ABSTRACT

Title of dissertation: SYSTEMATIC EXPLORATION OF TRADE-OFFS BETWEEN APPLICATION THROUGHPUT AND HARDWARE RESOURCE REQUIREMENTS IN DSP SYSTEMS

Hojin Kee, Doctor of Philosophy, 2010

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

Dataflow has been used extensively as an efficient model-of-computation to analyze performance and resource requirements in implementing DSP algorithms on various target architectures. Although various software synthesis techniques have been widely studied in recent years, there is a distinct lack of efficient synthesis techniques in the literature for systematically mapping dataflow models into efficient hardware implementations. In this thesis, we explore three different aspects that contribute to the development of a powerful dataflow-based hardware synthesis framework:

1. Systematic generation of 1D/2D FFT implementation on field programmable gate arrays (FPGAs). The fast Fourier transform (FFT) is one of the most widely-used and important signal processing functions. However, FFT computation generally becomes a major bottleneck for overall system performance due to its high computational requirements. We propose a systematic approach for synthesizing FPGA implementations of one- and two-dimensional (1D and 2D) FFT computations, and

rigorously exploring trade-offs between cost (in terms of FPGA resource requirements) and performance (in terms of throughput). Our approach provides an efficient hardware synthesis framework that can be customized to specific design constraints. In our FFT synthesis approach, we apply two orthogonal techniques in FPGA implementation to realize data-parallelism and parallel processing in FFT computation, respectively. These techniques can be applied to various 1D FFT algorithms, including Radix-2 and Radix-4 algorithms, and extended naturally and efficiently to 2D FFT implementation.

2. Buffer optimization under self-timed execution. Self-timed execution is known to provide the maximum achievable throughput when mapping DSP dataflow graphs into hardware under certain technical constraints. Throughput-constrained buffer minimization under self-timed execution is a key question in efficient hardware synthesis for practical design scenarios. Previous approaches to this problem have suffered from high worst case complexity or loose buffer bounds, which lead to inefficient resource utilization. In this thesis, we integrate a novel constraint into traditional self-timed execution to obtain a modified form of self-timed execution, which we call *MSTE* (Modified Self-Timed Execution). We show that MSTE greatly improves the efficiency with which we can accurately analyze and optimize hardware configurations of dataflow graphs, and furthermore, the additional execution constraints imposed in MSTE result in relatively minor performance overhead. Based on MSTE, we explore novel methods for self-timed analysis and associated techniques for buffer optimization subject to given throughput constraints.

3. Efficient scheduling techniques for hardware synthesis. The methodology of *parameterized dataflow modeling*, which has been developed in prior work, allows for modeling of dynamic application behavior in DSP applications without excessively compromising key analysis properties of existing static dataflow modeling techniques — in particular, properties associated with compile-time predictability and the potential for rigorous optimization. Building on the existing theory of parameterized dataflow, we develop a novel scheduling algorithm for mapping parameterized dataflow graphs into FPGA implementations. Our mapping approach is geared towards minimizing buffer requirements based on given throughput constraints, and is useful in optimizing the cost of DSP systems that must satisfy predefined real-time requirements.

SYSTEMATIC EXPLORATION OF TRADE-OFFS BETWEEN APPLICATION THROUGHPUT AND HARDWARE RESOURCE REQUIREMENTS IN DSP SYSTEMS

by

Hojin Kee

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
2010

Advisory Committee
Professor Shuvra S. Bhattacharyya, Chair/Advisor
Professor Gang Qu
Professor Raj Shekhar
Professor Ramani Duraiswami
Professor Andrew Harris, Deans Representative

Dedicated to

my parents
my wife & our son

Acknowledgments

First of all, I would like to thank Dr. Shuvra S. Bhattacharyya for giving me an invaluable opportunity to work on challenging and interesting projects. While working with him, I have learned how to define an engineering problem, approach in solving the given problem, and describe my approach technically in presentations and papers. Furthermore, his positive advice and encouragement on my works motivated me in doing my best, and became a great lesson when I co-worked with my colleagues.

I also would like to thank the people of National Instruments, including Dr. Jacob Kornerup, Newton Petersen, Minhaz Khan, Ben Weidman, Dr. Ian Wong, Dr. Kaushik Ravindran, and Yon Rao. It has been my pleasure to work and interact with good and smart engineers from National Instruments. Also, it was always exciting to apply my academic solutions to cutting-edge applications used in the real-engineering field.

It is my pleasure to thank the members of the committee, including Dr. Gang Qu, Dr. Raj Shekhar, Dr. Ramani Duraiswami, and Dr. Harris for guiding me to complete this thesis with constructive feedbacks and helpful discussions.

My gratitude goes to the DSPCAD research group, including Will, Chung-ching, Ruirui, Nimish, Hsing-Huang, George, Inkeun, Scott, and Soujanya. I also feel grateful to all my friends here for their generous friendship, which became a great support in completing my Ph.D study.

I would like to give my deepest gratitude to my parents. “You can do it because you are our proud son.” Their endless support and love motivated me in moving forward to become their proud son when I was having hard times. Looking back, it was one of the

best motivation to achieve this milestone in my academic career.

Finally, I thank my beloved wife, who accompanied me from the beginning to the end of this thesis with full support, and my new born son, who is crying at this very moment.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Dataflow-based digital signal processing (DSP) system design methods include three levels of design. Each level of design is closely related to the other levels, as illustrated in Fig. 1.1. In this thesis, we explore techniques at each of these design levels, and corresponding advances that are applicable to DSP system design flows at each of these levels. Furthermore, novel trade-offs of performance enhancement techniques that are enabled by our techniques are considered jointly to realize optimized DSP implementations subject to given constraints on performance and resource requirements. We specifically consider techniques for efficient implementation of DSP-based application representations on field programmable gate array (FPGA) devices, which are attractive targets for rapid prototyping and high performance signal processing in many application contexts.

For actor-level design, we explore trade-offs between throughput and resource requirements (hardware cost) in implementing computational modules for the fast Fourier transform (FFT), which is a fundamental function in many signal processing applications. Due to its computational complexity — $O(NlogN)$, where N the number of inputs — and the large amount of data that must be processed, FFT computation often becomes a major bottleneck for overall system performance. In this thesis, we develop a systematic approach for generating a cost–efficient, FPGA–based FFT implementation based on a

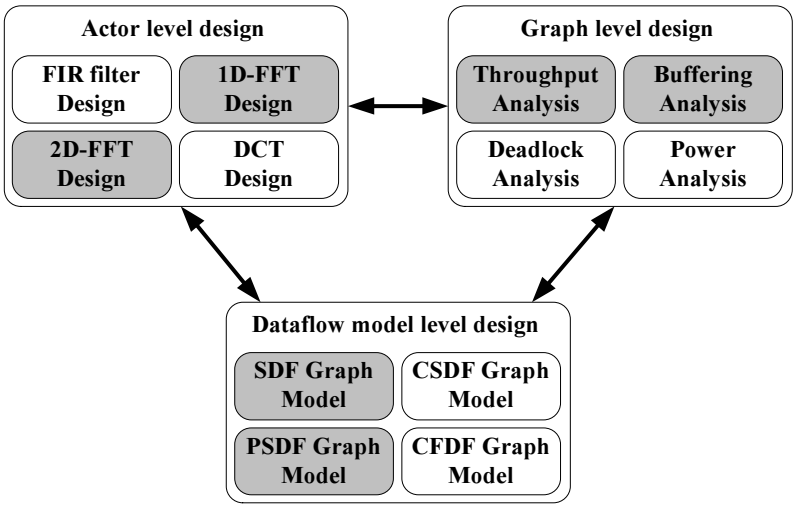


Figure 1.1: Relationships among different levels of dataflow-based design methods for DSP systems.

designer–specified throughput requirement.

In dataflow-based system design, functional blocks and communication channels for transferring data between adjacent blocks are modeled as graph vertices (*actors*) and edges, respectively. When mapping dataflow graph edges into storage locations, care must be taken to make effective use of limited storage locations (e.g., on-chip memory in programmable digital signal processors, and block RAM and distributed memory in FPGAs). However, reducing the storage space for transferring data between actors may result in decreased throughput due to idle time that is required to prevent buffer overflow — as buffers become smaller, the frequency and duration for such overflow-avoiding idle time generally increases, which leads to decreased throughput. The limited amounts of storage available in DSP implementation targets, and the importance of meeting real-time performance constraints motivate the goal of *throughput-constrained buffer minimization* for SDF graphs. In this thesis, we study this problem in the context of FPGA-based

implementation.

When mapping DSP dataflow graphs into FPGA implementations, it is important to consider real-time constraints as well as optimization of hardware resources. *Synchronous dataflow* (*SDF*) [1] has been used widely as an efficient model of computation for analyzing performance and resource requirements of DSP applications that are implemented on various target architectures (e.g., see [2, 3, 4, 5, 6]). In recent years, the *parameterized dataflow* meta modeling approach has evolved as a useful framework for modeling graphs in which arbitrary actor, edge, and graph parameters can be changed dynamically. However, the potential to enable efficient hardware synthesis has been treated relatively sparsely in the literature for traditional dataflow modeling techniques, and even more so for the newer, more general parameterized dataflow model. In this thesis, we develop efficient techniques to synthesize SDF-based and parameterized-dataflow-based dataflow models onto FPGA platforms.

1.1 Contributions of this thesis

1.1.1 Systematic generation of FPGA-based 1D-FFT implementation

We propose a systematic approach for synthesizing FPGA implementations of FFT computations. Our approach considers both cost (in terms of FPGA resource requirements), and performance (in terms of throughput), and optimizes for both of these dimensions based on user-specified requirements. Our approach involves two orthogonal techniques — FFT inner loop unrolling and outer loop unrolling — to perform design space exploration in terms of cost and performance. By appropriately combining these

two forms of unrolling, we can achieve cost-optimized FFT implementations in terms of FPGA slices or block RAMs in an FPGA subject to constraints on the required throughput.

We compared the results of our synthesis approach with a recently-introduced commercial FPGA intellectual property (IP) core — the FFT IP module in the Xilinx LogiCore Library, which provides different FFT implementations that are optimized for a limited set of performance levels. Our results demonstrate efficiency levels that are in some cases better than these commercial IP blocks. At the same time, our approach provides the advantages of being able to optimize implementations based on arbitrary, user-specified performance levels, and of being based on general formulations of FFT loop unrolling trade-offs, which can be retargeted to different kinds of FPGA devices.

1.1.2 Resource-efficient acceleration of 2D-FFT on FPGAs

The 2-dimensional (2D) FFT is a fundamental, computationally intensive function that is of broad relevance to multidimensional signal processing computations, such as those found in smart camera systems, medical imaging tools, and other important applications. In this thesis, we develop a systematic method for improving the throughput of 2D-FFT implementations on FPGAs. Our method is based on a novel loop unrolling technique for FFT implementation, which is extended from our work on FPGA architectures for 1D-FFT implementation described in Section 1.1.1.

Our unrolling technique deploys multiple processing units within a single 1D-FFT core to achieve efficient configurations of data parallelism while minimizing memory

space requirements, and FPGA slice consumption. Furthermore, using our techniques for parallel processing within individual 1D-FFT cores, the number of input/output (I/O) ports within a given 1D-FFT core is limited to one input port and one output port. In contrast, previous 2D-FFT design approaches require multiple I/O pairs with multiple FFT cores. This streamlining of 1D-FFT interfaces makes it possible to avoid complex interconnection networks and associated scheduling logic for connecting multiple I/O ports from 1D-FFT cores to the I/O channels of external memory devices. Hence, our proposed unrolling technique maximizes the ratio of the achieved throughput to the consumed FPGA resources under pre-defined constraints on I/O channel bandwidth.

To provide generality, our framework for 2D-FFT implementation can be efficiently parameterized in terms of key design parameters such as the transform size and I/O data word length.

1.1.3 Efficient static buffering for throughput-optimal FPGA Implementation of synchronous dataflow graphs

When designing DSP applications for implementation on FPGAs, it is often important to minimize consumption of limited FPGA resources while satisfying real-time performance constraints. We develop efficient techniques to determine dataflow graph buffer sizes that guarantee throughput-optimal execution when mapping synchronous dataflow (SDF) representations of DSP applications onto FPGAs. Our techniques are based on a novel modeling technique, which we call the *two-actor SDF graph model* (*TASM*). The TASM technique efficiently captures important characteristics relating to the behavior

and costs associated with SDF graph edges. With our proposed techniques, designers can automatically generate upper bounds on SDF graph buffer distributions that realize maximum achievable throughput performance for the corresponding applications. Furthermore, our proposed technique is characterized by low polynomial time complexity, which is useful for rapid prototyping in DSP system design.

1.1.4 Hardware synthesis techniques for parameterized dataflow

Parameterized SDF (PSDF) has evolved as a useful framework for modeling SDF graphs in which arbitrary parameters can be changed dynamically. However, the potential to enable efficient hardware synthesis has been treated relatively sparsely in the literature for SDF and even more so for the newer, more general PSDF model. This chapter investigates efficient FPGA-based design and implementation of the physical layer for 3GPP-Long Term Evolution (LTE), a next generation cellular standard. To capture the SDF behavior of the functional core of LTE along with higher level dynamics in the standard, we use a novel PSDF-based FPGA architecture framework. We implement our PSDF-based, LTE design framework using *National Instrument's LabVIEW FPGA*, a recently-introduced commercial platform for reconfigurable hardware implementation. We show that our framework can effectively model the dynamics of the LTE protocol, while also providing a synthesis framework for efficient FPGA implementation.

1.2 Outline of thesis

The rest of this thesis is organized as follows. Chapter 2, Chapter 3, Chapter 4, and Chapter 5 develop the contributions discussed in Section 1.1.1, Section 1.1.2, Section 1.1.3, and Section 1.1.4, respectively. In chapter 6, we present conclusions of the thesis, and discuss useful directions for future work that emerge from the contributions in this thesis.

Chapter 2

Systematic generation of FPGA-based 1D-FFT implementation

2.1 Introduction

The fast Fourier transform (FFT) is one of the most widely-used and important signal processing functions, for example, in applications related to digital communications and image processing. Since the computational complexity of the FFT is $O(NlogN)$, where N the number of inputs, the FFT potentially requires multi-cycle processing, and can become a major bottleneck for overall system performance. Thus, care must be taken in FFT module development at the actor level of the design methodology illustrated in Fig. 1.1.

To relieve this bottleneck, many commercial IP blocks provide a streaming form of the FFT with single–cycle–per–sample throughput. This high–throughput form of FFT comes at the expense of increased hardware cost, which in turn can lead to costly, over–designed hardware in situations where single–cycle–per–sample throughput is not required — that is, in situations where the FFT bottleneck is significant, but not so severe as to require such a high degree of throughput optimization. This chapter develops a systematic approach for generating a cost–efficient, FPGA–based FFT implementation based on a designer–specified throughput requirement. Our approach carefully integrates two orthogonal methods for trading–off hardware cost and performance. The first method, which can be viewed as outer loop unrolling of the targeted FFT, realizes parallelism by

instantiating multiple processing cores (dedicated hardware subsystems) across FFT butterfly stages. The second method, which can be viewed as unrolling of the FFT inner loop, allocates multiple cores within each stage. Each of these methods has advantages and drawback compared to the other, and in general, an integrated application of both methods can lead to a more cost–effective solution for a given throughput constraint — e.g., a more cost–effective solution compared to a solution that applies only one of these methods, or that is based on the high performance/high cost streaming FFT implementation. Depending on the given throughput constraint, one of these unrolling methods may be of more critical utility than the other. Furthermore, the proposed intergrated unrolling technique can be applied to the radix–4 FFT algorithms as well as the radix–2. It enables designers to choose the different processing unit showing different features in the performance/cost trade–off, based on their performance requirement.

Motivated by these observations, we develop a comprehensive approach to mixing and matching outer and inner loop unrolling for cost–efficient, throughput–constrained synthesis of FPGA hardware. In FPGA synthesis, slices (basic logic cells) and block RAMs (BRAMs) are limited, and usage in terms of these two resources is important in evaluating hardware cost [7]. Our synthesis approach is prototyped in National Instruments LabVIEW FPGA 8.5. LabVIEW is a graphical, dataflow–based programming environment for embedded systems design. LabVIEW features for HDL (hardware description language) synthesis and fixed point data types, along with LabVIEWs dataflow orientation make the tool well–suited to FPGA-based design of signal processing applications. The output of our techniques for synthesis and optimization of FFT configurations is a LabVIEW dataflow diagram that specifies the structure and functionality of an opti-

mized FFT configuration. This diagram is then synthesized to an FPGA device by first invoking LabVIEW's HDL synthesis tool, and then mapping the resulting HDL code using the platform-specific tools of the targeted FPGA. In our experiments, we have targeted the Xilinx Virtex II Pro FPGA.

In our experiments, we have compared the targeted cost metric — the usage of FPGA slices and BRAMs — between implementations generated by our novel synthesis flow, and those obtained from the Xilinx LogiCore library [8] for identical levels of throughput. The results demonstrate that our synthesis approach provides results that are of similar cost to those from the commercial library. This is encouraging since our approach provides the unique advantage of being synthesis-driven (as opposed to library-based) so that it can be driven by arbitrary performance levels rather than being restricted to a pre-determined subset of FFT configurations. Also, because it is based on an abstract synthesis formulation, it can be retargeted to different FPGA devices . e.g., by weighting or otherwise revising the cost function in terms of the resources that are most critical for a particular target.

2.2 BACKGROUND AND RELATED WORK

The discrete Fourier transform (DFT) for N points is given by

$$X_k = \sum_{i=0}^{N-1} x_i \cdot W_N^{ik} \tag{2.1}$$

where

$$W_N^{ik} = exp(-2\pi ik/N),\ k = 0, 1, \cdots, N-1 \tag{2.2}$$

The computational complexity of the DFT is $O(N^2)$. The radix–2 fast Fourier transform (FFT) algorithm proposed by Cooley and Tukey [9] in Figure 2.1, is widely used to compute the DFT with a complexity of $O(NlogN)$. After that, numerous FFT algorithm has been proposed to reduce the order of the algorithm complexity such as radix–2^m algorithms, Winograd algorithm (WFTA) [10], prime factor algorithms (FPA) [11], and fast Hartley transform (FHT) [12]. Because of the simple structure with a constant butterfly geometry, radix–2^m FFT algorithm is one of the most popular algorithm implemented in the hardware for the practical application.

FFT implementation in FPGA takes advantages of being reconfigured based on the user-specified design parameters — a variable FFT size, a variable data word length, a performance, and a hardware resource, compared to general purpose processors [13] [14] and dedicated FFT processor ICs [15] [16]. Various research efforts are involved in developing FPGA–based FFT library. Uzun [17] developed a framework covering different types of 1–D FFT algorithm. [18, 19, 20, 21] show the efficient FFT implementation on FPGA target meeting one throughput requirement. Xilinx LogiCore [22] provides radix–2/4 FFT library with a variable FFT size with two performance levels — burst mode and streaming mode. These works are under limited throughput level restricting the design space.

In achieving the various speed–up in the performance in radix-2/4 algorithm which requires to run a butterfly operation (dragonfly in radix–4) iteratively in Figure 2.1, it needs to execute multiple butterfly operations in parallel or a pipelined manner. Since a pair of inputs for the butterfly operation are changed in each FFT stage in Figure 2.1, a careful address scheme to read/write a data from/to storage spaces. Ma [23] developed

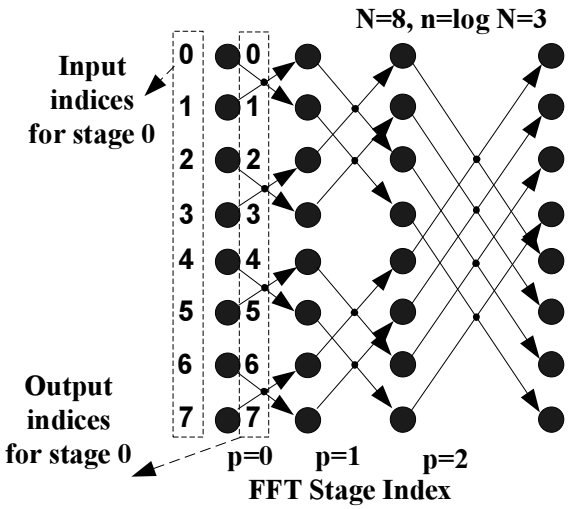


Figure 2.1: Signal flow graph of 8–point FFT with notational conventions illustrated. For each stage $p < n$, the data written through an output index for stage p corresponds to the data read through an input index for stage $(p+1)$.

an efficient method for the conflict–free memory management in FFT implementation. In Ma's approach an conflict–free strategy is employed to store butterfly outputs in the same memory locations that are used by the inputs to the butterfly. Such an conflict–free strategy is useful in reducing memory requirements, and enabling pipelining in terms of memory reads, butterfly operations, and memory writes. However, Ma's scheme is also developed for an FFT core that involves a single butterfly unit, so the overall approach is limited in terms of throughput improvement. Our proposed address scheme realizes the multiple butterfly operations in parallel and a pipelined manner by expanding Ma's work.

Nordin [24] presented a parameterized soft core generator for the FFT based on the Peace FFT algorithm with the stride permutation approach proposed by Takala et al. [25]. By running multiple butterflies simultaneously with a scalable stride permutation, the generated FFT achieves an effective balance between hardware costs and performance

features, and is also customizable based on given design constraints. A distinguishing aspect of the approach that we develop in this chapter is the realization of data parallelism with a carefully–configured address generator, and the integration of this address generation approach with an inner loop unrolling technique. This is in contrast, for example, to introducing special permutation structures for butterfly operations. Our approach, which is especially targeted to FPGA implementation, results in efficient utilization of FPGA slices.

A preliminary version of this work has been presented in [26], and this chapter goes beyond the developments of [26] by applying the proposed address scheme to radix–4 FFT algorithm as well as radix–2 FFT. For saving the storage space for a twiddle factor table, Hassan's method[27] has been applied in the proposed implementation.

2.3 UNROLLING TECHNIQUES

The radix–2/4 FFT algorithm involves running the butterfly/dragonfly operation iteratively. Using a conflict–free memory management scheme, we roll the butterfly operations within a given stage using a *for*–loop, which we refer to as the *inner loop*. Across different stages, we then employ another *for*–loop, which we call the *outer loop*. A *basic FFT core*(*BFC*) provides dedicated hardware for butterfly/dragonfly operation inside *for*–loop, and we can execute a *BFC* iteratively with the aforementioned inner and outer *for*–loops to achieve a complete FFT computation. However, rather than instantiating just one *BFC* for computing all FFT stages, we can achieve k times throughput improvement by running k *BFC*s simultaneously across stages, or by incorporating parallelism inside

the *BFC* so that multiple butterfly/dragonfly operations can be executed in parallel within a given stage. We propose two orthogonal unrolling techniques to allocate and utilize *BFC*s in an efficient and scalable manner on FPGAs. The techniques have different cost functions in terms of usage of FPGA slices or BRAMs, and we show that in general, the two approaches should be considered jointly for cost–efficient FPGA–based, FFT implementation.

2.3.1 Outer Loop Unrolling

The iteration count for the outer for–loop in the FFT is equal to the total number of stages, $logN$. Unrolling the outer loop by an *unrolling factor* $(k \geq 1)$ instantiates k *BFC*s. $(k-1)$ of these *BFC*s have $\lceil logN/k \rceil$ outer loop iterations each, while the remaining one has $(logN - \lceil logN/k \rceil)$ outer iterations. The designs of *BFC*s are identical to each other as described in 2.3.2 and 2.3.3 , except for some initialization details, and the iteration count. In this approach, k *BFC*s are running simultaneously, and up to a factor of k improvement in throughput can be achieved. This approach introduces k identical copies of *BFC*s, so that it is expected that a factor of k increase in hardware cost results — in terms of BRAMs and FPGA slices. The trade-offs associated with outer loop unrolling are complemented by inner loop unrolling, which we elaborate on in the following section.

2.3.2 Inner Loop Unrolling in radix–2 FFT

While unrolling the outer loop is realized by adding more copies of the *BFC*, the inner loop unrolling could be realized by executing multiple butterfly units in parallel inside

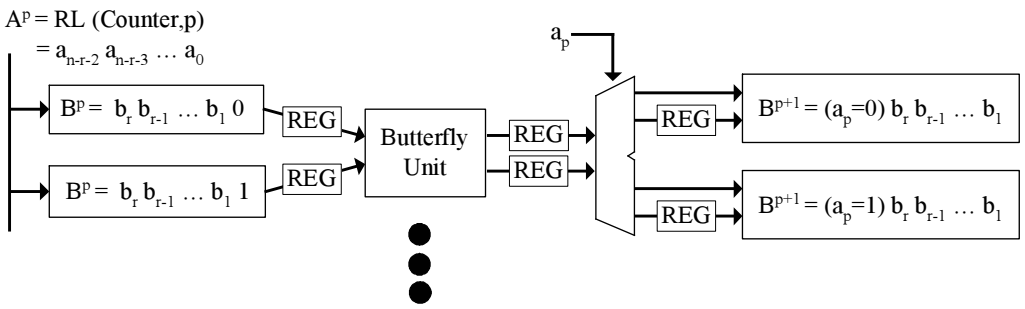


Figure 2.2: The pipelined radix–2 FFT implementation.

a *BFC*. That is, we parameterize the *BFC* with the number of hardware butterfly units, and we increase the value of the associated parameter to trade-off increased area for improved throughput. Ma [23] indicated that indices of two inputs, u and l, for the butterfly unit in the pth stage are identical, except for the pth bit in their binary patterns. Based on his observation, we propose a conflict–free memory addressing assignment for inputs/outputs of multiple butterfly operations in parallel. With k of an inner loop unrolling factor, k butterfly units within a *BFC* require $2k$ independent (parallel) data memory banks (*DM banks*); however, the amount of storage required in each *DM bank* is reduced by a factor of k so that the total amount of *DM bank* storage required after inner loop unrolling is unchanged, compared with *BFC* that has a single butterfly unit. Note that in an FPGA device, each *DM bank* will normally be implemented by one or more BRAMs [7].

2.3.2.1 Address for the read

For efficiency in a hardware utilization, we restrict the inner loop unrolling factor to be a power of 2 ; that is, $k = 2^r$ for some non-negative integer r. Each *DM bank* contains $2^{(n-r-1)}$ data locations that are accessed during FFT computation, where $n = log_2 N$, and

N is the number of sample points involved in the overall FFT computation. Given an inner loop unrolling factor $k = 2^r$, there are k hardware butterfly units in the parameterized *BFC*, and $2k$ *DM bank* (two for each butterfly unit). If x denotes a binary bit pattern, and y denotes a non-negative integer, let $RL(x,y)$ denote the bit pattern that results from left-rotation of x by y bit positions, and similarly, let RR(x,y) denote the bit pattern that results from right-rotation of x by y bit positions. Also, for bit patterns x_1 and x_2, let $CONCAT(x_1,x_2)$ denote the concatenation of x_1 and x_2. For example, if $x_1 = 110$, and $x_2 = 01100$, then $RL(x_1,2) = 011$, $RR(x_2,3) = 10001$, and $CONCAT(x_1,x_2) = 11001100$. Let *DM banks* have indices of $0,1,\ldots,(2k-1)$. Suppose that p is the index of a given FFT stage (i.e., $0 \leq p \leq n-1$); let $B^p = b_r b_r - 1 \ldots b_0$ be the binary pattern of the *DM bank* index in the pth stage; and let $A^p = a_{n-r-2}a_{n-r-3}\ldots a_0$ be the bit pattern for data address in the *DM bank* in this stage. For clarity, our conventions for input indices, and FFT stage indices, as well as N and n are illustrated in Figure 2.1. In the proposed memory addressing scheme, the input index in the pth stage that corresponds to address A^p in *DM bank* index B^p can be derived as

$$\begin{aligned} u &= RL(CONCAT(RR(A^p,p),B^p),p) \\ &= a_{n-r-2}a_{n-r-3}\ldots a_p b_r b_{r-1}\ldots b_0 a_{p-1}a_{p-2}\ldots a_0 \end{aligned} \tag{2.3}$$

With this notation, the least significant bit (LSB) in a given *DM bank* index b_0, represents the pth bit of the corresponding input index in the pth FFT stage. Since two input indices for a given butterfly operation in the pth stage are the same except for the pth bit, the input index u and the index l for the other input in the same butterfly operation stand

for their data stored in *DM bank* $b_r b_{r-1} \ldots b_1 0$ and $b_r b_{r-1} \ldots b_1 1$, respectively. These two *DM banks*, whose indices are identical except for their LSBs, become a pair of *DM bank* that is assigned to the same hardware butterfly unit in Figure 2.2. Thus, we entirely avoid any selection logic between among all *DM bank* in order to match one accessed input data to the other input data for the correct butterfly computation in each FFT stage. Moreover, all pairs of inputs to multiple butterfly operations can be accessed by the same address because the indices of corresponding input pairs are identical, except for the pth bit, and the pth bit is the one that the selects the *DM bank* for a given butterfly unit. One accessing address for all input pairs in every iteration helps to implement the input address generator in an efficient manner.

2.3.2.2 Address for the write

In the proposed FFT implementation, it does not introduce additional *DM bank* for storing butterfly outputs to realize the pipelined operation in read, butterfly computation, and write. After a butterfly operation in the pth stage, outputs should be written back to *DM bank* where butterfly input pairs come from. Butterfly output pairs stored in *DM bank* should be ready for the read in the $(p+1)$th stage with the same manner in 2.3.2.1. In other words, the destined *DM bank* index and addresses for writing back an output pair indexed by u and l in the pth stage are equivalent, respectively, to the *DM bank* indices and the address for reading the input indexed by u and l in the next stage, stage $(p+1)$. Thus, the destined *DM bank* index and its associated addresses for writing butterfly output pair can be generated by an inverse mapping from (2.3) with output indices of u and l,

and stage index $(p+1)$. This inverse mapping is given by

$$B^{p+1} = a_p b_r b_{r-1} \dots b_1 \tag{2.4}$$

$$A^{p+1} = a_{n-r-2} a_{n-r-3} \dots a_{p+1} b_0 a_{p-1} \dots a_0 \tag{2.5}$$

As reminding that pairs of inputs are accessed with the same address in pairs of *DM bank* whose indices are same but for b_0, the destined *DM bank* indices for each butterfly output pair should be the same as in (2.4). In other words, each butterfly unit produces an output pair which should be stored in one *DM bank* every cycle. Since 2 read operations and 2 write operations in each butterfly unit should be done simultaneously for the maximal pipelined operation and each BRAM in FPGA has only two ports, two ports of the bank must be used as the read and write port respectively. In the situation of a single port for the write, one of outputs is delayed by a inserted register to write a output pair with one allowed port as in Figure 2.2. Also, pairs of destined *DM bank* is connected to each butterfly units by a simple 1–by–2 demux selected by a_p other than the worst case of 1–by–$2k$ demux to store butterfly output pairs in the proper *DM bank* by a help of (2.4). For alternating the destined *DM bank* every cycle, the address, A^p,can be generated efficiently by

$$A^p = RL(Counter, p) \tag{2.6}$$

Here, the value of *Counter* is increased by one every clock cycle, so that bit a_p in A^p is flipped on each clock cycle. This provides a resource-efficient mechanism for generating a_p, and (via (2.4)), generating the required sequence of B^{p+1} selections.

2.3.2.3 Conflict–free property in read/write

To figure out whether the proposed addressing scheme works well, it requires to make sure that butterfly outputs should be written back to the address where data has been read to avoid the read–after–write hazard. From all *DM bank* pairs, pairs of data in $A_0^p(a_p = 0)$ are read out in even $Counter$ as in (2.6). In the next cycle (odd $Counter$), data of $A_0^p(a_p = 0)$ becomes inputs of butterfly units, while pairs of data in $A_0^p(a_p = 1)$ are read out in the pipelined FFT implementation in Figure 2.2. It means that there are a pair of free slots, $A_0^p(a_p = 0)$ and $A_0^p(a_p = 1)$, in all *DM bank* after the butterfly computation, and this is the same to a pair of output addresses in (2.5) whatever b_0 is. Therefore, our proposed method makes it possible to read and write a data simultaneously without additional *DM bank* for output pairs in a conflict–free manner. In this way, unrolling inner loop makes a *BFC* achieve k times throughput due to running k butterfly units simultaneously.

2.3.3 Inner Loop Unrolling in radix–4 FFT

The dragonfly unit in radix–4 FFT algorithm is equivalent to 4 butterfly units in radix–2 FFT algorithm, while it requires only 75% less multiplications and the same additions to 4 butterflies [28]. Hence, it takes advantages over radix–2 FFT in terms of less hardware resources. However, the size of FFT should be a power of 4 in radix-4 FFT so that the resolution of coverage in radix–4 FFT is worse, compared to a power of 2 in radix–2 case. For inputs for the dragonfly unit in the pth stage of radix–4 FFT, four input indices are identical in their binary pattern, except for $(2p)$th bit and $(2p+1)$th bit . Similar to the radix–2 FFT implementation, a group of four *DM bank* belongs to each

dragonfly unit, and the last two bits, b_1 and b_0 in a binary pattern of *DM bank* index, B^p, specify four different inputs for a dragonfly computation. As the base of the logarithm in radix–4 FFT is changed to 4, the total FFT stage is $log_4 N$ and input indices in the pth stage can be derived as

$$\begin{aligned} x_i &= RL(CONCAT(RR(A^p, 2p), B^p), 2p) \\ &= a_{n-r-3}a_{n-r-4}\ldots a_{2p}b_{r+1}b_r\ldots b_0 a_{2p-1}\ldots a_0 \end{aligned} \tag{2.7}$$

,where A^p, B^p, and r is the address, the *DM bank* index, and $log_2 k$ respectively when k is the inner loop unrolling factor.

2.3.3.1 Address for the read

The group of four *DM banks*, whose indices are the same except for the last two bits, belongs to the dragonfly unit in the radix–4 FFT implementation. From these *DM banks*, a group of four inputs for each dragonfly unit are read out from the address of A^p. In the pth FFT stage, (2.7) verifies that indices of the input group are identical except for the $(2p+1)$th and $(2p)$th bit which come from the last two bits of the *DM bank* group, and the correct dragonfly input group.

2.3.3.2 Address for the write

A destined *DM bank* index and its associated four addresses corresponding to four outputs from a dragonfly operation can be derived by an inverse mapping from (2.7) with a stage index of $(p+1)$.

$$
\begin{aligned}
B^{p+1} &= a_{2p+1}a_{2p}b_{r+1}b_r\ldots b_2 \qquad (2.8)\\
A_0^{p+1} &= a_{n-r-3}\ldots a_{2(p+1)}(b_1=0)(b_0=0)a_{2p-1}\ldots a_0\\
A_1^{p+1} &= a_{n-r-3}\ldots a_{2(p+1)}(b_1=0)(b_0=1)a_{2p-1}\ldots a_0\\
A_2^{p+1} &= a_{n-r-3}\ldots a_{2(p+1)}(b_1=1)(b_0=0)a_{2p-1}\ldots a_0\\
A_3^{p+1} &= a_{n-r-3}\ldots a_{2(p+1)}(b_1=1)(b_0=1)a_{2p-1}\ldots a_0
\end{aligned}
$$

Four dragonfly outputs are written back to one *DM bank*, as indices of a *DM bank* group are same except for the last two bits. Because only one port is allowed for the write due to BRAM port limit, it requires a parallel-in/serial-out shift register to write an output group through the allowed port in Figure 2.3. From (2.8), a selector of 1–by–4 demux, providing the dragonfly output group to the destined *DM bank*, is determined as a_{2p+1} and a_{2p}, two bits out of the address A^p. As it takes four cycles to pass loaded outputs to the *DM bank* port, a change in the selector of demux should be repeated in a period of four cycles. To meet this requirement, the address A^p is generated as

$$A^p = RL(Counter, 2p) \qquad (2.9)$$

, where $Counter$ is increased by one every clock cycle. Similar to radix–2 FFT implementation, a group of *DM bank* belonging to the dragonfly unit is pre-determined by (2.8), and it makes the design of *DM bank* selection logic be simple.

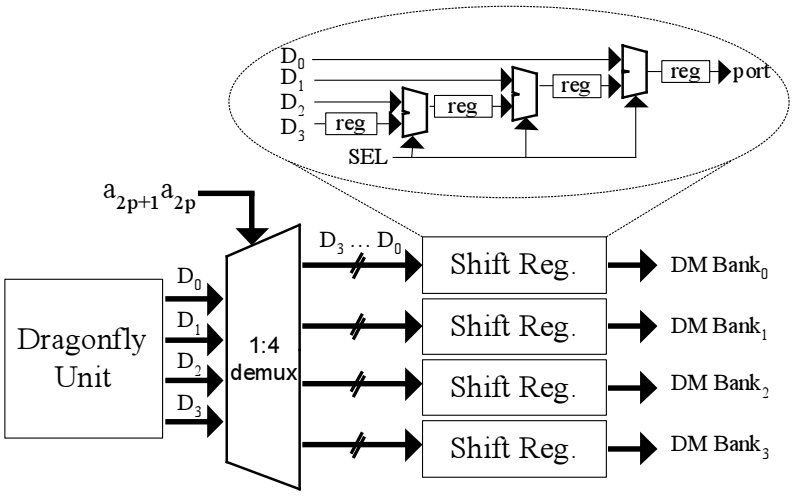


Figure 2.3: *DM bank* selection logic and parallel-in/serial-out shift register.

2.3.3.3 Conflict–free property in read/write

After the dragonfly operation, the output group is written back to memory banks where it has been read for the dragonfly operation. With the proposed address scheme, the read-after-write (*RAW*) hazard can be avoided in the pipelined radix–4 FFT implementation like the radix–2 case. In a period of four cycles, the demux provides four dragonfly outputs to each destined *DM bank* out of the group at each cycle. The output addresses (2.8) for all of four *DM banks* in the period are $A_0^p(a_{2p+1}=0,a_{2p}=0)$, $A_0^p(a_{2p+1}=0,a_{2p}=1)$, $A_0^p(a_{2p+1}=1,a_{2p}=0)$, and $A_0^p(a_{2p+1}=1,a_{2p}=1)$. As the demux passes out outputs first to the shift register of *DM bank*$_0$ in Figure 2.3 every period and writes outputs to the same four address in the identical order in all of four *DM banks*, the conflict–free property between the read and the write in *DM bank*$_0$ guarantees that there is no *RAW* hazard in the remaining *DM banks*. Input addresses have been generated by (2.9), while output address are generated based on input address and the last two bits of *DM banks* indices (2.8). Because of the shift register after the dragonfly unit in Figure 2.3,

Table 2.1: Time When address is accessed for read/write

Address	*Time for READ*	*Time for WRITE in* DM bank$_0$
$A_0^p(a_{2p+1}=0, a_{2p}=0)$	t_0	t_0+2
$A_0^p(a_{2p+1}=0, a_{2p}=1)$	t_0+1	t_0+3
$A_0^p(a_{2p+1}=1, a_{2p}=0)$	t_0+2	t_0+4
$A_0^p(a_{2p+1}=1, a_{2p}=1)$	t_0+3	t_0+5

reading the input from the address A^p in any *DM bank* always occurs before writing the output to A^p in *DM bank*$_0$. Table. 2.1 shows the time for the read and write operation in each address. With this proposed address scheme, multiple dragonfly operations inside *BFC* could be executed simultaneously without introducing additional *DM banks*.

2.4 COST/PERFORMANCE ANALYSIS

The two orthogonal unrolling techniques developed in the previous section exhibit different profiles of FPGA resource consumption. While outer loop unrolling pipelines multiple *BFC*s, inner loop unrolling executes multiple butterfly/dragonfly units in parallel inside *BFC*. Since the inner loop unrolling technique involves more localized control (i.e., control over a single *BFC*) it generally consumes less FPGA logic resources compared with the more extensive control structures needed for outer loop unrolling. However, inner loop unrolling is less flexible in terms of the set of possible unrolling factors — to preserve the applicability of our streamlined approach for inner loop memory management, while the inner loop unrolling factor must be a power of two. This requirement makes the range of achievable speedups for the inner loop unrolling technique to be limited to powers of two, while outer loop unrolling can be applied with arbitrary positive

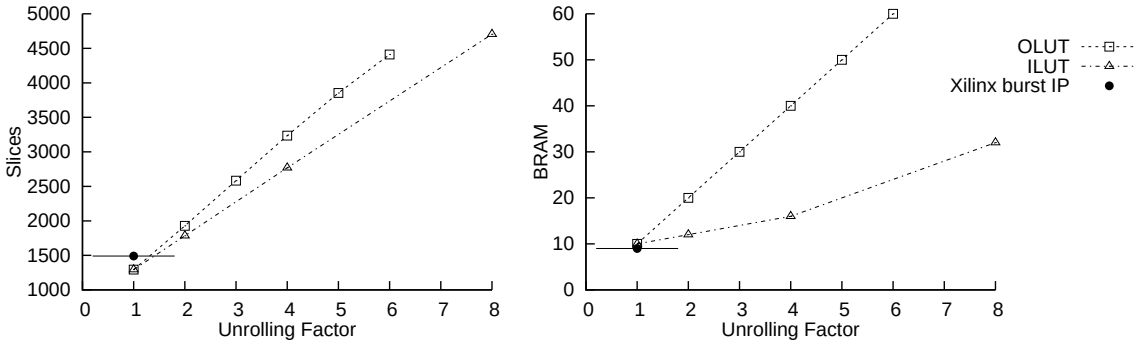


Figure 2.4: Resource utilization in Radix–2 FFT implementation with 4096 samples.

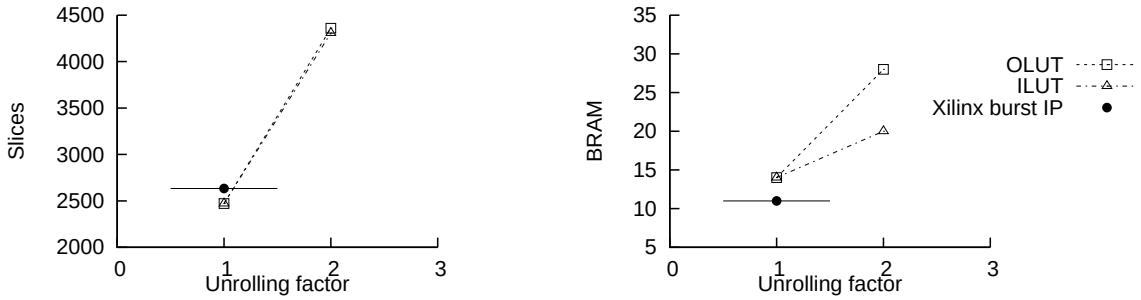


Figure 2.5: Resource utilization in Radix–4 FFT implementation with 4096 samples.

integer factors. Thus, for example, if the degree of speedup required to achieve the given throughput constraint is not a power of two, then a combination of inner–loop and outer–loop unrolling may lead to the most cost–effective solution.

Figure 2.4 shows FPGA slice and BRAM utilization as functions of the unrolling factor for both inner and outer loop unrolling. These results are obtained after synthesis, and include the streamlining effects of our proposed schemes for address generation and memory management. For both kinds of unrolling, BRAM and FPGA slice utilization increase linearly with the degree of speedup achieved (unrolling factor). Also from Figure 2.4 and 2.5, we see that inner loop unrolling is more area-efficient compared to outer loop unrolling for the same throughput increase. However, recall that inner loop unrolling

is restricted to factors that are powers of 2. In increasing FFT length, we take advantage of more fully using BRAMs in a wider range of inner loop unrolling factors. For use in analytical design space exploration, the following cost functions can be derived from these synthesis results:

$$u_{inner} = s_{inner} \cdot u_{initial}(k_{inner} - 1) + u_{initial} \tag{2.10}$$

$$u_{outer} = s_{outer} \cdot u_{initial}(k_{outer} - 1) + u_{initial} \tag{2.11}$$

Here, u_{inner} and u_{outer} are the amounts of utilization (FPGA slice or BRAM utilization) after inner and outer loop unrolling, respectively; $u_{initial}$ represents the amount of resource utilization without any unrolling; k_{inner} and k_{outer} are inner and outer loop unrolling factors, respectively; and $s_{inner}(s_{outer})$ is a constant factor that represents the slope of the linear plots for inner (outer) loop configurations in Figure 2.4 and Figure 2.5.

The cost functions (2.10) and (2.11) are for inner and outer loop unrolling in isolation. If both forms of unrolling are applied in combination, then the total hardware resource requirements can be expressed as

$$u_{combined} = s_{outer} \cdot u_{inner}(k_{outer} - 1) + u_{inner} \tag{2.12}$$

$$k_{combined} = k_{inner} \cdot k_{outer} \tag{2.13}$$

Given a throughput constraint, (2.12) and (2.13) can be used to efficiently search the space of feasible designs (i.e., designs with satisfactory throughput) for a cost-optimal solution. In particular, candidate pairs (k_{outer}, k_{inner}) that satisfy the throughput constraint

Table 2.2: Comparing synthesis report between radix–2 and radix–4 under the same performance level

Algorithm	*FPGA slices*	*BRAM*	*Multipliers*
Radix–2 FFT with (1,4)	2770	16	16
Radix–4 FFT without (1,1)	2471	14	12

(based on (2.13)) can be evaluated to select the one that minimizes cost (based on (2.12)). This evaluation can be pruned by noting that whenever a particular pair (k'_{outer}, k'_{inner}) is found to satisfy the throughput constraint, we need not consider any additional pairs $(k''_{outer}, k''_{inner})$ such that $k''_{inner} \geq k'_{inner}$ and $k''_{outer} \geq k'_{outer}$ are both satisfied. This approach allows for very rapid, pre–synthesis determination of cost–effective architectures for given throughput constraints.

2.5 EXPERIMENTAL RESULTS

We have targeted the Xilinx Virtex II Pro P30 embedded in the National Instruments PCI-5640R to synthesize implementations derived by our architecture generation techniques for the FFT. For a fair comparison with Xilinx library, HDL code for FFT generated from LogiCore is encapsulated by a wrapper from LabVIEW FPGA before the synthesis. The specific form of FFT implemented in these results is a radix–2/4 FFT with 4096 samples, with each sample represented as a fixed–point data type with 16-bit word length.

Table 2.2 shows to compare the synthesis report of the radix–4 FFT to the radix–2 FFT under the same performance level. For all kinds of FPGA resource, radix–4 FFT implementation shows a superior results to radix–2. It shows radix–4 FFT core is more

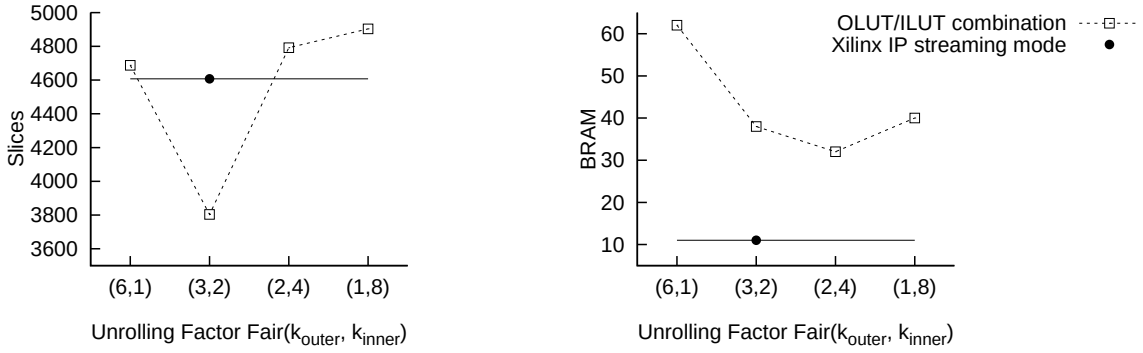


Figure 2.6: Resource utilization in the streaming Radix–2 FFT with 4096 samples

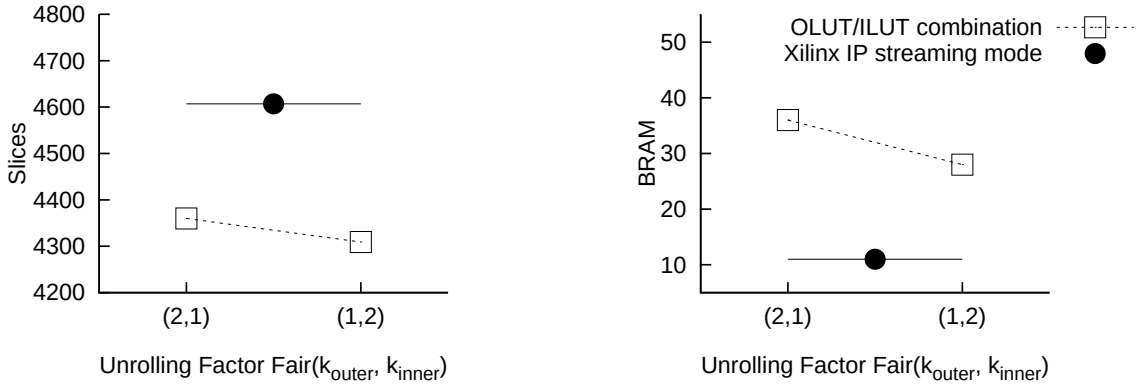


Figure 2.7: Resource utilization in the streaming Radix–4 FFT with 4096 samples

preferable IP in a power of 4 FFT size.

Figure 2.6 and Figure 2.7 indicates the synthesis report comparison between the proposed method and Xilinx IP core under the steaming performance level. In radix–2 FFT case, we take a target speedup of 6 here because the throughput of a sequential implementation (no unrolling) on this device is 6 cycles per sample, and 6 is the lowest integer speedup needed to achieve the common ''streaming FFT" target of 1 cycle per sample. Using the high level exploration approach developed in Section 2.5, and the device–specific slopes and initial utilizations from the curves in Figure 2.2, we can calculate analytically that when (k_{outer}, k_{inner}) is equal to (3,2) and (2,4), respectively, then

the generated FFT core is optimized in terms of FPGA slice usage and BRAM utilization. These results agree with the optimal values observed from the two curves from actual synthesis results in Figure 2.6, thereby demonstrating the accuracy of our high level exploration method. To compare our approach with relevant commercially–available FFT core, we evaluated the FFT core that is available from the Xilinx LogiCore library under the two different throughput levels that are available for it — streaming throughput and sequential (resource–optimized) throughput. For streaming FFT performance (one cycle per sample throughput), our approach required 25% less FPGA slices compared to the Xilinx core, but 190% more BRAMs. In radix–4 case, the target speedup for achieving the streaming performance is 2, because the throughput of the sequential FFT is 1.5 cycles per sample. The inner loop unrolling is always more efficient than outer loop unrolling and the best unrolling factor pair is (1,2). This implementation required 10% less FPGA slices but 150% more BRAMs, compared to the Xilinx IP.

For the sequential performance level(burst mode), our approach required 14% fewer slices, and 10% more BRAMs in radix–2 FFT implementation and 10% fewer slices 22% more BRAMs in radix–4 FFT, compared to Xilinx IP core as shown in Figure 2.4 and 2.5. Note that this comparison ("sequential performance") does not include any unrolling and is therefore essentially a comparison with Ma's FFT configuration, which is the special case of our approach that results when no unrolling is carried out. As well as the burst FFT implementation, our method provides various FFT implementation having a wider range of performance by help of two orthogonal unrolling technique. As the consumed resource in FFT implementation increases slower than the achieved speed-up, it can be the ready-to-use and cost-efficient FFT core meeting the user-specific target throughput. While the

heavy streaming FFT implementation is only one option in case of the target throughput larger than the sequential performance for a commercial IP, the proposed method provides well-tailored FFT library fit to the performance requirement.

Chapter 3

Resource-efficient Acceleration of 2D-FFT on FPGAs

3.1 Introduction

Fourier image analysis plays a key role in many image processing applications by making it possible to replace convolution operations in the spatial domain to simpler multiplication operations in the frequency domain, and enabling FFT convolution and various deconvolution techniques [29]. In spite of its wide use, FFT computation often becomes a major application bottleneck due to its high computational complexity. Thus, improving the throughput of 2D-FFT computation is useful to enhance overall system performance of the target application. Field-programmable gate arrays (FPGAs) are attractive for acceleration of FFT computations since FPGAs allow for configuration of customized digital logic structures that exploit the parallelism and regularity of FFT computations. However, achieving the full potential of 2D-FFT throughput acceleration under FPGA resource constraints is challenging since parallelism, interconnection complexity, FPGA logic gate utilization, and memory utilization must be carefully traded off at the actor level of the design methodology illustrated in Fig. 1.1.

The 2D-FFT is typically implemented as repeated invocations of 1D-FFT computations. Therefore, techniques for efficient FPGA-based 2D-FFT computations can be derived by considering two key design issues — improving the throughput of 1D-FFT computation with efficient FPGA resource consumption, and carefully utilizing the lim-

ited bandwidth of data transfer between the targeted FPGA device and external memory. Since 2D-FFT computation consists of $2N$ 1D-FFT computations, the throughput of 1D-FFT computation directly influences that of the enclosing 2D-FFT. In [26], we introduce an *inner loop unrolling technique*(ILUT) with an associated memory addressing scheme to achieve resource-efficient throughput improvement of the 1D-FFT. This technique can be parameterized by the required throughput to generate an FFT IP (intellectual property) subsystem such that the resource consumption is streamlined based on the targeted performance, which avoids over-designed hardware.

A 2D-FFT for an N-by-N image can be computed by performing N row-wise 1D-FFTs followed by N column-wise 1D-FFTs. Such an approach requires us to store N^2 intermediate data values between the row-wise and column-wise phases of computation. Due to the limited storage space within FPGA devices, external memory is often used to store such high-volume sets of intermediate data. When external memory is employed in this way, it is essential to carefully utilize the available bandwidth between the FPGA and associated external memory.

This chapter presents the efficient application to 2D-FFT implementation of our previously-developed ILUT technique [26], which is a systematic approach for generating 1D-FFT IP cores that are customized based on user-specified cost/performance trade-offs, as described above. We show that by carefully building on our ILUT-based 1D-FFT architecture to implement FPGA-based 2D-FFTs, we achieve significantly better cost/performance efficiency compared to previous techniques for implementation of 2D-FFTs on FPGAs. Here, by cost/performance efficiency we mean specifically the ratio of consumed FPGA resources to the achieved throughput.

In our ILUT-based approach to 2D-FFT implementation, only a single pair of I/O ports is needed, regardless of the inner loop unrolling factor, in the underlying 1D-FFT IP core to transfer data with external memory. This provides significant improvements in interconnect complexity and I/O scheduling overhead compared to related work on 2D-FFT implementation.

We prototyped our 2D-FFT implementation techniques in National Instruments LabVIEW (LV) FPGA 8.6 — a graphical, dataflow-based programming environment for embedded system design. LabVIEW includes a feature called Component-Level IP (CLIP), which allows designers to create wrappers around existing FPGA IP cores so that they can be used as components within LV FPGA. Designers can also write code for custom-designed subsystems in a hardware description language (HDL) and integrate this HDL code into LV FPGA using CLIP. In the experiments that we report on in this chapter, we have used CLIP to interface platform-specific IP for sending and receiving data between the targeted FPGA device and external memory.

For our experiments, we specified our optimized FFT architecture in the LV FPGA design environment, and implemented the architecture on the targeted FPGA by first invoking the LV FPGA HDL synthesis tool, and then mapping the resulting HDL code using the platform-specific tools of the targeted FPGA. The target FPGA that we used was the Xilinx Virtex-5. More specifically, our experimental platform was the National Instruments FlexRio board, which includes a Xilinx Virtex-5 device that is integrated with 128 MB of external memory (DRAM). Only the details in our implementation that pertain to synthesis and memory interfacing are related to the FlexRio board; the core FFT architecture that we present can be retargeted to other kinds FPGA platforms.

The organization of the chapter is as follows: In Section 3.2, we review background on the 1D and 2D-FFT algorithms, and describe challenges in implementing these computations efficiently. Subsequently, we present details of our ILUT-based, 2D-FFT architecture in Section 3.3. In Section 3.4, we show how our proposed 2D-FFT architecture provides significantly improved trade-offs between throughput and FPGA resource consumption. Section 3.5 demonstrates experimental results from our proposed architecture and comparisons with previous approaches. Section 3.6 provides a summary of the chapter and concluding remarks.

3.2 Background

The discrete Fourier transform (DFT) for N samples is defined as follows.

$$X_k = \sum_{i=0}^{N-1} x_i \cdot W_N^{ik}, \tag{3.1}$$

where

$$W_N^{ik} = exp(-2\pi ik/N) \ \ \forall k = 0, 1, \cdots, N-1.$$

As shown in Equation 3.1, a direct computation of the DFT suffers from $O(N^2)$ complexity. After Cooley and Turkey [9] proposed the FFT algorithm to decrease the computational complexity of the DFT to $O(N \cdot \log N)$, a large body of research has been focused on realizing the proposed 1D-FFT algorithm on various kinds of hardware platforms, including general purpose processors, programmable digital signal processors, and FPGAs. Ma [23] proposed an effective memory addressing scheme for a single FFT core to promote reuse of memory locations, and thereby reduce overall memory requirements.

Takala et al. [25] proposed a stride permutation for FFT computation, and Nordin et al. [24] developed a parameterized FFT soft core generator with a scalable stride permutation.

While many approaches have been developed to implement the 1D-FFT, research on design and implementation for 2D-FFT computations has centered around the approach of deploying multiple 1D-FFT cores, where each 1D-FFT core embeds a single processing unit — the butterfly unit for the radix-2 FFT or the dragonfly unit for the radix-4 FFT.

Jung et al. [30] developed a design methodology for exploring area/performance trade-offs in hardware implementation, and demonstrated this methodology using a 2D discrete cosine transform (DCT) benchmark. In this approach to DCT implementation, larger numbers of 1D-DCT blocks are deployed to achieve increasing levels of speed-up with corresponding increases in hardware resource consumption. The instantiated 1D-DCT blocks communicate with one another through a shared memory, which is implemented by an array of registers.

For a small input image, implementing the memory space for the image with an array of registers can be a reasonable design option. Such a design avoids limitations due to limited numbers of I/O channels and limited bandwidth between the FPGA device and external memory. However, since using arrays of registers is costly in terms of FPGA resources, the approach of Jung et al. can be expected to result in very large FPGA resource requirements for large input images. To avoid such dramatic increases in FPGA resource requirements, image storage is generally implemented in external memory, which has limited numbers of ports (typically dual ports) and limited bandwidth. However, external-memory-based implementation of image storage requires careful at-

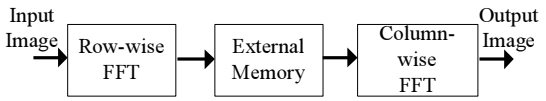


Figure 3.1: Functional block diagram of 2D-FFT computation.

tention to memory interfacing in the design of the FPGA architecture The approach developed in this chapter examines 2D-FFT acceleration from such a viewpoint of efficient integration of FPGA-based acceleration and external-memory-based image storage.

Uzun et al. [17] proposed a high level framework covering 1D and 2D-FFT implementations for real-time applications. In this framework, the parallelism in 2D-FFT computation is realized by allocating multiple 1D-FFT processors with a shared external memory. Since the input and output data vectors associated with each 1D-processor are transferred into a shared external memory, conflicts arise from multiple requests to read and write to the shared memory. Resolving these conflicts requires a relatively complex interconnection network, and also a complex control unit for scheduling data transfers between the 1D-FFT cores and the shared memory.

2D-FFT computation can be executed by a combination of N row-wise and N column-wise 1D-FFTs, as shown in Figure 3.1. Typically, 2D-FFT computation is performed on large images, which require external memory for their storage. Thus, the performance of the 2D-FFT is limited by the bandwidth of external memory, and the FFT computation must be designed carefully to achieve parallelism in conjunction with efficient communication with memory.

Previous work has emphasized accelerating 2D-FFT computation by employing multiple 1D-FFT cores. In our approach, we build on this general multiple-core approach,

and to make the approach more efficient, we incorporate our recently-developed methods to realize data parallelism within each of the instantiated 1D-FFT cores [26]. We do this by allocating multiple processing units to an individual 1D-FFT core, and incorporating a novel memory addressing scheme.

A distinguishing aspect of this approach is that our realization of data parallelism inside a single 1D-FFT core requires only a single pair of vector reading and writing requests to the external memory, regardless of the speed-up factor. Our architecture therefore prevents conflicts among requests from multiple cores in 2D-FFT implementation, and enables better utilization of memory bandwidth. Furthermore, by regularizing the access patterns to external memory, our approach reduces controller complexity and improves predictability.

3.3 2D-FFT Design

As described above, our approach for ILUT-based acceleration of the 1D-FFT, along with a formal development of the associated addressing scheme, are developed in [26]. In this section, we summarize important features of the ILUT-based approach that are relevant when applying it to 2D-FFT implementation, and we present details of the 2D-FFT architecture that we have developed by building on our ILUT-based 1D-FFT accelerator.

Henceforth, for conciseness, we refer to our ILUT approach simply as *ILUT* — that is, by ILUT, we mean our specific approach for FFT inner loop unrolling, as developed in [26], as opposed to the general concept of unrolling inner loops.

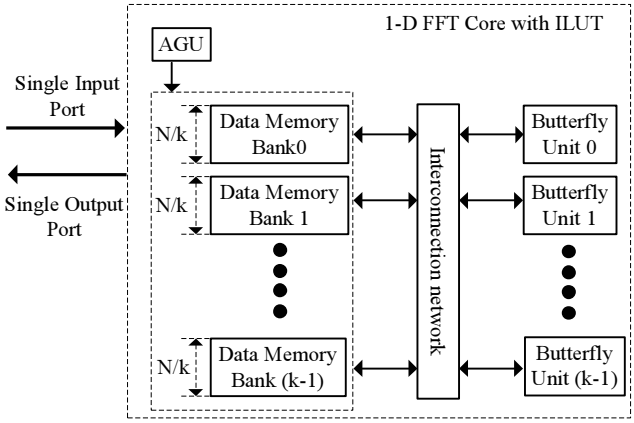


Figure 3.2: Functional block diagram of ILUT-based, 1D-FFT implementation.

3.3.1 Inner Loop Unrolling Technique (ILUT)

(1D) FFT computation involves $\log N$ FFT stages, where each FFT stage consists of $N/2$ butterfly computations. In ILUT, we refer to each FFT stage as an *inner loop* that "rolls" the butterflies. Also, we roll iterations across FFT stages through a conceptual *outer loop*. Intuitively, ILUT involves unrolling a given FFT stage by running multiple butterfly operations in parallel. Figure 3.2 shows an architectural block diagram of an FFT core after applying ILUT. We parameterize the core with a configurable number B of butterfly units, and increase the value of B to trade-off increased area for improved throughput. Addresses for input/output and for the butterfly units are controlled by the address generation unit (AGU). The AGU in our design allows conflict-free, simultaneous read and write accesses to the same dual-ported data memory bank. With this carefully-designed addressing scheme, the size of an individual data memory bank can be reduced by a factor of k when unrolling the inner loop by k (i.e., when $k = B$). Thus, since k data memory banks are required for an unrolling factor of k, the application of ILUT results in no net change in the overall data memory requirement, regardless of the unrolling factor.

In contrast to ILUT, the outer loop unrolling technique (OLUT) allocates multiple FFT cores to achieve parallelism in FFT implementation. OLUT-based approaches have been explored extensively in previous research efforts, such as [17, 30]. Figure 3.3 illustrates a functional block diagram of OLUT-based FFT implementation. For an unrolling factor of k, OLUT generally requires a factor of k in memory space increase compared to a single core implementation with no outer loop unrolling applied. Furthermore, OLUT introduces k identical copies of the underlying AGU, so it also involves an increase in the number of FPGA slices required.

3.3.2 2D-FFT Architecture

Figure 3.4 shows a functional block diagram of our proposed 2D-FFT architecture, which we refer to as the *IBTF* (ILUT-Based Two-dimensional FFT) architecture. The IBTF deploys a single 1D-FFT core with ILUT applied within the 1D core to achieve the desired level of parallelism. The 1D-FFT core employed has a single input port and a single output port, regardless of the degree of inner loop unrolling applied to the 1D core. Each of these ports is connected to a dual-port memory, which we call the *local memory* (*LM*). The LM is used to buffer data between the external memory and the ILUT-based 1D-FFT core. More specifically, the LM is used for sending and receiving vectors of FFT outputs and inputs, respectively, through an external memory interface that operates concurrently with the transform computation within the 1D-FFT core. The LM is divided into two separate regions — the LM_R provides a buffer for reading from external memory, and similarly, the LM_W provides a buffer for writing to external memory. Both the LM_R

and LM_W have the same size S_{buffer} (in bytes).

The LM is implemented on the targeted FPGA device. In general, it can be implemented in FPGA block ram (BRAM) or in FPGA slices (distributed memory). For small to moderate LM sizes, BRAM implementation has the disadvantage that the BRAMs used for the LM are largely underutilized. In our experiments, we have used distributed memory to implement the LM. Such an approach frees up the BRAMs to support other applications or subsystems that co-exist with the IBTF core on the same FPGA device.

The *control unit* (*CU*) handles the scheduling of all requests for transferring data between the LM buffers and the external memory. Since external memory is volatile, the CU must also take steps to ensure that the data stored in the external memory remains valid throughout its required lifetime. Furthermore, to increase the efficiency of data transfers, the CU accesses external memory through groups of sequential addresses, which are further clustered together in terms of common types of accesses (read or write). This kind of clustered, sequential access pattern is more efficient than more irregular types of patterns (e.g., see [17]). For every iteration of the underlying 1D-FFT transformation, the CU issues S_{buffer} read requests followed by S_{buffer} write requests.

In contrast to ILUT, OLUT-based approaches require k pairs of I/O ports in the external memory interface, along with k 1D-FFT cores. Furthermore, the external memory interface in the OLUT approach requires a complex interconnection network, including a crossbar switch, to connect the k pairs of I/O ports, and provide the required external memory access from the set of parallel FFT cores. Furthermore, since the *CU* must control multiple memory requests from multiple pairs of I/O ports, it needs to incorporate complex scheduling logic to manage contention among multiple requests. Hence,

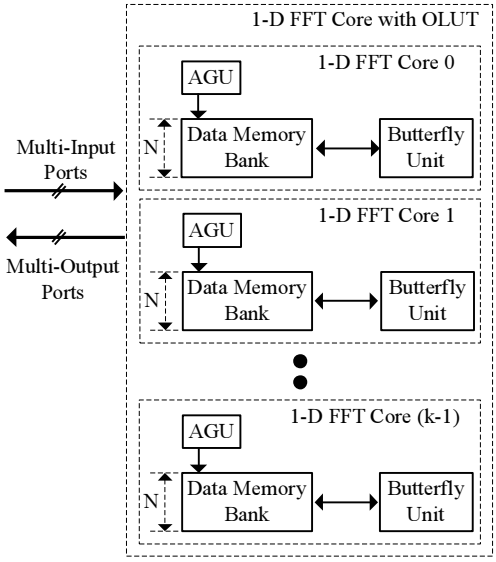


Figure 3.3: Functional block diagram of 1D-FFT with OLUT

OLUT-based implementations of the 2D-FFT can be expected to consume more FPGA slices compared to ILUT-based implementations under the same unrolling factor. We will provide a more in-depth comparison on these points in Section 3.4.

3.4 Analysis and Comparison ILUT-based and OLUT-based Implementation

As described previously, when external memory is involved, the achievable speed-up for a 2D-FFT implementation depends heavily on the bandwidth available for external memory accesses. The on-board external memory on the NI-FlexRio platform provides a bandwidth of 320 MB/s under a 40MHz base clock. Since the default size of data in the interface to the memory is 64 bits, the bandwidth can be viewed as a single sample per a cycle.

In both OLUT- and ILUT-based approaches, the CU needs to provide N samples

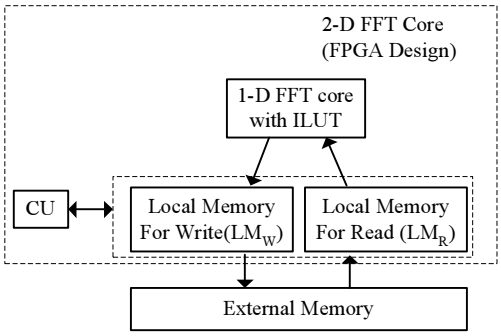


Figure 3.4: Functional block diagram of 2D-FFT with ILUT.

in LM_R for the next 1D-FFT computation, and write out N samples from LM_W to external memory. Here, N represents the input image size — i.e., the input image contains NxN pixels. During the k-th 1D-FFT computation, the CU must transfer N inputs for the $(k+1)$-th 1D-FFT computation in LM_R from the external memory. In the same computation frame, the CU also needs to transfer N outputs (produced by the $(k-1)$th 1D-FFT computation) from LM_W to the external memory.

In other words, $2N$ cycles of data communication are required between the local memory (LM) and the external memory for each 1D-FFT computation, and this is a limiting factor in the achievable throughput.

3.4.1 Operation of ILUT-based 2D-FFT Implementation

A timing diagram for an iteration of ILUT-based 2D-FFT computation is shown in Figure 3.5. The data loading and unloading processes can be overlapped in the proposed 1D-FFT IP, and with such overlapping, N clock cycles are required to process N samples. FFT computation follows the loading/unloading process, and for this computation, $N/2 \cdot \log N$ cycles are required if no unrolling is applied in the underlying radix-2 1D-FFT. If

we apply ILUT with unrolling factor k, then k butterfly units are deployed inside the 1D-FFT core so that the execution time for each 1D-FFT computation can be decreased by a factor of k. Therefore, the total time, as a function of the unrolling factor, for 1D-FFT computation is

$$T_{\text{inner}}(k) = N + \frac{N/2 \cdot \log N}{k}. \tag{3.2}$$

Now recall that it requires $2N$ cycles of data communication to prepare the next 1D-FFT computation after the previous 1D-FFT computation has completed. Thus, an upper bound on the achieved speed-up can be expressed as $S_{\text{inner}}^{MAX} = \log N/2$. Up to this level of speedup, ILUT exhibits speed-up that is linear to the unrolling factor k. The achieved speedup saturates, however, at S_{inner}^{MAX} due to bandwidth limitations in the target platform.

Note that this analysis is based on our use of $2N$ cycles as a bound for the required LM-external-memory data transfer between FFT computations. This *transfer rate* bound is applicable, for example, in the NI FlexRio target platform that we have targeted in our experiments. This bound is also applicable in the OLUT- and external-memory-based 2D-FFT implementations explored in [17]. Changes in this bound, however, require corresponding changes to the speedup analysis presented in this section.

ILUT-based implementation promotes efficient utilization of FPGA resources. To see this, recall that LM_R (LM_W) connects the input (output) port of the underlying 1D-FFT core to the output (input) channel of the external memory. Because of the regular access patters to and from LM, LM_R and LM_W can be implemented by FIFO buffers that operate based on standard (push and pop) FIFO access operations, and simple inter-

facing logic. More specifically, data transfers involving the *LM* can be controlled by a simple rule — data is pushed or popped as needed whenever the FIFO status is neither "full" nor "empty." This simplicity is facilitated by the form of data parallelism provided by the ILUT architecture, which is implemented entirely in the 1D-FFT core, and does not require parallel or random-access interfaces to LM. Exploiting this feature allows for resource-efficient implementation of LM and its associated interfaces in distributed memory or BRAM, and allows also for simple, resource-efficient implementation of the CU.

3.4.2 Operation of OLUT-based 2D-FFT Implementation

In OLUT-based FFT implementation, $k \geq 1$ FFT cores operate simultaneously, and each of these cores contains a single butterfly unit. Therefore, OLUT enables a reduction in 1D-FFT processing time by a factor of k. To run k 1D-FFT cores in parallel, the associated memory access controller must periodically fill up input data and clear out output data local memory at a sufficient rate. Since $2N$ cycles are needed for the data transfers associated with each 1D-FFT core, the controller can set up a single 1D-FFT computation frame for each of k 1D-FFT cores every $k \cdot 2N$ cycles.

Thus, if T_{base} represents the time for 1D-FFT computation without acceleration ($k = 1$), then we can write

$$\begin{aligned} T_{\text{outer}}(k) &= \frac{max(T_{\text{base}}, k \cdot 2N)}{k} \\ &= \max(\frac{N + N/2 \cdot \log N}{k}, 2N). \end{aligned} \tag{3.3}$$

As with the ILUT-based architecture, the throughput improvement with OLUT is limited by the bandwidth between the FPGA device and external memory.

Note also that the minimum inner loop unrolling factor k_{inner} (for ILUT) that is required to reach a given level of throughput is generally larger than the minimum outer loop unrolling factor k_{outer} required to achieve the same level of performance. This is because the total size of the required data memory space (the storage space represented by the blocks labeled as "Data Memory" banks in Figure 3.3) for OLUT is k_{outer} times larger than the data memory space required by ILUT, and hence, the net time required for loading and unloading local memory is reduced by a factor of k_{outer} by the ILUT approach compared to OLUT. Note that ILUT requires constant data memory size (independent of the inner loop unrolling factor), and therefore, the net time required by ILUT for loading and unloading local memory is also constant.

Overall, even though the larger unrolling factors required by ILUT (for given levels of performance) result in correspondingly higher factors of FPGA resource usage increase due to parallel resource instantiation, this increase is more than compensated by the improvement in the storage requirements of the data memory banks (especially for larger unrolling factors). Thus, when FPGA distributed memory is used to implement local memory, ILUT exhibits a significantly better ratio of achieved throughput to consumed resources (FPGA slices) compared to the OLUT approach. This is demonstrated quantitatively in section 3.5 through our experiments.

Furthermore, OLUT requires a relatively complex interconnection network to switch paths from multiple I/O ports of the 1D-FFT cores to the local memory subsystem. To maintain peak performance, this interconnection network must be capable of supplying

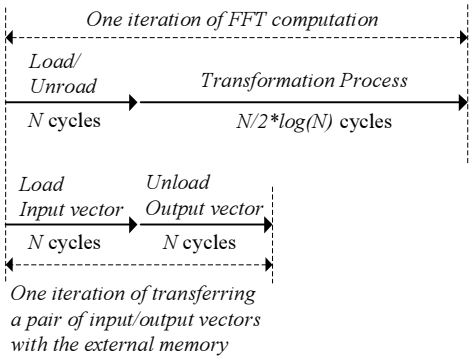


Figure 3.5: A timing diagram of ILUT-based FFT computation.

an input vector before each transform computation and receiving an output vector after each computation. Due to the reduced regularity of the memory accesses across the OLUT interconnection network, the local memory cannot be managed in the form of a simple FIFO, as can be done with our ILUT-based architecture. In OLUT, the local memory controller must keep track of the associated row/column vector set for each 1D-FFT transform computation, and must continuously perform book-keeping to switch the interconnection paths. Also, the OLUT controller must perform inter-core synchronization across the set of 1D-FFT cores. As we demonstrate in the next section, the increased control complexity in OLUT results in significant FPGA resource consumption increase compared to ILUT.

3.5 Experimental Results and Discussions

In our experiments, 2D-FFT designs have been implemented and evaluated for two different sizes of images — 256x256 and 2048x2048. Since a 2048x2048 image requires approximately 33MB of memory, and our targeted platform has 128MB of exter-

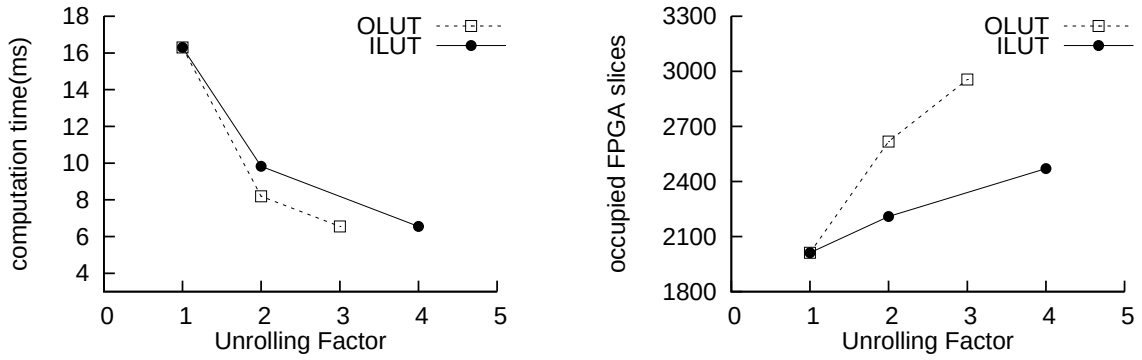


Figure 3.6: Computation time and FPGA resource utilization for 2D-FFT with an image size of 256x256.

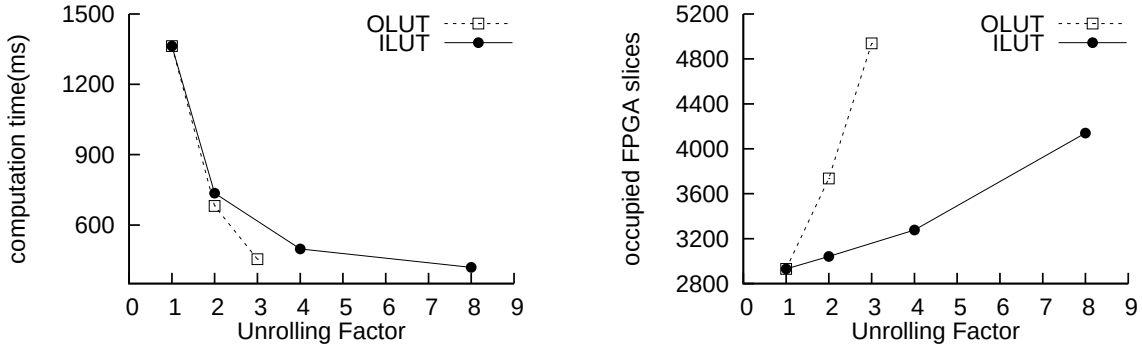


Figure 3.7: Computation time and FPGA resource utilization for 2D-FFT with an image size of 2048x2048.

nal memory, 2048x2048 is the largest standard image size (i.e., the number of rows and columns is a power of two) that can be supported on our platform — the next largest image size, 4096x4096 requires approximately 4×33MB, which slightly exceeds the available 128MB.

We have implemented both inner loop (ILUT) and outer loop (OLUT) unrolling separately in alternative 2D-FFT implementations, and we have carefully compared the results. Each unrolling technique has been applied with increasing unrolling factors until the maximal throughput allowed by the external memory bandwidth was achieved. While the given FPGA target allows us to transfer data between the external memory and FPGA device at a clock frequency of 100MHz, our 2D-FFT implementation cannot operate on such a fast clock. Thus, multiple clock domains are required to support the highest possible memory bandwidth. In this chapter, we focus on exploring 2D-FFT design trade-offs for conventional, single-clock-domain implementation, and therefore, we slow the memory interface down to the same speed as the 2D-FFT computation subsystem. More specifically, we use a single clock domain that operates at 40MHz. Applying heterogeneous clock domains to explore further performance enhancement is a useful direction for further investigation.

In our OLUT implementation, we employed the LabVIEW FPGA 1D-FFT library module, which is a widely-used commercial 1D-FFT library module that has competitive performance compared to related commercial FPGA cores [26]. In both the OLUT and ILUT implementations, we employed distributed memory to implement the local memory subsystems, as described earlier in Section 3.3. For this purpose, we used the distributed memory library from Xilinx LogiCore [22]. Another useful direction for follow-on re-

search is the integration of block RAM (BRAM) into the design space for optimized ILUT-based 2D-FFT implementation.

For our experiments with ILUT, we have restricted the inner loop unrolling factor to be a power of 2 for efficiency in hardware utilization. When ILUT is applied with unrolling factors that are not powers of two, significant resource usage inefficiency results. This is because the 1D-FFT data memory indices cannot be generated simply by concatenating the binary bit patterns of the memory addresses to that of the associated memory bank addresses, and thus significant overhead results in the address generation logic. While multiple butterfly units jointly compute a single input vector and are controlled by a novel memory address scheme in ILUT, each butterfly unit handles its own individual input vectors separately in OLUT. In this sense, the unrolling factor in OLUT can be a natural number rather than a power of two as in the ILUT case. We compare the proposed ILUT to OLUT under the performance levels that ILUT provides. This comparison may not be a comprehensive comparison between two techniques, but it clearly demonstrates the advantages of ILUT in terms of resource utilization across all of its allowed performance levels.

Given a 2D-FFT implementation based on the assumptions described above (a single clock domain and distributed-memory-based local memory), we define the *relative resource utilization* as the quotient R/T, where R denotes the total number of FPGA slices (including resources for computation and for distributed memory) required for the implementation, and T denotes the throughput in 2D-FFT computations per second. Thus, decreasing levels of relative resource utilization indicate increasing levels of cost-efficiency relative to the achieved processing performance (or conversely, increasing lev-

els of performance-efficiency relative to the achieved cost).

Figure 3.6 shows the computation time and the number of occupied FPGA slices for 2D-FFT implementation under both ILUT- and OLUT-based approaches with an image size of 256x256. Corresponding values of relative resource utilization are given in Table 3.1. From Figure 3.6, we see that OLUT exhibits a smaller computation time compared to ILUT under for an unrolling factor of 2 ($k = 2$). This is due to the reduced time for local memory loading and unloading, which we discussed in Section 3.4. Even though there is a difference in throughput for $k = 2$, both ILUT and OLUT techniques exhibit similar levels of relative resource utilization in Table 3.1. OLUT achieves the maximal achievable throughput (as constrained by the external memory bandwidth) at an unrolling factor of 3, while ILUT achieves the maximal achievable throughput at an unrolling factor of 4. At this maximal performance level, ILUT exhibits 20% less relative resource utilization compared to OLUT, as shown in Table 3.1. This demonstrates the significant resource-efficiency advantage offered by our ILUT-based approach compared to the more conventional approach of OLUT-based 2D-FFT implementation.

Computation time and FPGA slice usage results for an image size of 2048x2048 are shown in Figure 3.7. While OLUT has a smaller execution time than ILUT with a small unrolling factor, ILUT consistently exhibits better relative resource utilization than OLUT under similar levels of performance. For example, even though ILUT at $k = 4$ and OLUT at $k = 3$ employ different unrolling factors, both of these configurations exhibit similar levels of performance, as shown in Fig 3.7, and these configurations can be compared in terms of the relative resource utilization metric. This is shown in Table 3.2.

Furthermore, ILUT consumes a smaller number of FPGA slices at the highest per-

formance level. The lowest unrolling factor at which OLUT achieves maximal throughput is $k_{\text{outer}} = 4$. However, OLUT cannot be synthesized on our target platform at this unrolling factor. This is because, as indicated by the results from our synthesis attempts, the number of FPGA slices required at this unrolling factor exceeds the number of available slices in the FPGA device. In Table 3.2, we make a note of the "compile error" in the OLUT value at $k = 4$ to describe that we could not synthesize this case due to limited FPGA resources on the target platform. Even though we cannot synthesize this case, we can estimate its relative resource utilization. Since this case reaches the maximal achievable throughput, it will have the same throughput as the ILUT case with $k = 8$, as shown in Fig 3.7. Also, this OLUT configuration ($k = 4$) is expected to consume more FPGA resources than the OLUT configuration with $k = 3$ due to increases in the butterfly unit and its associated control logic. In Fig 3.7, the ILUT configuration with $k = 8$ shows much better resource utilization than OLUT with $k = 3$. Hence, we can expect that the ILUT approach has smaller relative resource utilization compared to OLUT when we compare their respective maximal-performance configurations.

Another interesting result from our experiments is that the relative resource utilization of ILUT at $k_{\text{inner}} = 4$ is smaller than that at $k_{\text{inner}} = 8$. This is because the potential speed-up at $k_{inner} = 8$ is not fully realized due to the limited external memory bandwidth. This saturation of performance can be seen in Figure 3.7.

Table 3.1: Relative resource requirements for an image size of 256x256.

Unrolling Factor	*ILUT*	*OLUT*
$k = 1$	32.96	32.96
$k = 2$	21.71	21.43
$k = 3$	N/A	19.36
$k = 4$	16.18	N/A

Table 3.2: Relative resource requirements for an image size of 2048x2048.

Unrolling Factor	*ILUT*	*OLUT*
$k = 1$	3994	3994
$k = 2$	2391	2546
$k = 3$	N/A	2244
$k = 4$	1632	Compile Error
$k = 8$	1736	N/A

3.6 Conclusion

In this chapter, we have developed a systematic approach for generating dedicated 2D-FFT subsystems for FPGA implementation. Our approach realizes data parallelism within an individual 1D-FFT core, and minimizes the interface complexity between the underlying 1D-FFT core and local memory. Our approach allows for scalable, parallel 2D-FFT implementation with a relatively simple interconnection network, and correspondingly simple control logic. These features contribute to improved FPGA resource consumption at a given level of performance compared to previous 2D-FFT FPGA architectures.

Our methods are demonstrated through extensive synthesis experiments using the Xilinx Virtex-5 FPGA device. Our synthesis results quantify the cost-performance trade-

offs provided by our proposed class of FFT architectures. A distinguishing characteristic of our approach, compared to previous techniques for 2D-FFT implementation, is that we provide a systematic method to generate streamlined, FPGA-based, 2D-FFT architectures while taking into account trade-offs between performance and cost.

Chapter 4

Efficient Static Buffering to Guarantee Throughput-Optimal FPGA Implementation of Synchronous Dataflow Graphs

4.1 Introduction and related work

At the graph level of the design methodology illustrated in Fig. 1.1, it is important to consider real-time constraints as well as optimization of hardware resources. When describing a DSP application with an graph, functional blocks and storage space for transferring data between adjacent blocks are modeled as graph vertices (*actors*) and edges, respectively. When mapping graph edges into storage locations, care must be taken to make effective use of limited storage locations (e.g., on-chip memory in programmable digital signal processors, and block RAM and distributed memory in FPGAs). However, reducing the storage space for transferring data between actors may result in decreased throughput due to idle time that is required to prevent buffer overflow — as buffers become smaller, the frequency and duration for such overflow-avoiding idle time generally increases, which leads to decreased throughput. The limited amounts of storage available in DSP implementation targets, and the importance of meeting real-time performance constraints motivate the goal of guaranteed, throughput-optimal buffer configuration for SDF graphs. In this chapter, we study this throughput and buffering analysis problem in the context of FPGA-based implementation.

Synchronous dataflow (SDF) [1] has been used widely as an efficient model of computation for analyzing performance and resource requirements of DSP applications that are implemented on various target architectures (e.g., see [2, 3, 4, 5, 6, 31]). Traditionally, throughput analysis for SDF graphs is performed by solving an instance of the maximum mean cycle problem (e.g., see [32, 33]) after converting the input SDF graph into an equivalent homogeneous SDF (HSDF) graph [1]. HSDF is a special case of SDF in which the production and consumption rates are identically equal to unity for all input and output ports of all actors. These rates are in terms of data values (*tokens*) per actor execution (*firing*). Throughput analysis based on SDF-to-HSDF conversion suffers from high worst case complexity because neither the time nor space required to perform this conversion is polynomially bounded (e.g., see [34]).

This complexity arises from the nature of *periodic schedules* of SDF graphs, which are used for static scheduling. A periodic schedule for an SDF graph is a schedule that produces no net change in the *buffer state* — i.e., the numbers of tokens that are queued on the buffers associated with the graph edges. The total number of actor firings in a periodic schedule can scale exponentially even for simple classes of SDF graphs [34]. Since each actor firing corresponds to a separate vertex in the HSDF version of an SDF graph, the SDF-to-HSDF transformation process can result in similar exponential growth.

Ghamarian et al. [35] have developed a method for SDF throughput analysis that avoids conversion to an HSDF graph, and uses state space exploration techniques — in terms of the buffer state — instead. In general, executions of actors change the buffer state by removing (consuming) tokens from input edges of the actors that fire, and inserting (producing) tokens onto output edges. Ghamarian exploits the property that when SDF

graphs execute in a purely data driven ("self-timed") manner under bounded memory space, the state space is also bounded, and execution eventually settles into a periodic pattern (periodic steady state or *PSS*). In Ghamarian's method for throughput analysis, only selected states need to be stored when detecting the PSS of execution, and through Ghamarian's careful pruning technique for state storage, significant improvements can be achieved in the efficiency of performance analysis. However, the technique requires simulation of the overall schedule, and the worst case complexity is linear in the length (number of firings in) the given periodic schedule, which, as described above, is not polynomially bounded in the size of the input SDF graph.

Buffer minimization in SDF graph has been studied to mainly focus on single-processor target, see for example [36, 37, 38, 39, 40]. FPGA implementation, however, allows simultaneous actor firings by assigning each actor into its own dedicated FPGA slices. Thus, the minimal buffer solution given by the previous deadlock-free schedule can not be applied to this FPGA implementation domain.

Horstmannshoff et al. [41, 42] developed the scheduling method for complex RT level building blocks from SDF graph. Based on timing patterns of producing and consuming token in each block, it constructed the retiming graph to generate the schedule of generating the stall signal for each SDF actor with a minimum buffer cost.

Stuijk [43] develops a systematic approach for exploring throughput and storage trade-offs for SDF graphs. This approach applies methods developed in [44] for determining minimum storage requirements based on state-space analysis of buffer states. Stuijk's approach operates by first finding a minimal storage distribution, and then recursively increasing the storage space for each edge that has a storage dependency. This results in a

family of buffer distribution-throughput pairs as a representation of Pareto solutions for the graph. Although this approach prunes the search space to reduce complexity, schedule simulation is still required in the search process, so again, worst case complexity is not polynomially bounded.

Wiggers [45] presents an algorithm with linear computational complexity to determine close-to-minimum buffer capacities for a given throughput constraint. However, this approach imposes a form of strictly periodic scheduling that requires a counter in every functional block, which leads to resource overhead in FPGA and other hardware-oriented implementations. Also, since Wiggers's approach assumes that execution will enter the required periodic steady-state only with the timely availability of sufficient starting tokens for every actor, it may not adequately handle irregular streaming inputs, where token arrival times are less predictable.

In contrast to the related prior work, we propose a heuristic algorithm with low polynomial time complexity that provides upper bounds on buffer requirements to guarantee throughput-optimal FPGA realizations of SDF graphs. Our approach focuses on the restricted class of tree-structured SDF graphs — that is, the input application model (*application graph*) must be in the form of an SDF tree. We emphasize that our algorithm is a heuristic only in the sense of the buffer sizes that are computed; in terms of achieved throughput performance, our approach guarantees optimality.

We first analyze relationships of firing patterns between actors and buffer requirements for the *two-actor SDF graph model* (*TASM*), which is a specialized form of SDF graph that we propose for efficient analysis of data communication on individual edges in a given SDF application graph. We then apply this two-actor firing pattern analysis

repeatedly when traversing an application graph to determine buffer configurations that guarantee maximum achievable throughput.

In our buffer optimization scenario and our associated TASM analysis, we consider self-timed dataflow graph execution (e.g., see [46, 47]), which means that an actor is fired as soon as all of its input edges have enough tokens — that is, as soon as the number of tokens on each input edge e is at least $c(e_i)$. If each actor is mapped to a separate hardware resource, and the overhead of communication and synchronization between actors is negligible, then self-timed execution leads to the maximum achievable throughput (e.g., see [48, 47]). Moreover, this form of execution does not require any global schedule, and therefore storage, performance, and interconnect overhead associated with implementing a global schedule is avoided.

With predominantly coarse-grained dataflow actors (e.g., digital filters, and transform computations as opposed to adders and multiplers), and streamlined implementation of dataflow edges, one can reduce the relative overhead of inter-actor communication and synchronization significantly so that self-timed scheduling becomes an effective approach. This context of coarse-grain actors and streamlined edge implementation is the form in which we explore self-time implementation and associated buffer configuration strategies in this chapter.

We first present precise definition and notations related to buffer analysis of SDF-based implementations. Using these concepts, we analyze the data transfer behavior on an SDF edge by the TASM model described earlier. Based on this analysis, we develop an algorithm for buffer analysis based on the TASM model, and we show an overall design flow for applying this algorithm for efficient synthesis of FPGA implementations.

The proposed algorithm is implemented in the dataflow interchange format (*DIF*) package [49], which provides a standard language and associated toolset that is founded in dataflow semantics and tailored for DSP system design [5].

4.2 Background

4.2.1 Application representation

We represent a DSP application with a dataflow graph $G = (V, E)$, where each computational module is mapped to a vertex (*actor*) $v \in V$ and each directed edge $e \in E$ corresponds to a FIFO buffer for communicating data from the source actor $src(e)$ to the sink actor $snk(e)$ of e. We assume that the given dataflow model adheres to the assumptions of SDF, which require that the production and consumption rates of all actor output and input ports, respectively, are constant [1]. The SDF model is used widely in tools for DSP system design, and powerful analysis techniques have been develop for mapping SDF representations into various kinds of platforms (e.g., see [47]).

Given an SDF edge e, we represent the associated production rate of $src(e)$ by $p(e_i)$, and we represent the associated consumption rate of $snk(e)$ by $c(e_i)$. An SDF edge e also has associated with it a non-negative *delay*, denoted $del(e)$, which represents the number of *initial tokens* that reside on the corresponding buffer at the start of execution.

A necessary condition for executing (*firing*) an SDF actor v is that the number of tokens on every input edge e_{in} of v is greater than or equal to $c(e_{in})$. While v consumes $c(e_{in})$ tokens from each input edge e_{in} during its execution — i.e., during the execution of a single *invocation* or *firing* of the actor — it produces $p(e_{out})$ tokens onto each output

edge e_{out}.

If an SDF graph is properly constructed in a certain technical sense, then there exists a *periodic schedule* for the graph — that is, a schedule that is free from deadlock, fires each actor at least once, and produces no net change in the number of tokens on every edge e in the graph. This concept of a properly constructed SDF graph is referred to as *consistency*; efficient algorithms have been developed to determine whether or not an SDF graph is consistent, and therefore has a periodic schedule [1]. For a consistent SDF graph, the relative rates at which actors need to fire can be determined from the the *balance equations* [1]:

$$p(e_i) \times q[src(e)] = c(e_i) \times q[snk(e)] \text{ for all } e \in E. \tag{4.1}$$

For a consistent, connected SDF graph G there is a unique minimum integer solution the balance equations in G. This solution is called the *repetitions vector*, and is often denoted by q. *repetition count*. For each actor v in G, we refer to $q[v]$ as the *repetition count* of v. A valid and minimal periodic schedule should fire each actor a number of times equal to the repetitions count of the actor. Such a periodic schedule can then be iterated as many times as needed with guaranteed bounded memory for all of the edges in the graph [1].

If an SDF graph G is not connected, then repetitions vectors and periodic schedules can be computed separately for each connected component, and these "connected component schedules" can be iterated at arbitrary rates relative to one another to achieve bounded memory execution of G. The relative rates of execution for the connected com-

ponents can be managed — based on relevant characteristics of the associated signal processing subsystems — using a vector called the *blocking vector* [50]. The blocking vector has elements that are non-negative integers, and is indexed by the connected components in G.

In the remainder of this chapter, we assume that we are working with connected SDF graphs. However, through appropriate use of blocking vectors, the developments in the chapter can be extended naturally to handle SDF graphs that are not connected.

4.3 Target platform model

Since resource sharing is often avoided in FPGA implementation due to the relatively high cost of multiplexing and routing resources (e.g., see [51]), we assume that each computational block (SDF actor) is assigned to a dedicated set of FPGA logic cells without any sharing. Integrating resource sharing considerations into the developments of this chapter is an interesting direction for future work, and may be useful in cases where resources are limited compared to the amount of required computation.

FPGAs provide two ways of implementing memory space between functional blocks — such memory space can be implemented using *block RAMs*, which provide dedicated memory hardware within an FPGA, and *distributed RAM* using FPGA slices. The number of ports for reading (writing) data from (into) both forms of RAM is limited, and these limitations must be taken into account carefully for correct buffer management. In the Xilinx Virtex-II Pro FPGA, which we target in this chapter, the number of ports is limited to two, and therefore, only a single pair of simultaneous read/write operations to each

RAM subsystem is possible.

To support this limitation on RAM access, we incorporate in our dataflow-based architecture model, a self-loop on each actor, and we add a single unit of delay to each such self-loop. By a self-loop, we mean an edge whose source and sink actors are identical. By adding a self-loop with unit delay to each actor, we ensure that successive executions of the same actor are always serialized, which guarantees that the memory requests of one actor invocation do not conflict with those of another invocation of the same actor.

Our overall mapping approach therefore maps each actor in the SDF application model to a single actor (dedicated hardware resource) in the architecture model, along with a self-loop connection for that actor. Thus, we allow for concurrent execution of distinct actors, while serializing successive invocations of the same actor since such successive invocations must access the same memory ports for buffer access.

4.4 Design flow

Fig. 4.1 illustrates the overall design flow for our proposed buffer optimization technique under performance constraints. The proposed technique is implemented in the *dataflow interchange format* (*DIF*) package [5]. The DIF package provides a flexible dataflow design language, intermediate representations, and transformations for specifying, analyzing, and optimizing implementations of DSP applications. While DIF supports a variety of dataflow models of computation, including synchronous [1], cyclo-static [52], and enable-invoke [53] dataflow support for SDF in DIF is especially well-developed and mature. We leverage this support for SDF graph techniques in the DIF package to de-

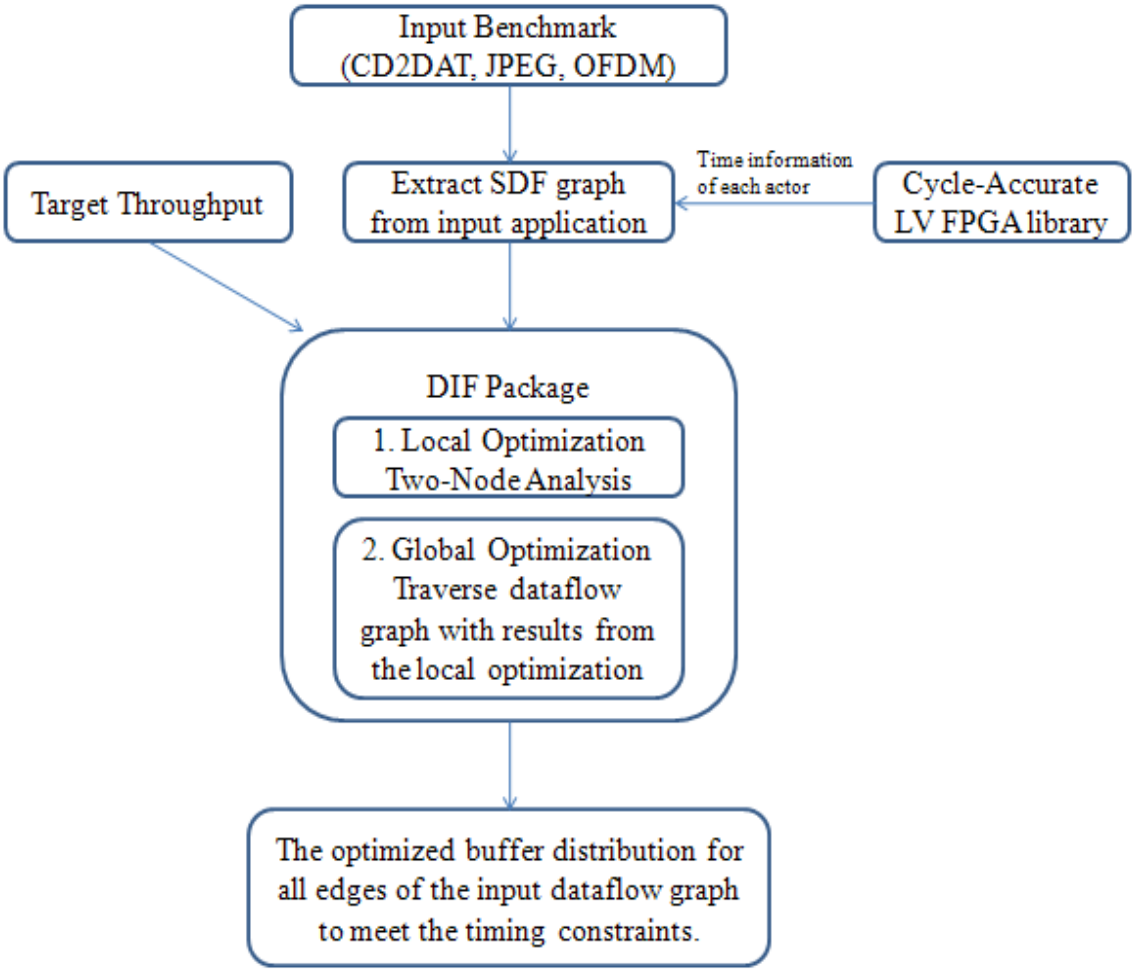


Figure 4.1: Overall design flow.

velop and experiment with the novel buffer optimization techniques that we introduce in this chapter.

The buffer optimization module that we have developed in the DIF package has two inputs — a DIF language specification of the the application to be implemented, and the throughput constraint — that is, the required rate at which the implementation must be able to process samples. As part of the DIF application specification, we include timing characteristics that are derived by profiling the execution of each actor on the target hardware. In the experiments that we report on in this chapter, the timing characteristics are derived from the LabVIEW FPGA library, which is targeted to Xilinx Virtex II-Pro FPGA platform.

The DIF package front-end parses the DIF-based application specification and constructs an intermediate representation on which we apply our new algorithm for analysis and optimization of throughput-constrained buffer configurations. This algorithm is the main contribution of this chapter.

Our proposed algorithm for buffer optimization aims to minimize FPGA resource requirements for implementing the buffers associated with the edges in the input SDF graph. This buffer resource minimization is performed subject to the given throughput constraint. Careful estimation of throughput is performed in conjunction with our buffer optimization technique to ensure that the throughput constraint is satisfied without significant performance over-design (i.e. with actual throughput that is significantly higher that what is required based on the throughput constraint). Such over-design in general leads to buffer allocations that are larger than what is required to achieve the given throughput constraint, and is therefore counter-productive in terms of our throughput-constrained

buffer minimization problem.

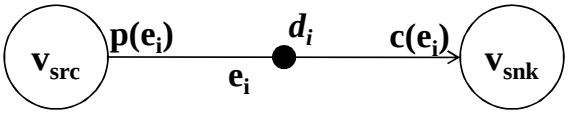
In our experimental setup, the result of our buffer optimization technique is applied by hand to the LabVIEW FPGA code of the target DSP system. Thus, we demonstrated a semi-automated design flow, where the result of our fully-automated buffer optimization algorithm is translated by hand to configure the buffers in the target implementation. Although this process is generally more time consuming compared to an end-to-end automated flow, it is a highly flexible approach because it can easily be adapted to different target platforms and back-end synthesis tools.

Through experiments with LabVIEW FPGA, we demonstrate significant improvements in resource efficiency, and minimal levels of performance over-design that result from the buffer configurations derived from our optimization technique.

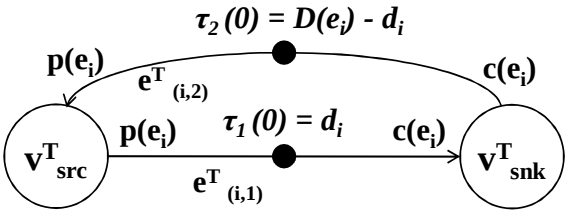
4.5 Two-actor SDF graph model (TASM)

We assume a *static buffering* approach for SDF graphs, which means that for each SDF edge we allocate a fixed amount of memory space at compile time. We refer to the fixed amount of space that is allocated for an edge e_i as the *buffer size* of e_i, and we denote this buffer size by the symbol $D(e_i)$. For real-time implementation of SDF graphs, static buffering is often preferable due to its enhanced predictability and elimination of overhead due to dynamic memory allocation.

In this section, we introduce a model called the *Two-Actor SDF Graph Model* (TASM). For any edge $e_i \in E$ in an arbitrary SDF graph $G = (V, E)$, the TASM for e_i, analyzed in terms of in SDF semantics, accurately captures the token transfer across e_i



(a) SDF Graph



(b) TASM

Figure 4.2: An example of an SDF edge and its TASM model.

as the enclosing SDF graph G executes under bounded memory. Also, TASM facilitates the formalization of our proposed synthesis approach, and its feature of computing buffer space requirements for throughput-optimal implementation.

4.5.1 Two-actor SDF graph model (TASM)

Suppose that edge e_i, shown in Fig. 4.5, is part of some arbitrary enclosing SDF graph $G = (V, E)$ (i.e., $e_i \in E$), and suppose that $src(e_i) = v_{src}$, $snk(e_i) = v_{snk}$, $del(e_i) = d_i$, and the production and consumption rates of e_i are denoted by $p(e_i)$ and $c(e_i)$, respectively. Suppose also that e_i is assigned a pre-specified buffer size $D(e_i)$. Then the TASM graph associated with e_i, which we denote by G_i^T, is defined as illustrated in Fig. 4.5(b). Here, $v_{src}^T = v_{src}$ and $v_{snk}^T = v_{src}$. Furthermore, $e_{(i,1)}^T$ connects v_{src}^T to v_{snk}^T with delay d_i, and $e_{(i,2)}^T$ connects v_{snk}^T to v_{src}^T with delay $(D(e_i) - d_i)$. The production and consumption rates for edges in the TASM graph are set as follows.

$$p(e_{(i,1)}^T) = c(e_{(i,2)}^T) = p(e_i) \tag{4.2}$$

and

$$c(e^T_{(i,1)}) = p(e^T_{(i,2)}) = c(e_i). \tag{4.3}$$

At any given time, buffer slots (cells in the memory that are allocated for the buffer) are categorized into two types based on whether they contain live data (*filled*) or whether they are available for storing new data (*empty* or *free*). The filled space in the buffer for e_i is modeled by $e^T_{(i,1)}$ in TASM. Thus as G^T_i executes, each token on $e^T_{(i,1)}$ represents a live token in the buffer associated with e_i in a corresponding execution of G. Since the source actor $src(e_i)$ can be fired only when e_i has enough free space to store all of the tokens produced by a firing of $src(e_i)$, each firing of $src(e_i)$ can be viewed as consuming $p(e_i)$ free cells from the buffer space available on e_i. Conversely, each execution of $snk(e_i)$ expands the free space on e_i by $c(e_i)$ cells. Hence, the free space on e_i can be modeled by the edge $e^T_{(i,2)}$ shown in Fig. 4.5(b), where each token on $e^T_{(i,2)}$ during an execution of G^T_i represents an empty cell in the buffer associated with e_i in a corresponding execution of G.

4.5.2 Modified self-timed execution (MSTE) in TASM

We use the self-timed execution model when mapping the input SDF graph into an FPGA implementation. Self-timed execution of SDF graphs can in general lead to execution periods (the patterns in which actors execute on the available resources) that are of exponential length in terms of the size of the of the graph (e.g., see [47]). Such exponential growth of execution periods can significantly complicate static analysis. To help address this difficulty, we add an additional firing rule, which we call the *MSTE firing*

rule: actor v_{src}^T in G_i^T (see Fig. 4.5(b)) of the TASM model cannot be fired if

$$\tau_1(t) \geq max(p(e_i), c(e_i)), \tag{4.4}$$

where $\tau_j(t)$ represents the number of tokens on $e_{(i,j)}^T$ at time t for $j = 1,2$. By imposing the MSTE firing rule, we obtain a modified form of self-timed execution, which we refer to in the remainder of this chapter as *modified self-timed execution* (*MSTE*).

We have empirically observed that this additional firing rule usually results in only relatively minor deviations from self-timed execution. However, imposing the rule leads to a periodic execution pattern S_P that is defined by the repetition vector of G. More precisely, by S_P in this context, we mean a finite-duration schedule onto the disjoint subsets, r_{src} and r_{snk}, of FPGA resources that are occupied by the actors v_{src} and v_{snk}, respectively. In other words, S_P can be viewed as a mapping

$$S_P : [0, 1, \ldots, (t_i - 1)] \times \{r_{src}, r_{snk}\} \rightarrow \{v_{src}^T, v_{snk}^T, v_{idle}\}, \tag{4.5}$$

where t_i is the length of the schedule (the period of the periodic pattern), and v_{idle} represents a void computation (idle resource). Note that even though S_P is formulated as a fully static schedule, it is implemented using our modified form of self-timed execution — i.e., the constraints imposed by our modified form of self-timed execution lead naturally to this kind of periodic pattern in the steady state. If t_s represents the time when this steady state pattern first emerges, (i.e., just after then point when the transient ends), then the schedules for all time intervals of the form $[(t_s + kt_i), (t_s + (k+1)t_i - 1)]$, for $k = 0, 1, \ldots$, can be obtained by appropriately-shifted versions of the schedule defined by (4.5).

Thus, if $\vec{q}$ represents the repetitions vector of G, then one period of S_P contains $q[v_{src}]$ and $q[v_{snk}]$ firings of v_{src}^T and v_{snk}^T, respectively. This kind of periodic schedule helps to significantly reduce the complexity of performance analysis since the iterative dataflow execution is characterized by a relatively compact periodic structure.

4.5.3 Subperiods in TASM

The entire firing pattern in an iteration (i.e., a single execution in the periodic repetition) of S_P can be expressed as a sequence of *subperiod*s, where by a subperiod (*SP*), we mean a smaller firing pattern within S_P. From the additional firing rule that we introduce in our implementation model, we are able to constrain execution so that it becomes more structured, which leads to potential for more efficient static analysis. Fortunately, the constraints imposed by the MSTE firing rule do not impose significant performance limitations, which we will demonstrate in the experiments that we present in Section 4.10.

An *SP* is defined as the time period between two consecutive *breakpoints* of actor execution, where the breakpoints are derived from two key conditions. The first condition, which we denote by $c_1(t)$, is

$$c_1(t) = c_{1,a}(t) \text{ and } c_{1,b}(t), \tag{4.6}$$

where

$$c_{1,a}(t) = (\tau_1(t) \geq max(p(e_i), c(e_i))), \text{ and} \tag{4.7}$$

$$c_{2,a}(t) = (\tau_1(t) < p(e_i) + c(e_i)). \tag{4.8}$$

We say that the first condition, Condition 1, "holds" or "is true" at time a given time instant θ if (4.7) is satisfied for $t = \theta$ (i.e., if both $c_{1,a}(\theta)$ and $c_{2,a}(\theta)$ hold). As we see from the MSTE firing rule, the source actor cannot be fired when Condition 1 is satisfied.

To introduce Condition 2, denoted by $c_2(t)$, it is useful to first define the following notion of *inter* and *intra* firing times of an actor. The set of inTRA firing times of an actor X, denoted by $TRA(X)$, is defined as the set of time instants during which actor X is executing. This set can be formulated as follows.

$$TRA(X) = \cup_{j=1}^{\infty} \{t \mid start(X, j) \leq t < end(X, j)\}.$$

Similarly, the set of inTER firing times of an actor X is defined as the set of time instants during which actor X is not executing ("idle"). This set can be expressed as the complement of the inTRA firing times of X with respect to the set $\mathbb{Z}^+$ of non-negative integers:

$$TER(X) = \mathbb{Z}^+ - TRA(X).$$

Condition 2 associated with the definition of breakpoints is defined as

$$c_2(t) = \begin{cases} \text{true}, & \text{if } t \in \{TER(A) \cap TER(B)\} \\ \text{false}, & \text{otherwise} \end{cases} \tag{4.9}$$

This condition represents that breakpoints do not occur during the execution time of either actor in TASM — breakpoints occur only "between" executions of v_{src}^T and v_{snk}^T. Based on the two conditions, $c_1(t)$ and $c_2(t)$, the k-th breakpoint, denoted $BP(k)$, is defined by

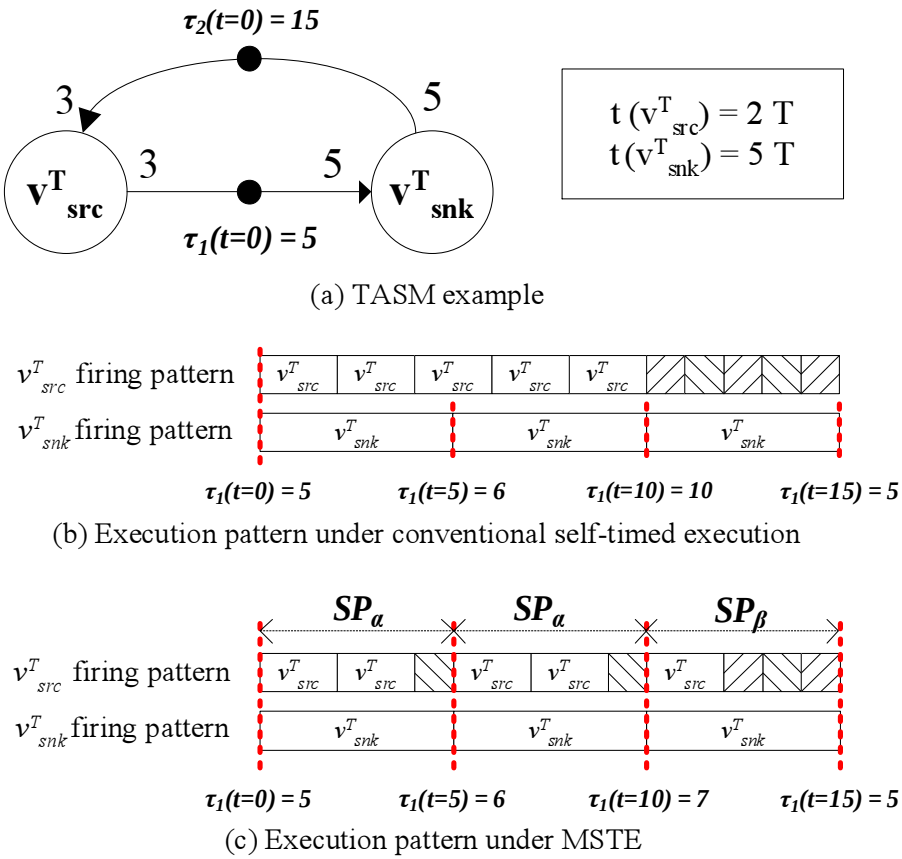


Figure 4.3: Example of TASM-based modeling approach, and execution patterns under conventional self-timed execution and MSTE.

$$BP(k) = min\{t \mid c_1(t) \wedge c_2(t) \wedge (t > BP(k-1))\}. \tag{4.10}$$

Fig. 4.3 shows a concrete example to illustrate our TASM-based modeling approach; Fig. 4.3(b) shows how the associated self-timed schedule evolves under conventional self-timed execution; and Fig. 4.3(c) shows how execution of the TSNM evolves under MSTE, our modified form of self-timed execution. In Fig. 4.3(c), $c_1(t)$ in (4.6) is true in the shaded areas of the timeline for v^T_{src}, and $c_2(t)$ is true at $t = 0, 5, 10, 15$ when both actor are idle. Hence, the dotted lines represents the breakpoints, and these break-

points divide an iteration of S_P into three subperiods.

Since there are enough tokens on edge e_1 to keep actor v^T_{snk} running after the second and fourth firings of actor v^T_{src} in Fig. 4.3(c), it is not essential (to achieve maximum throughput) for v^T_{src} to start its third and fifth firings immediately after its corresponding previous firings (i.e., v^T_{src} can remain idle for some time while v^T_{snk} consumes tokens from the edge (v^T_{src}, v^T_{snk}). Thus, in this example, MSTE does not reduce throughput performance compared to standard self-timed execution. Indeed on practical examples, we have found that typically the constraints imposed in MSTE do not affect throughput. At the same time, MSTE results in subperiods, which improve the efficiency with which we can analyze execution in terms of metrics that include performance and buffering requirements.

4.6 Properties of subperiods in TASM

As described in Section 4.5.2 and 4.5.3 , MSTE leads to efficient static analysis because an execution pattern under MSTE can be decomposed into a periodic pattern, and such a pattern can be further decomposed into a sequence of subperiods (smaller patterns). A subperiod can be more precisely defined as the time between successive breakpoints. For convenience in this discussion, let the greatest common divisor (GCD) of $p(e_i)$ and $c(e_i)$ be denoted by $g(e_i)$, and consider the following two mutually exclusive scenarios:

$$g(e_i) \neq min(p(e_i), c(e_i)), \tag{4.11}$$

and

$$g(e_i) = min(p(e_i), c(e_i)). \tag{4.12}$$

Under Scenario (4.11), we distinguish between two different types of subperiods that occur, and we refer to these types as SP_α and SP_β. Each of these two types consists of a fixed number of firings of v^T_{src} and v^T_{snk}. Thus, an iteration of S_P is a sequence of subperiods, where each subperiod in the sequence takes on one of two statically-known forms — SP_α and SP_β. The specific numbers of firings are summarized in Table 4.1. Here, $f_{SP_\lambda}(X)$ represents the number of firings of actor X that occur in a subperiod of type $\lambda \in \{\alpha, \beta\}$. For example, in Fig. 4.3(c), the first two of these subperiods are of type α, and the third is of type β.

Under Scenario (4.12), there exists only one type of subperiod in S_P. In this case, $p(e_i)$ divides $c(e_i)$ or $c(e_i)$ divides $p(e_i)$, and it follows that the numbers of firings in Table 4.1 for the source and sink actors are the same between the rows corresponding to type α and type β. In other words, under Scenario (4.12), SP_α and SP_β are identical, and thus, execution proceeds based on only one type of subperiod.

In summary, there are in general two types of subperiods to consider — SP_α and SP_β, and these forms are identical under Scenario (4.12). Execution within S_P can always be broken down into a succession of subperiods, where each individual subperiod conforms to one of these two forms. This is established by the following two lemmas. Basic notation related to TASM, which is used in our formulation of these lemmas, is summarized in Fig. 4.5(b). Proofs of theorems and lemmas are omitted throughout the chapter

Table 4.1: The number of firings of v_{src}^T and v_{snk}^T in subperiod α and β of TASM

	# of firings	$p(e_i) \geq c(e_i)$	$p(e_i) < c(e_i)$
Type α	$f_{SP_\alpha}(v_{src}^T)$	1	$\lceil c(e_i)/p(e_i) \rceil$
	$f_{SP_\alpha}(v_{snk}^T)$	$\lfloor p(e_i)/c(e_i) \rfloor$	1
Type β	$f_{SP_\beta}(v_{src}^T)$	1	$\lfloor c(e_i)/p(e_i) \rfloor$
	$f_{SP_\beta}(v_{snk}^T)$	$\lceil p(e_i)/c(e_i) \rceil$	1

due to lack of space.

Lemma 1. *Suppose that we are given a TASM G_i^T under MSTE. If $p(e_i) \geq c(e_i)$, then in each subperiod, v_{src}^T has exactly one firing, and v_{snk}^T has either $\lfloor p(e_i)/c(e_i) \rfloor$ or $\lceil p(e_i)/c(e_i) \rceil$ firings.*

Proof. Suppose that

$$p(e_i) \geq c(e_i) \text{ in } G_i^T. \tag{4.13}$$

Then from the definition of TASM, v_{src}^T produces $p(e_i)$ tokens on $e_{(i,1)}^T$. Thus, after the first firing of v_{src}^T in each sub-period, $\tau_1(t) \geq p(e_i)$. From (4.4), v_{src}^T cannot be fired again until $\tau_1(t)$ becomes smaller than $p(e_i)$. Meanwhile, each firing of v_{snk}^T reduces $\tau_1(t)$, and the breakpoint condition (see (4.10)) is satisfied before the next firing of v_{src}^T. Since, by definition, each sub-period ends at a breakpoint, there exists only one firing of v_{src}^T in each sub-period.

Now, we determine the number of firings of v_{snk}^T in a subperiod. Let breakpoint $BP(k)$ denote the time at which the kth subperiod starts within the encolsing schedule period S_P. As shown above, v_{src}^T is fired exactly once in each subperiod. Also, v_{snk}^T is fired continuously until $\tau_1(t)$ meets the next breakpoint condition. Thus, the number of v_{snk}^T

firings in a subperiod can be computed as

$$
\begin{aligned}
& f_{SP_\lambda}()v_{snk}^T \\
& = min\{j \mid \tau_1(BP(k)) + p(e_i) - j*c(e_i) < p(e_i) + c(e_i)\} \\
& = \lfloor \tau_1(BP(k))/c(e_i) \rfloor, \qquad (4.14)
\end{aligned}
$$

where $\lambda \in \{\alpha, \beta\}$. The left hand side in the inequality of (4.14) represents the change in $\tau_1(t)$ due to j firings of v_{snk}^T. The initial value of $\tau_1(t)$ is $\tau_1(BP(k))$ and $p(e_i)$ is added from the single firing of v_{src}^T. As v_{snk}^T gets fired, $\tau_1(t)$ is reduced until it meets the next breakpoint condition. From (4.6), the minimum and maximum of $\tau_1(BP(k))$ are $p(e_i)$ and $(p(e_i)+c(e_i)-g(e_i))$, respectively, because $\tau_1(t)$ is always a multiple of $g(e_i)$. We assign these minimum and maximum values of $\tau_1(BP(k))$ to (4.14) for computing $f_{SP_\lambda}()v_{snk}^T$. If $g(e_i) \neq c(e_i)$, then $f_{SP_\lambda}()v_{snk}^T$ is either $\lfloor p(e_i)/c(e_i) \rfloor$ or $\lceil p(e_i)/c(e_i) \rceil$ $(= \lfloor p(e_i)/c(e_i) + 1 \rfloor)$. On the other hand, if $g(e_i) = c(e_i)$, then $f_{SP_\lambda}()v_{snk}^T = p(e_i)/c(e_i)$. □

Lemma 2. *Suppose that we are given a TASM G_i^T under MSTE. If $p(e_i) < c(e_i)$, then in each subperiod, v_{snk}^T has exactly one firing, and v_{src}^T has either $\lfloor c(e_i)/p(e_i) \rfloor$ or $\lceil c(e_i)/p(e_i) \rceil$ firings.*

Proof. Suppose that

$$p(e_i) < c(e_i) \text{ in } G_i^T. \qquad (4.15)$$

First, we count the number of firings of v_{snk}^T in each subperiod. Let t_a be the time at which the first firing of v_{snk}^T completes. If $\tau_1(t_a) < c(e_i)$, then v_{snk}^T cannot be fired at time t_a because $\tau_1(t_a) < c_(e_i)$. In this case $\tau_1(t)$ will be increased as time progresses due to one or more firings of v_{src}^T, and eventually $\tau_1(t)$ will exceed $c(e_i)$ to satisfy the breakpoint condition (4.10) and terminate the subperiod.

On the other hand, if $\tau_1(t_a) \geq c(e_i)$, then it follows from (4.15) that condition $c_{1,a}$ in (4.7) holds. Furthermore, observe from the MSTE firing rule together with (4.15) that v_{src}^T cannot be fired if $\tau_1(t_a) \geq c(e_i)$. Also, a single firing of v_{src}^T adds $p(e_i)$ tokens to $\tau_1(t)$. Therefore, $\tau_1(t_a)$ is always smaller than $p(e_i) + c(e_i)$. Thus $c_{2,a}$ in (4.7) holds, and the overall breakpoint condition (see (4.10)) also holds. Thus, t_a marks the end of a subperiod, and by the definition of t_a, v_{snk}^T fires exactly once within this subperiod.

Next, we determine the number of firings of v_{src}^T in each subperiod. In a similar fashion to (4.14), the number of firings of v_{src}^T in each subperiod can be computed as

$$\begin{aligned} & f_{SP_\lambda}()v_{src}^T \\ &= min\{j \mid \tau_1(BP(k)) - c(e_i) + j * p(e_i) \geq c(e_i)\} \\ &= \lceil \{2 * c(e_i) - \tau_1(BP(k))\} / p(e_i) \rceil, \end{aligned} \tag{4.16}$$

where $\lambda \in \{\alpha, \beta\}$. The minimum and maximum of $\tau_1(BP(k))$ are then applied to (4.16) for computing $f_{SP_\lambda}()v_{src}^T$. If $g(e_i) = p(e_i)$, then $f_{SP_\lambda}()v_{src}^T$ is $c(e_i)/p(e_i)$. On the other hand, if $g(e_i) \neq p(e_i)$, then $f_{SP_\lambda}()v_{src}^T$ is either $\lceil c(e_i)/p(e_i) \rceil$ or $\lfloor c(e_i)/p(e_i) \rfloor$ $(= \lceil c(e_i)/p(e_i) - 1 \rceil)$.

□

From Lemma 1 and 2, it follows that the numbers of firings of v_{src}^T and v_{snk}^T in a subperiod can be determined as shown Table 4.1.

4.7 Throughput analysis in TASM

In this section, we analyze the pattern of actor firings in TASM and analyze the impact of allocated buffer sizes on the achieved throughput.

4.7.1 Firing pattern analysis

We begin with the following lemma, which relates tokens produced and consumed by the source and sink actors, respectively, in TASM.

Lemma 3. *Given a TASM G_i^T under MSTE, tokens produced by v_{src}^T in a subperiod are never consumed by v_{snk}^T in the same subperiod.*

Proof. We prove this Lemma by a contradiction. Suppose that

$$C_k > \tau_1(BP(k)), \tag{4.17}$$

where C_k is the number of tokens consumed by v_{snk}^T in the kth subperiod. (4.17) represents that at least one token produced by v_{src}^T in the kth subperiod is consumed by v_{snk}^T within the same kth subperiod. Let k' denote the index of the subperiod that immediately follows subperiod k — that is, $k' = 1$ (the first subperiod in the next periodic schedule iteration) if k represents the last subperiod within S_P, and otherwise, $k' = (k+1)$.

We examine two separate cases, and derive contraditions in both cases.

Case 1:

$$p(e_i) \geq c(e_i) \text{ in } G_i^T, \tag{4.18}$$

Let $\tau_1(BP(k)) = p(e_i) + \varepsilon$, where $BP(k)$ is the beginning of the kth subperiod, and $0 \leq \varepsilon < c(e_i)$. From (4.17), we have that the total number of tokens consumed by v_{snk}^T in the kth

subperiod must be greater than $(p(e_i)+\varepsilon)$. Since there is a single firing of v^T_{src} in each subperiod (from Lemma 1), we have that at the end of the kth subperiod,

$$\tau_1(BP(k')) = (p(e_i)+\varepsilon+p(e_i)-C_k). \tag{4.19}$$

Since $C_k > (p(e_i)+\varepsilon)$, $\tau_1(BP(k\prime)) < p(e_i)$. This contradicts $c_1, a(t)$ (see (4.6)), which was assumed to hold by the definition of a breakpoint.

Case 2: Now suppose that

$$p(e_i) < c(e_i) \text{ in } G^T_i. \tag{4.20}$$

From Lemma 2, we have that

$$f_{SP_\lambda}()v^T_{snk} = 1. \tag{4.21}$$

Let $\tau_1(BP(k)) = c(e_i)+\varepsilon$, and $0 \leq \varepsilon < p(e_i)$. From (4.17), we have that the total number of tokens consumed by v^T_{snk} in the kth subperiod must be greater than $(c(e_i)+\varepsilon)$. Since, from the definition of TASM, v^T_{snk} consumes $c(e_i)$ tokens on every firing, the number of firings of v^T_{snk} should be greater than one in the kth subperiod. This contradicts (4.21). □

Lemma 3 states that firing of v^T_{snk} is never *delayed* in a subperiod (i.e., it is not preceded by any idle time at the *start* of the subperiod). This is because v^T_{snk} does not need to wait for tokens produced from v^T_{src} in the same subperiod. While $(\tau_1(BP(k))$ tokens are always sufficient to avoid delaying v^T_{snk} during each subperiod k, v^T_{src} may be delayed (due to a value of $(\tau_2(BP(k))$ that is too small) if the allocated buffer size $D(e_i)$

is not sufficient. In other words, v_{src}^T can be delayed to wait for tokens on $e_{(i,2)}^T$ that must be produced by v_{snk}^T or equivalently, v_{src} waits until one or more firings of v_{snk} generate sufficient empty space in the buffer shown in Fig. 4.5(a). Hence, the firing pattern of v_{src}^T in a subperiod is in general a function of the allocated buffer size $D(e_i)$.

Before exploring a relationship between $D(e_i)$ and a firing pattern, we first show that $(\tau_1(BP(k))$ determines a type of kth subperiod. From the second breakpoint condition $c_2(t)$ in (4.9), the size of buffer($D(e_i)$) allocated on e_i of Fig. 4.5(a) can be represented by

$$D(e_i) = (\tau_1(BP(k)) + \tau_2(BP(k))) \tag{4.22}$$

Thus, $(\tau_1(BP(k))$ condition deciding a type of the subperiod is important in deriving a buffer size equation subject to a certain firing pattern of v_{src}^T and v_{snk}^T. The type of a subperiod is determined as a function of $\tau_1(BP(k))$. Our analysis here is divided into two cases — $p(e_i) \geq c(e_i)$, and $p(e_i) < c(e_i)$.

Case 1: first suppose that $p(e_i) \geq c(e_i)$. Then from Lemma 1, there exist $\lfloor p(e_i)/c(e_i) \rfloor$ and $\lceil p(e_i)/c(e_i) \rceil$ firings of v_{snk}^T in SP_α and SP_β, respectively. Since v_{snk}^T only consumes $\tau_1(BP(k))$ in each kth subperiod, as established by Lemma 3, we have that $\tau_1(BP(k))$ determines the type of a subperiod as follows.

$$SP_\lambda = \begin{cases} SP_\alpha, & \text{if } \tau_1(BP(k)) < \lceil p(e_i)/c(e_i) \rceil * c(e_i) \\ SP_\beta, & \text{otherwise} \end{cases} \tag{4.23}$$

Case 2: now suppose that $p(e_i) < c(e_i)$. Then from reasoning that is analogous to that in Case 1 above, the type of a subperiod is determined as follows.

$$SP_{\lambda} = \begin{cases} SP_{\alpha}, & \text{if } \tau_1(BP(k)) < 2*c(e_i) - \left\lfloor \frac{c(e_i)}{p(e_i)} \right\rfloor * p(e_i) \\ SP_{\beta}, & \text{otherwise} \end{cases} \tag{4.24}$$

We first derive a buffer size that is sufficient to guarantee that firings of v_{src}^T are never delayed. This derivation is general in the sense that it holds in the absence of information about the execution times of v_{src}^T and v_{snk}^T (beyond the assumption that the execution times are constant). Thus, this execution pattern analysis is useful for applications in which actor execution times are known to be constant, but whose constant values are not known exactly. Furthermore, this analysis provides a foundation for computing more tight buffer size requirements in the presence of known (constant) execution times (as we show in Section 4.8).

Theorem 1. *Suppose that we are given a TASM G_i^T under MSTE, and suppose that the buffer size is given by*

$$D(e_i) = max(p(e_i), c(e_i)) + p(e_i) + c(e_i) - g(e_i). \tag{4.25}$$

Then firings of v_{src}^T are never delayed in any subperiod.

Proof. We examine two possible cases for G_i^T, and derive a common buffer size equation in both cases.

Case 1:

$$p(e_i) \geq c(e_i) \text{ in } G_i^T, \tag{4.26}$$

From Lemma 1, there is a single firing of v_{src}^T in a subperiod. To fire v_{src}^T without any delay at the beginning of the kth subperiod, we must have that $\tau_2(BP(k)) \geq p(e_i)$ From (4.22),

the corresponding buffer size requirement is given by

$$D(e_i) \geq \tau_1(BP(k)) + p(e_i).$$

From (4.6), we know that $(p(e_i) + c(e_i) - g(e_i))$ is an upper bound for $\tau_1(BP(k))$. Thus, v_{src}^T can be executed without delay if

$$D(e_i) = 2 * p(e_i) + c(e_i) - g(e_i). \tag{4.27}$$

Case 2:

$$p(e_i) < c(e_i) \text{ in } G_i^T, \tag{4.28}$$

We divide Case 2 into two sub-cases:

- Case 2a: $g(e_i) \neq p(e_i)$.
- Case 2b: $g(e_i) = p(e_i)$.

We begin with Case 2a. In this case, we have that from Lemma 2, the numbers of firings of v_{src}^T in SP_α and SP_β are $\lceil c(e_i)/p(e_i) \rceil$ and $\lfloor c(e_i)/p(e_i) \rfloor$, respectively. Thus, in a type α subperiod, $\lceil c(e_i)/p(e_i) \rceil * p(e_i)$ is a lower bound for $\tau_2(BP(k))$ to achieve $\lceil c(e_i)/p(e_i) \rceil$ firings of v_{src}^T in a type α subperiod. From (4.24), an upper bound on $\tau_1(BP(k))$ for SP_α is given by

$$(2 * c(e_i) - \lfloor c(e_i)/p(e_i) \rfloor * p(e_i) - g(e_i)). \tag{4.29}$$

Thus, v_{src}^T can be executed without delay in a type α subperiod if

$$
\begin{aligned}
D(e_i) &\geq \tau_1(BP(k)) + \tau_2(BP(k)) \\
&= (2*c(e_i) - \lfloor c(e_i)/p(e_i) \rfloor * p(e_i) - g(e_i)) + \\
&\quad \{\lceil c(e_i)/p(e_i) \rceil * p(e_i)\} \\
&= 2*c(e_i) + p(e_i) - g(e_i). \qquad (4.30)
\end{aligned}
$$

Similarly, in a type β subperiod, $\lfloor c(e_i)/p(e_i) \rfloor * p(e_i)$ is a lower bound on $\tau_2(BP(k))$ to achieve $\lfloor c(e_i)/p(e_i) \rfloor$ firings of v_{src}^T. Also, from (4.6), we have that $c(e_i)+p(e_i)-g(e_i)$ is an upper bound on $\tau_1(BP(k))$ in a type β subperiod.

Thus, v_{src}^T can be executed without delay in a type β subperiod if

$$
\begin{aligned}
D(e_i) &\geq \tau_1(BP(k)) + \tau_2(BP(k)) \\
&= \{c(e_i)+p(e_i)-g(e_i)\} + \\
&\quad \{\lfloor c(e_i)/p(e_i) \rfloor * p(e_i)\} \qquad (4.31) \\
&= \lceil c(e_i)/p(e_i) \rceil * p(e_i) + c(e_i) - g(e_i).
\end{aligned}
$$

Because (4.30) is a sufficient condition of (4.31), v_{src}^T can be executed without delay in both α- and β-type subperiods if

$$D(e_i) = 2*c(e_i) + p(e_i) - g(e_i)$$

The right hand side of the last equation in (4.32) matches (4.25).

Now, we examine Case 2b: $g(e_i) = p(e_i)$, which means that $c(e_i) = z \times p(e_i)$ for some integer z, and from Table 4.1, the type α and type β subperiods are identical. In this

case, we have from (4.28), that the number of firings of v_{src}^T in any subperiod is z. Thus, to achieve z firings of v_{src}^T, we must have that $\tau_2(BP(k)) \geq c(e_i)$. Also, from (4.6), we have that $\tau_1(BP(k)) \leq c(e_i)$.

Thus, v_{src}^T can be executed without delay in any subperiod if

$$\begin{aligned} D(e_i) &\geq \tau_1(BP(k)) + \tau_2(BP(k)) \\ &= 2 * c(e_i) \\ &= max(p(e_i), c(e_i)) + p(e_i) + c(e_i) - g(e_i). \end{aligned} \tag{4.32}$$

Again, the right hand side of the last equation (in (4.32)) matches (4.25).

In summary, from our analysis of Cases 1, 2a, and 2b, we have that v_{src}^T is never delayed delayed in a subperiod if

$$D(e_i) \geq max(p(e_i), c(e_i)) + p(e_i) + c(e_i) - g(e_i). \tag{4.33}$$

□

In the next two theorems, we derive buffer size levels that are sufficient to guarantee certain kinds of firing patterns for v_{src}^T and v_{snk}^T. These firing patterns are useful in throughput analysis.

Theorem 2. *Suppose that G_i^T is executed under MTSE; $p(e_i) \geq c(e_i)$; γ is an integer satisfying $0 \leq \gamma \leq f_{SP_\alpha}(v_{snk}^T)$; and*

$$D(e_i) = 2 * p(e_i) + (1 - \gamma) * c(e_i) - g(e_i). \tag{4.34}$$

Then in any given subperiod, there is exactly one firing of v^T_{src}. Furthermore, if n firings of v^T_{snk} precede v^T_{src} in a given subperiod, then $n \leq \gamma$. In other words, the single firing of v^T_{src} in a subperiod occurs after at most γ firings of v^T_{snk}.

Proof. From Lemma 1, there is exactly one firing of v^T_{src} in any subperiod. Also, from the definition of TASM, v^T_{src} can be fired whenever $\tau_2(t) \geq p(e_i)$. Now recall that the number of tokens on $e^T_{(i,2)}$ at the beginning of the kth subperiod is denoted by $\tau_2(BP(k))$. Clearly, the first γ firings of v^T_{snk} in a subperiod produce $(\gamma * c(e_i))$ tokens on $e^T_{(i,2)}$. Thus, v^T_{src} can be fired after the γ-th firing of v^T_{snk} within the kth subperiod if

$$\tau_2(BP(k)) + (\gamma * c(e_i)) \geq p(e_i). \tag{4.35}$$

From (4.6),

$$\tau_1(BP(k)) \leq p(e_i) + c(e_i) - g(e_i) \text{ for all } k. \tag{4.36}$$

It follows from (4.22), (4.35), and (4.36) that the single firing of v^T_{src} occurs after at most γ firings of v^T_{src} if

$$D(e_i) \geq 2 * p(e_i) + (1-\gamma) * c(e_i) - g(e_i).$$

□

Theorem 2 tells us how long a single firing of v^T_{src} is delayed in each subperiod when the buffer size for e_i in Fig. 4.5(a) is bounded.

Theorem 3. *Suppose that G_i^T is executed under MTSE; $p(e_i) < c(e_i)$; δ is an integer satisfying $0 \leq \delta \leq f_{SP_\beta}(v_{src}^T)$; and*

$$D(e_i) = (1+\delta)*p(e_i)+c(e_i)-g(e_i). \tag{4.37}$$

Then in any given subperiod, there is exactly one firing of v_{snk}^T. Furthermore, if n firings of v_{src}^T occur before the end of the v_{snk}^T firing in a given subperiod, then $n \leq \delta$. In other words, v_{src}^T is fired at most δ times in a subperiod before the single firing of v_{snk}^T completes.

Proof. From Lemma 2, there is exactly one firing of v_{snk}^T. Also, from the definition of TASM, v_{snk}^T does not produce any tokens on $e_{(i,2)}^T$ within a given subperiod before v_{snk}^T completes its firing in that subperiod. In other words, only $\tau_2(BP(k))$ "empty buffer slots" are filled when firing v_{src}^T. Thus, in each kth subperiod, v_{src}^T can be fired δ times before the single firing of v_{snk}^T completes if

$$\tau_2(BP(k)) \geq \delta * p(e_i). \tag{4.38}$$

From (4.6),

$$\tau_1(BP(k)) \leq p(e_i)+c(e_i)-g(e_i) \,\text{for all}\, k. \tag{4.39}$$

It follows from (4.22), (4.38), and (4.39) that v_{src}^T is fired at most δ times in a subperiod before the single firing of v_{snk}^T completes if

$$D(e_i) \geq (1+\delta)*p(e_i)+c(e_i)-g(e_i).$$

□

Theorem 3 tells us, for a given bounded buffer size, how many times v^T_{src} can be fired independently of v^T_{snk} within a given subperiod. After firing v^T_{src} δ times in a subperiod, any remaining firings of v^T_{src} are delayed until v^T_{snk} completes its execution.

4.7.2 Saturated TASM systems

In this section, we assume that the execution times of actors are constant and known apriori, and we develop methods for throughput analysis of MSTE under this assumption.

We begin by defining some notation.

Definition 1. *Suppose that we are given a TASM G^T_i. Then the execution times of v^T_{src} and v^T_{snk} are denoted by T(v^T_{src}) and T(v^T_{snk}), respectively.*

Intuitively, T(v^T_{src}) and T(v^T_{snk}) give the time required for each actor to complete a single firing on r_{src} and r_{snk}, respectively. Our development of throughput analysis for MSTE also involves the following definition.

Definition 2. *Suppose that we are given a TASM G^T_i that executes under MSTE, and suppose that in each subperiod, the resource r_{src} operates without any idle time — that is, $S_P(t, r_{src}) = v^T_{src}$ for all $t \in 0, 1, \ldots, (t_i - 1)$. Then we say that G^T_i is* source-saturated. *Similarly, if r_{snk} executes without any idle time, then we say that G^T_i is* sink-saturated.

For example, the execution pattern shown in Fig. 4.3(c) illustrates a sink-saturated scenario. Clearly, since the net production and consumption rates of v^T_{src} and v^T_{snk} are balanced across S_P, it follows that the original SDF graph (Fig. 4.5(a)) executes at its maximum achievable throughput if G^T_i is source- or sink-saturated. This is summarized in the following property.

Property 1. *A TASM that is source-saturated or sink-saturated executes at its maximum achievable throughput when it executes under MSTE.*

Due to the additional firing rule of MTSE (see (4.4)), the execution of TASM under MTSE has the following property.

Property 2. *Suppose that we denote the total non-idle time of the resource r_η in a type λ subperiod by*

$$T_{SP_\lambda}(r_\eta) = f_{SP_\lambda}(v_\eta^T) * T(v_\eta^T). \tag{4.40}$$

Then G_i^T is either source- or sink-saturated if

$$T_{SP_\lambda}(r_{src}) \geq T_{SP_\lambda}(r_{snk}) \text{ for all } \lambda \in \{\alpha, \beta\} \tag{4.41}$$

or

$$T_{SP_\lambda}(r_{src}) < T_{SP_\lambda}(r_{snk}) \text{ for all } \lambda \in \{\alpha, \beta\}. \tag{4.42}$$

In particular, G_i^T can be neither source- nor sink-saturated under two *corner cases*, which we denote as corner case 1 (CC1) and corner case 2 (CC2). CC1 corresponds to the condition that the following two inequalities both hold:

$$T_{SP_\alpha}(r_{src}) \geq T_{SP_\alpha}(r_{snk}) \tag{4.43}$$

and

$$T_{SP_\beta}(r_{src}) < T_{SP_\beta}(r_{snk}). \tag{4.44}$$

Similarly, CC2 corresponds to the condition that (4.45) and (4.46) both hold:

$$T_{SP_\alpha}(r_{src}) \quad < \quad T_{SP_\alpha}(r_{snk}) \tag{4.45}$$

and

$$T_{SP_\beta}(r_{src}) \quad \geq \quad T_{SP_\beta}(r_{snk}). \tag{4.46}$$

Equation (4.43) means that r_{snk} has nonzero idle time in each α-type subperiod, while (4.44) holds if r_{src} has nonzero idle time in each β-type subperiod. Clearly, neither r_{src} nor r_{snk} is saturated in such a system.

The corner cases CC1 and CC2 represent limitations in our MSTE approach since our guarantee of maximal throughput, as given by Property 1, does not apply under these cases. However, we observe that CC1 and CC2 do not apply to a broad class of practical systems — in particular, systems that contain functional blocks that perform as bottlenecks, where by a "bottleneck", we mean a block whose computational complexity is dominant over other functional blocks. For example, in the dataflow-based 3GPP-Long Term Evolution (LTE) protocol application developed in [54], the FFT block can be observed to be a bottleneck.

In Section 4.10 we present detailed experimental studies with three practical applications, all of which involve bottleneck actors and corresponding avoidance of the corner cases (CC1 and CC2) that prevent source- and sink-saturated execution.

4.8 Analysis of saturated systems

Motivated by our discussion on bottleneck actors and the practical relevance of source- and sink-saturated systems, we develop in this section a detailed analysis of

throughput-constrained buffer optimization for such systems. Throughout the remainder of this section, we assume that we are working with a source- or sink-saturated TASM — i.e., we assume that the corner cases CC1 and CC2 (defined in Section 4.7.2 do not hold).

Definition 3. *Suppose that we are given an SDF Graph $G = (V,E)$ that executes under MSTE, and suppose that the time duration of S_P (i.e., a single iteration of the periodic schedule) is denoted by t_i. Then by the throughput of an actor $v \in V$, which we represent by $\Phi(v)$, we mean the number of firings of v that execute per unit time. Since $q[v]$ firings of an actor v execute in each iteration of S_P, we have that*

$$\Phi(v) = q[v]/t_i \text{ for all } v \in V. \tag{4.47}$$

Furthermore, by the throughput of G_i^T, which we refer to as the *TASM throughput*, we mean the reciprocal of the time duration of S_P — i.e., $(1/t_i)$.

In this section, we show how to determine an upper bound on the buffer size required to execute G_i^T at its maximum achievable throughput. Henceforth, we refer to the reciprocal this maximum achievable throughput as t_{min}.

We remind the reader that although this analysis is developed for two-actor SDF graphs, the methods can be applied to arbitrary tree-structured SDF graphs, as described in Section 4.1, by using them on each edge (and the underlying two-actor subgraph) separately and combining the results.

Property 3. *Suppose that we are given a TASM G_i^T. From the definitions of S_P, t_i, t_{min}, and actor throughput, we have that*

$$(\Phi(v^T_{src}) \geq q[v^T_{src}]/t_{min}) \text{ and } (\Phi(v^T_{snk}) \geq q[v^T_{snk}]/t_{min})$$

$$\Rightarrow t_i \leq t_{min}$$

From Lemma 3, firings of v^T_{snk} are never delayed in a subperiod. Also, from Theorem 1, firings of v^T_{src} are not delayed in a subperiod if $D(e_i)$ is set according to (4.25). Thus, under such a setting for $D(e_i)$, each actor in G^T_i is fired throughout a subperiod without any dependency on the other TASM actor. Hence, we establish the following lower bound on the throughput of an actor in G^T_i under the buffer size given by (4.25):

$$\Phi(v^T_\eta) \geq \frac{min_{\lambda \in \{\alpha,\beta\}} f_{SP_\lambda}(v^T_\eta)}{max(T_{SP_\alpha}(r_\eta), T_{SP_\beta}(r_\eta))}$$
$$\text{for } \eta \in \{v^T_{src}, v^T_{snk}\}. \tag{4.48}$$

If the execution times of v^T_{src} and v^T_{snk} are known, then it may be possible to exploit this knowledge to relax the buffering requirements, and thereby save resources on the target FPGA device. In particular, we can reduce buffering requirements if after applying the reduced buffer size given by Theorem 2 or Theorem 3 (based on whether $p(e_i) \geq c(e_i)$ or $p(e_i) < c(e_i)$, respectively), the resulting throughput given by (4.48) still meets the given throughput constraint.

In a given enclosing SDF graph $G = (V, E)$, the minimum achievable iteration period for S_P is given by

$$t_{min} = T(v_{btlneck}) * q[v_{btlneck}], \tag{4.49}$$

where $v_{btlneck} = max_{v \in V}\{v|q[v] * T(v)\}$. If an iteration of S_P completes exactly every t_{min} time units, then we can conclude that $v_{btlneck}$ is source- or sink-saturated, and the overall TASM throughput cannot be increased further.

4.9 Application to general tree-structured SDF graphs

Our TASM analysis can be applied iteratively to determine buffer sizes for all edges in an arbitrary tree-structured SDF graph. This assumption of a tree-structured graph is needed to ensure that the "extra (feedback) edges" added by the TASM models for different SDF graph edges do not "interact" (i.e., introduce new directed cycles in the overall graph model). Many practical SDF graphs or subsystem models are tree-structured, including models for multi-stage sample rate conversion, and various kinds of filterbanks, as well as the JPEG and OFDM transmitter applications that we examine in Section 4.10.

Algorithm 1 provides a systematic procedure for determining buffer sizes for an arbitrary, tree-structured SDF graph in a way that guarantees that the achieved performance will satisfy a given throughput constraint. The output of this procedure is a buffer size function $D : E \rightarrow Z_{pos}$, where Z_{pos} denotes the set of positive integers. The complexity of this algorithm is $O(E)$, which renders the approach practical for DSP and FPGA design tools.

4.10 Experimental results

We have implemented Algorithm 1 in the DIF environment [5], and applied it to three relevant signal processing applications — a CDtoDAT (compact disc to digital audio

Algorithm 1

INPUT : Tree-structured SDF graph $G = (V, E)$
: Actor execution times $T : V \rightarrow Z_{pos}$
OUTPUT: Buffer sizes, $D : E \rightarrow Z_{pos}$
procedure TASM-BUFFERING(G)
  for each $e \in E$ **do**
    $p \leftarrow p(e); c \leftarrow c(e);$
    $t_{src} \leftarrow T(src(e)); t_{snk} \leftarrow T(snk(e))$
    if $p \geq c$ **then**
      $\gamma = f_{\alpha}(snk(e)) - \lceil t_{src}/t_{snk} \rceil$
      $D(e) \leftarrow$ apply Theorem 2 with γ
    else
      $\delta = \lfloor t_{snk}/t_{src} \rfloor$
      $D(e) \leftarrow$ apply Theorem 3 with δ
    end if
  end for
end procedure

tape) sample rate converter, JPEG encoder, and DVB-T OFDM transmitter for digital video broadcasting as shown in Fig. 4.4. Using National Instruments LabVIEW FPGA 8.5, we have developed FPGA implementations of these three applications along with corresponding buffer size computation results from Algorithm 1.

LabVIEW is a graphical, dataflow-based programming environment for embedded system design. LabVIEW features for HDL (hardware description language) synthesis along with LabVIEW's dataflow orientation make the tool well-suited to FPGA-based design of signal processing applications. We have targeted the Xilinx Virtex II Pro P30 embedded in the National Instruments PCI-5640R digital system prototyping board to synthesize SDF-based application graphs with the buffer size functions computed by Algorithm 1. The base clock rate for our experiments is 40 MHz.

In the CDtoDAT application, the FIR filter in the first conversion stage becomes the bottleneck ($v_{btlneck}$ in (4.49)) of the system due to the high number of taps. Similarly, a

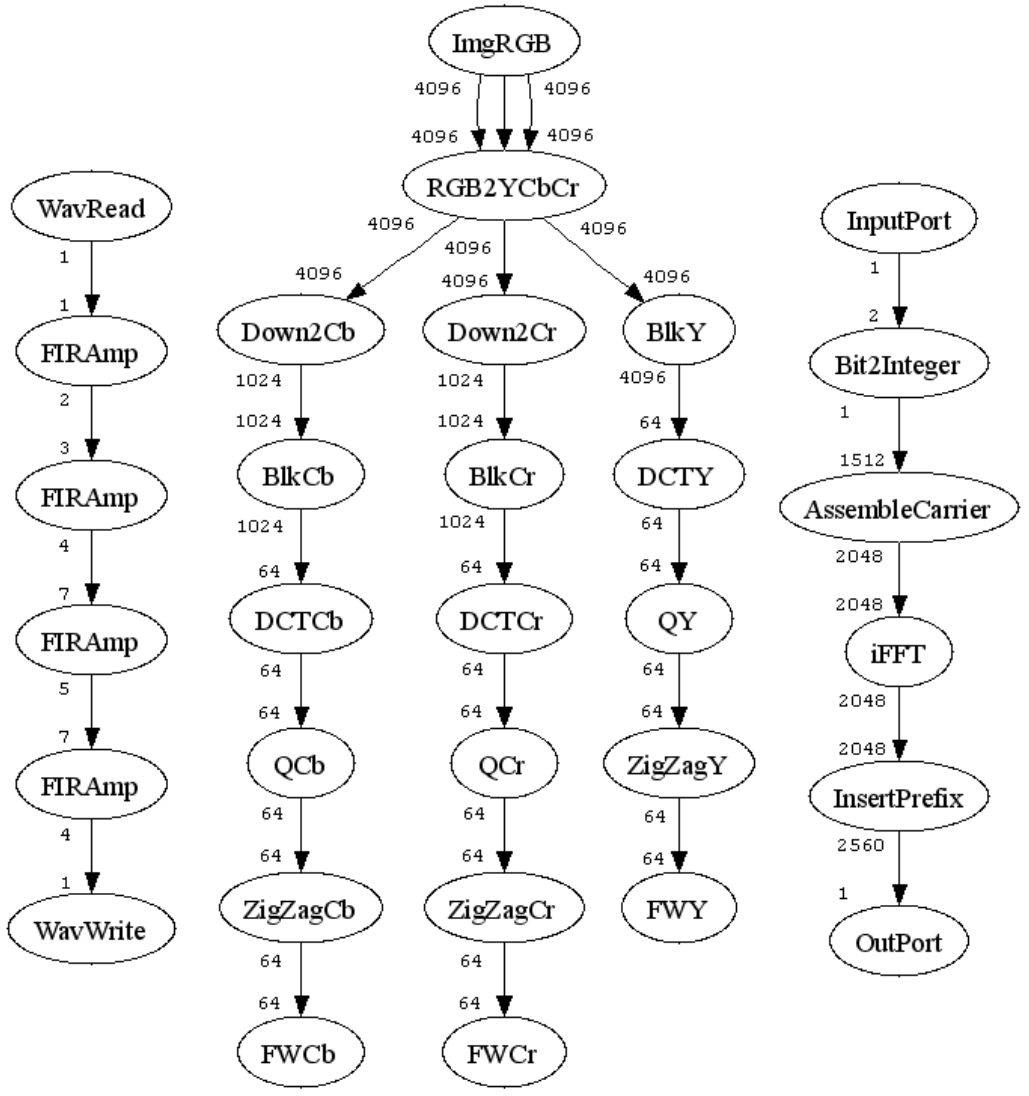


Figure 4.4: DIF-based Application specifications

Table 4.2: Sum of result buffer distribution under the maximum throughput(samples/cycle) and its synthesis result

		CDtoDAT	JPEG Encoder	DVB-T OFDM
Algorithm Result	Throughput	$5.6 * 10^{-3}$	.159	.191
	Buffer Sum	32	34112	9179
Synthesis Result	FPGA Slices	5438	8105	2810
	Block RAM	5	41	36
	18x18 MULT	5	41	36

discrete cosine transform (DCT) block in the JPEG encoder and the inverse FFT block in the DVB-T OFDM transmitter are bottlenecks for their respective applications.

Table 4.2 shows the results of Algorithm 1 and the associated synthesis results on targeted FPGA. Based on the synthesis results for the three applications, we verified that all of the solutions operate at the corresponding maximum achievable throughput levels, which correspond to the absence of idle time in the execution profiles of the resources that execute the associated bottleneck actors. Our results are therefore consistent with our theoretical results, which guarantee throughput optimality under the buffer sizes derived from Algorithm 1.

Chapter 5

Hardware synthesis technique for parameterized dataflow model

5.1 Introduction

The ever increasing demand for richer applications and multimedia content in mobile devices has fueled the continuous evolution of wireless standards towards bringing higher data rates and lower latencies to the end user. The third-generation partnership project (3GPP) has responded to this by recently finalizing the latest cellular standard called *long-term evolution* (*LTE*) [55]. LTE promises data rates of up to 300 Mbps in the downlink, 150 Mbps in the uplink, spectrum flexibility from 1.4 to 20 MHz, and mobility support from stationary users all the way to high-speed train speeds with a graceful degradation of service. In order to meet these demanding requirements, both base station and user equipment also require much higher complexity than ever before. In order to meet the ever tightening time-to-market requirements and resource constraints, the ability to quickly design, simulate, and prototype complex communication systems such as LTE is becoming more and more valuable to equipment vendors and network operators alike. The ability to input a design at an appropriate level of abstraction, and having the tools to make necessary trade-offs early in the design process are becoming more and more crucial in this rapidly evolving marketplace.

Synchronous dataflow (SDF) [1] has been used widely as an efficient model of computation (MOC) to analyze performance and resource requirements when implementing

DSP algorithms on various kinds of target architectures (e.g., see [56, 57]). The SDF model has been incorporated in many commercial tools for DSP system design, such as ADS from Agilent, Signal Processing Designer from CoWare, and System Studio from Synopsys. In SDF semantics, DSP applications are modeled by directed graphs in which vertices (*actors*) correspond to computational blocks, and edges represent the passage of data between blocks. SDF imposes the restriction that the number of data values (tokens) that is produced on each output edge is constant per actor execution (*firing*), and similarly, the number of tokens consumed per firing is constant for each actor/input-edge pair. Thus, SDF does not accommodate actors that can have dynamically varying token production and consumption rates. Such "dynamic dataflow" actors are employed in many modern DSP applications, including the LTE physical layer, and therefore, when developing such applications, we must explore models of computation that are more general than pure SDF.

Parameterized dataflow (PSDF) is a generalization of SDF that allows dynamically-changing production and consumption rates that are formulated in terms of changes to parameters of *parameterized SDF graphs* (PSDF graphs) [58]. A PSDF graph can be viewed as a parameterized family of graphs such that each instance in the family (i.e., each specific setting of the parameters) corresponds to an SDF graph. PSDF significantly improves upon the expressive power of SDF while providing a framework in which many SDF analysis techniques can be naturally adapted into parameterized versions. For example, techniques for constructing efficient parameterized looped schedules have been developed for PSDF graphs [58]. These scheduling techniques can provide for efficient simulation or software synthesis from PSDF specifications.

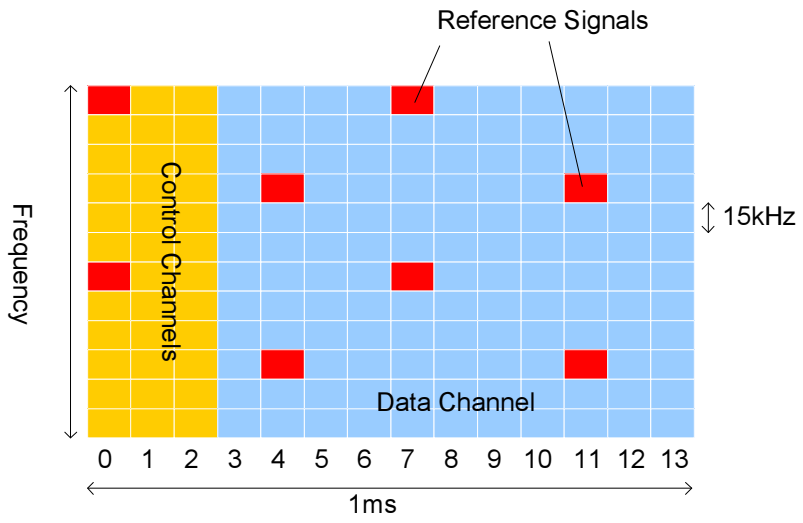


Figure 5.1: Example LTE subframe showing multiplexing of various channels on a 2D time-frequency grid (not to scale).

In this chapter, we apply PSDF to modeling the LTE physical layer protocol at the dataflow model level of the design methodology illustrated in Fig. 1.1. A distinguishing aspect of our approach is that we develop a PSDF-based *hardware synthesis* framework for efficient utilization of parallel processing capabilities in FPGAs. In contrast, the parameterized looped schedules described above have been designed for single-processor, software-based implementations. Also, our work develops novel connections among model-based DSP system design, FPGA implementation, and next generation wireless communication systems, which lead to systematic, formally supported design methods for hardware implementation in this domain.

5.2 Background

5.2.1 LTE downlink physical layer

The LTE downlink physical layer is based on the modulation and multiple access scheme called Orthogonal Frequency Division Multiple Access, or OFDMA. OFDMA uses an IFFT to divide a wideband channel into multiple narrowband subchannels. This creates a two-dimensional resource grid in frequency and time. In LTE, each element of this grid is called a *resource element*. This 2D grid allows multiplexing various physical channels, e.g., data and control channels, which could be intended for possibly multiple users. An example 1ms LTE subframe comprising 14 OFDMA symbols in the normal cyclic prefix mode is shown in Fig. 5.1. LTE can be configured for 6 different bandwidths, namely 1.4, 3, 5, 10, 15, and 20 MHz, but still maintain a constant 15 kHz subcarrier spacing. The LTE physical layer can also support multiple antenna transmission schemes, including transmit diversity, beamforming, and spatial multiplexing, but we primarily focuses on implementation for the single-antenna transmission mode.

5.2.2 Parameterized Synchronous Dataflow

Parameterized Synchronous Dataflow(PSDF) [58] extends the expressive power of SDF to manage DSP application dynamics in terms of run-time configuration of dataflow actor, edge, and subsystem parameters. A PSDF subsystem that is enabled for run-time configuration involves two separate "parameter configuration controllers," which are referred to as the *init* and *subinit* graphs of the associated subsystem. These controllers

provide two different levels of granularity in the run-time configuration processing — the init graph can form parameter configurations that are in general less restricted but also less frequent compared to the kinds of configurations that are allowed by the subinit graph.

The modeling discipline imposed by the subinit and init graphs in PSDF is designed to provide significant flexibility in how and when parameters are configured, while ensuring that configurations that affect the structure of subsystem schedules are allowed to occur only between iterations (in terms of SDF repetitions vectors) of the associated subsystems. This allows each subsystem to be viewed as a dynamically evolving sequence of SDF graphs whose SDF properties can change only at well-defined points in time (between SDF graph iterations). Such a structured view of dynamic dataflow graph execution is valuable for efficient quasi-static scheduling [58, 59, 60].

5.3 Parameterized SDF Model of LTE

5.3.1 LTE specification

Fig. 5.2 shows our PSDF model for a single-antenna LTE Base Station Modulator, which is the basis of our FPGA implementation. Each of the solid blocks correspond to PSDF actors whose production and consumption rates at their solid edges can change given the value of the parameters indicated by the dashed blocks communicated by the dashed edges. The data, control, and reference symbol generation blocks provide the QPSK, 16-, or 64-QAM symbols that are multiplexed via the Resource Element (RE) mapper. The RE mapper takes in different numbers of symbols s_1, s_2, and s_3 from the available input ports as a function of the number of control symbols

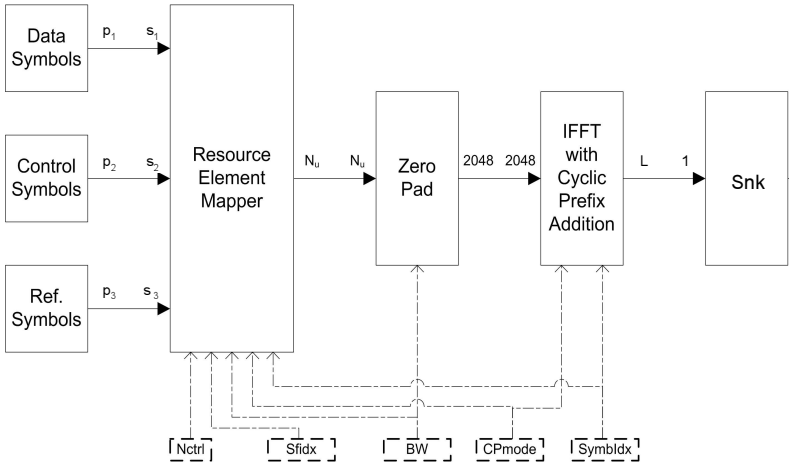


Figure 5.2: PSDF Model for LTE BS Modulator.

($N_{ctrl} \in \{1,2,3,4\}$), subframe index ($Sfidx \in \{0,..,9\}$), bandwidth configuration ($BW \in \{1.4,3,5,10,15,20\}$), cyclic prefix mode ($CP_{mode} \in \{Normal, Extended\}$), and symbol index ($SymbIdx \in \{0,..,13\}$). These symbols are multiplexed into $N_u \in \{72,180,300,600,900,1200\}$ used subcarriers, which is a direct map from the bandwidth configuration BW. The Zero Pad block then takes in N_u symbols and appends zeros at the DC and edge subcarriers forming 2048 frequency domain complex values. The following block then performs a 2048-pt IFFT, and appends a cyclic prefix of length that is a function of the CP_{mode} and $SymbIdx$ parameters. The rate at the output of this block should be 30.72 Ms/s with a worst case bandwidth of 20 MHz, and so in order to interface to the 25 MHz D/A converter in our hardware platform, we require a fixed 625/768 FIR rational sampling rate converter.

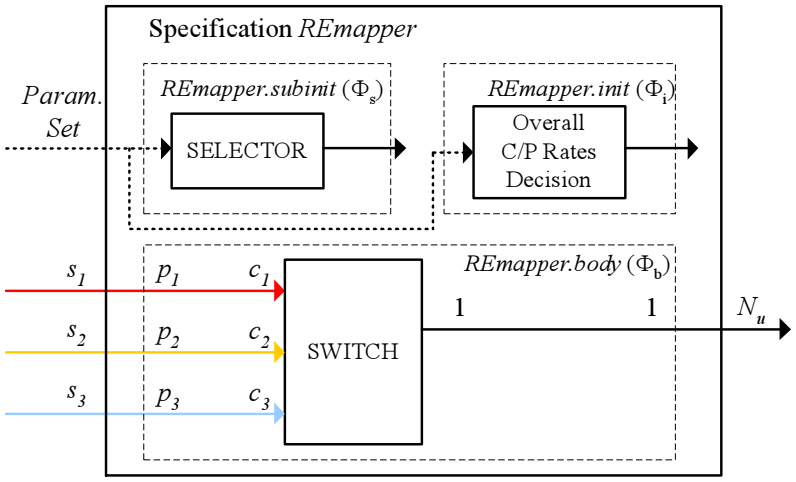


Figure 5.3: PSDF specification of RE Mapper.

5.3.2 PSDF Modeling Details

A PSDF specification for the RE mapper is shown in Fig. 5.3. Since there are different bandwidth configurations allowed, and each symbol of the LTE subframe is composed of different combinations of physical channel symbols (see Fig. 5.1), production and consumption rates in the RE mapper subsystem can be changed across OFDMA symbols, i.e., across the invocations of the RE mapping subsystem. Meanwhile, in order to multiplex the combination of physical channel types in each OFDMA symbol, the appropriate input edge is connected to the output edge for each resource element in the OFDMA symbol during each invocation of the RE mapper. We have likewise modeled the other processing blocks in the downlink LTE physical layer protocol, and have verified that PSDF has sufficient expressive power for describing the full functionality of our target LTE protocol.

PSDF specifications support *hierarchical* reconfigurable subsystem modeling structures in that a PSDF specification can be abstracted as a hierarchical PSDF actor, and embedded in a parent (higher level) PSDF graph. For example, a PSDF abstraction of the

RE mapper in Fig. 5.2 is considered as a PSDF specification consisting of a *body* graph (Φ_b), *init* graph (Φ_i), and *subinit* graph (Φ_s), as shown in Fig. 5.3.

Before the invocation of the *RE Mapper* PSDF specification, the *init* graph receives the parameter set, determines the physical channel data combinations in the particular OFDMA symbol, and counts the number of REs allocated for each physical channel to determine production rates on input edges and consumption rates on output edges in the parent graph. During the invocation of the specification, the subinit graph determines production and consumption rates on internal edges in order to switch the input edge connected to an output edge depending on the value of the received remapping matrix data at run-time. Based on the distribution of active and inactive edges, the body graph, which implements the computational core of the subsystem, can produce a sequence of data corresponding to the OFDMA symbol index. Hence, in the architecture of our parameterized dataflow framework, the body graph models the main functional behavior of the RE mapper, while the init and subinit graphs provide two different levels of control based on the given, dynamically arriving parameter sets.

5.3.3 PSDF Execution Model

Each LTE subframe is composed of multiple OFDMA symbols, and each OFDMA symbol in our PSDF specification is processed after all actors in the graph are fired at the rate determined by the repetitions vector of the enclosing graph. Because PSDF semantics guarantees that any specific configuration of a PSDF graph is an SDF graph, and that such configurations can only be changed between SDF graph iterations, there is always a well-

defined repetitions vector that governs the processing of a given OFDMA symbol. For details on fundamental relationships between SDF graphs and repetitions vectors, we refer the reader to [1].

When executing the LTE FPGA implementation, we apply a self-timed execution model, which means that each actor should be fired as soon as all of its input edges have sufficient data. When actors execute and communicate on dedicated resources (so that resource contention is not an issue), this type of execution generally enhances throughput by facilitating the exploitation of parallel processing capabilities on the target hardware. This type of distributed-control execution model also avoids hardware and run-time overhead due to the stronger synchronization requirements that are associated with centralized-control schedules.

FPGA targets allow dataflow actors to be assigned onto independent, dedicated processing units that are implemented by FPGA slices. In such a computing environment, signal processing throughput can be significantly increased due to the possibility for simultaneous firings of multiple actors. To ensure valid, distributed firing rule checking in our PSDF-based implementation framework, we model empty memory spaces on dataflow graph edges by adding feedback edges with appropriate numbers of initial tokens (based on the sizes of the corresponding buffers) in the execution model graph (an intermediate dataflow graph representation used to map the application into hardware), and enable actors for execution using principles of efficient self-timed execution [47].

Wiggers et al. have employed a similar *backpressure-driven*, self-timed execution model to implement cyclo-static dataflow (CSDF) graphs in multi-processor system-on-chip devices [61]. Our approach is in this chapter differs in its exploration of PSDF,

Table 5.1: FPGA resource utilization for LTE implementation.

Occupied FPGA Slices	5,244 out of 14,720 (35%)
Number of BlockRAM	96 out of 244 (39%)
Number of DSP48Es	54 out of 640 (8%)

which is a significantly more dynamic form of dataflow compared to SDF or CSDF, and its application to FPGA implementation.

5.4 LTE Prototype Implementation

As a proof-of-concept of our PSDF LTE model, we have designed and implemented from the top down an LTE real-time base station emulator prototype [62]. The prototype is based on a PXI-express system with an embedded real-time controller PC running a real-time operating system, which handles the link control, higher-layer software, and communication with an optional host PC via TCP-IP. The PSDF LTE model is designed in LabVIEW FPGA [63], and implemented on the PXIe-5641R, Intermediate Frequency (IF) Transceiver module, which includes a Xilinx Virtex-5 SX95T FPGA with integrated 2-input and 2-output IF ports. The IF signals are then modulated onto a radio frequency carrier using the PXI-5610 2.7 GHz RF upconverter, and looped-back to a PXI-5600 2.7 GHz RF downconverter, where the downconverted IF signal is fed back to the IF Transceiver for receiver processing. The base clock for our experiments with this system is 160 MHz. Synthesis results from the experiments are shown in Table 5.1.

As an illustrative example, we detail the implementation of the 625/768 sample rate conversion block of Fig. 5.2, which converts a 30.72MSPS LTE signal to the DAC at 25MSPS. In order to save hardware resources, we divide the filter into a cascade of two rational resampling stages, namely a 25/24 and a 25/32 stage. Using the LabVIEW Digital Filter Design Toolkit (DFDT), the individual floating point rational filters are designed and the fixed point behavior of the overall filter is simulated. We then used Xilinx's FIR compiler to implement the filter using the IP integration node from NI-Labs. This node uses an XCO or VHDL file as imports to build a simulation and implementation model compatible with LabVIEW FPGA.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we have presented new design techniques and methodologies for dataflow-based synthesis of field programmable gate array (FPGA) implementations for digital signal processing (DSP) applications. We have focused mainly on formulating and exploring design spaces for finding cost-efficient solutions subject to given constraints on performance. Our experimental results have demonstrated that the proposed techniques are highly effective in improving the efficiency of implementations on FPGA platforms.

In chapter 2, we developed a systematic approach for generating dedicated fast Fourier transform (FFT) subsystems for FPGA implementation. Our approach incorporates efficient FFT address generation and memory management, and applies two orthogonal loop unrolling methods to provide a tunable trade-off between performance and FPGA resource costs. We also developed an analytical approach for high level design space exploration. This approach allows one to derive a resource-efficient FFT architecture configuration for a given throughput constraint, and a given critical target resource (e.g., FPGA BRAM or logic slices).

Our methods are demonstrated through extensive synthesis experiments using the Xilinx Virtex II Pro FPGA device family. Our synthesis results quantify cost-performance trade-offs provided by our proposed class of FFT architectures. A distinguishing charac-

teristic of our approach, compared to commercially available FFT IP cores and other specialized FFT implementations, is that we provide a systematic method to generate an FPGA-based FFT architecture while taking into account trade-offs between performance and cost.

In chapter 3, we extended our 1D-FFT implementation technique to generate dedicated 2D-FFT subsystems for FPGA implementation. Our approach realizes data parallelism within an individual 1D-FFT core, and minimizes the interface complexity between the underlying 1D-FFT core and local memory. Our approach allows for scalable, parallel 2D-FFT implementation with a relatively simple interconnection network, and correspondingly simple control logic. These features contribute to improved FPGA resource consumption at a given level of performance compared to previous 2D-FFT FPGA architectures. Our synthesis results quantify the cost-performance trade-offs provided by our proposed class of FFT architectures.

In chapter 4, we presented a novel algorithm to provide upper bounds on FPGA buffer distributions for throughput-optimal execution of synchronous dataflow graphs that are in the form of tree-structured, directed acyclic graphs. The resulting bounds can be employed directly as buffer sizes when mapping SDF graphs into digital hardware. A distinguishing aspect of our proposed algorithm is that it has low polynomial time complexity, which makes it especially useful for rapid prototyping and for implementation of large scale or heavily multirate designs. Our work appears promising for integration into high-level design processes for FPGA-based DSP system implementation, as our experiments with the LabVIEW FPGA demonstrate.

In chapter 5, we presented a framework for the modeling and FPGA implementation

of LTE downlink physical layer processing using the parameterized synchronous dataflow (PSDF) model of computation. The results of our study and our associated prototype provide a concrete demonstration of PSDF-based design and implementation techniques for emerging wireless communication systems. Due to its formal properties, support for systematic scheduling and implementation techniques, and capabilities for efficient frame-based dynamic dataflow modeling, PSDF is promising as a semantic foundation for future design tools, and as an architectural foundation for digital system design methodologies in the domain of fourth generation wireless communication systems.

6.2 Future work

In this section, we describe a number of useful directions for future work that build on the results of this thesis.

The FFT actor architecture developed in chapter 2 has provided the framework and core design for the *burst mode built-in FFT IP block*, which is a new feature introduced in LabVIEW FPGA 8.6, released by National Instruments. This released version does not employ the inner/outer loop unrolling features in our FFT actor architecture framework. These features are presently integrated together with the other components of our FFT actor implementation approach within in-house distributions that are being used experimentally at National Instruments. These unrolling-enabled versions provide higher throughput FFT realizations, but are more complex and require more extensive experimentation before integration into the commercially released product.

Overall, our FFT architecture techniques are suitable as the basis for FFT IP blocks

that can be configured across a wide range of trade-offs between resource cost and achievable performance based on implementation requirements.

While radix-4 FFT designs are generally less resource-consuming compared to radix-2 FFT designs, radix-4 designs are more restricted in that the FFT size must be a power of 4, while the size for radix-2 must be a power of 2. To implement FFT sizes that are powers of 2, combined radix-2/4 architectures are attractive candidates. One interesting direction for further study is to apply our proposed unrolling techniques to combined radix-2/4 FFT architectures.

The proposed buffering algorithm in chapter 4 is restricted to SDF graphs, which are dataflow graphs that have static dataflow (production and consumption rate) behavior. While many DSP applications can be modeled using SDF graphs, an increasing range of applications require more flexibility and cannot be fully represented by SDF semantics. Furthermore, SDF-compatible behaviors can sometimes be synthesized more effectively if they are converted to alternative representations that employ more flexible modeling techniques such as cyclo-static dataflow (CSDF) [52]. Thus, extending our techniques for buffer analysis and optimization to more expressive dataflow models is a useful direction for further investigation.

Another interesting direction for future work is buffer optimization for parameterized dataflow graphs under self-timed execution. In chapter 5, we developed a novel PSDF-based FPGA architecture design approach, and demonstrated this approach using National Instrument's LabVIEW FPGA. In this work, we exploited the expressive power of parameterized dataflow, and demonstrated the mapping from a complex PSDF application specification into an FPGA implementation. Integrating buffer optimization into the

PSDF-to-FPGA mapping process will be a useful direction for further study to achieve more efficient hardware utilization in derived implementations.

Bibliography

[1] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, 1987.

[2] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck, "Software synthesis for dsp using ptolemy," *Journal of VLSI Signal Processing*, vol. 9, pp. 7–21, 1995.

[3] W. Sung, M. Oh, C. Im, and S. Ha, "Demonstration of codesign workflow in peace," in *in Proc. of International Conference of VLSI Circuit, Seoul, Koera*, 1997.

[4] J. Buck and R. Vaidyanathan, "Heterogeneous modeling and simulation of embedded systems in el greco," in *CODES '00: Proceedings of the eighth international workshop on Hardware/software codesign*. New York, NY, USA: ACM, 2000, pp. 142–146.

[5] C. Hsu, M. Ko, and S. S. Bhattacharyya, "Software synthesis from the dataflow interchange format," in *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, Dallas, Texas, September 2005, pp. 37–49.

[6] C. Hsu, J. L. Pino, and S. S. Bhattacharyya, "Multithreaded simulation for synchronous dataflow graphs," in *Proceedings of the Design Automation Conference*, Anaheim, California, June 2008, pp. 331–336.

[7] W. Wolf, *FPGA-Based System Design*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[8] Xilinx, "Xilinx core generator 10.1," 2008. [Online]. Available: http://www.xilinx.com/ipcenter/coregen/updates_101.htm

[9] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, April 1965.

[10] S. Winograd, "On computing the discrete fourier transform," *Mathematics of Computation*, vol. 32, no. 141, pp. 175–199, 1978. [Online]. Available: http://www.jstor.org/stable/2006266

[11] D. Kolba and T. Parks, "A prime factor fft algorithm using high-speed convolution," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 25, no. 4, pp. 281–294, Aug 1977.

[12] R. Bracewell, "The fast hartley transform," *Proceedings of the IEEE*, vol. 72, no. 8, pp. 1010–1018, Aug. 1984.

[13] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *IEEE Intl. Conf. Acoustics Speech and Signal Processing*, vol. 3, 1998, pp. 1381–1384.

[14] A. Ganapathiraju, J. Hamaker, and J. Picone, "Contemporary view of fft algorithms," in *Proceedings of the IASTED International Conference on Signal and Image Processing (SIP '98)*, 1998, pp. 130–133.

[15] B. Baas, "A low-power, high-performance, 1024-point fft processor," *Solid-State Circuits, IEEE Journal of*, vol. 34, no. 3, pp. 380–387, Mar 1999.

[16] W. Li and L. Wanhammar, "A pipeline fft processor," in *In IEEE Workshop on Signal Processing Systems*, 1999, pp. 654–662.

[17] I. Uzun, A. Amira, and A. Bouridane, "Fpga implementations of fast fourier transforms for real-time signal and image processing," *Vision, Image and Signal Processing, IEE Proceedings* -, vol. 152, no. 3, pp. 283–296, June 2005.

[18] S. Sukhsawas and K. Benkrid, "A high-level implementation of a high performance pipeline fft on virtex-e fpgas," *VLSI, 2004. Proceedings. IEEE Computer society Annual Symposium on*, pp. 229–232, Feb. 2004.

[19] J. Vite-Frias, R. Romero-Troncoso, and A. Ordaz-Moreno, "Vhdl core for 1024-point radix-4 fft computation," *Reconfigurable Computing and FPGAs, 2005. ReConFig 2005. International Conference on*, pp. 4 pp.–24, Sept. 2005.

[20] C. Chad, Z. Qin, X. Yingke, and H. Chengde, "Design of a high performance fft processor based on fpga," *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, vol. 2, pp. 920–923 Vol. 2, Jan. 2005.

[21] C. Gonzalez-Concejero, V. Rodellar, A. Alvarez-Marquina, E. M. d. Icaya, and P. Gomez-Vilda, "An fft/ifft design versus altera and xilinx cores," *Reconfigurable Computing and FPGAs, 2008. ReConFig '08. International Conference on*, pp. 337–342, Dec. 2008.

[22] Xilinx, "Fast fourier transform v4.1," 2007.

[23] Y. Ma, "An effective memory addressing scheme for fft processors," *Signal Processing, IEEE Transactions on*, vol. 47, no. 3, pp. 907–911, Mar 1999.

[24] G. Nordin, P. A. Milder, J. C. Hoe, and M. Püschel, "Automatic generation of customized discrete fourier transform ips," in *DAC '05: Proceedings of the 42nd annual conference on Design automation*. New York, NY, USA: ACM, 2005, pp. 471–474.

[25] J. H. Takala, T. S. Ja"rvinen, P. V. Salmela, and D. A. Akopian, "Multi-port interconnection networks for radix-r algorithms," in *In Proc. IEEE Intl. Conf. Acoustics, Speech, Signal Processing*, 2001, pp. 1177–1180.

[26] H. Kee, N. Petersen, J. Kornerup, and S. S. Bhattacharyya, "Systematic generation of FPGA-based FFT implementations," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Las Vegas, Nevada, March 2008, pp. 1413–1416.

[27] M. Hasan and T. Arslan, "Fft coefficient memory reduction technique for ofdm applications," *Acoustics, Speech, and Signal Processing, 2002. Proceedings. (ICASSP '02). IEEE International Conference on*, vol. 1, pp. I–1085–I–1088 vol.1, 2002.

[28] C. Burrus, "Unscrambling for fast dft algorithms," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 36, no. 7, pp. 1086–1087, Jul 1988.

[29] J. M. Blackledge, *Digital Image Processing*. Horwood Publishing, 2005.

[30] H. Jung and S. Ha, "Hardware synthesis from coarse-grained dataflow specification for fast hw/sw cosynthesis," in *In Proceedings of Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES/ISSS*, 2004, pp. 24–29.

[31] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "Synthesis of embedded software from synchronous dataflow specifications," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 21, no. 2, pp. 151–166, June 1999.

[32] R. Reiter, "Scheduling parallel computations," *Journal of the Association for Computing Machinery*, October 1968.

[33] A. Dasdan, A. Dasdan, R. K. Gupta, and R. K. Gupta, "Faster maximum and minimum mean cycle algorithms for system-performance analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, pp. 889–899, 1998.

[34] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee, "A hierarchical multiprocessor scheduling system for DSP applications," in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, November 1995, pp. 122–126 vol.1.

[35] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. J. G. Bekooij, "Throughput analysis of synchronous data flow graphs," in *ACSD '06: Proceedings of the Sixth International Conference on Application of Concurrency to System Design*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 25–36.

[36] J. T. Buck, "Scheduling dynamic dataflow graphs with bounded memory," Berkeley, CA, USA, Tech. Rep., 1993.

[37] R. Govindarajan, G. R. Gao, and P. Desai, "Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks," *Journal of VLSI Signal Processing*, vol. 31, p. 2002, 1994.

[38] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A formal approach to the scheduling problem in high level synthesis," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 10, no. 4, pp. 464–475, 1991.

[39] Q. Ning and G. R. Gao, "A novel framework of register allocation for software pipelining," 1993.

[40] H. Oh and S. Ha, "Efficient code synthesis from extended dataflow graphs for multimedia applications," in *DAC '02: Proceedings of the 39th annual Design Automation Conference*. New York, NY, USA: ACM, 2002, pp. 275–280.

[41] J. Horstmannshoff and H. Meyr, "Efficient building block based rtl code generation from synchronous data flow graphs," in *DAC '00: Proceedings of the 37th Annual Design Automation Conference*. New York, NY, USA: ACM, 2000, pp. 552–555.

[42] ——, "Optimized system synthesis of complex rt level building blocks from multirate dataflow graphs," in *ISSS '99: Proceedings of the 12th international symposium on System synthesis*. Washington, DC, USA: IEEE Computer Society, 1999, p. 38.

[43] E. Stuijk, M. Geilen, and T. Basten, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *In DAC*. ACM Press, 2006, pp. 899–904.

[44] M. Geilen, T. Basten, and E. Stuijk, "Minimising buffer requirements of synchronous dataflow graphs with model checking," in *in Proceedings of the Design Automation Conference*. ACM, 2005, pp. 819–824.

[45] M. Wiggers, M. Bekooij, P. G. Jansen, and G. J. M. Smit, "Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure," in *CODES+ISSS*, 2006, pp. 10–15.

[46] E. A. Lee and S. Ha, "Scheduling strategies for multiprocessor real time DSP," in *Proceedings of the Global Telecommunications Conference*, November 1989.

[47] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, 2nd ed. CRC Press, 2009.

[48] S. Y. Kung, P. S. Lewis, and S. C. Lo, "Performance analysis and optimization of VLSI dataflow arrays," *Journal of Parallel and Distributed Computing*, pp. 592–618, 1987.

[49] C. Hsu, F. Keceli, M. Ko, S. Shahparnia, and S. S. Bhattacharyya, "DIF: An interchange format for dataflow-based design tools," in *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, Samos, Greece, July 2004, pp. 423–432.

[50] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[51] W. Sun, M. J. Wirthlin, and S. Neuendorffer, "Fpga pipeline synthesis design exploration using module selection and resource sharing," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 254–265, 2007.

[52] 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.

[53] 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.

[54] H. Kee, I. Wong, Y. Rao, and S. S. Bhattacharyya, "FPGA-based design and implementation of the 3GPP-LTE physical layer using parameterized synchronous dataflow techniques," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Dallas, Texas, March 2010.

[55] G. Americas, *The Mobile Broadband Evolution: 3GPP Release 8 and beyond*, Feb. 2009.

[56] C. Hsu, S. Ramasubbu, M. Ko, J. L. Pino, and S. S. Bhattacharyya, "Efficient simulation of critical synchronous dataflow graphs," in *Proceedings of the Design Automation Conference*, San Francisco, California, July 2006, pp. 893–898.

[57] C. B. Robbins, *Autocoding Toolset Software Tools for Automatic Generation of Parallel Application Software*. Technical report, Management Communications and Control, Inc., 2002.

[58] 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.

[59] S. Saha, S. Puthenpurayil, and S. S. Bhattacharyya, "Dataflow transformations in high-level DSP system design," in *Proceedings of the International Symposium on System-on-Chip*, Tampere, Finland, November 2006, pp. 131–136, invited paper.

[60] M. Ko, C. Zissulescu, S. Puthenpurayil, S. S. Bhattacharyya, B. Kienhuis, and E. Deprettere, "Parameterized looped schedules for compact representation of execution sequences in DSP hardware and software implementation," *IEEE Transactions on Signal Processing*, vol. 55, no. 6, pp. 3126–3138, June 2007.

[61] M. Wiggers, M. Bekooij, P. Jansen, and G. Smit, "Efficient computation of buffer capacities for cyclo-static real-time systems with back-pressure," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, 2007.

[62] I. Wong, Y. Rao, and M. Santori. Video: Prototyping complex communications systems. http://zone.ni.com/wv/app/doc/p/id/wv-1696.

[63] N. Instruments, *LabVIEW FPGA User Manual*, 2009.