calculated by summing the strengths of all individuals  $\mathbf{j} \in \bar{\rho}$  that dominate  $\mathbf{i}$ , and then adding one to this sum. We add one to the sum here in order to guarantee that members in  $\bar{\rho}$  have better fitness than members in  $\rho$  (since fitness is to be minimized).

### 3.5.4 Recombination Operator

Recombination is a process of selecting parent solutions and producing child solutions from them that integrate properties of the corresponding parent solutions. The inputs of the recombination operation are the configuration vectors of the two selected parents  $\mathbf{Y}_1$  and  $\mathbf{Y}_2$ , and the outputs are either (a) the same two parents  $\mathbf{Y}_1$

and  $\mathbf{Y}_2$  (with probability  $(1 - p_r)$ ) or (b) the configuration vectors of two generated children (with probability  $p_r$ ), denoted by  $\mathbf{C}_1$  and  $\mathbf{C}_2$ .

In the latter case (when children are generated), the process of generating each child individual  $\mathbf{C}_k$ ,  $k = 1, 2$  from the two parents is summarized as follows: (i) assign  $P$ ,  $Q$  values (polynomial orders) from  $\mathbf{Y}_1$  or  $\mathbf{Y}_2$  to  $\mathbf{C}_k$  with equal probability subject to the requirement that the generated pair of  $P$  and  $Q$  values for  $C_1$  and  $C_2$  cannot be identical to each other; (ii) set the bit-width and filter order values of each child  $\mathbf{C}_k$  to the corresponding values of an average vector  $Y_{avg}$ :  $Y_{avg} = \gamma(Y_1, Y_2)$ , where  $\gamma(Y_1, Y_2)$  first computes the average  $(\mathbf{Y}_1 + \mathbf{Y}_2)/2$ , and for each component in this average vector that is not integer-valued, the operator replaces the component by its floor or ceiling with equal probability; and (iii) set the bit-widths and filter orders of the unused branches in the children to be zero.

### 3.5.5 Mutation Operator

In EAs, mutation operators are employed to help promote diversity from one generation of a population to the next by randomly modifying selected solution components (“genes”) within individuals. In the EADI Framework for ADDA implementation, the genes for potential mutation are taken to be the vector-valued settings of **BW** and **FO**. The specific gene (**BW** or **FO**) to which modification is to be applied is selected randomly with equal probability, and then a single component of the selected vector that is to be modified is selected randomly (with equal probability among all vector components). The mutation operator replaces the value



of the selected vector component with a uniform random value drawn between the given upper and lower bounds for that component.

### 3.6 Experimental Setup and Simulation Results

To validate the EADI Framework and ADDA platform, and to demonstrate their capabilities, we experiment with three Long-Term Evolution (LTE) modulation schemes — Quadrature Phase Shift Keying (QPSK), 16-QAM, and 64-QAM. The multiobjective optimization process is performed separately for each of the three modulation schemes, and then the resulting Pareto-optimized solution sets are integrated into the ADDA as discussed in Section 3.5. For all three modulation schemes, we employ the following SPEA parameter settings: (i)  $\mathbf{T} = 100$  (number of generations); (ii)  $\mathbf{N} = 50$  (population size); (iii)  $\bar{\mathbf{N}}_{\mathbf{max}} = 20$  (maximum size of external set); (iv)  $p_r = 0.8$  (recombination rate); (v)  $p_m = 0.2$  (mutation rate). These values for generic SPEA settings are values that are commonly used in the literature (e.g., see [21, 25]).

The constraint on ACPR used in the EADI Framework for all three modulations is  $-45.0$  dBc. The constraints on EVM are 17.5% , 12.5%, and 8% for QPSK, 16-QAM, and 64-QAM, respectively. The constraint on SER is that it should be zero.

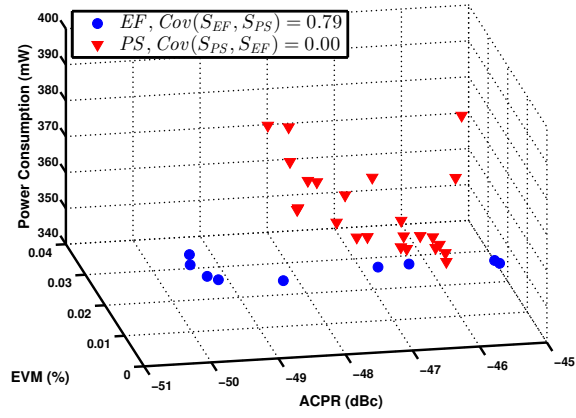
To help validate the effectiveness of the EADI Framework in deriving high quality DPD configurations, we apply a *partial search (PS)* method to solve the same multiobjective optimization problem. PS involves performing a complete search on

a reduced design space. PS is also a widely-applied method for obtaining Pareto fronts in multiobjective optimization problems (e.g., see [26]).

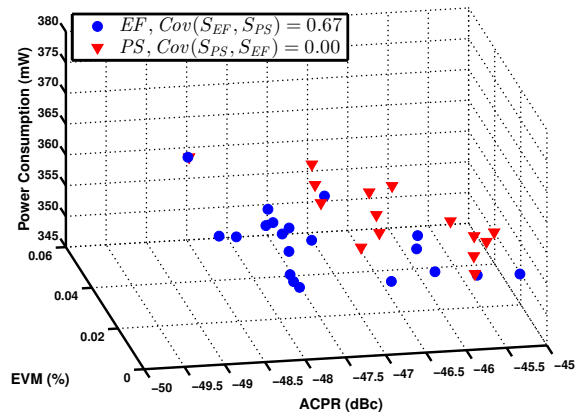
In our PS approach, we reduce the search space by equalizing the bit-widths and filter orders of all the filters used in all branches and apply the same valid parameter value ranges as used in the SPEA process. Thus, the reduced design space contains  $5 \times 5 \times 11 \times 5 = 1375$  configurations. We evaluate these 1375 configurations exhaustively with the  $P_{FIR}$ , ACPR, SER and EVM computations, as described in Section 3.4. We then remove the undesirable solutions based on the same SER, ACPR and EVM constraints as applied in the SPEA. Finally, we collect all of the non-dominated configurations from the resulting design space as the Pareto front obtained by the PS.

In the PS process, we estimate  $P_{FIR}$  using relevant FPGA design tools (Altera PowerPlay Analyzer), while in the EADI process, we estimate  $P_{FIR}$  using the power estimator introduced in Section 3.4. The estimator of Section 3.4 enables faster power estimation (at some expense in accuracy), which is important because very large numbers of candidate solutions are evaluated during the EADI process. For the Pareto-optimized configurations achieved by EADI, we also estimate  $P_{FIR}$  using FPGA tools to obtain more accurate power estimation results for the derived Pareto front. In the results that we report in the remainder of this section, the comparison between the quality of the two solution sets (PS and EADI) is based on the same (more accurate) power estimation method — i.e., using FPGA tools.

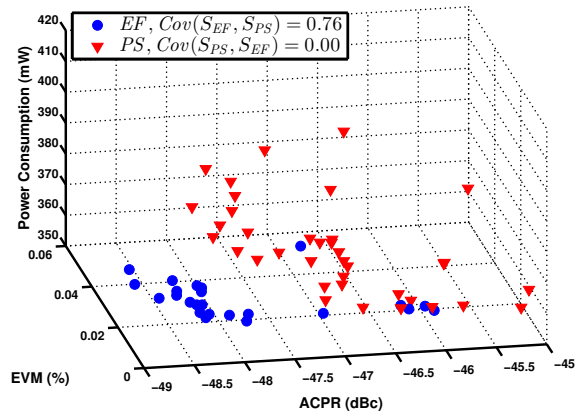
The Pareto fronts derived by the EADI Framework and PS for the three selected modulations are shown in Fig. 3.3(a) to 3.3(c). We use *coverage of two sets*



(a)



(b)



(c)

Figure 3.3: Pareto-optimized solutions obtained from the EADI Framework and PS for (a) QPSK, (b) 16-QAM, (c) 64-QAM.

(*Cov*) measurements [21] to evaluate the quality of the solution sets produced by the EADI Framework and PS, which we denote by  $S_{EF}$  and  $S_{PS}$ , respectively. Given a multiobjective design space, and two sets  $\alpha$  and  $\beta$  of candidate solutions in this space,  $Cov(\alpha, \beta) = dom(\alpha, \beta)/size(\beta)$ , where  $dom(\alpha, \beta)$  is the number of solutions in  $\beta$  that are dominated by at least one solution in  $\alpha$ . Coverage results for each of the three modulation schemes are given in Fig. 3.3(a) to 3.3(c) along with plots of  $S_{EF}$  and  $S_{PS}$ . Here, we see that  $Cov(S_{PS}, S_{EF})$  is uniformly zero over all three modulations, while the values for  $Cov(S_{EF}, S_{PS})$  indicate that significant proportions of the PS solutions are dominated by results from the EADI Framework.

We also measured that the PS method requires approximately 91 hours to evaluate the three optimization metrics for the 1375 given configurations, and extract the Pareto front, while the evaluation and Pareto front extraction by the EADI Framework takes only about 1 hour. We conclude from these results involving *Cov* and optimization time that the EADI Framework significantly outperforms the PS method in terms of both the quality of the obtained Pareto fronts and run-time efficiency.

To concretely demonstrate DPD performance trade-offs realized in the proposed ADDA architecture, we first classify the individuals in the Pareto front obtained by EADI into three groups according to their power consumption levels. Then we select one representative individual in each group and store it in ADDA as a DPD working mode. The selected design vectors and their corresponding  $P_{FIR}$ -ACPR-EVM measurements under three modulations in LTE are listed in Table 3.1. From this table, we see that for the Pareto-optimized parameter settings obtained

	Power Level	$P, Q$	BW		FO		Performance
			Direct	Conj.	Direct	Conj.	
QPSK	Low	3, 1	11, 9	5	3, 2	3	352.27, -45.35, 1.20
	Medium	3, 1	11, 9	9	2, 2	4	354.91, -47.22, 0.75
	High	3, 1	14, 10	11	4, 2	2	361.84, -50.13, 0.64
16-QAM	Low	3, 1	11, 8	11	3, 1	1	353.11, -45.16, 1.09
	Medium	3, 3	11, 10	11, 5	3, 1	3, 1	359.04, -46.48, 0.84
	High	3, 1	15, 11	13	5, 4	5	375.71, -49.30, 0.96
64-QAM	Low	3, 3	11, 9	11, 5	3, 2	1, 1	354.64, -46.19, 1.38
	Medium	3, 3	13, 9	11, 5	3, 2	3, 1	361.16, -48.33, 1.15
	High	5, 1	15, 12, 9	15	5, 4, 3	3	381.53, -47.35, 0.74

Table 3.1: Selected Pareto-optimized parameter settings for LTE under different modulations. The design evaluation metrics are shown in the format  $(P_{FIR}, ACPR, EVM)$  with units (mW, dBc, %).

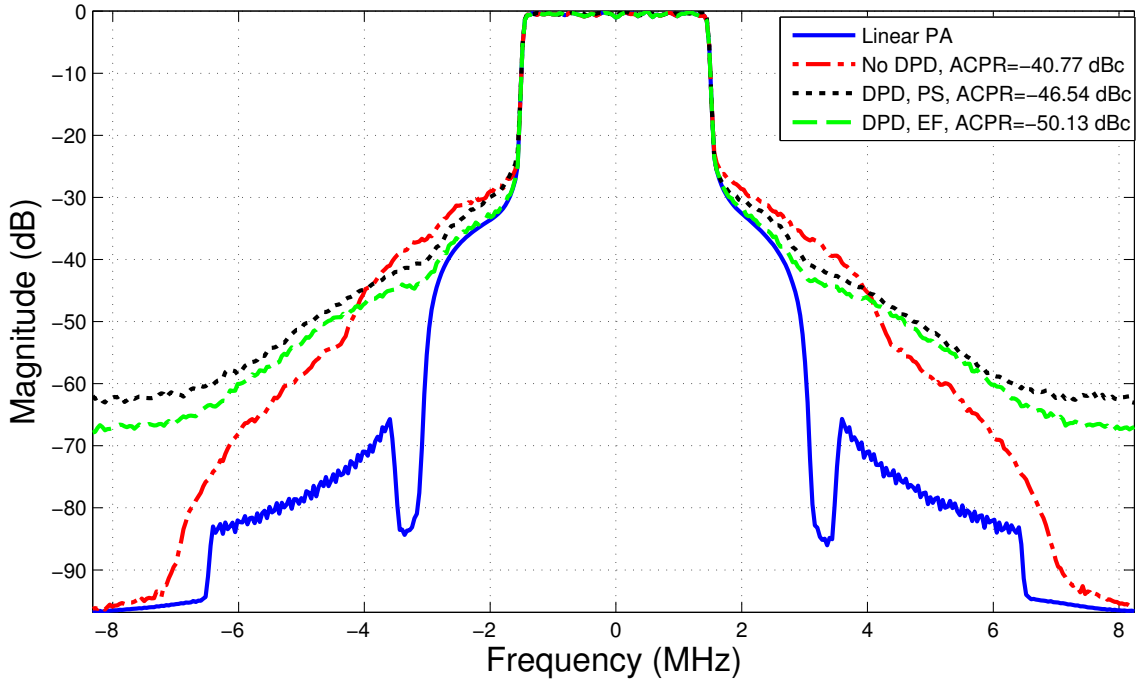


Figure 3.4: Output spectra of the ideal linear PA, the Wiener PA model without DPD, with DPD under configuration obtained from PS and EF.

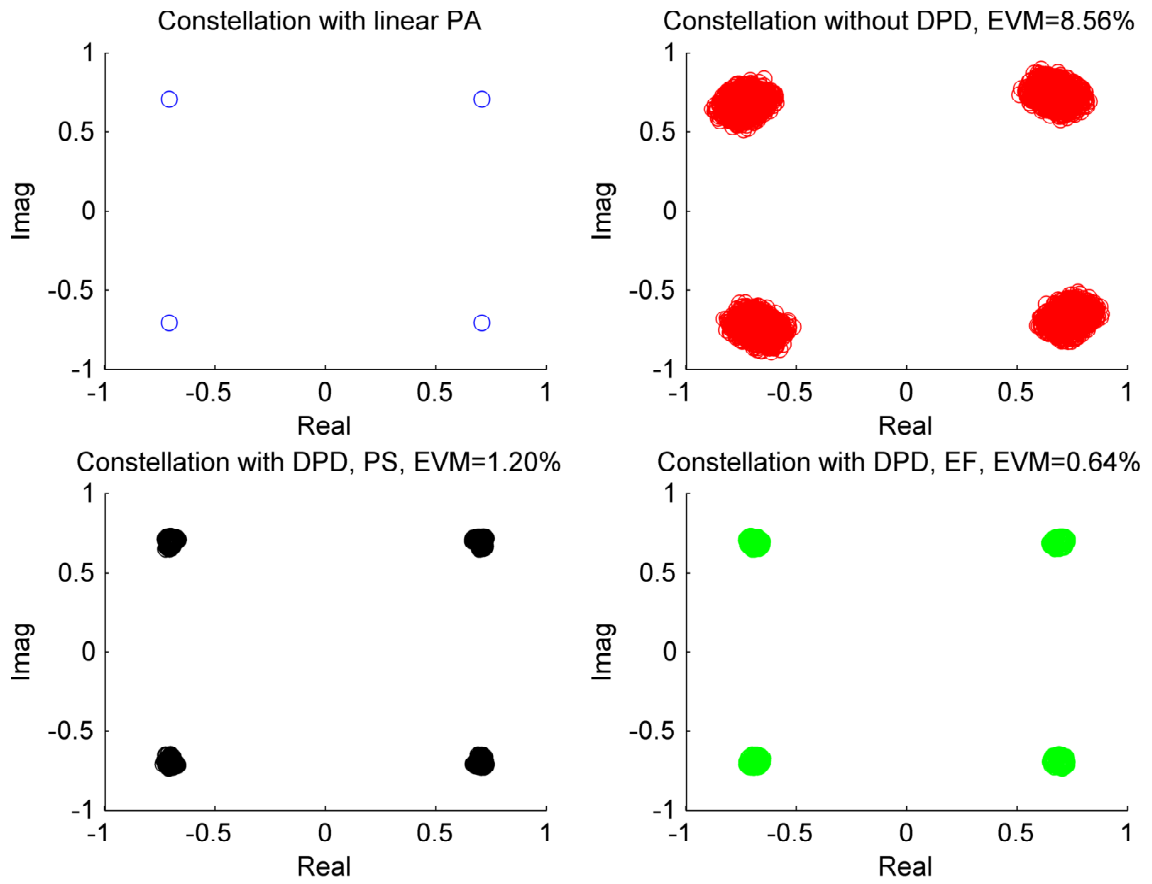


Figure 3.5: Output constellation of the ideal linear PA, the Wiener PA model without DPD, with DPD under configuration obtained from PS and EF.

by EADI,  $P$  is always greater than or equal to  $Q$ , which validates the argument in [5] that the higher orders of the conjugate predistorters are weak, and a smaller  $Q$  value is therefore preferred. Also, in general, the branches corresponding to the lower polynomial orders are configured with higher bit-widths and filter orders compared to the branches corresponding to higher polynomial orders. This results from the the higher order signals being relatively weak for both direct and conjugate parts. Fig. 3.4 and Fig. 3.5 show the *power spectral density (PSD)* and constellation of the PA output without DPD, with DPD under one configuration obtained by SPEA, and with DPD under one configuration obtained by PS with a similar power level for LTE QPSK modulation as an example. PSD and constellation of the output with an ideal linear PA is also presented as a reference. It can be seen from Fig. 3.4 and Fig. 3.5 that working under the same power level, the DPD system with the configuration selected from SPEA results outperforms that with the configuration selected from PS results in terms of both ACPR and system accuracy.

### 3.7 Summary

In this chapter, we have presented a novel framework, called the Evolutionary Adaptive DPD Implementation (EADI) Framework, for multiobjective optimization of digital predistortion (DPD) systems. The targeted optimization objectives include system energy consumption, adjacent channel power ratio (ACPR), and system accuracy. We apply the EADI Framework to develop an architecture, called the adaptive, dataflow-based DPD architecture (ADDA), where Pareto-optimized DPD

parameter settings are derived to support efficient, adaptive predistorter operation. Simulation results demonstrate the effectiveness of the EADI Framework in deriving efficient DPD configurations across time-varying modulation schemes subject to multidimensional constraints. The extracted Pareto-optimized configurations also help to validate assumptions in the DPD literature about preferred DPD parameter settings. Finally, the EADI Framework is shown to significantly outperform a partial search method in terms of both optimization time efficiency and the quality of the derived Pareto fronts.



## Chapter 4

### Design of Adaptive Signal Processing Systems Using Markov

#### Decision Processes

In Chapter 2 and Chapter 3, we proposed two different schemes for design space exploration of DPD systems. In this chapter, we apply the framework of Markov decision processes (MDPs) for design optimization in adaptive embedded signal processing systems. In contrast to the contributions of Chapter 2 and Chapter 3, which are specialized for DPD systems, the contribution in this chapter is more general, and can be applied across a wide variety of signal processing applications.

The design optimization framework introduced in this chapter, called the Hierarchical MDP framework for Compact System-level Modeling (HMCSM), embraces MDPs to enable autonomous adaptation of embedded signal processing under multi-dimensional constraints and optimization objectives. The framework integrates automated, MDP-based generation of optimal reconfiguration policies, dataflow-based application modeling, and implementation of embedded control software that carries out the generated reconfiguration policies. HMCSM systematically decomposes a complex, monolithic MDP into a set of separate MDPs that are connected hierarchically, and that operate more efficiently through such a modularized structure. We demonstrate the effectiveness of our new MDP-based system design framework through experiments with an adaptive wireless communications receiver.

Material in this chapter has been published in [27].

## 4.1 Introduction

Modern signal processing applications impose increasing demands of adaptivity, flexibility and reconfigurability. This trend presents challenges at many levels of system design, implementation and optimization. On one hand, adaptive signal processing systems must adjust to dynamically-changing environmental conditions, system status or user requirements; on the other hand, the systems must often satisfy stringent constraints on energy-efficiency and real-time performance.

In this chapter, we apply Markov decision processes (MDPs) to address this challenge of autonomous adaptation of embedded signal processing under multidimensional constraints and optimization objectives. MDPs have been used in many application areas as a foundation for dynamic determination of system configurations in stochastic environments. Representative areas include artificial intelligence [28], mobile systems [29], and wireless sensor networks [30].

Various methods have been developed to improve the practical utility of MDPs in complex design problems involving dynamically adaptive systems. For example, Boutilier et al. propose factored MDPs as a method for compact representation of large, structured MDPs [31]. Benini et al. introduce a finite-state, abstract system model for power-managed systems [32]. In their approach, the system and its external environment are modeled as a service provider and a service requester, respectively, in the format of Markov chains. Each of these Markov chains has a set of

states and a matrix of state transition probabilities. The computed state-to-policy mapping is then stored in a local memory or controller and is used in real time to dynamically reconfigure the system according to its current state.

However, the complexity of the MDP algorithms in general grows exponentially with increases in the size of the state space. Jonsson and Barto present an algorithm that performs hierarchical decomposition of factored MDPs to help alleviate this growth in complexity [33]. Their approach to hierarchical decomposition systematically allows irrelevant state variables to be ignored. However, their development of hierarchical MDPs is focused on algorithms and theoretical analysis for state abstraction and MDP computation, and the connection to implementation of the hierarchical MDPs and application to real-world systems is not addressed. One objective of this chapter is to help bridge this gap in the context of embedded signal processing systems.

In particular, in this chapter, we integrate the MDP schemes presented in [34] and [33]. This results in a novel approach to formulating MDPs for policy optimization in embedded signal processing systems with complex state spaces, and stringent implementation constraints. In our proposed design framework, we apply hierarchical MDPs to decompose the modeling of the application and embedded processing system into multiple MDPs. Each smaller MDP is formulated using an approach similar to that developed in [32]. We refer to this hybrid MDP approach as the *Hierarchical MDP approach for Compact System-level Modeling (HMCSM)*.

To promote systematic derivation of embedded implementations using the HMCSM approach, we integrate the approach into the framework of dataflow-

based design of signal processing systems. Model-based design in terms of dataflow graphs helps to ensure properties, such as determinacy, deadlock-free operation, and bounded memory requirements, which are of great importance in the reliable implementation of embedded signal processing systems (e.g., see [3]). Dataflow also orthogonalizes the implementation of individual functional components (actors) from the system-level control and coordination among the actors. In our context, this separation of concerns is especially useful because it enables efficient and reliable switching across different system-level configurations while reusing individual actors across the configurations. With these motivations, we develop in this chapter a dataflow-based framework for design and implementation of adaptive signal processing systems using HMCSM.

To demonstrate the efficiency and flexibility of the proposed design framework and the corresponding libraries and tools, we implement an adaptive wireless communication receiver that dynamically optimizes its system configuration in response to changes in different use cases. As a key part of the receiver, the *channelizer* extracts multiple radio channels of distinct bandwidths from a digitized wideband input signal. Among various computing components of the receiver, the channelizer operates at the highest sampling rate in the system and accounts for most of the computational complexity and energy consumption [35]. By adapting the configuration of the channelizer based on the communication scenario, we seek to optimize its energy efficiency while ensuring that it extracts the number of channels that is required by the communication scenario at any given time. We design an HMCSM MDP to perform this adaptation, and apply our dataflow-based MDP implementa-

tion framework to realize the resulting adaptive signal processing on a state-of-art embedded processing platform. Through experiments based on this embedded implementation, we demonstrate the effectiveness of our proposed design framework on the adaptive channelizer application.

## 4.2 Background

In Section 4.1, we introduced MDP methods and dataflow-based design as two key foundations of the contributions in this chapter. In this section, we elaborate on background in these areas that is relevant to development of our proposed HMCSM design framework.

### 4.2.1 MDP Methods

In Benini’s work on MDP-based methods for system-level power management, the service provider, service requester, and power manager are defined as key system components. The policy is composed of a finite discrete sequence of decisions taken by the power manager. A generic deterministic stationary policy can be represented as a table with the rows representing all possible states and the columns representing all possible actions. The size of the policy table grows geometrically when the number of system states increases. As a result, the policies derived from the MDP techniques proposed in [32] are practical only for problems with relatively small numbers of system states.

A factored MDP only requires specification of the conditional probabilities

with respect to dependent state variables, in contrast with traditional MDPs where the probabilities with respect to independent variables must be specified. The resulting modeling components are smaller in size and their policies are more compact compared to traditional MDPs. A more detailed and systematic introduction to factored MDPs can be found in [31]. Based on the idea of factored MDPs, a number of factorization methods has been proposed (e.g., see [36]).

As mentioned in Section 4.1, hierarchical factored MDPs are explored by Jonsson and Barto [33]. They use a dynamic Bayesian network and a causal graph to identify relationships among state variables and construct a hierarchical MDP for a given policy optimization problem. In this work, we build on these theoretical foundations of hierarchical factored MDPs, and apply this class of MDPs to design and implementation of adaptive signal processing systems. Through a case study involving a reconfigurable wireless communications channelizer, we experimentally evaluate an embedded implementation derived using hierarchical factored MDPs, and we compare its performance with an implementation that is based on a traditional, single-MDP scheme.

The application of MDP techniques to reconfigurable channelizer implementation has been presented recently in [34]. Our chapter goes significantly beyond this previous work on channelizer implementation; details on these novel developments are discussed in Section 4.4.2.

## 4.2.2 Dataflow-based Modeling and Design

An important contribution of this chapter is the integration of MDP-based design methods into a model-based design framework based on dataflow models of computation. In the form of dataflow that we apply, signal processing applications are modeled as directed graphs, called dataflow graphs, in which vertices (actors) represent computations of arbitrary complexity; edges represent first-in, first-out (FIFO) communication channels between actors; and actors represent discrete units of computation, called *firings*, that consume and produce well-defined amounts of data from and to the incident FIFOs [2].

Conceptually, data is encapsulated in objects called *tokens* as they pass through FIFOs from one actor to another. In signal processing oriented dataflow models, special attention is given to the rates at which actors produce and consume data to and from their ports, respectively. These rates are referred to as the production rates and consumption rates of the associated actor ports or incident edges. Collectively, production rates and consumption rates are referred to as *dataflow rates*. Analysis in terms of dataflow rates can be useful for many kinds of optimizations, such as those involving scheduling and memory management (e.g., see [3]). Different forms of dataflow have been proposed based on different restrictions on the dataflow rates or how dataflow rates across an actor or throughout a graph are related. Examples are cyclo-static dataflow [37], scenario-aware dataflow [38], and synchronous dataflow (SDF) [39].

In this chapter, we apply a form of dataflow called *parameterized synchronous*

*dataflow (PSDF)* to demonstrate the model-based integration of the proposed MDP-based design techniques [40]. We use PSDF because it is useful in modeling dataflow graphs that have dynamically varying parameters. Quasi-static scheduling techniques have also been developed for these graphs that systematically derive *parameterized looped schedules* [41]. A parameterized looped schedule involves loops that iterate across subsets of actors, and have iteration counts that can be symbolic expressions in terms of static or dynamically-varying actor, edge or graph parameters.

We apply parameterized dataflow due to its natural match with our objective of developing a model-based framework for adaptive signal processing. However, we envision that the framework can be adapted into modeling environments that are based on other parametric dataflow models, such as Boolean parametric dataflow [42] and the parameterized and interfaced dataflow meta-model [43]. Investigating such adaptations along with application of the distinguishing tools and analysis techniques available for such alternative models is an interesting direction for future work.

To implement PSDF-based models of adaptive signal processing systems, we apply the Lightweight Dataflow Environment (LIDE) [1]. LIDE is a software tool for dataflow-based design and implementation of signal processing systems. LIDE is based on a compact set of application programming interfaces (APIs) that is used for constructing and connecting dataflow actors, edges, and graphs. These APIs have been implemented in a variety of implementation languages. In this chapter, we apply LIDE-C, which is based on C language implementation of the LIDE APIs. LIDE includes facilities for dynamically manipulating actor, edge and graph param-



eters. We use these facilities to incorporate PSDF semantics in the LIDE-based implementations that we develop when applying the HMCSM framework.

### 4.3 Hierarchical MDP Approach for Compact System-level Modeling

The HMCSM framework is illustrated in Figure 4.1. At design time, the application functionality is modeled using dataflow techniques, as illustrated in the top right region of the figure. The design process also involves modeling the environmental and system-level dynamics, and reconfiguration process in the form of a hierarchical MDP, as illustrated by the part of Figure 4.1 that is labeled Hierarchical MDP Subsystem. As part of this modeling process, Markov models are created of both the processing demands imposed on the system by the application, and the dynamics of the processing system components. In a classical MDP formulation, these elements are combined into a single MDP. In this chapter, we additionally explore the use of Hierarchical MDPs in comparison to a single MDP to address the scaling problems that are well known to be a major weakness of classical MDPs (see Section 4.1).

The single MDP is transformed into a hierarchy of multiple MDPs by first factoring the elements in the MDP based on their stochastic interdependencies. Once the MDP has been factored, it can be decomposed into sub-problems that can be independently solved by multiple MDPs arranged in a hierarchy. For details on the processes involved in factoring and transforming the original MDP, we refer the reader to [33]. In the HMCSM framework, the factoring and decomposition are

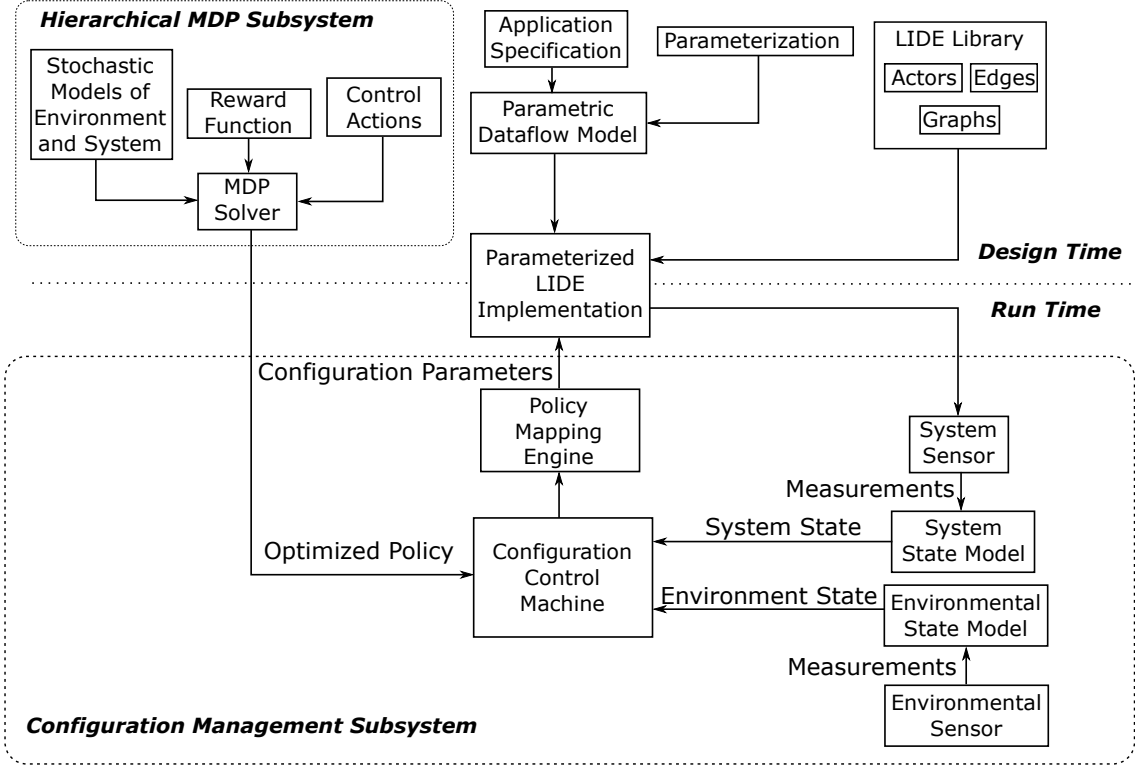


Figure 4.1: An illustration of the HMCSM framework for design and implementation of adaptive signal processing systems.

carried out by hand, using knowledge of the application domain and the processing system.

The Configuration Control Machine (CCM), shown in the lower (run time) portion of Figure 4.1, is used to manage dynamic system-level reconfiguration throughout operation of the embedded signal processing system. Here, by *dynamic reconfiguration*, we mean changes to any system-level parameters, including software, platform, and algorithmic parameters, that can be manipulated while the system is executing. At run-time, the CCM executes periodically, where the period  $T_c$  of its execution is a system parameter.

We refer to each periodic execution of the CCM as a *reconfiguration round*.

During a given reconfiguration round, the CCM determines, based on the current environmental state and system state, whether or not to perform a dynamic reconfiguration operation. Furthermore, if the determination is to perform such an operation, the CCM also determines the specific reconfiguration operation that is to be applied to the system. The blocks labeled System Sensor, System State Model, Environmental Sensor, and Environmental State Model represent measurements and models that are used by the CCM to determine the system and environmental state during a given reconfiguration round.

The block labeled *Control Actions*, in the design time portion, encompasses the set of possible reconfiguration operations that can be applied by the CCM in a given reconfiguration round. Examples of control actions in the context of HMCSM are changes to the type of digital filter that is applied to process a given signal in the application flowgraph, changes to the coefficients in a filter of a fixed type, and changes to the input port from which a given actor will read input data.

If  $A$  denotes the set of control actions, then the CCM can be viewed as an implementation of a function  $P : S_e \times S_s \rightarrow A$ , where  $S_e$  is the set of environmental states, and  $S_s$  is the set of system states. This function  $P$  is referred to as the *reconfiguration policy* or simply “policy” in an adaptive signal processing system that is developed using the HMCSM framework. In HMCSM, the policy is applied at run-time, and derived at design-time. It is derived using an MDP Solver, which is a software module that automatically generates optimized policies from MDP model specifications.

The Policy Mapping Engine, shown near the center of Figure 4.1, translates

control actions into updates to dynamic parameters in the embedded software that achieve the intended actions. In our implementation of the HMCSM framework, these parameter updates are made by setting appropriate variables in an implementation of the application dataflow graph that is developed using the Lightweight Dataflow Environment (LIDE) (see Section 4.2.2). This dataflow graph implementation is represented by the block labeled Parameterized LIDE Implementation, and the software tool that we use to construct this implementation is represented by the block labeled LIDE Library.

In the HMCSM framework, each control action is formulated in terms of specific changes to specific parameters in the parameterized dataflow application model  $M$ . In other words, a given control action  $A$  can be represented as  $A = \{(p_1, v_1), (p_2, v_2), \dots, (p_{N(A)}, v_{N(A)})\}$ , where each  $p_i$  is a distinct parameter in the application model  $M$ , each  $v_i$  is an admissible value of parameter  $p_i$ , and  $N(X)$  is the number of parameters in  $M$  that are manipulated by a given control action  $X$ . Execution of the control action  $A$  at run-time involves setting each parameter  $p_i$  to the corresponding value  $v_i$  such that subsequent operation of  $M$  will be performed using these new parameter settings. Operation continues with the new parameter settings until a new control action is applied to the system (in some subsequent reconfiguration round).

The formulation of an MDP in HMCSM includes three main components, which are represented by the blocks in Figure 4.1 that are labeled Stochastic Models of Environment and System, Reward Function, and Control Actions. These components are developed by hand using well-established foundations of MDP modeling,

along with domain knowledge of the targeted application.

We have discussed the Control Actions part of the MDP formulation earlier in this section. The Stochastic Models of Environment and System include, for each of the two models (environment and system), the definition of the state space and the state transition matrix (STM). The STM is a stochastic matrix that defines the probability of the transition to the next state given the existing state, and conditioned on a given action. Intuitively, the Reward Function maps state-action pairs into *scores* that assess the utility of performing the associated action (control action) during the given state. We apply an approach for incorporating multidimensional design objectives into the scores produced by the reward function. For details on this multidimensional reward function approach, we refer the reader to [34]. In our wireless communications case study, which we present in Section 4.4, the specific design objectives that we target are (a) energy consumption and (b) probability of packet collisions.

In summary, the HMCSM framework presented in this section provides a comprehensive methodology and supporting tools for design and implementation of adaptive embedded signal processing systems. The specific tools that we apply in our demonstration of the framework are MDPSOLVE [44] and LIDE, which correspond to the blocks labeled MDP Solver and LIDE Library, respectively, in Figure 4.1. However, the framework is not intended to be specific to these tools, and can readily be adapted to other tools for solving MDPs and implementing parametric dataflow graphs, respectively.

## 4.4 Case Study of Channelizer/Receiver Application

### 4.4.1 Background

In this section, we describe a detailed case study as an illustrative example of the methodology introduced in Section 4.3. In addition to providing a demonstration of our proposed HMCSM design framework on a practical, adaptive signal processing application, the case study also serves as a demonstration of a novel, application-specific, MDP-based system design.

### 4.4.2 Adaptive Receiver Architecture

Our case study is a wireless receiver for a Low Power Wide Area Network (LPWAN) used in a “Smart Cities” Internet of Things (IoT) application [45]. The example network consists of a basestation centrally located in a star topology providing coverage to hundreds or thousands of end nodes.

A key aspect of popular LPWAN protocols, such as LoRaWan and SigFox, is that the frequency subcarriers are not orthogonal, and hence are Frequency Division Multiplexed (FDM) but not orthogonal FDM (OFDM) systems. This represents a notable change of direction in wireless networks compared to modern protocols (all OFDM-based) that have dominated the landscape in recent years, for instance, Long-Term Evolution (LTE), Wi-Fi, and WiMax. The signal processing in an FDM receiver is quite different from an OFDM receiver, and has different design challenges. Another differentiation in LPWAN protocols compared to OFDM-based systems is that these protocols do not have explicit centralized scheduling of the

frequency spectrum (i.e., time and frequency reservations). Rather, the end nodes use decentralized heuristics like Carrier Sense Multiple Access (CSMA), commonly referred to as a *Listen Before Transmit* scheme, in order to avoid transmission collisions.

These two system aspects are a result of the design goal of minimizing complexity at the sensor nodes, to enable years of battery life in some applications. This long battery life is not currently possible on the OFDM-based systems it replaces. There are a number of major implications of these system aspects. For example, the removal of the explicit centralized transmit coordination by the basestation means that end nodes occasionally transmit in the same time and frequency, leading to packet collisions. Such a collision represents a wasted transmission that cannot be demodulated properly by the basestation. Additionally, the removal of the OFDM scheme means that subcarriers must be separated by guard bands. This in turn implies that the receiver must split the received signal into channels and run parallel banks of demodulators to recover the transmitted data. Correspondingly, the physical layer signal processing workload (and power consumption) increases proportionally with the number of channels enabled.

In order to address these challenges in LPWAN systems, we propose an adaptive LPWAN receiver that dynamically adjusts the system bandwidth continually, and periodically transmits the new bandwidth setting to end nodes through the use of a downlink beacon. This case study implements the physical layer signal processing for such an adaptive receiver. The implemented architecture consists of reconfigurable channelizer and baseband processing algorithms. In this chapter,

we build on the dynamically reconfigurable channelizer presented in [34]. In that work, a parameterized channelizer was designed for a reconfigurable platform and the channelizer was created with multiple implementation options optimized for different use cases and applications. An MDP was employed to determine reconfiguration decisions at runtime and was shown to produce a dynamic system that was more robust to changing environments and use cases when compared to the prior state of the art.

Our work on this case study and the underlying HMCSM framework developed in this chapter goes beyond the developments of [34] by (1) our application of hierarchical MDP techniques; (2) integrating MDP-based control with model-based methods for signal processing system implementation; (3) expanding the role of the MDP to control not just the channelizer configuration, but also the system bandwidth; and (4) comparing a baseline MDP framework with our proposed hierarchical MDP approach that reduces the model size.

### 4.4.3 Application Specification

Figure 4.2 shows a block diagram of the adaptive receiver architecture that is investigated in this case study. The channelizer is used to separate the incoming wideband signal into multiple data streams, where each of the data streams is associated with a distinct channel. Each channel is then oversampled for symbol timing recovery, and then processed by a Generalized Likelihood Ratio Test (GLRT) detector, which looks for the transmission preamble. Once a detection is successful,



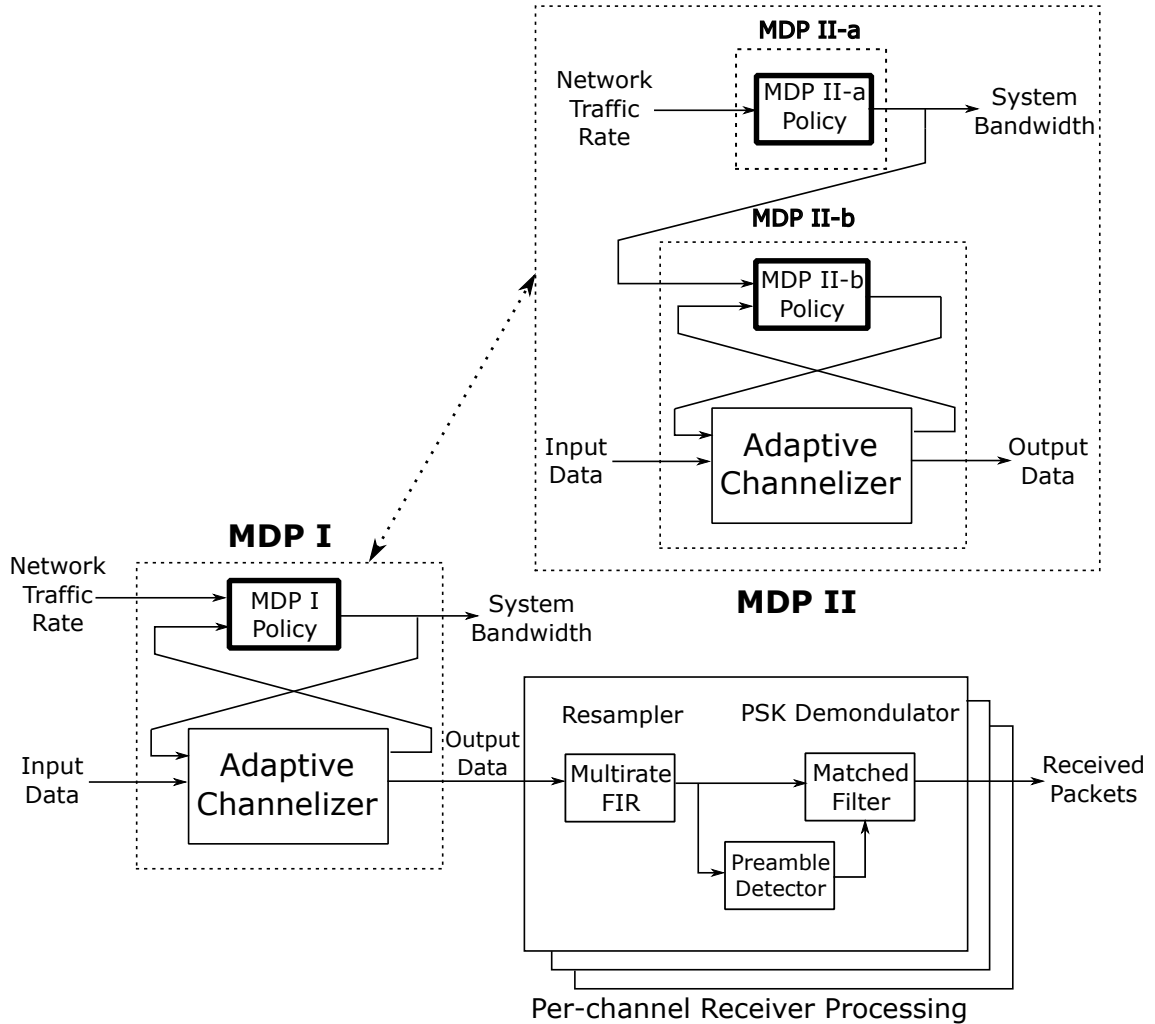


Figure 4.2: Block diagram of receiver signal processing and two MDP schemes.

a matched filter demodulator recovers the transmitted data and confirms it with an error detection function (e.g., CRC32).

In our case study, we compare the relative merits of two separate MDP schemes, labeled MDP-I and MDP-II. These two schemes are illustrated together in Figure 4.2. MDP-I consists of a single MDP employed for control of both the dynamic bandwidth as well as the channelizer processing configuration. MDP-II splits the modeling into two MDPs arranged in a hierarchy. These two MDPs are labeled MDP-II-a and MDP-II-b.

The channelizer can be implemented by one of two options, as detailed in [34] — a bank of  $M$  polyphase decimators and mixers (labeled as DCM[1], DCM[2], ..., DCM[ $M$ ]) or a Discrete Fourier Transform Filter Bank (labeled as DFTEFB). Both options are capable of meeting the processing requirements of the channelizer subsystem. However, they do so using different algorithmic means, and the DCM is generally a more power-efficient implementation when a relatively small number of channels is enabled. Conversely, the DFTEFB is more efficient when a higher number of channels is enabled. The exact crossover point of efficiency is highly platform- and implementation-dependent. This is one of the primary advantages to the MDP-based technique: the reconfiguration policy is optimized for the exact processing characteristics of a specific platform (e.g., measured power consumption). This will in general lead to an application-specific, MDP-generated control policy for one system that is different from that of another system where the efficiency crossover of the DCM vs. DFTEFB occurs at a different point.

#### 4.4.4 PSDF Model

Since the reconfiguration in our adaptive receiver system is focused on the channelizer subsystem, we implement this subsystem as a PSDF design with dynamic parameters. Our PSDF model of the channelizer system is illustrated in Figure 4.3. Here,  $M$  is a static parameter of the application that represents the total number of available channels.

At run-time, the MDP-generated reconfiguration policy determines how many

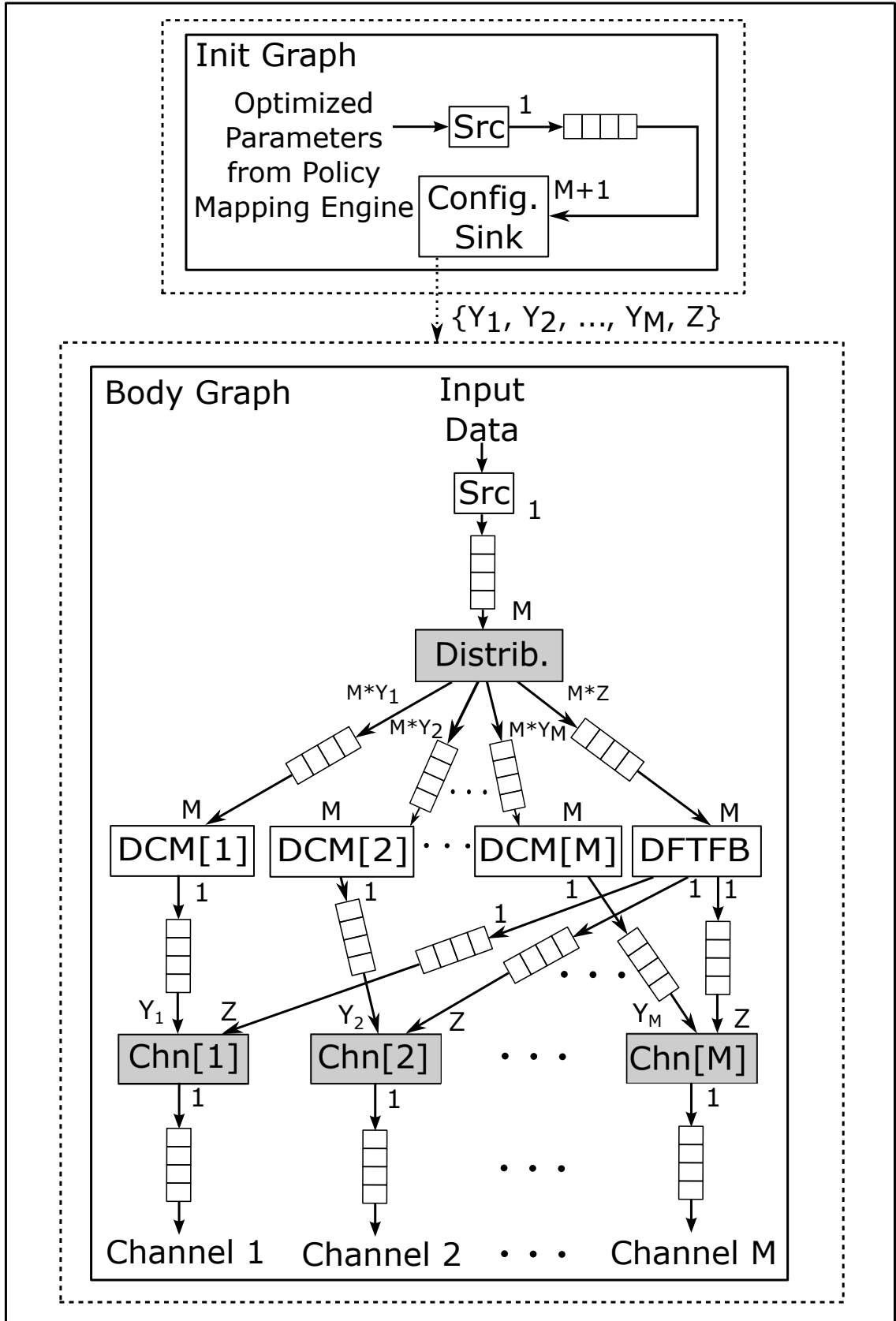


Figure 4.3: PSDF specification of channelizer subsystem.

channels to enable, based on the data rate, and whether to apply DCM processing or DFTFB processing. These policy decisions are used to manipulate a set of dynamic parameters  $\{Y_1, Y_2, \dots, Y_M\}$  that is associated with  $M$  distinct DCM actors  $\text{DCM}[1], \text{DCM}[2], \dots, \text{DCM}[M]$  (one DCM actor for each channel). The policy decisions are also used to manipulate a parameter  $Z$  that is associated with an actor labeled DFTFB, which represents DFTFB processing on all of the enabled channels. Each of these  $(M + 1)$  parameters is binary valued — that is, it takes on values only within the set  $\{0, 1\}$ . In particular, for each  $i \in \{1, 2, \dots, M\}$  if DCM processing is enabled, and the  $i$ th channel is enabled, then  $Y_i = 1$ . Conversely, if DFTFB processing is enabled, then  $Z = 1$ , and  $Y_i = 0$  for all  $i$ . If DCM processing is enabled, then  $Z = 0$ .

After each reconfiguration round, updated values of  $\{Y_1, Y_2, \dots, Y_M\}$  and  $Z$  are propagated to the adaptive channelizer flowgraph through the Init Graph, as illustrated in Figure 4.3. The channelizer flowgraph is parameterized in terms of these  $(M + 1)$  dynamic parameters and encapsulated within the Body Graph, as shown in the figure. The Init Graph and Body Graph are modeling constructs in PSDF that are used, respectively, for reconfiguration functionality (determination and propagation of new parameter values), and core signal processing functionality associated with an application. For details on the semantics of PSDF and operation of the Init and Body Graphs, we refer the reader to [40].

The Config. (configuration) Sink actor in Figure 4.3 is a special actor that we use in LIDE to propagate parameter updates from the Init Graph to the Body Graph in PSDF-based implementations. The shaded actors in the Body graph are

dynamically parameterized actors. The expressions next to the input and output ports represent the consumption and production rates associated with the ports, respectively. The Distrib. (distributor) actor takes its input data and distributes copies of it to the appropriate subset of DCM/DFTFB actors depending on the current values of the dynamic parameters in the graph.

For example, suppose that DCM processing is enabled, and that Channels 1, 2, and 3 are enabled. Then  $Y_1 = Y_2 = Y_3 = 1$ ,  $Y_j = 0$  for  $j > 3$ , and  $Z = 0$ . This means that the production rates on the three leftmost output ports of the Distrib. actor will be  $M$ , while the production rates on all of the other ports of this actor will be 0. Similarly, the consumption rates on the left-side input ports of Chn[1], Chn[2], and Chn[3] are identically equal to 1, while all of the other input ports across the bank of Chn actors have consumption rates that are identically equal to 0.

The actors labeled Chn[1], Chn[2],  $\dots$ , Chn[ $M$ ] in Figure 4.3 simply copy data from their input ports to their output ports based on which (if any) input ports are “active” (have nonzero consumption rate). These actors generate samples on each of the  $M$  output channels of the channelizer subsystem.

## 4.5 Experiments

In this section, we discuss our experiments and results based on the case study developed in Section 4.4.

## 4.5.1 Comparison of Alternative MDP Configurations

As mentioned in Section 4.4, we compare the relative merits of a conventional, monolithic MDP approach with our proposed hierarchical MDP approach to adaptive signal processing system design. The conventional and hierarchical approaches are labeled MDP-I and MDP-II, respectively. In both cases, the MDP formulations require suitable definitions for the core MDP components: the state space, action space, STMs, and reward function. These definitions are detailed as follows.

### 4.5.1.1 MDP-I

In MDP-I, the (single) MDP state space consists of the instantaneous rate of uplink packets generated by all end nodes, and the configuration of the basestation processor (number of channels enabled and channelizer configuration).

The action space consists of the next configuration of the basestation processor (number of channels enabled and channelizer configuration).

The STM for the packet generation rate is computed a priori from the traffic rate measured in a design-time simulation. The STM for the processing system is obtained from the dynamics of the implementation on the reference platform.

The MDP reward metric is a linear combination of two competing metrics: the probability of packet collision and the power consumption expended in basestation processing. This linear combination of conflicting metrics tasks the MDP with finding an optimal trade-off at any time based on relative weightings that are provided by the designer for the two metrics.

### 4.5.1.2 MDP-II

In MDP-II, the control task is split across two hierarchically arranged MDPs: MDP-II-a and MDP-II-b. The decomposition serves to separate two independent aspects of the control, namely the optimal system bandwidth (MDP-II-a) and the optimal configuration to implement a specified system bandwidth (MDP-II-b). MDP-II-a is used to determine the optimal number of channels to enable at a given time, while being agnostic to what processing configuration is actually used in the receiver to implement that configuration. MDP-II-b is used to determine the optimal processing configuration for an exogenously specified number of channels.

## 4.5.2 Implementation

The configurable components of the processing system were implemented using the method detailed in Section 4.3, and deployed to a Raspberry Pi 3 Model B computing platform. Each of the available configurations was run in a test mode, and the average processing power consumed was measured and tabulated as shown in Table 4.1. The device we used for measuring power consumption is the Tektronix Keithley Series 2280 Precision Measurement DC Power Supply. The measurements were then provided as input to the MDP models, and used to generate control policies using these empirically measured characteristics of the processing system.

Table 4.1: Platform measurements.

Processing Configuration	Num Channels Processed	Average Power
DCM	1	1.4406 W
DCM	2	1.4781 W
DCM	3	1.5203 W
DCM	4	1.5660 W
DCM	5	1.6025 W
DCM	6	1.6524 W
DCM	7	1.7013 W
DCM	8	1.7453 W
DFTFB	8	1.6754 W

### 4.5.3 Simulation Results

We performed a physical layer signal processing simulation in MATLAB to generate uplink traffic from 1,000 end nodes. The simulation compared the use of the two (adaptive) MDP Schemes and also (non-adaptive) cases where only the fixed channelizer configurations were used. Each run of the simulation generated results of the form shown in Figure 4.4. This figure shows results from one of the runs where the MDP was used. In this figure, the MDP (MDP-I) is in control of the number  $R$  of receiver channels enabled, and makes decisions in response to the exogenous end node traffic rate as it is incoming. Note that when the traffic rate is high, the system increases  $R$  in order to reduce packet collisions. Also notable is that the MDP jumps from 5 channels to 8 (skipping the cases  $R = 6$  and  $R = 7$ ) during peak hours. This is a result of the processing characteristics in Table 4.1 — the MDP determines that it is preferable (with respect to both performance metrics) to transition directly from  $R = 5$  to  $R = 8$ .

In our experiments, the two competing MDP schemes, MDP-I and MDP-II,



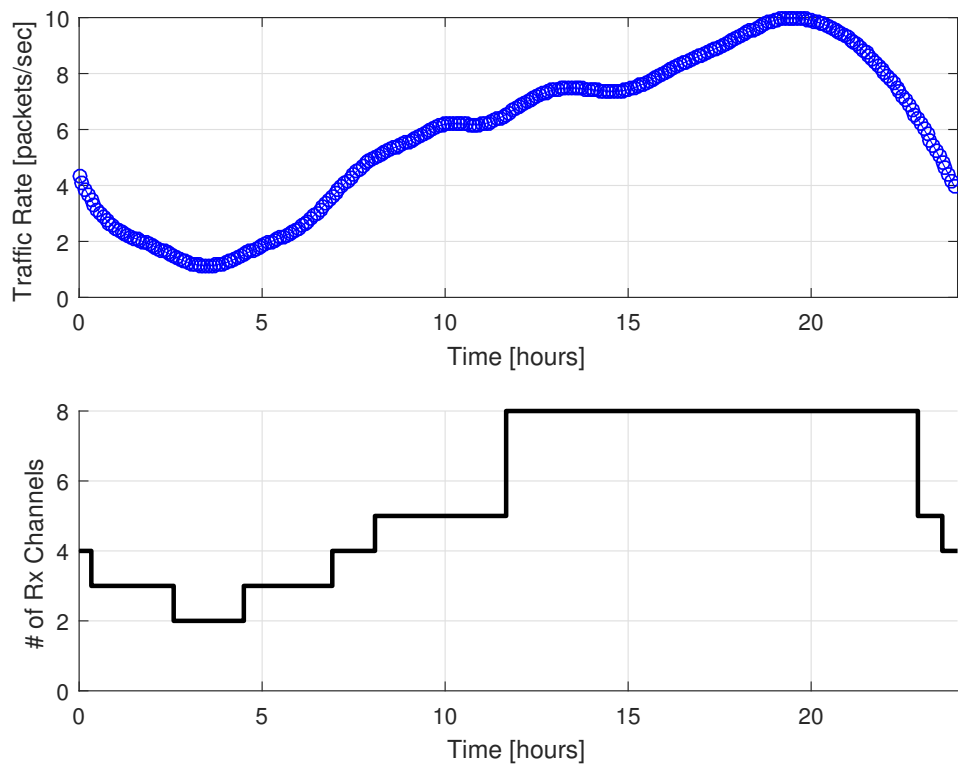


Figure 4.4: Simulation results for MDP-I.

produced the same control policy. However, a key difference is that the hierarchical MDP reduced the model size from 1.63MB to 265kB, which is a factor of 6.1 times smaller than the original size. This reduction becomes especially relevant when housing the MDP model and policy generation code on the processing platform itself. Such an *embedded MDP* realization is useful because it allows the MDP and generated policy to adapt dynamically based on learned characteristics of the operating environment. Integration of embedded MDP techniques into the HMCSM framework is a useful direction for future work.

All of the simulation runs that we carried out are compared in Figure 4.5. Different simulations for the MDP-generated policy were carried out using different relative weightings for the two performance metrics. Simulations were also carried out for fixed signal processing configurations. The square-shaped points in Figure 4.5 correspond to DCM-based channelizer operation with the number of enabled channels ranging from 1 to 8. The fixed configuration DFTFB case is represented by the triangle-shaped point in the top left region of the figure. In the context of all of the design points evaluated and the two performance metrics, we see from the figure that the MDP-based approach generates a Pareto front. This demonstrates that the adaptive reconfiguration capability provided by the MDP leads to better performance in comparison with fixed configurations for each of the available signal processing algorithms.

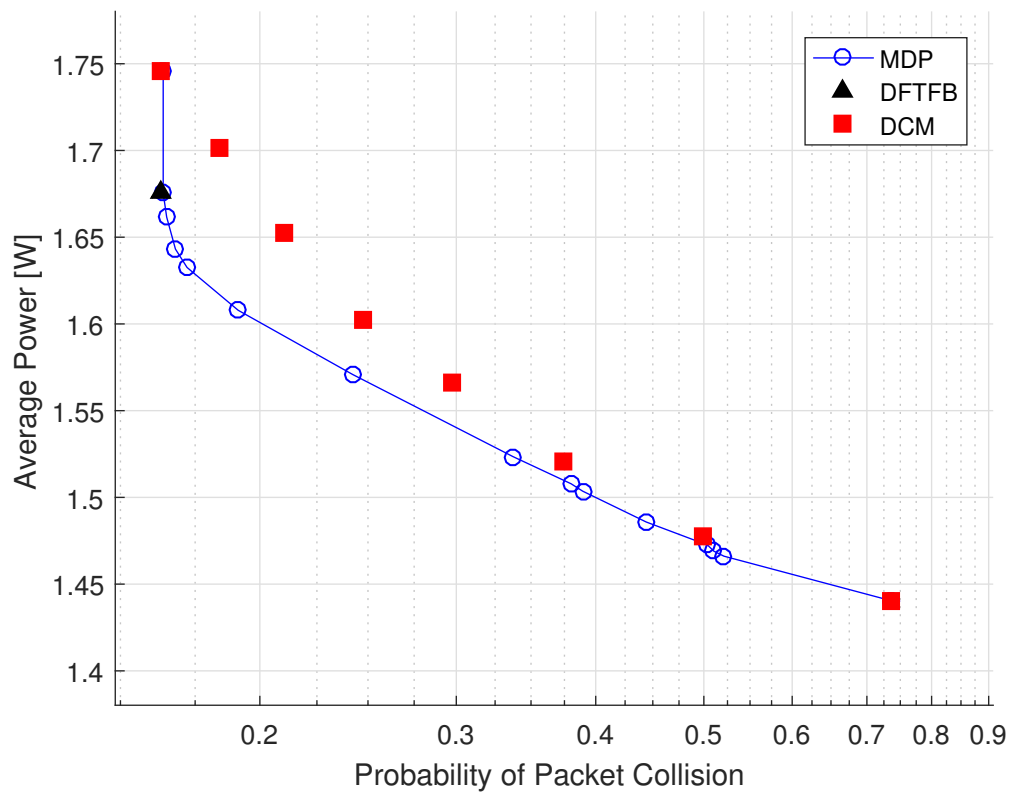


Figure 4.5: Comparison among MDP-generated policies and fixed-configuration designs.

#### 4.5.4 Solver Running Time

The Hierarchical MDP also provides a benefit in the execution time required for the MDP solver to compute the optimal policy. In our experiments, we applied the MATLAB-based open source solver called MDPSOLVE [44]. The execution times were measured as 294ms for MDP-I, 50.8ms for MDP-II-a and 41.5ms for MDP-II-b. A key feature of the Hierarchical MDP is that the stochastic model of the changing external environment is captured in MDP-II-a, and the model of the processing system is captured separately in MDP-II-b. As a result, in a deployment with a fixed processing system that periodically re-computes the control policy in response to a changing external environment, the hierarchical MDP scheme reduces the solver time from 294ms to 50.8ms, which is a factor of over 5.7X smaller.

#### 4.6 Summary

In this chapter, we have developed the Hierarchical MDP framework for HMCSM and its application to design and implementation of adaptive embedded signal processing systems. HMCSM provides a structured design methodology that integrates model-based design for embedded signal processing in terms of dataflow methods; MDP formulation using compact, hierarchical models; optimal policy generation from these models at design time; and dynamic, system-level reconfiguration at run time. The framework enables systematic derivation of system-level reconfiguration policies that are based on application-specific functional requirements and operational constraints. We have demonstrated the utility of HMCSM concretely

through a case study involving an adaptive receiver for wireless communications. Useful directions for future work include adapting the HMCSM framework for use with other parametric dataflow models (beyond PSDF), development of tools to help automate the factoring and hierarchical decomposition of MDPs, and integration of embedded MDP techniques into HMCSM.

## Chapter 5

# An Integrated Hardware/Software Design Methodology for Signal Processing Systems

In the previous chapters, we proposed two frameworks for design space exploration of signal processing systems, with an emphasis on embedded software implementation. This chapter presents a new methodology for design and implementation of signal processing systems on system-on-chip (SoC) platforms, which may in general include a mix of software and hardware (field programmable gate array) subsystems. The methodology is centered on the use of lightweight application programming interfaces for applying principles of dataflow design at different layers of abstraction.

The development processes integrated in the approach of this chapter are software implementation, hardware implementation, hardware-software co-design, and optimized application mapping. The proposed methodology facilitates development and integration of signal processing hardware and software modules that involve heterogeneous programming languages and platforms.

As a demonstration of the proposed design framework, we present a dataflow-based deep neural network (DNN) implementation for vehicle classification that is streamlined for real-time operation on embedded SoC devices. Using the proposed methodology, we apply and integrate a variety of dataflow graph optimizations that

are important for efficient mapping of the DNN system into a resource constrained implementation that involves cooperating multicore CPUs and field-programmable gate array (FPGA) subsystems. Through experiments, we demonstrate the flexibility and effectiveness with which different design transformations can be applied and integrated across multiple scales of the targeted computing system.

A preliminary, partial version of the material in this chapter has been published in [46, 47]. The work in this chapter was developed through a collaboration across different institutions — the University of Maryland, USA; University of Cagliari, Italy; University of Sassari, Italy; and Tampere University of Technology, Finland. My contribution to the work presented in this chapter has been focused on the dataflow-based design methodologies, software design and optimization, and the overall design space exploration in the case study of the DNN system.

## 5.1 Introduction

Model-based design has been widely studied and applied over the years in many domains of embedded processing. Dataflow is well-known as a paradigm for model-based design that is effective for embedded digital signal processing (DSP) systems [3]. Many dataflow-based design methods for DSP systems have been explored in recent years to support various aspects of design and implementation, including modeling and simulation; scheduling and mapping of actors to heterogeneous platforms; and buffer management (e.g. see [3, 48]).

The diversity of design scales and dataflow techniques that are relevant to sig-

nal processing systems poses major challenges to achieving the fully potential that is offered by signal processing platforms under stringent time-to-market constraints. While automated techniques, such as those referred to above for scheduling and buffer mapping, are effective for specialized combinations of platforms and dataflow models (e.g., multicore CPUs and synchronous dataflow, respectively), they are limited in their ability to support more comprehensive assessment of the design space, where the models and target platforms themselves have great influence on addressing implementation constraints and optimization objectives. System designers must therefore resort to ad-hoc methods to explore design alternatives that span multiple implementation scales, platform types, or dataflow modeling techniques.

In this work, we propose a design methodology and an integrated set of tools and libraries that are developed to help bridge this gap. We refer to this methodology as the STMC Methodology or STMCM, which is named after the different institutions across which it is developed (Sassari, Tampere, Maryland, Cagliari). STMCM focuses on enabling experimentation across different levels of abstraction throughout the design process, and allowing designers to experiment productively and iterate rapidly on complex combinations of design options, including dataflow models, heterogeneous target platforms, and integration with platform-specific languages and back-end tools. Special emphasis is placed on enabling effective experimentation with hardware/software design trade-offs, as well as trade-offs involving performance, resource utilization, and power consumption. These are trade-offs that are especially important and challenging to navigate efficiently in design processes for system-on-chip implementation of signal process systems.



The utility of STMCM is facilitated by the use of lightweight dataflow (LWDF) programming [22], and its underlying core functional dataflow (CFDF) model of computation [49]. LWDF provides a compact set of application programming interfaces (APIs) that allows one to apply signal-processing-oriented dataflow techniques relatively easily and efficiently in the context of existing design processes, target platforms, and simulation- and platform-oriented languages, such as MATLAB, C, CUDA, and VHDL. Additionally, CFDF is a general form of dataflow that accommodates more specialized forms of dataflow, such as Boolean dataflow [50], cyclo-static dataflow [37], synchronous dataflow [39], and RVC-CAL [51] as natural special cases. This accommodation of different dataflow models in turn provides potential to integrate designs with other dataflow frameworks and DSP libraries, such as those described in [51, 52, 53, 54, 55, 56]. Furthermore, LWDF is granularity-agnostic, in the sense that actor complexity does not limit the applicability of the framework.

To demonstrate the capabilities of STMCM in addressing the challenges of mapping practical dataflow-based structures on heterogeneous signal processing platforms, we explore different implementations of a deep neural network (DNN) for vehicle classification on an heterogeneous, embedded system-on-chip (SoC), the Xilinx Zynq Z-7020 SoC. DNN applications pose great challenges in their deployment on embedded devices. Investigation of DNN implementations on embedded SoC devices is challenging due to the limited resources for processing and storage in these devices, and especially, due to the high computational complexity of DNNs. They involve very large and complex signal flow structures that involve intensive computation, data exchange, and multi-layer processing. These characteristics make

embedded DNN implementation highly relevant as a case study for STMCM.

## 5.2 Related Work

Dataflow provides valuable model-based design properties for signal processing systems, and has been adopted in a wide variety of tools for both software and hardware design. For example, LWDF APIs for CUDA and C have been targeted in the DIF-GPU tool for automated synthesis of hybrid CPU/GPU implementations [57]. The CAL programming language and the Open RVC-CAL Compiler (Orcc) toolset provide a dataflow environment for generating dataflow implementations in a number of languages, such as C, Jade, and Verilog [51, 52, 58] (note that the Verilog backend of Orcc has been discontinued and Xronos synthesizer has been replaced). The CAPH language and framework generate hardware description language (HDL) code from high-level dataflow descriptions [53]. The work in [59] presents an integrated design flow and tools for the automatic optimization of dataflow specifications to generate HDL designs. The Multi-Dataflow Composer (MDC) tool is a dataflow-to-hardware framework able to automatically create multi-functional reconfigurable architectures. In addition to this baseline functionality MDC offers three additional features: (1) a structural profiler to perform a complete design space exploration, evaluating trade-offs among resource usage, power consumption and operating frequency [60]; (2) a dynamic power manager to perform, at the dataflow level, the logic partitioning of the substrate to implement at the hardware level, and apply a power saving strategy [61]; (3) a coprocessor generator to perform the complete

dataflow-to-hardware customization of a Xilinx compliant multi-functional IP [59].

All of the methodologies and tools described above are limited by the programming language, adopted dataflow description, or implementation target. For example, HDL code can be highly optimized for a given target (such as a Xilinx FPGA) but not usable for an application specific integrated circuit (ASIC) flow (e.g., see [58, 62, 63]). Automatic methods and tools require significant effort in development and maintenance of graph analysis and code generation functionality, and may be too costly for models and design approaches that are not mature. Such scenarios may arise for emerging applications or platforms that do not match effectively with the models or methods supported by available tools.

STMCM is complementary to these efforts that emphasize dataflow design automation. By applying LWDF APIs in novel ways, STMCM facilitates implementation of and iterative experimentation with new dataflow-based hardware/software architectures and design optimization techniques. LWDF is applied as an integral part of STMCM because of LWDF's minimal infrastructure requirements and its potential for rapid retargetability to different platforms and actor implementation languages. Furthermore, LWDF does not have any restriction in terms of actor granularity and can be extended with different combinations of dataflow graph transformations, as well as other forms of signal processing optimizations (e.g., see [3]).

In [46], we presented an efficient integration of the LWDF methodology with hardware description languages (HDLs). Building on this HDL-integrated form of LWDF, we developed methods for low power signal processing hardware implementation, and system-level trade-off exploration. In this chapter, we apply the hard-

ware design techniques introduced in [46] as part of a general methodology that spans software, hardware, and mixed hardware/software design, implementation, and trade-off exploration. Thus, while the focus in [46] is on rigorous integration across digital hardware design, lightweight dataflow programming, and low power optimization, the emphasis in this work is on a methodology for applying LWDF concepts in an integrated manner across complete hardware/software development processes for embedded signal processing systems.

In summary, STMCM provides methods to seamlessly and comprehensively integrate LWDF-based actor implementation techniques with design processes for real-time, resource-constrained signal processing systems. STMCM can be used as an alternative to or in conjunction with more conventional automated dataflow tools (e.g., for disjoint subsystems). STMCM requires more effort in programming compared to fully automated toolchains, however it provides more agility in terms of retargetability and experimentation, as described above. This is a useful trade-off point to have available for model-based design of complex signal processing systems.

### 5.3 Proposed Design Methodology

Our proposed methodology STMCM is illustrated in Figure 5.1. As motivated in Section 5.1 and Section 5.2, STMCM is a design methodology that emphasizes LWDF concepts, and is specialized for SoC-based signal processing systems. The upper part of Figure 5.1 represents application-specific and algorithmic aspects, while the lower part represents the general part of the methodology that is reusable

across different applications. The upper part is illustrated concretely in the context of DNN system design; this part can be replaced with other application/algorithm level design aspects when applying STMCM to other applications.

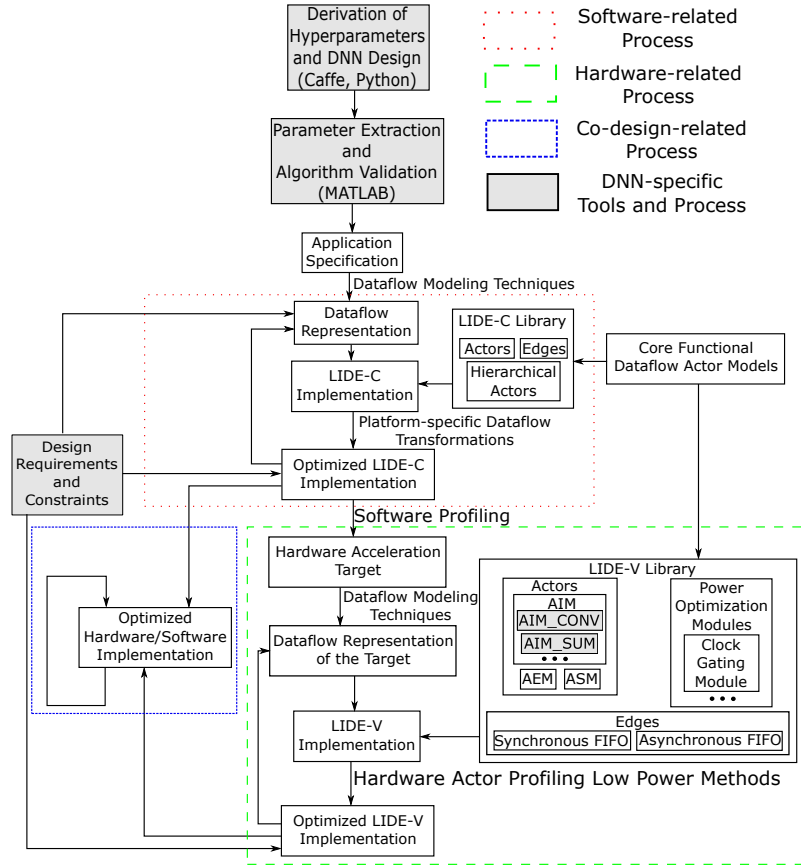


Figure 5.1: An illustration of STMCM in the context of DNN system design.

### 5.3.1 Overview of Dataflow Models

*Enable-invoke dataflow (EIDF)* is a general dataflow model of computation that supports dynamic dataflow behavior in actors [64]. In EIDF, each actor is divided into a set of *modes*, where each mode, when it executes, has static dataflow rates — i.e., each mode has a fixed *consumption rate* and *production rate* associated with each input and output port, respectively. Dynamic dataflow behavior can be

achieved by switching among different modes of the same actor that have different dataflow (production or consumption) rates associated with the same port. The firing rule for a given EIDF actor is dependent on the current mode  $M$  of the actor. Intuitively, this mode-dependent firing rule is that there must be sufficient data on the actor input buffers (as determined by the fixed consumption rates associated with  $M$ ), and sufficient vacant space on the actor output buffers (as determined by the production rates associated with  $M$ ). For more details on the semantics of EIDF, we refer the reader to [64].

The specification of an EIDF actor includes a method called the *enable method*, which checks whether there is sufficient data available on the actor's input ports and sufficient data available on its output ports to fire the actor in its current mode. The enable method returns a Boolean result that is true-valued when the aforementioned availability conditions are met. Each EIDF actor also has an *invoke* method, which executes the actor operation according to its current mode, consumes and produces data on actor ports with the fixed dataflow rates of the current mode, and returns a set of admissible *next modes* that can be used for the next firing of the actor. Any mode in the set of next modes can be further checked for readiness by the enable method, and invoked once the enable method returns true. Since the next mode for firing is not uniquely specified (in cases where the admissible set of next modes has more than one element), EIDF can be used to specify non-deterministic behaviors.

CFDF, which the LWDF programming approach is based on, is a special case of the EIDF model. In CFDF, the set of admissible next modes for an actor has exactly one element. In other words, the invoke method only returns one valid mode

of execution, which means that the sequence of executed actor modes proceeds down a single deterministic path.

A key feature of EIDF and CFDF is a clean separation of the enable and invoke capabilities. Once the invoke function is enabled, it assumes that sufficient data is available to execute the actor operation associated with the current mode — the firing condition is assumed to have already been checked by the corresponding enable function, through appropriate compile-time analysis, or through some combination (e.g., using quasi-static scheduling analysis). This feature improves the predictability of actor firings and facilitates efficient scheduling techniques. The modeling approach imposes restrictions (“dataflow design rules”) only on the structure and interface of dataflow actors instead of on details of actor implementation, which in turn facilitates retargetability of actors that are developed using the approach.

#### 5.3.1.1 Software-based Actor Implementation Using LWDF

For software implementation, an LWDF actor is implemented as an abstract data type that has four interface functions, which are referred to as the *construct*, *enable*, *invoke*, and *terminate* functions. The *construct* and *terminate* functions can be viewed as an object-oriented constructor and destructor for instantiation and removal of actors, respectively. The *enable* and *invoke* functions in LWDF provide concrete mechanisms for implementing in software the corresponding functions of the same name from the abstract CFDF semantics. The LWDF *enable* function returns a Boolean value; the returned value is `true` if (1) there is sufficient data

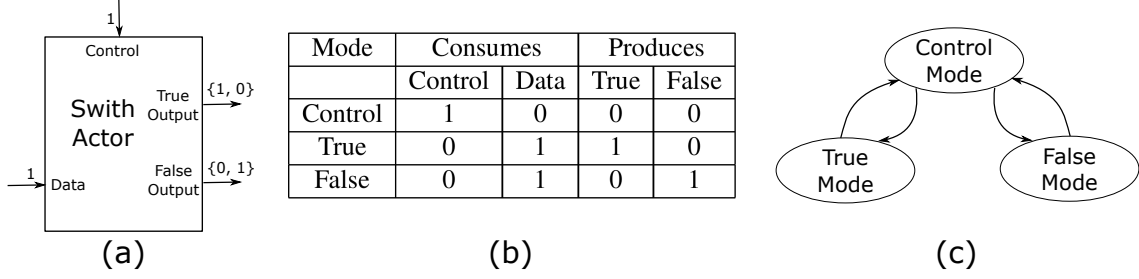


Figure 5.2: Switch actor in CFDF. (a) Switch Actor, (b) Dataflow Table, (c) Mode Transition Diagram between CFDF Modes.

on the actor’s input edges to execute the current mode; and (2) the output edges of the actor have sufficient empty space to accommodate the data that would be produced if the next mode were to be executed. This formulation makes sense in LWDF because of the restriction in CFDF semantics that the dataflow rate on a given port is constant for a given mode.

The invoke function of an LWDF actor  $a$  carries out a single firing of  $a$  according to its current mode; determines the next mode for  $a$ ; and changes the current mode of  $a$  to be equal to this newly-determined next mode just before returning.

We present the switch actor as an example of CFDF actor. Switch actor has three modes: Control, True and False. In Control mode, the switch actor consumes one token from Control port. In True or False mode, the switch actor consumes one token from Data port and forward that token to True or False Output port accordingly. The dataflow table and mode transition diagram between CFDF modes of switch actor are illustrated in Figure 5.2.

The enable and invoke functions provide interfaces for implementing schedulers to coordinate execution of dataflow graphs that are implemented using LWDF. A broad class of schedulers can be implemented using these interfaces, including many



types of static, quasi-static, and dynamic schedulers (e.g., see [49, 65]). Static schedules are generated at compile time and specify fixed sequences of actor firings, thus, the actors can be directly invoked by static schedulers without checking the firing conditions using enable functions. Quasi-static schedules are generated at compile-time, but may contain code for performing some data-dependent, scheduling-related computations at run-time. Dynamic schedulers schedule dynamic dataflow applications in ways that involve relatively large amounts of run-time decision-making.

A simple example of a dynamic scheduler is a canonical CFDF scheduler [49]. A canonical scheduler  $S$  calls the enable functions of the actors in some order in a round robin fashion. Each time the enable function of an actor  $A$  is called by  $S$  and the function returns “true”, the invoke function of  $A$  is immediately executed. This process of visiting and conditionally invoking actors is repeated until no actors are enabled or some other termination condition of the application is satisfied.

### 5.3.1.2 LWDF FIFO Implementation

As with actor design, LWDF provides a compact set of interfaces for implementing the FIFO buffers that correspond to dataflow graph edges. These interfaces provide standard functions that are used in LWDF-based actors and schedulers to work with FIFOs. Details of the implementation are unspecified so that designers have full flexibility in developing and applying different FIFOs in different applications or in different parts of the same application to achieve desired trade-offs in inter-actor communication performance (e.g., by mapping dataflow edges into dif-

ferent types of memories). LWDF is formulated to orthogonalize FIFO, actor, and scheduler implementation so that, for example, modifications to or replacement of a FIFO implementation do not require modifications to actors that communicate with the FIFO or scheduling logic that coordinates execution of the graph. For general background on the utility of orthogonalization in system-level design, we refer the reader to [66]. LWDF interface functions defined for FIFOs include functions for construction and termination (as with actors); reading tokens from FIFOs; writing tokens into FIFOs; and checking their token populations and amounts of available free space.

In STMCM, we apply the LWDF programming model through the Lightweight Dataflow Environment (LIDE). LIDE is a software tool for dataflow-based design and implementation of signal processing systems [22, 67]. LIDE is based on a compact set of application programming interfaces (APIs) that is used for instantiating, connecting, and executing dataflow actors. These APIs have been implemented in a variety of implementation languages. For example, LIDE-C [67] and LIDE-V [46] provide C and Verilog language implementations of the LIDE APIs, respectively.

In the remainder of this section, we discuss in detail the application-, software-, and hardware-specific processes illustrated in Figure 5.1.

### 5.3.2 Application-specific Tools and Processes

In Figure 5.1, application-specific tools and associated design processes are illustrated by gray blocks. Throughout this chapter, we adopt a DNN application

as a concrete demonstration of how such application-specific aspects are used as an integral part of STMCM. The DNN-focused design process illustrated in Figure 5.1 starts with the derivation of DNN hyperparameters and the network configuration. Then the parameters associated with the derived DNN structure are extracted and the DNN algorithm is carefully validated to ensure that target levels of accuracy are satisfied.

The block labeled “Design Requirements and Constraints” refers to the application- and platform-specific requirements and constraints on the DNN implementation. Examples of these include the accuracy and throughput requirements for image classification DNN systems, and constraints on available power and hardware resources for a targeted SoC platform.

In the remainder of this section, we introduce the software-related and hardware-related design processes that provide the core of STMCM. These processes are applied in an integrated manner for hardware/software co-design, as represented by the lower left hand part of Figure 5.1. Detailed explanations of the major components in STMCM are provided in Section 5.4.3.

### 5.3.3 Software-related Process

In the next main phase of the proposed design methodology, the DNN network configuration derived using application-specific, algorithm-level tools is mapped to a software implementation using LIDE-C. LIDE-C-based implementations are not restricted to DNN systems, and can be adapted readily to facilitate the implementa-

tion and optimization of other signal processing systems. For example, in the work of [19], the design space exploration of a digital predistortion system in wireless communication is based on the LIDE-C implementation. In [27], LIDE-C has been extended to support parameterized synchronous dataflow and applied to the implementation of an adaptive wireless communication receiver. In [68], vectorization is applied to LIDE-based actors for throughput optimization. When the vectorized actor is scheduled to run on a processor that supports data-parallel execution, its throughput can be improved. For more details about LIDE-C and the development of DNN components in LIDE-C, we refer the reader to [67, 69].

Working with the LIDE-C implementation of the DNN, a number of optimization processes are carried out iteratively to streamline the software implementation in terms of the relevant design objectives and constraints. This iterative optimization process is illustrated in Figure 5.1 by the cyclic path that involves the blocks labeled *Dataflow Representation*, *LIDE-C Implementation*, and *Optimized LIDE-C Implementation*. The proposed approach supports efficient application of commonly-used DNN software optimization methods such as for-loop tiling and buffer memory sharing among dataflow graph edges. We refer the reader to Section 5.4.1 for more details about these optimization methods and the integration of them with the LIDE-C implementation.

Next, software profiling is performed on the optimized LIDE-C implementation of the DNN system to extract profiling data. This data is extracted for each dataflow component of the DNN architecture. In the profiling process applied in STMCM, the memory sizes of the buffers and execution time of the actors in the graph are

measured. According to the characteristics of DNN architecture, the DNN system is divided into multiple computation layers. In our application of STMCM, software profiling is specialized to DNN implementation by measuring the total memory sizes for the buffers both inside each layer and between pairs of adjacent layers. We also measure the total time complexity of each DNN layer.

### 5.3.4 Hardware-related Process

The dataflow model of the subgraph to accelerate is implemented in hardware using LIDE-V. Hardware profiling based on the specific implementation platform is performed on the LIDE-V implementation. This profiling is used to collect measurements on hardware performance and help identify possible optimizations. In [8], LIDE-V is applied to the implementation of digital predistortion filters and specific power estimation methods are developed to facilitate the design space exploration of the predistortion systems. Details on hardware profiling are demonstrated concretely through the case study presented in Section 5.4.2. Like the software implementation, the hardware implementation will in general go through multiple optimization iterations before it is finalized.

As discussed in Section 5.1, LWDF has primarily been targeted to DSP software implementation. In this section, we extend the general LWDF methodology for efficient digital hardware implementation. As part of this extension, we adapt the LWDF methodology based on HDL coding styles. In particular, we adapt the methodology based on the standard convention of HDL design construction in terms

of *module* definitions — e.g., as opposed to classes, methods, and functions, which are standard units of program construction in languages that are oriented to software implementation.

The design techniques developed in this section are formulated concretely in the context of the Verilog HDL. We refer to this integration of LWDF with Verilog as *LWDF-V*. The design concepts underlying LWDF-V can be adapted to other HDLs as they do not depend on specialized aspects within the Verilog language (i.e., aspects that do not have natural counterparts in other common HDLs). Such adaptation of LWDF for use with other HDLs is a useful direction for future work.

In LWDF-V, the enable, invoke and scheduling functions for an actor are implemented as three coupled Verilog modules, which we refer to as the *actor enable module* (*AEM*), *actor invoke module* (*AIM*) and *actor scheduling module* (*ASM*), respectively. Dataflow edges are implemented as *dataflow edge modules* (*DEMs*) to provide communication channels for connections between actors. Since DEMs buffer data through a first-in first-out protocol, we refer to them also simply as *FIFOs*.

Figure 5.3 illustrates an example of an LWDF-V actor.

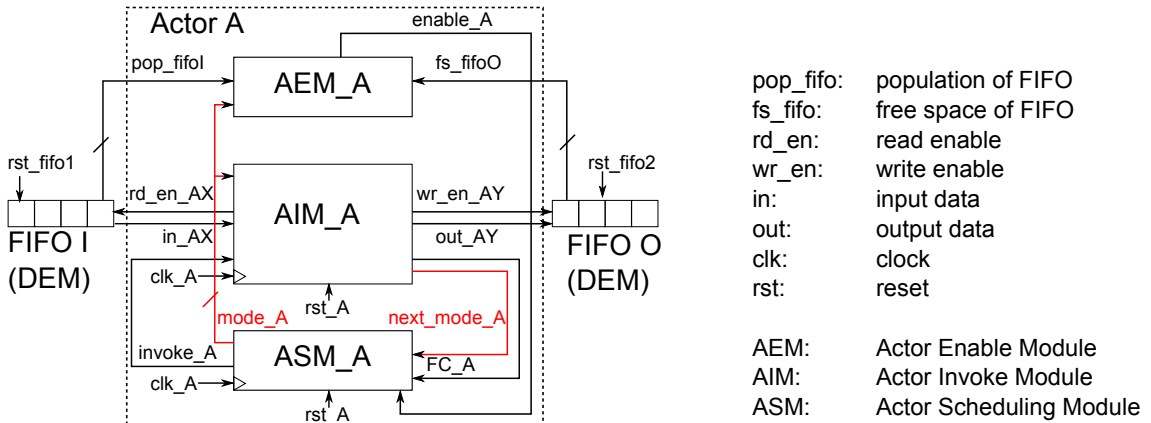


Figure 5.3: Illustration of an LWDF-V-based actor.

In the remainder of this section, we provide details on the design and interfacing conventions for the central module types — AIM, AEM, ASM, and DEM — in LWDF-V.

#### 5.3.4.1 Actor Invoke Module

Recall that LWDF imposes minimal constraints on component designs and the only requirement for an AIM is that the standard AIM operational states and interfaces (described below) are maintained. Beyond that, we allow the AIMs to be decomposed into arbitrary hierarchies of sub-modules, and described using any Verilog coding style, including behavioral, structural, or mixed behavioral/structural coding.

LWDF-V enhances the reusability and retargetability of the modules, and also facilitates evolutionary design, where sub-module designs associated with different subsystems can be progressively refined as more and more details of the targeted implementation are determined.

The high level operation of the AIM is required to have two states: the *actor idle state* and *actor firing state*, which are called the *AIM operational states* of the associated actor. The interfacing requirements of AIMs are defined in terms of these two states. The required interface ports for the AIM are divided into the following four groups.

- **Dataflow-related input ports:** This group of ports contains one input port corresponding to each input port  $X$  of the associated CFDF actor  $A$ , and a Boolean

input port called `invoke` to initiate the next firing of the actor on the next clock cycle if the actor is currently in the idle state. In Figure 5.3, examples of signals sent to ports within this group include `in_AX` and `invoke_A` .

- **Dataflow-related output ports:** This group of ports contains one output port corresponding to each actor output port `Y`, one Boolean write enable port corresponding to each output port for submitting write requests to the output edge, and one Boolean read enable port corresponding to each input port `X` for submitting read requests to the input edge. Examples in Figure 5.3 of signals sent from ports in this group include `out_AY`, `wr_en_AY`, and `rd_en_AX`.
- **Platform-related input ports:** This group of ports contains a clock input port for relevant synchronization with interfacing circuitry, and a synchronous Boolean reset input port to bring the actor to its idle state on the next clock cycle. Examples in Figure 5.3 of signals sent to ports in this group are `clk_A` and `rst_A`.
- **Control-related input ports:** For a given actor `A`, this group contains a port called `mode` that provides the current CFDF mode of the actor. In Figure 5.3, examples of signals sent to this port are `mode_A` and `FC_A` .
- **Control-related output ports:** For a given actor `A`, this group contains a port called `next_mode` that provides the next possible CFDF mode of the actor, and a port called `FC`, which stands for *firing complete*, that sends a Boolean signal to indicate when a firing of `A` completes during the current clock cycle. In Figure 5.3,



examples of signals sent from these ports include `next_mode_A` and `FC_A` .

Figure 5.4 provides an example of the FSM control flow for an AIM with three modes. The AIM stays in the current mode `x` until the firing related to the mode is completed. When this firing is completed, the `FC` signal is driven high, and the `next_mode` is suggested. In each mode, a sub-FSM controls the execution; the AIM waits in an `IDLE` state until `invoke` is high, then the state is updated to `FIRING`, where the AIM consumes input data, executes the actor operation and produces output data according to the current mode.

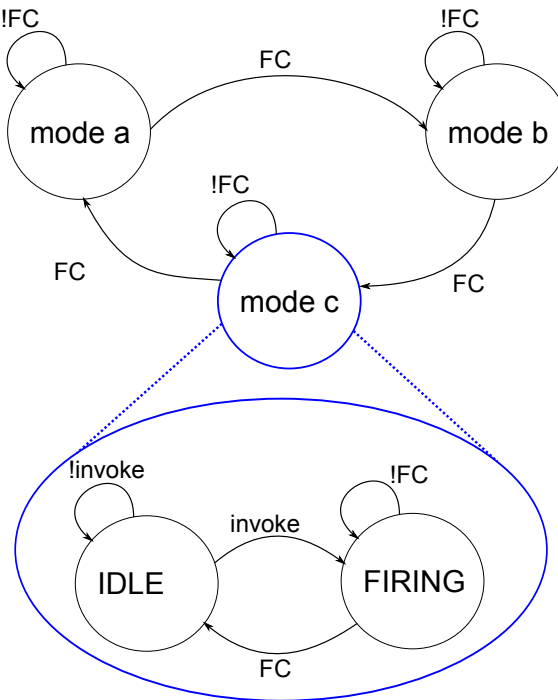


Figure 5.4: Example of an AIM FSM for a CFDF actor with three modes.

### 5.3.4.2 Actor Enable Module

To provide a standard interface for the CFDF-based enable function described in Section 5.2, the AEM module for an LWDF-V actor contains the following re-

quired interface ports.

- **Population and free space ports:** for each input port  $X$  of the associated CFDF actor  $A$ , the AEM has one input port for accessing information about the buffer state. This port provides the current buffer population (the number of tokens in the input buffer) for the FIFO  $I$  that is connected to port  $X$ . Similarly, for each output FIFO  $O$  that is connected to an output port  $Y$ , the AEM has one input port that provides the free space level (the output buffer capacity minus the population). These input ports are named, in terms of the associated FIFO names, as `pop_fifoI` and `fs_fifoO`, respectively.

Figure 5.5 shows two examples of inter-actor communication. In the first example, two actors  $A$  and  $B$  exchange data through two FIFOs. Actor  $A$  has three ports to access the buffers' state: `pop_fifo1` provides the population of `FIFO1` connected to `input 1`, and `fs_fifo2` and `fs_fifo3` provide the free space levels of `FIFO2` and `FIFO3` connected to output ports `output 1` and `output 2`. In the second example, an actor  $C$  sends output data to two actors  $D$  and  $E$  through two FIFO channels, `FIFO5` and `FIFO6`. For each output FIFO  $C$  has one free space input signal — these signals are labeled as `fs_fifo5` and `fs_fifo6`.

- **Mode port:** This input port is used to specify the CFDF mode (for the enclosing actor  $A$ ) relative to which the AEM will perform its next fireability testing operation (i.e., its operation to determine whether or not there is sufficient data and free space available to permit a firing of the actor). This port is named in terms of the enclosing actor  $A$  as `mode_A`. Figure 5.6 shows an example with three

different firing conditions  $\{C_x\}$  for three different modes `mode x`. When the firing condition is `true`, the `enable` signal is set high.

- **Enable port:** This output port is driven with the Boolean result produced by checking the fireability of the enclosing actor in the mode specified by the AEM mode port. This port is named in terms of the enclosing actor A as `enable_A`.

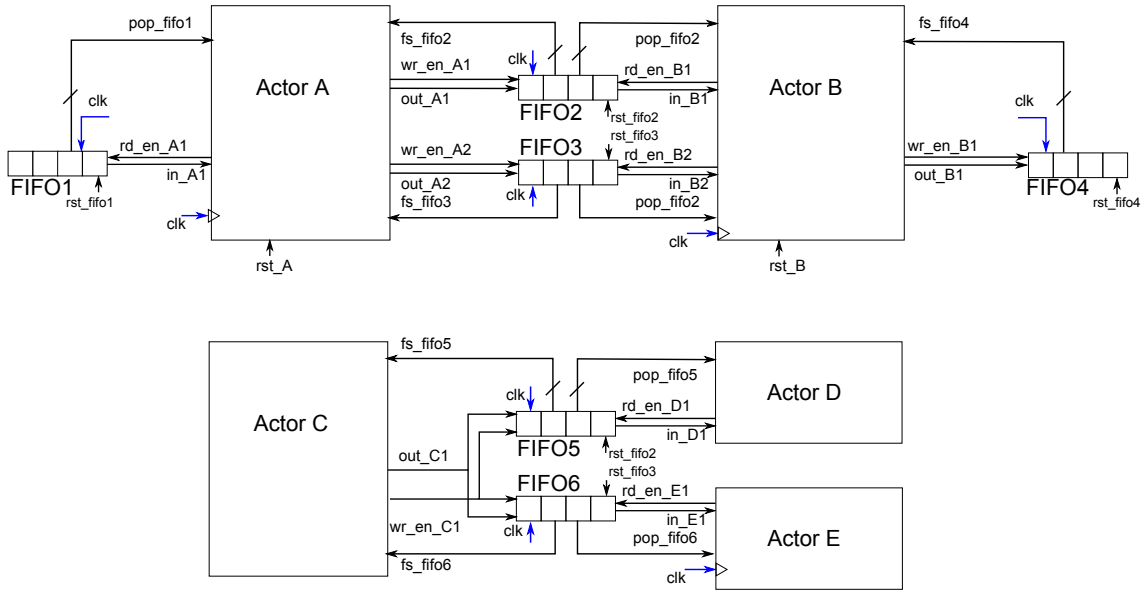


Figure 5.5: Illustration of LWDF-V-based actors communication.

mode	condition
mode a	$C_a = (\text{pop\_fifo1} \geq \text{consumption\_rate\_of\_a}) \&\& (\text{fs\_fifo2} \geq \text{production\_rate\_of\_a})$
mode b	$C_b = (\text{pop\_fifo1} \geq \text{consumption\_rate\_of\_b}) \&\& (\text{fs\_fifo2} \geq \text{production\_rate\_of\_b})$
mode c	$C_c = (\text{pop\_fifo1} \geq \text{consumption\_rate\_of\_c}) \&\& (\text{fs\_fifo2} \geq \text{production\_rate\_of\_c})$

condition	enable
true	1
false	0

Figure 5.6: Example of a AEM with three different firing condition for three possible modes.

The AEM can be implemented using combinational or sequential logic. The latter form may be preferred, for example, if large numbers of ports are involved and it is desired to share hardware resources across the comparison operations that are involved in the fireability checking process. However, for many practical actors and implementation scenarios, the number of inputs is relatively small and combinational AEM realization is a reasonable design choice. Thus, in the remainder of this chapter, we assume combinational AEM implementation. An additional subset of standard ports is used in LWDF-V to support sequential AEM implementation; further details on their implementation are beyond the scope of this chapter.

#### 5.3.4.3 Actor Scheduling Module

The ASM is an actor-level subsystem that determines the next mode of the associated actor after the actor firing is completed, and invokes the actor firing after the actor is enabled again. Compared to schedulers discussed in Section 5.2, which control groups of actors (i.e., related to specific design subsystems or to the entire digital system that is being developed), the ASM can be used to implement a *fully distributed* scheduling approach. In such an approach, an actor is scheduled to begin a new firing whenever it is idle and its enable condition is satisfied. Scheduling of LIDE-V actors is not restricted to such a fully distributed scheduling approach. For example, with appropriately-designed control logic, subsets of actors can be serialized to allow sharing of resources within the subsets. In this chapter, however, we restrict our attention to fully distributed scheduling. Fully distributed scheduling of

dataflow graphs has been analyzed in various contexts. For example, Ghamarian et al. have developed methods for throughput analysis of synchronous dataflow graphs that are scheduled in a fully distributed manner [70]. Such analysis techniques can be applied to hardware subsystems in STMCM.

The interface ports of an ASM are listed as follows. For an ASM that is associated with a given actor  $A$ , these ports are described here in relationship to the AIM and AEM of the same actor  $A$ .

- **Dataflow-related input ports:** This group of ports contains a Boolean input port that is connected to the output port `FC` of the AIM, a Boolean input port that is connected to the output port `enable` of the AEM, and an input port that is connected to the output port `next_mode` of the AIM.
- **Platform-related input ports:** This group of ports is the same as the platform-related input ports introduced in Subsection 5.3.4.1.
- **Control-related output ports:** This group contains an output port called `mode` and an output port that is connected to the input port `invoke` of the AIM. The ASM makes the decision on the current actor mode and sends the resulting mode signal to the AEM and AIM through the output port `mode`. Thus, the ASM is responsible for carrying out mode transitions of the actor after firings are completed. Generally, the ASM can either set the actor mode to be the next mode signal received from the `next_mode` input port or ignore the next mode signal and set the mode according to some user-defined logic that is integrated as part of the ASM. The ASM also launches the next actor firing once the previous firing is

completed and the actor is enabled again.

Figure 5.7 shows an example of an FSM that controls an ASM. The ASM waits in the state `WAIT_EN` for the `enable` signal to become high. When `enable = 1`, the ASM invokes the AIM (`invoke = 1`). Then the ASM waits in the state `WAIT_FC` for the firing completion (`FC = 1`), and sets the next actor mode.

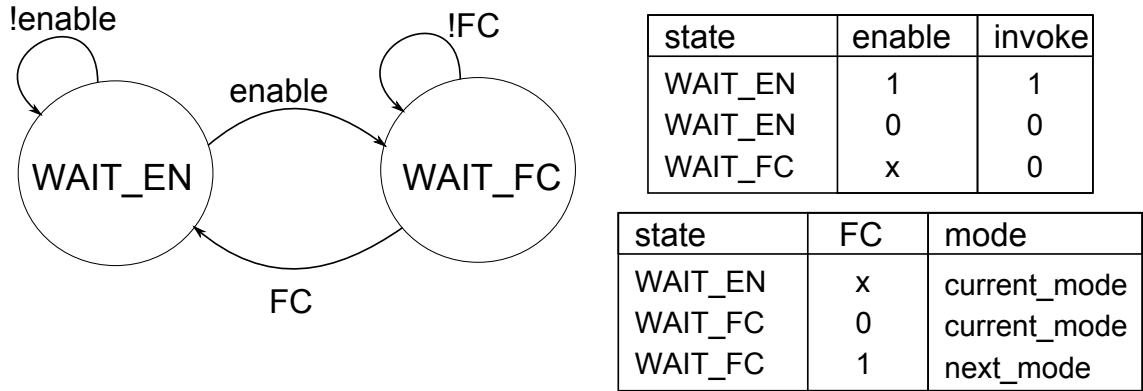


Figure 5.7: An example of an FSM that controls an ASM.

The example in Figure 5.8 illustrates temporal relationships among the `enable`, `invoke` and `FC` signals. After the `enable` signal is high, the ASM raises the `invoke` signal. The AIM then executes its operation and the ASM waits for the firing completion signal `FC`.

Here, as illustrated in Figure 5.8, we define  $T_{ei}$  to be the elapsed time between the instant when the actor becomes enabled, and when the corresponding firing is invoked. Similarly,  $T_{ic}$  is the invocation to firing completion time, and  $T_{ci}$  is the time between the firing completion for one invocation and the start of the next invocation. Finally,  $T_{ec}$  is the enable to firing completion time, and  $T_{ii}$  is the elapsed time between two successive invocations. One can, for example, measure average values for these timing characteristics across all firings of an actor within a given

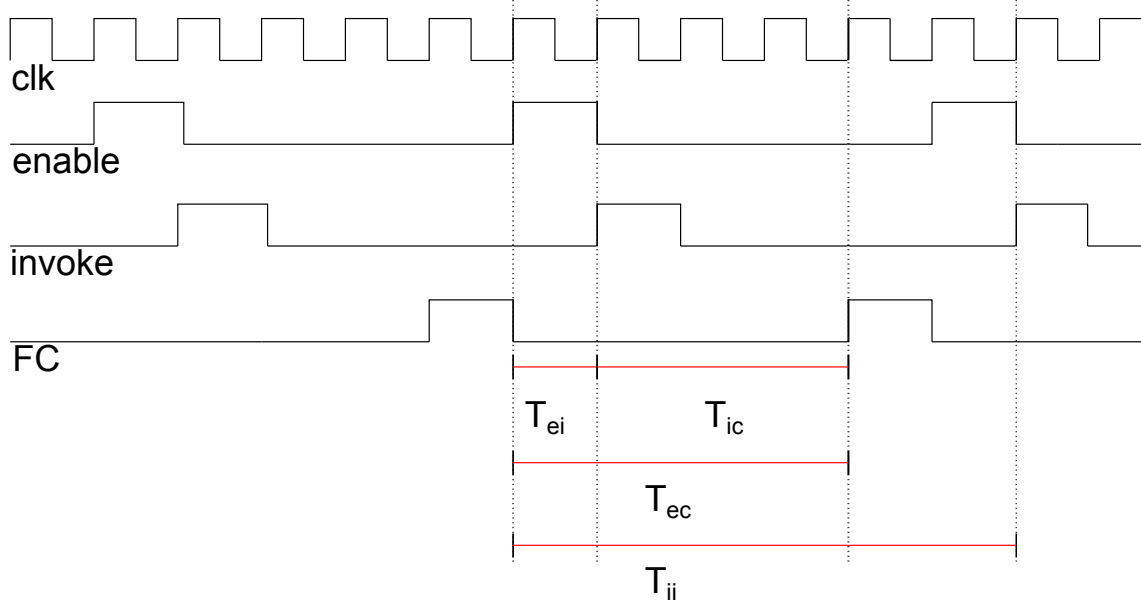


Figure 5.8: Examples of signal waveforms during execution of an LWDF-V actor.

execution of the enclosing dataflow graph. Such measurements can provide insight into the performance of the given actor in the context of the applied scheduling strategy.

#### 5.3.4.4 Dataflow Edge Module

The DEM is used in LWDF-V to implement a dataflow edge. The required interface ports include the following.

- **Enable ports:** These two Boolean input ports provide read enable and write enable signals, `rd_en` and `wr_en`, for accessing the FIFO storage.
- **Data input/output ports:** These ports, named `in` and `out`, are used by the FIFO to read input data and send output data, respectively, when a read or write operation is initiated.

- **Population port:** This output signal, named `pop`, is driven with a non-negative integer value that gives the number of tokens that is currently stored in the FIFO.
- **Free space port:** This output signal, `fs`, provides the current value of  $(c - p)$ , where  $c$  and  $p$  are the capacity and population, respectively, of the buffer.

Figure 5.9 depicts an overview of an LWDF-V-based synchronous FIFO design. Once the FIFO read or write operation is enabled, the `rd_addr` or `wr_addr` module will update the read or write pointer accordingly. The population `pop` and free space `fs` signals will be updated as well. In this chapter, we optimized the synchronous FIFO design by further streamlining the design developed in [47]. The optimized FIFO contains fewer registers while achieving the same functionality as the original FIFO design. Further details about the resource utilization of the two FIFO designs are presented in Section 5.3.4.5.5.

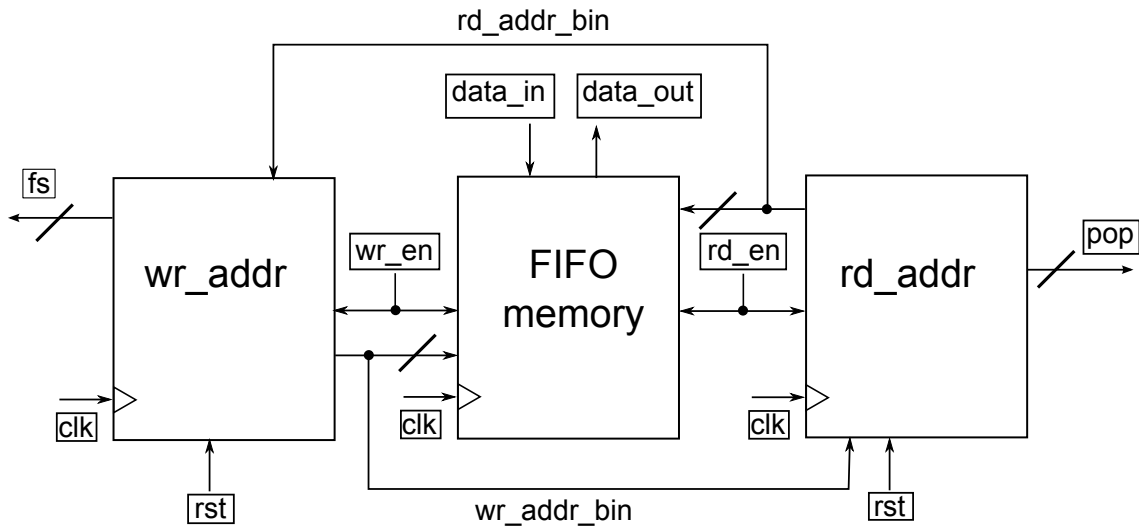


Figure 5.9: Synchronous FIFO design

Similar to the AIM, the DEM also has clock and reset ports, named `clk` and `rst`. To support asynchronous DEM integration, two separate clock inputs are



provided — a read clock `rd_clk` and write clock `wr_clk`. More about asynchronous design using LWDF-V is discussed in Section 5.3.4.5.

#### 5.3.4.5 Low Power Techniques for LWDF-V

The class of dataflow-based signal processing applications that we target with LIDE-V encompasses complex systems whose actors may exhibit large variations in complexity. Such variation is common when applying dataflow based techniques at the application level rather than restricting their use to the level of standard signal processing modules, such as digital filters and FFT computations. In this section, we demonstrate capabilities in LIDE-V that help to address challenges brought about by the need to handle actors with arbitrary variations in complexity, and we also discuss the relevance of these capabilities to low power signal processing. In particular, we propose an asynchronous and GALS-oriented design methodology using LIDE-V for heterogeneous-complexity and low power implementation. Specifically, our proposed methodology applies (1) asynchronous communication between actors that utilizes multiple clock domains, where the bottleneck actors are executed at higher clock frequencies and the others at lower frequencies; and (2) a clock gating technique that “switches off” idle actors to reduce dynamic power consumption.

The novelty of this development centers on the systematic integration of asynchronous design, GALS, and clock gating techniques with lightweight dataflow programming interfaces and their underlying CFDF model of computation. This integration with CFDF is notable in turn due to the utility of CFDF as a foundation for

working with various specialized and heterogeneous forms of dataflow (e.g., see [49]). Furthermore, the orthogonality among CFDF components lays a valuable foundation for asynchronous design, and our proposed clock gating techniques exploit the enable/invoke semantics in CFDF.

#### 5.3.4.5.1 Asynchronous LIDE-V Design

We define a *dataflow clock domain* (or simply “clock domain” when the dataflow context is understood from context) as a maximal set of actors that is driven by the same clock signal. In asynchronous LIDE-V designs, different parts of a dataflow graph can be driven by different clock signals, thus forming multiple clock domains. This in turn allows “slower” actors to be placed in higher frequency clock domains, so that they can be accelerated without having to increase the power consumption of the whole design linearly with the clock frequency, and “faster” actors to be placed in relatively low frequency clock domains, so that the downtime between “faster” and “slower” actors can be reduced.

Figure 5.10 depicts an example of a LIDE-V design with two clock domains, where actor A is driven by `clk_1`, and actors B and C are driven by `clk_2`. Communication channels between actors in the same clock domain are called *synchronous FIFOs*. These FIFOs outline the “synchronous islands” within the overall GALS design. The problem of passing data between synchronous islands is addressed using the asynchronous DEM implementations introduced in Section 5.3.4. We also refer to asynchronous DEMs as *clock domain crossing (CDC) FIFOs*. In Figure 5.10 FIF03 is a *synchronous FIFO* while FIF02 is a *CDC-FIFO*.

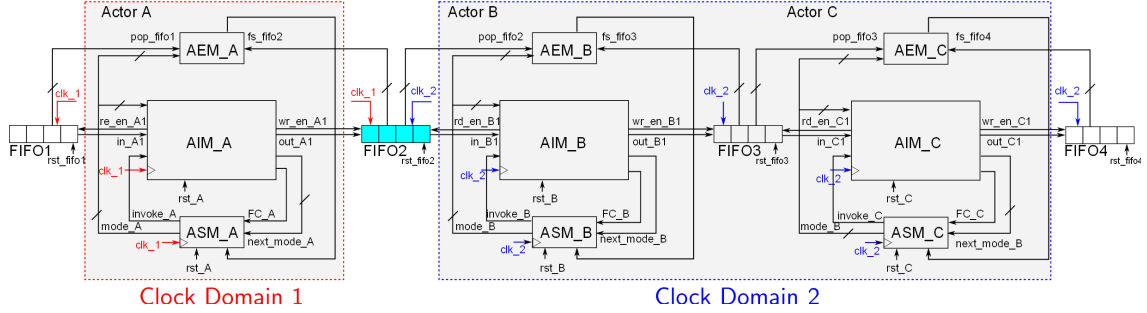


Figure 5.10: Illustration of an LWDF-V-based implementation of a CFDF graph that consists of three actors.

Figure 5.11 illustrates the CDC FIFO design adopted in LIDE-V, which is based on Cummings’s design presented in [71]. CDC FIFOs are driven by two different clock signals, one for read operations (`rd_clk`), and another for write operations (`wr_clk`). We made some adaptations to Cummings’s design so that it is consistent with the LIDE-V framework. For example, Cummings’s design only provides the `empty` and `full` signals, calculated inside the `wr_addr` and `rd_addr` modules. In order to generate the required population `pop` and free space `fs` signals, the `wr_addr` and `rd_addr` modules are modified to calculate these signals by computing the offset between the write pointer and read pointer.

To support clock gating within FIFOs, we make another modification compared to Cummings’s design. The objective here is to allow all of the logic units belonging to the corresponding clock domain to be turned off when `rd_clk` or `wr_clk` is off. In Figure 5.11, for example, the modules in gray are disabled when the `rd_clk` signal is off. This behavior is not considered in Cummings’s design and if the synchronization circuits are off, in that design, the read and write pointers would not be sent to the `wr_addr` and `rd_addr` modules. In order to guarantee that the updates of population `pop` and free space `fs` are performed properly, the `syn_r2w` and

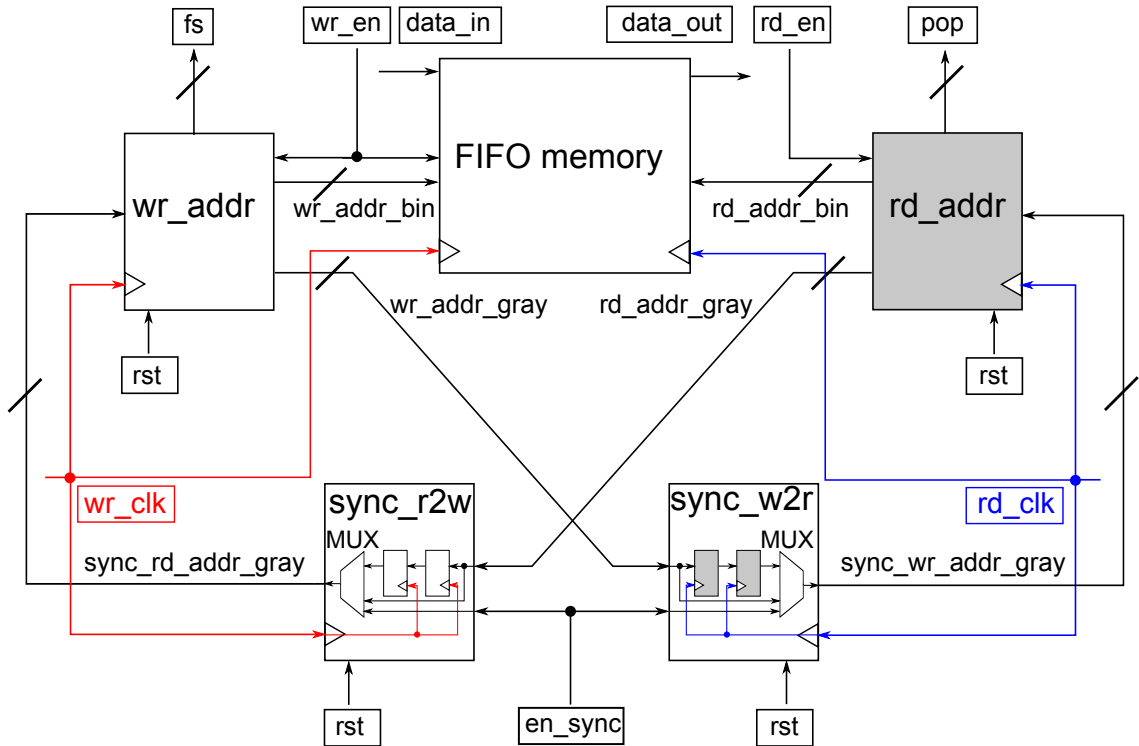


Figure 5.11: Asynchronous FIFO design in LIDE-V.

`sync_w2r` modules both have multiplexers that are responsible for sending read and write pointers, respectively, when one or both of the two clocks is disabled.

### 5.3.4.5.2 Clock Gating

In LIDE-V, clock gating is applied at the actor level to switch off idle components in the dataflow graph. Figure 5.12 depicts an overview of a clock-gated LIDE-V design. Actor-level clock gating is achieved systematically as a natural by-product of the LIDE-V design technique, which in turn allows designers to apply clock-gating more thoroughly and more reliably. We apply clock gating to a LIDE-V actor by adding a clock gating module `CGM_A` to the original LIDE-V actor design illustrated in Figure 5.3.

The LIDE-V-based clock gating technique exploits the graph execution in-

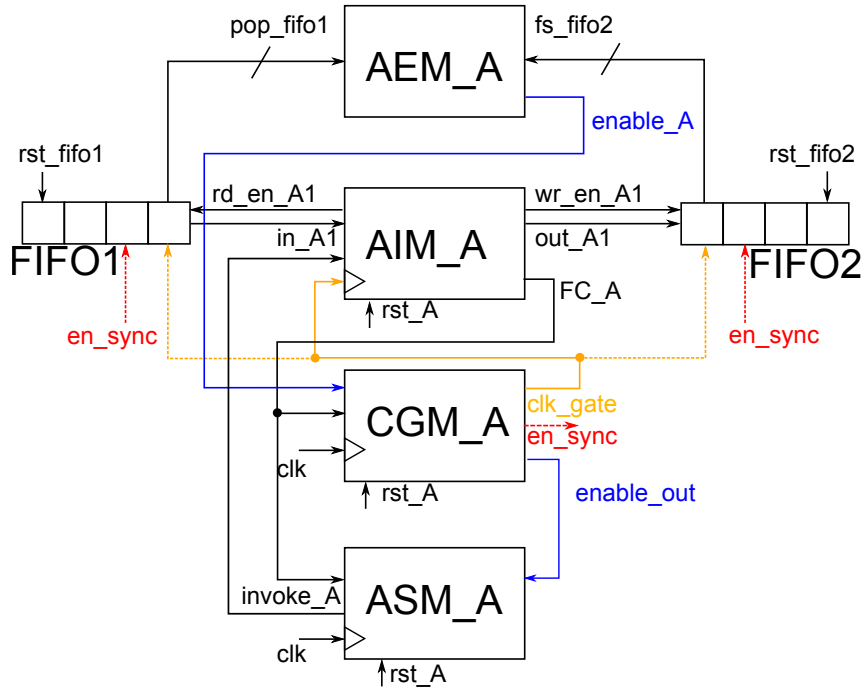


Figure 5.12: Clock gating in a LIDE-V actor.

formation provided by the enable `enable_A` and the firing complete `FC_A` signals. Figure 5.13 illustrates how the signals are related: when `enable_A` from the `AEM_A` is high, `CGM_A` enables the clock (`en_clk` is high), and switches the clock off when the actor has finished its computation — i.e., when the `FC_A` signal from `AIM_A` is high.

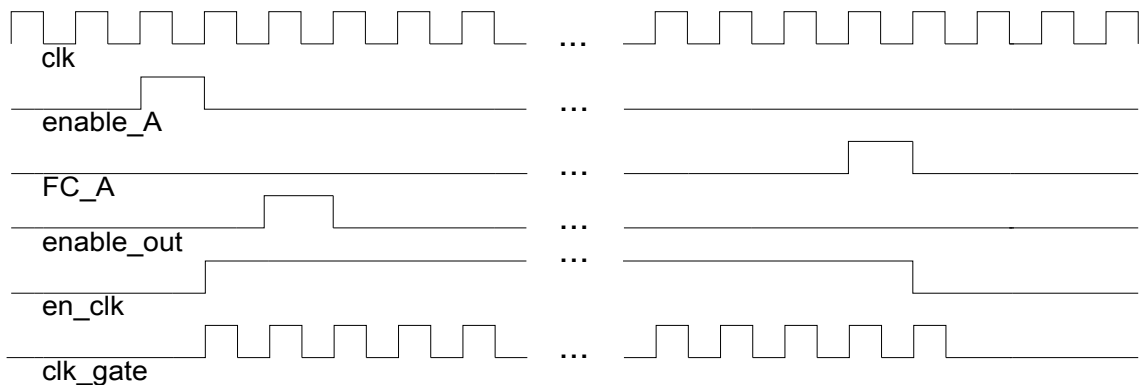


Figure 5.13: Signal waveforms in the clock gating module.

From a technical point of view, the clock is disabled in two different ways,

depending on the target technology. In ASIC designs, it is possible to modify the clock by exploiting some custom logic (e.g., a simple *AND* gate can be used to disable the clock signal), while in FPGAs it is necessary to use dedicated blocks (*BUFGCEs*) to switch it off. The *CGM\_A* also delays the *enable\_A* signal by two clock cycles so that, after an OFF-to-ON transition, the AIM has enough time to be active before it receives the *invoke\_A* signal from the scheduler.

#### **5.3.4.5.3 Clock Gating of Dataflow Edge Modules in Asynchronous Designs**

The clock gating technique introduced above can be applied to both synchronous and asynchronous designs. Moreover, in asynchronous designs, the clock gating technique can be applied not only to actor modules but also to the CDC FIFO modules, as mentioned in Section 5.3.4.5.1. When an actor is idle, it does not read/write data from/to its input/output FIFOs, so the read clock of its input FIFOs and the write clock of its output FIFOs can be disabled to save even more power. Then the CDC FIFO will turn off the corresponding logic units as mentioned in Section 5.3.4.5.1.

To ensure correct updates of the output signals *pop* and *fs*, one additional signal called *en\_sync* is sent from the *CGM* to the CDC FIFO module indicating that either one or both of the *rd\_clk* and *wr\_clk* is/are disabled .

#### 5.3.4.5.4 Clock Gating of Dataflow Edge Modules in Synchronous Designs

In synchronous designs, the clock gating technique cannot be applied to the synchronous FIFO design introduced in Section 5.3.4.4 because the modules that update the signals related to read and write operations are driven by the same clock signal.

One way to enable clock gating of DEMs in synchronous designs is to replace the synchronous FIFOs with the CDC FIFOs mentioned in Section 5.3.4.5.3. However, compared with synchronous FIFOs, CDC FIFOs require more hardware resources and consume more power. Thus, the power saved by switching off the unused logic units may be counteracted by the power overhead introduced by the additional resources. For this reason, we designed a new FIFO, where the `wr_addr` and `rd_addr` blocks are synchronized to different clock signals, and a dual-clock FIFO memory is developed. This new FIFO design, which we call a *pseudo-CDC FIFO*, is illustrated in Figure 5.14. Compared with the CDC FIFO design, the pseudo-CDC FIFO design does not contain synchronization or gray coding circuits, since the `wr_clk` and `rd_clk` clock signals are always connected to the main clock. Similar to CDC FIFOs, the unused logic units in the pseudo-CDC FIFOs can be turned off by clock gating technique to save more power.

#### 5.3.4.5.5 FIFOs comparison

Table 5.1 compares the resource utilization of the synchronous, asynchronous (CDC) and pseudo-asynchronous (pseudo-CDC) FIFO designs presented in this

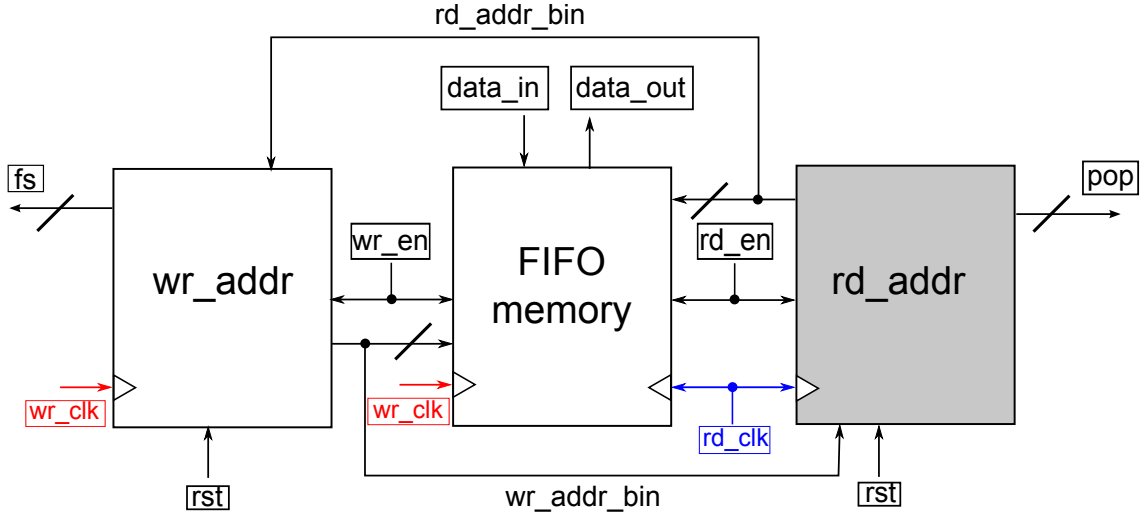


Figure 5.14: Pseudo-CDC FIFO design in LIDE-V.

chapter, and the synchronous FIFO we presented in [47]. The data is extracted from the post-implementation reports of the four FIFOs, all with the capacity being 768 and bit-width being 64.

Table 5.1: Resource utilization of the implemented FIFOs.

	FIFO Type	<i>LUTs</i>	<i>REGs</i>	<i>BRAMs</i>
<i>FIFO</i>	Synch.	73	22	2
<i>CDC FIFO</i>	Asynch.	199	88	2
<i>Pseudo-CDC FIFO</i>	Pseudo-Asynch.	73	22	2
<i>[47] FIFO</i>	Synch.	90	105	2

As we can see from Table 5.1, the CDC FIFO requires the most resources compared to the other FIFO designs. This is due to the additional synchronization and gray coding circuits in the CDC FIFO. Additionally, the pseudo-CDC FIFO has the same resource utilization as the synchronous FIFO, but is adapted to support clock gating in synchronous designs.

Through the comparisons between the synchronous FIFOs that are developed in this chapter and in [47], we conclude that the former requires less resources than



the latter. This is because in the latter design, the `population` and `free\_space` signals are two counters that are updated after every read/write operation, while in the former design, these signals are calculated as offsets between the write and read pointers by means of a combinational circuit. Additionally, the bypass circuit has been removed in the former FIFO design.

The orthogonality (separation of concerns) among actor, edge, and scheduler design in LIDE-V lays a valuable foundation for rigorous integration of power-management within the associated APIs. In particular, we demonstrated in [47] and [46] that methods for asynchronous design, Globally Asynchronous Locally Synchronous (GALS) design, and clock gating can be applied efficiently through natural extensions of the LIDE-V APIs. We also demonstrated the use of these extensions to power optimization.

To manage complexity and improve reuse of subsystems within and across designs, one can encapsulate subgraphs in *LIDE-V* within *hierarchical actors (HAs)*. An HA in LIDE-V appears from the outside as a regular (non-hierarchical) LIDE-V actor with an associated AEM, AIM, and ASM. Execution of an HA as an actor in the enclosing dataflow graph is coordinated by the *external scheduler* associated with the HA. When an HA is fired by its external scheduler, the *internal scheduler* of the HA coordinates the firings of actors that are encapsulated within the HA (nested actors). The internal scheduler carries out the set of nested actor firings that must be completed for a given firing of the HA. An example of an HA with internal and external schedulers is discussed in detail and illustrated in Figure 5.18.

Since it appears from the outside as a regular actor, an HA can be clock gated

in exactly the same way, allowing the designer to efficiently switch off the whole subgraph at appropriate times during operation.

## 5.4 Case Study: A Deep Neural Network for Vehicle Classification

As a concrete demonstration of STMCM, we adopt a DNN use-case for automatic discrimination among four types of vehicles — bus, car, truck, and van. This implementation is based on a neural network design presented in [72], where a network configuration — i.e., the number and types of layers and other DNN hyperparameters — was carefully derived and demonstrated to have very high accuracy. The accuracy of the methods is validated with a database of over 6500 images, and the resulting prediction accuracy is over 97%. The work in this chapter and the work in [72] have different focuses. This prior work focused on deriving hyperparameters, network design, and demonstrating network accuracy, and did not address aspects of resource-constrained implementation or hardware/software co-design. In this chapter, we go beyond the developments of [72] by investigating resource constrained implementation on a relevant SoC platform, and optimized hardware/software co-design involving an embedded multicore processor and FPGA acceleration fabric that are integrated on the platform. In [72], the proposed DNN architectures are evaluated based on the classification accuracy, while in our work, the objectives that we are trying to optimize are system throughput, memory footprints and power efficiency. In addition, this work could be generalized to the design and implementation of any DNN architectures and the work in [72] is selected as a

case study to demonstrate the usage of the methodology proposed in this work. In relation to Figure 5.1, we apply the results from [72] in the block labeled “derivation of hyperparameters and DNN design” as part of the design methodology that is demonstrated in this chapter.

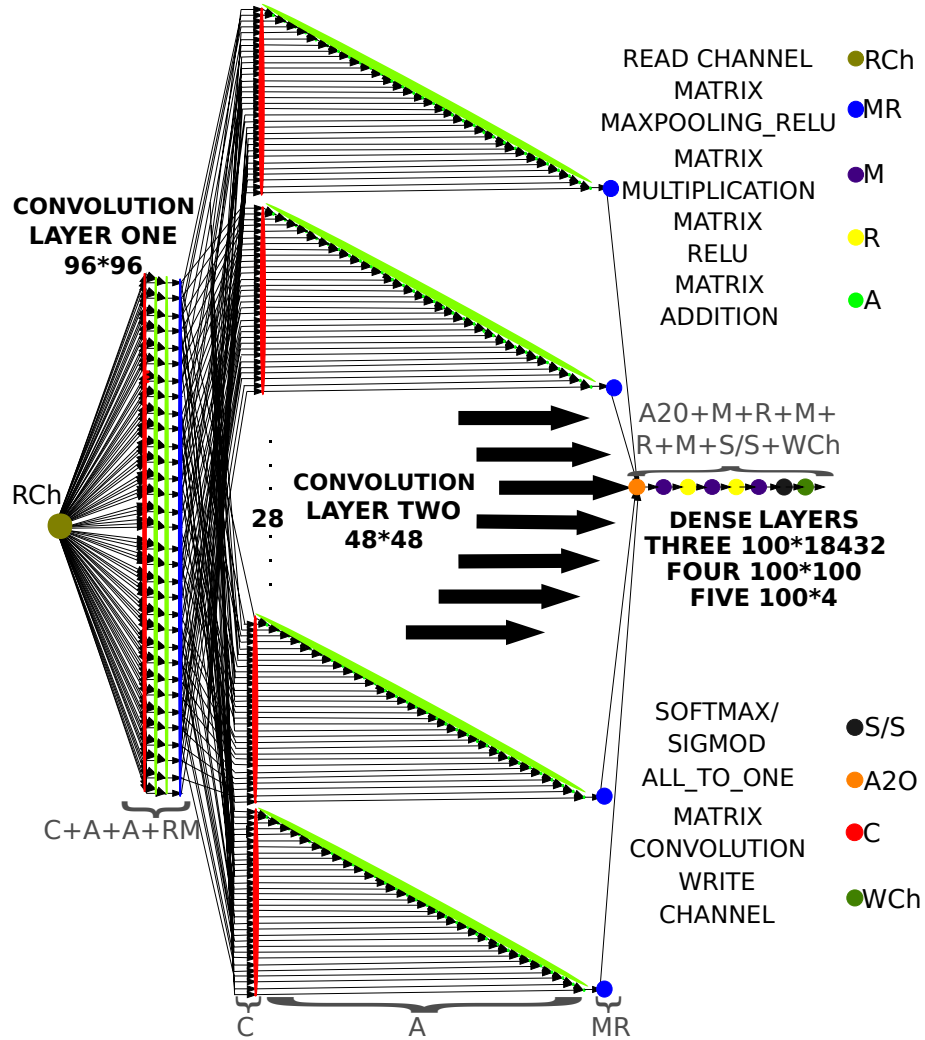


Figure 5.15: DNN for automatic discrimination of four types of vehicles.

The DNN network design that we implement in this work is composed of two convolutional layers, two dense layers and one classifier layer, as depicted in Figure 5.15. The first convolutional layer takes an RGB image ( $3 \times 96 \times 96$ ) as input, and produces 32 feature maps, each with dimensions ( $48 \times 48$ ). The second

convolutional layer takes these 32 feature maps as input and produces 32 smaller feature maps, each having dimensions  $(24 \times 24)$ . We refer to a subsystem that processes multiple input images to produce a single feature map as a *branch*. Thus, the first and second convolutional layers have 32 branches each. The two dense layers combine to transform the feature maps into a  $(1 \times 100)$  vector, which is then multiplied in the classifier layer by a  $(100 \times 4)$  matrix to determine the  $(1 \times 4)$  classification result. Each of the four values in the result corresponds to the likelihood that the vehicle in the input image belongs to one of the four vehicle types (i.e., bus, car, truck and van).

The studied use case is relatively easy to solve compared to common image recognition benchmarks, such as MSCOCO [73], or ImageNet [74]. Therefore, one can reach high accuracy with a relatively simple network requiring significantly lower resources than common network topologies intended for mobile use (such as Mobilenets). As such, the focus of our work is not in mobile devices (e.g., smartphones), but in simpler IoT devices targeted to solving less complex machine learning problems at low cost. For further details on the DNN network design and hyperparameter specifications, we refer the reader to [72].

The specific platform and associated platform-based tools that we employ are based on the Xilinx Zynq Z-7020 SoC. The remainder of this Section focuses on details associated with STMCM and its associated design processes. These details are presented concretely through the development of this DNN case study.

### 5.4.1 Software Implementation and Optimization

In this section, we discuss dataflow-graph- and actor-level optimizations and associated design iterations, as illustrated in Figure 5.1 by the blocks labeled Dataflow Representation, LIDE-C Implementation, and Optimized LIDE-C Implementation. We start with a dataflow graph implementation that is derived using LIDE-C [22, 67], which provides a C-language implementation of the LWDF APIs so that CFDF-based actors and dataflow graphs can be implemented in a structured manner using C. The initial (sequential) LIDE-C design is developed in a design phase that corresponds to the block labeled LIDE-C Implementation in Figure 5.1.

After validating the correct, dataflow-based operation of the initial DNN dataflow graph implementation in LIDE-C, we experiment with various transformations at the actor, subgraph, and dataflow graph levels. Here, we exploit the orthogonality of actor, edge, and graph implementation in LIDE-C, which allows designers to flexibly and efficiently perform experimentation with a wide variety of transformations, and with different combinations of applied transformations. The actor-level transformations performed here are focused on optimization methods applied to the convolution actor, which is a major performance bottleneck in the design. The subgraph-level transformations involve memory management optimizations performed on FIFOs both inside each subgraph (DNN layer) and between pairs of adjacent layers.

### 5.4.1.1 Actor-Level Optimization

We demonstrate actor-level optimization at this stage of the design process using the convolution actor in our DNN example. In our LIDE-C implementation of this actor, we apply a transformation of the convolution computation that is commonly used to simplify the design, and improve classification speed. The transformation involves loop tiling to reduce the cache miss rate. The utility of loop tiling in DNN implementation has been demonstrated previously, for example, in [75]. Using loop tiling, we decompose the main loop of the convolution computation into an inner loop that iterates within contiguous “strips” of data, and an outer loop that iterates across strips. Applying loop tiling in this way allows one to enhance cache reuse based on an array size (strip length) that fits within the cache.

Figure 5.16 shows a segment of code from our application of the tiling transformation to the convolution actor.

```
for (row = 0; row < L; row += tile_num) //tiling for row
  for (col = 0; col < L; col += tile_num) //tiling for col
    for (row_tile = row; row_tile < ((row + 1) < L ? (row + 1) : L); row_tile++)
      for (col_tile = col; col_tile < ((col + 1) < L ? (col + 1) : L); col_tile++)
        for (i = 0; i < n; i++) //sliding window
          for (j = 0; j < n; j++)
            *(output+row_tile*L+col_tile) +=
              *(zero_pad+(row_tile+i)*(L+4)+(col_tile+j)) *
                *(conv_wgt+ (conv_num-1-i)*conv_num +(conv_num-1-j));
```

Figure 5.16: The code segment that implements loop tiling within the LIDE-C actor for convolution.

Through the orthogonality provided by the model-based design rules in LIDE-C, this transformation can be applied at a late stage in our design process, in a way that is interoperable with previously applied transformations, and in a way

that requires no modifications to other actor or edge implementations. In this case, no modification is needed to the dataflow graph scheduler implementation as well, although for some transformations, scheduler adjustments can be useful to integrate transformed actors into the overall system in an optimized way. The CFDF-based APIs (enable and invoke functions) in LIDE-C for scheduler implementation allow the designer to experiment efficiently with such scheduling adjustments as needed.

#### 5.4.1.2 Buffer Memory Management

A major challenge in resource-constrained implementation of a DNN architecture is managing the large volume of data transfers that are carried out during network operation. Each DNN layer typically processes a large amount of data, and requires memory to store the input data from the previous layer or subsystem, the intermediate data during the computation processing, and the computation results that will be transmitted to the following layer or subsystem.

Consider, for example, the buffer memory costs (the storage costs associated with the dataflow graph edges) for the DNN of Figure 5.15. In our LIDE-C implementation, the second convolutional layer requires the most buffer memory. In this layer, each of the 32 branches is composed of 32 convolution actors, 31 addition actors and one actor performing both maxpooling and ReLU (Rectified Linear Unit). Given that the size of the input feature map processed by each branch is  $48 \times 48$  pixels, the buffer memory required for actor communication inside each branch is  $image\_size \times (number\_of\_conv\_actors + number\_of\_output\_feature\_maps)$ ,

which is  $48 \times 48 \times (32 + 1) = 76,032$  pixels. Thus, the total buffer memory inside the second convolutional layer is  $76,032 \times 32 = 2,433,024$  pixels. The buffer memory required for data communication between the first and the second layer can be computed as  $48 \times 48 \times 32 = 73,728$  pixels.

In STMCM, we apply a buffer memory optimization technique that is useful for resource-constrained DNN implementation. In particular, we incorporate a new FIFO abstract data type (ADT) implementation in LIDE-C, called *shared FIFO*, that enables multiple dataflow edges in a graph to be implemented through FIFO ADT instances that share the same region of memory. Such *buffer sharing* in dataflow implementations has been investigated in different forms for various contexts of automated scheduling and software synthesis (e.g., see [76, 77, 78]). In STMCM, we make it easy for the system designer to apply buffer sharing *explicitly* within her or his implementation rather than depending on its implicit support through the toolset that is used. This is an example of the agility that is supported in STMCM, as described at the end of Section 5.2.

Again, by exploiting the orthogonality among dataflow components, buffer sharing in STMCM is performed only on the targeted dataflow edges and requires no modification to other actors or subgraphs. Through the support for such separation of concerns in LIDE-C, different ADT implementations for a FIFO or group of FIFOs can be interchanged without affecting overall system functionality.

There are three key aspects to our application of shared FIFOs in our LIDE-C DNN implementation. First, at the input of each convolutional layer  $L$ , input data from the previous layer is stored centrally instead of being copied separately into



each branch of  $L$ . Second, edges in different layers share the same memory so that the memory is time-division multiplexed between the layers — the processing of a given layer overwrites memory in its shared FIFOs without introducing conflicts that affect the computation results. Third, actors operate on data from shared input FIFOs directly through their read pointers into the FIFO (rather than first copying the data locally within the actor’s internal memory). This kind of copy-elimination is similar to dataflow memory management techniques introduced by Oh and Ha [77].

Improvements resulting from our application of shared FIFOs are demonstrated quantitatively in Section 5.5.1.

### 5.4.1.3 Software Profiling

In this subsection, we demonstrate the process of software profiling, as illustrated in Figure 5.1, in the context of our optimized LIDE-C implementation of the DNN architecture. The implementation platform is an Intel i7-2600K running at 3.4GHz. Table 5.2 and Table 5.3 show layer- and actor-level software profiling measurements, respectively.

Table 5.2: Layer-level software profiling. Here, the row labeled “T” gives the execution time of each layer, and the row labeled “T%” gives the percentage of the total DNN execution time that is attributed to each layer.

	Layer					Total
	1	2	3	4	5	
T [ms]	18.71	22.08	0.0149	0.0034	0.0036	40.812
T%	45.84	54.10	0.04	0.01	0.01	100

In Table 5.3,  $T_{ic}$  denotes the *invoke to firing completion time* of a given actor.

Table 5.3: Actor-level software profiling.

Layer	Convolution Layer 1			Convolution Layer 2		
Actor	Conv	Add	M&ReLU	Conv	Add	M&ReLU
$T_{ic}$ [ $\mu$ s]	230.10	0.03	0.025	59.77	0.005	0.006
Layer	Dense Layer 3		Dense Layer 4		Output Layer 5	
Actor	Mult	ReLU	Mult	ReLU	Mult	Softmax
$T_{ic}$ [ $\mu$ s]	5.1	0.0012	0.029	0.0012	0.0023	0.0031

This is the average time that elapses between the time that an actor firing is initiated and when the firing completes. We also refer to  $T_{ic}$  as the *average execution time* of the associated actor. The abbreviations Add, Conv, Mult, and M&ReLU stand, respectively, for Addition, Convolution, Multiplication, and Maxpool-and-ReLU.

Layer- and actor-level software profiling provide insight into the processing complexity of actors in each layer. According to Table 5.2, the convolutional layers account for 99.94% of the system execution time. Also, the execution time of Layer 2 is very close to that of Layer 1. In both convolutional layers, the Conv actors account for most of the processing time compared with the other two actors — Add and M&ReLU — in the convolutional layers. Additionally, the average execution time of the Conv actors in Layer 2 is only about a quarter of that of the Conv actors in Layer 1. This is primarily because each of the Conv actors in Layer 1 processes input images of size  $96 \times 96$ , while the Conv actors in Layer 2 process input feature maps that have size  $48 \times 48$ .

## 5.4.2 Hardware Implementation and Design Exploration

In this section, we describe the main capabilities of the design flow depicted in Figure 5.1 with respect to design and implementation of hardware accelerators.

These capabilities are represented by the blocks in the region labeled “Hardware-related Process”.

For example, through a preliminary hardware profiling phase of the DNN application described in Section 5.4.2.1, we can identify three hardware design aspects that are interesting to investigate in detail — the adoption of clock gating techniques, exploitation of asynchrony that is inherent in dataflows, and exploration of different levels of actor granularity.

We demonstrate the hardware-related design process of STMCM using a hardware accelerator that is introduced in [46]. The accelerator provides a subsystem for producing feature maps from the first convolutional layer of the DNN application. In the remainder of this chapter, we refer to this subsystem as the *Subtree for Feature Map* (SFM).

Due to the interfacing consistency that is maintained across LIDE actor implementations in different languages, one can readily convert the LIDE-C based SMF subsystem implementation into hardware by replacing each software actor with a hardware module that is designed in LIDE-V, and by connecting the derived hardware actors with LIDE-V FIFOs. Following the general approach of realizing LIDE actors in hardware, each LIDE-V actor implementation is decomposed into an AEM and AIM. The AEM is reusable among different actors in our implementation, although in general it can be useful to have specialized AEM implementations that are streamlined for the specific requirements of individual actors [46].

The hardware implementation diverges from the LIDE-C design in two major ways. First, we feed the input data in an interleaved format, reducing the complexity

of the hardware interface and driver software since there is only one input FIFO to manage. Second, the hardware actors are designed to produce one row per firing instead of entire images. This reduces the FIFO size requirements in the first layer from  $96 \times 96$  pixels to only 96 pixels. The hardware actors in our implementation are scheduled using a fully distributed approach.

The resulting SMF is shown in Figure 5.17. The implemented hardware is verified against reference outputs extracted from the LIDE-C implementation. In this Figure, production and consumption rates (dataflow rates) are annotated next to actor ports, and  $w$  is the input image width. The convolution actor has multiple operating modes (CFDF modes) with different consumption rates.

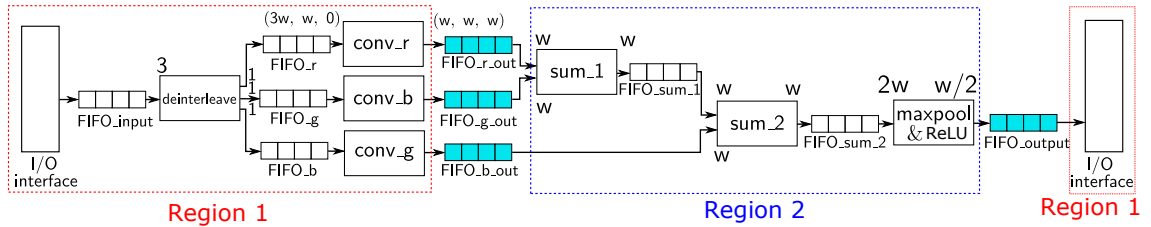


Figure 5.17: LIDE-V implementation for the accelerated SFM.

#### 5.4.2.1 Hardware Profiling

We employ hardware profiling in STMCM to extract execution time data, which is later used to guide the process of iterative design optimization. In this section, we demonstrate hardware profiling in the context of our DNN application. Profiling is performed using the target platform, which in our demonstration is the Zynq Z-7020 SoC. We profile the LIDE-C implementation on the ARM A9 MPCores provided by the target platform and develop a first version implementation of the

SFM on this platform and extract execution time data from this implementation.

Table 5.4 depicts various data associated with execution times and waiting times for the SFM hardware accelerator illustrated in Figure 5.17. Here, the symbol  $t_{tot}$  represents the total time necessary to execute the *SFM*;  $T_{ic}$  is the average time period between an actor invocation and its corresponding firing completion;  $T_{ci}$  is the average time period that an actor has to wait to be fired after its previous firing completion; *firings* is the number of firings of a given actor during execution of *SFM*; *Tot*, calculated as  $(T_{ic}) \times (\textit{firings})$ , gives the total execution time of a given actor during the execution of *SFM*;  $T_{ii} = (T_{ic} + T_{ci})$  denotes the average time period between the beginning of one invocation to the beginning of the next; and the ratio  $T_{ii}/T_{ic}$  measures the extent of actor idleness.

This rich collection of metrics, which is supported by the underlying CFDF model computation, provides various insights on the dataflow-based system architecture and its implementation. For example, the  $T_{ii}/T_{ic}$  ratio provides insight on differences in processing speed that are useful in exploiting the inherent asynchrony between dataflow actors.

Table 5.4: Measured data associated with actor execution times and waiting (idle) times.

<i>SFM</i> $t_{tot}$	232,831				
	$T_{ic}$	$T_{ci}$	<i>firings</i>	<i>Tot</i> ( <i>Tot</i> %)	$T_{ii}/T_{ic}$
<i>Deinterleave</i>	3	2	9216	27,648 (11.87)	1.67
<i>Convolution</i>	2402	2	96	230,592 (99.04)	1.00
<i>Sum</i>	107	2297	96	10,272 (4.41)	22.46
<i>Maxpool&amp;ReLU</i>	195	4613	48	9360 (4.02)	24.66

From analysis of our hardware profiling results (Table 5.4), we can derive different versions of the SFM hardware accelerator with different trade-offs among

power consumption, system throughput, and hardware resource cost. Firstly, looking at column  $Tot\%$ , we see that all of the actors except for Convolution are inactive throughout most of the execution time. The maximum proportion of active time among these actors is 11.87%, reached by Deinterleave. Gating the clock of these frequently inactive actors can provide more energy efficient accelerator operation by eliminating dynamic power consumption during idle phases.

Furthermore, the Deinterleave and Convolution actors have relatively small idleness levels ( $T_{ii}/T_{ic}$ ), with a waiting time  $T_{ci}$  equal to 2 clock cycles for both of them. On the other hand, Sum and Maxpool&ReLU exhibit much larger waiting times and idleness levels. An important hint coming from the  $T_{ci}$  values is that, thanks to the inherent asynchrony of dataflow actors, it is possible to partition the design into different clock regions working at different frequencies, thus obtaining a GALS design. In particular, the *Deinterleave* and *Convolution* actors can be placed in one clock region (Region 1), driven by *clock 1*, while *Sum* and *Maxpool&ReLU* can be placed in another region (Region 2), driven by *clock 2*. On the basis of the measured  $T_{ii}/T_{ic}$  values, we can set *clock 2* to be 20 times slower than *clock 1*.

Moreover, the subgraph included in Region 2 can be encapsulated into a hierarchical actor (see Section 5.3.4). This actor, seen from the “outside”, is like any other LIDE-V actor. The actor and its encapsulated subsystem can be clock gated or clocked with a different frequency, providing additional candidate solutions for SFM accelerator optimization.

### 5.4.2.2 SFM Exploration

Based on the hardware profiling analysis discussed in Section 5.4.2.1, we explored six different variants of the SFM design:

- $SFM_a$ : This is an asynchronous design where actors belonging to different logic regions run at different clock frequencies. In particular, the clock frequency for *clock 1* is set to 100 MHz, and the clock frequency for *clock 2* is set to 5 MHz. Referring to Figure 5.17, the only modification required in the design is the replacement of FIFOs that are placed *between* the two clock regions. These FIFOs need to be replaced with asynchronous FIFOs — for this purpose, we employ the clock domain crossing (CDC) FIFOs presented in [46]. CDC FIFOs are designed with read and write logic that can be driven by different clocks. At the same time, their module interfaces conform to standard LIDE-V edge interfaces so they can replace other FIFO implementations without requiring changes to actors that communicate with them.
- $SFM_{CG}$ : Based on our hardware profiling results, we apply clock gating to the Deinterleave, Sum and Maxpool&ReLU actors. To be clock gated, a LIDE-V actor needs only the instantiation of a clock gating module (CGM) [46]. The CGM involves a BUFG primitive that physically enables/disables the clock signal in the target SoC. Thus for each clock gated actor  $A$  in  $SFM_{CG}$ , a CGM is instantiated and connected to the clock inputs of  $A$  and to the read- and write-clock inputs, respectively, of the FIFOs that  $A$  reads from and writes to.
- $SFM_{aCG}$ : This design incorporates both asynchronous design and clock gating

techniques. As in  $SFM_a$ , the FIFOs between the two clock regions are replaced with CDC FIFOs. Additionally, the Deinterleave, Sum and Maxpool&ReLU actors are clock gated as in  $SFM_{CG}$ , and a CGM is instantiated for each of these actors.

- $SFM_h$ : This is a hierarchical SFM design, which can be viewed as a baseline for evaluating our enhanced hierarchical design  $SFM_{hCG}$  (defined below). In  $SFM_h$ , Region 2 (see Figure 5.17) is encapsulated in a hierarchical actor  $H$ . An illustration of this hierarchical actor is provided in Figure 5.18. The subgraph that is encapsulated by  $H$  contains three actors  $A1$ ,  $A2$  and  $B$ . We denote this subgraph by  $G_H$ . Actors  $A1$  and  $A2$  correspond to  $Sum\ 1$  and  $Sum\ 2$ , respectively, which are two actors that add outputs from the three convolution actors. Actor  $B$  corresponds to the Maxpool&ReLU actor.

When  $H$  is viewed as a single actor from the outside, a firing of  $H$  starts when the internal scheduler  $L\_ASM\_HA$  for  $G_H$  receives the  $invoke\_HA$  signal from the external scheduler  $E\_ASM\_HA$ . Inside the subgraph  $G_H$ , the  $invoke\_HA$  signal is received by  $ASM\_A1$ , which is the ASM of actor  $A1$ . Once  $ASM\_A1$  receives the  $invoke\_HA$  signal, the firing of the subgraph  $G_H$  starts.

- $SFM_{hCG}$ : This design is the same as  $SFM_h$ , except that the Deinterleave actor and the hierarchical actor are clock gated. It is important to highlight that the application of clock gating at the region level is advantageous if the execution times of the actors within the region are overlapped. In this design, however, the execution times of the three actors are not overlapped. When one actor is executed, the others wait in an idle state and waste power. Therefore, we



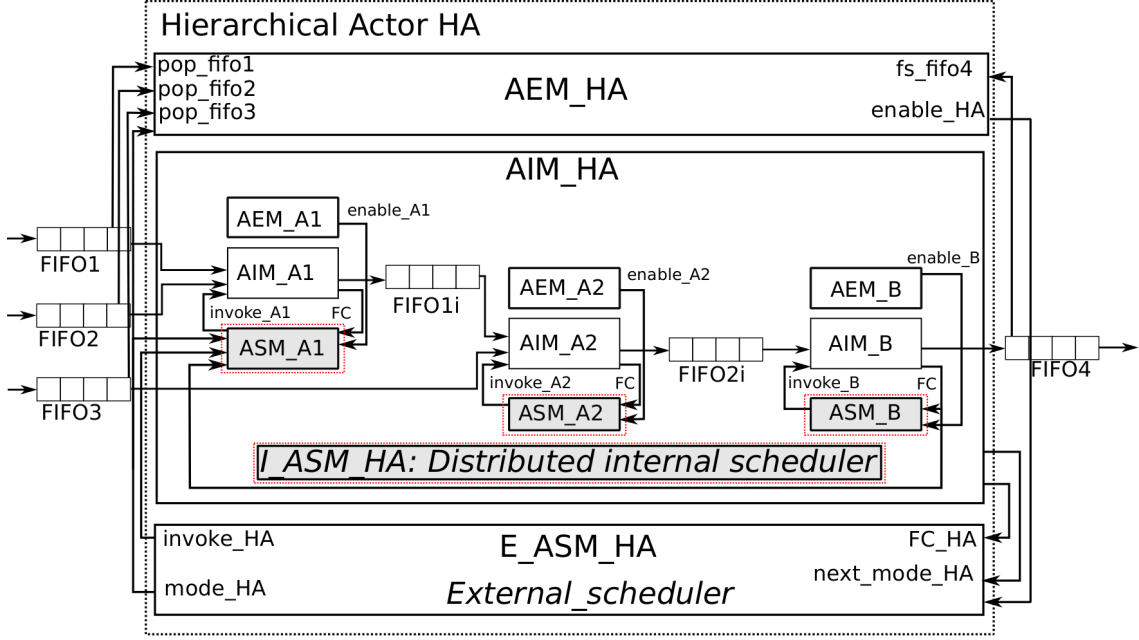


Figure 5.18: An illustration of the hierarchical actor associated with Design  $SFM_h$ .

expect that this configuration would not be really effective in reducing power consumption as  $SFM_{CG}$  in the targeted DNN case. However, we include the test in our explorations to present the complete wide variety of options made available by STMCM (even if some of them may be less efficient than others for this particular application scenario).

- $SFM_{auto}$ : This is a version of the SFM that is synthesized and implemented by enabling the automatic power optimization available within the adopted Xilinx Vivado environment. This design applies fine-grain clock-gating and fine-grain logic-gating at the Verilog level and *excludes* all of the higher-level, dataflow-based optimizations (coarse-grain asynchronous design, clock-gating, and hierarchical decomposition) that are applied in the other five investigated designs. Thus,  $SFM_{auto}$  is useful as a common baseline to assess the higher-level models and transformations provided by  $STMCM$  compared to existing off-the-shelf

synthesis techniques.

### 5.4.3 Joint Hardware/Software Implementation and Optimization

This section shows how the proposed design flow (summarized in Figure 5.1) provides a variety of interesting hardware/software co-design implementation choices and optimization possibilities. In particular, these features are represented by the “Co-design-related Process” area of Figure 5.1. For a given high-level LWDF model, the interaction between software (see Section 5.4.1) and hardware (see Section 5.4.2) actors or subgraphs can be shaped and refined depending on the specific constraints and requirements of the application.

In particular, we demonstrate two main implementation aspects that can be efficiently explored with STMCM: parallelism across actor execution, and the adopted communication interfaces. The degree of parallelism can be tuned depending on the number of software and/or hardware cores adopted for the execution of a certain computational step, while different communication interfaces allow different levels of coupling between hardware and software actors. Both of these dimensions for exploration therefore represent important sources of trade-offs to consider during the implementation process.

For the purpose of our co-design explorations, the DNN application has been split into two parts to be executed respectively in software (PS) and hardware (PL). Here, PS and PL stand for Processing System and Programmable Logic, respectively. In our experiments, we consider the SFM subsystem introduced in Section 5.4.2

as the portion of DNN application that will be accelerated in the PL, while the remaining part, involving the second convolutional layer, two dense layers and final classification layer, will be executed by the PS.

Note that the first convolutional layer constitutes only one of the main computationally intensive steps of the DNN application. According to software profiling results that are based on the SoC platform that we applied for hardware/software co-design (see Table 5.9), the first convolutional layer only accounts for about 27% of the prediction time. For this reason, the speedup brought by hardware acceleration to the overall DNN application is not dramatic, as will be discussed further in Section 5.5. However, the results concretely demonstrate how STMCM can be applied to perform extensive design space exploration across a variety of diverse designs to achieve system performance enhancement under highly-constrained hardware resource availability.

The SFM accelerator has been integrated into the LIDE-C design presented in Section 5.4.1 by replacing the SFM software implementation with function calls to driver software that is capable of offloading the computation to the PL. We have experimented with using a Linux kernel driver based on the Userspace I/O (UIO) framework [79], and a driver that is independent of the Linux kernel and operates by directly accessing memory with the mmap system call. The UIO approach is more suitable for production use, while mmap works well for prototyping, and this latter approach has been used in this work for evaluation. The PS and PL can communicate by means of AXI interfaces exploiting General Purpose (GP) ports; 32-bit width PS master or slave ports with 600 Mbps bandwidth for both read

and write channels; High Performance (HP) ports or Accelerator Coherency Ports (ACP); and 64-bit width PS slave ports with 1200 Mbps bandwidth for both read and write channels.

Figure 5.19 depicts the reference configuration for the co-design explorations. In order to integrate the accelerator into the SoC, a generic AXI wrapper for hardware dataflow subgraphs has to be provided. The wrapper is compliant with the adopted AXI interface and lets the programmer access the input and output FIFOs of the dataflow graph and monitor their populations. For this purpose, the wrapper includes all the necessary logic for the communication management.

In our hardware acceleration approach, we map the SFM subsystem to hardware. This subsystem produces a 48x48 feature map on each execution. Thus, in order to perform the entire first convolution layer of the DNN application, which must produce 32 48x48 feature maps, the SFM accelerator has to be executed 32 times with the appropriate convolution coefficients. For each of these SFM executions, the PS will send the corresponding convolution coefficients to the accelerator. The input image, which remains the same across all 32 executions, is sent only once from the PS and stored within a local buffer within the accelerator. All 32 executions of the SFM access the input image from this local buffer. In this way, we avoid the large amount of data transfer that would be required if the input image had to be sent separately from the PS to the PL for each SFM execution. Upon completion of each SFM execution, the PS retrieves the resulting feature map from the accelerator.

In the remainder of this section, we discuss in detail three different sets of

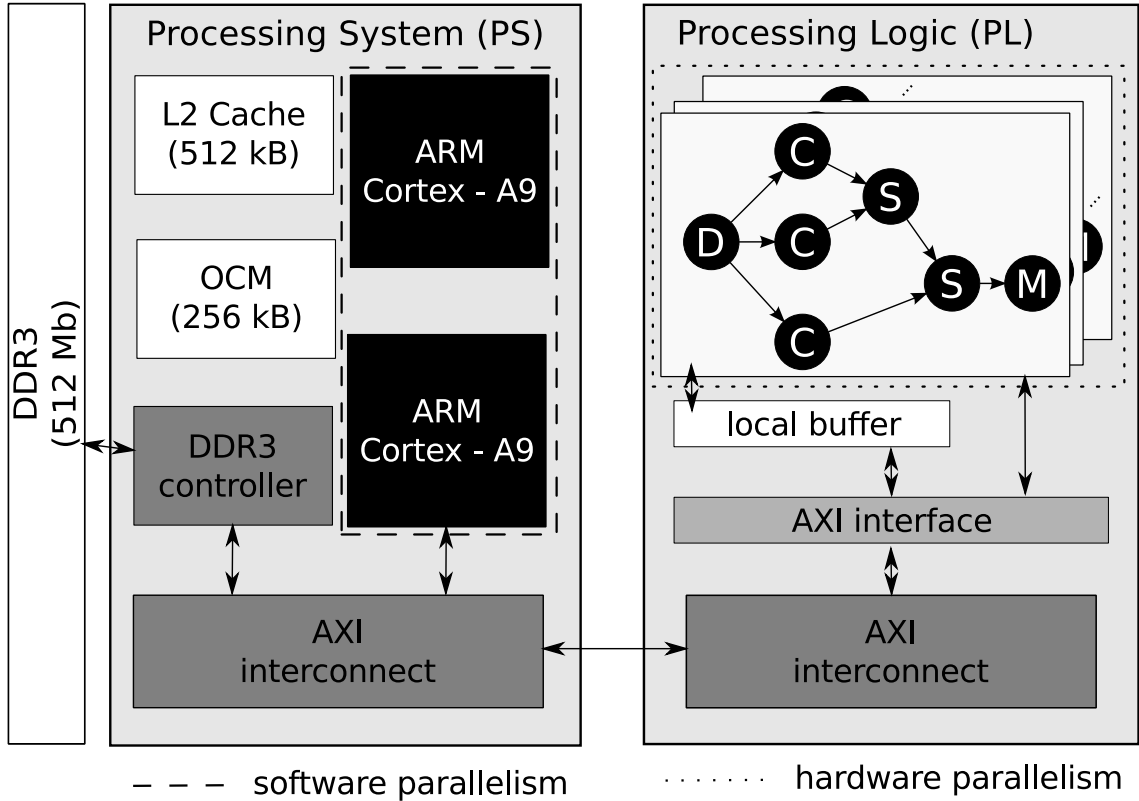


Figure 5.19: Reference configuration for hardware/software co-design exploration in our experiments.

co-design implementations and optimizations that are facilitated by STMCM:

- the amount of parallelism that is exploited in the software and hardware subsystems;
- two alternative communication interfaces that offer different trade-offs in terms of resource requirements and execution speed; and
- local buffering to avoid redundant transmission of common data across different branches of the SFM accelerator.

These three sets of co-design explorations are discussed further in Section 5.4.3.1, Section 5.4.3.2, and Section 5.4.3.3, respectively.

### 5.4.3.1 Exploiting Parallelism

STMCM allows the designer to experiment efficiently with the amounts of parallelism that are exploited in both the hardware and software subsystems (see the dashed squares in Figure 5.19). In particular, depending on the specific application requirements, multiple parallel instances of software cores or hardware accelerators can be utilized. While software cores are able to execute all DNN application steps, hardware accelerators can only perform the steps that they have been conceived for. Generally speaking, hardware accelerators achieve higher efficiency than software cores when executing a given computational step, both in terms of execution time and resource efficiency (resource utilization and consumption).

In the targeted Xilinx Zynq Z-7020 SoC platform, a pair of homogeneous cores is available, so that the maximum degree of software parallelism in our implementations is 2. The available cores are both ARM A9 MPCores with two levels of cache and access to a 512 Mb off-chip DDR RAM. In our experiments, we have exploited software parallelism for the two most computationally intensive steps of the application — the two convolutional layers.

When using FPGA fabric, designers have the possibility to utilize as much parallelism as the FPGA resources allow. In this work, we have investigated three alternative designs that utilize 1, 2 or 4 parallel SFM instances, respectively, in the same hardware accelerator. In the first case, the accelerator is executed 32 times in order to complete the 32 branches of the first convolutional layer. This design executes a different branch with different convolution coefficients for each

accelerator invocation. In the second case (2 parallel SFM instances), the accelerator execution time is halved, but for each run, two new sets of convolution coefficients are necessary. Finally, with 4 parallel SFM instances, only 8 accelerator executions are needed, with each requiring the updating of four different sets of coefficients.

### 5.4.3.2 Communication Interfaces

During the process of co-design exploration, STMCM gives the designer significant flexibility to select interfaces for communicating data between the hardware and software subsystems. This flexibility is provided by the general dataflow model of computation that underlies STMCM. Flexibility in selecting a communication interface can be very useful in the context of resource- or performance-constrained design. This is demonstrated, for example, by the work of Silva et al., which analyzes trade-offs among the different AXI interface options [80].

We investigated the usage of two different AXI communication interfaces located at the extremes of the resource-versus-performance trade-off:

- the memory-mapped AXI4-lite (*mm-lite*) interface; and
- FIFO-based AXI4-stream (*stream*) interface.

Compared to the stream interface, the mm-lite interface has lower resource requirements, but it also exhibits lower performance. The mm-lite interface uses memory-mapped, one-by-one transfer of data items. The interface is particularly intended for control signals and small-scale data accesses. It does not need any additional modules beyond those depicted in Figure 5.19, and it uses only one of

the PS master GP ports. For example, the execution of one branch of the first layer requires the input images (3 RGB images with  $96 \times 96$  pixels each) and kernel coefficients (3 kernels with  $5 \times 5$  coefficients each). Since the mm-lite interface uses a separate data transfer operation for each pixel, this results in a total of  $3 \times 96 \times 96 + 3 \times 5 \times 5$  data transfer operations. Once the accelerator completes its computation, the mm-lite interface requires  $48 \times 48$  data transfer operations to enable the processor to read the output feature map.

Unlike the mm-lite interface, which performs data transfers one-by-one, the stream interface employs a DMA engine that transfers data between processor memory and the accelerator in blocks, where the block size can be up to 256 bytes. Successive data items within a block are transferred in consecutive clock cycles. The stream interface requires a DMA engine, as mentioned above, and additional FIFO buffers, and therefore incurs significant overhead in terms of resource requirements. Note that the additional hardware required by the stream interface is not depicted in Figure 5.19. The DMA engine is configured through one of the PS master GP ports, and requires two different PS slave HP ports to directly access the memory where data to be transferred to/from the accelerator is stored.

To execute one branch of the first DNN layer, the stream interface performs (a) 96 memory-to-accelerator DMA operations to send the input images, with  $96 \times 3$  pixels for each DMA operation, and (b) one memory-to-accelerator DMA operation to send  $5 \times 5 \times 3$  kernel coefficients. Additionally, the stream interface needs 48 accelerator-to-memory DMA operations to retrieve the computed feature map, with 48 pixels for each DMA operation.



### 5.4.3.3 Local Buffering

As mentioned previously, we incorporate local buffering of image pixels in the SFM accelerator to avoid redundant transmission of common data across different branches of the accelerator. This local buffering optimization is applied to both the mm-lite-interface- and stream-interface-based accelerator implementations.

For an accelerator configuration with a single SFM instance, the input image data is transferred to the accelerator only during execution of the first branch. After being transferred, this data is retained in a local buffer within the accelerator for reuse by the remaining 31 executions. For accelerator configurations that have multiple (parallel) SFM instances, the input image is also transferred only once to the accelerator. For these configurations, the image data is reused by the remaining executions of all of the SFM instances. Thus, our incorporation of local buffering optimization eliminates input image data transfers for all branches except the first one.

## 5.5 Results

In this section, we present experimental results to demonstrate the design and implementation methods provided by STMCM based on the detailed case study presented in Section 5.4. The main contribution of this section is to demonstrate that the proposed methodology facilitates efficient experimentation with alternative dataflow-based architectures, design optimization methods, and implementation trade-offs.

## 5.5.1 Embedded Software Implementation

In this section we present results of our experimentation using STMCM to explore alternative embedded software implementations. We focus specifically on the optimized application of loop tiling and buffer memory management.

### 5.5.1.1 Loop Tiling

As introduced in Section 5.4.1.1, in the optimization of our LIDE-C implementation of the DNN application, we explored loop-tiled convolution actor designs with different tile sizes. Specifically, we measured the number of cache load misses and the cache load miss rates during execution of a convolution actor. The valid tile sizes for each convolution actor were those within the range of 1 to  $D$ , where  $D$  is the dimension of input images to the actor. For example, for the convolution actors in Layer 1, which process input images with size  $96 \times 96$  pixels, we explored tile sizes within the range of 1–96.

Figure 5.20 shows the number of cache load misses and cache load miss rate under different tile sizes for convolution actors with different input image dimensions ( $48 \times 48$ ,  $96 \times 96$ ,  $750 \times 750$ , and  $1500 \times 1500$ ). As we can see from the results, the cache load miss rates are very small for image dimensions  $D \in \{48, 96, 750\}$ . This indicates that the data can be fully stored or almost fully stored in the cache with any valid tile size.

For  $D = 1500$ , however, there is significant variation in the cache load miss rate across different tile sizes. The rate reaches its lowest value when the tile size

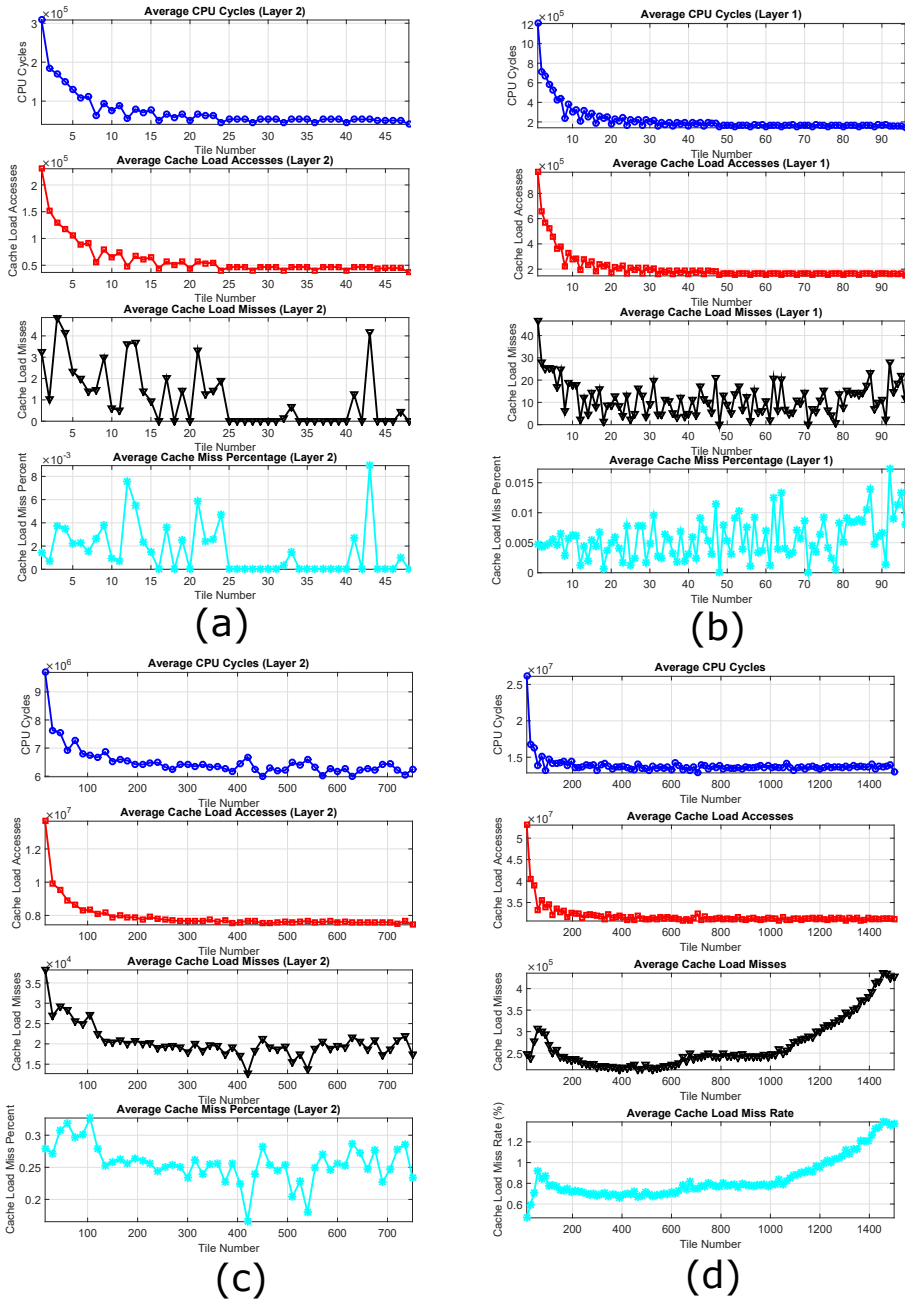


Figure 5.20: Performance evaluation of convolution actors with different image dimensions: (a)  $48 \times 48$ , (b)  $96 \times 96$ , (c)  $750 \times 750$ , (d)  $1500 \times 1500$ .

is approximately 400. With careful setting of the tile size, loop tiling significantly reduces the cache miss rate for convolution actors that have relatively large image dimensions.

Additionally, we can see that there is a large average CPU cycle count for

small tile sizes in all figures. We expect that this is due to the overhead caused by the additional `for` loops that are introduced by the loop tiling transformation.

In summary, based on our simulation analysis for small image dimensions ( $96 \times 96$  and  $48 \times 48$ ), loop tiling does not help to reduce the cache miss rate on the target platform, and furthermore, it introduces overhead due to the additional `for` loops. Thus, loop tiling should not be applied to this DNN application for low image dimensions. However, our experiments also show that for larger image dimensions, loop tiling does help to improve the efficiency by reducing the cache load miss rate.

### 5.5.1.2 Buffer Memory Management

Figure 5.21 shows the amount of memory required for data storage in each DNN layer. We report memory requirements in this section in terms of pixels. In our experiments, we used a 4-byte floating point data type for each pixel. Figure 5.21 also shows the amount of data communication that is needed between adjacent layers, and the amount of memory that must be active simultaneously during the computation associated with each layer. The memory needed for input is calculated as  $input\_image\_size \times number\_of\_input\_images$ . The memory needed for execution of each layer is calculated as  $input\_image\_size \times (number\_of\_input\_images + 1) \times number\_of\_output\_feature\_maps$ .

As we can see from Figure 5.21, the processing in Layer 2 requires the largest amount of active memory, and a minimum of 2,525,184 pixels must be allocated for buffer storage. The memory size can be optimized subject to this constraint

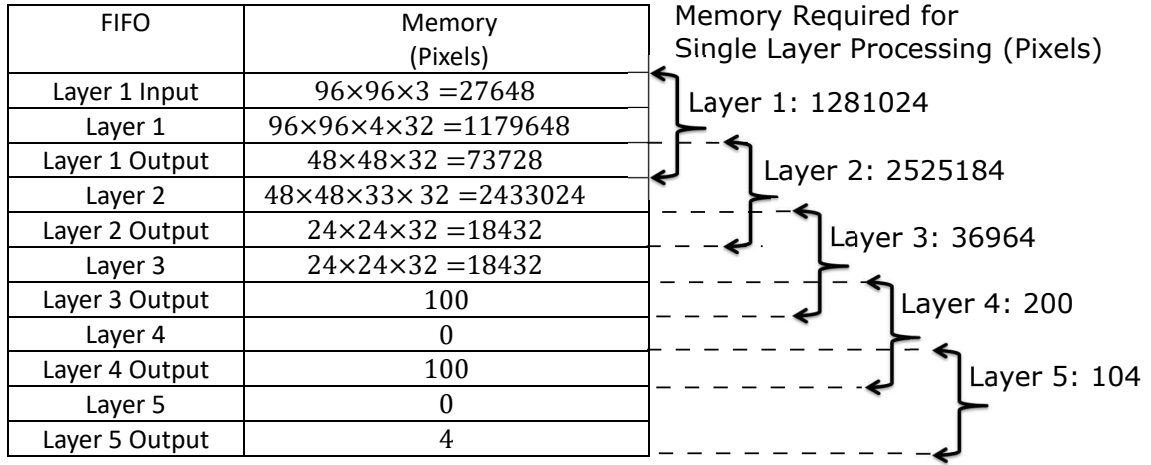


Figure 5.21: Buffer memory and communication requirements in the DNN architecture.

through the application of shared FIFOs, which were introduced in Section 5.4.1.2. The buffer memory allocation that we propose for this DNN application based on shared FIFOs is illustrated in Figure 5.22.

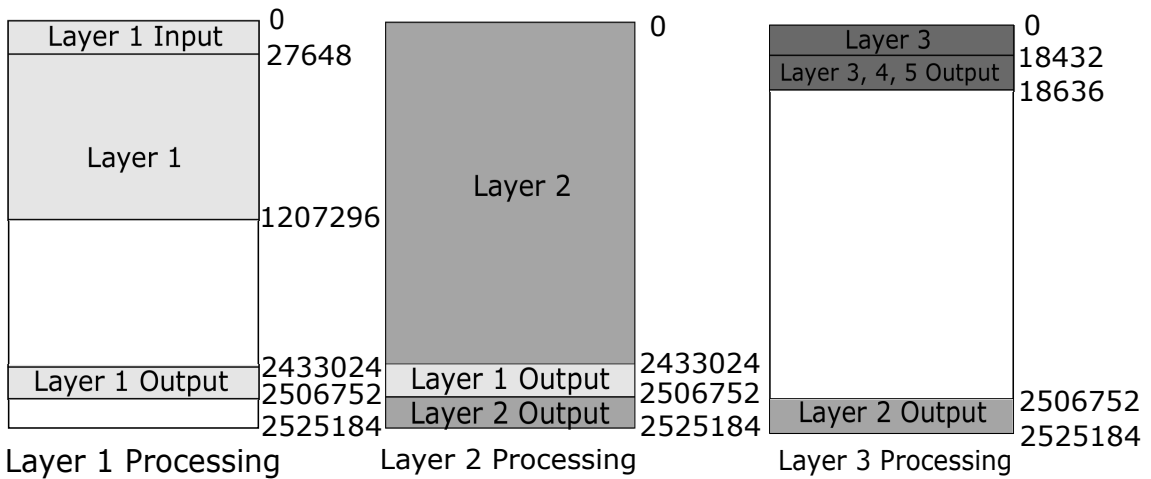


Figure 5.22: Buffer memory allocation for the DNN application.

Table 5.5 summarizes the memory requirements for dataflow edges (FIFO buffers) and actors in the two convolutional layers, which require most of the memory among the five layers. These memory requirements are shown both with and without the use of shared FIFOs. As discussed in Section 5.4.1.2, actors operate on data

from shared input FIFOs directly without copying data to its internal memory. Thus, convolution actors only need memory for its intermediate computation results. Add and Maxpool&ReLU actors do not require additional memory. The results presented in this table quantitatively demonstrate the utility of shared FIFOs for this application. In particular, the application of shared FIFOs reduces the memory requirements by 65%.

Table 5.5: Memory requirements (in pixels) for the first two layers. In bracket in the last column: the percentage of memory requirement of DNN with shared FIFOs with respect to that of DNN with common FIFOs.

	FIFOs	Convolutional Layer 1			Convolutional Layer 2			Total
		Conv.	Add	Maxpool&ReLU	Conv.	Add	Maxpool&ReLU	
Common FIFOs	6875136	1806720	1179648	368640	5128192	2433024	92160	17883520
Shared FIFOs	2525184	921984	0	0	2768896	0	0	6216064 (34.8)

## 5.5.2 Hardware Implementation

In this section, we investigate trade-offs among the variants of the SFM design that were introduced in Section 5.4.2.2. STMCM and the underlying LIDE-V approach allow one to perform such trade-off exploration, based on different combinations of high-level optimization techniques, in a systematic manner. In particular, STMCM allows the designer to focus on different strategies for instantiating, configuring, and coordinating different combinations of actor and buffer (edge) implementations, and eliminates the need for modification inside the actor and edge implementations. We exploited these advantages of STMCM when deriving the results presented in this section.

Table 5.6 depicts resource utilization data that is extracted from the post-place and route reports generated by the Xilinx Vivado tool using the targeted

Zynq Z-7020 SoC. From the results in Table 5.6, we see that the different design variants all exhibit similar levels of resource cost. The asynchronous designs  $SFM_a$  and  $SFM_{aCG}$  incur the highest resource costs due to the additional logic required by the CDC FIFOs. The number of BUFGs varies significantly among the different designs, depending on the number of clock domains and the number of clock gated actors.

Table 5.6: Resource utilization. In parentheses: the percentage of utilization with respect to the resources available on the targeted FPGA.

Available	LUTs 53200	REGs 106400	BUFGs 32	BRAMs 140	DSPs 220
$SFM$	5188 (9.75)	3472 (3.26)	1 (3.1)	11 (7.9)	13 (5.9)
$SFM_a$	5430 (10.20)	3687 (3.47)	2 (6.3)	11 (7.9)	13 (5.9)
$SFM_{CG}$	5206 (9.79)	3496 (3.29)	5 (15.6)	11 (7.9)	13 (5.9)
$SFM_{aCG}$	5479 (10.30)	3704 (3.48)	6 (18.8)	11 (7.9)	13 (5.9)
$SFM_h$	5170 (9.72)	3472 (3.26)	1 (3.1)	11 (7.9)	13 (5.9)
$SFM_{hCG}$	5198 (9.77)	3480 (3.27)	3 (9.4)	11 (7.9)	13 (5.9)
$SFM_{auto}$	5230 (9.83)	3472 (3.26)	1 (3.1)	11 (7.9)	13 (5.9)

Each of the implemented designs has been simulated in order to generate a switching activity file, which has been back-annotated to Vivado Power Estimation to extract power consumption data. Since the designs have different execution times, the energy consumption levels do not vary in the same proportions as the power consumption levels. Table 5.7 summarizes the power consumption, execution time and energy consumption of the six alternative designs.

In these experiments, the clock frequencies of the synchronous designs and of Region 1 (CLK 1) in the asynchronous designs are all set to 100 MHz, which is the maximum achievable frequency for the targeted platform. For Region 2 (CLK 2) in the asynchronous designs, the frequency is set to 5 MHz. This setting of 5 MHz is derived from the hardware profiling data (see Table 5.4) as 1/20 of CLK 1. These

clock frequencies are specified in Table 5.7 with the suffix  $_F$ , where  $F$  represents the frequency value in MHz.

Table 5.7: Dynamic power consumption, execution time and energy consumption of the different SFM variants. In parentheses: the percentage difference with respect to the baseline  $SFM$ .

	Power [ $mW$ ]	Time [ns]	Energy [ $\mu J$ ]
$SFM$	115	2329165	268
$SFM_{a_5}$	89 (-22.61)	2407300 (+3.354)	214 (-20.01)
$SFM_{CG}$	89 (-22.61)	2329245 (+0.003)	207 (-22.61)
$SFM_{aCG_5}$	88 (-23.48)	2408100 (+3.389)	212 (-20.89)
$SFM_h$	117 (+1.74)	2329155 (-0.000)	273 (+1.74)
$SFM_{hCG}$	105 (-8.70)	2329175 (+0.000)	244 (-8.70)
$SFM_{auto}$	113 (-1.74)	2329165 (+0.000)	263 (-1.74)

According to Table 5.7, the clock gated designs  $SFM_{CG}$  and  $SFM_{aCG_5}$  have the best capabilities for saving energy, reducing the total energy consumption by 22.61% and 20.89%, respectively. Design  $SFM_{aCG_5}$  saves less energy than  $SFM_{CG}$  since the former employs one more BUFG. Furthermore, in  $SFM_{aCG_5}$ , the actors in the slower domain (Region 2) are active for a relatively large portion of the execution time, and thus, they cannot be switched off for large proportions of time. In contrast, according to Table 5.4, the Deinterleave actor in Region 1 can be switched off for almost 90% of the total execution time.

The designs  $SFM_{aCG_5}$  and  $SFM_{a_5}$ , both of which employ two clock domains with CLK 1 at 100 MHz and CLK 2 at 5 MHz, have similar capabilities to save energy. The former design is slightly more energy efficient compared to the latter. The results for these two designs show that the energy saved by switching off the actors, when inactive, and also the saving of the unused logic in the CDC FIFOs counterbalance the energy overhead due to the additional circuitry.

As expected,  $SFM_h$  has a small amount of energy overhead due to the logic



necessary to encapsulate Sum1, Sum2 and Maxpool&ReLU into the hierarchical actor. The design  $SFM_{hCG}$ , among the clock gated designs, is not as advantageous as the previously analyzed designs in terms of energy saving. This is because even though it employs only three BUFs, the hierarchical actor is switched off only when none of the underlying actors are working. This means that, for instance, while Sum1 is active, the actors Sum2 and Maxpool&ReLU will have an active clock even when the actors are in an idle state (so that they keep wasting energy). Finally  $SFM_{auto}$  is the design with the smallest energy saving, only 1.74% compared to  $SFM$ . Even considering the same optimization technique (clock gating), the level on which it is applied turns out to be fundamental: at a low level (single flip-flops in  $SFM_{auto}$ ) only the dynamic power of a restricted number of gates can be saved. On the other hand, at a coarse-grain level (groups of dataflow actors in  $SFM_{CG}$ ), it is possible to act also on the clock tree, which is highly effective for improving power saving.

### 5.5.3 Hardware/Software Co-design Results

In this section, we investigate different hardware/software co-design configurations. As anticipated in Section 5.4.1, depending on the portion of the application that is accelerated in hardware and on the given requirements and constraints, different design choices regarding the hardware/software communication interface lead to different trade-offs between resource requirements and performance. For the SFM accelerator, we investigated several implementation and optimization solutions, ex-

ploring three key aspects: exploiting parallelism, communication interfaces and local buffering (see Section 5.4.3). In this section, by an *SFM accelerator*, we mean specifically a hardware accelerator.

Different software and hardware configurations that we explored in our co-design exploration are summarized as follows.

- *SW1* — The application runs in software on a single ARM core.
- *SW2* — The application runs in software by using both of the ARM cores on the target platform.
- *HW1* — A single-branch SFM accelerator is employed to execute the first convolutional layer.
- *HW2* — An SFM accelerator with two parallel branches. In this configuration, a local buffer is shared between the branches.
- *HW4* — An SFM accelerator with four parallel branches. Again, a local buffer is shared among the branches.

For multicore software implementations and hardware implementations with multiple branches, the layer or layers that are executed in parallel (i.e., intra-layer parallelism is exploited) are indicated in parentheses. Similarly, hardware configurations are annotated with *-mm* or *-s* depending, respectively, on whether a memory-mapped AXI-lite communication interface is used, or a FIFO-based AXI-stream interface is used.

For example, *SW2(L1, L2)* represents a software-only implementation in which layer 1 and layer 2 are executed in parallel. As another example, *SW2(L2)/HW2(L1)-*

mm represents a hardware/software implementation based on configurations SW2 and HW2; in this implementation, layer 2 is executed across multiple cores, layer 1 is parallelized in hardware with 2 parallel branches, and AXI-lite is used as the communication interface.

Note that the SFM accelerators are able to execute only the first convolutional layer. Thus, in all of the DNN system implementations, the accelerators are coupled with one of the software configurations.

### 5.5.3.1 Resource Costs of Accelerator Implementations

Table 5.8 depicts the resource occupancy in the targeted Zynq Z-7020 device for the different SFM accelerator implementations that we experimented with. As expected, a higher level of parallelism (going from HW1-mm to HW4-mm) requires more resources, and our experiments here help to quantify the associated trends. For example, fine-grained and computation-related resources (LUTs, REGs and DSPs) increase linearly with the number of parallel branches placed in the accelerator (about +100% with one more branch and about +300% with three more branches), while coarse-grained memory resources (BRAMs) exhibit a gentler slope. We expect that this gentler slope results because the primary BRAM-demanding module, the local buffer, is shared across parallel branches.

Table 5.8: Resource occupancy for different SFM accelerator implementations. In parentheses: the percentage of utilization with respect to the resources available on the targeted FPGA. The bottom part of the table depicts the percentage of variation with respect to HW1-mm.

Available	LUTs	REGs	BRAMs	DSPs
	53200	106400	140	220
HW1-mm	5395(10.14)	4668(4.39)	43 (30.71)	13 (5.91)
HW2-mm	10890 (20.47)	8197 (7.70)	54 (38.57)	26 (11.82)
HW4-mm	21474 (40.36)	16331(15.35)	76 (54.29)	52 (23.64)
HW2-mm	+101.85	+75.60	+25.58	+100.00
HW4-mm	+298.04	+249.85	+76.74	+300.00

### 5.5.3.2 Comparison of Co-Design Solutions

Table 5.9 presents performance results for different software-only and hardware/software solutions that we investigated using STMCM. In particular, the table reports the execution time in terms of milliseconds (ms) for different execution phases: reading the input file (column input), computing the first and the second layers, and computing the deep layers (Layers 3, 4 and 5). The table also reports the execution time of the overall application (prediction) for different degrees of software and hardware parallelism.

The reference time is given by the execution of the entire DNN application on a single ARM core (SW1), which is capable of completing the prediction in about 2.4 seconds. From this reference configuration, it is also possible to appreciate the computational load of the different application phases. The heaviest part is Layer 2, which is responsible for more than 65% of the overall execution time, while most of the remaining load is attributable to Layer 1 (around 25%), and to reading of the input file (about 5%). For this reason, software parallelization has been evaluated only on Layer 1 (SW2(L1)), and on both Layers 1 and 2 together (SW2(L1,L2)).

Table 5.9: Performance of different co-design solutions. The top part of the table depicts execution time in milliseconds (ms). The bottom part depicts the percentage of execution time variation for each configuration with respect to SW1.

	input	Layer			prediction
		1	2	3:5	
SW1	118.9	640.3	1594.7	34.4	2388.2
SW2(L1)	118.7	368.3	1609.8	34.0	1639.5
SW2(L1,L2)	117.4	354.7	842.1	33.8	1348.0
SW2(L2)/HW1(L1)-mm	118.9	118.5	856.7	35.4	1129.5
SW2(L2)/HW2(L1)-mm	118.4	74.6	866.5	35.1	1094.5
SW2(L2)/HW4(L1)-mm	117.9	54.5	859.0	35.4	1066.8
SW2(L1)	-0.13	-42.48	+0.95	+1.12	-10.75
SW2(L1,L2)	-1.22	-44.61	-47.19	-1.72	-43.56
SW2(L2)/HW1(L1)-mm	-0.00	-81.49	-46.28	+2.89	-52.71
SW2(L2)/HW2(L1)-mm	-0.40	-88.36	-45.66	+2.09	-54.17
SW2(L2)/HW4(L1)-mm	-0.81	-91.48	-46.13	+2.85	-55.33

The execution time needed by each of the major execution phases is almost halved when two cores are adopted. A precise 50% reduction is not reached because of the software overhead necessary to manage multitasking. With software parallelization only, the overall execution time is reduced to 1.13 seconds, about 44% less than the SW1 configuration. Hardware acceleration and related parallelization are only applied to the first convolutional layer, while only software parallelization is applied to Layer 2. If we consider only the execution time of layer 1, then SW2(L2)/HW1(L1) reduces execution time by more than 80% compared to SW1, and more than 65% compared to SW2(L1,L2).

If multiple branches of Layer 1 are processed in parallel, the hardware accelerator achieves further performance benefits — a time saving up to 88% for a 2-branch configuration (SW2(L2)/HW2(L1)-mm), and up to 91% for a 4-branch configuration (SW2(L2)/HW4(L1)-mm). These performance improvements are with respect

to SW1. Note that the speed-up obtained by doubling the number of branches (going from 1 to 2 and from 2 to 4) is less than 2 in either case (1.6 from 1 to 2 and 1.4 from 2 to 4). This is due to the software overhead related to managing multiple branches. Due to the limited computational load of Layer 1, the benefits of hardware acceleration and parallelization on the overall system are somewhat limited. The best solution, SW2(L2)/HW4(L1)-mm, requires 1.07 seconds to perform the whole application, 55% less than a full software execution on a single core (SW1) and 21% less than a full software execution on two cores (SW2(L1,L2)).

Another aspect that has been studied in our co-design experiments is the interfacing between system components. As discussed in Section 5.4.3.2, the adopted communication interface between software and hardware portions of a design can have a significant impact on overall system performance. In our co-design experiments, we have applied two very interfaces — mm-lite and stream, which are discussed in Section 5.4.3.2.

Table 5.10 helps to understand differences between the resource costs of these two interfaces. The first row of this table shows resource availability on the target platform. The second and third rows show resource costs for the HW1-mm accelerator, and HW1-s accelerator. The fourth and fifth rows show resource costs for FIFO and DMA modules (external to the accelerator) that are necessary for the stream interface. The sixth row shows total resource costs induced by use of the stream interface (the sums of the costs in the preceding three rows). The last two rows of the table represent percentage increases in resource costs relative to the HW1-mm accelerator.

From Table 5.10, we see that the HW1-mm and HW1-s accelerators alone require approximately the same amount of resources: HW1-s requires 7.21% more LUTs and 6.66% less REGs compared to HW1-mm. However, when the overhead due to the DMA and FIFO modules necessary for AXI-stream communication is considered, significantly more resources are required when the stream interface is used: about 38% more LUTs and REGs are required by the overall stream design ((1)+(2)+(3)), while over 50% more BRAM cost is incurred.

Table 5.10: Differences in resource costs between communication interfaces when applied to HW1. In parentheses: percentage of utilization with respect to the resources available on the targeted FPGA. The bottom part of the table depicts the percentage utilization variation with respect to HW1-mm.

Available	LUTs 53200	REGs 106400	BRAMs 140	DSPs 220
HW1-mm	5395(10.14)	4668(4.39)	43 (30.71)	13 (5.91)
(1) HW1-s	5784 (10.87)	4357 (4.09)	43 (30.71)	13 (5.91)
(2) FIFOs (stream)	212 (0.40)	242 (0.23)	10 (7.14)	0 (0.00)
(3) DMA (stream)	1490 (2.80)	1881 (1.77)	3 (2.14)	0 (0.00)
(1)+(2)+(3)	7486 (14.07)	6480 (6.09)	56 (40.00)	13 (5.91)
HW1-s	+7.21	-6.66	+0.00	+0.00
(1)+(2)+(3)	+38.75	+38.81	+53.49	+0.00%

To make the stream interface a useful option in our system design, its significant increase in resource costs should be accompanied by tangible advantages in execution performance. Table 5.11 shows results pertaining to the impact of the communication interface on execution time. In order to better expose the effects of the selected communication interface, details on data transfers (input, convolution coefficients and outputs) between the hardware (accelerator) and software subsystems is reported. For the HW1-s design, two different sets of results are reported depending on whether program data is directly accessible by the DMA engine. For one set, the program data is located in a memory that is not directly accessible

by the DMA. This scenario corresponds to the design that we have implemented. It requires an additional copy of the program data in a memory that is accessed directly by the DMA. For the other set, the program data is located in a memory that is directly accessible by the DMA. This set is indicated in Table 5.11 using the annotation *HW1-s dir*. We have not implemented HW1-s dir; instead, we have estimated the corresponding results to gain some idea about the maximum achievable performance. Details on the estimation approach are omitted for brevity.

Table 5.11: Results pertaining to the impact of the communication interface on execution time. The top part of the table depicts the execution time of the different DNN application steps. The bottom part depicts the execution time variation of each configuration with respect to HW1-mm.

	File input [ms]	Layer 1				Layer 2 [ms]	Layers 3, 4, 5 [ms]	Prediction [ms]
		input tx [ $\mu$ s]	coeffs tx [ $\mu$ s]	output tx [ $\mu$ s]	total [ms]			
HW1-mm	118.9	5222	15	2339	118.5	856.7	35.4	1129.5
HW1-s	117.5	854	5	2333	114.6	864.3	34.9	1131.3
HW1-s dir	118.3	418	3	2333	114.1	869.5	35.0	113.7
HW1-s	-1.17	-83.65	-66.67	-0.26	-3.27	+0.87	-1.39	+0.16
HW1-s dir	-0.49	-92.00	-80.00	-0.26	-3.69	+1.49	-1.22	+0.66

The results in Table 5.11 demonstrate the utility of the resource-hungry HW1-s design, and quantify its clear ability to outperform HW1-mm. In particular, the input data and transmission of convolution coefficients are respectively about 84% and 67% faster when the AXI-stream protocol is adopted. This leads to an estimated time saving of up to 92% and 80%, respectively, when the DMA has direct access to the program data (HW1-s dir).

On the other hand, the output data transmission time is the same among all of the reported configurations. We expect that this is because the outputs are produced in a row-by-row fashion (48 data units at a time), and the timing of output production is determined by the computation latency, which is greater than the



communication latency for all of the interfacing configurations. However, looking at the total Layer 1 and DNN application execution times, we see that the advantages of adopting the stream interface are no longer visible. Indeed, for the considered SFM accelerator, the input data is transmitted only during the first branch execution due to our use of local buffering. Additionally, even though the coefficients are transmitted for each branch, their transmission requires a relatively small amount of time.

These results involving communication interface selection illustrate the importance of comprehensive system-level evaluation of alternative design options, which is one of the key parts of the design process that is facilitated by STMCM.

## 5.6 Conclusions

In this chapter, we have introduced a design methodology, called the STMC Methodology or STMCM, and an integrated set of tools and libraries that support the application of this methodology. STMCM is developed to assist designers of signal processing systems in exploring complex design alternatives that span multiple implementation scales, platform types, and dataflow modeling techniques. We have demonstrated the capabilities of STMCM through a detailed case study involving a deep neural network (DNN) for vehicle classification. The demonstration encompasses dataflow-based application modeling, profiling, embedded software optimization, hardware accelerator design, hardware/software co-design, and hardware/software interface design, all in the context of mapping the given DNN

into an efficient implementation on a resource-constrained, system-on-chip platform. Through this case study, it is shown how STMCM provides a unified, model-based framework for conducting comprehensive empirical evaluations of diverse hardware/software design alternatives. Through its application of lightweight dataflow techniques, STMCM is complementary to dataflow tools that emphasize specialized design flows and high degrees of automation.

## Chapter 6

### Conclusions and Future Work

In this chapter, we first summarize the contributions presented in the previous chapters of this thesis. Then, we enumerate several directions for future research.

#### 6.1 Conclusions

In this thesis, we have developed novel design space exploration methods using compact system level models, and dataflow-based techniques for design and implementation of signal processing systems. The models and methods developed are geared toward digital signal processing (DSP) systems having complex, multi-dimensional design spaces involving conflicting objectives and constraints. Based on these contributions, we have developed new signal processing systems and associated design optimization techniques for relevant applications in adaptive wireless communications and deep learning.

First, we have presented an adaptive digital predistortion (DPD) system that reconfigures itself across multiple Pareto-optimized DPD configurations. Different from most works in the literature on DPD optimization, which consider a single objective, namely, the adjacent channel power ratio (ACPR), we have taken ACPR, error vector magnitude (EVM), and power consumption jointly into consideration. We have taken these three design evaluation metrics into account to treat DPD

optimization as a multiobjective optimization problem. We have developed efficient methods to (1) extract Pareto-optimized DPD configurations at design time, and then (2) integrate these configurations with a run-time controller in an optimized adaptive DPD (OAD) system. The OAD controller switches among different DPD configurations to satisfy constraints on ACPR and EVM with optimized power consumption. We have performed extensive simulations to validate that the proposed OAD system can support highly diverse signal processing trade-offs and significantly outperform a fixed DPD configuration under time-varying operational conditions.

Second, we have presented a new design framework, called the Evolutionary Adaptive DPD Implementation (EADI) Framework, for systematic derivation of Pareto-optimized DPD configurations that can be applied to adaptive DPD implementations. We have shown the effectiveness of the EADI Framework by applying it to derive a novel DPD architecture, called the adaptive, dataflow-based DPD architecture (ADDA). The design of this architecture applies multiobjective evolutionary algorithms to derive Pareto-optimized DPD configurations. A reconfigurable dataflow graph implementation is then used to adapt the DPD system efficiently among the derived configurations at run-time. We have presented experiments to show the effectiveness of the proposed EADI framework in generating efficient DPD configurations subject to multidimensional constraints with performance exceeding the OAD scheme.

Third, we have presented the Hierarchical MDP framework for Compact System-level Modeling (HMCSM), and its application to the implementation of adaptive embedded signal processing systems. HMCSM provides a structured design methodol-

ogy that integrates model-based, reconfigurable embedded signal processing system design using parameterized dataflow models, and Markov decision process (MDP) methods for optimal control policy generation. HMCSM enables systematic derivation of dynamic reconfiguration policies based on functional requirements and operational constraints. The effectiveness of the framework has been demonstrated through a detailed case study on the design of a channelizer for wireless communications.

Finally, we have developed a compact set of retargetable application programming interfaces (APIs) and associated libraries and software tools for dataflow-based digital hardware design. We have also built upon these new design components to develop a dataflow-based methodology, along with supporting software tools and libraries, for integrated hardware/software co-design and design optimization of signal processing systems. Our developed APIs, libraries, and tools extend the previously-developed lightweight dataflow environment (LIDE) tool with new capabilities for model-based design and implementation of digital hardware systems. Our contributions to LIDE and hardware/software co-design methodologies have been demonstrated on a dataflow-based, deep neural network (DNN) implementation for vehicle classification that is streamlined for real-time operation on embedded system-on-chip (SoC) devices.

## 6.2 Future Work

Various useful directions for future work are motivated from the developments of this thesis. These include the following.

1. The hierarchical structure of the MDP architecture proposed in Chapter 4 is developed manually. An interesting future direction is the development of tools to help automate the factoring and decomposition processes involved in this architecture, and to further investigate optimization issues involved in these processes.
2. In the current implementation of MDP methods in the HMCSM Framework, the optimal policies are pre-computed offline. A useful direction for future work is to study schemes for incorporating real-time MDP solvers into the framework. Such solvers would be capable of allowing the MDP and generated policy to adapt dynamically based on learned characteristics of the operating environment. Moreover, the study of methods for dataflow modeling and design optimization of real-time, embedded MDP solvers is an interesting direction for future work to improve the efficiency of this class of solvers.
3. We have employed parameterized dataflow modeling in our design of the HMCSM Framework. Various other kinds of parametric dataflow modeling methods have been introduced in recent years, such as Boolean parametric dataflow [42] and the parameterized and interfaced dataflow meta-model [43]. Adapting and experimenting with the HMCSM Framework based on para-

metric dataflow approaches such as these is an interesting direction for future work.

4. In relation to the work presented in Chapter 5, interesting directions for future work include development of automation support for the lightweight dataflow models and methods introduced in this chapter, adaptation of lightweight dataflow for use with other hardware description languages (HDLs) beyond Verilog, defining automatic system optimization support for extracting Instruction Level Parallelism (ILP) from lightweight dataflow models, and the targeting of application specific integrated circuit (ASIC) technology.

## Bibliography

- [1] 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*, Washington DC, USA, November 2010, pp. 640–645.
- [2] E. A. Lee and T. M. Parks, “Dataflow process networks,” *Proceedings of the IEEE*, pp. 773–799, May 1995.
- [3] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds., *Handbook of Signal Processing Systems*, Springer, second edition, 2013.
- [4] S. Lin, Y. Liu, K. Lee, L. Li, W. Plishker, and S. S. Bhattacharyya, “The DSPCAD framework for modeling and synthesis of signal processing systems,” in *Handbook of Hardware/Software Codesign*, S. Ha and J. Teich, Eds., pp. 1–35. Springer, 2017.
- [5] L. Anttila, P. Händel, and M. Valkama, “Joint mitigation of power amplifier and I/Q modulator impairments in broadband direct-conversion transmitters,” *IEEE Transactions on Microwave Theory and Techniques*, vol. 58, no. 4, pp. 730–739, 2010.
- [6] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, 2016.
- [7] Y. Jia, “Delivering real-time AI in the palm of your hand,” <https://code.fb.com/android/delivering-real-time-ai-in-the-palm-of-your-hand/>, 2016, Posted in November 2016, visited on October 12, 2018.
- [8] L. Li, A. Ghazi, J. Boutellier, L. Anttila, M. Valkama, and S. S. Bhattacharyya, “Design space exploration and constrained multiobjective optimization for digital predistortion systems,” in *Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors*, London, England, July 2016, pp. 182–185.
- [9] C. Çiflikli and A. Yapıcı, “Genetic algorithm optimization of a hybrid analog/digital predistorter for RF power amplifiers,” *Analog Integrated Circuits and Signal Processing*, vol. 52, no. 1, pp. 25–30, 2007.
- [10] L. Ding et al., “Compensation of frequency-dependent gain/phase imbalance in predistortion linearization systems,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 55, no. 1, pp. 390–397, 2008.
- [11] A. Ghazi et al., “Low power implementation of digital predistortion filter on a heterogeneous application specific multiprocessor,” in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Florence, Italy, 2014, pp. 8391–8395.



- [12] M. Nizamuddin, *Predistortion for Nonlinear Power Amplifiers with Memory*, Ph.D. thesis, Virginia Polytechnic Institute and State University, December 2002.
- [13] J. A. Sills and R. Sperlich, “Adaptive power amplifier linearization by digital pre-distortion using genetic algorithms,” in *Proceedings of the Radio and Wireless Conference*, 2002, pp. 229–232.
- [14] D. S. Hilborn, S. P. Stapleton, and J. K. Cavers, “An adaptive direct conversion transmitter,” *IEEE Transactions on Vehicular Technology*, vol. 43, no. 2, pp. 223–233, 1994.
- [15] R. Sperlich, J. A. Sills, and J. S. Kenney, “Power amplifier linearization with memory effects using digital pre-distortion and genetic algorithms,” in *Proceedings of the Radio and Wireless Conference*, 2004, pp. 355–358.
- [16] A. H. Abdelhafiz, O. Hammi, A. Zerguine, A. T. Al-Awami, and F. M. Ghanouchi, “A PSO based memory polynomial predistorter with embedded dimension estimation,” *IEEE Transactions on Broadcasting*, vol. 59, no. 4, pp. 665–673, 2013.
- [17] K. Freiberger, M. Wolkerstorfer, H. Enzinger, and C. Vogel, “Digital predistorter identification based on constrained multi-objective optimization of WLAN standard performance metrics,” in *Proceedings of the International Symposium on Circuits and Systems*, 2015, pp. 862–865.
- [18] N. K. Bambha and S. S. Bhattacharyya, “A joint power/performance optimization technique for multiprocessor systems using a period graph construct,” in *Proceedings of the International Symposium on System Synthesis*, Madrid, Spain, September 2000, pp. 91–97.
- [19] L. Li, A. Ghazi, J. Boutellier, L. Anttila, M. Valkama, and S. S. Bhattacharyya, “Evolutionary multiobjective optimization for digital predistortion architectures,” in *Proceedings of the International Conference on Cognitive Radio Oriented Wireless Networks*, Grenoble, France, May 2016, pp. 498–510.
- [20] L. Li, A. Ghazi, J. Boutellier, L. Anttila, M. Valkama, and S. S. Bhattacharyya, “Evolutionary multiobjective optimization for adaptive dataflow-based digital predistortion architectures,” *EAI Endorsed Transactions on Cognitive Communications*, vol. 3, no. 10, pp. 1–9, February 2017.
- [21] E. Zitzler, *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*, Ph.D. thesis, Swiss Federal Institute of Technology (ETH) Zurich, December 1999.
- [22] 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*, 2010, pp. 640–645.

- [23] L.-H. Wang et al., “Dataflow modeling and design for cognitive radio networks,” in *Proceedings of the International Conference on Cognitive Radio Oriented Wireless Networks*, 2013, pp. 196–201.
- [24] A. Konak, D. W. Coit, and A. E. Smith, “Multi-objective optimization using genetic algorithms: A tutorial,” *Reliability Engineering & System Safety*, vol. 91, no. 9, pp. 992–1007, 2006.
- [25] K. Sindhya, K. Miettinen, and K. Deb, “A hybrid framework for evolutionary multi-objective optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 17, no. 4, pp. 495–511, 2013.
- [26] D. Llamocca and M. Pattichis, “Dynamic energy, performance, and accuracy optimization and management using automatically generated constraints for separable 2D FIR filtering for digital video processing,” *Transactions on Reconfigurable Technology and Systems*, vol. 7, no. 4, 2015, Article No. 31.
- [27] L. Li, A. Sapio, J. Wu, Y. Liu, K. Lee, M. Wolf, and S. S. Bhattacharyya, “Design and implementation of adaptive signal processing systems using Markov decision processes,” in *Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors*, Seattle, Washington, July 2017, pp. 170–175.
- [28] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, 1994.
- [29] S. Chen, Y. Wang, and M. Pedram, “A semi-Markovian decision process based control method for offloading tasks from mobile devices to the cloud,” in *Proceedings of the IEEE Global Telecommunications Conference*, 2013, pp. 2885–2890.
- [30] A. Munir and A. Gordon-Ross, “An MDP-based application oriented optimal policy for wireless sensor networks,” in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, 2009, pp. 183–192.
- [31] C. Boutilier, R. Dearden, and M. Goldszmidt, “Exploiting structure in policy construction,” in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1995, pp. 1104–1111.
- [32] L. Benini, A. Bogliolo, G. A. Paleologo, and G. De Micheli, “Policy optimization for dynamic power management,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 6, pp. 742–760, June 1999.
- [33] A. Jonsson and A. Barto, “Causal graph based decomposition of factored MDPs,” *Journal of Machine Learning Research*, vol. 7, pp. 2259–2301, 2006.

- [34] A. Sapio, M. Wolf, and S. S. Bhattacharyya, “Compact modeling and management of reconfiguration in digital channelizer implementation,” in *Proceedings of the IEEE Global Conference on Signal and Information Processing*, Washington, D.C., December 2016, pp. 595–599.
- [35] S. J. Darak, A. P. Vinod, R. Mahesh, and E. M-K. Lai, “A reconfigurable filter bank for uniform and non-uniform channelization in multi-standard wireless communication receivers,” in *Proceedings of the International Conference on Telecommunications*, 2010, pp. 951–956.
- [36] O. Sigaud and O. Buffet, Eds., *Markov Decision Processes in Artificial Intelligence*, Wiley, 2010.
- [37] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, “Cyclo-static dataflow,” *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, February 1996.
- [38] B. D. Theelen, M. C. W. Geilen, T. Basten, J. P. M. Voeten, S. V. Gheorghita, and S. Stuijk, “A scenario-aware data flow model for combined long-run average and worst-case performance analysis,” in *Proceedings of the International Conference on Formal Methods and Models for Codesign*, July 2006.
- [39] E. A. Lee and D. G. Messerschmitt, “Synchronous dataflow,” *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.
- [40] B. Bhattacharya and S. S. Bhattacharyya, “Parameterized dataflow modeling for DSP systems,” *IEEE Transactions on Signal Processing*, vol. 49, no. 10, pp. 2408–2421, October 2001.
- [41] B. Bhattacharya and S. S. Bhattacharyya, “Quasi-static scheduling of reconfigurable dataflow graphs for DSP systems,” in *Proceedings of the International Workshop on Rapid System Prototyping*, Paris, France, June 2000, pp. 84–89.
- [42] V. Bebelis, P. Fradet, A. Girault, and B. Lavigueur, “BPDF: A statically analyzable dataflow model with integer and Boolean parameters,” in *Proceedings of the International Workshop on Embedded Software*, 2013, pp. 1–10.
- [43] K. Desnos, M. Pelcat, J.-F. Nezan, S. S. Bhattacharyya, and S. Aridhi, “PiMM: Parameterized and interfaced dataflow meta-model for MPSoCs runtime reconfiguration,” in *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2013, pp. 41–48.
- [44] P. L. Fackler, “MDPSOLVE a MATLAB toolbox for solving Markov decision problems with dynamic programming — user’s guide,” Tech. Rep., North Carolina State University, January 2011.
- [45] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, “Internet of things for smart cities,” *IEEE Internet of Things Journal*, vol. 1, no. 1, pp. 22–32, 2014.

- [46] T. Fanni, L. Li, T. Viitanen, C. Sau, R. Xie, F. Palumbo, L. Raffo, H. Huttunen, J. Takala, and S. S. Bhattacharyya, “Hardware design methodology using lightweight dataflow and its integration with low power techniques,” *Journal of Systems Architecture*, vol. 78, pp. 15–29, August 2017.
- [47] L. Li, T. Fanni, T. Viitanen, R. Xie, F. Palumbo, L. Raffo, H. Huttunen, J. Takala, and S. S. Bhattacharyya, “Low power design methodology for signal processing systems using lightweight dataflow techniques,” in *Proceedings of the Conference on Design and Architectures for Signal and Image Processing*, Rennes, France, October 2016, pp. 81–88.
- [48] S. Ha and J. Teich, Eds., *Handbook of Hardware/Software Codesign*, Springer, 2017.
- [49] 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*, Samos, Greece, July 2008, pp. 157–166.
- [50] J. T. Buck and E. A. Lee, “Scheduling dynamic dataflow graphs using the token flow model,” in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April 1993.
- [51] J. Eker and J. W. Janneck, “Dataflow programming in CAL — balancing expressiveness, analyzability, and implementability,” in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, 2012, pp. 1120–1124.
- [52] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet, “Orcc: multimedia development made easy,” in *Proceedings of the ACM International Conference on Multimedia*, 2013, pp. 863–866.
- [53] J. Sérot, F. Berry, and S. Ahmed, “CAPH: A language for implementing stream-processing applications on FPGAs,” in *Embedded Systems Design with FPGAs*, P. Athanas, D. Pnevmatikatos, and N. Sklavos, Eds. Springer, 2013.
- [54] J. Mcallister, R. Woods, R. Walke, and D. Reilly, “Multidimensional DSP core synthesis for FPGA,” *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 43, no. 2–3, June 2006.
- [55] 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*, 2009, pp. 1552–1555.
- [56] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubühr, A. Deyhle, A. Hadert, and J. Teich, “A SystemC-based design methodology for digital signal processing systems,” *EURASIP Journal on Embedded Systems*, vol. 2007, pp. Article ID 47580, 22 pages, 2007.

- [57] S. Lin, Y. Liu, W. Plishker, and S. S. Bhattacharyya, “A design framework for mapping vectorized synchronous dataflow graphs onto CPU–GPU platforms,” in *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, Sankt Goar, Germany, May 2016, pp. 20–29.
- [58] S. Casale-Brunet, M. Wiszniewska, E. Bezati, M. Mattavelli, J. W. Janneck, and M. Canale, “TURNUS: An open-source design space exploration framework for dynamic stream programs,” in *Proceedings of the Conference on Design and Architectures for Signal and Image Processing*, 2014, pp. 1–2.
- [59] C. Sau et al., “Automated design flow for multi-functional dataflow-based platforms,” *Journal of Signal Processing Systems*, pp. 1–23, 2015, DOI: 10.1007/s11265-015-1026-0.
- [60] F. Palumbo, T. Fanni, C. Sau, and P. Meloni, “Power-awareness in coarse-grained reconfigurable multi-functional architectures: a dataflow based strategy,” *Journal of Signal Processing Systems*, vol. 87, no. 1, pp. 81–106, 2017.
- [61] T. Fanni, C. Sau, P. Meloni, L. Raffo, and F. Palumbo, “Power and clock gating modelling in coarse grained reconfigurable systems,” in *Proceedings of the ACM International Conference on Computing Frontiers*, 2016, pp. 384–391.
- [62] S. C. Brunet, E. Bezati, C. Alberti, M. Mattavelli, E. Amaldi, and J. W. Janneck, “Multi-clock domain optimization for reconfigurable architectures in high-level dataflow applications,” in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, 2013, pp. 1796–1800.
- [63] E. Bezati, S. C. Brunet, M. Mattavelli, and J. W. Janneck, “Coarse grain clock gating of streaming applications in programmable logic implementations,” in *Proceedings of the Electronic System Level Synthesis Conference*, 2014, pp. 1–6.
- [64] 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*, Monterey, California, June 2008, pp. 17–23.
- [65] 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*, Nice, France, April 2009, pp. 111–116.
- [66] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, “System-level design: orthogonalization of concerns and platform-based design,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, December 2000.
- [67] C. Shen, L. Wang, I. Cho, S. Kim, S. Won, W. Plishker, and S. S. Bhattacharyya, “The DSPCAD lightweight dataflow environment: Introduction

- to LIDE version 0.1,” Tech. Rep. UMIACS-TR-2011-17, Institute for Advanced Computer Studies, University of Maryland at College Park, 2011, <http://hdl.handle.net/1903/12147>.
- [68] S. Lin, J. Wu, and S. S. Bhattacharyya, “Memory-constrained vectorization and scheduling of dataflow graphs for hybrid CPU-GPU platforms,” *ACM Transactions on Embedded Computing Systems*, 2018, To appear; preprint available at <http://arxiv.org/abs/1711.11154>.
- [69] R. Xie, H. Huttunen, S. Lin, S. S. Bhattacharyya, and J. Takala, “Resource-constrained implementation and optimization of a deep neural network for vehicle classification,” in *Proceedings of the European Signal Processing Conference*, Budapest, Hungary, August 2016, pp. 1862–1866.
- [70] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, A. J. M. Moonen, M. J. G. Bekooij, B. D. Theelen, and M. R. Mousavi, “Throughput analysis of synchronous data flow graphs,” in *Proceedings of the International Conference on Application of Concurrency to System Design*, June 2006.
- [71] C. E. Cummings, “Simulation and synthesis techniques for asynchronous FIFO design,” in *Proceedings of the Synopsys Users Group Conference*, 2002.
- [72] H. Huttunen, F. Yancheshmeh, and K. Chen, “Car type recognition with deep neural networks,” *ArXiv e-prints*, 2016, arXiv:1602.07125v2, To appear in proceedings of IEEE Intelligent Vehicles Symposium 2016.
- [73] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, “Microsoft COCO: Common objects in context,” in *Proceedings of the European Conference on Computer Vision*, 2014, pp. 740–755.
- [74] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “ImageNet: A large-scale hierarchical image database,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.
- [75] T. Chen et al., “DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning,” in *Symposium on Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 269–284.
- [76] P. K. Murthy and S. S. Bhattacharyya, “Shared buffer implementations of signal processing systems using lifetime analysis techniques,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 2, pp. 177–198, February 2001.
- [77] H. Oh and S. Ha, “Memory-optimized software synthesis from dataflow program graphs with large size data samples,” *EURASIP Journal on Applied Signal Processing*, vol. 2003, no. 6, May 2003.

- [78] K. Desnos, M. Pelcat, J.-F. Nezan, and Slaheddine Aridhi, “Buffer merging technique for minimizing memory footprints of synchronous dataflow specifications,” in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 2015, pp. 1111–1115.
- [79] H.-J. Koch, *The Userspace I/O HOWTO*, Linutronix, 2006.
- [80] J. Silva, V. Sklyarov, and I. Skliarova, “Comparison of on-chip communications in Zynq-7000 all programmable systems-on-chip,” *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 31–34, 2015.