System-Level Architectural Hardware Synthesis for Digital Signal Processing Sub-Systems

SHUO LI

Doctoral Thesis in Electronic and Computer Systems
Stockholm, Sweden 2016

TRITA-ICT 2015:28 School of ICT, KTH
ISBN 978-91-7595-799-9 KTH Kista, Kistagången 16, SE-164 40 Kista, Sverige

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framläges till offentlig granskning för avläggande av doktorsexamen in Elektronik och datorsystem torsdagen den 18 februari 2016 klockan 13.00 i Electrum Ka-Sal C, KTH Kista, Kistagången 16, SE-164 40 Kista, Sverige.

Tryck: Universitetsservice US AB

Abstract

This thesis presents a novel system-level synthesis framework called System-Level Architectural Synthesis Framework (SYLVA), which synthesizes Digital Signal Processing (DSP) sub-systems modeled by synchronous data flow into hardware implementations in Application-Specific Integrated Circuit (ASIC), Field-Programmable Gate Array (FPGA) or Coarse-Grained Reconfigurable Architecture (CGRA) style. SYLVA synthesizes in terms of pre-characterized Function Implementations (FIMPs). It explores the design space in three dimensions, number of FIMPs, type of FIMPs, and pipeline parallelism between the producing and consuming FIMPs. SYLVA also introduces timing and interface model of FIMPs to enable reuse and automatic generation of Global Interconnect and Control (GLIC) to glue the FIMPs together into a working system. SYLVA has been evaluated by applying it to several real and synthetic DSP applications and the experimental results are analyzed for the design space exploration, the GLIC synthesis, the code generation, and the CGRA floorplanning features. The conclusion from the experimental results is that by exploring the multi-dimensional design space in terms of pre-characterized FIMPs, SYLVA explores a richer design space and does it more effectively compared to the existing High-Level Synthesis (HLS) tools to improve both engineering and computational efficiency.

Sammanfattning

Denna avhandling presenterar ett nytt syntes-ramverk på systemnivå som kallas System Level Synthesis Architectural Framework (SYLVA), som syntetiserar Digital Signal Processing (DSP) delsystem som modelleras av synkron data till implementationer i hårdvara som Application Specific Integrated Circuit (ASIC), Field-Programmable Gate Array (FPGA) eller Coarse-Grained Reconfigurable Architecture (CGRA). SYLVA syntetiserar i form av förkarakteriserade Function Implementations (FIMPs). Denna metod utforskar designutrymmet i tre dimensioner, antal FIMPs, typ av FIMPs och pipeline parallellism emellan de producerande och konsumerande FIMPs. SYLVA introducerar också en timing-och gränssnittsmodell av FIMPs för att möjliggöra återanvändning och automatisk generering av Global Interconnect and Control (GLIC) att kunna klistra samman FIMPs till ett fungerande system. SYLVA har utvärderats genom att applicera det i verkliga samt syntetiska DSP applikationer och experimentella resultat analyseras för design space exploration, GLIC-syntesen, kodgenereringen och CGRA floorplanning funktioner. Slutsatsen från de experimentella resultaten är att genom att utforska den flerdimensionella design space när det gäller förkarakteriserade FIMPs, utforskar SYLVA en bredare design space och gör det mer effektivare gentemot de befintliga högnivåsyntes (HLS) verktyg med målet att förbättra större teknik- och beräkningseffektivitet.

Acknowledgment

First of all, I would like to express my special thanks of gratitude to my supervisor Prof. Ahmed Hemani and scientific reviewer Dr. Kolin Paul. Prof. Ahmed Hemani gave me the golden opportunity to do this wonderful project on system-level hardware synthesis. He also helped me in doing a lot of research and I came to know about so many new things. I am really thankful to them. Secondly I would also like to thank my parents, Prof. Zheying Li and Dr. Xueling Liu, who supported me a lot both financially and mentally during my PhD research. Without their continuous support, I cannot even start this project. I also would like to thank my aunt Dr. Li Li and her husband Peter Jianhua Guo for their help during these years in Sweden. They have supported me a lot and have acted as my parents in Sweden.

Next, I would like to express my thanks to Prof. Shashi Kumar and my colleagues Dr. Zhonghai Lu, Prof. Christian Schulte, Prof. Elena Dubrova, Dr. Qiang Chen, Dr. Ingo Sander, Dr. Johnny Öberg, Alina Munteanu, Dr. Syed Jafri, Dr. Hassan Sohofi, Nasim Farahini, Dr. Sha Tao for their kindly help during my PhD research. Then I would like to thank all my friends: Yuan Yao, Yuecheng Yang, Hao Yan, Liyi Meng, Xiaoyu Lan, Ming Liu, Shaoteng Liu, Pei Liu, Xueqian Zhao, Jue Shen, Chuanying Zhai, Qiansu Wan and Dr. Jun Zhu for the happiness we have shared together.

Finally, I would like to express my special thank to my beloved wife Dr. Ding Ding for her understanding, patience and support. I am very lucky to have her came into my life in the middle of this project.

Shuo Li
2016-01-12
Stockholm, Sweden

Contents

List of Figures

List of Tables

List of Abbreviations

128-QAM	128 Quadrature Amplitude Modulation
16 QAM	16-Quadrature Amplitude Modulation
256-QAM	256 Quadrature Amplitude Modulation
64-QAM	64 Quadrature Amplitude Modulation
ABB	AIR Building Block
ADL	Architecture Description Language
AGU	Address Generation Unit
AIR	Abstract Intermediate Representation
ALP	Arithmetic-Level Parallelism
ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
BLP	Buffer-Level Parallelism
BPSK	Binary Phase-Shift Keying
CCK	Complementary Code Keying
CDFG	Control Data Flow Graph
CGRA	Coarse-Grained Reconfigurable Architecture
CIC	Cascaded Integrator-Comb
CP	Constraint Programming
CRASIC	Customization of Coarse Grain Reconfigurable
CSOP	Constraint Satisfaction Optimization Problem
CSP	Constraint Satisfaction Problem
ConTile	Configuration Tile
DCT	Discrete Cosine Transform
DFG	Data Flow Graph
DiMARCH	Distributed Memory Architecture
DPU	DataPath Unit
DRRA	Dynamically Reconfigurable Resource Array
DSE	Design Space Exploration
DSP	Digital Signal Processing
EDA	Electronic Design Automation

FFT	Fast Fourier Transform
FIFO	First In First Out
FIMP	Function Implementation
FIR	Finite Impulse Response
FLP	Function-Level Parallelism
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
FSMD	FSM with Datapath
GCC	GNU Compiler Collection
GLIC	Global Interconnect and Control
HLS	High-Level Synthesis
HSDF	Homogeneous SDF
ILP	Integer Linear Programming
IP	Intellectual Property
KPN	Kahn Process Network
LSA	Layered Spiral Algorithm
LTE	Long Term Evolution
MAC	Multiplication-Accumulation
MILP	Mixed Integer Linear Programming
MIMO	Multi-Input Multi-Output
MPEG	Moving Picture Experts Group
MPSoC	Multi-Processor System-on-Chip
MUX	Multiplexer
NoC	Network-on-Chip
OVSF	Orthogonal Variable Spreading Factor
QoR	Quality of Result
QPSK	Quadrature Phase-Shift Keying
RTL	Register-Transfer Level
SA	Simulated Annealing
SADF	Scenario-Aware Data Flow
SC-FDMA	Single Carrier Frequency Diversity Multiple Access
SDF	Synchronous Data Flow
SLS	System-Level Synthesis
SRAM	Static Random-Access Memory
STile	SRAM Tile
SYLVA	System-Level Architectural Synthesis Framework
VLC	Variable Length Coding
VLSI	Very-Large-Scale Integration
WLAN	Wireless Local Area Network

List of Publications

- Entire System-Level Architectural Synthesis Framework (SYLVA) flow

  Shuo Li, Nasim Farahini, Ahmed Hemani, Kathrin Rosvall, Ingo Sander, *System Level Synthesis of Hardware for DSP Applications Using Pre-Characterized Function Implementations*, the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013

- Design Space Exploration (chapter 5)

  1. Shuo Li, Ahmed Hemani, *Three-Dimensional Design Space Exploration for System Level Synthesis*, the 17th Euromicro Conference on Digital System Design (DSD), 2014
  2. Shuo Li, Ahmed Hemani, *Accurate and Efficient Three Level Design Space Exploration Based on Constraints Satisfaction Optimization Problem Solver*, the 22nd Annual IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2014
  3. Fahimeh Jafari, Shuo Li, Ahmed Hemani, *Optimal Selection of Function Implementation in a Hierarchical Configware Synthesis Method for a Coarse Grain Reconfigurable Architecture*, the 14th Euromicro Conference on Digital System Design (DSD), 2011

- Global interconnect and control synthesis (chapter 6)

  1. Shuo Li, Ahmed Hemani, *Global Interconnect and Control Synthesis in System Level Architectural Synthesis Framework*, the 16th Euromicro Conference on Digital System Design (DSD), 2013
  2. Shuo Li, Nasim Farahini, Ahmed Hemani, *Global Control and Storage Synthesis for a System Level Synthesis Approach*, the 21st Annual IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2013

- Code Generation (chapter 7)

  1. Shuo Li, Ahmed Hemani, *Memory Allocation and Optimization in System-Level Architectural Synthesis*, the 8th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2013
  2. Shuo Li, Jamshaid Malik, Shaoteng Liu and Ahmed Hemani, *A Code Generation Method For System-Level Synthesis*, the ACM International Workshop on Manycore Embedded Systems, 2013
  3. Shuo Li, Guo Chen, Ahmed Hemani, *A Code Reuse Method for Many-Core Coarse-Grained Reconfigurable Architecture Function Library Development*, The 13th International Symposium on Integrated Circuits (ISIC), 2011

- Other publications during SYLVA research

  1. Nasim Farahini, Ahmed Hemani, Hassan Sohofi, Shuo Li, *Physical Design Aware System Level Synthesis of Hardware*, SAMOS XV, 2015
  2. Nasim Farahini, Shuo Li, Muhamamd Adeel Tajammul, Guo Chen, Muhammad Ali Shami, Ahmed Hemani, *39.9 GOPs/Watt Multi-Mode CGRA Accelerator for a Multi-Standard Basestation*, the IEEE International Symposium on Circuits and Systems (ISCAS), 2013
  3. Shuo Li, Fahimeh Jafari, Ahmed Hemani, Shashi Kumar, *Layered Spiral Algorithm for Memory-Aware Mapping and Scheduling on Network-on-Chip*, the 28th NORCHIP conference, 2010

Chapter 1

Introduction

System-Level Synthesis (SLS) is an evolutionary next step from High-Level Synthesis (HLS). The objective of SLS is to provide a solution to close the design productivity gap as much as possible by automatic synthesizing from high levels of abstraction to improve design productivity while achieving high computational and silicon efficiency. The design productivity gap, which is illustrated by Figure 1.1, has been identified as a key challenge by the semiconductor industry (1997 by SEMATECH [1, 2] and 1999 by ITRS [3, 4]).

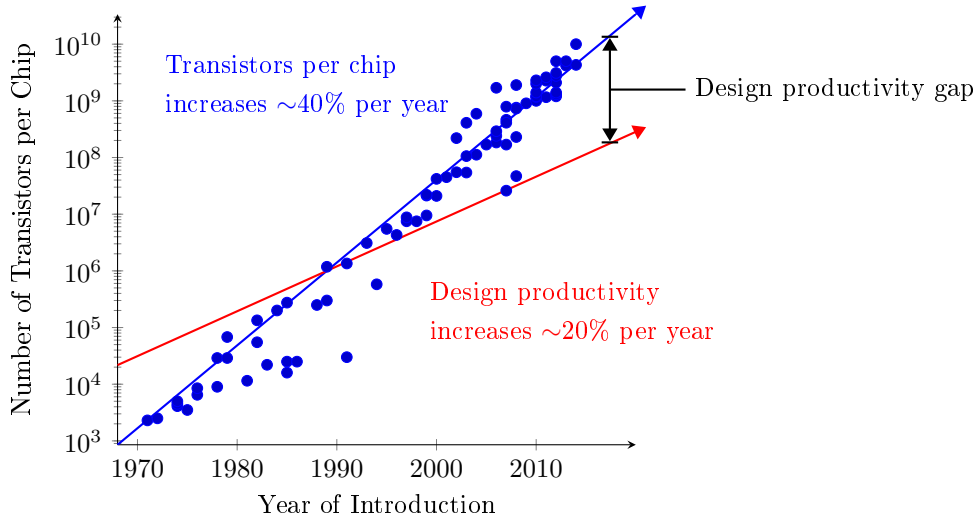


Figure 1.1: 1971 to 2014 Transistor Count Chart

1.1 Brief History of EDA

To cope with the increasing complexity, Electronic Design Automation (EDA) has relied on increasing the abstraction at which functionality is specified and automating the synthesis of design data from it that can be used to create the lithographic masks for manufacturing the Very-Large-Scale Integration (VLSI) circuit.

The history of EDA can be summarized by the famous Gajski-Kuhn's Y-Chart [5] shown in Figure 1.2. The hardware implementation is in the form of transistor layout that is in the center of the chart. The objective of EDA tools is to help the designers to synthesize their design to the corresponding transistor layout. EDA has a long and rich history. The following is a synopsis of it to place the research presented in this thesis in its historical context, starting from layout-level design to High-Level Synthesis (HLS).

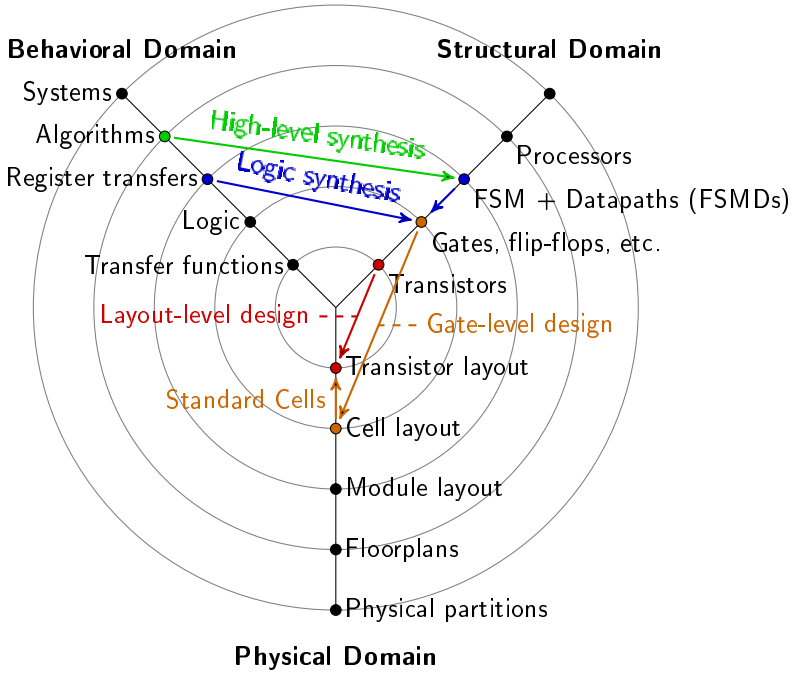


Figure 1.2: Gajski-Kuhn's Y-Chart

The first generation of EDA tools used stick diagrams [6] as the abstraction. The stick diagram was a symbolic cartoon of the actual layout. It did not have information about the actual location of the components, the length and the width of the wires, the size of the tub, etc. The first generation of EDA tools refined the stick diagrams into actual layout that respected design rules for a certain manufacturing process. These tools evolved into a structured VLSI design methodology, advanced by Carver and Lynn [7] that relied on regularity of digital design structures in terms of bit slices or matrices of transistors. Micro-architectural blocks were composed by replicating bit slices in a single dimension. Storage units (e.g. registers), and control units (e.g. programmable logic arrays), were also built on the regularity of bit-level transistor pattern in two dimensions, in the form of a matrix. Larger systems were composed by abutting these blocks. This methodology was succeeded by silicon compilers that stored the structured layout pattern of micro-architectural blocks in a parametric form, allowing automatic generation of, for instance adders of particular width. This design methodology required the designer to refine the design manually in a schematic form, where all elements of the schematic had a corresponding macro generator. The designs were restricted to be in the structural form and in terms of functions for which macro generators existed.

A more significant breakthrough came with the introduction of standard cells [8] layout scheme. In these scheme, elementary gates and commonly used combinations of them like AND, OR, INVERT, etc., flip-flops, and latches of different types were designed as standard cells – transistor-level design with the same height or pitch. A design was expressed in terms of instances of these standard cells that were organized in a row. Since the standard cells had the same height or pitch, it became possible to feed them by common power supply and ground rails. Rows of standard cells were separated by sufficient space to accommodate wires connecting the standard cell instances. This layout scheme simplified the physical design and enabled automatic placement and routing of gates, flip-flops, and latches implemented as standard cells. Gates and flip-flops being smaller and thereby more general building blocks allowed the designer a greater degree of freedom in expressing the intended design. This is similar to being able to draw any arbitrary drawing with greater freedom with small pixels compared to large rectangles. This in turn enabled the design to be expressed in a behavioral form, optimize the design and map it to standard cells available in the library. Starting from a behavioral representation had an advantage compared to structural schematic form because it was more natural and closer to the math and programming notation that is closer to natural language compared to the schematics. It also allowed a more compact textual representation compared to large multi-page schematics that becomes unwieldy to draw and comprehend. By automating most aspects of physical design, the synthesis approach to hardware design took root and has ever since been the mainstay of hardware design and came to be known as Register-Transfer Level (RTL)/logic synthesis and physical syntheses. An incremental improvement in terms of starting from RTL was developed that further enhanced the efficiency of this flow.

After RTL/logic synthesis, EDA researchers attempted moving to a clock free algorithmic level of abstraction as the starting point and attempted automatic synthesis of hardware from it in terms of generating an Finite State Machine (FSM) to implement the control flow and a datapath to implement the computation and storage. This synthesis flow has come to be known as HLS. While the research in HLS started already in early 80s, the first commercial product were introduced in early 90s. However, HLS has taken a long time to mature and even after the two decades of first commercial products, the number of hardware design done using HLS is insignificant [9]. A key problem with HLS is that it uses soft micro-architectural building blocks like arithmetic units, multiplexors, registers etc. When micro-architectural building blocks are soft, even if their composition in terms of standard cells is known, the position of standard cells with respect to each other is not fixed. In contrast, standard cells are hard building blocks because not only the transistors and their dimensions composing a standard cell is known, their physical location and wires connecting them are also known. This intra-micro-architectural building block softness degrades the ability of HLS tool to know what will be the final cost metric - area, latency and energy.

In addition, HLS tools also suffer from inter-building block softness, i.e., the position of micro-architectural building blocks with respect to each other is also unknown. This degrades the ability to predict cost metrics even more compared to intra-micro-architectural building blocks. Since the final synthesis results are in terms of standard cells, the design space increases exponentially with the problem size. These factors, the intra and inter building block softness and the large design space, contribute to making HLS an extremely hard optimization problem that takes significant time to reach physical design. Even after overcoming these challenges, HLS in principle synthesizes only one algorithm at a time. An algorithm in today's context is a trivially small piece of functionality compared to the complex sub-systems like modems and codecs that need to be mapped to hardware. Many HLS tools have evolved to superficially handle multiple functions: But firstly, these tools do not optimize across function boundaries making the optimization local. Secondly, these tools require a manual RTL interface to be refined between individual functions, i.e., the tools relies on the designer to design logic at RTL to glue individual functions. These limitations of HLS prevents it from being an effective and efficient approach to address the challenge posed by ITRS: 1000× improvement in performance with 120% increase in power budget and no increase in the design team size to cope with a 10× increase in design complexity.

1.2 Introduction to System-Level Synthesis

System-Level Synthesis (SLS) is an evolutionary next step from High-Level Synthesis (HLS) as shown in Figure 1.3 and they differ in the abstraction as illustrated in Figure 1.4. SLS is meant to complement the existing system-on-chip design flows by making the functional hardware design more efficient and thus encourage a larger portion of functionality to be implemented as hardware and SLS only handles statically scheduled functionalities.

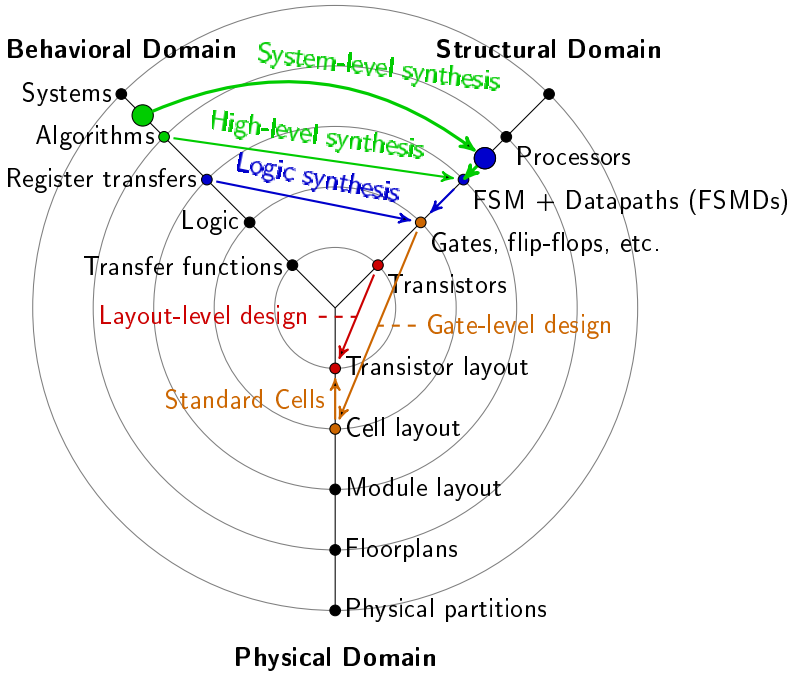


Figure 1.3: System-Level Synthesis in Gajski-Kuhn's Y-Chart

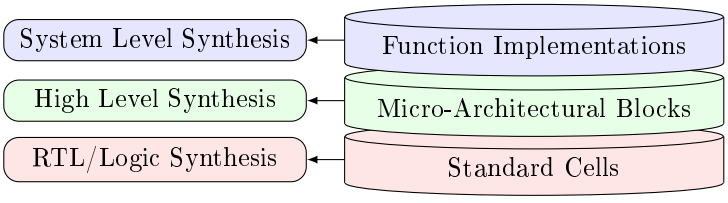


Figure 1.4: RTL/Logic Synthesis to System-Level Synthesis

HLS synthesizes algorithm that is expressed by arithmetic, logic, and control operations to implementation in terms of corresponding micro-architecture level building blocks like arithmetic and logic units, registers, register files, SRAM banks, and state machines to implement the control flow. Correspondingly, SLS defines a system-level as a hierarchy of algorithms glued together with control constructs. An algorithm at system-level is a unit computational operation in a similar sense that an arithmetic operations is at algorithmic level. Further, SLS synthesizes in terms of Function Implementations (FIMPs) that are FSM with Datapath (FSMD). This thesis uses the term *function* because in many programming languages, algorithms are modeled by a language construct *function*. FIMPs can be primitive like sum, product, max, min etc. to more complex like convolution and Fast Fourier Transform (FFT). SLS has access to a library of FIMPs similar to HLS tools having access to a library of micro-architecture level building blocks. Multiple FIMPs exist for the same algorithm, differing in architecture and parallelism and thereby varying in their cost metrics - area, latency and energy. For instance, FFTs can be implemented in radix-2, radix-4 architecture and with different numbers of butterflies. This is also similar to HLS tools having multiple adders and multiplier implementations with varying cost metrics. More information about FIMP can be found in section 4.3.

HLS optimizes by selecting type and number of resources, typically arithmetic units and schedules them to minimize area, latency, or both. HLS also infers the registers required for storing data between different states and interconnect required to transfer data among the arithmetic units. Registers and interconnect are also required for reusing resources. Finally HLS synthesizes an FSM to implement the control flow by controlling the datapath in terms of selecting the Multiplexer (MUX) inputs and modes of arithmetic units depending on the state.

Correspondingly, SLS optimizes in terms of number and types of FIMPs and schedules them to minimize area, energy, and latency. The specific SLS tool proposed in this thesis is called System-Level Architectural Synthesis Framework (SYLVA) and it optimizes area and energy while meeting the sampling rate and total latency constraints. It also infers interconnect to glue the FIMPs together and transfer data between them. Finally, it generates global control to orchestrate the execution of FIMPs. Note that each FIMP is an FSMD and the Finite State Machine (FSM) in it serves as the local control. Unlike the HLS tools that superficially supports synthesis of a hierarchy of algorithms, the proposed SLS does perform a global optimization when it selects the number and type of FIMPs; it does not optimize each algorithm individually. The cost function is a global cost function that includes the cost of all the FIMPs involved. Secondly, the global interconnect and control that glues the FIMPs together is automatically inferred and synthesized and is not manually refined as is required by the HLS tools.

It is obvious that dedicated hardware is more energy efficient compared to general software. The large engineering cost associated with hardware design is partially responsible for designs implementing the bulk of functionality in software and only implementing performance and power critical functionalities in hardware [10]. As the move to software implementation style is also partially motivated by the need to have extensible product categories, the software and hardware design styles for implementing functionalities will co-exist. However, by making it easier to implement functionalities in hardware, SLS enables a greater percentage of functionality to be realized as hardware, thereby improving the computational efficiency. SLS also improves the design productivity as described below.

1. By using FIMPs as the building blocks, SLS enables significant increase in automatic reuse compared to what is achieved today by HLS that reuses micro-architectural blocks. Note that the reuse of complex Intellectual Properties (IPs) like processors, bus/memory controller etc. is manual and not automatic.

2. By enabling automatic synthesis of significantly more complex functionality, compared to what is achieved today by HLS tools, the SLS tool will lower the engineering cost of designing complex hardware sub-systems like modems and codecs. Note that SLS is not positioned to implement infrastructural hardware like processors, memory/bus/cache controller etc.

3. By virtue of correct by construction, SLS eliminates the most significant engineering cost component in design of hardware, the functional verification.

4. Since the FIMPs are characterized with post-layout sign-off quality data, the SLS has the advantage of knowing the cost metrics accurately and at the same time logic and physical synthesis of the FIMPs is factored out as a onetime engineering effort, thus speeding up the synthesis process.

5. The Design Space Exploration (DSE) in SLS is in terms of FIMPs that are predesigned and characterized and not in terms of finer grain micro-architectural blocks. This makes the DSE more effective and efficient.

1.3 Thesis Contributions

The main contribution of this thesis is System-Level Architectural Synthesis Framework (SYLVA). It is a System-Level Synthesis (SLS) tool that synthesizes system-level specifications with constraints and optimization goals to an Register-Transfer Level (RTL) model for Application-Specific Integrated Circuit (ASIC) and Field-Programmable Gate Array (FPGA), or a configware (configuration and software) for Coarse-Grained Reconfigurable Architectures (CGRAs) based on the guide from designers and the provided Function Implementation (FIMP) library for all implementation style candidates. In the current stage, SYLVA targets Digital Signal Processing (DSP) applications that can be statically scheduled and typical examples of functions are Fast Fourier Transform (FFT), Finite Impulse Response (FIR) filters, and Viterbi etc. Salient points about SYLVA and the key contributions of this thesis consist of as follows:

1. The first one is the use of FIMPs for synthesizing system-level specifications. The detailed information of FIMP can be found in section 4.3. A FIMP implies physical/layout level implementation and characterization with post-layout data. Use of FIMPs as reusable object has four major advantages.

    a) Using FIMPs reduces the search space and thus reduces the size of the optimization problem.

    b) FIMPs can be designed by special arrays with greater engineering effort to improve the design quality and the engineering cost amortized by multiple end users.

    c) Pre-designed FIMPs enable system-level estimation of energy and performance.

    d) Using pre-verified FIMPs in system-level synthesis makes verifying the synthesis results much simpler than traditional approaches.

2. A key challenge in the automatic reuse by synthesis tools of complex and coarse grain objects like FIMPs is generalizing their interface, timing and cost models. For automatic reuse, SYLVA defines a FIMP using three views: interface view (section 4.3.2), execution view (section 4.3.2), and implementation view (section 4.3.2).

    a) Interface view of a FIMP provides the designed function of this FIMP, the designed implementation style of this FIMP (one function may have multiple FIMPs differing in latency, energy, area, and also implementation style), the cost and performance of this FIMP, and its data and control interface to other FIMPs at high-level (e.g. number of data tokens transmitted).

    b) Execution view of a FIMP provides the execution energy consumption, area usage, and the execution timing model of this FIMP.

c) Implementation view of a FIMP provides the hardware description of this FIMP (e.g. RTL model, CGRA configware), the interface for data communication at low-level (e.g. the actual width of a data input port), and the placement and routing details (only for CGRA).

3. SYLVA explores the design space in three dimensions (detailed elaborations can be found in chapter 5).

   a) The number of deployed FIMPs. This dimension reflects the degree of parallelism in function level (section 5.1).
   b) The types of deployed FIMPs (latency, energy, and area for each FIMP instance). This dimension reflects the degree of parallelism in arithmetic level (section 5.2).
   c) The data communication scheme. This dimension reflects the degree of parallelism in buffer level (section 5.3).

   High-Level Synthesis (HLS) tools also explore a corresponding design space, albeit at a finer level of granularity: number and type (more parallel or serial) of arithmetic units.

4. SYLVA automatically generates the necessary glue logic to combine the FIMPs and buffers into a working system that implements the system model (more information can be found in chapter 6). The generated glue logic consists of following components:

   a) A hierarchy of Finite State Machines (FSMs) to orchestrate the execution of FIMPs.
   b) Selection of input and output sources for individual FIMPs.
   c) Multiplexers for transferring data among the FIMPs.
   d) A set of output buffers for each FIMPs.

5. SYLVA is also capable of performing floorplanning for CGRAs (section 7.6) to minimize the total energy consumption, latency, and/or sample interval of the final system implementation.

1.3.1 SYLVA Design Flow

SYLVA design flow is illustrated by Figure 1.5 and it will be elaborated in four chapters: system modeling (chapter 4), design space exploration (chapter 5), global interconnect and control synthesis (chapter 6), and code generation (chapter 7). More information can be found in section 1.4 and individual chapters.

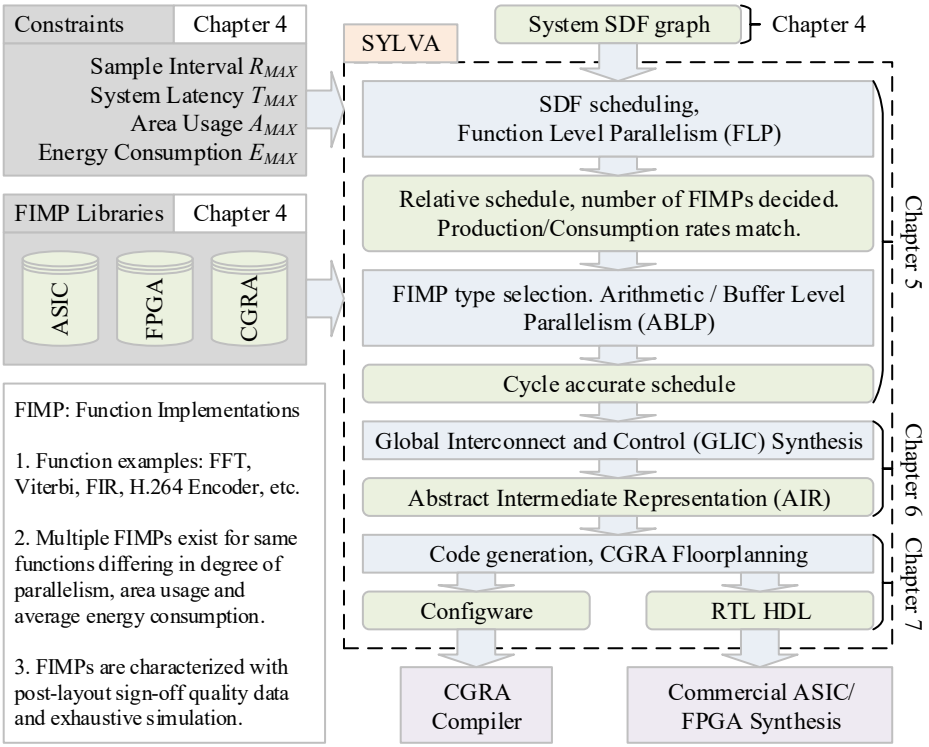


Figure 1.5: SYLVA Design Flow

1.3.2 Publications

The publications of this thesis are categorized based on the related SYLVA design low steps and listed below.

- Entire SYLVA flow

  Shuo Li, Nasim Farahini, Ahmed Hemani, Kathrin Rosvall, Ingo Sander, *System Level Synthesis of Hardware for DSP Applications Using Pre-Characterized Function Implementations*, the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013

- Design Space Exploration (chapter 5)

  - Generalizing the FIMP execution view.
  - Automatically creating the Constraint Satisfaction Optimization Problem (CSOP) for arithmetic-level design space exploration.
  - Automatically Synchronous Data Flow (SDF) schedule generation for function-level design space exploration.
  - Automatically creating the CSOP for both level design space exploration.

  Related publications:

  1. Shuo Li, Ahmed Hemani, *Three-Dimensional Design Space Exploration for System Level Synthesis*, the 17th Euromicro Conference on Digital System Design (DSD), 2014

     This paper elaborates a three-dimensional design space exploration. The dimensions in that paper are schedule level, function level and arithmetic/buffer level. In addition to the function and arithmetic/buffer level, the schedule level is proposed for generating the load balanced relative schedule. The generated CSOP for design space exploration is still two-dimensional.

  2. Shuo Li, Ahmed Hemani, *Accurate and Efficient Three Level Design Space Exploration Based on Constraints Satisfaction Optimization Problem Solver*, the 22nd Annual IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2014

     This paper elaborates the three-dimensional design space exploration, while the schedule level is considered as a separate level.

  3. Fahimeh Jafari, Shuo Li, Ahmed Hemani, *Optimal Selection of Function Implementation in a Hierarchical Configware Synthesis Method for a Coarse Grain Reconfigurable Architecture*, the 14th Euromicro Conference on Digital System Design (DSD), 2011

This paper proposes an algorithm for optimal selection of FIMPs for CGRA. The contribution from the author of this thesis includes proposing the selection algorithm, implementing it in C#, and preparing experiment test cases.

- Global interconnect and control synthesis (chapter 6)
  - Proposing the data structure of the intermediate representation.
  - Automatically creating FSMs, buffers and interconnect based on the cycle accurate schedule.
  - Proposing the algorithm for refining the cycle accurate schedule based on mapped FIMPs, buffers and interconnect.

  Related publications:

  1. Shuo Li, Ahmed Hemani, *Global Interconnect and Control Synthesis in System Level Architectural Synthesis Framework*, the 16th Euromicro Conference on Digital System Design (DSD), 2013
     This paper describes the interconnect and control generation step of SYLVA.
  2. Shuo Li, Nasim Farahini, Ahmed Hemani, *Global Control and Storage Synthesis for a System Level Synthesis Approach*, the 21st Annual IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2013
     This paper gives a brief overview of the interconnect and control generation step of SYLVA. The contribution from the author of this thesis includes algorithm proposal, implementation and experiments.

- Code Generation (chapter 7)
  - Proposing the algorithm for automatically merging buffer instances.
  - Proposing the algorithm for automatically merging FIMP instances.
  - Proposing a model for CGRA structure.
  - Proposing a CSOP solving based CGRA floorplanning algorithm.
  - Proposing a heuristic CGRA floorplanning algorithm.

  Related publications:

  1. Shuo Li, Ahmed Hemani, *Memory Allocation and Optimization in System-Level Architectural Synthesis*, the 8th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2013
     This paper proposes the algorithm used in the memory allocation phase in the code generation step.

2. Shuo Li, Jamshaid Malik, Shaoteng Liu and Ahmed Hemani, *A Code Generation Method For System-Level Synthesis*, the ACM International Workshop on Manycore Embedded Systems, 2013

   This paper describes the code generation step in detail. The contribution from the author of this thesis includes algorithm proposal and implementation.

3. Shuo Li, Guo Chen, Ahmed Hemani, *A Code Reuse Method for Many-Core Coarse-Grained Reconfigurable Architecture Function Library Development*, The 13th International Symposium on Integrated Circuits (ISIC), 2011

   This paper describes the implementation view of FIMP.

- Other publications during SYLVA research are listed below.

1. Nasim Farahini, Ahmed Hemani, Hassan Sohofi, Shuo Li, *Physical Design Aware System Level Synthesis of Hardware*, SAMOS XV, 2015

   This paper We propose a grid-based regular physical design platform composed of large grain hardened building blocks called SiLago blocks. The author in this thesis contributed a part of the experiments.

2. Nasim Farahini, Shuo Li, Muhamamd Adeel Tajammul, Guo Chen, Muhammad Ali Shami, Ahmed Hemani, *39.9 GOPs/Watt Multi-Mode CGRA Accelerator for a Multi-Standard Basestation*, the IEEE International Symposium on Circuits and Systems (ISCAS), 2013

   This paper elaborates an application of CGRA on multi-standard basestation. The contribution from the author of this thesis includes CGRA platform verification and modification, development of CGRA application development tools, and development of one application (correlation pool).

3. Shuo Li, Fahimeh Jafari, Ahmed Hemani, Shashi Kumar, *Layered Spiral Algorithm for Memory-Aware Mapping and Scheduling on Network-on-Chip*, the 28th NORCHIP conference, 2010

   This paper proposes a heuristic algorithm for memory-aware mapping and scheduling applications on Network-on-Chip (NoC). The contribution from the author of this thesis includes algorithm proposal, algorithm implementation, and preparing experiment test cases.

1.4 Thesis Layout

- Chapter 2 provides some required preliminaries for understanding the proposed System-Level Synthesis (SLS) methodology and the corresponding System-Level Architectural Synthesis Framework (SYLVA) tool.
  - Section 2.1 gives the detailed definition and properties of Synchronous Data Flow (SDF) graph.
  - section 2.2 introduces Dynamically Reconfigurable Resource Array (DRRA), a Coarse-Grained Reconfigurable Architecture (CGRA) under development in KTH including the hardware specification and configware formulation.
  - section 2.3 gives a brief introduction to Constraint Programming (CP), including the problem modeling using the CP concepts and a brief exploration of the existing CP solvers.
- Chapter 3 introduces other hardware synthesis approaches, software synthesis, and parallel compilation approaches, and compares them with the corresponding aspects of SLS and SYLVA.
  - Section 3.1 compares SLS and High-Level Synthesis (HLS).
  - Section 3.2 is about the related work for hardware synthesis to CGRA.
  - Section 3.3 is about the related work on Global Interconnect and Control (GLIC) generation.
  - Section 3.4 gives some related work on CGRA programming.
- Chapter 4 describes the proposed system modeling method for modeling Digital Signal Processing (DSP) sub-systems using an SDF graph with constraints and optimization objectives.
  - Section 4.1 elaborates the SDF used in SYLVA. It is slightly different from the original definition in [11].
  - Section 4.2 introduces the Homogeneous SDF (HSDF) graph used in SYLVA.
  - Section 4.3 defines Function Implementation (FIMP).
  - Section 4.4 gives the detailed procedure of system modeling including the method to create SDF actors (section 4.4.1) and the method to create SDF edges (section 4.4.2).
  - Section 4.5 gives an detailed example, Long Term Evolution (LTE) uplink transmitter, of system modeling in SYLVA.
  - Section 4.6 summarizes chapter 4.

- In chapter 5, the proposed three-dimensional Design Space Exploration (DSE) in SYLVA is elaborated in detail, including the DSE problem formulation, the size of the design space, and the CP solver based DSE algorithm.
  - Section 5.1 defines Function-Level Parallelism (FLP).
  - Section 5.2 defines Arithmetic-Level Parallelism (ALP).
  - Section 5.3 defines Buffer-Level Parallelism (BLP).
  - Section 5.4 quantifies the design space.
  - Section 5.5 gives the method of DSE.
  - Section 5.6 summarizes chapter 5.
- Chapter 6 elaborates the proposed algorithms for automatically generation and optimization of interconnect and control logics in SYLVA, as well as the definition of the proposed intermediate representation.
  - Section 6.1 defines Abstract Intermediate Representation (AIR).
  - Section 6.2 defines AIR Building Block (ABB).
  - Section 6.3 elaborates the data communications among ABBs.
  - Section 6.5 summarizes chapter 6.
- Chapter 7 elaborates the code generator in SYLVA. Currently, it is capable of producing Register-Transfer Level (RTL) VHDL codes for Application-Specific Integrated Circuit (ASIC) and Field-Programmable Gate Array (FPGA), and DRRA configware as well as performing architecture independent and architecture dependent optimization on FIMP, interconnect and buffers. All the proposed algorithms and techniques for code generation are elaborated in detail.
  - Section 7.1 formulates the code generation problem in SYLVA.
  - Section 7.3 summarizes the architecture independent optimizations during code generation in SYLVA.
  - Section 7.4 gives the resource allocation method used in the code generation in SYLVA.
  - Section 7.5 summarizes the architecture dependent optimizations during code generation in SYLVA.
  - Section 7.6 elaborates the CGRA floorplanning methods, one CP solver based method and one heuristic method, used by SYLVA.
  - Section 7.7 summarizes chapter 7.

- In chapter 8, the efficacy and efficiency of SYLVA for ASIC and FPGA are evaluated by synthesizing a set of benchmark systems using SYLVA and comparing the synthesis results of SYLVA to the synthesis results of two commercial HLS tools. The efficacy and efficiency of the CGRA floorplanning is evaluated by comparing the results obtained by the proposed two floorplanning algorithms and the result obtained by an exhaustive search.
  - Section 8.2 gives the experimental procedure.
  - Section 8.1 lists the applications to be used for evaluating SYLVA.
  - Section 8.3 gives the synthesis results and analyzes the results.
- Finally, chapter 9 gives the conclusion (section 9.1) of this thesis and discusses the possible future work in SLS and SYLVA research (section 9.2).

Chapter 2

Preliminaries

This chapter introduces the preliminary concepts that are necessary for describing System-Level Synthesis (SLS) and System-Level Architectural Synthesis Framework (SYLVA) design flow in later chapters.

- Section 2.1 addresses the Synchronous Data Flow (SDF) graph [11]. SYLVA uses SDF graph since it is widely used in system-level design flows for modeling and analyzing real-time streaming applications [12] [13] that matches the application domain of SYLVA. In SYLVA design flow, the SDF graphs along with user-defined constraints and optimization objectives (such as performance and cost metrics) serve as the top-level system model.
- Section 2.2 introduces Dynamically Reconfigurable Resource Array (DRRA) [14], which is a Coarse-Grained Reconfigurable Architecture (CGRA) [15] under development in KTH. The CGRA code generation of SYLVA currently only supports DRRA. Although, there are a large number of CGRAs, DRRA is chosen since it is proven to be an energy efficient platform for hosting Digital Signal Processing (DSP) sub-systems [16]. This section also coves the architecture, the development tools, and the model of DRRA that are used in SYLVA.
- Section 2.3 covers Constraint Programming (CP) [17] including the elaboration of Constraint Satisfaction Problem (CSP) [18]. The motivation of using CP is that CP is effective on solving resource allocation [19], areas of planning [20], and scheduling [21] problems [22]

2.1 Synchronous Data Flow Graph

Synchronous Data Flow (SDF) graphs are a special kind of Data Flow Graph (DFG). So before introducing the SDF graph, we will have a brief introduction to the DFG.

2.1.1 Introduction to Data Flow Graph

In computer science, a DFG is a graphical representation of the *flow* of data through a data processing system, modeling its process aspects. A DFG is often used to create an overview of the system (which can later be elaborated) and used for the visualization of data processing. The author of the first DFG article [23] described that a DFG is to serve as an intermediate-level language for high-level languages and to serve as a machine language for parallel machines. The article [24] gives the formal definition of a DFG (Eq. 2.1) that a DFG G is a bipartite labeled graph where the two types of nodes are called actors (A) and links (L).

$$G = ((A \cup L), E) \tag{2.1}$$

- $A = \{a_1, a_2, ..., a_n\}$ is the set of actors.
- $L = \{l_1, l_2, ..., l_m\}$ is the set of links.
- $E \subseteq (A \times L) \cap (L \times A)$ is the set of edges.

Actors represent processes and links receive data from a single actor and transmit values to one or more actors by way of edges. Edges are the communication channels representing the data dependencies among the actors. Figure 2.1 shows an example DFG with three processes (p_0, p_1, and p_2), one data source (*src*), one data sink (*snk*), and one link (the black dot). Actors *src* and *snk* are for communicating with the external world outside the modeled system.

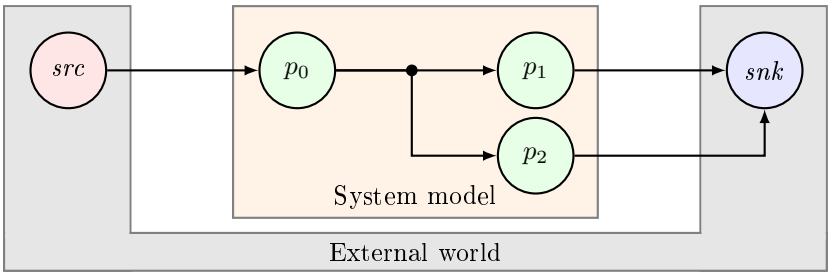


Figure 2.1: Data Flow Graph Example

2.1.2 Introduction to SDF Graph

The definition of SDF graph can be found in the original SDF publication [11]:

> Synchronous Data Flow (SDF) graph is a special case of data flow (either atomic or large grain) in which the number of data samples produced or consumed by each function on each invocation is specified a priori.

The authors in [11] also claimed that SDF graph has the following properties.

1. The actors of an SDF graph can be scheduled statically, since the producing and consuming data rates of them are known in compile time.
2. An SDF graph is capable of modeling concurrent processes. Actors that have no data dependency among each other can be invoked in parallel.
3. An SDF graph captures data dependencies among processes (cyclic and acyclic) via directed edges. Pipeline stages can be easily extracted from an SDF graph.

According to these properties, the implementation of algorithms that expressed as SDF graphs is guaranteed to use finite memory and execute all processes in a finite time. An SDF graph can be executed in a periodic fashion without requiring additional resources during its execution. The periodic execution is well-suited to Digital Signal Processing (DSP) sub-systems and communications systems, which often process an endless supply of data. There are some efficient techniques for computing the throughput and buffer size of SDF graphs [25] [26], making it analyzable. Therefore, SDF graph is widely used in system-level design flows for modeling and analyzing real-time streaming applications [12] [13]. One simple example is a digital Finite Impulse Response (FIR) filter whose sample frequency (samples per second, e.g. 44.1 kHz) and sample size (bits per sample, e.g. 16-bit wide integer number) are constant. SDF can be used to model signal processing in complex standards like Wireless Local Area Network (WLAN), Long Term Evolution (LTE), Moving Picture Experts Group (MPEG) codecs, etc. [27].

Synchronous Data Flow Graph

Since SDF is a DFG, an SDF graph is also defined as a graph (Eq. 2.1) comprising a set of actors A, a set of link nodes L, and a set of communicational edges E. In this thesis, the actors and link nodes are generalized as actors, since a link node is essentially an actor that only duplicates its input data. SDF graph is then defined as in Eq. 2.2

$$G = (A, E) \tag{2.2}$$

- Each ***actor*** a in the actor set A is a function, e.g. addition, subtraction, Fast Fourier Transform (FFT), FIR filter, Viterbi decoder, matrix multiplication, etc.
- Each ***edge*** e in the edge set E is a directed communicational edge representing the data dependency between two actors. The edge that connects a_i and a_j is denoted as $e_{a_i \to a_j}$.
- The produced or consumed data samples are called ***data tokens*** (denoted as d) that can have any data type, e.g. *integer*, *real number*, *complex number*, *vector*, *matrix*, etc.
- The ***edge weight*** w_e on an edge e representing the number of produced data tokens and the number of consumed data tokens in the format of $w_e = p_e/c_e$, where p_e is the number of produced data tokens and c_e is the number of consumed data tokens. For example, an edge weight of 2/1 means two produced data tokens and one consumed data token. Notice that the edge weight is not a fraction but a record of two integers.

Figure 2.2 shows an example SDF graph with four actors and three edges. The actor set A contains a_0, a_1, a_2, and a_3. The edge set E includes $e_{a_0 \to a_1}$, $e_{a_1 \to a_2}$, $e_{a_3 \to a_1}$. The edge weights are $w_{e_{a_0 \to a_1}} = 2/1$, $w_{e_{a_3 \to a_1}} = 1/2$, and $w_{e_{a_1 \to a_2}} = 2/1$. In each execution, actor a_0 produces two data tokens to a_1. Actor a_1 consumes one data token from a_0 and one data token from a_3. It also produces two data tokens to a_2. Actor a_2 consumes one data token from a_1 and actor a_3 produces one data token to a_1.

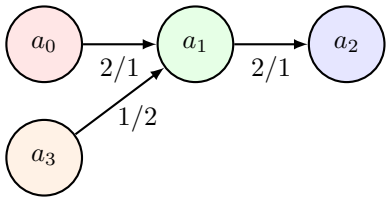


Figure 2.2: SDF Graph Example

Topology Matrix

The edges in an SDF graph can be represented in a compact manner by using a *topology matrix* T. This concept is used in later on in this section with the *repetition vector* for computing the number of executions of each actor. Each row in T represents the data communications on one edge. Each column in T represents the number of data tokens that are produced or consumed by one actor on different edges. The element at row e, column a is the number of produced (positive value) or consumed (negative value) data tokens. For a given SDF graph, the element values of its topology matrix T can be obtained by using Eq. 2.3, where p_e is the number of produced data tokens on edge e, c_e is the number of consumed data tokens on edge e, s_e is the source actor of edge e, and d_e is the destination actor of edge e.

$$T_{e,a} = \begin{cases} p_e & a = s_e \\ -c_e & a = d_e \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

The topology matrix of the example SDF graph (Figure 2.2) is shown below. The first row is for the edge $e_{a_0 \to a_1}$, the second row is for the edge $e_{a_3 \to a_1}$, and the third row is for the edge $e_{a_1 \to a_2}$. the first column is for actor a_0, the second column is for actor a_1, the third column is for actor a_2, and the fourth column is for actor a_3.

$$T = \begin{bmatrix} 2 & -1 & 0 & 0 \\ 0 & -2 & 0 & 1 \\ 0 & 2 & -1 & 0 \end{bmatrix}$$

Repetition Vector

A *repetition vector* q of an SDF graph represents the number of invocations in one system iteration for all SDF actors. The i^{th} element in q is the number of invocations of the actor a_i. The mathematical definition of the repetition vector q is in Eq. 2.4, where $|A|$ is the number of actors in the actor set A.

$$q = \{q_i \mid q_i \in \mathbb{Z}_+,\ 0 \leq i < |A|\} \tag{2.4}$$

Since the number of produced data tokens may not be equal to the number of consumed data tokens on an edge e, sometimes, it is necessary to invoke data source actor s_e and data destination actor d_e several times during a system iteration to ensure that all produced data tokens on that edge are consumed. A repetition vector is valid if it ensures the number of produced data tokens equals to the number of consumed data tokens in one system iteration for all edges, which means Eq. 2.5 should be valid [28]. The notation $\vec{0}$ is an all zero vector.

$$T \cdot q = \vec{0} \tag{2.5}$$

SDF Scheduling

An SDF graph can be scheduled statically at compile time since the data rates for all data paths are constant. SDF scheduling is the process of finding the number of actor invocations (the repetition vector) in one system iteration such that the total number of produced and consumed data tokens are identical. However, not all SDF graphs can be scheduled. An SDF graph can be scheduled only if Eq. 2.6 is met [11]. T is the topology matrix of the given SDF graph and $|A|$ is the number of nodes in that SDF graph.

$$\text{rank}(T) = |A| - 1 \tag{2.6}$$

An example of an unschedulable SDF graph is shown in Figure 2.3. It has three actors a_0, a_1, a_2 and three edges. One invocation of actor a_0 requires two invocations of a_1 to consume all the produced data tokens (2/1). Similarly, one invocation of actor a_1 requires two invocations of a_2 to consume all the produced data tokens (2/1). Therefore, one invocation of actor a_0 requires four invocations of a_2. However, by looking to the weight of the edge between a_2 and a_0, we can see that one invocation of actor a_2 requires one invocation of a_0. Different invocation requirements make this SDF graph unschedulable.

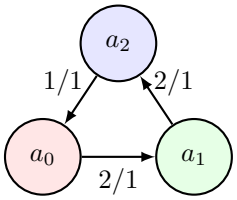


Figure 2.3: An SDF Graph without Static Schedule

The topology matrix of this SDF graph is as follows:

$$\begin{bmatrix} 2 & -1 & 0 \\ 0 & 2 & -1 \\ -1 & 0 & 1 \end{bmatrix}$$

The rank of its topology matrix is 3 and $|A| - 1$ is 2. Therefore, Eq. 2.6 is not met for the SDF graph. Consider the SDF graph example in Figure 2.2. Its topology matrix is as follows:

$$\begin{bmatrix} 2 & -1 & 0 & 0 \\ 0 & -2 & 0 & 1 \\ 0 & 2 & 1 & 0 \end{bmatrix}$$

The rank of the topology matrix is 3 and $|A| - 1$ is also 3. Therefore, Eq. 2.6 is met for that SDF graph. The repetition vector q of a schedulable SDF graph with the actor array A and edge set E can be computed by the linear-time algorithm shown in Algorithm 2.1, where q_i is the number of repetitions of the actor a_i, q_j is the number of repetitions of the actor a_j, w_e is the edge weight of e, c_e is the number of consumed data tokens on edge e, and p_e is the number of produced data tokens on edge e.

1 Set $E' = E$ and all the elements in q to 0.
2 **Check**:
3 **if** $E' = \varnothing$ **then** End of algorithm
4 Pick an edge $e = e_{a_i \to a_j} \in E'$ at random and remove e from E'
5 **if** $q_i \neq 0$ *and* $q_j \neq 0$ **then** Go to **Check**
6 **if** $q_i = 0$ *and* $q_j = 0$ **then** Set $q_i = c_e$ and $q_j = p_e$ and go to **Check**
7 **if** $q_j \neq 0$ **then** Set $q_i = q_j / w_e$ and go to **Check**
8 **else** Set $q_j = q_i \cdot w_e$ and go to **Check**

Algorithm 2.1: Computing Repetition Vector

If one of the actor a_i of an edge e has a repetition count, the repetition count of the other actor a_j can be assigned using Eq. 2.7.

$$q_j = \begin{cases} q_i \cdot w_e & a_j \text{ is source} \\ q_i / w_e & a_j \text{ is destination} \end{cases} \tag{2.7}$$

For the SDF graph example in Figure 2.2, if we start from the edge connecting a_0 to a_1, first we assign q_0 to 1 and q_1 to 2. After that we assign q_2 to 4 by checking the edge connecting a_1 to a_2. Then we assign q_3 also to 4 by checking the edge connecting a_3 to a_1. The resulting repetition vector q for the SDF graph is then $[1, 2, 4, 4]$, indicating that a_0 executes once, a_1 executes twice, a_2 executes four times, and a_3 executes four times in one system iteration.

In System-Level Architectural Synthesis Framework (SYLVA), the repetition vector is computed by using a Constraint Programming (CP) solver based method. Finding a repetition vector is modeled as a Constraint Satisfaction Problem (CSP). The variable to assign is the repetition vector q. The domain of each element in q is $\mathbb{Z}_+$. The constraint is $T \cdot q = 0$. The variable assignment strategy is assigning the minimum value for each element of q. By using this CP solver based method, the implementation of SYLVA does not have to include the algorithm for computing the repetition vector q.

2.2 Dynamically Reconfigurable Resource Array

In this section, we introduce Dynamically Reconfigurable Resource Array (DRRA) and Distributed Memory Architecture (DiMARCH), a representative Coarse-Grained Reconfigurable Architecture (CGRA) [15] design. System-Level Architectural Synthesis Framework (SYLVA) targets three implementation styles as stated earlier in chapter 1. This includes Application-Specific Integrated Circuit (ASIC), Field-Programmable Gate Array (FPGA), CGRA. Here we describe DRRA and DiMARCH to help reader familiarize with these designs so that when we discuss mapping and code generation in terms of DRRA and DiMARCH, in chapters 7, the reader is primed with their details.

DRRA is a CGRA under development in KTH. The DRRA fabric is a parallel distributed Digital Signal Processing (DSP) fabric with distributed arithmetic, logic, interconnect and control resources [14]. DRRA deploys pools of small, simple and agile resources for computation, storage and interconnect. In DRRA, it is also possible to create runtime partitions that are customized to the requirements of individual applications. We will briefly present the DRRA architecture in this section for completeness, with emphasis on the control and storage aspects. An example DRRA fabric with size 7×2 is shown in Figure 2.4. Note that this is a fragment of the DRRA fabric. A DRRA fabric of any dimension can be instantiated.

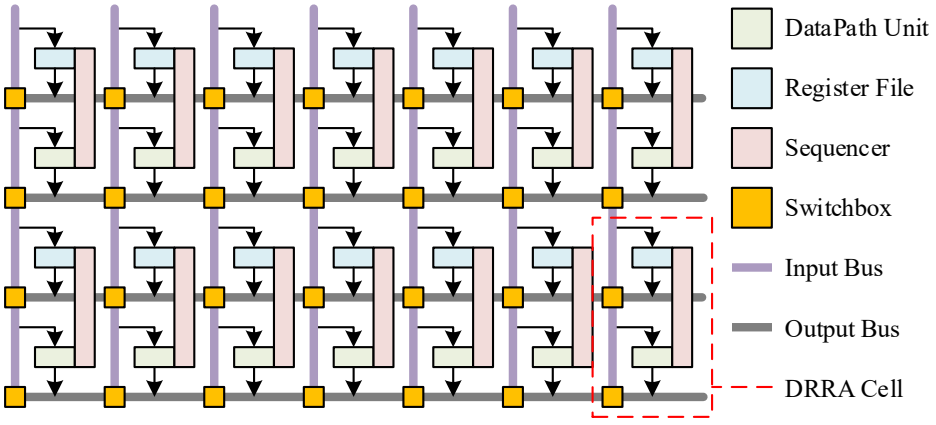


Figure 2.4: DRRA Fabric

DRRA Cell

A CGRA node in DRRA is called a *DRRA cell* and each DRRA cell is identified by its location (x, y) as shown in Figure 2.4. The PEs in DRRA are DataPath Units (DPUs). One DPU has an array of operating modes and the implemented operating modes can be tailored based on the target application domain. Each DRRA cell also has a register file and a sequencer.

DataPath Unit

A DPU is a typical DSP unit with four inputs corresponding to two complex numbers and two outputs corresponding to a single complex number. The operand width is a design time parameter and in our experiments we have set it to 16 bits. The modes supported are add, subtract, multiply, Multiplication-Accumulation (MAC). The accumulation can be internal, external, and an optional add before MAC allows symmetric MAC. Half radix-2 butterfly for Fast Fourier Transform (FFT) type operations, sum of difference and difference of sum are other arithmetic modes. DPU also deals with over/under flow detection, saturation, and rounding and DPU has sophisticated configurable truncation mechanism to deal with arithmetic gain. The DPU also has logic, comparison and shift operations.

Register File

The register file has two read and two write ports. Each port has an independent Address Generation Unit (AGU) that can be independently configured to work in different addressing modes like linear vectorized, bit-reverse, and circular buffer with configurable offsets and increment. The AGU also has fully configurable temporal behavior to act as sophisticated streaming device. This temporal configurability is a key feature to implement one of the system level control mechanism.

Sequencer

The sequencer controls the local DPU and programs the local register file and switches. The sequencer acts like a programmable FSM and has branch and wait instructions. Sequencers also have an interrupt mechanism, whereby they can interrupt and be interrupted by neighboring sequencers up to three columns and three rows away. This feature is also fundamental to implementing one of the mechanisms for system level control.

Sliding Window Circuit Switched Interconnect

The DRRA interconnect scheme as shown in Figure 2.4 involves two categories of buses, the output horizontal buses and the input vertical buses. The horizontal output buses are segmented and straddle three columns on each side of each output from DPU or register file unless there are fewer or no columns as one reaches the edges of the fabric. The vertical input buses intersect the horizontal buses with switches that are single cycle, partially reconfigurable by the local sequencer. Each sequencer has access to two local switches, one for feeding the DPU and the other for feeding the register file as shown in Figure 2.4.

The interconnect scheme provides a sliding 7 column window connectivity, where every DPU can, in a non-blocking manner, output to or input from all DPUs and register files in a 7 column wide window, i.e. the self column, three columns to the right and three columns to the left. The sliding window, with segmented wires, makes the interconnect scheme scalable. The 7 column limitation is that imposed by the current technology and clock speed for which the fabric is synthesized. By slowing the clock, the window of reachability can be widened at the expense of increased interconnect cost (energy consumption). This enables creation of arbitrarily wide multiple concurrent data paths that can be created dynamically. A similar concept extends and enhances the fabric with distributed partitionable memory that connects the DRRA fabric fragment shown on top and bottom sides (see DiMARCH in the next sub-section).

2.2.1 Distributed Memory Architecture

DiMARCH is an existing distributed memory architecture that can solve the data transfer conflict at compile time for DRRA. By using DiMARCH, part of the DRRA cells are assigned with on-chip SRAMs and the corresponding controls. The tiles within DiMARCH can be categorized into SRAM Tile (STile) and Configuration Tile (ConTile). STile is the single memory block that constitute most tiles of DiMARCH as it encompasses the SRAM memory cell. ConTile is a layer of tiles between other DRRA Cells and STile. It encompasses all the component of an STile, sequencers, and the memory interface of the register files. More detailed information can be found in [29]. A simplified example view of DiMarch with DRRA is shown in Figure 2.5. Note that the numbers of STiles, ConTiles, and DRRA cells are expendable. This figure only for an illustration.

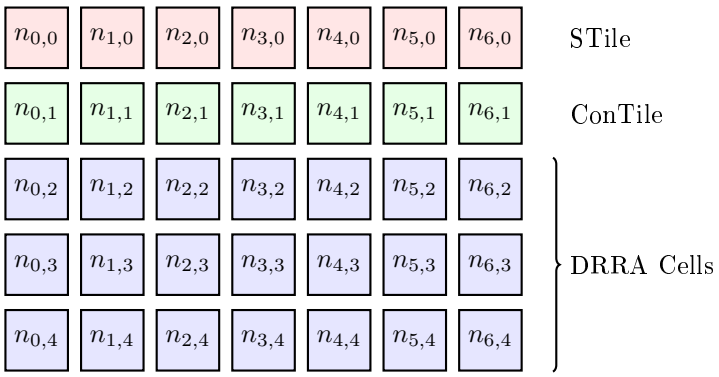


Figure 2.5: DiMarch with DRRA

2.2.2 VESYLA Compiler

VESYLA is a High-Level Synthesis (HLS) like library development tool that provides the function implementations to the system-level synthesis framework. Like any HLS tool, VESYLA translates an algorithm level description into an FSM with Datapath (FSMD). But unlike typical HLS tools, the Finite State Machine (FSM) in DRRA architecture is distributed: a datapath can be composed from an arbitrarily large number of DRRA cells. Each DRRA cell has its own sequencer that acts like an FSM. And the AGUs in the register files have their own small FSMs to control the streaming of data. The AGU FSMs in the register files are configured by the FSMs in their respective sequencers.

In VESYLA, the allocation and binding of DRRA and DiMARCH resources is done by pragmas. These pragmas are parametric in terms of the dimension of the problems and the desired degree of parallelism. By controlling these two parameters, it is possible to create Function Implementations (FIMPs) for the same function like FFT, Finite Impulse Response (FIR) filter, etc. in varying dimensions and degrees of parallelism. VESYLA is used to generate a library of FIMPs varying in architecture style and degree of parallelism. These variants of FIMPs with varying cost metrics (area, latency in cycles and energy) is the basis for Design Space Exploration (DSE) in SYLVA.

2.3 Constraint Programming

In System-Level Architectural Synthesis Framework (SYLVA), Constraint Programming (CP) [17] concepts are applied for modeling and solving its problems that involve resource allocation (Design Space Exploration (DSE), chapter 5), area of planning (Coarse-Grained Reconfigurable Architecture (CGRA) floorplanning in 7), and scheduling (DSE, chapter 5). This section will give a brief introduction to CP and list the motivation of using CP.

CP is software technology for declarative description and effective solving of large, particularly combinatorial, problems especially in areas of planning and scheduling [22]. The problems to be solved by CP are referred as Constraint Satisfaction Problems (CSPs) [18]. A CSP is denoted as a tuple $P = P(X, D, C)$, where V is the array of variables to be assigned values, D is the array of variable domains specifying the possible value of the variables regardless the constraints, and C is the array of constraints specifying the relations among all the variables in V. Each variable $v_i \in V$ has its own variable domain $d_i \in D$. Only the values in d_i can be assigned to v_i. Note that the variable domain can be finite or infinite in CP [30][31] and the constraints can be linear (e.g. $y = x$, where x is a variable) or non-linear (e.g. $y = x^2$, or $y = x[z]$, where x and z are variables).

Since CP allows non-linear constraints [32], modeling in CP is easier than the approaches that disallow non-linear constraints such as Integer Linear Programming (ILP)[33]. This is the first reason of using CP in SYLVA. An example of non-linear constraints is the area usage constraint. We need to constrain the total area usage such that the sum of the area usage of all deployed Function Implementation (FIMP) instances is smaller than the user defined maximum area usage. The individual area usage variable equals to the value that is indexed by the FIMP type variable. Assume the area usage variable is denoted as a, the FIMP type variable is denoted as f, and the area usage list of all the FIMPs of this function is denoted as L. We have following constraint: $a = L[f]$, which is not linear.

We can also implement this constraint by using only linear equations in ILP but the number of constraints will be large. First, we need to convert each FIMP type variable f to one array of variables f'. Assume we have m types of FIMPs for a given function, the FIMP type variable f will be converted into m variables: $f' = [\ f'_i\ \forall\ 0 \geq i < m\ ]$. All the variables f'_i are in the domain of $[\ 0, 1\ ]$. 0 means we do not use this FIMP type, while 1 means we use this FIMP type. After adding f', we also need to constrain that only one FIMP type is used for a FIMP. This constraint can be written as $\sum f' = 1$. Then the area usage a of a FIMP can be written as $a = L \cdot f'^T$. Note that the length of L is also M. By using ILP, we need more variables (f' vs. f) and more constraints ($\sum f' = 1$) for modeling the same problem.

Although it is also possible of using $a = L[f]$ in a ILP model as a syntactic sugar, CP and ILP solvers treat it completely different. CP solvers have dedicated and efficient implementation of $a = L[f]$, instead of replacing it with the bunch of new variables and constraints. In this manner, the degree of modeling freedom in CP is highly depend on the constraints that can be used and the solving efficiency is highly depended on the constraint implementations. Since CSP is for finding feasible solution instead of optimal solution, the solution search is in a branch and prune manner and often referred to constraint propagation [17] [30] and the constraint implementations are referred as constraint propagators [34]. Compared to the branch and bound searching strategy that is for finding the optimal solution, a CP solver searches faster since it do not need to evaluate the upper and the lower cost bounds but only evaluate if this branch may or cannot have feasible solutions. After decades of research and development, we have a large set of available constraints [35] and they are suitable of solving resource allocation, placement, and scheduling problems [22]. This is the second reason of using CP in SYLVA.

As we stated that CP it intended for finding feasible solutions. But it can also deal with optimization, however, in a iterating manner. To do that, we need a *cost function* (c) and the *optimization objective* (minimizing or maximizing c) just as the case for ILP. Assuming we are minimizing c, the iteration works as follows: First, find a feasible solution s_0 with a cost $c(s_0)$. Then add a new constraint $c(s) < c(s_0)$ and tries to find a feasible solution for this enhanced model. Iterate until cannot find a feasible solution. The last feasible solution s_i is thus an optimal solution. The total search time may be quite long since we need to explore the entire design space at least once (the last iteration).

Chapter 3

Related Work

In this chapter, we will list the literature research related to System-Level Architectural Synthesis Framework (SYLVA). The related work is categorized based on features of SYLVA. In section 3.1, we will list the related work in the literature of High-Level Synthesis (HLS) and System-Level Synthesis (SLS), and compare them with SYLVA. In section 3.2, we will discuss the related Coarse-Grained Reconfigurable Architecture (CGRA) synthesis and compilation researches. Section 3.3 is about the relate work of the Global Interconnect and Control (GLIC) synthesis in SYLVA, and section 3.4 gives some related work on CGRA programming.

3.1 System-Level and High-Level Synthesis

In this section, we will review the state of the art High-Level Synthesis (HLS) and System-Level Synthesis (SLS) tools and compare them to System-Level Architectural Synthesis Framework (SYLVA) with respect to its key features: Design Space Exploration (DSE) by considering different Synchronous Data Flow (SDF) schedules, arithmetic and buffer level parallelism, use of pre-designed and characterized function implementations, automatic global interconnect and control synthesis, and ability to target different implementation styles. Some previous approaches that use Constraint Programming (CP) on hardware synthesis will also be covered.

SYLVA bears similarity to HLS [36] [37] tools in generating hardware from specifications that are more abstract than the Register-Transfer Level (RTL). The fundamental difference between SYLVA and HLS tools is the level of granularity at which they do design space exploration and the reusable objects they use to synthesize the specification. HLS tools explore design space at a relatively finer granularity level of micro-architectural blocks (e.g. arithmetic units and registers etc). HLS tools decide on the number and type of arithmetic units, whereas SYLVA decides on the number and type of Function Implementations (FIMPs).

HLS tools also do storage (registers), control, and interconnect synthesis to glue the micro-architectural building blocks. SYLVA does global interconnect and control synthesis that glues the FIMPs, which are FSM with Datapaths (FSMDs) [37] and significantly more complex compared to the micro-architectural blocks. HLS tools and SYLVA also differ in the abstraction of the specification they synthesize from. HLS tools used to be restricted to synthesis from a single untimed algorithm and transforming it into an FSMD.

More recent HLS tools C-to-Silicon Compiler [38], Catapult [39], Vivado [40] (formally AutoPilot [41]), LegUp [42], and Bluespec [43]. do deal with multi-algorithm specifications, typically in C, C++ or SystemC. HLS tools have since evolved into accepting a hierarchy of algorithms and automatically generating a hierarchy of FSMDs. While this level of specification outwardly looks like what SYLVA also accepts as input, there is a fundamental difference. This is best explained in terms of the example shown in Figure 3.1, whose detailed description can be found in section 5.1. Figure 3.1(a) is the original SDF graph and Figure 3.1(b-e) are the possible load balanced Homogeneous SDF (HSDF) schedules (function executions are evenly distributed).

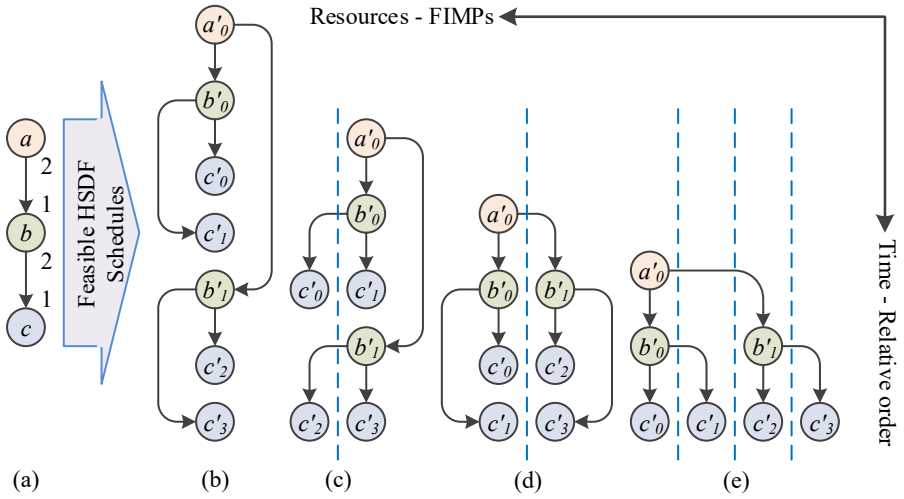


Figure 3.1: SDF Graph and a Subset of Feasible Schedules

SYLVA starts with the original SDF graph and evaluates all the HSDF schedules shown in Figure 3.1(b-e) as potential solutions. HLS tools, in contrast, starts from and considers only one of the HSDF schedules shown in Figure 3.1(b-e) as the starting point specified by the user, e.g. in Xilinx Vivado the user should specify the loop implementation to be inline or pipelined based on the performance of the current synthesis result (slack, area, etc.). This is where the design space explored by SYLVA becomes more rich compared to what is explored by HLS tools. And the use of FIMPs as granularity of reuse is not only more effective. It is intrinsically tied to this dimension of design space exploration. HLS tools differ in the way they deal with the hierarchy of algorithms. Commercial HLS tools like C-to-Silicon Compiler [38] from Cadence, Vivado [40] from Xilinx transform each algorithm individually into a RTL specification and the optimization is local, i.e., restricted to individual algorithms being synthesized. A global cost function factoring the cost of all algorithms is not taken into account.

A recent work by Carloni et. al. [44] addresses exactly this issue by proposing a global cost function that considers the collective cost (area, energy) of all the functions in the input specification, while considering the constraints to set for individual functions during HLS. It has also proposed a library of such functions to encourage reuse, similar in spirit to the FIMP library, but not pre-characterized as is the case in SYLVA. The constraints for HLS are decided during SLS as mentioned above. The other key differences compared to SYLVA is that like other HLS tools, this work also starts with a specific scheduled system and does not consider multiple schedules like SYLVA does to explore the function level parallelism. In summary, it explores the arithmetic level parallelism with a global cost function in contrast to other HLS tools but it does not explore the function level parallelism like SYLVA. More information about parallelism can be found in chapter 5.

Gordon et. al.'s work StreamIt [45] provides task, data and pipeline parallelism. The task level parallelism is explicitly specified by the user. The more interesting dimension is that of data level parallelism, which the StreamIt explores at arithmetic level but is able to apply more coarse grain transform to fusion and fission to create more serial and parallel solutions. It does not have a concept of pre-characterized FIMPs, as it targets programmable coarse grain RAW tiles [46]. The data level parallel design space it considers is comparable to what SYLVA considers but it explores at a finer granularity of arithmetic units and it targets programmable RAW tiles, instead of FIMPs, that are designed to execute just one algorithm/function. SYLVA also targets Coarse-Grained Reconfigurable Architectures (CGRAs) like Dynamically Reconfigurable Resource Array (DRRA) [14] that are similar to the RAW fabric. However, SYLVA treats the FIMPs that are compiled for DRRA like hard macros and DRRA as a sea of micro-architectural resources that can be clustered to implement a function, similar to Field-Programmable Gate Arrays (FPGAs) but at significantly coarser granularity. In effect, SYLVA does a comparable design space exploration as StreamIt but more effectively in terms of more coarse grain pre-characterized reusable objects called FIMPs. This strategy, we believe is more scalable with the size of the problems.

SLS expression is also used by design frameworks targeting Multi-Processor System-on-Chips (MPSoCs). ESPAM [47] takes application behavior modeled (Kahn Process Network (KPN) [48]) along with platforming and mapping specification and automatically generates the MPSoC and the application code. Daedalus-RT [49] targets hard real-time streaming applications and builds on the ESPAM framework. The applications of Daedalus-RT system are static affine nested loop programs [50], which are then parallelized by the PN compiler [51] and fed to the ESPAM system. Gladigau et.al [52] have presented another SLS tool targeting MPSoCs. This tool uses a template based approach and it performs the decision making and refinement for computation and communication to map behaviors to the structure (instances of MPSoC templates). SYLVA differs from these tools as it targets hardware implementation instead of mapping software to MPSoCs.

There are a list of research efforts of applying CP on hardware synthesis. In [53], the author describes a method for modeling and solving different scheduling and resource assignment problems that are common in HLS and SLS. The methods proposed in [54, 55] are both CP solver based and take SDF graph as inputs. The authors in [54] proposed an approach focusing on allocating and scheduling a SDF graph onto a multi-core platform subject to a minimum throughput requirement. This approach is a complete algorithm based on CP that solves the allocation and scheduling problems as a whole and a number of search acceleration techniques are used to significantly reduce run time. It is similar to DSE in SYLVA, which is also CP based and only examines the load balanced schedules (section 5.5.1) to significantly reduce the search space. The approach used in SYLVA is also more complex since it does not only consider scheduling and resource allocation, but also the number of FIMPs (section 4.3) to be used in the system. More over, the optimization objective of SYLVA can be area, latency, sample rate, and energy, individually or as a whole instead of throughput only. In [55], the authors use CP to calculate the repetition vector and generate the schedule using a CP solver called JaCoP (Java Constraint Programming) [56]. This approach is also similar to DSE in SYLVA but is much more simpler since the DSE step in SYLVA not only computes the repetition vector and generates the schedule but also generates the number of FIMPs to be used and performs resource allocation (i.e. which types of FIMPs are used). Authors in [57] proposed a method for a systematic design space exploration for application specific processors, intended to reduce the design effort in the initial phase, by proposing a set of alternative potential solutions. Each solution is in terms of the processor configuration that includes number and width of SIMD units and the number of scalar units that allow for the required performance, without wasting resources. The authors employ constraint programming and formulate the problem as a Pareto optimization problem, using modulo scheduling. Authors in [58] proposed a constraint programming based design space exploration for streaming applications (such as MPEG) running on multi-processor platforms with guaranteed service interconnects. It is similar to the DSE in SYLVA and SDF is also used there for modeling the input system as SYLVA does.

3.2 Hardware Synthesis for CGRA

In this section, we review automatic synthesis approaches for Coarse-Grained Reconfigurable Architectures (CGRAs) and also array the related work for CGRA floorplanning. The hardware synthesis approaches targeting CGRAs is the more general case of automating synthesis from algorithmic level, which has two decades long history in the High-Level Synthesis (HLS) research and is gaining some acceptance in the industry [9].

3.2.1 CGRA Architectures and Synthesis Tools

Large number of CGRAs and the corresponding synthesis tools have been proposed in the literature. By the type of function units, CGRAs can be categorized into microprocessor-based CGRAs and accelerator-based CGRAs. For example, ADRES [59] using VLSI and SmartCell[60] using simple micro-processors are microprocessor-based CGRAs. EGRA [61] using customized cores and Dynamically Reconfigurable Resource Array (DRRA) [16] using sequencer and customizable accelerators are accelerator-based CGRAs.

In [59], authors from IMEC developed the DRESC framework to target a family of CGRAs. This framework mainly focuses on loop-level application code segment parallelization. By identifying the critical segments, Digital Signal Processing (DSP) functions are accelerated and mapped to ADRES [59] CGRA architecture. In [62], authors from University of Twente developed a HLS like compiler targeting the MONTIUM architecture for mapping Control Data Flow Graph (CDFG) to MONTIUM Arithmetic Logic Units (ALUs) and corresponding scheduling and allocation. In [63], authors from Aristotle University, Greece, proposed an automated approach for software acceleration on RISCs. It maps set of task graphs presented in behavioral VHDL model to target Field-Programmable Gate Array (FPGA) using temporal partitioning algorithm which is modeled as an Integer Linear Programming (ILP) technique. Instructions like Discrete Cosine Transform (DCT), Finite Impulse Response (FIR) filtering are created for acceleration. In [64], authors from IISc Bangalore proposed an automatic approach to code acceleration for their CGRA. This approach involves identifying hyperops and runtime CGRA configuration. Another related work is the Altera's C2H compiler[65]. It automatically identifies hotspots, creates corresponding accelerators in FPGA and then invokes them from the software. In [66], authors from Munich developed a C language frontend compiler. It temporally partitions code and generates a structural representation in terms of the XPP's PEs for each partition in native mapping language. Then the generated representations are automatically mapped by their xmap tool. In [67], authors built a compiler to map instructions extracted from a CDFG to a reconfigurable hardware by using GNU Compiler Collection (GCC) framework. The most common and regular instructions are mapped onto a reconfigurable functional unit operation.

In [68], authors introduced an Architecture Description Language (ADL) for CGRAs. ADL defines the number of functional units, ports, processing elements used and describes the architecture with the interconnect methodology. The research work presented in [69] proposes a language called RaPiD-C for mapping algorithms on specified architecture. The programmer should select and identify the parallelism, data movement, data partitioning among various functional units and ALUs of RaPiD array. After compilation, loops are identified. The outermost and innermost loops are specified as time and space loops in RaPiD. Then the control trees are extracted to determine the loop structure in the algorithm. The drawback of RaPiD-C is the need of parallelism extraction by the programmer.

Most of the CGRA synthesis tools tied to a specific architecture and/or are only simple assemblers. The most flexible synthesis tools are SPR [70] (the compiler in Mosaic project), DRESC [59] (the compiler for ADRES) and [71]. However, they do not support sliding window circuit switching CGRAs and pipeline stage modification during floorplanning, which are supported by the proposed floorplanning tool in System-Level Architectural Synthesis Framework (SYLVA) (it supports Application-Specific Integrated Circuit (ASIC), FPGA, and DRRA).

While all the approaches reviewed above differ in detail in their approaches, broadly they can be categorized as HLS type techniques. HLS has been under intensive research in academia and industry for more than two decades and still has two issues. The first one is that compared to manual design, HLS still produces inferior design, as was reported in [72]. Secondly, most of the HLS approaches attempt at automating synthesis of individual algorithms/functions of the physical layer. The user has to partition the entire application into individual algorithms, synthesize them and integrate them back to obtain the final results. This issue was also reported in [72].

All the above mentioned approaches have extremely large search space since they all have to consider in micro-architectural element level (like arithmetic units, flops and multiplexers etc). They act just like what the Register-Transfer Level (RTL) synthesis does today. In fact, the abstraction of HLS is raised beyond RTL, without the corresponding increase in the granularity of the building blocks. Additionally, the optimization decisions taken at algorithmic level require very accurate estimation or look-ahead functions for latency and energy that have not been available for the standard cell based ASIC style implementations or even FPGAs.

In our approach, the Function Implementation (FIMP) library for CGRA comes with a large set of DSP functions equivalent to the Simulink blocks at this moment and gives the user the ability to modify the existing functions and add architectural alternatives for them and also add new functions. A vectorizing symbolic language assembler for CGRA called VESYLA [73] is used to efficiently and compactly represent the micro-architectural space of the DSP functions. And our approach is not to accelerate DSP algorithms but to implement the entire physical fabric in ASIC style.

FPGA vendors, Altera and Xilinx, and Electronic Design Automation (EDA) vendors Synopsys also offer DSP libraries in the form of generators for popular DSP functions [74]. The fact that these vendors have resorted to function specific generators, hints at either the generator based approach is more efficient or better quality solutions can be reached or both compared to HLS of the same DSP functions for which generators are being marketed.

A further downside of these function based generator approach is that they are closed solutions; the end user is restricted to the architectural style and degrees of freedom provided by the generators. More critically, only the available generators can be used in the first place. Moreover, these generators do not have the ability to estimate accurately the energy or the performance. For ASICs, it is hard to predict energy and performance without physical design. For FPGAs, the estimation attempts reported in [72] show a margin of error up to 20%.

3.2.2 CGRA Floorplanning

Authors in [75] differentiate the floorplanning for CGRA from ASIC and FPGA cases such that the number of resources allocated is more flexible by trading physical resources for time through time multiplexing. FPGA floorplanning algorithms ([76] using a hierarchical clustering approach and [77] using a slicing technique with compaction) focus on a much smaller grain size than CGRA floorplanning. Poor utilization in a CGRA is much more costly than in an FPGA. This is not considered yet in the proposed algorithm. We plan to extend it in the near future.

In [75], the kernels (equivalent to FIMPs in this thesis) can be shared among different functions with essentially different functionality. We do not consider time multiplexing since we assume the combination of FIMPs and schedules are optimized before floorplanning as elaborated in [78]. We focus on minimizing the communication cost and rescheduling executions. The CGRA floorplanning is similar to the Network-on-Chip (NoC) core mapping problem. The latter one is to assign cores to mesh cross-points so that the performance constraints are satisfied.

Most CGRA floorplanning tools like the placer of SPR [70] and DRESC [59] use simulated annealing algorithm. Compared to the constraint programming technique used in SYLVA, modeling using simulated annealing is much more complicated due to the difficulty of defining the constraints. The proposed heuristic algorithm in SYLVA for the target CGRA is also expected to be out-performed than simulated annealing.

3.2.3 Application Mapping Problem

The CGRA floorplanning problem is similar to the memory-aware application mapping problem for NoC in terms of the optimization objectives: less area usage and energy consumption while keeping the constrained performance (latency and sample interval), and the target architecture: CGRAs are mostly connected via mesh NoCs. The CGRA floorplanning is differentiated on the grain size. If every function implementation takes one node in the CGRA, the CGRA floorplanning problem is equivalent to the NoC mapping problem. Thus, the proposed heuristic algorithm and also works for NoC mapping and also supports the sliding window circuit switching.

The heuristic algorithms proposed in [79] and [80] focus on bandwidth and energy, respectively. They are similar to the proposed heuristic algorithm in SYLVA. Evolutionary-like algorithms are also widely used, e.g. genetic algorithm is used in [81] and particle swarm optimization is used in [82]. They are similar to solving the proposed constraint programming model.

Different mapping algorithms [83] [81] are proposed without memory-awareness and scheduling coverage. In [84], scheduling problem is covered with mapping but without memory-awareness. As claimed in [85], memory is critical in NoC design process and consequently should be considered during mapping and scheduling practical applications onto practical platforms. In [86], memory-awareness is covered while scheduling is not considered.

In the previous research [87], we propose a very fast algorithm called Layered Spiral Algorithm (LSA) to solve memory-aware application mapping and scheduling problem. The objective of LSA is minimizing total energy consumption while keeping high task level parallelism. The proposed algorithm is based on the spiral algorithm [88]. It is a very fast mapping algorithm without memory-awareness. We extend spiral algorithm by introducing memory-aware concepts and task layers to cover both memory-awareness and scheduling problem. LSA is able to solve large scale problems with acceptable accuracy. Although dynamic mapping is a cut above static mapping in terms of platform utilization [89], extra control logics are required. Therefore, we consider static application mapping and scheduling for simplification and its dynamic extension is planned in the future work.

3.3 Global Interconnect and Control Synthesis

The automatic Global Interconnect and Control (GLIC) generation is a forsaken in all High-Level Synthesis (HLS) synthesis tools and methods (e.g. LegUp [42], PipeRench [90], Cadence C-to-Silicon Compiler [91], Bluespec Compiler [92], Calypto Catapult [39], Vivado [40], AutoPilot [41], Altera OpenCL Compiler [93], FCUDA [94], and the automatic method in [95]). For the HLS tools working on behavioral models (e.g. actor-oriented model in SystemC), the system designer needs to manually generate following items:

1. Control logics for implementing the communication schedule.
2. Detailed interconnect specification for connecting components.
3. Data buffers (if necessary) to glue pre-synthesized component together.

In the context of software development for parallel computing, StreamIt [96] provides a feature which is comparable with automatic GLIC generation. When StreamIt implements a compile time computed data parallelism, the parallelized filters are automatically connected to the other parts of the system. Since the abstraction level of HLS is already high, we consider the automatic GLIC synthesis would be a new and crucial challenge for researchers to further improve the design productivity in the automatic hardware synthesis domain.

3.4 Programming CGRA

Programming a reconfigurable architecture is already discussed in many literals. In [97], the needs of compiler and application support for reconfigurable SoCs are addressed. By using System-Level Architectural Synthesis Framework (SYLVA), the designer is able to program Coarse-Grained Reconfigurable Architectures (CGRAs) either using Synchronous Data Flow (SDF) defined by programming languages (Python, C#, or Javascript), or directly in Simulink graphic environment. The Dynamically Reconfigurable Resource Array (DRRA) compiler in SYLVA is provides a dedicated compiler and the DRRA Function Implementation (FIMP) library provides optimally compiled functions for application development.

In [98], high-level programming of coarse-grained reconfigurable architecture is discussed. Occam-pi was used as the programming language. A programming language called ARMLang is proposed in [99]. It is for regular processor arrays in particular reconfigurable meshes. In [100], Reconfigurable Computing C (RCC) is proposed. It is a subset of the ANSI-C. Hence, programmers do not have to learn a new language. In SYLVA, Simulink is used as the application development language since 1) Simulink implicitly provides data/control dependency as well as synchronization information and parallelism specification, 2) the simulation and debugging is visualized, 3) it is already well developed and algorithm designers know it well.

Chapter 4

System Modeling

As introduced in chapter 1, the input to System-Level Architectural Synthesis Framework (SYLVA) is a Digital Signal Processing (DSP) sub-system that is modeled by an Synchronous Data Flow (SDF) graph with user provided constraints and optimization objectives. This chapter focuses on the system modeling method that will be used to construct the input SDF graph to SYLVA as well as the related SYLVA concepts. The layout of this chapter is shown below.

- Section 4.1 defines the SDF graph in the context of SYLVA. Besides the basic concepts of actors and edges, SYLVA also associates many of its design flow concepts to SDF graph.
- Section 4.2 describes the concepts of Homogeneous SDF (HSDF) graph, which is a special case of SDF graph and acts as the connection between the system model, SDF graph, and the hardware implementation, Function Implementations (FIMPs).
- Section 4.3 describes the concept of FIMP. FIMPs are the basic building blocks to be used by SYLVA to construct the system implementation.
- Section 4.4 describes the method of system modeling with a middle size example: the physical layer of Long Term Evolution (LTE) [27] uplink transmitter.
- Section 4.6 gives a summary of this chapter.

4.1 Synchronous Data Flow Graph

The targeting application domain of System-Level Architectural Synthesis Framework (SYLVA) is Digital Signal Processing (DSP) sub-systems that are statically schedulable: the execution start times of all the actors are determined at synthesis time [101]. Each Synchronous Data Flow (SDF) actor in the SDF graph represents a DSP function with fixed input and output data rates (data token per invocation). The example of such DSP functions are Fast Fourier Transforms (FFTs) and Finite Impulse Response (FIR) filters. Each SDF actor has three properties: function name, connectivity, and unique index. Each SDF edge has four properties: source actor, source port, sink actor, and sink port.

1. Function name, e.g. *fft* for FFT, *fir* for FIR, etc. identifies the function that is represented by the actor and this property will be used for filtering the candidate hardware implementations.

2. The connectivity in terms of data input ports and data output ports. Each input port represents a data sink that can be used as the end point of an SDF edge while each output port represents a data source that can be used as the starting point of an SDF edge. For example, an actor representing a 64-point FFT may have 64 input ports and each input port consumes one complex number per invocation. Note that, in SYLVA, we assume one output port cannot be directly connected to any input port on the same SDF actor.

3. The unique actor index in the entire system. This index will be used to identify actors in the system. It is also required to distinguish the actors with the same function name. For example, a DSP sub-system may have multiple FFTs. All of them have the same function name *fft* and they differ from each other by its unique index.

4. The source actor, source port, sink actor, and sink port will be used to determine the data dependency.

These properties of SDF actor and edge should be provided by the system designer. Modeling a system using SDF graph is providing such information.

4.2 Homogeneous Synchronous Data Flow (HSDF) Graph

4.2.1 Basis of HSDF Graph

Edward and David [11] introduced Homogeneous SDF (HSDF) graph as a special Synchronous Data Flow (SDF) graph where all the actors produce and/or consume one data token per invocation. As a consequence, the repetition vector of any HSDF graph will be list of 1's, i.e. all actors execute once. The reason is that since the numbers of produced and consumed data tokens are the same, the number of repetitions of the producing actor should equal to the number of repetition of the consuming actor. In this thesis, System-Level Architectural Synthesis Framework (SYLVA) extends the definition of HSDF graph such that if for all edges in one SDF graph, the number of produced data tokens on the edge equals to the number of consumed data tokens on the same edge, that SDF graph is an HSDF graph. Therefore, the number of produced and consumed data tokens can be one or more and the repetition vectors are still list of 1's. Figure 4.1 illustrates the difference between the original HSDF definition and the HSDF definition in SYLVA.

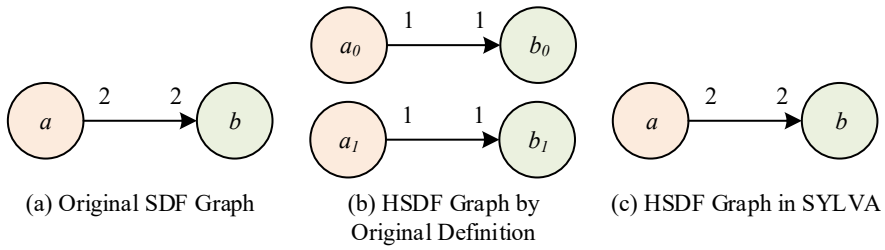


Figure 4.1: HSDF Graph in SYLVA

Figure 4.1(a) is the original SDF graph. The equivalent HSDF graph by the original definition is shown in Figure 4.1(b) that all the actors produce and/or consume one data token per invocation. The method of converting an SDF graph to its equivalent HSDF graph can be found in [11]. The equivalent HSDF graph by the SYLVA definition is shown in Figure 4.1(c). There are two data tokens produced and consumed per invocation. According to [11], an SDF graph can be converted into an HSDF if the SDF graph is statically schedulable. This also applies for the HSDF graph in SYLVA. The method of converting an SDF graph to its equivalent HSDF graph in SYLVA can be found in Algorithm 4.1 in the next page.

```
1  Find repetition vector p
2  Create HSDF graph G = (A', E')
3  Set HSDF actor list A' = ∅ and HSDF edge list E' = ∅
4  foreach actor a_i ∈ A do
5  |  Create HSDF actor list A'_i = {a'_{i,j} | a'_{i,j} = copy of a_i, 0 ≤ j < p_i}
6  |  foreach actor a'_j ∈ A'_i do  Append a'_j to A'
7  end
8  foreach edge e_i ∈ E do
9  |  Set s = source actor index in A and d = destination actor index in A
10 |  if p_s > p_d then  Set less = p_d and more = p_s
11 |  else  Set less = p_s and more = p_d
12 |  Set r = more/less
13 |  foreach integer l, 0 ≤ l < less do
14 |  |  foreach integer m, 0 ≤ m < r do
15 |  |  |  Create new edge e_new
16 |  |  |  if p_s > p_d then
17 |  |  |  |  Set source actor of e_new = a'_{s,m}
18 |  |  |  |  destination actor of e_new = a'_{d,l}
19 |  |  |  |  a'_{s,m} is the mth actor in the previously created actor list A'_s
20 |  |  |  |  a'_{d,l} is the lth actor in the previously created actor list A'_d
21 |  |  |  else
22 |  |  |  |  source actor of e_new = a'_{s,l}
23 |  |  |  |  destination actor of e_new = a'_{d,m}
24 |  |  |  |  a'_{s,l} is the lth actor in the previously created actor list A'_s
25 |  |  |  |  a'_{d,m} is the mth actor in the previously created actor list A'_d
26 |  |  |  end
27 |  |  |  Set c = min(produced tokens of e_i, consumed tokens of e_i)
28 |  |  |  Set source port of e_new = source port in e_i
29 |  |  |  Set destination port of e_new = destination port in e_i
30 |  |  |  Set produced data token count of e_new = c
31 |  |  |  Set consumed data token count of e_new = c
32 |  |  |  Append e_new to E'
33 |  |  end
34 |  end
35 end
```

Algorithm 4.1: SDF Graph to HSDF Graph Conversion in SYLVA

The HSDF graph in SYLVA also assumes that each SDF edge is a First In First Out (FIFO) style channel with zero delay. It only represents the data dependency and the number of produced and consumed data tokens of the two actors. The channel delay will be modeled as a special SDF actor that only bypasses its input data to its output with a predefined delay. This data communication model enables SYLVA to model arbitrary communication delay.

In addition, SYLVA assumes that the number of produced/consumed data tokens are integers and the data communication between two SDF actors will be implemented by homogeneous data communications between two sets of HSDF actors (source actor and destination actor sets) that are derived from one source - destination actor pair, i.e. all the producing HSDF actors have the same number of consuming HSDF actors and the same number of data tokens to each consuming HSDF actor. This assumption implies that the number of produced data tokens p_e and the number of consumed data tokens c_e on one edge e should fulfill Eq. 4.1. Otherwise the produced and/or consumed data tokens on the converted HSDF graph will not be integer or the data communication will not be homogeneous.

$$\forall\, e \in E, \quad \frac{p_e}{c_e} \in \mathbb{Z}_+ \vee \frac{c_e}{p_e} \in \mathbb{Z}_+ \tag{4.1}$$

The examples of acceptable and unacceptable conditions are illustrated in Figure 4.2. The a_4 to a_5 edge is unacceptable since 2/3 is not integer. The a_6 to a_7 edge is also unacceptable since 3/2 is not integer.

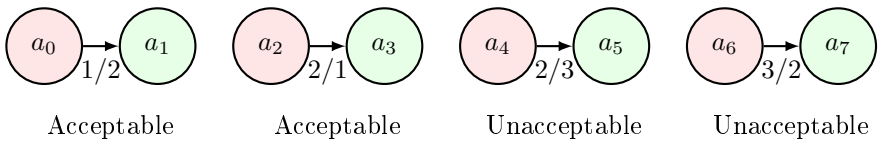


Figure 4.2: Example of Accepted and Unaccepted Data Communications in SYLVA

An example of SDF graph to HSDF graph conversion is shown in Figure 4.3. Figure 4.3(a) shows an example SDF graph with 4 functions a, b, c, and d. Figure 4.3(b) shows the equivalent HSDF graph, where all the edges have the same weight 1/1. More information about the notations related to SDF graph can be found in section 2.1.2. Note that there could be multiple output ports connected to the same input port or multiple input ports connected to the same output port. These will be handled during the Design Space Exploration (DSE) step. See section 5.4.2 for the detailed description.

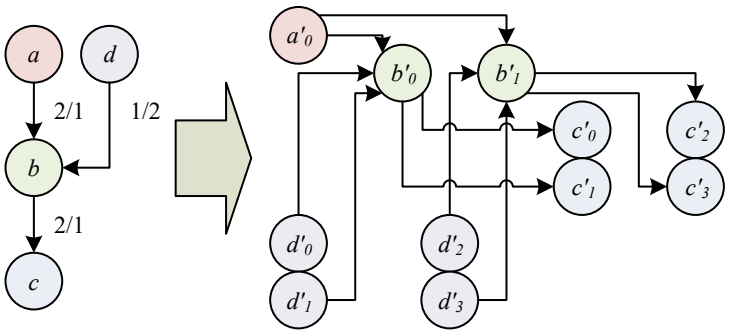


Figure 4.3: HSDF Graph Example

In the DSE step (chapter 5), the input SDF graph will be automatically converted into HSDF graph using Algorithm 4.1. After that, the HSDF graph is annotated with notations for invocation time, execution hardware, etc. during the SYLVA design flow. Detailed descriptions on those notations can be found in later chapters.

4.2.2 Feedback Loops

In SYLVA, each communication edge in feedback loop is modeled as one data sink/source pair. The data sink is for collecting the data from the actor that provides the feedback data. The data source is for sending data to the actor that is under feedback. The feedback loop modeling in SYLVA is illustrated as one example shown in Figure 4.4, where the feedback edge from actor c to actor a is modeled by a data sink *snk* and a data source *src* pair.

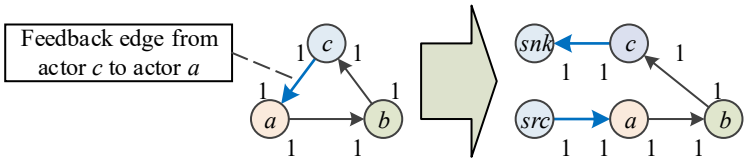


Figure 4.4: Feedback Support in SYLVA

Note that the feedback edges (e.g. from a to c in Figure 4.4) should be manually marked by the user when he/she modeling the system using SDF graph. This can be done by setting the attribute *feedback* in an edge to true. When SYLVA converts an SDF graph to HSDF graph, the feedback edges will not be touched. The data sinks and data sources will be added by SYLVA after it have the HSDF graph ready. These added dummy actors will have zero execution time, zero area and zero execution energy since they are only for recording the input and output ports information of the feedback edge and breaking the loops when calculating the system latency/sample rate during the DSE step. In the Global Interconnect and Control (GLIC) synthesis step, SYLVA will skip these dummy actors when constructing Abstract Intermediate Representation (AIR) building blocks and automatically adjust the input/output timing of the actors connected by the feedback edges. In this manner, the feedback information will be hidden from the code generator in SYLVA since it only sees AIR.

4.3 Function Implementation

Function Implementations (FIMPs) are the basic building blocks for constructing the system implementation. FIMPs are pre-designed, pre-verified, and pre-characterized. They are treated as black boxes. Each FIMP is used for implementing one Digital Signal Processing (DSP) function (e.g. Fast Fourier Transform (FFT), Finite Impulse Response (FIR) filter), with different architectures, degrees of parallelism, and implementation styles, resulting in different cost metrics in terms of area, latency, and energy. Each Synchronous Data Flow (SDF) actor may be implemented by one or multiple FIMP depending on the degree of Function-Level Parallelism (FLP) (see section 5.1 for more information). One FIMP can only implement one SDF actor. FIMPs are organized by FIMP libraries as illustrated in Figure 4.5.

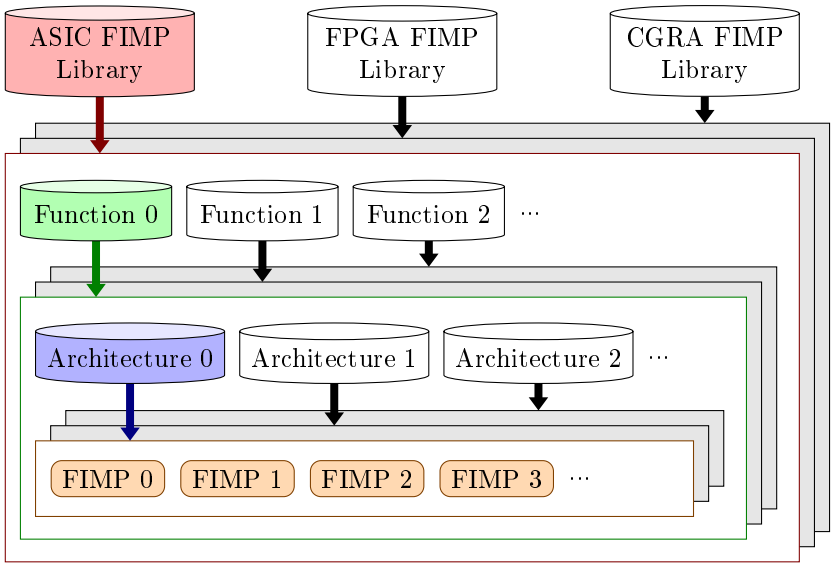


Figure 4.5: One Function, Multiple FIMPs

- Each implementation style, Application-Specific Integrated Circuit (ASIC), Field-Programmable Gate Array (FPGA), and Coarse-Grained Reconfigurable Architecture (CGRA), has one FIMP library.
- Each FIMP library contains the FIMPs for a number of functions (Function 0, Function 1, etc.). All the FIMPs in one FIMP library have the same target implementation style as indicated by the name of the FIMP library. Note that a function may not be contained in all the FIMP libraries. This is because some functions could have an FPGA version while the ASIC version may not be available.
- Each function has a number of implementation architectures. For example, an FFT can be implemented by using radix-2 as well as radix-4 butterfly operations.
- Each architecture has a number of FIMPs in different degrees of Arithmetic-Level Parallelism (ALP), e.g. an FFT can be implemented by different number of butterfly operators. The FIMPs for the same function and in the same FIMP library are differentiated by their unique function type.

In section 4.3.1, the three phase execution model of FIMP is elaborated, followed by section 4.3.2, where we elaborate the three views (interface, execution and implementation) of the FIMP model.

4.3.1 FIMP Execution Model

In System-Level Architectural Synthesis Framework (SYLVA), all DSP functions share the same execution model with three phases: input, computation and output as shown in Figure 4.6. These phases are associated with the timing parameters: t_s is execution start time, t_{ie} is input phase end time, t_{os} is output phase start time, and t_e is execution end time.

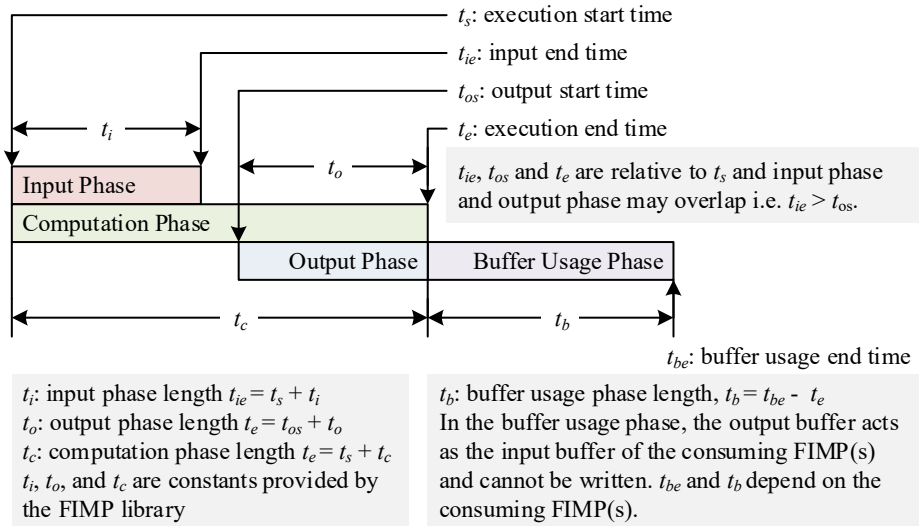


Figure 4.6: FIMP Execution Model

- Input phase: The FIMP collects input data tokens from its data source(s).
- Output phase: All the output data tokens are sent out irrespective of the presence/absence of data sink FIMPs.
- Computation phase: The FIMP performs computation on the input data tokens and possibly also on the data tokens in its internal buffer.

As described in section 4.2, one Homogeneous SDF (HSDF) actor represents one function execution. Since the function executions are implemented by FIMP instances, the FIMP execution model also applies for the HSDF actors such that every HSDF actor has these timing notations (t_s, t_{ie}, t_{os}, and t_e). In the Design Space Exploration (DSE) step (chapter 5), those timing notations will be assigned to integer values based on the cycle accurate schedule.

4.3.2 Views of the FIMP model

Interface View

The interface view of a FIMP provides a top-level view on the FIMP with three properties: function name, FIMP type, and data interface. The function name and FIMP type are used for interfacing a FIMP to SYLVA. The data interface is used for interfacing a FIMP to other FIMPs.

1. Function name

   This property is used for differentiating FIMPs based on their functionalities by specifying the name of the function that can be executed by this FIMP. For example, a FIMP has the name of fft_{64} can only implement FFT function, which is represented by an SDF or SDF actor with the same function name fft_{64}. This information is used both in the system modeling (this chapter) and DSE (chapter 5) of SYLVA. The details of SDF and HSDF can be found in section 4.1 and 4.2, respectively.

2. FIMP Type

   This property is used for differentiating FIMPs based on their cost metrics (area, latency, and energy). It is an unique index for indexing a FIMP among all the FIMPs with the same function name in the same FIMP library. For example, assume we have n FIMPs with the name of fft_{64}. The value of the FIMP type of them can be from 0 to $n-1$.

   In the DSE step (chapter 5), SYLVA will select the FIMP types of all the FIMP to be deployed in the final system implementation based on the system constraints and optimization objectives. Later on in the code generation step (chapter 7), SYLVA will generate the implementation code based on the target architecture and the FIMP type. For ASIC and FPGA implementation style, Register-Transfer Level (RTL) model will be generated. For CGRA implementation style, Dynamically Reconfigurable Resource Array (DRRA) configware will be generated.

3. Data Interface

   This property provides timing and spatial information of all the data communications of this FIMP and will be used in the Global Interconnect and Control (GLIC) synthesis step (chapter 6) to automatically generate GLIC. Since this property is closely related to GLIC synthesis, we will cover the details of this property in section 6.3.

Execution View

The execution view provides the performance and the cost metrics of the FIMP. The performance is in terms of the timing values in the FIMP execution model (Figure 4.6) and the cost metric has energy consumption in nano-joule (nJ), area usage in equivalent gate count, and the latency in terms of the number of clock cycles. Note that both FIMP execution model and energy consumption are for a single execution. In addition, although one FIMP instance may implement multiple function executions (i.e. HSDF actors) at different time, it can only execute one function at a time.

Implementation View

The implementation view provides the information of a FIMP for GLIC synthesis (chapter 6) and code generation (chapter 7). This information includes the targeting implementation style (ASIC, FPGA, or a specified CGRA), the implementation template (e.g. RTL model, CGRA configware), and the desired output buffer implementation of this FIMP.

For example, the implementation view of an 64-point FFT can be either a computation core with on-chip SRAM in VHDL for ASIC or a MATLAB function for DRRA. More detailed examples of this view will be shown in chapter 6 and chapter 7.

4.4 System Modeling Procedure

There are two mandatory steps in the system modeling in System-Level Architectural Synthesis Framework (SYLVA): ***1)*** create Synchronous Data Flow (SDF) actors and ***2)*** create SDF edges. However, if the provided Function Implementation (FIMP) library is not complete, i.e. some functions has no FIMP in the library, we need to first create the missing FIMPs to make the design synthesizable. The detailed description of FIMP and FIMP library can be found in section 4.3. The system modeling flow is depicted in Figure 4.7, where the blue blocks are procedures and the green blocks are data. To model a system, we start from the given functions, data dependencies, and FIMP library.

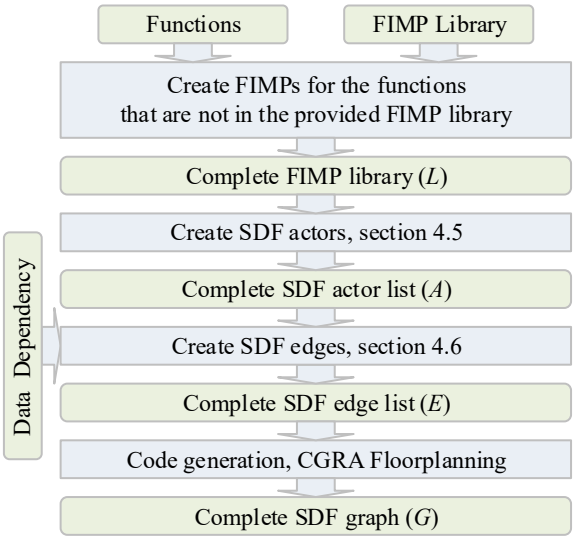


Figure 4.7: System Modeling Flow in SYLVA

4.4.1 SDF Actor Creation

Assume that we already have a FIMP library that contains the FIMPs of all the functions in the system as shown in Figure 4.7. The SDF actors are created either by using the method provided by the FIMP library or manually extracted from the FIMP library using the FIMP interface view information. The method provided by a FIMP library is for creating SDF actors, possibly based on compile time parameters. It is similar to the generic parameter in VHDL.

If the FIMP library does not provide SDF actor creation methods for the required functions, we must create the SDF actors based on the interface views of the FIMPs implementing these functions. As described in section 4.1, the required parameters for creating an SDF actor are the function name f, the input port array P_i, and the output port array P_o. All of them can be found in the FIMP's interface view (section 4.3.2).

In the rest of this section, examples in python will be used to clarify the system modeling procedure in SYLVA. Note that, besides the python frontend, there is also a SIMULINK frontend allowing system modeling in a GUI environment. An SDF creation function example is shown in Code 1. It takes three parameters: the function name as *name*, the SDF actor index as *index*, and the data token count as *data_count*. Data sink actors with one input port and with data token type of one bit can be generated by this method. The method *sdf.actor* is build-in in SYLVA and it is used for generating one SDF actor.

```
1  def create_sdf_actor(name = 'sink', index = 0, data_count = 1) :
2
3    data_in = sdf.port( name = 'data_in', index = 0,
4                        type = sdf.data_type('bit', 1),
5                        count = data_count)
6
7    return sdf.actor( name = name, index = index,
8                      input_ports = [data_in] )
```

Code 1: SDF Actor Creation Example

In SYLVA, the parameters to the SDF actor creation function can be anything. For example, the parameters can be as simple as the function name and the data count as in the previous example. The parameters can also be scenario-aware parameters that allow using the same SDF actor creation method to generate multiple similar FIMPs as we mentioned earlier in this section.

For example, let us consider a FIMP for Fast Fourier Transform (FFT) is modeled in VHDL with generic parameters *point* and *data_width*. This SDF actor creation method example is shown in Code 2. In line 4 and line 7, the method *complex_type* generates the VHDL data type *complex* with a provided data width. The *complex* data type is defined by two *sfixed* numbers with provided data width and their decimal point is before the first bit. More information about *sfixed* can be found at [102]. The method *complex_type* is a build-in method in SYLVA.

```
1  def fir(tap, data_width):
2
3    input_ports = [ sdf.port(name = 'din',
4      index = 0, type = complex_type(data_width), count = 1)]
5
6    output_ports = [ sdf.port(name = 'dout',
7      index = 0, type = complex_type(data_width), count = 1) ]
8
9    return sdf.actor( name = 'fir', input_ports = input_ports,
10                     output_ports = output_ports )
```

Code 2: Finite Impulse Response (FIR) SDF Actor Creation Example

Other SDF actor creation examples with more details will be given later on in this section along with the elaboration of the modeling procedure. We will model the uplink transmitter in the physical layer of Long Term Evolution (LTE) using a complex and parametric SDF graph as the example. After we have the SDF actor creation method, we store it in the FIMP library so we can reuse this function for other designs. When invoking SDF actor creation method, all the parameters should be presented to the corresponding FIMP, otherwise the absent parameters carry no meaning for the FIMP. For example, we cannot specify the port width of an VHDL design unit unless there is a generic parameter for it.

4.4.2 SDF Edge Creation

Creating SDF edges is to connect the SDF actors based on the data dependency to construct the system. It is easier to create an SDF edge than to create an SDF actor. However, the SDF edges can be only created manually while the SDF actors can be created by using the creation methods, since they represent the system structure (data/control flow). Before creating any SDF edges we should have all the SDF actors that will be used in the system. An SDF edge can also be created using a **connect** method as shown in Code 3. It also can be created using the generalized construction method shown in Code 4 by passing the references of the source actor, source port, destination actor, and destination port to it. When two SDF actors are connected via multiple ports, we have to create one SDF edge for each individual port. For example, if we want to connect a data source with the complex data output in two different ports (*d_re* for the real component and *d_im* for the imaginary component) to a data sink, we need to create one SDF edge for *d_re* and another SDF edge for *d_im*.

```
1 def connect(src_index, src_port_index,
2   dest_index, dest_port_index, actors) :
3
4   src_actor = actors[src_index]
5   src_port = src_actor.output_ports[src_port_index]
6   dest_actor = actors[dest_index]
7   dest_port = dest_actor.input_ports[dest_port_index]
8
9   return sdf.edge(src_actor, src_port, dest_actor, dest_port)
```

Code 3: Connecting Two SDF Actors

```
class edge :

  def __init__(self,
    src_actor = None, src_port = None,
    dest_actor = None, dest_port = None) :

    self.src_actor = src_actor
    self.dest_actor = dest_actor
    self.src_port = src_port
    self.dest_port = dest_port

    if type(src_actor) == type(actor()) :
      src_actor.next.append(self)
    if type(dest_actor) == type(actor()) :
      dest_actor.previous.append(self)

  def __str__(self) :
    return 'SDF edge \n  from %s (%s) \n  to %s (%s)' \
         % (self.src_actor, self.src_port, \
            self.dest_actor, self.dest_port )

  def serialize(self) :
    return { 'src_actor' : self.src_actor.index,
             'dest_actor' : self.dest_actor.index,
             'src_port' : self.src_port.index,
             'dest_port' : self.dest_port.index }
```

Code 4: SDF Edge Class (*edge*)

4.5 System Modeling Example

In this section, an example of modeling the Long Term Evolution (LTE) uplink transmitter will be given to clarify the flow of the system modeling in System-Level Architectural Synthesis Framework (SYLVA). The block diagram of the LTE uplink transmitter is shown in Figure 4.8. The functions and the data dependency are the starting point for creating the Synchronous Data Flow (SDF) graph. Each of the functions in Figure 4.8 will be modeled by one SDF actor. For Application-Specific Integrated Circuit (ASIC) and Field-Programmable Gate Array (FPGA), SYLVA assumes that every Function Implementation (FIMP) has a clock signal *clk* and an active low reset signal *nrst*. For Coarse-Grained Reconfigurable Architecture (CGRA), SYLVA assumes that every FIMP has an active low reset signal *nrst*. Those signals are not considered as data inputs to the SDF actors and will not be considered in the SYLVA design flow till the code generation phase.

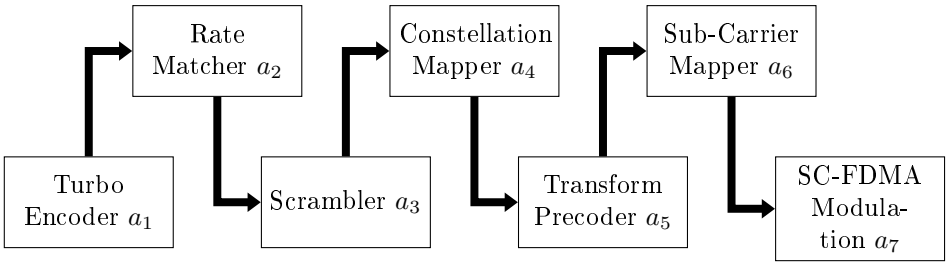


Figure 4.8: LTE Uplink Transmitter Block Diagram

The entire system can then be modeled as a flexible model (Figure 4.9) or as a dedicated model (Figure 4.10), which is for 64-QAM and $R = r = 1/3$. The dedicated system can be automatically generated based on the given parameters by constructing the SDF actors using input parameters. The detailed information about this example can be found in the appendix.

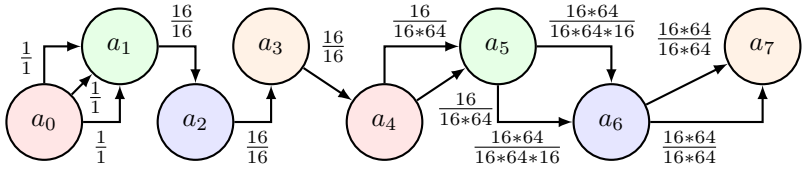


Figure 4.9: LTE Uplink Transmitter SDF Graph (Flexible)

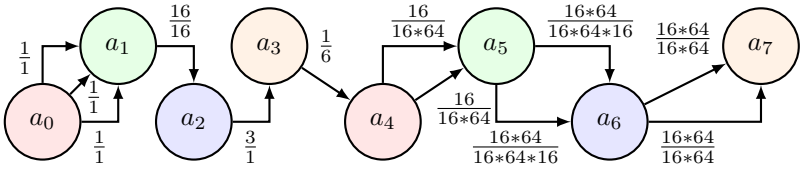


Figure 4.10: LTE Uplink Transmitter SDF Graph (Dedicated)

4.6 Summary

This chapter gives the definitions of Synchronous Data Flow (SDF) graph and Homogeneous SDF (HSDF) graph in the context of System-Level Architectural Synthesis Framework (SYLVA), the definition of Function Implementation (FIMP), as well as the method for modeling Digital Signal Processing (DSP) sub-systems using an SDF graph with constraints and optimization objectives.

Chapter 5

Design Space Exploration

The design space that System-Level Architectural Synthesis Framework (SYLVA) explores has three dimensions - Function-Level Parallelism (FLP), Arithmetic-Level Parallelism (ALP), and Buffer-Level Parallelism (BLP). In practice, the last two dimensions are folded into one. The Design Space Exploration (DSE) flow is shown in Figure 5.1 (part of the SYLVA design flow shown in Figure 1.5, section 1.3). The input to the DSE is the system Synchronous Data Flow (SDF) graph (section 4.1). The output from the DSE consists of the cycle accurate schedule and the number and types of the Function Implementations (FIMPs) (section 4.3) to be deployed.

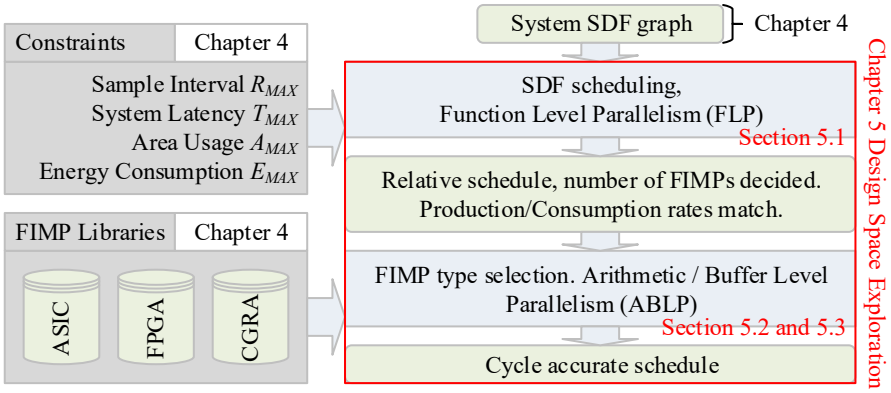


Figure 5.1: SYLVA Design Flow

The first three sections in this chapter introduce the concepts of FLP, ALP, and BLP that are corresponding to the three dimensions. Then, in section 5.4 we will quantify the design space by using these concepts. After that, section 5.5 will elaborate the DSE implementation in SYLVA.

5.1 Function-Level Parallelism

The first dimension of the design space explored by System-Level Architectural Synthesis Framework (SYLVA) is spanned by the number of Function Implementations (FIMPs) and is called Function-Level Parallelism (FLP), because functions that are not data dependent can be executed in parallel, a form of data-level parallelism. This dimension is explored by the variations in the feasible Synchronous Data Flow (SDF) schedules. Figure 5.2 shows an example SDF graph and some possible variations in its SDF schedules.

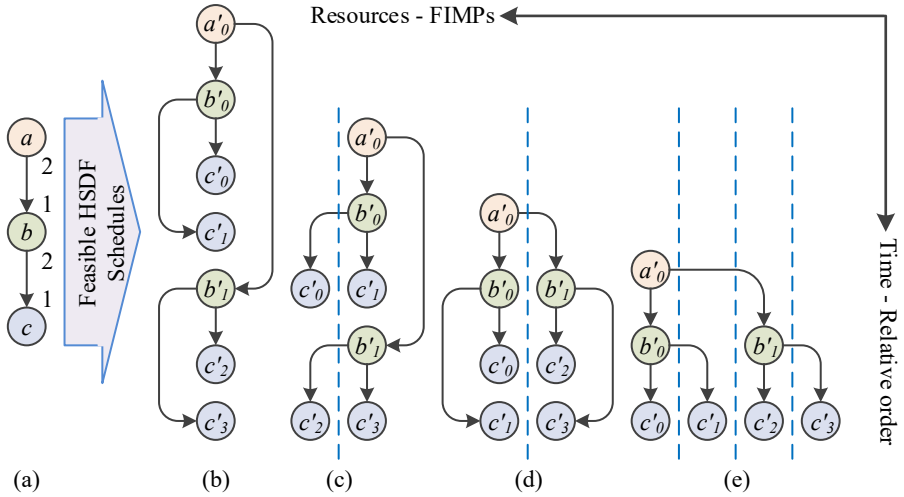


Figure 5.2: SDF Schedules

In all the shown schedules, the function b is fired twice for each firing of the function a; a produces two tokens every time it is fired and b consumes one token every time it is fired. The most parallel schedule in Figure 5.2(d) has two executions of function b and four executions of function c are in parallel. This implies that this schedule requires two FIMPs for function b and four FIMPs for function c. In contrast, the schedule in Figure 5.2(a) is serial and requires only one FIMP for each function. As the number of feasible schedules grow exponentially with the number of functions and the number of times each function is fired, the problem becomes intractable (as will be elaborated in section 5.4) and requires using a heuristic to consider a subset as elaborated in section 5.5. Exploring parallelism in this dimension is a key feature of SYLVA that distinguishes it from the High-Level Synthesis (HLS) tools as is discussed in chapter 3.

5.2 Arithmetic-Level Parallelism

The second dimension of the design space that System-Level Architectural Synthesis Framework (SYLVA) explores is the degree of parallelism within each Function Implementation (FIMP) controlled by the number of arithmetic units. e.g., Finite Impulse Response (FIR) filter FIMPs could vary in the number of Multiplication-Accumulations (MACs) they have. This dimension is called the Arithmetic-Level Parallelism (ALP). This is also a form of data-level parallelism, albeit at a finer granularity level compared to the Function-Level Parallelism (FLP). Figure 5.2 shows that a 16-tap FIR filter FIMP may have 1, 2, 4, or 8 MACs resulting in different area usage and latencies.

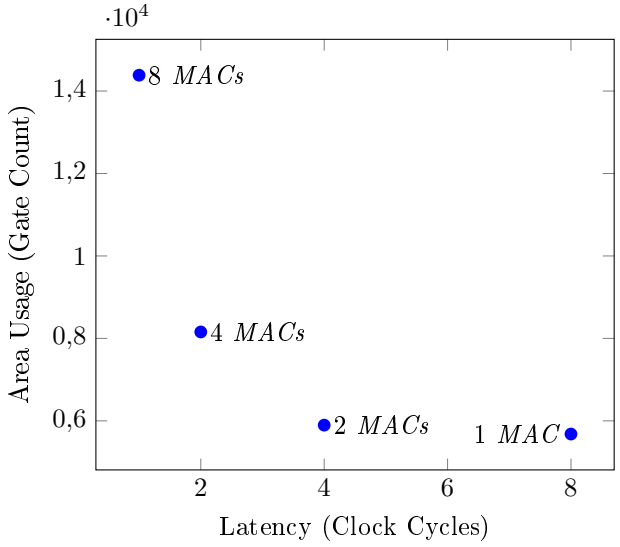


Figure 5.3: Arithmetic-Level Parallelism Example

In most of the cases, decreasing the number of arithmetic units will decrease the area usage but increase the latency. However, in some cases, decreasing the number of arithmetic units could result in more buffer and control logic resulting more total area usage and also increased latency. One such example is when in an FIR the number of MACs is decreased, it could result in increased storage for intermediate results and increased control logic. Figure 5.2 shows a concrete case for FIR, where decreasing the number of MACs from 8 to 4 dramatically decreases the area usage but decreasing the number of MACs from 2 to 1 almost has no impact on the area usage.

5.3 Buffer-Level Parallelism

The third dimension of the design space that SYLVA explores is the parallelism enabled by pipelining the producing and consuming Function Implementations (FIMPs) using multiple buffers; this dimension is called the Buffer-Level Parallelism (BLP). FIMPs internally are almost always pipelined; the pipelining being discussed here is the inter-FIMP and not intra-FIMP. By default, the producing and the consuming FIMPs share a single buffer (the output buffer in the producing FIMP), thereby serializing the execution of the two FIMPs; the producing FIMP cannot continue executing until the consuming FIMP has read all the data from the output data buffer. However, if a FIMP is instantiated with extra output buffer(s), the producing and the consuming FIMPs can execute in a pipelined fashion and therefore reduce the unnecessary FIMP idle time.

This is shown in Figure 5.4(b) and (c). Figure 5.4(b) shows how reading of shared buffer by the consuming FIMP delays the next firing of producing FIMP. However, addition of an extra buffer allows the producing FIMP to start outputting to second buffer while the consuming FIMP is still reading from the first buffer. This can be generalized to more that two buffers. To help the reader with the notation, we have repeated the FIMP execution timing model that was introduced in section 4.3.1. We have also annotated the the schedules for both the bases with expressions using notation that are specified in Figure 5.4.

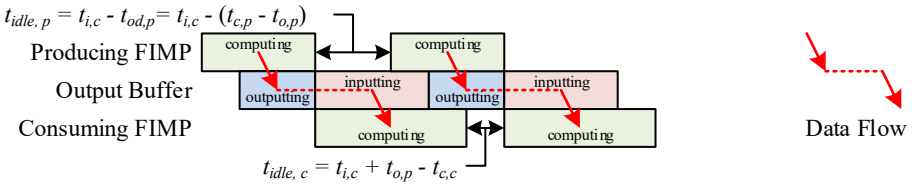
(a) Producing and Consuming FIMPs Share Output Buffer

$t_{idle,\,p}$: idle time of the producing FIMP $t_{c,p}$: computation phase length of the producing FIMP

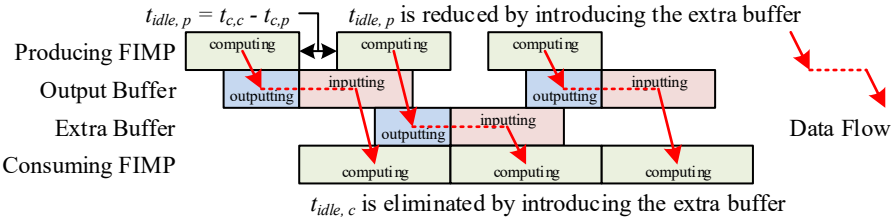
$t_{od,p}$: Output delay of the producing FIMP $t_{o,p}$: output phase length of the producing FIMP

$t_{idle,\,c}$: Consuming FIMP idle time $t_{c,c}$: computation phase length of the consuming FIMP

$t_{i,c}$: Input phase length of the consuming FIMP



(b) FIMP Execution Schedule without Extra Buffer



(c) FIMP Execution Schedule with Extra Buffer

Figure 5.4: Effectiveness of Extra Buffer

5.4 The Design Space

In this section, we quantify the design space that System-Level Architectural Synthesis Framework (SYLVA) explores. We begin by giving an intuitive quantification of the design space and later we present a more precise and rigorous quantification that requires some elaborate notations. As stated earlier, the design space that SYLVA explores has three dimensions - function level parallelism, arithmetic level parallelism, and buffer level parallelism. The intuitive and approximate design space quantification is summarized in Figure 5.5. The quantification of the value of $|S|$ and the exact size of the Arithmetic-Level Parallelism (ALP) design space will be elaborated later in this section. The elaborations will be divided into three subsections focusing the contributions to the design space by the Function Implementation (FIMP) instance set, the Homogeneous SDF (HSDF) actor to FIMP assignment, and the output buffer.

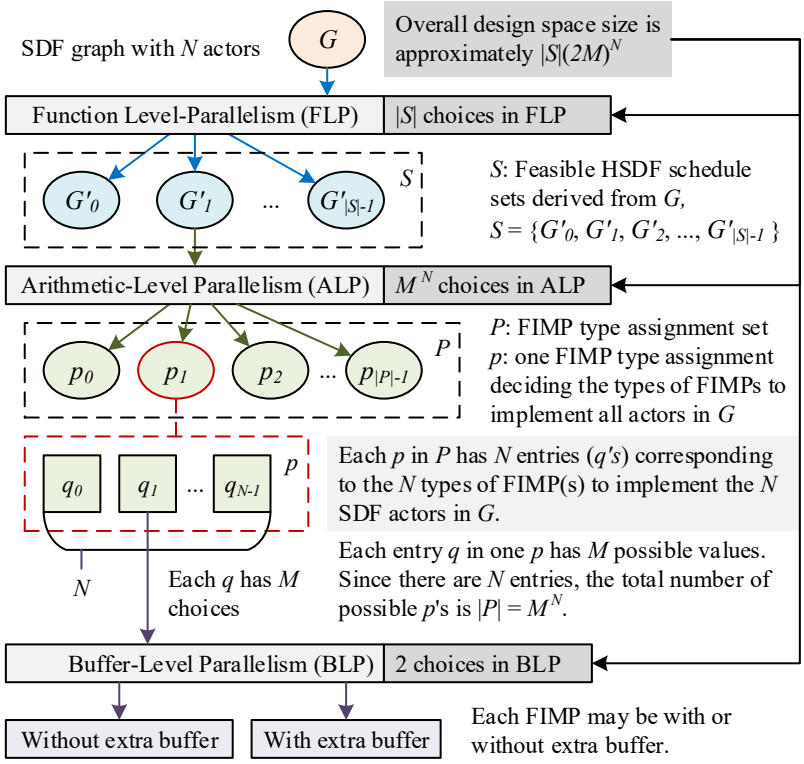
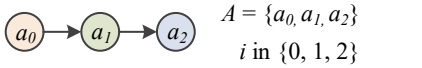


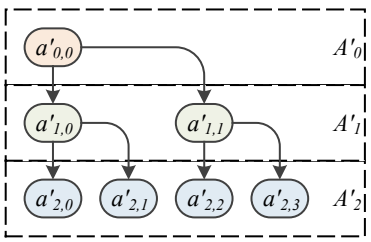
Figure 5.5: SYLVA Design Space

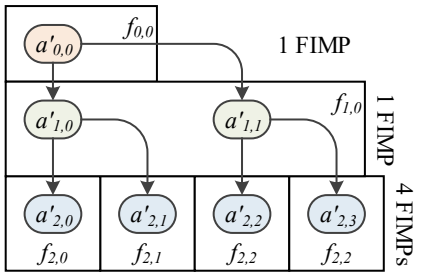
Before we elaborate the details of these components, let us review the notations in Table 5.1 and Table 5.2. The tick in A' and a' indicates that it is in an HSDF graph. i is to index one element among $|A|$ elements. k denotes the index of one HSDF actor among all actors form the same Synchronous Data Flow (SDF) actor. m indexes one FIMP among all FIMPs to implement the HSDF actors from the same SDF actor.

Table 5.1: Actor and FIMP Notations

Notation	Description
A	SDF actor set in the system, $\|A\| = N$
a_i	The ith SDF actor in A
A'	HSDF actor set derived from A
A'_i	HSDF actor set derived from a_i
	$A' = \bigcup_{i=0}^{i<\|A\|} A'_i$ e.g. $A' = \{A'_0, A'_1, A'_2\}$
$a'_{i,k}$	The kth HSDF actor in A'_i, $0 \le k < \|A'_i\|$
	e.g. $A'_0 = \{a'_{0,0}\}$, $A'_1 = \{a'_{1,0}\}$
	$A'_2 = \{a'_{2,0}, a'_{2,0}, a'_{2,0}, a'_{2,0}\}$
F	FIMP instance set to be deployed
F_i	FIMP instance set to be deployed for A'_i
$f_{i,m}$	The mth FIMP instance in F_i, $0 \le m < \|F_i\|$
	$F = \bigcup_{i=0}^{i<\|A\|} F_i$ $F_i = \bigcup_{m=0}^{m<\|F_i\|} f_{i,m}$
	e.g. $F = \{F_0, F_1, F_2\}$, $F_0 = \{f_{0,0}\}$, $F_1 = \{f_{1,0}\}$, $F_2 = \{f_{2,0}, f_{2,1}, f_{2,2}, f_{2,3}\}$
$A'_{i,m}$	The list of HSDF actors executed by $f_{i,m}$
	e.g. $A'_{1,0} = \{a'_{1,0}, a'_{1,1}\}$
m_i	The number of possible FIMP types for implementing a'_i or A'_i. m_i is provided by the FIMP library.
q_i	FIMP type index of all FIMPs in F_i. The possible values of q_i are decided by m_i.
	If actor a_i is an FFT then m_i is the number of FFT types in the library, e.g. radix-2, radix-4, etc. q_i is the index of the deployed FIMP in the FIMP type set. $0 \le q_i < m_i$







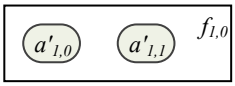
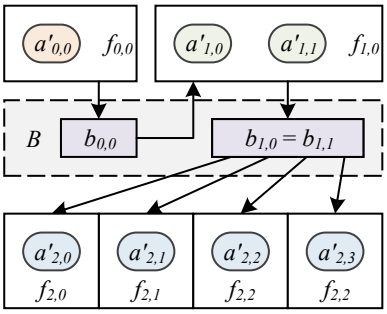
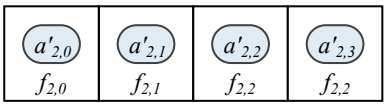


Table 5.2: Buffer and Assignment Notations

Notation	Description
B	Set of buffer instances to be deployed
$b_{i,k}$	Buffer instance set to be deployed for $a'_{i,k}$
	e.g. $B = \{b_{0,0}, b_{1,0}, b_{1,1}\}$
Z	Extra buffer decision set for all FIMPs in F
z_i	Extra buffer decision for all FIMPs in F_i
	e.g. $Z = \{ z_{0,0}, z_{1,0}, z_{1,1} \}$ $z_1 = false \rightarrow b_{1,0} = b_{1,1}$ $F_{1,0}$ has no extra buffer
	When the DSE make a decision to the extra buffer to pipeline the execution of producing and consuming FIMPs, if denote this decision in Boolean variable $z_{i,m}$ corresponding to the FIMP instance $f_{i,m}$ that should have extra buffer.
$M_{A' \rightarrow F}$	The assignment of HSDF actor to FIMP instance map ($A' \rightarrow F$) *e.g.* $A'_{2,0} \rightarrow f_{2,0}$ $A'_{2,2} \rightarrow f_{2,2}$ $A'_{2,1} \rightarrow f_{2,1}$ $A'_{2,3} \rightarrow f_{2,3}$





5.4.1 FIMP Instance Set

In the rest of this thesis, we represent the list of FIMP instances as F. It has the information of the number and the types of the deployed FIMP instances in the system implementation.

Number of FIMP Instances

Denote the total number of the deployed FIMP instances for the given system as $|F|$. It should meet the condition in Eq. 5.1, where $|A|$ is the number of SDF actors in the system and $|A'|$ is the number of the HSDF actors derived from A.

$$|A| \leq |F| \leq |A'| \tag{5.1}$$

As elaborated in section 4.3, each FIMP instance is able to execute one or more HSDF actors that are derived from the same SDF actor. Therefore, the lower bound $|A|$ in Eq. 5.1 happens when each SDF actor is implemented by one FIMP, while the upper bound $|A'|$ happens when each HSDF actor is implemented by one FIMP.

Note that even through two system implementations have the same $|F|$, the SDF actor to FIMP assignment may be different. For an SDF actor $a_i \in A$, denote the FIMP instance set to execute it as F_i. The number of the deployed FIMP instances $|F_i|$ should meet the condition in Eq. 5.2, where $|A'_i|$ is the number of HSDF actors that are derived from a_i. $|A'_i|$ also indicates the number of executions of a_i in one system iteration according to the definition of HSDF actor in section 4.2 and $|A'| = \sum_{i=0}^{i<|A|} |A'_i|$.

$$1 \leq |F_i| \leq |A'_i| \tag{5.2}$$

Therefore, the combinations of the $|F_i|$ to a_i assignment $C_{|F|}$ can be computed by Eq. 5.3. $C_{|F|}$ contributes to the design space in the Function-Level Parallelism (FLP) that was elaborated in section 5.1.

$$C_{|F|} = \prod_{i=0}^{i<|A|} |A'_i| \tag{5.3}$$

For the example shown in Figure 3.1, the system has 3 SDF actors and the corresponding HSDF graphs have 7 actors. The total number of FIMP instances can range from 3 to 7. The SDF actor a_0 can be implemented by one FIMP instance, The SDF actor a_1 can be implemented by 1 or 2 FIMP instances, and the SDF actor a_2 can be implemented by 1, 2, 3, or 4 FIMP instances. Then $C_{|F|}$ is $1 \times 2 \times 4 = 8$.

Types of FIMP Instances

For a given SDF actor a_i, we denote the total number of FIMPs that can implement it as m_i. The value of m_i is defined by the FIMP library. Therefore, the total number of the possible FIMP type combinations C_Q can be computed by Eq. 5.4. C_Q contributes to the design space in the ALP that was explained in section 5.2.

$$C_Q = \prod_{i=0}^{i<|A|} m_i^{|A'_i|} \tag{5.4}$$

Design Space Contribution

Combining the number and the types of FIMP instances, we can derived the total number of possible FIMP instances $|F|$. C_F or the overall contribution to the design space size by the FIMP instance set F is shown in Eq. 5.5.

$$\begin{aligned} C_F &= C_{|F|} \cdot C_Q \\ &= \prod_{i=0}^{i<|A|} |A'_i| \cdot \prod_{i=0}^{i<|A|} m_i^{|A'_i|} \\ &= \prod_{i=0}^{i<|A|} \left(|A'_i| \cdot m_i^{|A'_i|} \right) \end{aligned} \tag{5.5}$$

The value of C_F increases dramatically with $|A|$, m_i, and $|A'_i|$. Assume that all $m_i = |A| = |A'_i| = 5$, C_F will be a very big number as computed below.

$$\prod_{i=0}^{i<5} \left(5 \cdot 5^5\right) = (5 \cdot 5^5)^5 \approx 9.3 \times 10^{20}$$

Design Space Reduction

We reduce the design space by assuming all the FIMP instances for the same SDF actor are the same type q. This simplification reduces the design space significantly because the exponential part of $m_i^{|A'_i|}$ is eliminated and will not affect the quality of the solution as proved below.

Consider one SDF actor a_i. Assume one of the FIMP instances $f_{i,m}$ for implementing $A'_{i,m} \in A'_i$ has a different type than other FIMP instances in F_i that implement A'_i. The FIMP $f_{i,m}$ will have a different timing performance and/or cost (area and energy) than other FIMPs. If other FIMP instances in F_i are already Pareto optimal then a better timing performance will lead to a drop in performance in either area or energy cost. We then have two possibilities:

1. If the timing performance of $f_{i,m}$ is better than other FIMP instances in F_i, it should wait for others leaving the final timing performance unchanged. In this case, the area usage or the energy consumption of $f_{i,m}$ will be larger than other FIMP instances in F_i but contributing nothing on the timing performance of the system. Therefore, the solution with different FIMP types is worse than the one with a single FIMP type.

2. If the timing performance of $f_{i,m}$ is worse than other FIMP instances in F_i, other FIMP instances for implementing the same SDF actor with more cost should wait for it and leaving the final timing performance unchanged. In this case, the solution with different FIMP types is also worse than the one with a single FIMP type.

By applying the simplification, C_F will be reduced significantly as in Eq. 5.6.

$$C_F = \prod_{i=0}^{i<|A|} (|A'_i| \cdot m_i) \tag{5.6}$$

If all $m_i = |A| = |A'_i| = 5$, C_F will be as follows.

$$\prod_{i=0}^{i<5} (5 \cdot 5) = (5 \cdot 5)^5 = 9765625$$

5.4.2 HSDF Actor to FIMP Assignment

The HSDF actor to FIMP assignment is denoted as $M_{A' \to F}$ and assigns a sequence of HSDF actors to one FIMP. For each HSDF actor in A'_i, we have $|F_i|$ choices.

$$M = \{A'_{i,m} \ \rightarrow \ f_{i,m} | 0 \leq i < |A|, 0 \leq m < |F_i|\} \tag{5.7}$$

The total number of possible HSDF actor to FIMP instance maps (C_M) can be expressed by Eq. 5.8.

$$C_M = \prod_{i=0}^{i<|A|} |F_i|! \tag{5.8}$$

To reduce the design space, we assume that all the HSDF actors in A'_i are distributed on F_i in a fixed order from $a_{i,0}$ to $a_{i,|A'_i|-1}$ such that the permutation part $|F_i|!$ is eliminated to 1. After reduction, $C_M = 1$. Currently, SYLVA supports two fix orders: interleaving and linear. The HSDF actor set $A'_{i,m}$ to be executed on the m^{th} FIMP for a_i can be obtained by Eq. 5.9.

$$A'_{i,m} = \begin{cases} [a'_{i,(m \cdot |F_i|)+y}] & \text{linear} \\ [a'_{i,(y \cdot |F_i|)+m}] & \text{interleaving} \end{cases} 0 \leq y < \lceil \frac{|A'_i|}{|F_i|} \rceil \tag{5.9}$$

If $|A'_i| = 8$ and $|F_i| = 4$, the HSDF actor to FIMP assignment will be as follows.

interleaving		linear	
$[a'_{i,0}, a'_{i,4}]$	$\rightarrow \ f_{i,0}$	$[a'_{i,0}, a'_{i,1}]$	$\rightarrow \ f_{i,0}$
$[a'_{i,1}, a'_{i,5}]$	$\rightarrow \ f_{i,1}$	$[a'_{i,2}, a'_{i,3}]$	$\rightarrow \ f_{i,1}$
$[a'_{i,2}, a'_{i,6}]$	$\rightarrow \ f_{i,2}$	$[a'_{i,4}, a'_{i,5}]$	$\rightarrow \ f_{i,2}$
$[a'_{i,3}, a'_{i,7}]$	$\rightarrow \ f_{i,3}$	$[a'_{i,6}, a'_{i,7}]$	$\rightarrow \ f_{i,3}$

The HSDF actor to FIMP assignment does not contribute any thing to the design space with this reduction. The decision of using either interleaving or linear distribution is defined by user. Other distribution may improve the system performance/cost but the design space will be too large to search. Covering more design space is considered as a future work of SYLVA. The number of possible HSDF schedules $|S| = C_{|F|} \cdot C_M$ since one HSDF schedule is the number of FIMPs together with the sequence of HSDF actors to be executed on each FIMP. After reduction, $|S| = C_{|F|}$ and this will be the total number of degrees in the FLP.

5.4.3 Output Buffers

One FIMP may have different numbers of output buffers according to whether an extra buffer is used. The total number of output buffers as $|B|$ depends on the number of FIMP instances $|F|$ and whether extra buffer is used (Z). For the SDF actor a_i, $z_i \in Z$ $(0 \leq i < |A|)$ is a boolean value indicating extra buffer is used or not. Similar to the FIMP type, all the FIMPs that implement one SDF actor have the same decision. The total number of possible combinations of output buffer count ($C_{|B|}$) can be computed by Eq. 5.10.

$$C_{|B|} = \prod_{i=0}^{i<|A|} 2 = 2^{|A|} \tag{5.10}$$

5.4.4 Overall Design Space Size

By combining the contributions by the FIMP instance, HSDF to FIMP assignment, and the output buffers, we can derive the size of the design space $|DS|$ that SYLVA explores. For each SDF actor a_i, we need to first decide the number of FIMP instances $|F_i|$, then decide the type of FIMP instances q_i $(0 \leq q_i < m_i)$. This FIMP type applies for all HSDF actors in A'_i. We then decide the HSDF actor to FIMP assignment (the design space is reduced to 1). Finally, we decide whether extra buffer is used for each F_i. The size of the design space $|DS|$ can be expressed by Eq. 5.11.

$$|DS| = 2^{|A|} \cdot \prod_{i=0}^{i<|A|} (|A'_i| \cdot 1 \cdot m_i) \tag{5.11}$$

The term $2^{|A|}$ is the contribution by the output buffers (C_B, Eq. 5.10). This decision is independent on other contributors and reflects the Buffer-Level Parallelism (BLP) that was described in section 5.3. The term $|A'_i|$ is the contribution of the number of FIMPs ($C_{|F|}$, Eq. 5.1). The constant 1 is the contribution of HSDF actor to FIMP assignment (C_M, Eq. 5.8). $C_{|F|}$ and C_M reflect the FLP as explained in section 5.1. The term m_i is the contribution by the number FIMP types (C_Q, Eq. 5.4) that reflects the ALP that was elaborated in section 5.2.

5.5 Exploration

This section addresses the Design Space Exploration (DSE) algorithm used by System-Level Architectural Synthesis Framework (SYLVA). The algorithm is shown in Algorithm 5.1. It first explores the Function-Level Parallelism (FLP), then explores the Arithmetic-Level Parallelism (ALP) and the Buffer-Level Parallelism (BLP). The term CSOP is short for Constraint Satisfaction Optimization Problem. Google's OR-Tools [103] is used as the Constraint Programming (CP) solver since this tool has a good performance in MiniZinc Challenges [104] and it has many wraps to other programming languages including python and C# that are used in this thesis.

```
1  S: Automatically produced subset of all feasible schedules
2  User defines |S| (the number of schedules in S)
3  foreach s ∈ S do
4  |   Create and solve CSOP for s
5  |   Record all costs [R_sys, T_sys, A_sys, E_sys] that meet the constraints
6  |   R_sys is the sample interval
7  |   T_sys is the system latency
8  |   A_sys is the area usage
9  |   E_sys is the energy consumption of one system iteration
10 end
11 Find the optimal schedule s based on the given constraints
12 Read out the system cost value of s
```

Algorithm 5.1: Overall Design Space Exploration Strategy in SYLVA

The design space of FLP is explored by finding all cost matrices of every schedule s in S, where the value of $|S|$ is provided by the user and a heuristic algorithm is used to create S. The loop body in Algorithm 5.1 explores the arithmetic and buffer-level parallelism dimension for each s and finds all cost matrices for the given s by automatically constructing an CSOP and solving it using a CP solver [103]. More details will be elaborated in the later subsections.

5.5.1 Exploring Function-Level Parallelism

Since the number of possible schedules equals to $|C_{|F|}$ (Eq. 5.3) as elaborated in the previous section, covering all of them will lead to a very high and unacceptable synthesis time. To cope with the intractability of exhaustively considering all feasible Homogeneous SDF (HSDF) schedules, we first list all the load balanced schedules and then pick a subset $|S|$ of them. The schedules in S are as evenly distributed as possible between the two extremes: fully serial Synchronous Data Flow (SDF) schedule (Figure 5.2b) and fully parallel SDF schedule (Figure 5.2e). The load balanced schedules are better than other schedules. This will be elaborate in detail later in this subsection. When exploring the FLP, only $|S|$ schedules are explored to reduce the design space in function level.

Load Balanced Schedule

An HSDF schedule is load balanced if and only if the number of HSDF actors that are derived from the same SDF actor and executed by the Function Implementation (FIMP) instances with the same FIMP type are as equal as possible. This definition is formulated by Eq. 5.12. For a given SDF actor a_i, A'_i is the HSDF actor set derived from a_i. For a given set of FIMP instances F_i to implement A'_i, each FIMP instance $f_{i,m}$ implements a set of HSDF actors $A'_{i,m} \in A'_i$, where $0 \leq m < |F_i|$. More detailed description of these notations can be found in Table 5.1.

$$\left| |A'_{i,m}| - \frac{|A'_i|}{|F_i|} \right| \leq 1 \qquad \forall \quad 0 \leq i < |A| \ , \ 0 \leq m < |F_i| \tag{5.12}$$

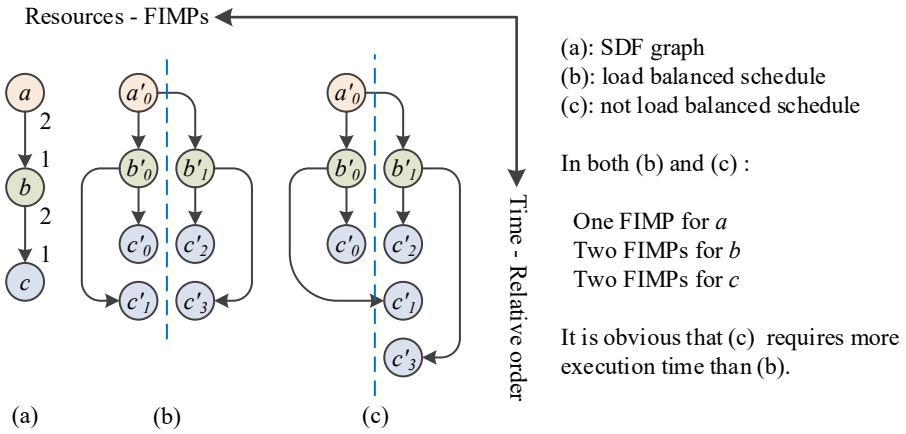


Figure 5.6: Load Balanced Schedule Example

The load balanced schedule will lead to the best performance for a given set of FIMP instances as shown in Figure 5.6. Assume we have a set of FIMP instances F_i for implementing SDF actor a_i. The latency of executing one FIMP instance for a_i is its computation phase length $t_{c,i}$. Denoting the overall latency of a_i as T_i, it can be derived by 5.13 for a load balanced schedule. The term $\lceil |A'_i|/|F_i| \rceil$ is the number of executions of each FIMP instance in one system iteration.

$$T_i = \left\lceil \frac{|A'_i|}{|F_i|} \right\rceil \cdot t_{c,i} \tag{5.13}$$

If the FIMP instances are not load balanced (Eq. 5.12 is not met), at least one FIMP instance will executes more than $\lceil |A'_i|/|F_i| \rceil$ and this will result a higher overall latency T_i. Assume $|A'_i = 5|$ (5 HSDF actors in A'_i) and $|F_i| = 2$ (2 FIMP instances to implement a_i). The best case is that one FIMP instance executes twice and the other one executes three times. Any other possibilities (1 and 4 times, or 0 and 5 times) will result a larger $\lceil |A'_i|/|F_i| \rceil$. Since we only want the optimal solution(s), SYLVA only considers the load balanced schedules.

By considering only the load balanced schedules, we could derive that once the number of FIMP instances ($|F_i| \mid \forall a_i \in A$) are fixed, the timing performance ($T_i \mid \forall a_i \in A$) will only dependent on the FIMP types since, in 5.13, $\lceil |A'_i|/|F_i| \rceil$ is fixed and $t_{c,i}$ is determined by the FIMP type. Therefore, the number of FIMP instances ($|F_i| \mid \forall a_i \in A$) can be used to represent the FLP. We can can generate the load balanced schedule by determining the HSDF actor to FIMP instance assignment M. The algorithm for generating all feasible schedules S_{all} (S in Algorithm 5.1 is a subset of S_{all}) is shown in Algorithm 5.2.

```
1  A is the actor set of the system SDF graph
2  F is the FIMP instance set for implementing A
3  F_i is the set of FIMP instances for a_i | ∀0 ≤ i < |A|
4  F_MAX is the set of maximum numbers of FIMP instances used for A
5  foreach SDF actor a_i ∈ A do
6  |  F_MAX,i = q_ai
7  |  q_ai is the number of repetitions of a_i, see section 2.1 for more details
8  end
9  foreach SDF actor a_i ∈ A do
10 |  V_i = possible_values(F_MAX,i) foreach v ∈ V_i) do
11 |  |  |F_i| = v {|F_i| | ∀a_i ∈ A} = modify_FIMP_count(i, v)
12 |  |  M = assign_HSDF_actors()
13 |  |  Save the schedule s = ({|F_i| | ∀a_i ∈ A}, M) to S_all
14 |  end
15 end
```

Algorithm 5.2: Load Balanced Schedule Generation

possible_values($F_{MAX,i}$): This method is for producing the feasible numbers of FIMP instances that could be used for constructing load balanced schedules based on the given maximum possible number of FIMP instances $F_{MAX,i}$. The produced numbers are the integer dividers of $F_{MAX,i}$. Initially, the value should be the repetition vector q. According to the definition of HSDF schedule, $q_i = |A'_i|$. Assume the produced number set is V_i, where $\forall v \in V_i$, $q_i \bmod v = 0$. Therefore, the HSDF actors can be evenly distributed on the FIMP instances and $||A'_{i,m}| - |A'_i|/|F_i||$ can be 0 or 1 (Eq. 5.12). For example, if $q_i = 8$, the resulting V_i will be [1, 2, 4, 8] and $|A'_{i,m}|$ will be [8, 4, 2, 1], respectively.

modify_FIMPs(i, v): This method is for modifying the number of FIMP instances to implement the system. Based on the number of FIMP instances v for implementing SDF actor a_i, this method will modify the $F_{MAX,i}$ for all SDF actors. The algorithm implemented by this method is shown in Algorithm 5.3.

```
1 i: the index of SDF actor
2 v: the number of FIMP instances to implement SDF actor a_i
3 if |F_i| > v then
4 |   r = |F_i|/v
5 |   foreach SDF actor a_i ∈ A do
6 |   |   |F_i| = ⌈|F_i|/r⌉
7 |   end
8 end
```

Algorithm 5.3: Modify FIMPs

assign_HSDF_actors(): This method is for assigning HSDF actors evenly to the FIMP instances. The assigning can be either in linear (default) or interleaving fashion. More details can be found in section 5.4.2.

Since S_{all} may be quite large for a large design, the user provided parameter $|S|$ can be used to further reduce the design space in FLP. We expect $|S|$ to be in tens on the lower side to a few thousands on the higher side. In spirit, $|S|$ is similar to the typical high, medium and low effort parameters in commercial logic and physical synthesis flows. The key idea is to start with the most parallel schedule by converting the input SDF into the corresponding HSDF graph, using the algorithm presented in [11] that also creates a relative order of the functions. The advantage of using the HSDF representation is that it explicitly contains the invocations of all functions that can be run in parallel and also the data dependencies among them. Next, the algorithm shown in Algorithm 5.4 creates $|S|$ SDF schedules that are near evenly spaced in terms of their parallelism. The outcome of the algorithm in Algorithm 5.4 is to create a feasible schedule subset S in the design space exploration flow shown in Algorithm 5.1. Algorithm 5.4 is similar to Algorithm 5.2. The former considers only one SDF actor, while the latter considers all SDF actor. Each schedule in S is unique and varies in its FLP like the schedules shown in Figure 5.2b-e.

1 A is the actor set of the system SDF graph
2 F is the FIMP instance set for implementing A
3 Find $a_i \in A$ with the maximum repetitions ($q_i = \max(q)$)
4 F_i is the set of FIMP instances for a_i
5 V_i = possible_values(q_i)
6 V_i' = evenly picked $|S|$ values in V_i
7 **foreach** $v \in V_i'$) **do**
8 | $|F_i| = v$ $\{|F_j| \mid \forall a_j \in A\}$ = modify_FIMP_count(i, v)
9 | M = assign_HSDF_actors()
10 | Save the schedule $s = (\{|F_j| \mid \forall a_j \in A\}, M)$ to S
11 **end**

Algorithm 5.4: Subset of Load Balanced Schedule Generation

5.5.2 Exploring Arithmetic and Buffer-Level Parallelism

Each schedule that is created by Algorithm 5.4 in section 5.5.1 has ***a)*** the number of FIMPs instances for each function (SDF actor) and ***b)*** the HSDF actor to FIMP instance assignment (it is also a relative order in which the HSDF actors to be executed, see 5.4.2 for more details). In this section, we present how SYLVA explores the arithmetic and buffer-level parallelism dimensions for each such SDF schedule (a fixed degree of FLP).

As a result, SYLVA will decide the FIMP types based on a given FIMP library. Once the FIMP types are decided, it becomes possible to create a cycle accurate schedule and evaluate its system cost $[R_{sys}, T_{sys}, A_{sys}, E_{sys}]$ (R_{sys}: sample interval, T_{sys}: system latency, A_{sys}: system area usage, and E_{sys}: system energy consumption) and if it meets the constraints (e.g. maximum sampling rate R_{MAX} and/or total latency T_{MAX}). In terms of the design exploration flow shown in Algorithm 5.1, the method described in this section represents the inner loop.

SYLVA automatically create one CSOP and solve it using OR-Tools [103] from Google as the CP solver. as the CP solver. The created CSOP covers the FIMP type and the extra buffer selection. The inputs to and the outputs from the CP solver are shown in Figure 5.7 and elaborated below.

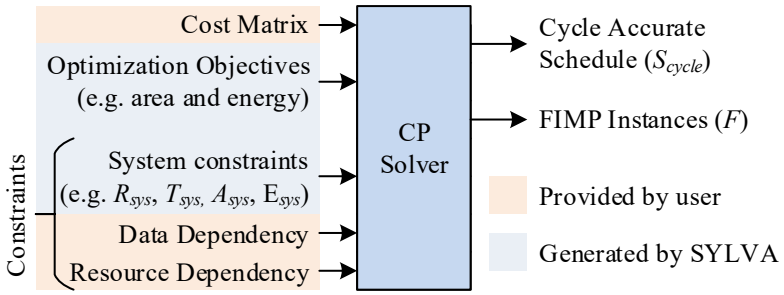


Figure 5.7: CSOP Solving

The cost matrix is a representation of the subset of a FIMP library for a specific SDF problem instance at hand that is input to the CP solver. The CP solver uses the cost matrix to find the system cost $[R, T, A, E]$ and check if the FIMP selection it is considering meets the constraints or not. The cost matrix is of the dimension $N \times 2M$, where N is the number of functions in the original, unscheduled SDF graph and M is the max number of FIMPs available to implement each function and the factor 2 in $2M$ comes because each FIMP can be with or without dedicate output buffers for each HSDF actor.

Each element in the cost matrix is a five-tuple $<u_{i,m}$, $e_{i,m}$, $t_{c,i,m}$, $t_{ie,i,m}$, $t_{os,i,m}>$ representing area usage, average energy consumption, computation end time, input end time and output start time of the m^{th} FIMP instance in F_i (the FIMP instances to implement SDF actor a_i). The three timing parameters ($t_{c,i,m}$, $t_{ie,i,m}$, $t_{os,i,m}$) are in numbers of clock cycles and they also represent part of the FIMP execution timing model that is shown in Figure 5.8, where t_s is the execution start time, t_e is the execution end time, t_{ie} is the input end time, t_{os} is the output start time, and t_{be} is the buffer usage end time. Note that these timing values (except t_{be}) are pre-characterized values from the FIMP library and obtained from sign-off quality post-layout data and exhaustive simulation and thus highly accurate.

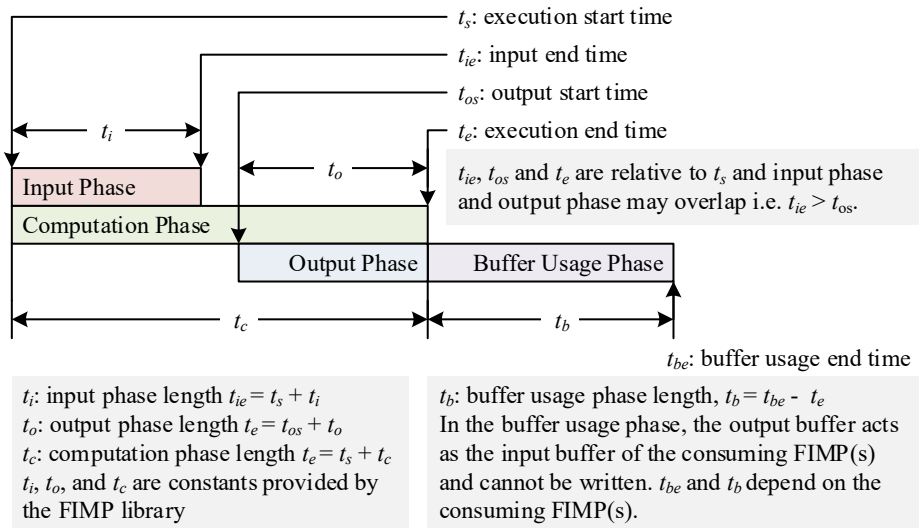
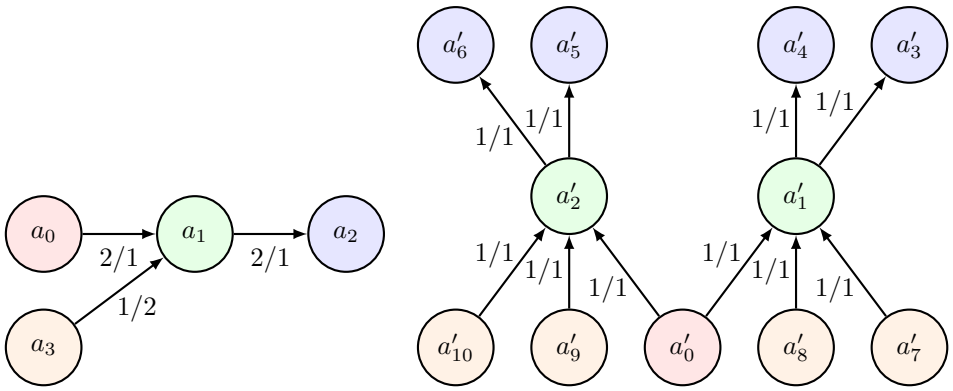


Figure 5.8: FIMP Execution Model

The optimization objectives and the system constraints are provided by the user. The optimization objectives can be minimizing one or more aspects of the system cost (system sample interval R_{sys}, system latency T_{sys}, system area usage A_{sys}, and system energy consumption E_{sys}). The system constraints are the limitations of the system cost, e.g. limiting the system sample interval to R_{MAX} or less. The data dependency and resource dependency constraints are automatically generated by SYLVA based on the system SDF graph.

The output of the CP solver consists of the cycle accurate schedule (S_{cycle} and the set of deployed FIMP instances F. Each HSDF actor has its own entry specifying the execution start time, execution end time, input end time, output start time, and the buffer end time (see Figure 5.8 for more details). Each FIMP instance $f_{i,m} \in F$ is the m^{th} FIMP instance to execute the i^{th} SDF actor $a_i \in A$ (see Table 5.1 for more details).

The cycle accurate schedule shown in Figure 5.10 will be used for elaborating these constraints. This schedule is derived from the SDF graph and the HSDF graph shown in Figure 5.9 and has following assumptions: First, the input phase lengths of the executions of a_0, a_1, a_2, and a_3 are 2, 1, 4, 2 clock cycles, respectively. Second, the computation phase lengths of the executions of a_0, a_1, a_2, and a_3 are 6, 4, 5, 4 clock cycles, respectively. Third, The output phase lengths of the executions of a_0, a_1, a_2, and a_3 are 2, 3, 2, 1 clock cycles, respectively.



(a) Original SDF Graph **(b)** Result HSDF Graph

Figure 5.9: SDF to HSDF Conversion Example

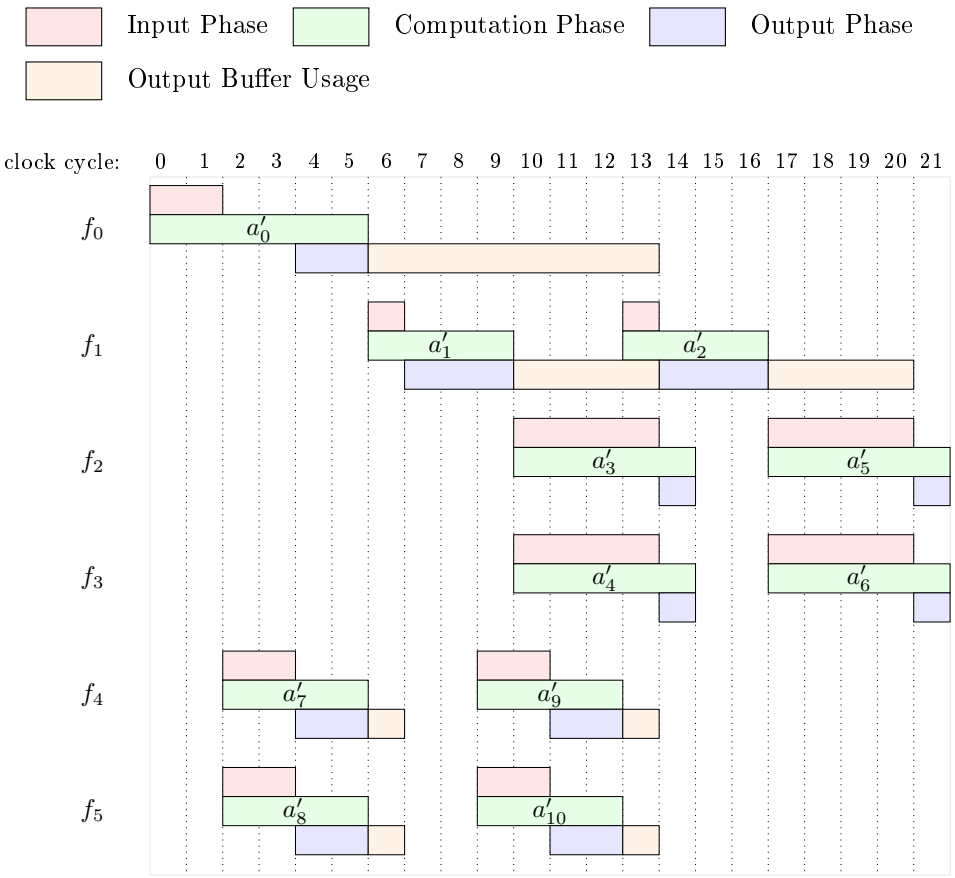


Figure 5.10: Cycle Accurate Schedule Example

Variables

In the generated CSOP, each HSDF actor $a'_j \in A'$ has one $t_{s,j}$ variable (execution start time, from 0 to T_{MAX}) and one $t_{be,j}$ variable (buffer usage end time from 0 to T_{MAX}). Each FIMP instance $f_{i,m}$ has one FIMP type variable $T_{i,m}$ (value from 0 to M_i-1, M_i is the total number FIMP types for SDF a_i and it is provided by the FIMP library) and one extra buffer variable $b_{i,m}$ (1 for use extra buffer, 0 otherwise). The total number of variables to be determined by the CP solver is $|var| = 2 \cdot (|A'| + |F|)$

System Constraints

The system constraints shown in Figure 5.7 are on the system cost [R_{sys}, T_{sys}, A_{sys}, E_{sys}]. In SYLVA, the user can provide maximum values of the system cost: R_{MAX} (maximum system sample interval), T_{MAX} (maximum system latency), A_{MAX} (maximum system area usage), and E_{MAX} (maximum system energy consumption). If any of them is not provided by user, the default value 2^{64} will be used. There are 4 system constraints as shown in Eq. 5.14.

$$\begin{array}{ll} R_{sys} & \leq R_{MAX}, \\ T_{sys} & \leq T_{MAX}, \\ A_{sys} & \leq A_{MAX}, \\ E_{sys} & \leq E_{MAX}. \end{array} \tag{5.14}$$

The system sample interval R_{sys} can be computed by Eq. 5.15, where $A'_{i,m}$, which is a actor set containing all HSDF nodes that are derived from SDF actor a_i and executed on the m^{th} FIMP instance for a_i. a'_y and a'_0 are the last and the first HSDF actor to be executed in one system iteration, respectively. The term $t_{e,y} - t_{s,0}$ is the time between the start time of a'_0 and the end time of a'_y. The term $t_{be,y} - t_{os,0}$ is the time between the output start time of a'_0 and the buffer end time of a'_y. y is determined by evaluating the HSDF graph. $t_{s,0}$ and $t_{be,y}$ are two variables to be determined by CP solver. According to Figure 5.8, $t_{e,y} = t_{s,y} + t_{c,y}$, and $t_{os,0} = t_{s,0} + t_{c,0} - t_{o,0}$. In the example shown in Figure 5.10, $R_{sys} = 14$ (Eq. 5.16).

$$R_{sys} = max(t_{e,y} - t_{s,0}, t_{be,y} - t_{os,0}) \qquad \forall\ a'_y, a'_0 \in A'_{i,m} \tag{5.15}$$

$$\begin{aligned} R_{sys} &= max(t_{e,0} - t_{s,0}, t_{be,0} - t_{os,0}, t_{e,2} - t_{s,1}, t_{be,2} - t_{os,1}, ...) \\ &= max(6, 7-4, 13-2, 14-4, 17-6, 21-7) \\ &= max(6, 3, 11, 10, 11, 14) = 14 \end{aligned} \tag{5.16}$$

The system latency T_{sys} (the time between the first input data token is consumed till the first output data token is produced) can be computed by Eq. 5.17, where A' is the actor set of the HSDF graph.

$$T_{sys} = max(t_{e,j}) \qquad \forall\ a'_j \in A' \tag{5.17}$$

The system area usage A_{sys} can be computed by Eq. 5.18, where $|F_i|$ is the number of FIMP instances for executing SDF actor a_i, $u_{i,m}$ is the area usage of $f_{i,m}$ (the m^{th} FIMP instance in F_i). The value of $u_{i,m}$ is determined by $T_{i,m}$ (the FIMP type of $f_{i,m}$ and the FIMP library.

$$A_{sys} = \sum_{i=0}^{|A|} \sum_{m=0} |F_i|(u_{i,m}). \tag{5.18}$$

The system energy cost E_{sys} can be computed by Eq. 5.19, where $e_{i,m}$ is the energy cost for executing SDF actor a_i once on FIMP instance $f_{i,m}$. $|A'_{i,m}|$ is the list of HSDF actors executed by $f_{i,m}$. Similar to $u_{i,m}$, the value of $e_{i,m}$ is also determined by $T_{i,m}$ (the FIMP type of $f_{i,m}$ and the FIMP library.

$$E_{sys} = \sum_{i=0}^{|A|} \sum_{m=0} |F_i|(e_{i,m} \cdot |A'_{i,m}|). \tag{5.19}$$

SYLVA Generated Constraints

The automatically generated data dependency and resource dependency constraints shown in Figure 5.7 can be sorted into four categories: data dependency for execution, data dependency for output buffer, resource dependency for execution, and resource dependency for output buffer.

1. **Data dependency for execution**

   A function can only start its execution after all the dependent functions are complete. For example, the HSDF actor a'_1 should start execution after HSDF actor a'_0, a'_7, and a'_8 are finished their execution. This category of constraints is formulated in Eq. 5.20, where $t_{s,d}$ is the execution start time of the destination HSDF actor a'_d and $t_{e,s}$ is execution end time of the source HSDF actor a'_s.

   $$t_{s,d} > t_{e,s} \qquad \forall\ e_{a_s \to a_d} \in E' \tag{5.20}$$

   For this example, the constraint for a'_1 can be expressed as $t_{s,1} > t_{e,0}$, $t_{s,1} > t_{e,7}$, and $t_{s,1} > t_{e,8}$. The number of constraints in this category is $|E'|$, which is the number of HSDF edges in the HSDF graph.

2. **Data dependency for output buffer**

   The producing and consuming nodes cannot output and input respectively at the same time from the output buffer. For example in Figure 5.10, the HSDF actor a'_1 should complete its input phase before the buffer end time of HSDF actor a'_0, a'_7, and a'_8.

   This category of constraints is formulated in Eq. 5.21, where $t_{ie,dst}$ is the input end time of the destination HSDF actor and $t_{be,src}$ is the output buffer end time of the source HSDF actor. For the example, it is expressed as $t_{ie,b_0} \leq t_{be,d_0}$.

   $$t_{ie,dst} \leq t_{be,src} \qquad \forall\ e \in E' \tag{5.21}$$

   The number of constraints in this category is $|E'|$, which is the number of HSDF edges in the HSDF graph.

3. **Resource dependency for execution**

   Only one actor can be executed on a FIMP instance at a time. For example in Figure 5.10, the HSDF actor a'_2 should start execution after HSDF actor a'_1 is complete. This category of constraints is formulated in Eq. 5.22. Assume $A_{i,j}$ is the HSDF actor set that is derived from the SDF actor a_i and executed on the j^{th} FIMP instances for a_i. $t_{s,x+1}$ is the execution start time for the HSDF actor a'_{x+1}, which is the $x+1$th actor to be executed by FIMP $f_{i,j}$. $t_{e,x}$ is the execution end time for HSDF actor a_x, which is the x^{th} actor in $A_{i,j}$. For the example in Figure 5.10, $A_{3,0}$ is [a'_7, a'_9] that are executed on FIMP instance $F_{3,0}$ and $A_{3,1}$ is [a'_8, a'_{10}] that are executed on FIMP instance $F_{3,1}$.

$$t_{s,x+1} > t_{e,x} \qquad \forall\ n_x \in n_{i,j}. \tag{5.22}$$

   There are $|A'| - |F|$ constraints in this category. $|A'|$ is the number of actors in the HSDF graph and $|F|$ is the total number of used FIMP instances.

4. **Resource dependency for output buffer**

   One output buffer can only be written by one HSDF actor at a time. For example in Figure 5.10, the HSDF actor a'_2 cannot start writing data into its output buffer before it is freed by a'_3 and a'_4. This category of constraints is formulated in Eq. 5.23. $t_{os,x+1}$ is the output start time for the HSDF actor $a"_{x+1}$, which is the $x+1^{th}$ actor in $A'_{i,j}$. $t_{be,x}$ is the output buffer end time for the HSDF actor $a"_x$, which is the x^{th} actor in $A'_{i,j}$. $A'_{i,j}$ is the HSDF actor set that is derived from the SDF actor a_i and executed on the j^{th} FIMP instances for a_i.

$$(t_{os,x+1} > t_{be,x} \vee b_i) = True \qquad \forall\ a"_x \in A'_{i,j} \tag{5.23}$$

   For the example in Figure 5.10, HSDF actors a'_1 and a'_2 are derived from SDF actor a_1 and t_{os,a'_2} should be larger than t_{be,a'_1} since the output buffers are shared by all the HSDF actors on the FIMP instance $F_{1,0}$ (the first FIMP instance for implementing SDF actor a_1). While in Figure 5.11, t_{os,a'_2} can be smaller than t_{be,a'_1} since each HSDF actor has its own output buffers. There are $|A'| - |F|$ constraints in this category. $|A'|$ is the number of nodes in the HSDF graph and $|F|$ is the total number of used FIMP instances.

The total number of constraints $|cst| = 2 \cdot |E'| + 2 \cdot (|A'| - |F|) + 4$, where $|E'|$ is the number of HSDF edges, A' is HSDF actor set, and $|F|$ is FIMP instance set.

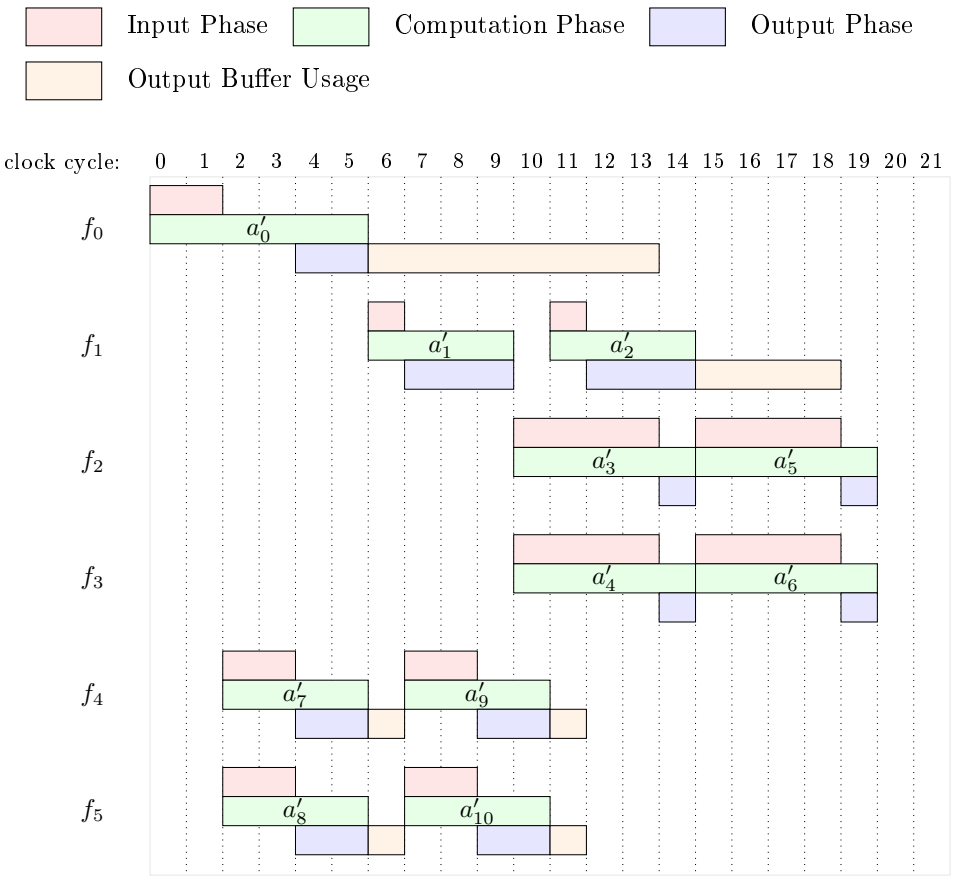


Figure 5.11: Cycle Accurate Schedule Example with More Buffers

Optimization Objectives

The optimization function of DSE is shown in Eq. 5.24, where C is the weighted total cost, which is formulated in Eq. 5.25. K_R, K_T, K_A, and K_E are four user-defined constants for specifying the optimization effort on the system sample interval, the system latency, the system area usage, and the system energy consumption in one iteration, respectively. For example, if K_R is 1 and other constants are zeros, SYLVA will minimize the system sample interval.

$$\text{Minimize}(C) \tag{5.24}$$

$$C = K_R \cdot R_{sys} + K_T \cdot T_{sys} + K_A \cdot A_{sys} + K_E \cdot E_{sys} \tag{5.25}$$

5.6 Summary

In this chapter, we have elaborated the Design Space Exploration (DSE) in System-Level Architectural Synthesis Framework (SYLVA). The input to DSE is the system Synchronous Data Flow (SDF) graph and performance/cost constraints and the optimization objectives. The output from DSE consists of the cycle accurate schedule, the number and types of the Function Implementations (FIMPs) to be deployed. The design space being explored has three dimensions for Function-Level Parallelism (FLP), Arithmetic-Level Parallelism (ALP), and Buffer-Level Parallelism (BLP), respectively. The last two dimensions are folded into one. The DSE is implemented by solving the automatically constructed Constraint Satisfaction Optimization Problems (CSOPs), which cover ALP and BLP, based on the user defined degrees of FLP.

Chapter 6

Global Interconnect and Control Synthesis

After the Design Space Exploration (DSE) step (chapter 5) is completes, System-Level Architectural Synthesis Framework (SYLVA) has synthesized the following information: number and types of Function Implementations (FIMPs) and a cycle accurate schedule that specifies when a FIMP is fired, for how long it remains active, which input/output buffer it uses and the period when it uses them. This information is the basis for synthesizing the control logic to orchestrate the execution of the FIMPs, usage of buffers and interconnect for the transfer of data. This process is called Global Interconnect and Control (GLIC) synthesis and it results in a cycle accurate, abstract simulation and synthesizable model of a hierarchy of FSM with Datapaths (FSMDs) that we call as Abstract Intermediate Representation (AIR). Note that, GLIC is only for ASIC and FPGA implementation styles. For CGRA, AIR is the bypassed output from the DSE step. And it can be directly used for generating CGRA configware since the timing and interconnect information have to be directly written into the deployed CGRA FIMPs instead of, like for ASIC and FPGA, creating logic (e.g. state machines) around the FIMPs to implement them. AIR is the common input to different back ends of the coder generator of SYLVA to implement the Digital Signal Processing (DSP) sub-system as ASIC macro, FPGA macro or a CGRA fabric instance as shown in the overall SYLVA design flow (Figure 6.1).

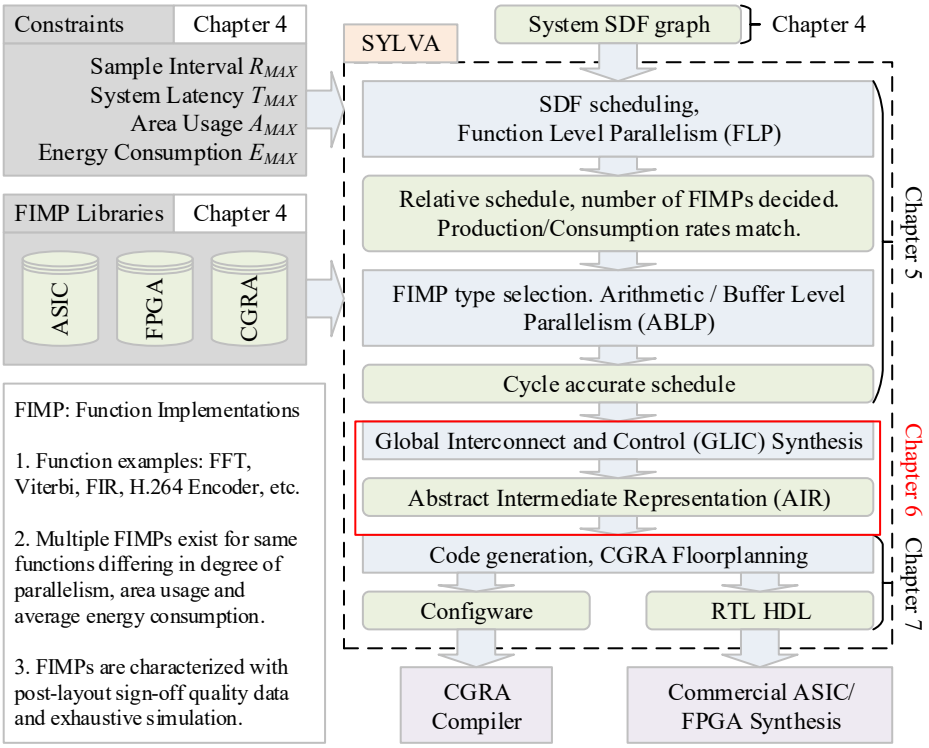


Figure 6.1: SLYVA Design Flow

6.1 Abstract Intermediate Representation (AIR)

As the name suggests, Abstract Intermediate Representation (AIR) contains all the information that is required by the code generator of System-Level Architectural Synthesis Framework (SYLVA) for generating the actual system implementation (VHDL codes for Application-Specific Integrated Circuit (ASIC)/Field-Programmable Gate Array (FPGA) and configware for Coarse-Grained Reconfigurable Architecture (CGRA)). For each system under going synthesis, its AIR has three parts: a set of AIR Building Blocks (ABBs) (each ABB is for one deployed Function Implementation (FIMP)), one global control for implementing the initial delay of all the FIMPs, and one global interconnect (the logical connection among all the ABBs). The initial delay is illustrated by the example shown in Figure 6.2 and the algorithm implemented by the global control is shown in Algorithm 6.1, where F is the set of deployed FIMPs and $t_{s,i}$ is the execution start time of the first Homogeneous SDF (HSDF) actors on the i^{th} FIMP f_i.

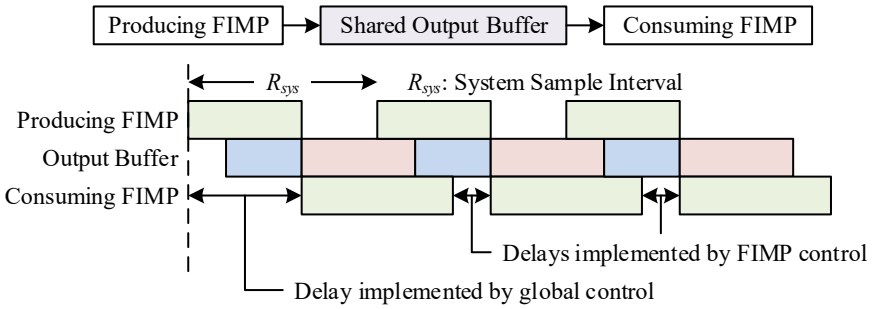


Figure 6.2: Initial Delay and Global Control

```
1 while NOT all FIMPs are enabled do
2 |   foreach f_i ∈ F do
3 |   |   if current cycle = t_{s,i} then
4 |   |   |   Enable f_i
5 |   |   end
6 |   end
7 end
```

Algorithm 6.1: Global Control

Assume we have one Synchronous Data Flow (SDF) graph example as shown in Figure 6.3 and each actor is implemented by one FIMP. The corresponding AIR is shown in Figure 6.4.

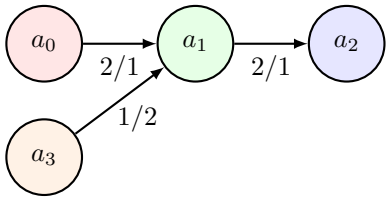


Figure 6.3: SDF Graph Example

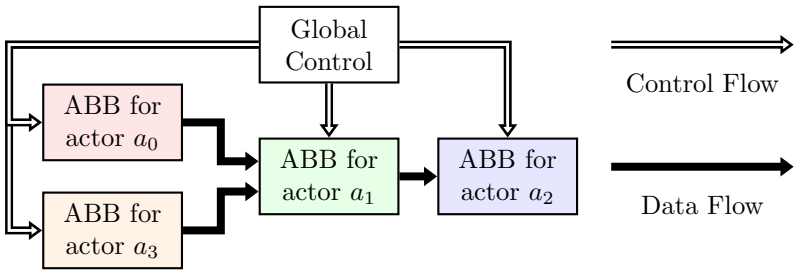


Figure 6.4: AIR Example

6.2 AIR Building Block

An AIR Building Block (ABB) module is a design template and is used as a building block to compose the aforementioned Abstract Intermediate Representation (AIR). The ABB module is a key contribution in System-Level Architectural Synthesis Framework (SYLVA) as it enables reuse of Function Implementations (FIMPs) and automatic synthesis of Global Interconnect and Control (GLIC). Each ABB module has six components: the FIMP, the buffer(s), two controllers (one for the FIMP and the other one for the buffer(s)), an input selector, and an output selector. The structural view of the ABB module is shown in Figure 6.5. The role and the interface of its six components are described next. Note that the global *clock* signal and the global asynchronous *reset* signal are available for all the components and are not explicitly shown.

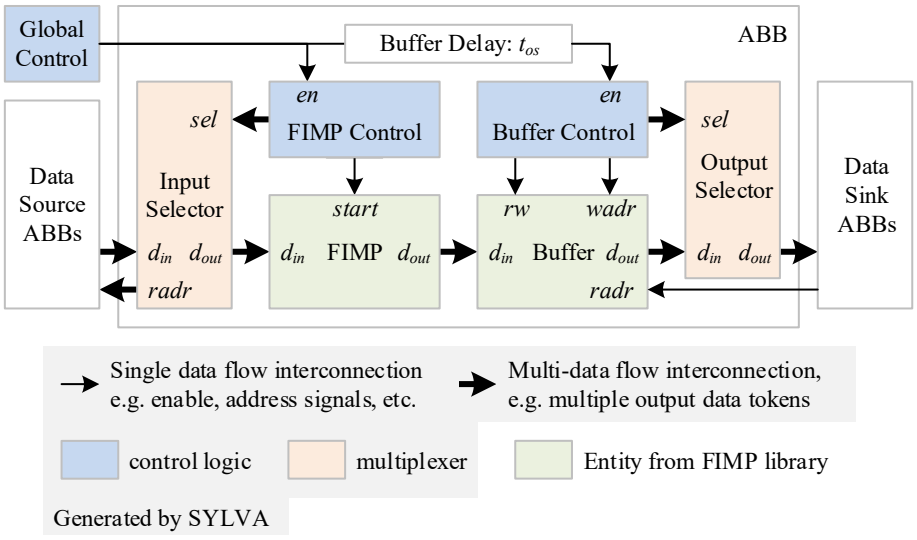


Figure 6.5: ABB Structure

As illustrated in Figure 6.5, each ABB module has the enable port *en*. If an ABB module requires input data from other ABB module(s), it will also have the data input port d_{in} and the read address port *radr* for the input data. If an ABB module outputs its data to other ABB module(s), it will also have the data output port d_{out} and the read address port *radr* for the output data.

6.2.1 FIMP Instance

Each ABB module (Figure 6.5) encapsulates a cycle accurate implementation of a specific pre-characterized instance of FIMP from the FIMP library and selected during the design space exploration phase as described in chapter 5. FIMPs internally can have their own buffers, data path and Finite State Machines (FSMs) that is unaffected by the GLIC synthesis. All FIMPs have a standard interface that plugs into the ABB. The FIMP is triggered by a pulse on its *start* port and it executes once and waits for the next pulse. In terms of the FIMP timing model (Figure 5.8, the *start* pulse marks the t_s and the FIMP instance initiates the input at its port d_{in} and ends the inputting at t_{ie}. Similarly, it initiates the outputting at its d_{out} port at t_{os} and ends it at t_e.

Note that the data input and output port (d_{in} and d_{out}) in Figure 6.5 may have multiple data flows. For example, the input port of a 64-point Fast Fourier Transform (FFT) may have 64 simultaneous data flows for the 64 input data tokens. Each data flow represents one port of the Synchronous Data Flow (SDF) actor model elaborated in chapter 4. The port *start* is not presented in the corresponding SDF actor as illustrated by Figure 6.6, where its left part is the FIMP block diagram and its right part is the corresponding SDF actor.

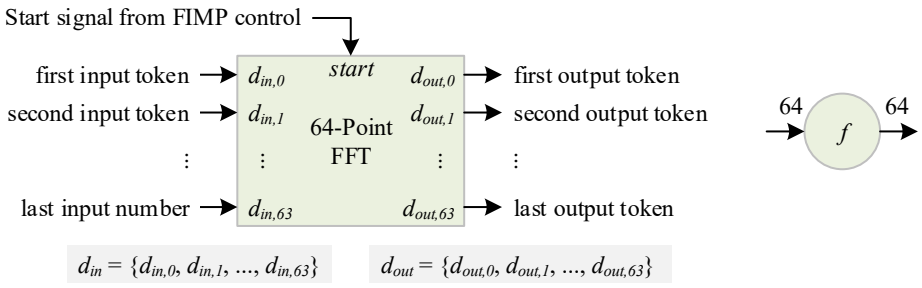


Figure 6.6: FIMP Instance Example, a 64-Point FFT

6.2.2 Input Selector

The input selector component at the input selects the input port which connects to the data input port d_{in} of the FIMP instance. An input selector is a parametric multiplexer. It is with the right dimension and automatically generated by SYLVA during the GLIC synthesis phase.

The value of the *sel* signal indicates the execution status of the Homogeneous SDF (HSDF) actor(s) that are hosted by the ABB module. Currently, the width of this signal is $3N$, where N is the number of HSDF actors to be executed on the ABB module and 3 is for the three phases in the FIMPexecution model. The meaning of the bits in *sel* signal will be elaborated later on in section 6.2.3. Similar to the data input port d_{in} of FIMP instance, the data input port of the input selector may also have multiple data flows. Each data flow represents one data port from one source SDF actor for one execution of the FIMP instance in the same ABB module. The *radr* signal is used for reading the correct data tokens from the output buffer of the data source ABB module. If each of the HSDF actors that are hosted by the data source ABB module has its own buffer, the *radr* signal will contain all the read addresses for all the buffers.

In the example shown in Figure 6.7, the *sel* for the ABB module with FIMP B_0 has two parts. $sel[0]$ is for HSDF actor b_0 and $sel[1]$ is for HSDF actor b_1. If $sel[0]$ indicates b_0 is in its input phase, the input selector will connect the first output port from FIMP A_0 to the first input port ($d_{in,0}$) of FIMP B_0 and connect the first output from D_0 and the first output from D_1 to the second input port ($d_{in,1}$) of FIMP B_0 and also sends read address signal *radr* to the corresponding ABB modules.

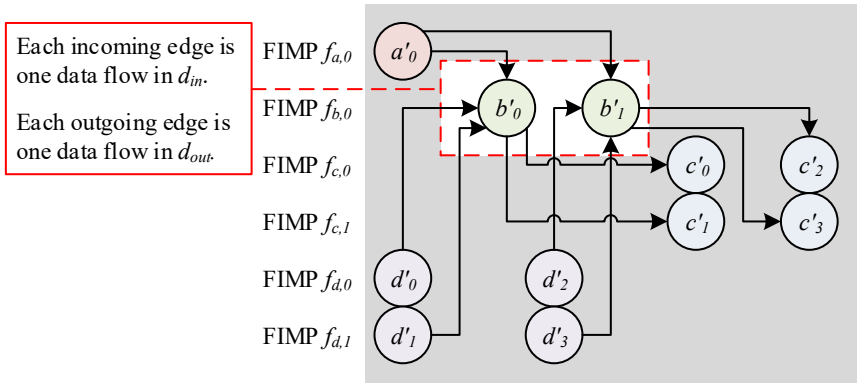


Figure 6.7: d_{in} and d_{out} in Input Selector

6.2.3 FIMP Control

FIMP control component activates the FIMP instance and selects the correct input in compliance with the cycle accurate schedule. Note that during one sample period, the same FIMP instance can potentially be fired multiple times to implement different HSDF actors, i.e. different executions of the function represented by one SDF actor in the system SDF graph. For example, e.g., the FIMP instance B_0 is fired twice to implement the HSDF actors b_0 and b_2. The FIMP control also decides when and which input port to select at the input multiplexer. For instance, the FIMP instance B_0 gets its inputs from three different FIMP instances, A_0, D_0 and D_1 as shown in Figure 6.7. GLIC synthesis generates an FSM to implement the cycle accurate schedule for the FIMP instance and the input selector.

Each FIMP control has a number of control FSMs, a clock counter, and a control decoder as shown in Figure 6.8, which is the structure of the FIMP control for FIMP B_0. The input to the FIMP control is the global control signal, which is an enable signal and it will be elaborated later in this chapter. The output from the FIMP control consists of the *sel* signal to the input selector and the *start* signal to the FIMP instance.

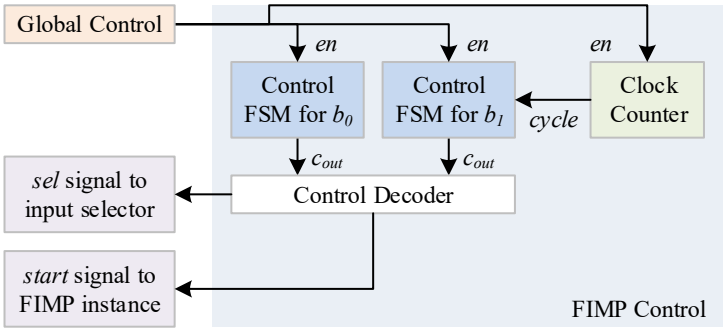


Figure 6.8: FIMP Control Structure Example for FIMP B_0

Each control FSM is for one HSDF actor. Each control FSM has a 3-bit output c_{out}, where $c_{out}[0]$, $c_{out}[1]$, and $c_{out}[2]$ indicate if the FIMP is computing, outputting, and inputting, respectively. A typical control FSM is shown in Figure 6.9, where the input phase and output phase do not overlap. Note that the timing values (t_s, t_e, t_{ie}, t_{os}, and t_e) of all the HSDF actors are relative values to the execution start time t_e of the first executed HSDF actor in this ABB. In Figure 6.9, all the timing values are relative to t_s of the HSDF actor b_0 in Figure 6.7.

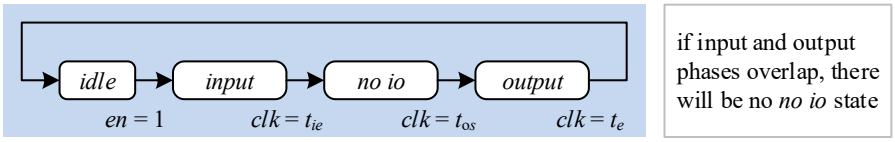


Figure 6.9: Control FSM Example

Each FIMP control has a local counter that counts the relative clock cycles to t_e of the first executed HSDF actor by the FIMP instance in the same ABB module. The output of the clock counter is the signal *cycle*. The clock counter counts when the global control signal is high ($en = 1$) and *cycle* is reset to zero when the global control signal is low ($en = 0$) or the global asynchronous *reset* signal is low (*reset* $= 0$) or *cycle* reaches the maximum value $R - 1$, where R is the system sample interval in number of clock cycles.

The control FSM provides the current state of each HSDF actor. The control decoder uses this information to generate the control signals: *sel* for the input selector and *start* for the FIMP instance. The decoding algorithm is shown in Algorithm 6.2.

```
1 C_out = all c_out for all HSDF actors
2 sel = C_out
3 foreach c_out ∈ C_out do
4 |   if c_out[0] is 1 then
5 |   |   Set start
6 |   end
7 end
```

Algorithm 6.2: Control Decoding in FIMP Control

6.2.4 Buffer Instance

The buffer component in ABB module (Figure 6.5) is clubbed with the producing FIMP instance for which it represents an output buffer but it also models as input buffer for the consuming FIMPs present in other ABB modules. For a given HSDF actor, the buffer is enabled by asserting the *en* signal, delayed by t_{os} cycles compared to the computation start time t_s. The delay is shown as Buffer Delay: t_{os} in Figure 6.5. The buffer acts as output buffer, i.e., in the write mode when the *rw* signal is asserted low; the data at d_{in} is written into the buffer at the address *wadr*. When the *rw* signal is asserted high, the buffer is in the read mode and acts as input buffer for the consuming FIMPs to which the d_{out} is connected via the output selector component. The read address is specified by the signal *radr*.

For a given FIMP instance, the input ports and the output ports of its buffer instance(s) are identical to the input ports and the output ports of the FIMP instance. For example, if a 64-point FFT has 64 data output ports for parallel output, there will be 64 data input and output ports on its buffer as illustrated in Figure 6.10, where d_{in} has $d_{in,0}$ to $d_{in,63}$. Each input port is for one complex number output from the FFT.

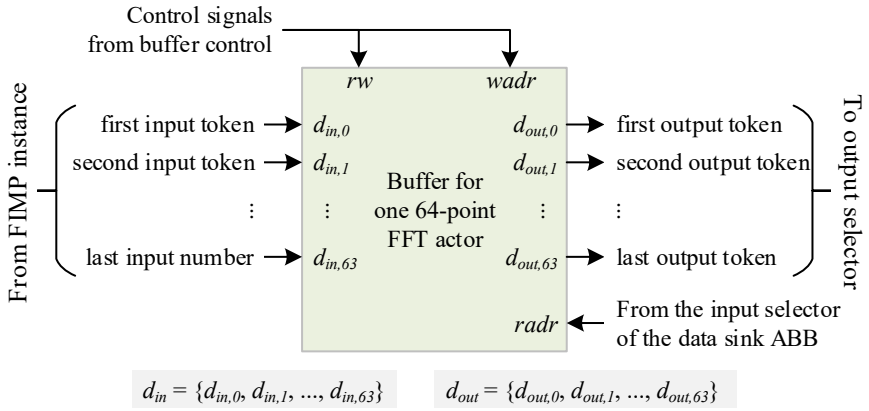


Figure 6.10: Buffer Instance Example

When a FIMP type selection indicates that each HSDF actor has its own buffer, each data output port will have multiple buffers. Each buffer is for one data output port of one HSDF actor. If the example FFT FIMP hosts four FFT actors (indexed from 0 to 3), there will be four data buffers for it and each data buffer has 64 input ports. An example of the data output port in such scenario is $d_{out,63,3}$ indicating the last output data port of the last HSDF actor (the 3rd one).

6.2.5 Output Selector

The output selector is a parametric de-multiplexer connecting the data buffer and the output ports of the ABB module. The input ports of the output selector are connected to the output ports of the buffer(s). If extra buffer is used, the output selector will directly connect its input ports to its output ports. Otherwise, the output selector will connect the selected input ports to the output ports. The port selection is decided by the buffer control component. The value of the *sel* signal indicates the current outputting HSDF actor. The output selector is with the right dimension and automatically generated by SYLVA during the GLIC synthesis step.

6.2.6 Buffer Control

Buffer control component in the ABB module (Figure 6.5) controls when and which output data is selected in compliance with the cycle accurate schedule. The buffer control and the FIMP control components share the same architecture as illustrated by the example shown in Figure 6.8. From the implementation prospective, The control FSMs are the same ones in the FIMP control, the clock counter is a independent counter component that initially starts counting t_{os} clock cycles later than the clock counter in the FIMP control. The control decoder of the buffer control will output *sel*, *rw*, and *wadr* signals. Currently, the buffer component is always enabled. The unintended output data tokens are simply discarded. The algorithm of the control decoding is shown in Algorithm 6.3.

```
1  C_out = all c_out for all HSDF actors and sel = C_out
2  T_be = all t_be for all HSDF actors
3  if Each HSDF actor has own buffer then
4  |  for j in 0 to |C_out| do  rw[j] = C_out[j][1]
5  |  Set the initial wadr[j] to 0
6  else
7  |  rw = ∃ c_out ∈ C_out that c_out[1] = 1
8  |  for j in 0 to |T_be| do
9  |  |  if current clock is T_be[j] then  Set the initial wadr to W · j
10 |  end
11 end
```

Algorithm 6.3: Control Decoding in Buffer Control

The *rw* signal reflects the current outputting HSDF actor, the *wadr* signal specifies the current outputting address in the buffer instance. In line 8 and line 14, the initial *wadr* is set to zero and $W \cdot j$, respectively. The successive values of *wadr* will be computed based the initial value. More information about how the successive *wadr* values are computed can be found in section 6.3.

6.3 Data Communications

System-Level Architectural Synthesis Framework (SYLVA) solves two types of problems caused by the data communications among AIR Building Block (ABB) modules: *spatial* and *timing*. The spatial problem is with respect to the structure of the data streams/flows when one ABB modules has multiple data sources providing different segments of the input data or what is the data structure of the output data streams/flows when one ABB module has multiple data sinks consuming different segments of the output data. The timing problem is when should a certain data token be available and the duration for which it should persist.

6.3.1 Spatial Problem

The spatial problem can be explained by considering the example shown in Figure 6.11, which was introduced in section 6.2.2. This Function Implementation (FIMP) B_0 requires data from FIMP A_0, FIMP D_0, and FIMP D_1. In the final implementation, e.g. Register-Transfer Level (RTL) VHDL codes, the input data port d_{in} has two data flows $d_{in,0}$ and $d_{in,1}$. $d_{in,0}$ is for A_0 and $d_{in,1}$ is for D_0 and D_1. The data structure of $d_{in,1}$ should be fixed in order to know if the data from D_0 is in the first half part of $d_{in,1}$ or the data from D_1 is in the first half part of $d_{in,1}$.

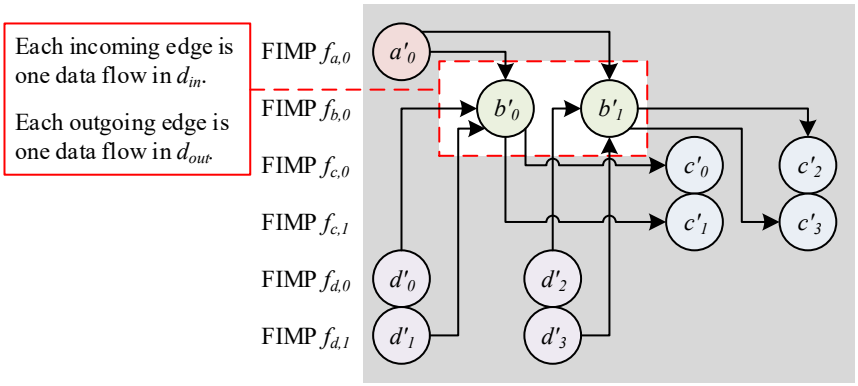


Figure 6.11: d_{in} and d_{out} in Input Selector

6.3.2 Timing Problem

The timing prospective can be explained by an 64-point Fast Fourier Transform (FFT) that may fetch all the 64 input data tokens in different number of clock cycles depending on the numbers and widths of its input ports. The consumed data tokens are 64 on each invocation and the data token indices will be from 0 to 63 identifying the 64 data tokens. If the FIMP has only one input port and it is one data token wide, the data distribution will be in a linear fashion, i.e. [0, 1, 2, ..., 63]. In case of multiple data sources, different data output ports and/or port widths, SYLVA needs to know how to connect them to the input port of this 64-point FFT FIMP.

6.3.3 Data Pattern

In order to solve these problems, we need to know the correct timing and spatial information of data communication, which are modeled by the data patterns on the data input and output ports. This pattern includes cycles where no data tokens are produced and in some cases the same data token can remain valid for multiple cycles; one being the minimum number of cycle. These pattern specifications are necessary to match and connect the ports of communicating FIMP instances.

Each data pattern on one port (either input or output) is an integer sequence. For example, assume we have a 64-point FFT FIMP with one input port ($d_{in} = [d_{in,0}]$) and it fetches data token every two clock cycles. The data pattern of this input port is shown in Eq. 6.1.

$$d_{in,0} = [0, -1, 1, -1, 2, -1, 3, -1, ..., 63, -1] \tag{6.1}$$

This sequence states that the first data token, indicated by 0, arrives at the FIMP start time t_s. Then, the second clock cycle ($t_s + 1$) is *don't care*, indicated by -1. Afterwards, the second data token, indicated by 1, arrives at cycle $t_s + 2$. Then, the fourth clock cycle ($t_s + 3$) is *don't care*, indicated by -1. This sub-sequence is repeated till $t_s + 127$, after the last data token, indicated by 63, and the last *don't care*. Note that in this example, the last *don't care* is redundant. Since a data sequence may be quite large, e.g. 4096-point FFT, 1920×1080 video compression, etc., SYLVA also supports defining a data sequence using Python syntax. The code shown below will produce the same sequence as in Eq. 6.1.

```
sum([[i, -1] for i in range(64)], [])
```

6.3.4 Communication Types and Solution

To solve the spatial and timing problems caused by the data communications, we categorize the data communication to two types: ***A*** single port to single port data communication and ***B*** single port to/from multiple ports data communication. The type ***A*** has no spatial problem and the type ***B*** may have both spatial and timing problems.

Single Port to Single Port Data Communication

The first type data communication is about directly connecting the output port of the data source ABB module to the input port of the data sink ABB module. If not specified, SYLVA assume a linear data pattern. For one port to one port data communication, SYLVA assumes that the data tokens are equally distributed during the input phase and the offset of the first data token to the input phase start time is zero. The input time for each data token is denoted as t_{last} and it can be computed by using Eq. 6.2, where M is the number of clock cycles in the input phase and N is the total number of data tokens.

$$t_{last} = \begin{cases} \lfloor \frac{M}{N} \rfloor & \lfloor \frac{M}{N} \rfloor > 0 \\ 1 & \text{otherwise} \end{cases} \tag{6.2}$$

For example, assume a 64-point FFT has 64 input data tokens and the input phase is 68 clock cycles long. In this case, $M = 68$ and $N = 64$. Then each data token takes one clock cycle ($t_{last} = 1$) and the last four clock cycles has no contribution to the input.

The start time of the i^{th} data token $t_{s,i}$ depends on the data token persisting time t_{last}, the number of clock cycles in the input phase M, and the number of input data tokens N. The value of $t_{s,i}$ can be computed by using Eq. 6.3, where n is the number of data tokens to be transmitted per t_{last} clock cycles. The value of n can be computed by using Eq. 6.4, where m is the expected number of distinct data token sets to be transmitted in the input phase. The value of m can be computed by using Eq. 6.4. Each set has n data tokens.

$$t_{s,i} = \lfloor \frac{i}{n} \rfloor \cdot t_{last} \tag{6.3}$$

$$n = \lceil \frac{N}{m} \rceil \tag{6.4}$$

$$m = \lfloor \frac{M}{t_{last}} \rfloor \tag{6.5}$$

For example, assume a 64-point FFT has 64 input data tokens and the input phase is 34 clock cycles long. Also assume that each data token should be stable for 4 clock cycles ($t_{last} = 4$). In this case, $M = 34$, $N = 64$, and $t_{last} = 4$. Then $m = \lfloor \frac{34}{4} \rfloor = 8$, $n = \lceil \frac{64}{8} \rceil = 8$, and $t_{s,i} = \lfloor \frac{i}{8} \rfloor \cdot 4 = \lfloor \frac{i}{2} \rfloor$

Note that the expected m may not equal to the actual number of transmitted data token m'. The reason is that n takes only the upper limit. Therefore, more data tokens than the expected value when we compute m may be transmitted on each t_{last} interval. The number of the additional data tokens accumulates and may result less data token sets to be transmitted. The value of m' can be computed using Eq. 6.6. The expected m is the upper bound of m' ($m' \leq m$). This can be proved as shown below.

$$m' = \lceil \frac{N}{n} \rceil \tag{6.6}$$

$$\begin{aligned} m' = \lceil \frac{N}{n} \rceil &< \frac{N}{n} + 1 \\ &< \frac{N}{\lceil \frac{N}{m} \rceil} + 1 \\ &< \frac{N}{\frac{N}{m}} + 1 \\ &< m + 1 \\ \rightarrow m' &< m + 1 \\ \rightarrow m' &\leq m \end{aligned}$$

The lower bound of m' can be obtained as shown below.

$$\begin{aligned} m' = \lceil \frac{N}{n} \rceil &\geq \frac{N}{n} \\ &\geq \frac{N}{\lceil \frac{N}{m} \rceil} \\ &> \frac{N}{\frac{N}{m} + 1} \\ &> \frac{N \cdot m}{N + m} \\ \rightarrow m' &> \frac{N \cdot m}{N + m} \end{aligned}$$

Therefore, $N \cdot m/(N + m) < m' \leq m$. For example, if the m of an 64-point FFT is 9 ($m = 9$ and $N = 64$), $n = \lceil N/m \rceil = \lceil 64/9 \rceil = 8$ and $m' = \lceil 64/8 \rceil = 8$.

Single Port to/from Multiple Ports Data Communication

Only the data sequence definition is not enough for defining which data token should be consumed by which Synchronous Data Flow (SDF) actor in a single source to multiple destination style data transmission. The data tokens may arbitrarily distributed among the destinations. SYLVA provides two method to handle this. The first one is creating a custom FIMP for complex data token distributions that normally has no regulation or the distribution regulation is hard to model. Another method is to use a predefined distribution regulation. Currently, SYLVA supports linear and interleaving data token distributions in the single to/from multiple style data communication.

1. **Linear**: The first block of data tokens goes to the first destination port, the second block of data tokens goes to the second destination port, and so on. The sizes of the data token blocks are the data token counts of the corresponding port. For example, assume we have an output data token sequence as defined as follows:

   $$l_d = [\ 0, 1, 2, 3, 4, 5, 6, 7\ ]$$

   Also assume that two input ports will consume the data tokens. The data token sequences on the input ports (port a and port b) should be as follows:

   $$l_{d,a} = [\ 0, 1, 2, 3\ ]$$
   $$l_{d,b} = [\ 4, 5, 6, 7\ ]$$

2. **Interleaving**: The data tokens go to the destination port in a round-robin style. The first data token block goes to the first destination port, the second data token block goes to the second destination port, and so on. The sizes of the data token blocks can be defined by the system designer and its value is 1 by default. For example, assume we have an output data token sequence as defined as follows:

   $$l_d = [\ 0, 1, 2, 3, 4, 5, 6, 7\ ]$$

   Also assume that two input ports will consume the data tokens. The data token sequences on the input ports (port a and port b) should be as follows:

   $$l_{d,a} = [\ 0, 2, 4, 6\ ]$$
   $$l_{d,b} = [\ 1, 3, 5, 7\ ]$$

6.4 AIR Generation

In System-Level Architectural Synthesis Framework (SYLVA), generating the Abstract Intermediate Representation (AIR) of one system is to create a new Homogeneous SDF (HSDF) graph G' consisting one global control actor and a number of sub-graphs. Each sub-graph corresponds to one AIR Building Block (ABB) and has following connected actors.

1. One actor for the Function Implementation (FIMP) instance connected with the FIMP control, the input selector and the output buffers.
2. One counter actor connected with the FIMP control and the buffer control. This counter provides the relative time in each sample interval in terms of number of clock cycles.
3. One actor for the FIMP control, which is one Finite State Machine (FSM), connected with the global control, the counter, the FIMP instance and the input selector.
4. One actor for the buffer control, which is one FSM, connected with the global control, the counter, the output buffers and the output selector.
5. One actor for the input selector, which is one Multi-Input Multi-Output (MIMO) multiplexer, connected with the FIMP control, the FIMP instance, the output selector of another ABB and its output buffers.
6. One actor for the output selector, which is one MIMO multiplexer, connected with the buffer control, output buffers and the input selector of another ABB.
7. A number of actors for the output buffers connected with the FIMP instance, the buffer control, the output selector and the input selector of another ABB. If extra buffer is not used, the number of output buffers equals to the number of output ports of the FIMP instance. Otherwise, the number of output buffers equals to the number of output ports of the FIMP instance times the number of HSDF actors that will be executed by the FIMP instance.

The interconnect within a sub-graph is the same as illustrated in Figure 6.5 except the buffer delay t_{os} is absorbed in the buffer control and there will be a new counter actor added for the sake of code generation. If single port to/from multiple ports data communication happens, each port in the multiple port side will have two communication edges to/from the single port side when connecting different ABBs. One edge is for the output selector in the data source ABB to the input selector in the data sink ABB. The other one is for the input selector of the data sink ABB to the address port of the output buffer in the data source ABB.

The generated AIR HSDF graph is illustrated in Figure 6.12. It follows the structure illustrated in Figure 6.5. All the actors are generated by SYLVA automatically except the FIMP actor, which is provided by the FIMP library.

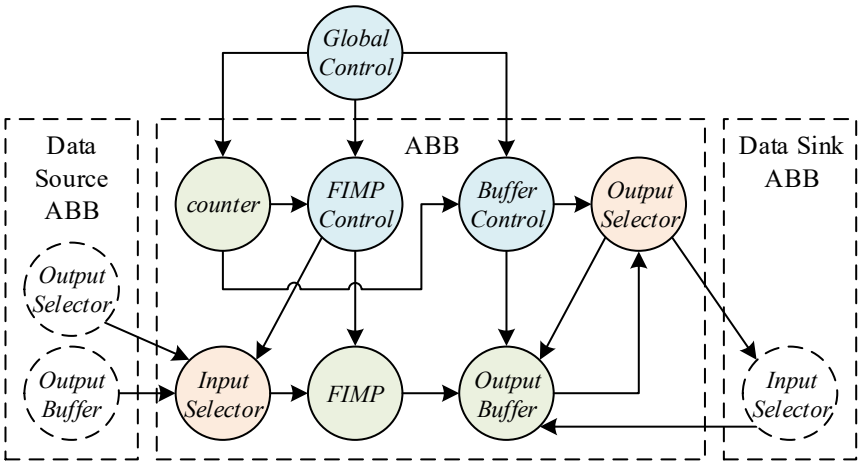


Figure 6.12: AIR HSDF Graph

The generation flow is shown in Algorithm 6.4. Note that the generated actors except the FIMP instance actor are only for storing required information for code generation, i.e. the control actors store FSM states and transaction edges, the selector actors store the selection to connection maps, and the output buffer actors store the buffer size.

```
1  foreach FIMP instance in the DSE result do
2  |  Create all FSM states and transaction edges for FIMP computations
3  |  Create all FSM states and transaction edges for input selectors
4  |  Create the FIMP control FSM based on above results
5  |  Create output buffers
6  |  Create all FSM states and transaction edges for output selectors
7  |  Construct one ABB that contains the above information.
8  end
9  Create global control FSM based on the DSE result
10 foreach ABB do
11 |  Create a counter actor counting from 0 to R_MAX − 1
12 |  Create FIMP control actor
13 |  Create FIMP instance actor
14 |  Create input selector actor
15 |  Create output selector actor
16 |  Create buffer control actor
17 |  Create output buffers
18 |  Connect the created actors
19 end
20 Connect the created ABBs and the global control actor
```

Algorithm 6.4: AIR Generation

Note that the loop in the original Synchronous Data Flow (SDF) graph will not lead to an infinite loop in AIR generation since we create ABBs FIMP by FIMP. The data dependency loop caused by the feedback edges will not have effort on the data input and output timing, which is only depend on the FIMP type and the FIMP start time.

6.5 Summary

This chapter describes Abstract Intermediate Representation (AIR) and the method to generate it. One AIR consists of a set of AIR Building Block (ABB) modules, one global interconnect, and one global control. Each ABB module has one Function Implementation (FIMP) instance, one input selector, one FIMP control, one buffer control, one output selector, and one or multiple buffer instances. The global interconnect is the logical interconnection among all the ABB modules and the global control, which is for implementing the initial delay of every ABB modules. AIR is stored as a Homogeneous SDF (HSDF) graph in System-Level Architectural Synthesis Framework (SYLVA) and each ABB is stored as a sub-graph in the HSDF graph.

This chapter also addresses two problems caused by the data communication among ABB modules are described in this chapter. One problem is the spatial problem about the structure of the data when a single port sends or receives data tokens from multiple ports. The other problem is the timing problem and concerns when a data token should be available and how long it should stay available. To solve these problems, data pattern is proposed and two types of communications are assumed. One data pattern is for storing the data structure including both timing and spatial information for one port. The two types of communications are single port to single port communication and single port to/from multiple ports communication. For single port to single port communication we assume all data tokens that are transmitted occupy the same size time slot. For single port to/from multiple ports communication, we assume two types of data splitting method: linear and interleaving.

Chapter 7

Code Generation

Once the Global Interconnect and Control (GLIC) is ready, the next step is to generate the actual implementation. For Application-Specific Integrated Circuit (ASIC) and Field-Programmable Gate Array (FPGA) implementation styles, Register-Transfer Level (RTL) VHDL codes (separate entities for Function Implementation (FIMP) instances and a top entity for the interconnect and control) will be generated. For Coarse-Grained Reconfigurable Architecture (CGRA), MATLAB codes (separate m-functions for FIMP instances and a top script for the interconnect and control) will be generated. If specified, System-Level Architectural Synthesis Framework (SYLVA) can also perform CGRA floorplanning.

As described in detail in chapter 6, Abstract Intermediate Representation (AIR) is constructed based on the execution schedule information, data dependency information, and FIMP instance information that are obtained in chapter 5. The AIR and the final implementation have the same structure. The code generation in SYLVA can be summarized in four steps as visualized in Figure 7.1.

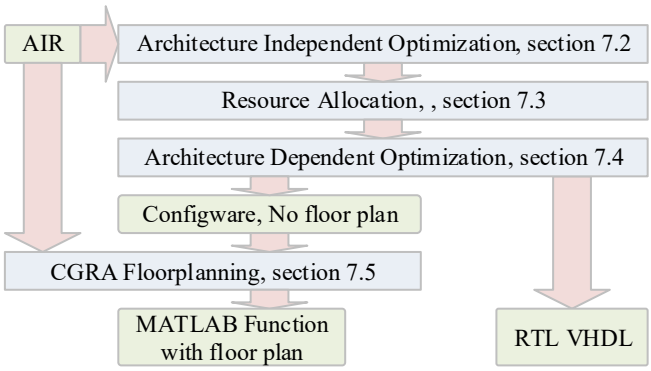


Figure 7.1: Code Generation in SYLVA

1. Architecture Independent Optimization (section 7.3)

   In this step, the AIR will be optimized regardless the target implementation style and architecture. Currently, the total size of the data buffer is optimized by introducing data buffer sharing in a time-multiplexing fashion.

2. Resource Allocation (section 7.4)

   All the components (input selector, output selector, FIMP instance, data buffer, FIMP control, and buffer control) in AIR Building Blocks (ABBs), the global control, and the global interconnect are mapped to implementation codes (VHDL for ASIC/FPGA and m-function for CGRA). The AIR to implementation codes conversion in this step is a template based code generation. The implementation codes of the FIMP instances are generated based on the templates in the FIMP library. The implementation codes of the other components are generated based on the SYLVA building templates.

3. Architecture Dependent Optimization (7.5)

   In this step, the implementation will be optimized based on the target architecture. Similar to the architecture independent optimization, currently, only the data buffer blocks can be optimized to fit in the actual memory blocks on the target architecture. For example, if one FPGA platform has limited on-chip memory (SRAM) that is less than the required total data buffer size (data buffer size cannot be a constraint currently), some of the data buffers have to be mapped on the off-chip memory blocks (SDRAMs). The partition of on-chip and off-chip data buffers will have critical impact on the system performance since the off-chip memory will be much slower than the on-chip memory. The execution schedule may be changed in this step. If the system performance does not meet the constraints (area, energy, latency, and/or sample interval), we need to change the constraint and repeat the entire synthesis flow. Currently, this step is done manually.

4. CGRA Floorplanning (section 7.6)

   If the target architecture is CGRA, an additional CGRA floorplanning step may be performed for producing the suitable configware. Currently, the floorplanning optimization objective can be one of the following.

   - Total energy consumption
   - Total resource usage
   - Implementation width in terms of CGRA grains
   - Implementation height in terms of CGRA grains

   SYLVA has two different CGRA floorplanners. One is a Constraint Programming (CP) solver based floorplanner. The other one is a heuristic algorithm based floorplanner. The first one is slower than the second one but often generates better results.

7.1 Problem Definition

Before detailing the code generation procedures, we first need to define the problems that need to be solved. The code generation problem is divided into two subproblems. The first one is a straightforward replacement problem that replace Abstract Intermediate Representation (AIR) by the corresponding VHDL/MATLAB codes (section 7.4). The second one is a memory allocation problem (section 7.3 and section 7.5).

7.1.1 Problem 1: Convert AIR to Implementation

We are given the following.

1. An AIR consisting of a set of AIR Building Blocks (ABBs) with global interconnect and control logics
2. A Function Implementation (FIMP) library L containing all the FIMPs involved in the AIR

The result should be a set of VHDL or MATLAB codes (denoted as C) that implement the AIR. This problem is basically a one-to-one mapping problem and can be solved in linear time. The solution to this problem is described in section 7.4.

7.1.2 Problem 2: Memory Allocation Problem

The second problem is far more complicated than the first one. Currently, System-Level Architectural Synthesis Framework (SYLVA) can only optimize memory blocks due to following reasons.

1. SYLVA cannot modify FIMPs
2. The performance and cost of the memory subsystem is critical in the whole system

As reported in [105] that as much as 40% of the total system energy is consumed by the main memory subsystem. Therefore, SYLVA tries to optimize the memory subsystem to reduce the overall energy consumption. We define the memory allocation problem as the following optimization problem.

Given a set of data buffer schedules B defined by Eq. 7.1, where $b(i, j)$ is one data buffer access schedule, N is the number of FIMPs in the system, and M is the maximum number of data buffers in one FIMP. Therefore, the data buffers are in a $N \times M$ matrix.

$$B = \{b(i, j) | 0 \leq i \leq N, 0 \leq j \leq M\} \tag{7.1}$$

Want a set of mapped memory blocks in the Register-Transfer Level (RTL) model MM defined in Eq. 7.2, where $mm(k)$ is one mapped memory block and O is the total number of mapped memory blocks. Each mapped memory block mm is either a VHDL entity for Application-Specific Integrated Circuit (ASIC)/Field-Programmable Gate Array (FPGA) or a MATLAB variable for Coarse-Grained Reconfigurable Architecture (CGRA). The set of mapped memory blocks MM should be only optimized for energy consumption since we will not change the system schedule nor the FIMP type in the code generation step.

$$MM = \{mm(k) \mid 0 \leq k \leq O\} \tag{7.2}$$

This B to MM conversion raises two critical problems. The first problem is modeling the existing memory systems. For example, in ASIC implementation style, a 64-point Fast Fourier Transform (FFT) may use on-chip SRAM or registers. In FPGA implementation style, the same FFT may use on-chip SRAM, LUTs (Look-Up Tables), or off-chip DRAM (e.g. CoRAM [106]). In CGRA implementation style, a 64-point FFT may use on-chip SRAM, registers, or off-chip DRAM. In addition, the memory can also be in a distributed style (e.g. [29] for a CGRA). All the possible memory implementation styles should be considered when allocating memory resources for the final RTL model. This requires a generic memory model for all the implementation styles (ASIC, FPGA and CGRA). Moreover, performance information (e.g. latency, throughput) should also be contained in the memory model. We will define the generic memory model M_g later on in section 7.2. The other problem is the allocation and optimization method. Some optimizations can be done directly on the data buffer schedules B without considering the actual implementation. We call them as architecture independent optimizations. For example, merge memory blocks to a suitable size that could be allocated as a single on-chip memory resource of an FPGA. The architecture independent optimization will be elaborated in section 7.3. Some optimization can only be done by considering the actual implementation. For example, replacing multiple on-chip SRAMs with off-chip DRAM + on-chip adapter if possible. We call them as architecture dependent optimizations. The architecture dependent optimization will be elaborated in section 7.5.

7.2 Generic Memory Model

As the target implementation style of the System-Level Architectural Synthesis Framework (SYLVA) can be either Application-Specific Integrated Circuit (ASIC), Field-Programmable Gate Array (FPGA), or Coarse-Grained Reconfigurable Architecture (CGRA), the generic memory model M_g should cover all the memory architectures in the target implementation styles. The information to be modeled is as follows.

1. Memory Capacity - the size of the memory block.
2. Memory Connectivity - single port or multiple ports.
3. Access Latency - memory access timing model.
4. Access Energy - memory access energy model.
5. Resource Usage - resource to host the memory block.
6. Implementation - VHDL design unit/MATLAB matrix.

Memory organization is hard to describe by a generic model due to large number of variations. The proposed generic memory model abstracts the common memory properties among the memory blocks used in ASIC and FPGA. The memory model (M_g) defined in Eq. 7.3 has following properties: capacity (S), connectivity (C), access latency (T_a), resource usage (R), and energy consumption (E).

$$M_g = \{S, C, T_a, R, E\} \tag{7.3}$$

Capacity

The capacity or size (S) of a memory is a critical property. It can be further divided into logical capacity (L) and physical capacity (P). S is defined in Eq. 7.4.

$$S = \{L, P\} \tag{7.4}$$

Logical capacity L is the memory size from the perspective of the system. It determines the maximum data size can be stored. The logical capacity consists of one or multiple physical capacity as distributed memory blocks. Physical capacity is the size of the data that can be stored in one physical memory block.

In the proposed memory model the capacity is in terms of bits. For example, if we model a 2 Mega bits SRAM consisting of 16 distributed SRAM blocks as one memory model, its capacity is denoted as { 2 M, 16 $\times$ 128 K }. The logical capacity is 2 M. The capacity of a 128 K bits SRAM block is denoted as { 128 K, 128 K}.

In most of the cases, the accesses to all the physical memory blocks are equivalent. However, they can be different in some cases, e.g. distributed memory blocks are modeled as one virtual memory. To access different portions of this virtual memory may result different latencies due to the varying distance to the accessed physical memory block. Or even some certain physical memory blocks may be inaccessible to a set of processing elements.

Connectivity

The connectivity (C) defined in Eq. 7.5 models the size of data that can be simultaneously accessed. For example, the connectivity of a single port SRAM block with data width of 64-bit is denoted as $C = \{1 \times 64\}$.

$$C = \{\text{ports} \times \text{port width}\} \tag{7.5}$$

Access Latency

The access latency (T_a) has a critical impact on the system latency and throughput. We assume that a memory access can be in random or burst style. T_a can be denoted as in Eq. 7.6.

$$T_a = \{T_i, T_m\} \tag{7.6}$$

In Eq. 7.6, T_i is initial delay. It is the time between the arrival of the first data and the arrival of its address. T_m is middle delay, which is the minimum time between two successive data changes. Figure 7.2 illustrates the initial delay T_i and the middle delay T_m. For random access, the actual access latency is T_i. For burst access, the actual access latency of the first data is T_i and new data can be read or written every T_m clock cycles.

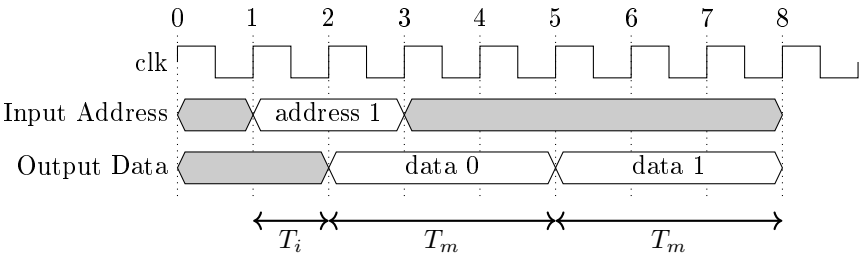


Figure 7.2: Memory Access Timing Model

Resource Usage

We model the resource usage (R) in terms of equivalent gate count, which is the relative number of two input NAND gates in CMOS technology. For example, a typical 6-transistor SRAM cell (4 transistors for data storing and 2 transistors for access control) is equivalent to 1.5 gates since one NAND gate with 2 inputs uses four transistors (2 NMOS, 2 PMOS). As the memory can be either on-chip or off-chip, the resource usage is the sum the used resources of both on-chip and off-chip part. For example, the resource usage of a one million bits on-chip SRAM is {1.5 M}. The resource usage of a one million bits off-chip SDRAM (1-transistor per bit) with its on-chip adapter (e.g. 5 K gates in ASIC style) is {0.255 M}, where 0.25 M gates for the off-chip SDRAM and 5 K gates for the on-chip adapter.

Energy Consumption

The energy consumption (E) is modeled as the average access energy per bit at 100 MHz and with 0.35 switching activity. We set this high switching activity due to that the target application domain is Digital Signal Processing (DSP) with static data rate, i.e. there is always new data. The energy consumption of the memory blocks that are not running at 100 MHz will be normalized to 100 MHz using the method stated in [107]. Although the actual energy consumption is related to the actual switching activity and the transmission distance, E only model the average energy for simplicity in this thesis.

7.3 Architecture Independent Optimization

We cannot always map data storage to on-chip SRAM. As mentioned earlier in this section, the architecture independent and dependent optimizations can help us to shrink the final memory resource usage. In the architecture independent optimization step, redundant address ranges are eliminated and the size of the data buffers are adjusted based on the Function Implementation (FIMP) instances. In the architecture dependent optimization step, the size of the data buffers are adjusted to be more suitable in the actual implementation.

For the example shown in Figure 7.3, the FIMP instance f_0 that executes a_1 and a_2 initially stores the outputs of a_1 and a_2 separately and the outputs are accessed in different time interval. This is an obvious waste and the size of the data buffer can be reduced by 50%. To find the redundant address ranges, liveness analysis is performed. Denoting the liveness analysis result as r_l, the pseudo-code for redundancy elimination is shown in Algorithm 7.1.

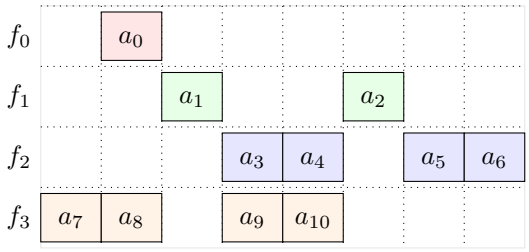


Figure 7.3: Schedule Example

```
1  r_l = ∅
2  foreach entry r in the read actions do
3  |   Find an entry w in the write actions that
4  |   r.A ⊂ w.A and r.T > w.T
5  |   if w exists then
6  |   |   Add (r.A, w.T, r.T) to r_l
7  |   |   Remove r.T from w.T
8  |   end
9  |   Merge all redundant address ranges in r_l
10 end
```

Algorithm 7.1: Redundancy Elimination

After redundant address ranges are eliminated, buffer sizes are adjusted based on the data buffer size limit L that is provided by the FIMP library. The data buffer resizing algorithm is shown in Algorithm 7.2.

```
1  B: the set of data buffers to be resized
2  L: the data buffer size limit
3  foreach data buffer b in B do
4  |  Set |b| be the size of b
5  |  if |b| > L then
6  |  |  Divide b into a set of smaller data buffers b_n
7  |  |  The number of new data buffers is ⌈|b|/L⌉
8  |  |  Each new data buffer b' ∈ b_n has a size of L
9  |  |  Change the control signals from accessing b to accessing b_n
10 |  end
11 end
```

Algorithm 7.2: Data Buffer Resizing

At the end of this step, large data buffers are divided into data buffers with smaller size. They are ready to be mapped onto actual memory blocks. Note that when the addresses are changed, this change should be reflected to the corresponding AIR Building Blocks (ABBs). For Coarse-Grained Reconfigurable Architecture (CGRA), accessing different memory blocks may result different access time. If this happens, the cycle accurate schedule should be modified. If any of the constraints are violated due to this schedule change, we can either discard this optimization or restart the synthesis flow with a more strict constraint. The decision is left to be made by the user.

7.4 Resource Allocation

In this sub-step, all the AIR Building Blocks (ABBs) and the global interconnect and control logic will be converted to implementation codes. The code generation algorithm is shown in Algorithm 7.3. This section focuses on the code generation for each component of ABB.

```
1  ABBs: the AIR building blocks
2  GC: the global control
3  L: the Function Implementation (FIMP) library
4  Set C as ∅
5  foreach ABB abb in ABBs do
6  |  Set c as the corresponding implementation template in L
7  |  Replace variables in c by the specified value in abb
8  |  Add c into C
9  end
10 Set gc as the implementation template of GC
11 Replace variables in gc by the specified value in GC
12 Add gc into C
```

Algorithm 7.3: Convert ABB to Codes

The sequence of code generation of the ABB components (FIMP instance, data buffer, input selector, output selector, FIMP control, buffer control) is shown in Figure 7.4. Note that the code generation of input selector should be after the codes of the FIMP instance and the output selector of the data source ABBs (if any) are generated.

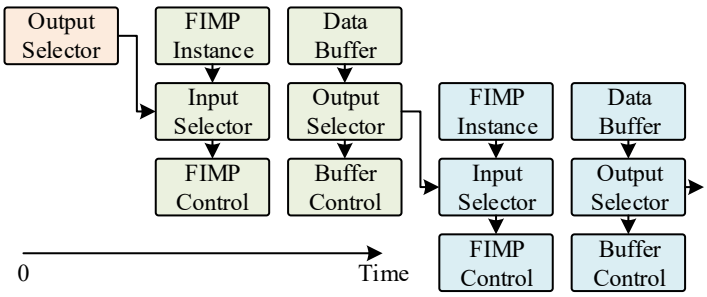


Figure 7.4: Code Generation Procedure

7.4.1 Code Generation of FIMP Instance

Code Generation of FIMP Instance for ASIC and FPGA

The actual implementation codes are generated based on the implementation templates stored in the FIMP library. For Application-Specific Integrated Circuit (ASIC) and Field-Programmable Gate Array (FPGA), these are in the form of parametric VHDL design units with generic parameters. Multiple FIMP instances may share the same VHDL entity but each FIMP instance has its own values of the generic parameters.

For example, consider a 16-tap Finite Impulse Response (FIR) symmetric filter with 8-bit input and output data. The implication is that we need 8 MACs for maximum parallelism. Assuming that it can be implemented by four FIMP instances with different number of Multiply-ACcumulators (MACs): 1, 2, 4, or 8. All the four FIMP instances share the same VHDL entity as the implementation template called *FIR16_1_1*. The first *_1* means its input data size is one and the other *_1* means its output data size is also one. The code segment containing port and generic sections is shown in Code 5. By removing the generic section from the VHDL code and replacing *stage* with 1, 2, 4 or 8, we can generate the VHDL design units for four different FIMP instances. If there are two FIMPs for 16-tap FIR filters with the same FIMP type (16FIR_1_1), they will have different FIMP names after instantiation (*16FIR_1_1_FIMP_0* and *16FIR_1_1_FIMP_1*).

```
1  entity FIR16_1_1 is
2  generic (
3    stage : integer range 1 to 8 := 1;
4    coeff : in coeff (7 downto 0) :=...
5  );
6  port (
7    clock : in std_logic;
8    nreset : in std_logic;
9    din : in integer range -128 to 127;
10   dout : out integer range -128 to 127
11 );
12 end FIR16_1_1;
```

Code 5: FIMP Instance Template Example in VHDL

Code Generation of FIMP Instance for CGRA

Code generation for Coarse-Grained Reconfigurable Architecture (CGRA) is similar for ASIC and FPGA as explained above. For the 16-tap FIR symmetric filter example, a constant *MACs* is used for the number of iterations in the outer loop reflecting the number of stages. The code segment is shown in Code 6.

```
1  % Function: FIR16 one input port and one ouput port
2  for j = 1 : MACs % SIMD threads: MACs
3    for i = 1 : (8/MACs)
4      % Inner loop: multiplications and additions
```

Code 6: FIMP Instance Template Example in MATLAB

Line 1 provides the architecture of the FIMP. Line 2 determines the number of MACs. The comment is a pragma that marks the number of intended SIMD threads. This comment is not executed by MATLAB but provides information to the the CGRA compiler. It only shows the method to provide information to the CGRA compiler and the actual acceptable pragma may not in this form. Line 3 determines the number of iterations. By replacing *MACs* with 1, 2, 4 or 8, we can generate the MATLAB function files for the four different FIMPs. Although there is no difference when these MATLAB functions are executed in MATLAB environment, they provide information to the CGRA compiler to generate FIR filter with the right degree of parallelism. The algorithm for generating the code of a FIMP instance is shown in Algorithm 7.4.

1 Set *CFG* = Configuration of the implementation template
2 Set *Temp* = Implementation template
3 **if** *target architecture is ASIC or FPGA* **then**
4 | Remove the generic section in *Temp*
5 **end**
6 Replace parameters in *Temp* with their values in *CFG*
7 Set i = FIMP instance index
8 Append suffix $\_i$ to the entity name of *Temp*
9 result = *Temp*

Algorithm 7.4: Code Generation Algorithm for FIMP Instances

In line 1, *CFG* represents the names and values of the generic parameters (for VHDL entity) or constants (for MATLAB function) to be used in implementation template. For example, if the number of stages is two, then *CFG* = {stage, 2} for ASIC and FPGA, and *CFG* = {REPLACE_stage, 2} for CGRA. In line 2, *Temp* represents the implementation template, which is a VHDL entity with generic section for ASIC/FPGA or a MATLAB function containing constants without value assignment. In line 7, *result* is the actual implementation code of the FIMP. Since there is no generic section in MATLAB file, it is no need to be removed.

7.4.2 Code Generation of Data Buffer Instance

Code Generation of Data Buffer Instance for ASIC

In ASIC implementation style, the data buffers are instantiated as on-chip SRAM blocks. Note that there can be multiple data buffer design units in one ABB. The type of the data buffer used for a FIMP is specified by the FIMP, i.e. the data buffer implementation is part of the FIMP implementation as described in section 4.3. Each data buffer in the ABB is instantiated as one or more memory blocks. The number of memory blocks is determined by the output port of the FIMP.

For example, each data buffer is instantiated as one memory block for the example 16-tap FIR filter, since it has only one output port. However, for an RGB to YUV converter, which has three separate output ports for Y, U and V signals, each data buffer is instantiated as three memory blocks. Following code segment is part of the VHDL module for one memory block.

```
1  entity SYLVADataBuffer is
2  generic (
3    capacity : integer := 256;
4    data width : integer := 8
5  );
6  port (
7    clock : in std logic;
8    nreset : in std logic;
9    enable : in std logic;
10   nwrite : in std logic;
11   address : in integer;
12   din : in std logic vector ( data width - 1 downto 0 );
13   dout : out std logic vector ( data width - 1 downto 0 )
14 );
15 end SYLVADataBuffer;
```

Code 7: Data Buffer Instance Template in VHDL

The pseudo-code for generating the VHDL code for ASIC implementation style is shown as follows:

```
1  Set result = ∅
2  foreach ABB a do
3  |  Set j = 0
4  |  foreach output port p do
5  |  |  foreach buffer instance b do
6  |  |  |  Set Temp = entity SYLVA_DataBuffer
7  |  |  |  Replace capacity in Temp with the capacity of b
8  |  |  |  Replace data width in Temp with the data width of b
9  |  |  |  Set i = ABB index (FIMP instance index)
10 |  |  |  Append suffix _j_i to SYLVA_DataBuffer
11 |  |  |  j = j + 1
12 |  |  end
13 |  end
14 end
15 Add Temp to result
```

Algorithm 7.5: Code Generation Algorithm for Buffer Instances for ASIC

Code Generation of Data Buffer Instance for FPGA

In the FPGA implementation style, the data buffers can either be implemented in ASIC style as shown above, or by using existing block memory. When using the FPGA specific memory resources, those memory resources have to be pre-wrapped to provide the same interface as the ASIC memory blocks. This resource mapping is decided in the design space exploration step. The following pseudo-code shows generating implementation code with specified memory blocks.

```
1 Set Temp = Implementation template
2 Set result = ∅
3 foreach given memory block M do
4 |  Set Temp = Wrapped M
5 |  Providing the same interface as ASIC memory blocks
6 |  Add Temp to result
7 end
```

Algorithm 7.6: Code Generation Algorithm for Buffer Instances for FPGA

Code Generation of Data Buffer Instance for CGRA

In the CGRA implementation style, the data buffer will be implemented by a vector with *capacity* number of elements. The type of the elements in one vector is decided by the output data type of the FIMP. The code generation algorithm is shown in Algorithm 7.7. For example, the output of a RGB to YUV converter has three components (Y, U and V) and each component is a decimal number. Thus, the data buffer for this RGB to YUV converter is instantiated as three vectors, whose elements are decimal numbers. If the converter is for one RBG to one YUV conversion, each vector has one element.

```
1  Set result = ∅
2  Set i = 0
3  foreach output port p in the FIMP instance do
4  |   foreach data buffer D in the ABB do
5  |   |   <name> = FIMP name concatenated with M_i
6  |   |   i = i + 1
7  |   |   <d> = capacity of D
8  |   |   Temp = <name> = zeros(<d>)
9  |   |   Add Temp into result
10 |   end
11 end
```

Algorithm 7.7: Code Generation Algorithm for Buffer Instances for CGRA

7.4.3 Code Generation of Output Selector

Code Generation of Output Selector for ASIC and FPGA

The output selector is a switch that connects the input ports to the corresponding output ports based on the control signals from the buffer control. The number of input ports of the output selector equals to the number of data buffers. The number of output ports equals to the number of data sink actors in the scheduled Synchronous Data Flow (SDF) graph multiplied by the number of output ports of the FIMP. For example, consider a 16-tap symmetric FIR filter, which is feeding an Fast Fourier Transform (FFT) and an 8-tap FIR filter. In this case, the output selector of the 16-tap FIR has one input port and two output ports.

In ASIC/FPGA implementation styles, the interface of the VHDL entity of an output selector is generated from the template shown in Code 8 (*FN* is short for function name). During code generation, <*FN*> will be replaced by the FIMP instance name, e.g. if two FIMP instances for 16-tap FIR filters have the same

```
1 entity SYLVAOutputSelector <FN> is
2 port (
3   clock : in std logic;
4   osel : in OS <FN>;
5   din : in inT <FN>;
6   dout : out outT <FN>
7 );
8 end SYLVAOutputSelector <FN>;
```

Code 8: Output Selector Template Example in VHDL

FIMP type, they will have different names such as *16FIR_1_1_FIMP_0* and *16FIR_1_1_FIMP_1*.

Line 4 declares the selection signal *osel* in the type of *OT<FN>*. This type is a vector, whose elements are integers from 0 to the number of output ports. Zero means no connection and other values are the one-based index of the output ports. The value of the i^{th} element indicates which input port is connected with the i^{th} output port. If *osel* = {0, 4, 1, 0, 0, 0}, the second input port is connected to the fourth output port, the third input port is connected to first output port, and other output ports are set to high impedance state. Lines 5 and 6 declare the input/output ports. For *din*, the number of elements equals to the number of input ports. The type of each element is the same as the corresponding data buffer output port. Algorithm 7.8 shows how to build the VHDL types and store it in the type library for this system.

Code Generation of Output Selector for CGRA

In CGRA implementation style, the output selector is constructed via a set of switch statements. Again, for the sake of brevity, the actual code is being skipped. The algorithm for generating codes of an output selector for CGRA is shown in Algorithm 7.9.

7.4.4 Code Generation of Input Selector

The code generation for the input selector is similar to the code generation of the output selector. The number of output ports, which connect the input selector to the FIMP, equals to the number of FIMP input ports. Similarly, the number of input ports, which connect the data sources to the input selector, equals to the number of data sources multiplied by the number of FIMP input ports. The generation procedure is exactly the same.

```
1  Set TL = FIMP template library file in VHDL
2  Set |data buffers| = number of data buffers
3  Set |FIMP output ports| = number of FIMP output ports
4  Set <I> = |data buffers| times |FIMP output ports|
5  Add type OS<FN> is array (<O> - 1 downto 0) of integer; into TL
6  Temp = type IE<FN> is (
7  foreach data port p in FIMP output ports do
8  |  Concatenate Temp with the type of p
9  end
10 Concatenate Temp with );
11 Add Temp into TL
12 Add type inT<FN> is array (<I> downto 0) of IE<FN>; into TL
13 Add type outT<FN> is array (<O> downto 0) of IE<FN>; into TL
14 Replace <FN> in TL with the FIMP name
15 result = entity SYLVA_ OutputSelector
16 Replace <FN> in result with the FIMP name
```

Algorithm 7.8: Code Generation Algorithm for Buffer Instances for ASIC

```
1  Set result = MATLAB function SYLVA_ OutputSelector
2  i = 0
3  foreach data sink do
4  |  Temp = switch In_ concatenate with i
5  |  i = i + 1
6  |  foreach data buffer d do
7  |  |  Add case concatenate with j into Temp
8  |  |  Add dout_ concatenate with i into Temp
9  |  |  Add = din_ concatenate with j into Temp
10 |  |  Add otherwise \n end into Temp
11 |  |  j = j + 1
12 |  end
13 end
```

Algorithm 7.9: Code Generation Algorithm for Buffer Instances for ASIC

7.4.5 Code Generation of FIMP Control

A buffer control is a Finite State Machine (FSM) controlling the data buffers and the output selector. For ASIC and FPGA, we use the generation method described in [108] for FSM code generation. For CGRA, one FSM is translated into two separate functions, which are *state control function* and *state output function*. Like the buffer control, the FIMP control and the global control are also FSMs. The code generation method is exactly the same as the buffer control.

7.4.6 Combine All Generated Components

After we have the implementation codes for all the individual components (FIMP, data buffer, output selector, input selector, buffer control, and FIMP control), we could combine them together into one VHDL module or MATLAB function. The combination algorithm is shown in Algorithm 7.10.

```
1  if ASIC or FPGA then
2  |  Top = final implementation VHDL entity
3  |  Add all components into architecture body
4  |  foreach component do Add its port into architecture body of Top
5  |  foreach ABB b do
6  |  |  Connect the components of b in Top
7  |  |  Connect the output selector with its data sink ABB in Top
8  |  end
9  |  Connect the ABB of the source/sink actor in the scheduled SDF graph
10 |  Include the library files generated for input/output selector
11 else
12 |  Top = final implementation MATLAB function
13 |  Add all functions for all the components
14 |  foreach component do Add its inputs and outputs as variables into Top
15 |  foreach ABB b do
16 |  |  Connect the components via the variables in Top
17 |  |  Connect the output selector with its data sink ABB in Top
18 |  end
19 |  Connect the ABB of the source/sink actor in the scheduled SDF graph
20 end
```

Algorithm 7.10: Combine All Components

7.4.7 Resource Allocation

In this step, each data buffer is mapped onto an actual memory block in the final system implementation. We first order the data buffers by their degree of connectivity (D), which is the sum of address range size multiplies the number of access actions as expressed in Eq. 7.7.

$$D = \sum(|A| \times NOA(A)) \; \forall \; A \in b \tag{7.7}$$

A is one used address range for the memory b, $|A|$ is the size of the address range, $NOA(A)$ returns the number of access actions to A and b is one memory access schedule in B (defined in the beginning of section 7.1). By allocating memory from the data buffer with the highest degree of connectivity, the communication energy is optimized since the available memory resources in the target implementation may be far away from the FIMP instance accessing it. This is similar to the method used in [87]. The algorithm is shown in Algorithm 7.11.

At this stage, we assume the CGRA has infinite memory resources since the data buffers are mapped on to MATLAB matrices and there is no limitation on the number of MATLAB matrices. We will improve it as a future work.

```
1 Sort B by the degree of connectivity
2 Set U = unused memory resources, k = 0
3 foreach b in B do
4 |   if U is not empty then
5 |   |   Allocate a memory block m for b as mm(k)
6 |   |   Remove m from unused resource
7 |   else  Create a VHDL design unit m for b as mm(k)
8 |   k = k + 1
9 end
```

Algorithm 7.11: Resource Allocation, ASIC and FPGA

7.5 Architecture Dependent Optimization

In System-Level Architectural Synthesis Framework (SYLVA), we focus only on the memory optimization. The optimization strategy is that we examine all the mapped memory blocks, mark the empty space and try to use that space for other data buffers. The empty space in the memory blocks due to the architecture independent optimization produced non-regular memory sizes. For example, 3.5 K bit memory block is fit into 4 K bit memory block and leaves 0.5 K unused space. In the architecture dependent optimization step, we use those unused space to implement some small data buffers that are accessed also in unused time intervals. The algorithm is shown in Algorithm 7.12.

```
1  Set Unused = All unused space
2  foreach b in B do
3  |  foreach empty space empty in Unused do
4  |  |  if b can be mapped onto empty then
5  |  |  |  Remove mm mapped for b
6  |  |  |  Map b onto empty
7  |  |  |  Update empty
8  |  |  |  Break
9  |  |  end
10 |  end
11 end
```

Algorithm 7.12: Code Generation Algorithm for FIMP Instances

The timing complexity is high due to the nested loops. However, we could further optimize the final system implementation. This optimization is useful especially for Field-Programmable Gate Array (FPGA), which has limited memory resources. It is obvious that using the on-chip memory resources is always better than using the LUTs in terms of resource usage and energy consumption.

According to our experiments, the allocated memory blocks are optimized enough and cannot be further optimized in the architecture independent optimization step. However, if we can move on-chip memory to off-chip, we will gain a lot in terms of on-chip gate count. Especially for Application-Specific Integrated Circuit (ASIC) implementation, less gate count on-chip would improve the yield and reduce the chip manufacturing cost. This is left as a future work for us.

7.6 CGRA Floorplanning

As described in chapter 1, System-Level Architectural Synthesis Framework (SYLVA) performs floorplanning when it targets Coarse-Grained Reconfigurable Architecture (CGRA). Considering the wide varieties of CGRA elements, efficient, effective, and automated compilation tools are the most important driver of CGRA development. In this section, we present the two CGRA floorplanners in SYLVA. Compared to Application-Specific Integrated Circuit (ASIC)/Field-Programmable Gate Array (FPGA) floorplanning, CGRA floorplanning does not consider the details such as the location of the power pads, power distribution, or clock distribution. The focuses of CGRA floorplanning are clock based communication synchronization and execution rescheduling, where design automation is highly demanded.

The proposed floorplanners automatically generate the optimized floor plan for hosting multiple scheduled communicating Function Implementation (FIMP) instances. The floorplanners also supports the sliding window circuit switching [109] Network-on-Chip (NoC) as the interconnection network. The global objective function is then as shown in Eq. 7.8. The object function can also be only minimizing the communication cost c, system sample interval t_s, system delay t_d, or total area a (utilization is not considered yet) by only setting K_C, K_S, K_T or K_A as one (rest set to zero), respectively.

$$\text{Minimize } C = K_C \cdot \frac{c}{C_{MIN}} + K_S \cdot \frac{t_s}{T_{s.MIN}} + K_T \cdot \frac{t_d}{T_{d.MIN}} + K_A \cdot \frac{a}{A_{MIN}} \tag{7.8}$$

The first part $K_C \cdot \frac{c}{C_{MIN}}$ is for making a compact system with the least communication cost. The constant K_C is a user defined constant and acts as a weight factor for representing the influence of the communication cost c over the total cost C. The constant C_{MIN} is the minimum or target communication cost that can be achieved regardless all other conditions (no area, sample interval, and latency constraints).

The communication cost c can be computed by Eq. 7.9, where s is the communication data size in number of bits and d is the corresponding communication distance in number of hops.

$$c = \sum s \cdot d \tag{7.9}$$

Assume the total communication data size between function f_i to f_j in one system iteration is denoted as $S_{i\to j}$. If $i = j$, then $S_{i\to j} = 0$. Assume the communication distance between the FIMP instance executing function f_i to the FIMP instance executing function f_j is denoted as $d_{i\to j}$. The communication cost c can be derived by Eq. 7.10.

$$c = \sum_{i=0}^{i<N} \sum_{j=0}^{j<N} S_{i\to j} \cdot d_{i\to j} \tag{7.10}$$

By default, SYLVA assume all the FIMP instances are rectangles and the input port is at the top-left and the output port is at the top-right. Assume we have a FIMP instance F_i executing function f_i with width W_i and height H_i and f_i will send data to function f_j that is executed by FIMP instance F_j. Also assume the input node of F_i is at (x_i, y_i) and output node of F_i is at $(x_i + W_i, y_i)$. Then we can derive that $d_{i\to j} = |x_i + W_i - x_j| + |y_i + H_i - y_j|$.

The second part $K_S \cdot \frac{t_s}{T_{s.MIN}}$ is for making a fast system with the least sample interval t_s. Since we are targeting the Digital Signal Processing (DSP) systems that with constant input/output data rate, the sample interval can be a used to measure the performance of the system. Similar to the communication cost part, the constant K_S is a user defined constant and acts as a weight factor for representing the influence of the sample interval t_s over the total cost C. The constant $T_{s.MIN}$ is the minimum sample interval that can be achieved regardless all other conditions (no energy, area, and latency constraints).

The sample interval t_s can be computed by Eq. 7.11, where t_i is the start time of function f_i, t_j is the start time of function f_j, F_i is the FIMP instance executing function f_i, F_j is the FIMP instance executing function f_j, $C_{i\to j}$ indicates if there is a connection from function f_i to function f_j. 1: true, 0: false. There can be multiple data paths with this length. In this thesis, we assume the data paths with the least distance are enough for all communications. If this is not so, no valid path will be produced and the corresponding floor plan is discard. The additional routing points will not affect the communication distance but the communication delay.

$$t_s = \text{MAX}(t_j - t_i) \;\forall\; F_i, F_j \text{ with } C_{i\to j} = 1 \tag{7.11}$$

The third part $K_T \cdot \frac{t_d}{T_{d.MIN}}$ is for making a fast system with the least latency t_d. Similar to the communication cost part, the constant K_T is a user defined constant and acts as a weight factor for representing the influence of the latency t_d over the total cost C. The constant $T_{d.MIN}$ is the minimum latency that can be achieved regardless all other conditions (no energy, area, and latency constraints).

The latency t_d can be computed by Eq. 7.12, where t_{N-1} is the start time of the last function and T_{N-1} is the execution time of the last function.

$$t_d = tN - 1 + T_{N-1} \tag{7.12}$$

The last part $K_A \cdot \frac{a}{A_{MIN}}$ is for making a compact system with the least area usage a. Similar to the communication cost part, the constant K_A is a user defined constant and acts as a weight factor for representing the influence of the area usage a over the total cost C. The constant A_{MIN} is the minimum area usage that can be achieved regardless all other conditions (no energy, area, and latency constraints).

The area usage a can be computed by Eq. 7.13, where x_i, y_i is location of FIMP instance F_i that executes function f_i. W_i is the width of F_i.

$$a = (\text{MAX}(x_i + W_i) - \min(x_i)) \cdot (\text{MAX}(x_i + W_i) - \min(x_i)) \tag{7.13}$$

The actual values of C_{MIN}, $T_{s.MIN}$, $T_{d.MIN}$, and A_{MIN} can be automatically computed or assigned by the system designer. They can be set to one if the actual value is not known. If C_{MIN} is assigned to its actual value, the term $\frac{c}{C_{MIN}}$ gives how good the current communication cost c comparing to the ideal case. If $T_{s.MIN}$ is assigned to its actual value, the term $\frac{t_s}{T_{s.MIN}}$ gives how good the current sample interval t_s comparing to the ideal case. If $T_{d.MIN}$ is assigned to its actual value, the term $\frac{t_d}{T_{d.MIN}}$ gives how good the current latency t_d comparing to the ideal case. If A_{MIN} is assigned to its actual value, the term $\frac{a}{A_{MIN}}$ gives how good the current area usage a comparing to the ideal case.

The proposed floorplanners can also be used to quickly find a valid floor plan, where the overall system latency t_d is smaller than the user defined maximum value T_{MAX}, for rapid design space exploration.

7.6.1 Sliding Window Circuit Switching

A CGRA consists of an array of atomic nodes and a mesh style Network-on-Chip (NoC). Each node has one Function Unit (FU), one or more Memory Units (MUs) and one switch. The interconnecting Network-on-Chip (NoC) can be packet switching or circuit switching with sliding window technique [109].

In the following part of this section, the notation of a node can be n_a or $n_{x,y}$ for indexing the node using its name (e.g. a) and location (x and y). The x and y start from top-left. The major advantage of circuit switching NoC over packet switched NoC is that it allows dedicated point to point physical connections for data paths [110]. The routing congestions could be avoided by assign more dedicated physical connections among switches. The upper bound of the number of physical connections is that each switch has dedicated connections to all other switches. In this case, no routing congestions would happen. However, the first thing we cannot afford to on a real chip is the large number of wires. This will make the layout too complex and will require many metal layers. Another thing we should avoid is the use of long wires, since they will reduce the communication performance and increase the power consumption due to the large capacitance.

The sliding window technique makes the NoC locally circuit switching and globally packet switching. Each node $n_{x,y}$ on the CGRA fabric has one window. The central node of this window is $n_{x,y}$. The nodes within this window can have dedicated connections to $n_{x,y}$. For the data communication within a window, circuit switching is used and it helps to avoid routing congestion. For the data communication between two nodes in different windows, at least one node should act as the relay point for buffering the incoming data and transmitting it to its final destination. There are two major advantages of using the sliding window technique.

1. The data communication benefits the use of dedicated connections.
2. The massive yet long wires are avoided to keep the communication performance high and communication energy low. The trade-off is the possibility of routing congestions.

Definition

For a $W \times H$ CGRA fabric, a node $n_{x,y}$, where $0 \leq x < W$ and $0 \leq y < H$, can only directly communicate with the set of nodes $D_{x,y}$ (Eq. 7.14) that is in the window.

$$D_{x,y} = \{n_{i,j} \mid x - W_X \leq i \leq x - W_X, y - W_Y \leq j \leq y - W_Y\} \tag{7.14}$$

W_X and W_Y are the maximum horizontal and vertical direct communication distances, respectively. The size of the window in terms of the number of nodes can be computed using W_X and W_Y. Denote the size of the window as $|D_{x,y}|$, which includes all the nodes in the window except the center node $n_{x,y}$. The expression for computing $|D_{x,y}|$ is given in Eq. 7.15.

$$|D_{x,y}| = (2 \cdot W_X + 1) \cdot (2 \cdot W_Y + 1) - 1 \tag{7.15}$$

Suitable values of W_X and W_Y and a good floor plan with low communication distances would reduce the possibility of routing congestions dramatically with few more long physical wires. W_X = W and W_Y = H means a non-windowed NoC. An example CGRA is illustrated in Figure 7.5. This CGRA has 56 nodes (width W = 8 and height $H = 7$). Each node includes one FU, one register file (MU) and one switch. We also assume that the nodes in the first row (red ones) have an extra SRAM block. In this example, $W_X = W_Y = 2$. Nodes $n_{2,4}$, $n_{4,2}$, and $n_{5,3}$ are the center nodes of window A, B, and C, respectively. The nodes within one window can directly communicate with the center node, e.g. $n_{3,2}$ can directly communicate with $n_{2,4}$, $n_{4,2}$, and also $n_{5,3}$ since $n_{3,2}$ is in the three windows.

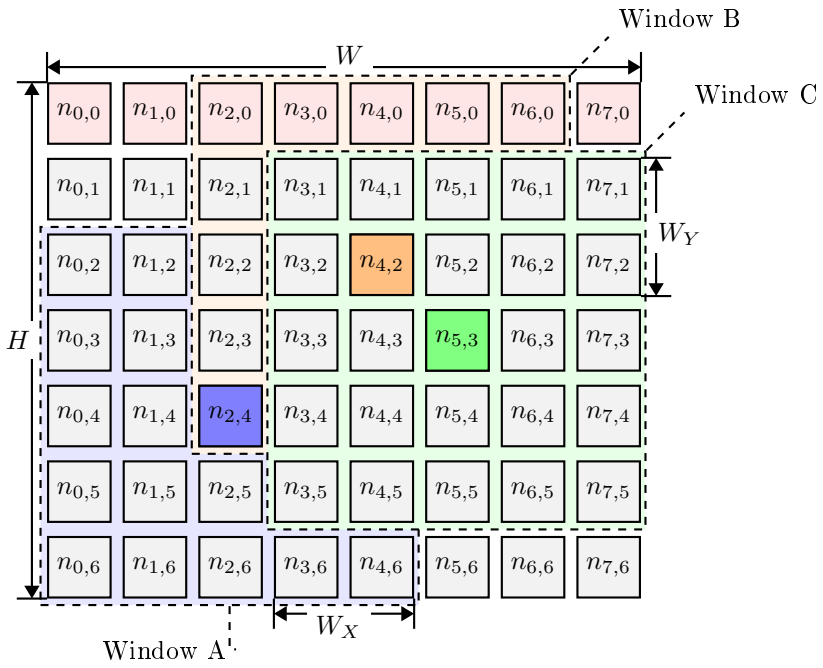
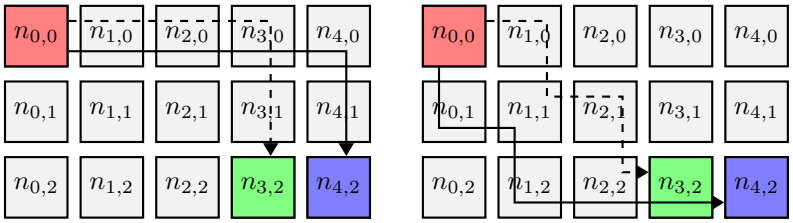


Figure 7.5: CGRA Structure

If one node cannot directly communicate with another node, e.g. $n_{2,4}$ and $n_{5,3}$, an variant of XY routing algorithm (IX/Y [111]) will be used. Since the NoC of our target CGRA is circuit switching, the IX/Y routing algorithm is more suitable than static XY or YX routing algorithm since it uses shorter channels to reduce the chance of channel conflict with the cost of more switches. Similar to the static XY or YX routing algorithms, IX/Y routing algorithm also chooses the shortest paths. The comparison of static XY and IX/Y routing algorithm is shown in Figure 7.6.

Denote the method to find a valid data path as FP, it requires four inputs: *destination*, *last_directions*, *initial_path*, *existing_paths*. The term *destination* is the location of the destination in the form of (horizontal location, vertical location), e.g. (i, j). The term *(*last_directions) indicates the last routing algorithm used. It is a $W \times H$ matrix of +1 (YX routing), -1 (XY routing), or 0 (not assigned). The term *initial_path* is the incomplete data path towards the destination node $n_{i,j}$. The term *existing_paths* records the occupied data paths in the CGRA. The algorithm of the FP method is shown in Algorithm 7.13. This recursive algorithm is for finding a path from $n_{x,y}$ to $n_{i,j}$.



(a) Static XY **(b)** Intermittent X, Y (IX,Y)

Figure 7.6: IX/Y Routing Algorithm

At line 6, rpX is the relay point horizontal position and rpY is the relay point vertical position. At line 7, $distanceX > W_X$ means the two nodes cannot directly communicate with each other. At line 8, $x + singX$ will result the neighbor node of $n_{x,y}$ that is on the way to $n_{i,j}$. The worst case is to explore all data paths between $n_{i,j}$ and $n_{x,y}$. The best case is to explore only one path. Assume in both cases $n = |i - x| + |j - y|$. The worst time complexity is $O(2n)$ (each node between them has to make XY or XY decision). The best time complexity is $O(n)$. The space complexity is $O(n)$ for both cases.

```
1  Set <x, y> = location of the last node in initial_path
2  Set distanceX = |x − i|, distanceY = |y − j|, <rpX, rpY> = <x, y>
3  Set signX = distanceX/(x − i), signY = distanceY/(y − j)
4  if distanceX > W_X then
5  |  rpX = max valid value from x + W_X · signX to x + signX
6  else
7  |  rpX = min valid value from x + distanceX · signX to x + signX
8  end
9  if distanceY > W_Y then
10 |  rpY = max valid value from y + W_Y · signY to x + signY
11 else
12 |  rpY = min valid value from y + distanceY · signY to y + signY
13 end
14 if rpX = x and rpY = y then
15 |  Remove the last node in initial_path
16 |  if initial_path = ∅ then Return ∅
17 |  if rpX = x then
18 |  |  Add node n_{rpX,rpY} into initial_path, last_directions[x, y] = 1
19 |  else if rpY = y then
20 |  |  Add node n_{rpX,rpY} into initial_path, last_directions[x, y] = -1
21 |  else
22 |  |  if last_directions[x, y] = 0 then dir = 1
23 |  |  dir = -last_directions[x, y]
24 |  |  nodeX = dir · rpX - dir · x, nodeY = dir · rpY - dir · y
25 |  |  Add node n_{nodeX,nodeY} into initial_path
26 |  |  last_directions[x, y] = dir
27 |  |  last_directions[nodeX, nodeY] = -dir
28 |  |  Add node n_{rpX,rpY} into initial_path
29 |  end
30 end
31 if rpX = i and rpY = j then
32 |  Add initial_path into existing_path, Return initial_path
33 else
34 |  Return FP(destination, last_direction, initial_path, existing_path)
35 end
```

Algorithm 7.13: Path Finding Algorithm of IX,Y

7.6.2 CP Solver Based Algorithm

Variables

The variables V to be assigned values are the locations of FIMP instances in the form of (horizontal position, vertical position). the variables V is defined by Eq. 7.16, where F_i is the i^{th} FIMP instances to be mapped onto the CGRA and N is the total number of FIMP instances to be mapped onto the CGRA.

$$V = \{(x_i, y_i) \mid \forall\ F_i,\ 0 \leq i < N)\} \tag{7.16}$$

Objective Function

We suppose the best floor plan is the one with the least communication cost c and/or with the least sample interval as defined in Eq. 7.17. The detailed description can be found in section 7.6.

$$\text{Minimize } C = K_C \cdot \frac{c}{C_{MIN}} + K_S \cdot \frac{t_s}{T_{s.MIN}} + K_T \cdot \frac{t_d}{T_{d.MAX}} + K_A \cdot \frac{a}{A_{MIN}} \tag{7.17}$$

Constraints

1. FIMP instances are inside the CGRA fabric.

$$0 \leq x_i + W_i < W, 0 \leq y_i + H_i < H$$

2. All FIMP instances are not overlapping. Denote the relative position of F_i and F_j as $(x_{i\to j},\ y_{i\to j})$, where $i \neq j$. The value = 0 means no overlapping and the value = 1 means overlapping. We have following constraints.

   - $x_{i\to j}$ in 0, 1
   - $y_{i\to j}$ in 0, 1
   - F_i is to the left of F_j

   $$x_i + W_i \leq x_j + W * (x_{i\to j} + y_{i\to j})$$

   - F_i is below F_j

   $$x_i - W_i \geq x_j - W * (1 - x_{i\to j} + y_{i\to j})$$

   - F_i is to the right of F_j

   $$y_i + H_i \leq y_j + H * (1 + x_{i\to j} - y_{i\to j})$$

   - F_i is above F_j

   $$y_i - H_i \geq y_j - H * (2 - x_{i\to j} - y_{i\to j})$$

3. Communication time can be computed using the following intermediate variables:

   - *direct_transfer* $(i, j) \in 0, 1$, 0 = false, 1 = true.
   - *direct_transfer* $(i, j) = |x_i + W_i - x_j| < W_X$ and $|y_i - y_j| < W_Y$.
   - *direct_transfer_time* $(i, j) =$ *direct_transfer* $(i, j) \cdot T_W$.
   - *indirect_transfer_time* $(i, j) =$ *reply_points* $(i, j) \cdot (T_C + T_W) + T_C$.
   - *delay* $(i, j) =$ *direct_transfer* $(i, j) +$ *indirect_transfer_time* (i, j).
   - *reply_points* $(i, j) > \max(\lfloor |x_i + W_i - x_j| \rfloor, \lfloor |y_i - y_j| \rfloor)$.
   - *reply_points* $(i, j) \leq |x_i + W_i - x_j| + |y_i - y_j|$.

4. The execution schedule can be computed as follows:

   - For all F_i, F_j that $C_{i \to j} = 1$, $t_j + (i > j) \cdot t_d > t_i + T_i +$ *delay* (i, j).
   - System latency (delay time) $t_d = t_{N-1} + T_{N-1}$.
   - System sample interval $t_s = \max(t_j - t_i)$ for all For all F_i, F_j that $C_{i \to j} = 1$.

5. The number of buffers for pipelined execution can be computed by introducing following intermediate variables.

   - $B_{i \to j}$: preset number of buffers before floorplanning.
   - $b_{i \to j}$: actual number of buffers after floorplanning.

   We also introduce a constant: the search space BS for the number of buffers to be searched so that we can limit the search time.

   $$B_{i \to j} - BS \leq b_{i \to j} \leq B_{i \to j} + BS$$

   Pipelining is possible when $\max(b_{i \to j}) \cdot t_s \leq t_d$.

6. The overall system delay should also be constrained that $t_d \leq T_{MAX}$.

7. The system sample interval should also be constrained that $t_s \leq R_{MAX}$

8. Special FIMP instances should have their own constraints, e.g. $y_i = 0$ for all F_i that have to be on the top row.

7.6.3 Heuristic Algorithm

The heuristic algorithm is inspired by the spiral algorithm proposed in [87]. The strategy is that FIMP instances in different layers of function executions are horizontally distributed and the FIMP instances within the same layer is distributed vertically on the CGRA. When the horizontal placement reaches the boundary of CGRA, turn back. The algorithm is shown in Algorithm 7.14.

```
1  Set layers = layers of function executions, direction = left to right
2  Map all special FIMPs (e.g. the ones have to be at row 0) from left to right
3  foreach layer in layers do
4  |  Try to map normal FIMP instances in layer below the last layer
5  |  if mapping fail then
6  |  |  Reverse direction and map FIMP next to the CGRA boundary
7  |  end
8  |  FIMPs of next layer to be mapped follows direction.
9  end
10 foreach data communication do
11 |  Assign a data path using the FP method
12 |  if FP fails then Return No valid solution
13 end
14 for MAX (b_{i→j}) · t_s < t_d − t_s do Add one pipeline stage
15 for MAX (b_{i→j}) · t_s > t_d do Remove one pipeline stage
```

Algorithm 7.14: Heuristic CGRA Floorplanning Algorithm

To illustrates this floorplanning algorithm, let us consider an Homogeneous SDF (HSDF) graph in Figure 7.7. It is randomly generated by using TGFF [112] and all the edge weights are set to 1/1.

The first subscript of each HSDF actor is its layer. They are different functions and each HSDF actor has its own FIMP instance. The window size is set to $W_X = W_Y = 3$. Figure 7.8 shows the floorplanning result.

In this example, 5 relay points (red ones) are added for data paths: from $F_{0,3}$ to $F_{0,4}$, from $F_{1,3}$ to $F_{1,4}$, and from $F_{1,3}$ to $F_{2,4}$; 2 relay points (blue ones)are added for data paths: from $F_{0,5}$ to $F_{0,6}$ and from $F_{2,5}$ to $F_{0,6}$; 4 relay points (orange ones)are added for data paths: from $F_{0,6}$ to $F_{0,0}$. The total number of relay points is 11.

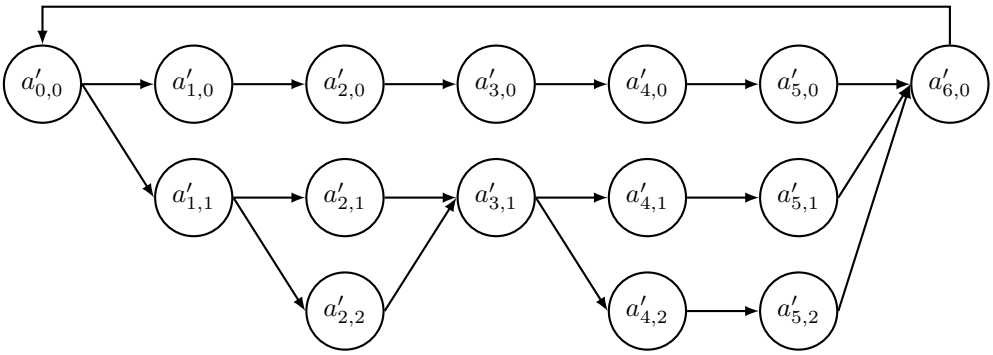


Figure 7.7: Floorplanning Example HSDF Graph

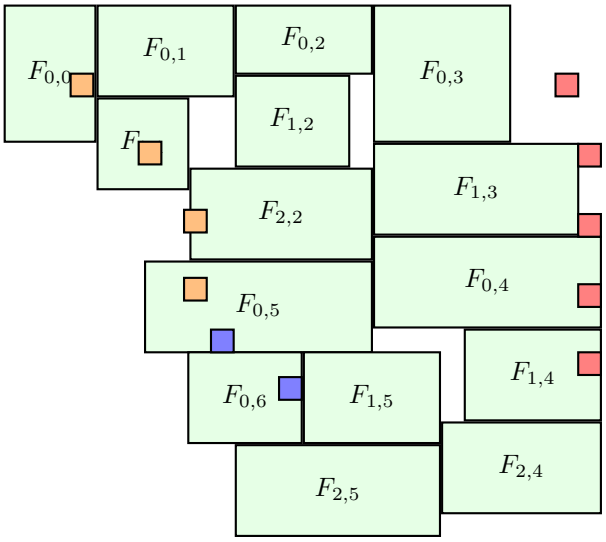


Figure 7.8: Floorplanning Example Result

7.7 Summary

This chapter elaborates the code generation in System-Level Architectural Synthesis Framework (SYLVA). Once the Global Interconnect and Control (GLIC) is ready, the next step is to generate the actual implementation. For Application-Specific Integrated Circuit (ASIC) and Field-Programmable Gate Array (FPGA) implementation styles, Register-Transfer Level (RTL) VHDL codes (separate entities for Function Implementation (FIMP) instances and a top entity for the interconnect and control) will be generated. For Coarse-Grained Reconfigurable Architecture (CGRA), MATLAB codes (separate m-functions for FIMP instances and a top script for the interconnect and control) will be generated. If specified, SYLVA can also perform CGRA floorplanning.

Chapter 8

Experimental Results

In this chapter, we present the experimental results of applying the System-Level Architectural Synthesis Framework (SYLVA) design flow to a list of small to large scale signal processing applications with given sampling interval and total latency constraints. The experimental results enable objective analysis of the core features of SYLVA, which are three-level Design Space Exploration (DSE) (chapter 5), automatically Global Interconnect and Control (GLIC) synthesis (chapter 6), and Coarse-Grained Reconfigurable Architecture (CGRA) floorplanning (chapter 7).

The sample applications are synthesized for three implementation styles: ASIC, FPGA, and CGRA. The target technology node is TSMC 65 nm. For the CGRA implementation style, we use Dynamically Reconfigurable Resource Array (DRRA) (section 2.2). It is a CGRA with cross-bar Network-on-Chip (NoC) communication scheme and its compiler VESYLA [73] supports MATLAB function with pragmas as input system model. DRRA supports a resource usage optimization method called Customization of Coarse Grain Reconfigurable (CRASIC). It shrinks the resource usage by cutting-off unused logics in the DRRA fabric. The numbers shown in this chapter are the numbers after CRASIC. The experimental results provide the basis for judging the following aspects. Note that we do not have the Function Implementations (FIMPs) for all implementation styles for all the applications. Therefore, some of the applications are only synthesized for one or two implementation styles for evaluating part of the SYLVA design flow.

1. Efficacy

   For each application, we examine the DSE of SYLVA. After feeding the Synchronous Data Flow (SDF) graph of each application to SYLVA, we measure the corresponding DSE time and the number of explored solutions. We also explicitly specify the Homogeneous SDF (HSDF) schedules to obtain solutions in different degrees of Function-Level Parallelism (FLP).

2. Efficiency

   For each application, we apply the SYLVA design flow and the High-Level Synthesis (HLS) design flow. Then, we measure and compare their total design times including the application modeling time, the synthesis time, and the verification time.

3. Accuracy

   SYLVA estimates the costs (area, energy, and time) of the synthesis result using pre-characterized FIMPs in the DSE step. Therefore, we also need to quantify the accuracy of this estimation. This is done by comparing the estimated costs to the costs of the synthesis result.

4. Quality of Result (QoR)

   In this thesis, we define the QoR of a synthesis result in two aspects:

   - Area usage in terms of equivalent gate count.
   - Energy consumption in terms of nano-joule (nj).

   This QoR judgment is done by comparing the automatically generated GLIC to the manually generated GLIC while keeping the used FIMP instances the same. The reason we do not consider the entire synthesis result is that SYLVA cannot change anything inside any FIMP instance and the FIMP instances are assumed to be optimal.

For a fair and meaningful comparison of the SYLVA generated design with the manually generated design, we restricted the manual design to GLIC, i.e. the manual design also used the same FIMPs as the SYLVA. We also restricted the manual designs to the same degree of function, arithmetic and buffer-level parallelisms. If the manual design uses a different parallelism, the comparison of GLIC would be meaningless. So the purpose of comparing manual versus SYLVA was to gauge how effective SYLVA is in automatically generating the GLIC.

8.1 Applications

In this section, we describe the 9 sample applications that are used for evaluating System-Level Architectural Synthesis Framework (SYLVA). Table 8.1 gives a brief description of the applications.

Table 8.1: Sample Applications

Application	Brief Description
1	Two FIR with one FFT
2	Correlation pool
3	Delta-sigma demodulator
4	Motion JPEG encoder
5	Simplified MPEG 2 encoder
6	IEEE 802.11a transmitter
7	IEEE 802.11a receiver
8	IEEE 802.11b transmitter
9	IEEE 802.11b receiver

All these applications have Synchronous Data Flow (SDF) actors *src* and *snk*. The SDF actor *src* is the data source actor that is used to supply data to the rest part of the application. The SDF actor *snk* is the data sink actor that is used to fetch the output data from the rest part of the application. In one SDF graph, there will be only one *src* actor and only one *snk* actor.

8.1.1 Synthetic Application: FFT and 2 FIR

This application has only two types of SDF actors for functionalities: *fft* and *fir* representing a 64-point FFT and a 15-tap FIR filter, respectively. The system is composed of two FIR filters feeding an FFT. The system SDF graph is shown in Figure 8.1.

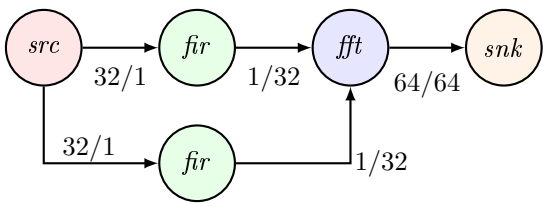


Figure 8.1: Application 1: FFT and 2 FIR

8.1.2 Correlation Pool

This application is part of the rake receiver in WCDMA physical layer. The system SDF graph is shown in Figure 8.2.

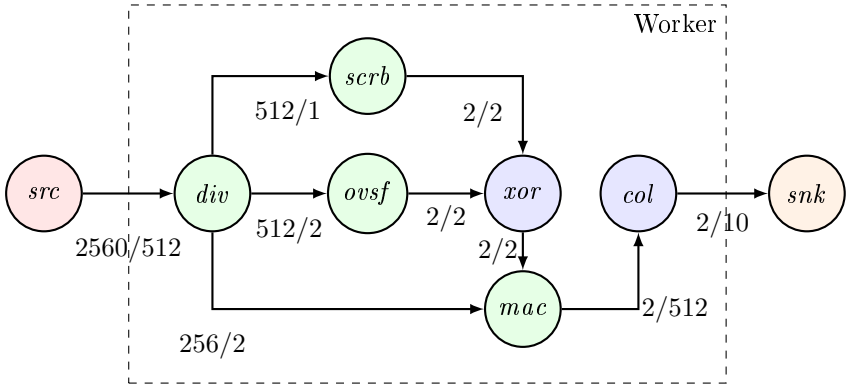


Figure 8.2: Application 2: Correlation Pool

It has six types of SDF actors for functionalities:

- *div*: the workload divider that only input 512 chips per firing, while the system throughput is 2560 chips per firing.
- *scrb*: scrambler code generator.
- *ovsf*: Orthogonal Variable Spreading Factor (OVSF) code generator.
- *xor*: simple bit-wise XOR logic operation unit.
- *mac*: Multiplication-Accumulation (MAC) unit.
- *col*: function to reorder the produced data from the *mac*.

The SDF actors *div*, *scrb*, *ovsf*, *xor*, *mac*, and *col* are considered as one *Worker*. Each worker has a throughput or 512 chips per firing as regulated by the SDF actor *div*.

8.1.3 Sigma Delta Demodulator

The sigma delta demodulator is for reconstructing one data token from 128 bits. The system SDF graph is shown in Figure 8.3.

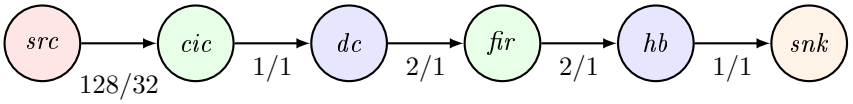


Figure 8.3: Application 3: Sigma Delta Demodulator

This application has four types of SDF actors for functionalities:

- *cic*: Cascaded Integrator-Comb (CIC) filter with a down-sampling rate of 1/32.
- *dc*: DC removal filter.
- *fir*: FIR filter with a down-sampling rate of 50%.
- *hb*: half-band filter with a down-sampling rate of 50%.

8.1.4 Motion JPEG Encoder

This application is a JPEG [113] encoder for full HD image (1920 × 1080) at 25 frame per second. The system SDF graph is shown in Figure 8.4.

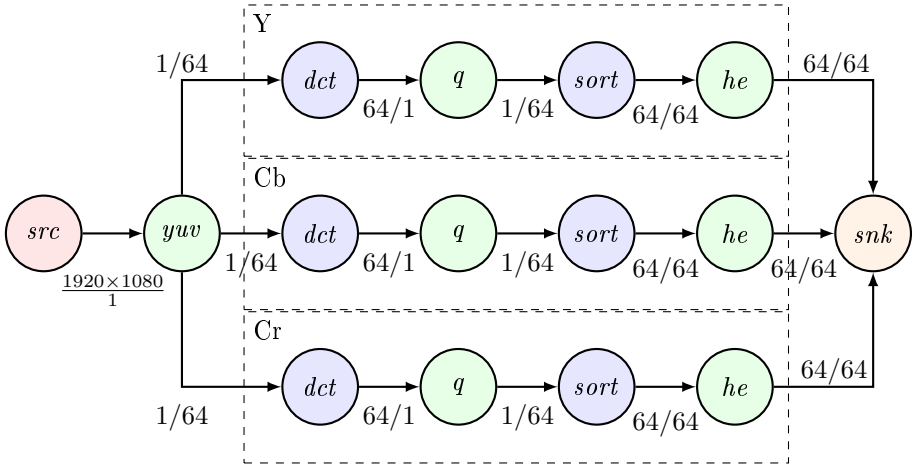


Figure 8.4: Application 4: Motion JPEG Encoder

It has five types of SDF actors for functionalities:

- *yuv*: RGB to YUV converter.
- *dct*: Discrete Cosine Transform (DCT) unit.
- *q*: quantizer.
- *sort*: quick sort unit.
- *he*: Huffman encoder.

8.1.5 MPEG2 Encoder

This application is a simplified MPEG 2 encoder [114] for compressing videos at 720 × 480 resolution and at 25 frames per second. The system SDF graph is shown in Figure 8.5. Note that the feedback edges in the SDF graph will be modeled by a data sink/source pair after the SDF graph is converted into Homogeneous SDF (HSDF) graph as described in section 4.2.2.

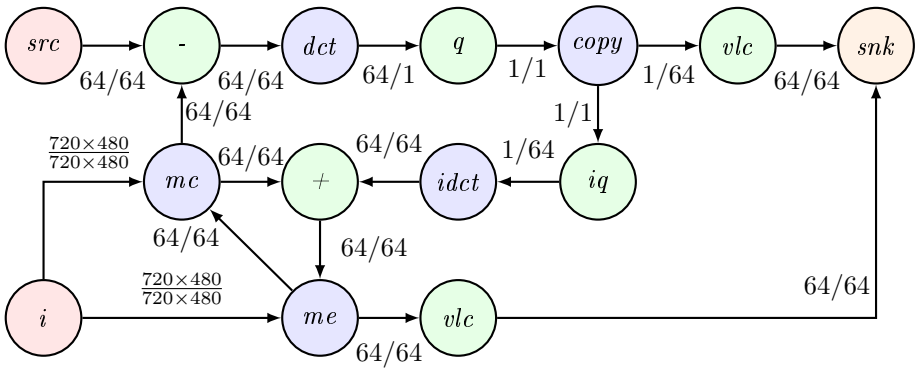


Figure 8.5: Application 5: MPEG2 Encoder

It has 11 types of SDF actors for functionalities:

- -: simple subtraction unit.
- *mc*: motion compensator.
- *me*: motion estimator.
- *i*: I image supplier.
- +: simple addition unit.
- *dct*: DCT unit.
- *idct*: inverse DCT unit.
- *q*: quantizer.
- *iq*: inverse quantizer.
- *vlc*: Variable Length Coding (VLC) unit.
- *copy*: data duplicator.

8.1.6 802.11a Transmitter

This application is the IEEE 802.11a physical layer transmitter. The system SDF graph is shown in Figure 8.6.

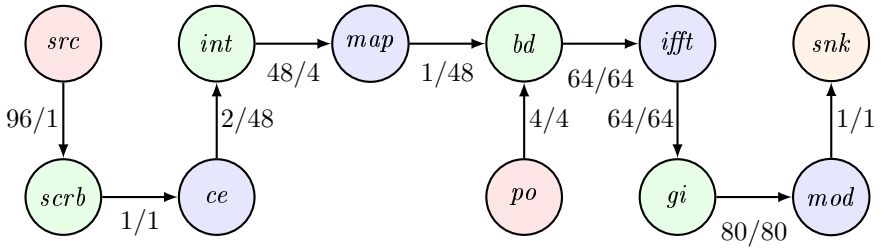


Figure 8.6: Application 6: 802.11a Transmitter

It has 9 types of SDF actors for functionalities:

- *scrb*: scrambler.
- *ce*: convolution encoder.
- *int*: interleaver.
- *map*: constellation mapper, e.g. 16-QAM.
- *bd*: data binding unit.
- *po*: pilot output.
- *ifft*: inverse FFT unit.
- *gi*: guard insertion unit.
- *mod*: IQ modulator.

The data tokens are bits before the actor *map* and are complex numbers after after it.

8.1.7 802.11a Receiver

This application is the IEEE 802.11a physical layer receiver. The system SDF graph is shown in Figure 8.7.

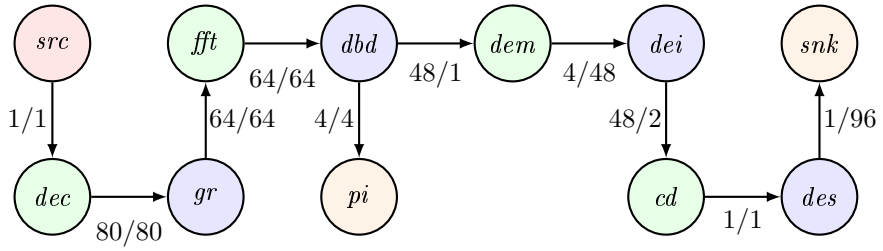


Figure 8.7: Application 7: 802.11a Receiver

It has 9 types of SDF actors for functionalities:

- *dec*: decimation interface.
- *gr*: guard removal unit.
- *fft*: FFT.
- *dbd*: data de-binding unit.
- *pi*: pilot input.
- *dem*: constellation demapper.
- *dei*: de-interleaver.
- *cd*: convolution decoder.
- *des*: de-scrambler.

8.1.8 802.11b Transmitter

This application is the IEEE 802.11b transmitter. The system SDF graph is shown in Figure 8.8.

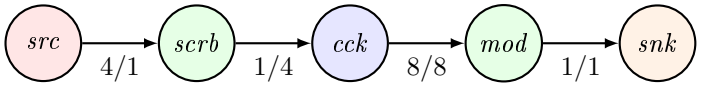


Figure 8.8: Application 8: 802.11b Transmitter

It has three types of SDF actors for functionalities:

- *scrb*: scrambler.
- *cck*: Complementary Code Keying (CCK) mapper.
- *mod*: IQ modulator.

8.1.9 802.11b Receiver

This application is the IEEE 802.11b receiver. The system SDF graph is shown in Figure 8.9.

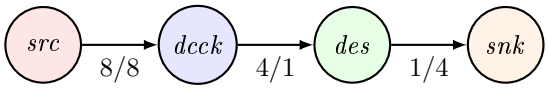


Figure 8.9: Application 9: 802.11b Receiver

It has two types of SDF actors for functionalities:

- *dcck*: CCK demapper.
- *des*: de-scrambler.

8.2 Experimental Procedure

The individual components of System-Level Architectural Synthesis Framework (SYLVA) are implemented in C# and Python. The SYLVA design flow is realized by integrating them at script level. This script was invoked for each of the applications with constraints and optimization objectives. The details of the applications can be found in section 8.1. The experimental procedure has three major steps:

1. Create the synthesizing environment and model the applications. In this step, we prepare the Function Implementation (FIMP) library including all the FIMPs that will be used for the system model. Before synthesizing the applications, there FIMP libraries were designed with great engineering effort and assumed to be optimal. Each FIMP library is for one implementation style among Application-Specific Integrated Circuit (ASIC), Field-Programmable Gate Array (FPGA), and Coarse-Grained Reconfigurable Architecture (CGRA). Each function in one FIMP library may have one to multiple FIMP types. All the FIMPs were synthesized using the same technology node (65 nm) and the same clock frequency constraint (1 GHz for ASIC, 200 MHz for FPGA, and 500 MHz for CGRA). The switching activity was set to 0.35 for characterizing the power consumption.

   For the SYLVA design flow, we model the applications using Synchronous Data Flow (SDF) graphs with constraints and optimization objectives. For the High-Level Synthesis (HLS) design flow, we model the applications using behavioral descriptions. The application modeling procedure is elaborated in chapter 4 with great detail. The behavioral description is either created for this thesis or imported from an external source [115] before being fed to the HLS tools. The details of all the sample applications can be found in section 8.1.

2. Synthesize the applications to Register-Transfer Level (RTL) using SYLVA and the HLS tools. The applications are synthesized by SYLVA and HLS tools. In this step, we provide individual timing and energy constraints for each application. We also recored all the synthesis information. For SYLVA, we record the synthesis result, overall synthesis time, as well as all the intermediate results including the Design Space Exploration (DSE) time, the number of explored solutions, the Global Interconnect and Control (GLIC) synthesis time, and the code generation time. For HLS, we only record the synthesis result and the overall synthesis time. The synthesis results from SYLVA and HLS tools are at RTL.

3. Synthesize the results at gate level. By synthesizing the RTL results to gate level, we could characterize the power consumptions more precisely. For both SYLVA and HLS design flow, we record the gate level power consumptions. Then we compare it to the estimated power consumptions to evaluate the accuracy of SYLVA. The synthesis results from SYLVA and the HLS tools are RTL descriptions, which are then fed to the corresponding back-end for gate level synthesis for each implementation style.

   - ASIC: Cadence RTL Compiler.
   - FPGA: Altera Quartus II.
   - CGRA: VESYLA compiler for DRRA.

8.2.1 Memory Allocation and Optimization

Besides the entire SYLVA design flow, we also evaluate the efficiency and efficacy of the proposed memory allocation and optimization method. The details of this method can be found in chapter 7. We apply the proposed method on five sample applications:

1. A synthetic application that one 64-point Fast Fourier Transform (FFT) is connected with two 15-tap Finite Impulse Response (FIR) filters (section 8.1.1).
2. The correlation algorithm in rake receiver[116] (section 8.1.2).
3. A sigma-delta demodulator [117] (section 8.1.3).
4. A JPEG encoder for motion JPEG [113] (section 8.1.4).
5. The motion estimation in a MPEG2 encoder [118] (section 8.1.5).

The SDF graphs of the five examples will be shown in section 8.3. We use Cadence RTL compiler to synthesize the resulting RTL description for ASIC and we use Altera Quartus II for FPGA implementation. Then we compare the automatically generated system implementations with the carefully manually generated implementations.

8.2.2 CGRA Floorplanning

The following five example applications are used to evaluate the efficiency and efficacy of the CGRA floorplanners of SYLVA.

1. The correlation algorithm in rake receiver[116] (section 8.1.2).
2. IEEE 802.11a physical layer transmitter section 8.1.6.
3. IEEE 802.11a physical layer receiver section 8.1.7.
4. A JPEG encoder for motion JPEG [113] (section 8.1.4).
5. The motion estimation in a MPEG2 encoder [118] (section 8.1.5).

We choose different sizes and orientations of the target CGRAs and solve the floorplanning problem using a Constraint Programming (CP) solver based and a Heuristic Algorithm (HA) based approaches. Then, we compare the efficiency and efficacy of HA over CP.

8.3 Synthesis Results

To evaluate the overall performance of System-Level Architectural Synthesis Framework (SYLVA), we use the following applications.

1. A synthetic application that is one 64-point Fast Fourier Transform (FFT) connected with two 16-tap Finite Impulse Response (FIR) filters.
2. The correlation algorithm in rake receiver.
3. A sigma-delta demodulator.
4. A JPEG encoder for motion JPEG at full HD resolution (1920×1080).
5. The motion estimation in MPEG2 encoder.

The details of these applications can be found in section 8.1. In this section, we only re-illustrate their Synchronous Data Flow (SDF) graphs to help for the sake of the readers.

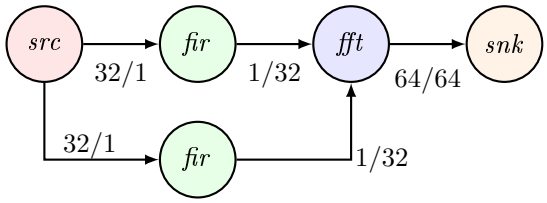


Figure 8.10: Application 1: FFT and 2 FIR

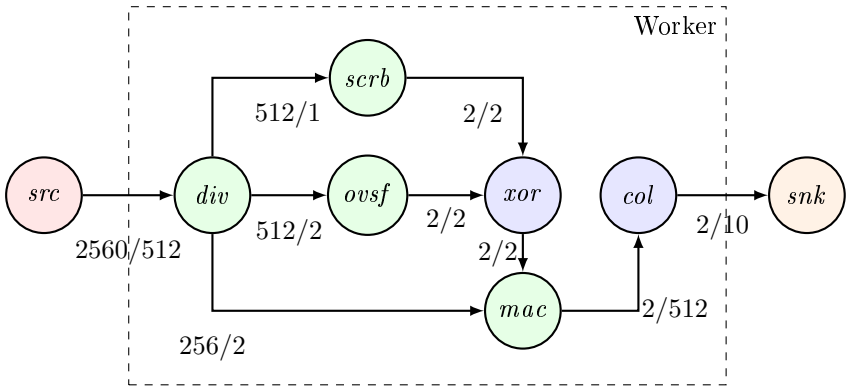


Figure 8.11: Application 2: Correlation Pool

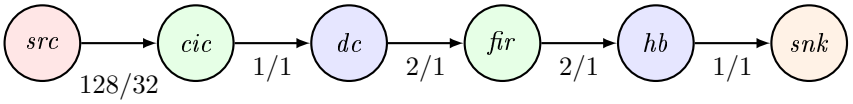


Figure 8.12: Application 3: Sigma-Delta Demodulator

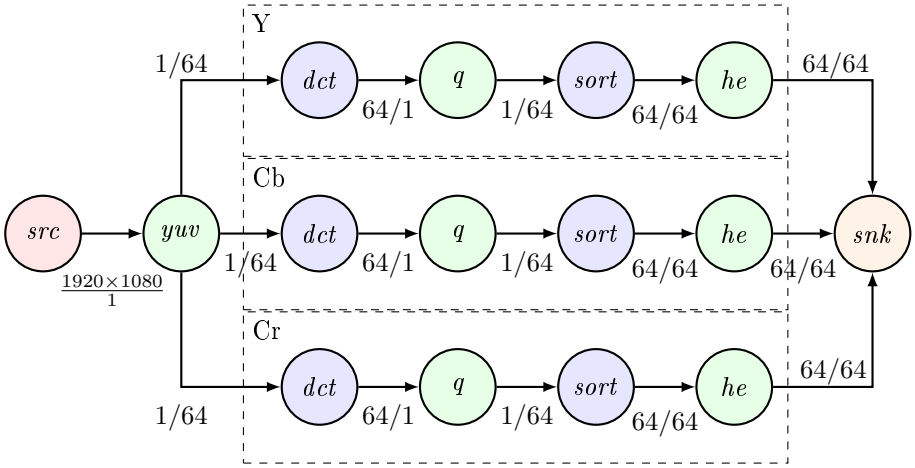


Figure 8.13: Application 4: JPEG Encoder

The individual components of the SYLVA flow have been implemented in Python and C# and integrated at script level. This script was invoked for each of the five applications (in the form of SDF graphs) with the maximum sampling interval (R_MAX) and the maximum total latency (T_MAX) as command line parameters. Function Implementation (FIMP) libraries with multiple FIMPs for the functions in the five examples for the three implementation styles were available. All the FIMPs in each of these three libraries were synthesized using the same clock frequencies, i.e., 1 GHz for Application-Specific Integrated Circuit (ASIC), 500 MHz for Dynamically Reconfigurable Resource Array (DRRA) and 200 MHz for Field-Programmable Gate Array (FPGA). The ASIC and the DRRA used the 65 nm technology. The FPGA used Cyclone III from Altera. This FPGA is also in the same 65 nm technology node.

The synthesis result in Register-Transfer Level (RTL) of the five sample applications were fed to the back-end for each implementation style. For the ASIC implementation, we used the Cadence environment (RTL Compiler and SoC Encounter), for the FPGA we used the Quartus II, and for DRRA we used the VESYLA compiler.

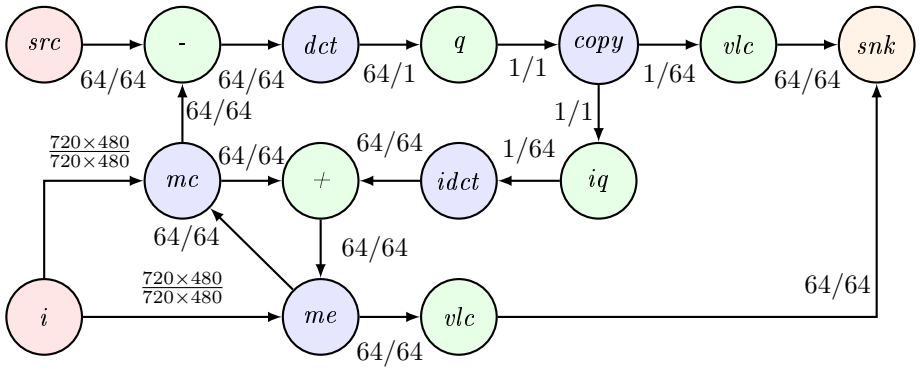
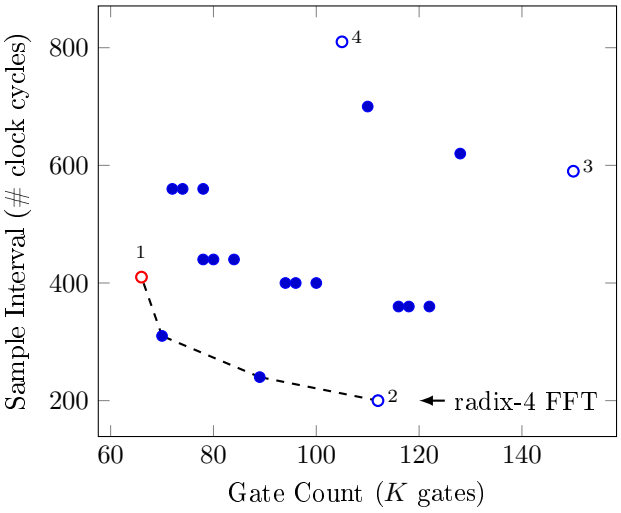


Figure 8.14: Application 5: Simplified MPEG 2 Encoder

We analyze the experimental results in three parts: Design Space Exploration (DSE), Global Interconnect and Control (GLIC) Synthesis, and accuracy of using pre-characterized FIMPs in DSE. The FIR-FFT example shown in Figure 8.10 is a Digital Signal Processing (DSP) sub-system with two 15-tap FIR filters and one 64-point FFT. The function *FIR* is the bottleneck as it consumes and produces one token at a time. However, in this example, it is not possible to enhance the system performance by increasing the number of FIMP instances for the function *FIR* via function level parallelism. The reason is that one FIR filter has an internal delay line, where the current output is dependent not just on the present input token but also on the previous 14 tokens as well. For this reason, the only sensible solution is using one FIMP instance for each *FIR* function.

The arithmetic level parallelism space is richer because the same function level parallel solution with two *FIR* can be implemented by more or fewer MACs (Multiply Add Accumulate) units. This design space is shown in Figure 8.3, where the total gate count (in K gates) of the selected FIR and FFT FIMPs and the sampling interval in number of clock cycles are shown.

Figure 8.15: Gate Count Versus Sample Interval



As expected, the general trend is that more gate count translates into smaller sampling interval. However, one can see that there are dots in the top right part, where the gate counts are quite high and yet the sample intervals are also higher than some other solutions in the bottom left part. This happens because these solutions have a bad mix of FIMP instances. Expensive *FFT* FIMP instances that increases the gate count quite a lot is combined with low cost *FIR* FIMP instances that increase the sample interval a lot. The Constraint Satisfaction Problem (CSP) solver picked up the solution shown as the left most solution in Figure 8.3 since it was the lowest gate count solution that met the sampling interval constraint.

8.3.1 Analysis of Design Space Exploration Results

We next analyze the synthesis results of SYLVA, which is shown in Table 8.2 and Table 8.3 (see the page after the next page). Table 8.2 lists the cost (area usage and energy consumption) and performance (system latency and sampling interval) of the synthesis result of the five sample applications. Table 8.3 gives the Function-Level Parallelism (FLP) and the Arithmetic-Level Parallelism (ALP) information of the synthesis results for evaluating the DSE step of SYLVA. As expected, SYLVA picks up solutions varying in their function and arithmetic parallelism depending on the provided constraints, nature of the problem, and the implementation style.

Application 1

As explained earlier in this section, there is no freedom in the function level due to the nature of the FIR filters. For the ALP, SYLVA decides to use the fully sequential solution,which uses a single MAC for each *fir* FIMP instance and a single radix-2 butterfly unit for *fft*, for ASIC implementation style since the ASIC FIMP instances are fast enough to sequentially process the data. For FPGA implementation style though, because of its lower speed, the implementation has to use a more expensive arithmetic solution in the form of 8 MACs and 1 radix-4 butterfly unit. For Coarse-Grained Reconfigurable Architecture (CGRA) implementation style, with its coarse grained structure, the arithmetic is nearly as efficient as ASIC and the same arithmetic solution suffices.

Application 2

For the correlation pool example, SYLVA generates solutions that differ only in FLP because the nature of functions is so simple that there are no multiple FIMP instances for them (SYLVA has only one choice). The ASIC implementation style is realized with 10 FIMP instances for each function of the worker, which is the sub-graph surrounded by the dashed box. The FPGA implementation style is realized with more FIMP instances (20 FIMP instances for each function of the worker) because the FIMP instances are slower than ASIC. Note that though CGRA is shown to use fewer FIMP instances but the CGRA FIMP instances are bigger composite ones that are twice the size of the ASIC FIMP instances to fit into the DRRA cell. This is due to the ASIC and CGRA libraries had different FIMP instances with different granularity allowing the SYLVA DSE to generate seemingly different solutions but in reality they have the same degree of function and arithmetic level parallelism.

Application 3

The sigma-delta demodulator is similar to the first example as it also has no freedom in the function level but SYLVA uses differing arithmetic level parallel solutions. The ASIC and CGRA implementation styles uses the FIMP instances with similar sizes since even the most sequential schedule is fast enough. The FPGA implementation style uses a much more complex set of FIMP instances. For example, the *htpb* (half-band filter) uses 48 MACs in the FPGA implementation but only 1 MAC in the ASIC and CGRA implementations.

Application 4

For the JPEG encoder, we artificially imposed a sampling rate of 25 frames per second to make an interesting test case otherwise there will be no timing constraints. In our case, we have freedom in both function and arithmetic levels. Unlike the previous examples,here even for ASIC the most serial solution did not suffice and SYLVA was forced to consider a more parallelized solution in function level but using the most serial implementation of the FIMP instances (ALP). For FPGA and CGRA,the synthesis result is parallelized in both function and arithmetic level to meet the constraints.

Application 5

For the simplified MPEG2 encoder, since the resolution is lower compared to the JPEG encoder (720×480 versus 1920×1080) both at 25 frames per second, SYLVA generates a solution that is cheaper than that of JPEG encoder in terms of area usage and energy consumption. For ASIC implementation style, it selected some parallelism in both function and arithmetic levels. Interestingly, for FPGAs and CGRA, it selected a serial solution at function level but increased the degree of parallelism in arithmetic level compared to the ASIC solution. This shows that SYLVA is able to search the entire design space and select a solution that reduces the overall gate count and energy.

Table 8.2: SYLVA Synthesis Result

	Manual / SYLVA	Area value[1]	Area error[2]	Energy value[3]	Energy error[2]	GLIC value[1]	GLIC %	T[4] ns	R[5] ns	CLK[6] MHz
Application 1	A	76.0		75		2.0	2.6	1634	1571	240
		76.4	2.0	76	4.9	2.4	3.1	1634	1571	240
	F	1225		2300		20	1.6	1760	1570	100
		2350	2.1	2320	2.4	45	3.6	1760	1570	100
	C	362		146		2.4	0.7	1634	1571	240
		363	0.7	146	0.9	2.6	0.7	1634	1571	240
Application 2	A	68		30		2.0	2.9	573	569	450
		70	0.3	32	1.6	4.0	5.7	591	571	450
	F	3511		835		402	11.5	643	640	200
		3894	10.9	865	9.3	402	11.5	655	643	200
	C	403		126		2.7	0.7	570	570	500
		405	1.3	126	1.3	5.0	1.2	570	570	500
Application 3	A	51		25		0.3	0.6	183	183	700
		53	1.5	25	1.0	2.3	4.3	183	183	700
	F	2156		310		99	4.6	182	182	200
		2176	5.1	310	4.2	120	5.5	182	182	200
	C	280		46		0.4	0.1	204	204	500
		283	1.0	46	0.9	2.9	1.0	204	204	500
Application 4	A	365.9		1.4K		1.8	0.5	42.3M	36.9M	1000
		366.0	0.6	1.4K	1.6	2.2	0.6	42.3M	36.9M	1000
	F	16.0K		17K		208	1.3	76.6M	33.6M	100
		16.2K	1.0	17K	3.1	405	2.5	76.6M	33.6M	100
	C	923		3.7K		2.5	0.3	45.7M	37.8M	500
		923	0.3	3.7K	1.1	2.5	0.3	45.7M	37.8M	500
Application 5	A	452.0		167		1.8	0.4	7.1K	6.0K	1000
		454.2	1.6	167	1.6	4.1	0.9	7.1K	6.0K	1000
	F	7092		2.4K		43	0.6	7.2K	6.1 K	100
		7152	4.1	2.4K	5.8	100	1.4	7.2K	6.1K	100
	C	843		276		2.5	0.3	7.4K	6.0K	500
		846	0.7	276	0.9	6.0	0.7	7.4K	6.0K	500

[1]: Area usage in number of 1K equivalent gates for ASIC and CGRA, and Logic Elements (LEs) for FPGA.
[2]: Percentage error.
[3]: Energy consumption pre system iteration in nano-Joule(nj).
[4]: System latency.
[5]: System sample interval (the time of each system iteration).
[6]: The required operating clock frequency.

Table 8.3: FLP and ALP of the SYLVA Synthesis Results

		Function Level Parallelism	Function Level Parallelism
Application 1	A	1 FIMP instance for each function	*fir*: 1 MAC *fft*: 1 radix-2 butterfly unit
	F	1 FIMP instance for each function	*fir*: 8 MACs *fft*: 16 radix-4 butterfly units
	C	1 FIMP instance for each function	*fir*: 1 MAC *fft*: 1 radix-2 butterfly unit
Application 2	A	10 FIMP instances for each function	*scrb*: 2 shift-registers, *ovsf*: 1 counter *mac*: 1 multiplier with 1 adder
	F	20 FIMP instances for each function	*scrb*: 2 shift-registers, *ovsf*: 1 counter *mac*: 1 multiplier with 1 adder
	C	5 FIMP instances for each function	*scrb*: 4 shift-registers, *ovsf*: 2 counters *mac*: 2 multipliers with 2 adders
Application 3	A	1 FIMP instance for each function	*cic*: 1 MAC, *fir*: 1 MAC *dc*: 1 MAC, *hb*: 1 MAC
	F	1 FIMP instance for each function	*cic*: 2 MACs, *fir*: 4 MACs *dc*: 8 MACs, *hb*: 48 MACs
	C	1 FIMP instance for each function	*cic*: 1 MAC, *fir*: 1 MAC *dc*: 1 MAC, *hb*: 1 MAC
Application 4	A	$9 \times yuv$, $6 \times q$, $3 \times dct$ $3 \times sort$, $1 \times he$	*yuv*: 1 MAC, *dct*: 1 MAC *q* : 1 multiplier
	F	$12 \times yuv$, $12 \times q$, $9 \times dct$ $9 \times sort$, $1 \times he$	*yuv*: 9 MACs, *dct*: 4 MACs *q* : 1 multiplier
	C	$2 \times yuv$, $6 \times q$, $3 \times dct$ $3 \times sort$, $1 \times he$	*yuv*: 9 MACs, *dct*: 1 MAC *q* : 1 multiplier
Application 5	A	$1 \times dct/idct$, $2 \times q/iq$ $1 \times me$, $1 \times vlc$, $1 \times mc$	*me*: 3 comparators, *dct*/*idct*: 1 MAC *vlc*: 4×1 comparator sort *q*/*iq*: 1 multiplier
	F	$1 \times dct/idct$, $1 \times q/iq$ $1 \times me$, $1 \times vlc$, $1 \times mc$	*me*: 5 comparators, *dct*/*idct*: 4 MACs *vlc*: 1×32 comparator sort *q*/*iq*: 1 multiplier
	C	$1 \times dct/idct$, $1 \times q/iq$ $1 \times me$, $1 \times vlc$, $1 \times mc$	*me*: 3 comparators, *dct*/*idct*: 1 MAC *vlc*: 1×4 comparator sort *q*/*iq*: 1 multiplier

8.3.2 Analysis of GLIC Synthesis

The experimental results are shown in Table 8.2 also gives the performance of the GLIC synthesis in SYLVA. In the GLIC column the SYLVA generated GLIC logics in terms of equivalent gate count for ASIC and CGRA, and of logic elements for FPGA are shown. We remind that the manual designs are manual only in designing the GLIC. The manual designs also use the same FIMP instances and the same degree of arithmetic and function level parallelism. As can be seen that the GLIC is a small percentage of the total area (at least 0.1% and at most 11.5% in the total resource usage) from Table 8.2. The values of the GLIC logic size and the runtime of the GLIC synthesis in the entire synthesis flow are shown in Table 8.4 for application 2 (correlation pool), application 4 (JPEG encoding), and application 5 (MPEG2 encoding).

Table 8.4: GLIC Percentage and Runtime

Application	Correlation Pool	JPEG	MPEG
GLIC % SYLVA	5.7	0.6	0.9
GLIC % Manual	2.9	0.5	0.4
Runtime (s)	0.5	2.4	4.3

The experimental result shows that the GLIC synthesis step within the SYLVA framework is able to generate GLIC logics comparable with the manual design (approximately double gate counts). The bulk of the area is consumed by the FIMP instances and the data buffer instances. The smaller percentage taken by the GLIC logics quantifies the high reuse factor and proves the efficacy of automatically reusing FIMPs as the granularity of reuse.

The automatically generated GLIC is slightly more than the manually generated GLIC. This gate count gap is cost by two factors. The first one is the Finite State Machine (FSM) based control scheme. No matter how simple the interconnect scheme is, several FSMs will be generated for each AIR Building Block (ABB). SYLVA has a template that it always instantiates a complete FSM for each function. The details can be found in chapter 6. We are currently working on post GLIC synthesis optimization to detect and eliminate these redundancies. The second one is the data buffer. In some cases, two FIMP instances can be directly connected if their input and output data matches in terms of timing and spatial. However, currently GLIC synthesis will allocate at least one data buffer instance for each FIMP instance.

8.3.3 Accuracy of DSE

Table 8.2 also shows area and energy values for the SYLVA and manually generated designs. The area numbers are from the synthesis reports and the energy numbers are generated by simulating the designs with back annotation of parasitic from physical design. The frequencies that SYLVA assumes when doing the DSE using the CSP solver is the max frequency at which they were synthesized and characterized. However, because the design space is discrete, the selected solution, even if it is the most serial, will have a positive slack. The achieved total latency and sampling interval is also shown in Table 8.2. This is exploited during the code generation phase to generate a lower cost logic solution. Note that this does not change the arithmetic or function level parallelism, just the logic and physical level realization.

For the FFT+FIR example (application 1), the ASIC solution has a significant negative slack and the code generation lowered the frequency to 240 MHz as that was enough to meet the sampling interval constraint. However, for the JPEG example, there is no negative slack and the frequency remains 1GHz at which the FIMPs were characterized.

To quantify the accuracy of using pre-characterized FIMPs, we have added columns in Table 8.2 to measure the error in final values of area and energy as compared to what was assumed during the DSE. As can be seen, this error is marginal with max error being 5.8%. This error comes from three sources.

1. The lowering of clock frequency changes the logic level realization.
2. During DSE, GLIC is ignored.
3. The third reason is the switching activity for the entire design can never be the same as that for which individual FIMP instances were characterized for. This reason applies only to the error in energy.

On average, the SYLVA runtime for the five sample applications are 15s, 13s, 20s, 74s, and 97s, respectively. We expect it to be stand out the high level synthesis tools.

8.3.4 Comparison with High-Level Synthesis (HLS) Tools

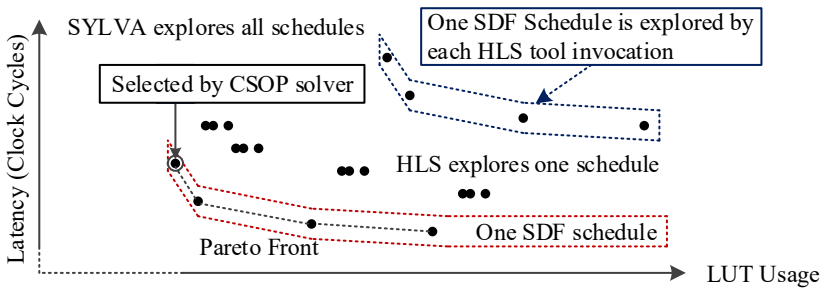
In this section, we validate the benefits of the three claims of the proposed DSE method: a wider DSE, the design space is explored more efficiently, and an accurate DSE. We compare the proposed DSE method in SYLVA with two commercial HLS flows. One of the flows uses Vivado from Xilinx and the other one is from a major EDA vendor (denoted as EDA). All the three flows target Xilinx FPGA (since we need to evaluate Vivado) and synthesis five applications (application 6-10). The experimental result is summarized in Table 8.3.4.

Table 8.5: Experimental Result

		A1	A2	A3	A4	A5
Exploration time	DSEM	13s	13s	9m	9m	38s
(h: hour, m: minute,	Vivado	1m	1m	2h	2h	2m
s: second)	EDA	2m	2m	2h	2h	2m
DSE Accuracy	DSEM	9	11	7	6	10
Compared with ISE	Vivado	17	15	11	10	14
synthesis result %	EDA	14	15	9	11	13
Quality of Result	DSEM	1.1	6.6	5.6	9.6	2.3
Resource Usage	Vivado	1.2	7.3	5.9	10.2	2.8
k LUTs	EDA	1.1	7.4	5.7	10.5	2.7

We also illustrate the schedules explored by SYLVA and the HLS tools in Figure 8.3.4, which is generated by plotting all the valid solutions instead of only the optimal one. The synthesis results, which are RTL models were verified by RTL simulation and further synthesized to FPGA bit-stream by the Xilinx ISE with the same constraints for fair comparison.

Figure 8.16: DSE: SYLVA Versus HLS



8.3.5 Memory Allocation and Optimization

We use Cadence RTL compiler to synthesize the resulting RTL description for ASIC and we use Altera Quartus II for FPGA implementation. Then we compare the automatically generated system implementations with carefully manually generated implementations. The experimental results for memory allocation and optimization are shown in Table 8.6. The values of the average runtime are measured on a Windows PC with 2.8GHz Intel core i7 CPU and 8G system main memory. Note that the CGRA case is not shown since the fully automation for CGRA implementation style is not finished yet. The CGRA floorplanning has to be manually integrated and the experimental result of it will be given later in this chapter.

Table 8.6: SYLVA Synthesis Runtime

Application	Method	ASIC		FPGA	
		Resource (K gates)	Average Runtime (s)	Resource total LEs	Average Runtime (s)
1	Manual	76		1225	
	Automatic	76	1	1250	11
2	Manual	68		$3.5k$	
	Automatic	70	1	$3.5k$	10
3	Manual	51		2156	
	Automatic	53	1	2176	14
4	Manual	365		$16.0k$	
	Automatic	366	2	$16.2k$	25
5	Manual	452		7092	
	Automatic	454	2	7152	30

For the ASIC implementation style, we do not have too many options other than mapping data buffers to the on-chip SRAMs. Thus the runtime is quite short. For the FPGA implementation style (Altera Cyclone III), we have some on-chip memory blocks so the runtime is longer than the ASIC implementation style. Note that since we do not have access to existing SDRAM controller IPs, we cannot evaluate the proposed method with off-chip memory blocks and this is one of our future work. The liveness analysis in the architecture dependent optimization substep considers the data token set transmitted between two actors as the analysis object instead of one data token.

8.3.6 CGRA Floorplanning

The experimental results of the CGRA floorplanning are shown in Table 8.7, where t_r is the run time, c is the total cost, ΔB is the number of added pipeline stages, t_s is the sample interval, and t_d is the system latency obtained by Constraint Programming (CP) and HA. The run time for CP and HA are also measured on a 2.8 GHz x86 Windows machine as other experiments. By comparing the CP and HA results we have the following conclusions:

1. For a small scale and simple problem like application 2, HA is able to get the optimal solution as CP.

2. HA is more suitable for the CGRAs with $W > H$ that is due to the default mapping direction is horizontal from left to right, and SYLVA cannot rotate the FIMP instances currently. When the FIMP instances have larger widths and smaller heights, HA may become less efficient. This may also happen when the number of layers is larger than the number of FIMP instances.

3. The run time of HA is approximately proportional to the number of unmapped FIMP instances while the run time of CP is approximately exponential to the number of unmapped FIMP instances since none of the sample applications has high utilization of the communication channels.

4. HA beats CP heavily in run time. The efficiency of HA takes maximum 1.58 ‰ run time of CP. The trade-off is the quality of result. HA cannot guarantee to get optimal solution. In the worst case, HA solution takes 13% more overall communication cost and 6% larger sample interval than the corresponding CP solution.

Table 8.7: CGRA Floorplanning Results

	N	CGRA		CP		HA		HA/CP		ΔB	
		W	H	t_r	c	t_r	c	t_r	c	CP	HA
Application 2	8	4	4	12	15	8	15	0.67	1.00	0	0
		6	2	13	15	8	15	0.67	1.00	0	0
		5	3	12	15	8	15	0.67	1.00	0	0
		2	6	13	15	9	17	0.69	1.13	0	0
		3	5	12	15	9	17	0.75	1.13	0	0
Application 6	10	16	16	60	86	71	96	1.18	1.12	0	0
		24	8	61	86	70	86	1.15	1.00	1	1
		20	12	62	86	71	86	1.15	1.00	0	1
		8	24	60	86	95	96	1.58	1.12	1	0
		12	20	61	86	91	96	1.49	1.12	0	0
Application 7	10	16	16	59	86	58	96	0.98	1.12	0	0
		24	8	58	86	58	86	1.00	1.00	1	1
		20	12	58	86	59	86	1.02	1.00	0	1
		8	24	58	86	69	96	1.19	1.12	1	0
		12	20	59	86	66	96	1.12	1.12	0	0
Application 4	15	16	16	109	38	91	39	0.83	1.03	0	0
		24	8	108	37	89	38	0.82	1.03	0	0
		20	12	108	37	89	38	0.82	1.03	0	0
		8	24	109	37	101	41	0.93	1.11	0	0
		12	20	107	37	96	41	0.90	1.11	0	0
Application 5	14	40	40	12K	49	262	51	0.02	1.04	0	0
		60	20	12K	49	274	49	0.02	1.00	1	1
		50	30	12K	49	265	49	0.02	1.00	1	1
		20	60	12K	49	294	52	0.02	1.06	1	0
		30	50	12K	49	289	53	0.02	1.08	1	0

N: The number of functions in the original synchronous data flow.
t_r: The run time in number of seconds for CP and milliseconds for HA, ‰ for CP/HA.
c: The normalized total cost in the range of 0 to 999 in CP and HA.
ΔB: The number of pipeline stages added during floorplanning.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

This thesis presents System-Level Architectural Synthesis Framework (SYLVA), which is an evolutionary next step after High-Level Synthesis (HLS). It synthesizes abstract Digital Signal Processing (DSP) sub-systems modeled as Synchronous Data Flow (SDF) graphs in terms of pre-characterized Function Implementations (FIMPs). SYLVA explores the design space in terms of number and type of FIMP instances, as well as pipeline parallelism between FIMP instances. It automatically generates Global Interconnect and Control (GLIC) for ASIC and FPGA implementation styles to glue the FIMPs together into a working system.

9.1.1 GLIC Synthesis

In this thesis, GLIC synthesis is shown to be effective reducing the manual effort from the system designer for building GLIC logics based on a static schedule. Since the abstraction level of HLS is already high, we consider the automatic GLIC synthesis would be a new and crucial challenge for researchers to further improve the design productivity in the automatic hardware synthesis domain. And we expect the concept of the GLIC synthesis in the SYLVA framework could also be used by other synthesis tools. However, the current GLIC synthesis approach is elementary. The automatically generated GLIC costs more area (approximately twice) compared to the manual design. One direct approach to shrink the automatically generated GLIC is to perform the liveness analysis for each data token. So the total size of data buffers could be reduced. However, this would involve more computation time.

Another possible approach to reduce the GLIC size is to change the structure of AIR Building Block (ABB) based on the nature of the FIMP instance so that the data buffers could be eliminated and two FIMP instances can directly connected to each other when possible. However, this also requires a huge amount of analysis and the timing model of FIMP should also be more accurate that each data token should has its own timing information. Then the data token can be used to check if two FIMPs can be directly connected without any manipulation of the input/output data schedule. However, from the system perspective, this timing information may be dramatically large and nearly impossible to be used in practical settings.

9.1.2 Code Generation

The proposed code generation method used in the SYLVA framework generates VHDL modules (for ASIC and FPGA implementation styles) or composite MATLAB function for Coarse-Grained Reconfigurable Architecture (CGRA) implementation style from the Abstract Intermediate Representation (AIR). The code generation process is automatic and efficient. This code generation can also be used in standalone mode. By applying the proposed memory allocation and optimization method, the code generation in the system level synthesis is automated. Thus, the design productivity is significantly improved. Moreover, design space explorations (at function level and arithmetic level) can be performed in less time and require less effort form the system designer. We also expect the proposed generic memory model could be used in other synthesis framework since it models memory blocks in a generic manner.

9.1.3 CGRA Floorplanning

In this thesis, we define the sliding window circuit switching CGRA model, apply the FIMP concept, convert the pipeline buffer-aware floorplanning problem into a constraint satisfaction optimization problem, and solve it using ***(a)*** an open-source solver (OR-Tools from Google [103]) and ***(b)*** a heuristic algorithm. As reported in section 8.2.2, the heuristic algorithm uses at most 1.58 ‰ run time comparing to the constraint programming approach with the cost of, at most, 13% additional overall communication cost in average for the test cases.

9.2 Future Work

This section summarizes some possible future work for each chapter in this thesis.

9.2.1 System Modeling

In order to deal with a wider classes of problems, one future work could be using one or more extensions of Synchronous Data Flow (SDF) such as Scenario-Aware Data Flow (SADF) [119] as input for supporting dynamic data rate applications. However, this may also require a huge improve of the System-Level Architectural Synthesis Framework (SYLVA) framework to support dynamic data rate.

9.2.2 Design Space Exploration

1. Improve the Constraint Satisfaction Optimization Problem (CSOP) model, e.g. using more global constraints and cumulative constraints instead of using primitive constraints.

2. Evaluate other search techniques like Tabu search [120] to exploit the fact that solutions often appear in clusters.

3. Eliminate redundant data buffers when possible, e.g. when two Function Implementations (FIMPs) can be directly connected without output buffer.

4. Time-multiplexing of FIMP instances is also a possible future work.

9.2.3 Code Generation

1. Improve the Coarse-Grained Reconfigurable Architecture (CGRA) capability together with the development of VESYLA compiler (the Dynamically Reconfigurable Resource Array (DRRA) compiler).

2. Verify and evaluate the proposed memory allocation method on off-chip memory blocks. As stated before, we have a method that is capable to model off-chip memory blocks but we do not have the related experimental results. The reason is that currently we do not have access to existing SDRAM controller IPs.

3. Another future effort could be the code optimization to achieve more compact implementation codes. For example, the size of the type library used for the Application-Specific Integrated Circuit (ASIC)/Field-Programmable Gate Array (FPGA) VHDL entities can be reduced by merging equivalent data types.

9.2.4 GLIC Synthesis

1. Find a solution that can reduce the scale of the generated logic according to the computation time assigned with the Global Interconnect and Control (GLIC) synthesis step. This requires more research efforts on the GLIC optimization and the structure of AIR Building Block (ABB).
2. Factor in the cost of GLIC and the slack related reduction in logic to make the Design Space Exploration (DSE) more precise.
3. Implement rule based optimization of GLIC by SYLVA.
4. Compare the runtime of SYLVA with the commercial High-Level Synthesis (HLS) tools on more sample applications.

9.2.5 CGRA Floorplanning

1. Make the proposed algorithm more adaptive to the orientation of the CGRA and the shape of the FIMP,
2. Make the algorithm and constraint programming model utilization-aware (as low utilization of CGRA is much more costly than of FPGA),
3. Test the algorithm on a Network-on-Chip (NoC) with high communication channel utilization,
4. Improve the heuristic algorithm so that valid t_s and t_d are guaranteed, and
5. Model and solve this CGRA floorplanning problem for the sliding window circuit switching CGRAs using Mixed Integer Linear Programming (MILP) and Simulated Annealing (SA) to have a more accurate comparison to other floorplanning tools.

System Modeling Example

In this appendix, an example of modeling the Long Term Evolution (LTE) uplink transmitter will be given to clarify the flow of the system modeling in System-Level Architectural Synthesis Framework (SYLVA). The block diagram of the LTE uplink transmitter is shown in Figure 9.1. The functions and the data dependency are the starting point for creating the Synchronous Data Flow (SDF) graph. Each of the functions in Figure 9.1 will be modeled by one SDF actor. For Application-Specific Integrated Circuit (ASIC) and Field-Programmable Gate Array (FPGA), SYLVA assumes that every Function Implementation (FIMP) has a clock signal *clk* and an active low reset signal *nrst*. For Coarse-Grained Reconfigurable Architecture (CGRA), SYLVA assumes that every FIMP has an active low reset signal *nrst*. Those signals are not considered as data inputs to the SDF actors and will not be considered in the SYLVA design flow till the code generation phase.

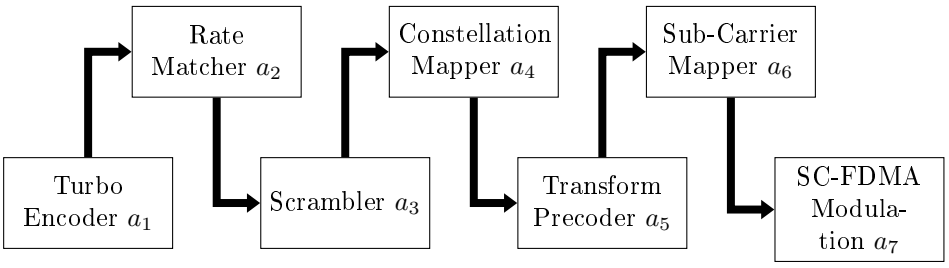


Figure 9.1: LTE Uplink Transmitter Block Diagram

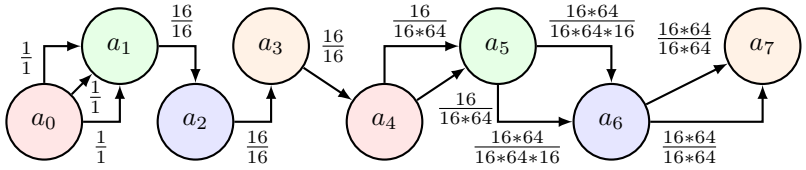


Figure 9.2: LTE Uplink Transmitter SDF Graph (Flexible)

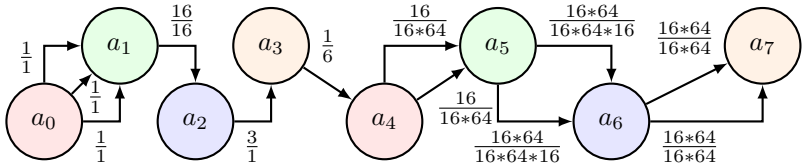


Figure 9.3: LTE Uplink Transmitter SDF Graph (Dedicated)

The entire system can then be modeled as a flexible model (Figure 9.2) or as a dedicated model (Figure 9.3), which is for 64-QAM and $R = r = 1/3$. The dedicated system can be automatically generated based on the given parameters by constructing the SDF actors using input parameters.

Turbo Encoder

The turbo encoder implements a channel coding for controlling errors in data transmission over noisy channels that enable reliable delivery of digital data. The turbo codes [121] belong to a high-performance forward error correction family of codes that are widely used in 3G/4G telecommunication.

Assume the FIMP library has a more general turbo encoder than the one only for LTE that the encoding rate of the turbo encoder FIMP can be changed at run time. The encoding rate R of the turbo encoder depends on the information bit length m and the parity bit length n as shown in Eq. 9.1.

$$R = \frac{m}{(m+n)} \tag{9.1}$$

Note that m is always 1 and the run time configuration can be done by changing the value of n. Assume that the turbo encoder FIMPs support n to be 1 or 2. In LTE, we use $m = 1$, $n = 2$, thus, $R = 1/3$. The input to the turbo encoder SDF actor is the data input D_{in}, and the parity bit length n. The data token type of D_{in} is *std_logic* representing one bit per invocation. The data token type of n is also *std_logic* representing the value of 1 and 2 using '0' and '1', respectively. The output from the turbo encoder consists of the data output D_{out} and two parity bits P_0 and P_1. If $n = 1$, $R = 1/2$ and P_1 has no meaning. If $n = 2$, $R = 1/3$ and P_1 is the second parity bit. The data token types of D_{out}, P_0, and P_1 are all *std_logic*. The data counts for all the inputs and outputs of the turbo encoder are all 1's indicating that each execution of the turbo encoder consumes or produces one data token on each input or output. The SDF actor of turbo encoder, which is denoted as a_0, is depicted in Figure 9.4.

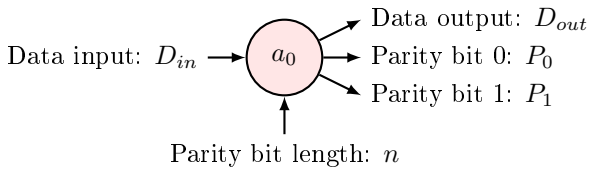


Figure 9.4: Turbo Encoder SDF Actor

Rate Matcher

The rate matcher provides rate conversion from a fixed mother code with a preset rate to a custom channel rate for increasing the flexibility of the system in terms of the performance-complexity trade-off of channel coding. The rate matching is performed by puncturing or repeating the coded bits based on the mother rate and the custom channel rate.

Similar to the turbo encoder, the rate matcher can also be changed at run time. The input to the rate matcher consists of the data input D_{in}, the mother (input) coding rate R (represented by R_m and R_n), and the channel (output) coding rate r (represented by r_m and r_n). The data input D_{in} consists of the data output and the parity bits from the turbo encoder. Therefore, its data token type is *std_logic_vector (m_{MAX} + n_{MAX} - 1 downto 0)*. m_{MAX} is the maximum information bit length and n_{MAX} is the maximum parity bit length. indicating that there should be three bits presented at the same time. We assume that $m_{MAX} = 8$ and $n_{MAX} = 8$. The data token count is 1 indicating that at each invocation only one data token is consumed. The data token type of the information bit length R_m in the mother coding rate R is *integer range 1 to m_{MAX}* and the data token type of the parity bit length R_n in the mother coding rate R is *integer range 1 to n_{MAX}*. Similarly, the data token type of the information bit length r_m in the channel coding rate r is *integer range 1 to m_{MAX}* and the data token type of the parity bit length r_n in the channel coding rate r is *integer range 1 to n_{MAX}*. Therefore, the case of $R = r$ is supported. The output from the rate matcher is the data output D_{out}. The data token types of D_{out} is *std_logic_vector (m_{MAX} + n_{MAX} - 1 downto 0)* and its data token count is one. The SDF actor of turbo encoder, which is denoted as a_2, is depicted in Figure 9.5.

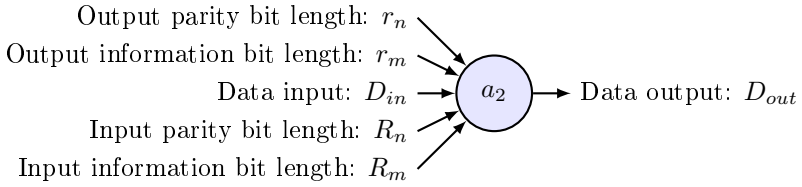


Figure 9.5: Rate Matcher SDF Actor

In LTE, the data token types of the outputs from the turbo encoder are all *std_logic* and the total data token count is 3. In this case, we have to manually add one *data size matcher* that convert three *std_logic* data tokens to one *std_logic_vector (m_{MAX} + n_{MAX} - 1 downto 0)* data token. We need to implement the data size matcher and add the corresponding SDF actor into our system. Assume $m_{MAX} = 8$ and $n_{MAX} = 16$. The input to the data size matcher consists of the data input D_{in}, the parity bit 0 input P_0, and the parity bit 1 input P_1. The output from the data size matcher is denoted as D_{out} and it is a combined data structure consisting of the information bit and the two parity bits. The SDF actor of the data size matcher, which is denoted as a_1, is depicted in Figure 9.6.

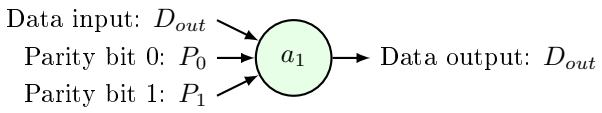


Figure 9.6: Data Size Matcher SDf Actor

The FIMP of the data size matcher can be automatically generated by SYLVA once the input ports and the output port are defined by the designer. The matching is done by assigning the input ports the the output port sequentially. The details of the FIMP generation can be found in chapter 7.

Scrambler

The scrambler introduces pseudo-noise to the data bits that makes it unintelligible to a potential eavesdropper. The bit stream in a sub-frame is scrambled with a user equipment specified scrambling sequence in the transmitter and can be reversed by descrambling at the receiver.

Similar to the turbo encoder and the rate matcher, the scrambler is for the general cases that the length of the scrambling code cam be changed at run time. The input to the scrambler consists of the data input D_{in} and the scrambler code length s. The data token type of D_{in} is *std_logic* and the data token type of s is *integer range 0 to 7* indicating the scrambling code lengths of 4, 8, 16, 32, 64, 128, 256, and 512, respectively. The data output from the scrambler is the data output D_{out}. Its data token type is *std_logic*. The SDF actor of the scrambler, which is denoted as a_3, is depicted in Figure 9.7.

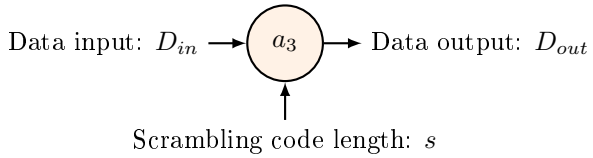


Figure 9.7: Scrambler SDF Actor

Note that the scrambler cannot be executed in parallel without modification of the scrambling code otherwise the output data will be wrong. Therefore, we mark the number of FIMPs used for the scrambler as one ($F_3 = [\ f\ ]$, where f is a dummy FIMP serves as a place holder) indicating that only one FIMP will be used for the scrambler no matter how many times it should be invoked. Since the output of the rate matcher is $m_{MAX} + n_{MAX} = 8 + 8 = 16$ and the input to the scrambler is one, we need to make a custom wrap around the scrambler such that it takes 16 bit wide data input as well as the channel coding rate parameters r_m and r_n as the inputs. The output should be $m_{MAX} + n_{MAX}$ wide to prevent loss of information. For a given r_m and r_n combination, the number of invocations of the scrambler equals to $r_m + r_n$. We now have two choices. The first one is designing the wrapped scrambler in such a way that it will invoked multiple times internally but considered as a black box from top of it. In this case, we assume the execution time of this wrapped scrambler to be $m_{MAX} + n_{MAX}$ to cover the worst case since we want to have static schedulability.

Another choice is modeling only for a fixed r_m and r_n. The first choice gives us a flexible system while the other choice gives us a dedicate system. Supporting dynamic schedule is a future work. Once its supported, we can combine the two choices. The SDF actor of the wrapped scrambler, which is denoted as a_3, is depicted in Figure 9.8. Remember, we also need to make the FIMP for it before we can model the entire system.

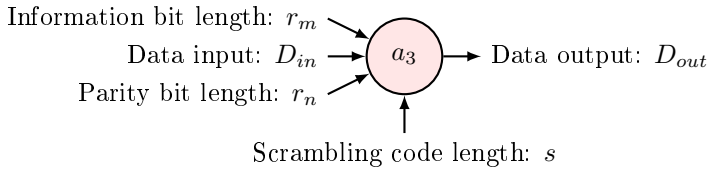


Figure 9.8: Wrapped Scrambler SDF Actor

Constellation Mapper

The constellation mapper is for matching the transmitted data to the characteristics of the communication channel [122]. The concept is to map a group of data bits to a symbol that will be transmitted via the communication channel. Each symbol is represented by a point in the complex plain in the time domain. Examples of the mapping methods are as follows: Binary Phase-Shift Keying (BPSK) that converts 1 data bit to one of the 2 points in the complex plain, Quadrature Phase-Shift Keying (QPSK) that converts 2 data bit to one of the 4 points in the complex plain, 16-Quadrature Amplitude Modulation (16 QAM) that converts 4 data bit to one of the 16 points in the complex plain, and 64 Quadrature Amplitude Modulation (64-QAM) that converts 6 data bit to one of the 64 points in the complex plain. 128 Quadrature Amplitude Modulation (128-QAM) that converts 7 data bit to one of the 128 points in the complex plain, and 256 Quadrature Amplitude Modulation (256-QAM) that converts 8 data bit to one of the 256 points in the complex plain.

One method with more distinct points in the complex plain can map more data bits in one symbol that typically will most likely increase the transmission bandwidth.

Similar to the other previous functions, the constellation mapper is for the general cases that the mapping method can be changed at run time. The input to the scrambler consists of the data input D_{in} and the mapping method m. The data token type of D_{in} is *std_logic_vector(D_{MAX} - 1 downto 0)*, where D_{MAX} is the maximum number of bits that can be mapped onto one symbol. The data token type of m is *integer range 0 to 5* indicating the modulation methods of BPSK, QPSK, 16-QAM, 64-QAM, 128-QAM, and 256-QAM, respectively. The data output from the constellation mapper consists of the real part $S_{out,re}$ and the imaginary part $S_{out,im}$ of the output symbol S_{out}. Assume each of the component of in one complex number represented by 16 bits. The data token types of both outputs are *std_logic_vector (15 downto 0)*. The SDF actor of the constellation mapper, which is denoted as a_4, is depicted in Figure 9.9.

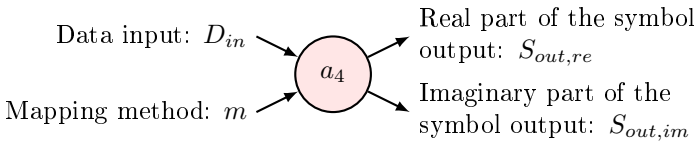


Figure 9.9: Constellation Mapper SDF Actor

Transform Precoder

The transform precoder performs Fast Fourier Transform (FFT) on the signals that converts a symbol stream from time domain to frequency domain for the further procedures.

The transform precoder is an 64-point FFT. The input to the transform precoder consists of the real part of the symbol input $S_{in,re}$ and the imaginary part of the symbol input $S_{in,im}$. The output from the transform precoder consists of the real part of the symbol output $S_{out,re}$ and the imaginary part of the symbol output $S_{out,im}$. Assume we use the FFT FIMPs with paralleled input ports and each of the component of in one complex number represented by 16 bits. Therefore, the data token types of all the inputs and outputs are all *std_logic_vector (16 * 64 - 1 downto 0)*. The data token counts of all the inputs and outputs are 1's. The SDF actor of the transform precoder, which is denoted as a_5, is depicted in Figure 9.10.

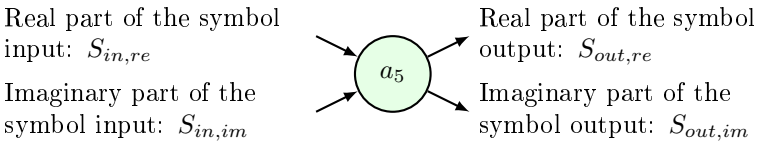


Figure 9.10: Transform Precoder SDF Actor

Sub-Carrier Mapper

The sub-carrier mapper inserts data symbol and reference symbols into the sub-carriers. Signal data from multiple users will be mapped into non-overlapping sub-carriers. Pilot signals are also mapped into sub-carriers is needed for reconstructing the channel state information using a channel estimator at the receiver.

The sub-carrier mapper is dedicated for LTE that the input to it consists of the real part of the symbol input $S_{in,re}$ and the imaginary part of the symbol input $S_{in,im}$. The output from the sub-carrier mapper consists of the real part of the symbol output $S_{out,re}$ and the imaginary part of the symbol output $S_{out,im}$. Assume the sub-carrier mapper supports up to 16 users including the pilot symbols. Also assume that its FIMP has only serialized input ports and each of the component of in one complex number represented by 16 bits. Therefore, the data token types of all the inputs are all *std_logic_vector (16 * 16 - 1 downto 0)* and the data token counts of all the inputs are 64. The data token types of all the outputs are all *std_logic_vector (16 * 64 - 1 downto 0)* and the data token counts of all the inputs are 1's. The SDF actor of the sub-carrier mapper, which is denoted as a_6, is depicted in Figure 9.11.

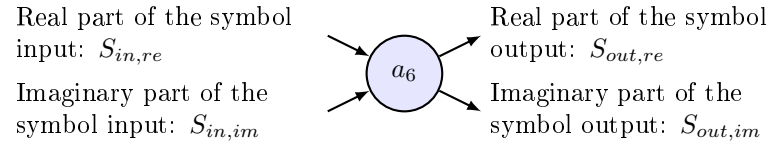


Figure 9.11: Sub-Carrier Mapper SDF Actor

SC-FDMA Modulation

The Single Carrier Frequency Diversity Multiple Access (SC-FDMA) modulation implements a precoded Orthogonal Frequency Diversity Multiplexing (OFDM) scheme. The the precoding is done by the transform precoder and the OFDM is done by performing IFFT and inserting a cyclic prefix, which will be eliminated at the receiver.

The SC-FDMA modulation is implemented as a 64-point IFFT and the input to it consists of the real part of the symbol input $S_{in,re}$ and the imaginary part of the symbol input $S_{in,im}$. The output from the sub-carrier mapper consists of the real part of the symbol output $S_{out,re}$ and the imaginary part of the symbol output $S_{out,im}$. Assume the sub-carrier mapper supports up to 16 users including the pilot symbols. Similar to the transform precoder, we assume that its FIMP has only paralleled input ports and each of the component of in one complex number represented by 16 bits. Therefore, the data token types of all the inputs and outputs are all *std_logic_vector (16 * 64 - 1 downto 0)*. The data token counts of all the inputs and outputs are 1's. The SDF actor of the SC-FDMA modulation, which is denoted as a_7, is depicted in Figure 9.12.

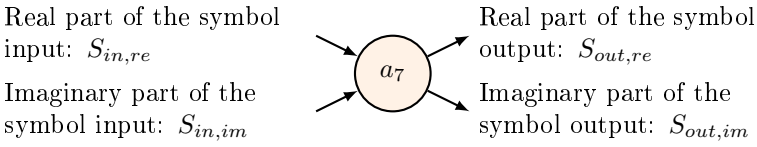


Figure 9.12: SC-FDMA Modulation SDF Actor

Bibliography

[1] Semiconductor manufacturing technology, sematech. [Online]. Available: http://public.sematech.org/

[2] H. D. Foster, "Why the design productivity gap never happened," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 581–584.

[3] International technology roadmap for semiconductors, itrs. [Online]. Available: http://www.itrs.net/

[4] T. C. Meyerowitz, *Single and Multi-CPU Performance Modeling for Embedded Systems.* ProQuest, 2008.

[5] D. D. Gajski and R. H. Kuhn, "Guest editors' introduction: New vlsi tools," *Computer*, vol. 16, no. 12, pp. 11–14, 1983.

[6] J. Williams, "STICKS - a graphical compiler for high level LSI design," in *International Workshop on Managing Requirements Knowledge*, 1978.

[7] C. Mead and L. Conway, *Introduction to VLSI systems.* Addison-Wesley Reading, 1980.

[8] T. Kozawa, H. Horino, K. Watanabe, M. Nagata, and H. Hukuda, "Block and track method for automated layout generation of MOS-LSI arrays," in *Digest of Technical Papers in International Solid-State Circuits Conference*, vol. XV, 1972, pp. 62–63.

[9] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design Test of Computers*, vol. 26, no. 4, pp. 18 –25, 2009.

[10] C. U. Smith, G. A. Frank, and J. Cuardrado, "An architecture design and assessment system for software/hardware codesign," in *Conference on Design Automation*, 1985, pp. 417–424.

[11] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[12] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *Compiler Construction*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2304, pp. 179–196.

[13] C. Lee, S. Kim, and S. Ha, "A systematic design space exploration of mpsoc based on synchronous data flow specification," *Journal of Signal Processing Systems*, vol. 58, no. 2, pp. 193–213, 2010.

[14] M. A. Shami, "Dynamically reconfigurable resource array," Ph.D. dissertation, KTH, 2012.

[15] V. Tehre and R. Kshirsagar, "Survey on coarse grained reconfigurable architectures," *International Journal of Computer Applications*, vol. 48, no. 16, pp. 1–7, 2012.

[16] N. Farahini, S. Li, M. A. Tajammul, M. A. Shami, G. Chen, A. Hemani, and W. Ye, "39.9 gops/watt multi-mode cgra accelerator for a multi-standard basestation," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2013, pp. 1448–1451.

[17] K. R. Apt, *Principles of Constraint Programming.* Cambridge University Press, 2003.

[18] E. Tsang, *Foundations of Constraint Satisfaction: The Classic Text.* BoD–Books on Demand, 2014.

[19] P.-E. Hladik, H. Cambazard, A.-M. Déplanche, and N. Jussien, "Solving a real-time allocation problem with constraint programming," *Journal of Systems and Software*, vol. 81, no. 1, pp. 132–149, 2008.

[20] P. Van Beek and X. Chen, "Cplan: A constraint programming approach to planning," in *AAAI-99 Proceedings.*, 1999, pp. 585–590.

[21] P. Baptiste, C. Le Pape, and W. Nuijten, *Constraint-based scheduling: applying constraint programming to scheduling problems.* Springer, 2001, vol. 39.

[22] F. Rossi, P. V. Beek, and T. Walsh, *Handbook of Constraint Programming.* Elsevier, 2006.

[23] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium*, ser. Lecture Notes in Computer Science, B. Robinet, Ed. Springer Berlin Heidelberg, 1974, vol. 19, pp. 362–376.

[24] K. M. Kavi, B. P. Buckles, and U. N. Bhat, "A formal definition of data flow graph models," *IEEE Transactions on Computers*, vol. C-35, no. 11, pp. 940–948, 1986.

[25] A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, A. Moonen, M. Bekooij, B. Theelen, and M. Mousavi, "Throughput analysis of synchronous data flow graphs," in *Sixth International Conference on Application of Concurrency to System Design*, 2006, pp. 25–36.

[26] S. Stuijk, M. Geilen, and T. Basten, "Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs," *IEEE Transactions on Computers*, vol. 57, no. 10, pp. 1331–1345, 2008.

[27] F. Siyoum, M. Geilen, O. Moreira, R. Nas, and H. Corporaal, "Analyzing synchronous dataflow scenarios for dynamic software-defined radio applications," in *International Symposium on System on Chip*, 2011, pp. 14–21.

[28] J. Buck, "Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams," in *Conference Record of the Twenty-Eighth Asilomar Conference on Signals, Systems and Computers*, vol. 1, 1994, pp. 508–513.

[29] M. A. Tajammul, M. A. Shami, and A. Hemani, "Segmented bus based path setup scheme for a distributed memory architecture," in *International Symposium on Embedded Multicore SoCs*. IEEE, 2012, pp. 67–74.

[30] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach (3rd Edition)*. Prentice Hall, 2009.

[31] F. S. de Boer, A. D. Pierro, and C. Palamidessi, "Nondeterminism and infinite computations in constraint programming," *Theoretical Computer Science*, vol. 151, no. 1, pp. 37–78, 1995.

[32] J. Cohen, "Constraint logic programming languages," *Communications of the ACM*, vol. 33, no. 7, pp. 52–68, 1990.

[33] S. Kumar, M. K. Luhandjula, E. Munapo, and B. C. Jones, "Fifty years of integer programming: A review of the solution approaches," *Asia Pacific Business Review*, vol. 6, no. 3, pp. 5–15, 2010.

[34] I. P. Gent, I. Miguel, and N. C. Moore, "Lazy explanations for constraint propagators," in *Practical Aspects of Declarative Languages*. Springer, 2010, pp. 217–233.

[35] Global constraint catalog. [Online]. Available: http://sofdem.github.io/gccat/

[36] M. C. McFarland, A. C. Parker, and R. Camposano, "The high-level synthesis of digital systems," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 301–318, 1990.

[37] D. D. Gajski and L. Ramachandran, "Introduction to high-level synthesis," *IEEE Design and Test*, vol. 11, no. 4, pp. 44–54, 1994.

[38] B. Bailey, M. McNamara, F. Balarin, M. Stellfox, G. Mosenson, and Y. Watanabe, *TLM Driven Design and Verification Methodology.* Cadence Design Systems, 2010.

[39] Catapult from Calypto. [Online]. Available: http://calypto.com/en/products/catapult/overview/

[40] Vivado design suite from Xilinx. [Online]. Available: http://china.xilinx.com/products/design-tools/vivado.html

[41] AutoPilot from AutoESL. [Online]. Available: http://www.bdti.com/MyBDTI/pubs/AutoPilot.pdf

[42] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *International Symposium on Field Programmable Gate Arrays*, 2011.

[43] Bluespec high-level synthesis toolset. [Online]. Available: http://www.bluespec.com/high-level-synthesis-tools.html

[44] H.-Y. Liu, M. Petracca, and L. P. Carloni, "Compositional system-level design exploration with planning of high-level synthesis," in *Conference and Exhibition on Design Automation & Test in Europe.* IEEE, 2012, pp. 641–646.

[45] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *ACM SIGOPS Operating Systems Review*, 2006.

[46] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to software: RAW machines," *Computer*, vol. 30, no. 9, pp. 86–93, 1997.

[47] H. Nikolov, T. Stefanov, and E. Deprettere, "Multi-processor system design with ESPAM," in *International Conference on Hardware/Software Codesign and System Synthesis*, 2006, pp. 211 –216.

[48] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprette, "System design using khan process networks: the compaan/laura approach," in *Conference and Exhibition on Design Automation & Test in Europe*, vol. 1, 2004, pp. 340–345.

[49] H. Nikolov, T. Stefanov, and E. Deprettere, "Systematic and automated multiprocessor system design, programming, and implementation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 3, pp. 542–555, 2008.

[50] E. F. Deprettere, T. Stefanov, S. S. Bhattacharyya, and M. Sen, "Affine nested loop programs and their binary parameterized dataflow graph counterparts," in *International Conference on Application-specific Systems, Architectures and Processors*, 2006, pp. 186–190.

[51] S. Verdoolaege, H. Nikolov, and T. Stefanov, "PN: a tool for improved derivation of process networks," *EURASIP journal on Embedded Systems*, vol. 2007, no. 1, pp. 19–19, 2007.

[52] J. Gladigau, A. Gerstlauer, C. Haubelt, M. Streubuhr, and J. Teich, "A system-level synthesis approach from formal application models to generic bus-based mpsocs," in *International Conference on Embedded Computer Systems (SAMOS)*, 2010, pp. 118 –125.

[53] K. Kuchcinski, "Constraints-driven scheduling and resource assignment," *ACM Transactions on Design Automation of Electronic Systems*, vol. 8, no. 3, pp. 355–383, 2003.

[54] A. Bonfietti, L. Benini, M. Lombardi, and M. Milano, "An efficient and complete approach for throughput-maximal sdf allocation and scheduling on multi-core platforms," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2010.

[55] C. von Platen, J. Eker, A. Nilsson, and K.-E. Årzén, "Static analysis and transformation of dataflow multimedia applications," *Technical Report from Lund University, Sweden*, 2012.

[56] K. Kuchcinski and R. Szymanek, "Jacop-java constraint programming solver," in *CP Solvers: Modeling, Applications, Integration, and Standardization, co-located with the 19th International Conference on Principles and Practice of Constraint Programming*, 2013.

[57] M. Arslan, F. Gruian, and K. Kuchcinski, "Application-set driven exploration for custom processor architectures," in *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, 2015, pp. 70–71.

[58] U. M. Mirza, F. Gruian, and K. Kuchcinski, "Design space exploration for streaming applications on multiprocessors with guaranteed service noc," in *Proceedings of the Sixth International Workshop on Network on Chip Architectures*, 2013, pp. 35–40.

[59] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "Dresc: A retargetable compiler for coarse-grained reconfigurable architectures," in *Proceedings of 2002 IEEE International Conference on Field-Programmable Technology (FPT)*, 2002, pp. 166–173.

[60] C. Liang, X. Huang, and M. Rupp, "Smartcell: An energy efficient coarse-grained reconfigurable architecture for stream-based applications," *EURASIP journal on embedded systems*, vol. 2009, p. 27, 2009.

[61] G. Ansaloni, P. Bonzini, and L. Pozzi, "EGRA: A coarse grained reconfigurable architectural template," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 19, no. 6, pp. 1062–1074, 2011.

[62] P. M. Heysters, G. J. Smit, and E. Molenkamp, "Montium - balancing between energy-efficiency, flexibility and performance," in *Engineering of Reconfigurable Systems and Algorithms*, T. P. Plaks, Ed. CSREA Press, 2003, pp. 235–241. [Online]. Available: http://doc.utwente.nl/46380/

[63] N. Vassiliadis, G. Theodoridis, and S. Nikolaidis, "An automated development framework for a RISC processor with reconfigurable instruction set extensions," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International.* IEEE, 2006, pp. 4–pp.

[64] M. Alle, K. Varadarajan, A. Fell, S. Nandy, and R. Narayan, "Compiling techniques for coarse grained runtime reconfigurable architectures," in *Reconfigurable Computing: Architectures, Tools and Applications.* Springer, 2009, pp. 204–215.

[65] Altera nios ii c2h compiler user guide. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_nios2_c2h_compiler.pdf

[66] J. M. Cardoso and M. Weinhardt, "XPP-VC: AC compiler with temporal partitioning for the PACT-XPP architecture," in *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream.* Springer, 2002, pp. 864–874.

[67] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 225–235, 2000.

[68] S. Masekowsky, T. Schweizer, W. Rosenstiel *et al.*, "Cgadl: an architecture description language for coarse-grained reconfigurable arrays," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 17, no. 9, pp. 1247–1259, 2009.

[69] D. C. Cronquist, P. Franklin, S. G. Berg, and C. Ebeling, "Specifying and compiling applications for rapid," in *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on.* IEEE, 1998, pp. 116–125.

[70] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, "Spr: an architecture-adaptive cgra mapping tool," in *Proceedings*

of the ACM/SIGDA international symposium on Field programmable gate arrays. ACM, 2009, pp. 191–200.

[71] J.-e. Lee, K. Choi, and N. D. Dutt, "Compilation approach for coarse-grained reconfigurable architectures," *IEEE Design & Test of Computers*, vol. 20, no. 1, pp. 26–33, 2003.

[72] R. Jevtic and C. Carreras, "Power estimation of embedded multiplier blocks in fpgas," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 18, no. 5, pp. 835–839, 2010.

[73] O. Malik, A. Hemani, and M. A. Shami, "A library development framework for a coarse grain reconfigurable architecture," in *VLSI Design (VLSI Design), 2011 24th International Conference on*. IEEE, 2011, pp. 153–158.

[74] J. J. Rodriguez-Andina, M. J. Moure, and M. D. Valdes, "Features, design tools, and application domains of FPGAs," *IEEE Transactions on Industrial Electronics*, vol. 54, no. 4, pp. 1810–1823, 2007.

[75] A. Wood, A. Knight, B. Ylvisaker, and S. Hauck, "Multi-kernel floorplanning for enhanced CGRAs," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. IEEE, 2012, pp. 157–164.

[76] J. M. Emmert and D. Bhatia, "A methodology for fast fpga floorplanning," in *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*. ACM, 1999, pp. 47–56.

[77] L. Cheng and M. D. Wong, "Floorplan design for multimillion gate FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 12, pp. 2795–2805, 2006.

[78] S. Li, N. Farahini, A. Hemani, K. Rosvall, and I. Sander, "System level synthesis of hardware for dsp applications using pre-characterized function implementations," in *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on*. IEEE, 2013, pp. 1–10.

[79] S. Murali and G. D. Micheli, "Bandwidth-constrained mapping of cores onto noc architectures," in *Conference and Exhibition on Design Automation & Test in Europe*. IEEE Computer Society, 2004, p. 20896.

[80] J. Hu and R. Marculescu, "Energy-aware mapping for tile-based noc architectures under performance constraints," in *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*. ACM, 2003, pp. 233–239.

[81] T. Lei and S. Kumar, "A two-step genetic algorithm for mapping task graphs to a network on chip architecture," in *Digital System Design, 2003. Proceedings. Euromicro Symposium on*. IEEE, 2003, pp. 180–187.

[82] W. Lei and L. Xiang, "Energy-and latency-aware noc mapping based on chaos discrete particle swarm optimization," in *Communications and Mobile Computing (CMC), 2010 International Conference on*, vol. 1. IEEE, 2010, pp. 263–268.

[83] A. Mehran, A. Khademzadeh, and S. Saeidi, "Dsm: A heuristic dynamic spiral mapping algorithm for network on chip," *IEICE Electronics Express*, vol. 5, no. 13, pp. 464–471, 2008.

[84] R. Pop and S. Kumar, "A survey of techniques for mapping and scheduling applications to network on chip systems," *School of Engineering, Jonkoping University, Research Report*, vol. 4, p. 4, 2004.

[85] G. Chen, F. Li, S. W. Son, and M. Kandemir, "Application mapping for chip multiprocessors," in *Design Automation Conference*. IEEE, 2008, pp. 620–625.

[86] A. Naeem, X. Chen, Z. Lu, and A. Jantsch, "Scalability of weak consistency in NoC based multicore architectures," in *International Symposium on Circuits and Systems*. IEEE, 2010, pp. 3497–3500.

[87] S. Li, F. Jafari, A. Hemani, and S. Kumar, "Layered spiral algorithm for memory-aware mapping and scheduling on network-on-chip," in *NORCHIP, 2010*. IEEE, 2010, pp. 1–6.

[88] A. Mehran, S. Saeidi, A. Khademzadeh, and A. Afzali-Kusha, "Spiral: A heuristic mapping algorithm for network on chip," *IEICE Electronics Express*, vol. 4, no. 15, pp. 478–484, 2007.

[89] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*. ACM, 1971, pp. 151–158.

[90] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "PipeRench: A reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70 –77, 2000.

[91] C-to-silicon compiler from Cadence. [Online]. Available: http://www.cadence.com/products/sd/silicon_compiler/

[92] N. Rishiyur, "Bluespec systemverilog: Efficient, correct RTL from high level specifications," in *The Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, 2004, pp. 69–70.

[93] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From opencl to high-performance hardware on fpgas," in *the 22nd International Conference on Field Programmable Logic and Applications (FPL)*, 2012, pp. 531–534.

[94] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.-M. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *Symposium on Application Specific Processors*, 2009, pp. 35–42.

[95] G. Roquier, M. Wipliez, M. Raulet, J. W. Janneck, I. D. Miller, and D. B. Parlour, "Automatic software synthesis of dataflow program: An MPEG-4 simple profile decoder case study," in *Workshop on Signal Processing Systems*. IEEE, 2008, pp. 281–286.

[96] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, "A stream compiler for communication-exposed architectures," in *ACM SIGPLAN Notices*, vol. 37, no. 10. ACM, 2002, pp. 291–303.

[97] A. Olugbon, T. Arslan, S. MacDougall *et al.*, "Providing compilers and application program support for reconfigurable socs: Radical but overdue," in *System-on-Chip, 2005. Proceedings. 2005 International Symposium on*. IEEE, 2005, pp. 54–57.

[98] B. Svensson *et al.*, "Occam-pi as a high-level language for coarse-grained reconfigurable architectures," in *International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW)*. IEEE, 2011, pp. 236–243.

[99] H. Giefers and M. Platzner, "ARMLang: a language and compiler for programming reconfigurable mesh many-cores," in *IEEE International Symposium on Parallel & Distributed Processing*, 2009, pp. 1–8.

[100] F. Qi, X. Zhang, S. Wang, and X. Mao, "RCC: A new programming language for reconfigurable computing," in *International Conference on High Performance Computing and Communications*. IEEE, 2009, pp. 688–693.

[101] S. Bhartacharyya, R. Leupers, and P. Marwedel, "Software synthesis and code generation for signal processing systems," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, vol. 47, no. 9, pp. 849–875, 2000.

[102] Fixed point package user's guide. [Online]. Available: http://www.eda.org/fphdl/Fixed_ug.pdf

[103] Operations research tools from google. [Online]. Available: https://developers.google.com/optimization/

[104] Minizinc challenge. [Online]. Available: http://www.minizinc.org/challenge.html

[105] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller, "Energy management for commercial servers," *Computer*, vol. 36, no. 12, pp. 39–48, 2003.

[106] E. S. Chung, J. C. Hoe, and K. Mai, "CoRAM: An in-fabric memory architecture for fpga-based computing," in *International symposium on field programmable gate arrays*. ACM, 2011, pp. 97–106.

[107] R. Jejurikar and R. Gupta, "Dynamic voltage scaling for systemwide energy minimization in real-time embedded systems," in *Proceedings of the 2004 International Symposium on Low Power Electronics and Design, ISLPED'04*. IEEE, 2004, pp. 78–81.

[108] A. T. Abdel-Hamid, M. Zaki, and S. Tahar, "A tool converting finite state machine to vhdl," in *Canadian Conference on Electrical and Computer Engineering*, vol. 4. IEEE, 2004, pp. 1907–1910.

[109] M. A. Shami and A. Hemani, "An improved self-reconfigurable interconnection scheme for a coarse grain reconfigurable architecture," in *NORCHIP Conference*, 2010, pp. 1–6.

[110] J. Liu, L.-R. Zheng, and H. Tenhunen, "Interconnect intellectual property for network-on-chip (noc)," *Journal of Systems architecture*, vol. 50, no. 2, pp. 65–79, 2004.

[111] A. M. Shafiee, M. Montazeri, and M. Nikdast, "An innovational intermittent algorithm in networks-on-chip (NoC)," *World Academy of Science, Engineering and Technology*, vol. 45, pp. 145–147, 2008.

[112] R. P. Dick, D. L. Rhodes, and W. Wolf, "Tgff: task graphs for free," in *Proceedings of the 6th international workshop on Hardware/software codesign*. IEEE Computer Society, 1998, pp. 97–101.

[113] G. K. Wallace, "The JPEG still picture compression standard," *Communications of the ACM*, vol. 34, no. 4, pp. 30–44, 1991.

[114] B. G. Haskell, *Digital Video: An Introduction to MPEG-2: An Introduction to MPEG-2*. Springer Science & Business Media, 1997.

[115] Opencores. [Online]. Available: http://opencores.org/

[116] 3gpp ts 25.213 spreading and modulation (fdd). [Online]. Available: http://www.etsi.org/deliver/etsi_ts/125200_125299/125213/12.00.00_60/ts_125213v120000p.pdf

[117] R. Schreier, G. C. Temes *et al.*, *Understanding Delta-Sigma Data Converters*. IEEE press Piscataway, NJ, 2005, vol. 74.

[118] S. Lee, J.-M. Kim, and S.-I. Chae, "New motion estimation algorithm using adaptively quantized low bit-resolution image and its vlsi architecture for mpeg2 video encoding," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 8, no. 6, pp. 734–744, 1998.

[119] S. Stuijk, M. Geilen, B. Theelen, and T. Basten, "Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications," in *International Conference on Embedded Computer Systems*. IEEE, 2011, pp. 404–411.

[120] F. Glover, "Tabu search: A tutorial," *Interfaces*, vol. 20, no. 4, pp. 74–94, 1990.

[121] C. Berrou and A. Glavieux, "Near optimum error correcting coding and decoding: Turbo-codes," *IEEE Transactions on Communications*, vol. 44, no. 10, pp. 1261–1271, 1996.

[122] Y. Li and X.-G. Xia, "Constellation mapping for space-time matrix modulation with iterative demodulation/decoding," *IEEE Transactions on Communications*, vol. 53, no. 5, pp. 764–768, 2005.