Clustering synchronous dataflow actors for efficient usage of configurable hardware

Sinha, S.S.

Award date:
2011

Master's Thesis

Clustering Synchronous Dataflow Actors for Efficient Usage of Configurable Hardware

Shubhendu Sinha
September 2011

Supervisors
prof. dr. Henk Corporaal (Eindhoven University of Technology)
prof. dr. ir. Marco Bekooij (NXP Semiconductors, University of Twente)
ir. Pjotr Kourzanov (NXP Semiconductors)

Eindhoven University of Technology
NXP Semiconductors

Abstract

In recent years, integration of digital wireless communication and embedded systems has facilitated a wide range of wireless services such as cellular, digital audio broadcasting, digital video broadcasting etc. These services can be represented under a common framework known as Software Defined Radio (SDR). SDR applications are firm real time streaming applications. For fast channel decoding of SDR streams FLexible Outer Receiver Architecture (FLORA) hardware accelerator has been designed as part of a heterogeneous MPSoC designed at NXP. FLORA consists of diverse hardware blocks for decoding common kernels used in SDR applications. FLORA enables high performance with hardware implementation but it is also configurable to support parametric kernels and multiple standards. FLORA has DMA blocks to transfer data in and out of FLORA. Since DMA blocks can be configured to serve any hardware block, there are finitely many possible ways to group tasks mapped to hardware blocks in FLORA as a FLORA configuration. FLORA is configured by external entity such as ARM processor which incurs a scheduling load. An analytical method is required to exploit the flexibility offered by FLORA and also to guarantee the throughput requirement of each SDR application ported to FLORA. This problem is addressed in this thesis.

Given a set of multiple independent applications represented in Synchronous Data Flow (SDF) model of computation we obtain optimal configurations for FLORA. An optimal configuration satisfies throughput and buffer constraints and offers minimum scheduling load. We model a FLORA configuration as a clustered atomic SDF actor which is obtained by clustering sub-graph(s) in the original SDF graph. We present a clustering function which given an input SDF graph and a set of SDF actors to be clustered, produces a clustered SDF graph. We provide a condition to check for deadlock-free clusters. We prove our clustering function results in consistent and deadlock-free clustered graphs. Using the clustering function we present an exact and a heuristic clustering algorithm to find the optimal clusterings in given set of input SDF graphs. Experiments show for SDR applications heuristic algorithm performs as good as exact algorithm in finding optimal configurations for FLORA.

Acknowledgement

This thesis summarizes the work of ten months graduation project carried out in NXP Semiconductors. The graduation would not have been possible without the help and support of several people. I would like to express my gratitude to them.

First and foremost I would like to express my gratitude to prof. Marco Bekooij, my supervisor in NXP. He made it possible for me to define the goal of the thesis and achieve them within the limitations of available time and my capacity. I owe my learning curve in understanding literature to him. He spent his valuable time in guiding and reviewing my work. He helped me to make practical decisions. His approach towards research motivated me to enjoy my work.

Secondly I must thank ir. Pjotr Kourzanov, my daily supervisor in NXP. He spent large amount of time to help solve my problems. With his exceptional talent in computer science he helped me with practical problems such as making algorithms and programming. He was a very useful critic and helped me understand the advantages and shortcomings of every step.

I must thank my mentor in TU/e, prof. Henk Corporaal who during his courses, inspired me for a career in computer architecture in the first place. He facilitated and arranged this thesis for me in NXP. During the course of thesis he provided me with valuable feedback and support.

I must also mention my gratitude to Derik and Arthur of the Modem & Signal Processing research group in NXP who helped me by providing useful information required for the project. Finally I must thank my family and friends for their unconditional love and support.

Contents

1 Introduction **1**
- 1.1 Baseband Processing in Software Defined Radios 2
- 1.2 FLORA Architecture 5
- 1.3 Problem Description 6
- 1.4 Contributions 12
- 1.5 Overview 12

2 Dataflow Preliminaries **17**
- 2.1 Synchronous Data Flow Graphs 17
- 2.2 Homogeneous Synchronous Dataflow Graph 18
- 2.3 Throughput Constraint for SDR applications 20

3 Problem Formulation **22**
- 3.1 Goal 22
- 3.2 Challenges 22
- 3.3 Assumptions 24
- 3.4 Formal Problem Statement 25
- 3.5 Related Work 26

4 Clustering Definition **30**
- 4.1 Clustering 30
- 4.2 Clustering Function (Φ_c) for a single sub-graph 32
- 4.3 Clustering Function (Φ_g) for sub-graphs 38
- 4.4 Deadlock - Free Clusters 40
- 4.5 Properties of Clustering Function 42
  - 4.5.1 Preservation of Consistency 42
  - 4.5.2 Deadlock Freedom 43
- 4.6 Effects of Clustering 44

5 Clustering Algorithms **50**
- 5.1 Algorithms For Checking Deadlock-free Clusters 50
- 5.2 Clustering Algorithms 53
  - 5.2.1 Exhaustive Search Approach 53
  - 5.2.2 Heuristic Approach 57
- 5.3 Implementation of Clustering Algorithm 59
- 5.4 Run-Time Evaluation of Algorithms 60

6 Conclusion **65**
6.1 Conclusion 65
6.2 Future Work 66
6.2.1 General Application of Clustering for SDF 67

Appendices **72**

A MARS MPSoC Platform **72**

B FLORA **74**

C List of Symbols **78**

Glossary **79**

Bibliography **81**

Chapter 1

Introduction

In the past century digitalization revolutionized almost all facets of human life. One of the recent revolutions empowered by digitalization was the revolution in wireless technology. Digital Wireless Communication has evolved into a powerful, robust and useful day-today phenomena. From the huge wooden box of 'radio receiver' in 1960s which allowed consumers to listen to only songs and news to the relatively tiny 'smartphones' in 2000s which allows calling, viewing/listening wireless video/audio, playing games, emailing, web-surfing, online shopping, etc. among other features. Indeed wireless technology is now used for day to-day purposes such as cellphone communication, wireless internet, mobile TV, wireless radio, etc.

Wireless technology was initially an analogue phenomena with weaker capabilities. Modern digital wireless technology owes its power to a number of innovations and significant research in subjects like information theory, signal modulation-demodulation, coding theory, digital signal processing, software architecture, etc. [7] to name a few. The electronic systems consisting of hardware and software which enables these functionalities come under the name of *Embedded Systems*. Embedded Systems are defined as microprocessor based systems that are built for certain specific function or range of specific functions [8].

Recently this complex integration of wireless communication and embedded systems was more formally and eloquently defined under the term *SDR*. SDR is defined in [22] as 'Radio in which some or all of the physical layer functions are Software Defined'. The term 'Software Defined' refers to the fact that a processor based software system is used to implement and/or control the available functionalities [22].

Smaller, faster and powerful. This thirst shaped the evolution of embedded systems from a single processor based to a Multi-Processor System on Chip (MPSoC) based. And indeed like many other consumer services/products SDR is developed on a MPSoC platform.

The physical layer of SDR consists of multiple stages (and as mentioned in [22], some or all of these are software defined). SDR applications have stringent throughput and/or latency requirements. Each stage in the physical layer can be optimized for the performance requirement of the particular SDR application. Some of the popular SDR applications can be broadly categorized as *Cellular*, *Data Communication*, *Digital Radio*, *Digital Video*. Therefore for each stage in the physical layer of SDR different algorithms exist for different categories of SDR applications. For proven robustness, inter-operability, re-usability and

to enable efficient commercial deployment, these algorithms or methods are standardized by the respective standards organization as a Wireless Standard or rather family of standards which keep on evolving with faster, robust and powerful services. Standards such as

- **Cellular**- Global System for Mobile Communications (GSM), Long Term Evolution (3GPP 4G technology) (LTE)
- **Data Communication** - Wireless Fidelity (IEEE 802.11) (WiFi), Bluetooth
- **Digital Radio Broadcasting** - Satellite radio (XM), Digital Audio Broadcasting (DAB)
- **Digital Video Broadcasting** - Digital Video Broadcasting (DVB) family, Integrated Services Digital Broadcasting (ISDB), Advanced Television Systems Committee (ATSC)

SDR are streaming multimedia applications with firm performance constraints. For cost effectiveness it is desired to have a single platform supporting as many standards as possible. At the same time the performance constraint of each standard must be satisfied. Flexibility and performance are contradictory goals. To achieve a desired balance between flexibility and required computational power for high performance a heterogeneous MPSoC called MARS has been developed at NXP. The heterogeneous MPSoC contains a configurable hardware accelerator called as FLORA. It consists of diverse hardware blocks meant for fast decoding of SDR standards. Since decoding is in hardware it enables higher performance. But at the same time FLORA is configurable so as to support multiple SDR standards.

In order to guarantee performance requirements analytical methods are preferred [1,3,18]. For our problem domain of SDR and FLORA hardware accelerator an analytical method is required which must be able to exploit the flexibility offered by FLORA hardware accelerator to support multiple SDR applications and at the same time guarantee performance requirements of each standard. We present such an analytical method in this thesis.

In the following section we describe the available degree of flexibility in the physical layer of SDR. In section 1.2 we describe the architecture of FLORA hardware accelerator. We summarize the background of thesis in section 1.3. At the end of chapter in section 1.4 we present the contributions of this thesis.

1.1 Baseband Processing in Software Defined Radios

Physical layer of SDR consists of two parts, Analog Front End (AFE) and digital baseband processing. We briefly describe in this section different stages in digital baseband processing and the flexibility available at each stage. The

physical layer of SDR consists of broadly three stages as shown in figure 1.1. The left side of the dotted box consisting of ADC(Analog to Digital Converter), DAC(Digital to Analog Converter) and RF-tuner form the AFE. The right side of the dotted box goes to the Media Access Control (MAC) layer. The dotted box forms digital baseband processing. We are concerned with this stage of the physical layer where most of the computational load lies. Digital baseband processing consists of following three parts.

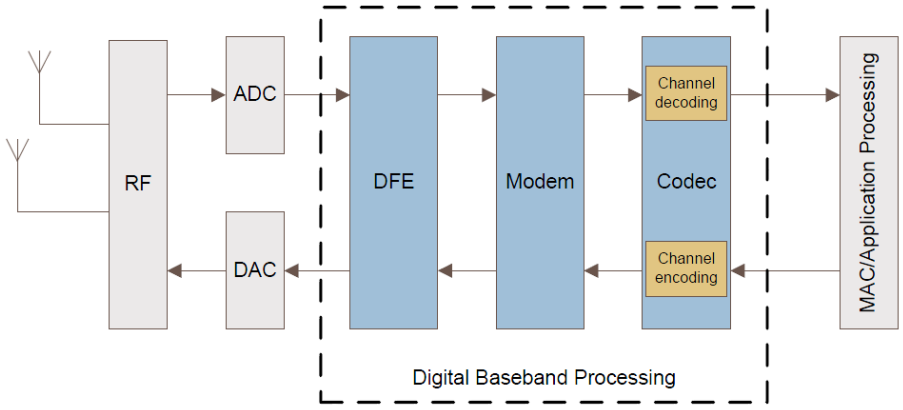


Figure 1.1: *Placement of Digital Baseband Processing*

- **Digital Front End (DFE)** DFE chiefly consists of various filters to achieve three essential functions [9]
  - IQ transposition. Convert the digitalized real signal to complex signal and vice versa.
  - Sample Rate Conversion. Convert the stream rate to the rate that fits the standard.
  - Channel Selection. Select the proper channel. It includes conversion to baseband and channel filtering.

  These functions have high computational load but the algorithms for these functions are similar in different standards. Therefore DFE is normally implemented on a configurable hardware, but not mapped to a programmable processor.

- **Modem** This stage is also called as 'Inner Transceiver'. It performs several functions such as modulation, demodulation, channel equalization, channel estimation, mapping, de-mapping and so on. This stage is usually implemented on a Digital Signal Processor (DSP). This is so because of following reasons.
  - The standards are highly diverse and algorithms are complex.

  - The functions in modem involve intense computational load, such as FFT, correlation. These functions can be efficiently implemented on a application specific processor such as DSP.
  - The standard leaves the freedom to manufacturers to design their own algorithms for better performance.

- **Codec** The codec stage is also called as 'Outer transceiver'. This stage is responsible for channel coding or decoding of a SDR. The decoding stage can correct data if they are in errors, hence it is also called as Forward Error Correction (FEC). It involves various mathematical functions described in table 1.1. These mathematical functions are limited in number. But each standard differs in the choice of functions used, the configuration parameters such as equation or size of input/output and the order in which they are used. Therefore for high-performance these functions are implemented on hardware accelerator with a certain degree of run-time configurability. We elaborate on the architecture of such a hardware accelerator designed at NXP in the following section.

Function	Description	Stateful/ Stateless	Granularity
(De)Interleaving	Within a block shuffle bits/bytes. Prevents burst errors.	Stateless	Bits to Kilo-Bytes
(De)Puncturing	Bits are removed at transmitter and dummy bits are filled back at receiver. Reduces bandwidth requirement of channel.	Stateless	Bits (0-64)
Low Density Parity Check (LDPC)	Block based error correcting code	Stateless	Bits
Reed Solomon (RS)	Block based error correcting code	Stateless	Hundreds of Bytes
Viterbi	Convolutional error correcting code	Stateful	Bits(2-6)
Turbo	Convolutional error correcting code	Stateful	Bits
(De)Scramble	Re-code bits with a distribution. Disperses signal energy.	Stateful	Bits
Cyclic Redundancy Check (CRC)	Error detecting code. Cannot correct data.	Stateless	Bits(8/16/32)

Table 1.1: *Summary of common coding techniques used for forward error correction in the physical layer of SDR*

1.2 FLORA Architecture

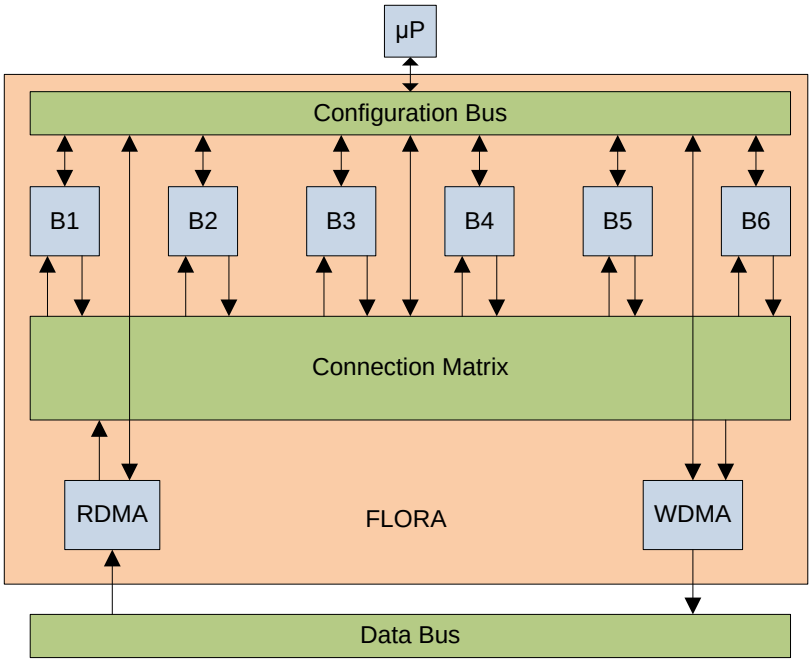


Figure 1.2: *Architecture of FLORA hardware accelerator.*

It was described in the previous section that codec stage in digital baseband processing of SDR is best implemented on a configurable hardware. We have FLORA as a hardware accelerator designed for fast forward error correction in the outer receiver. A general overview of architecture of FLORA is explained in this section. The specific details of the current FLORA architecture are described in appendix B.

Figure 1.2 shows a general architecture of FLORA. It consists of different hardware blocks. Each block is capable of processing data for a specific decoding technique, those mentioned in table 1.1. Each block can be configured for different set of equations and for different inputs/outputs. Blocks *B1 to B6* in figure 1.2 are hardware blocks. Read Direct Memory Access (RDMA) and Write Direct Memory Access (WDMA) are special hardware blocks designed to transfer data in and out of FLORA respectively. All the blocks can be connected to the RDMA and the WDMA via the *Connection Matrix*. The connection matrix also provides connections between certain hardware blocks. All blocks have input and output buffers. Some blocks have big buffers (1 Mbit) and some are have small buffers (8 bytes). All blocks are non-preemptible. Prempting by re-configuring the configuration which is still running will corrupt the data.

To configure FLORA we need to decide the set of blocks to be programmed. After deciding the set of blocks to be programmed, a necessary and complete configuration of FLORA implies

- configuration of each block in FLORA for the mathematical equation, parameters and input/output size.
- configuration of the connection matrix
- configuration of RDMA and WDMA with the appropriate data source and sinks addresses and sizes.

Such a combined complete configuration is called as a *slot*. Blocks within FLORA are data-driven, that is once configured a block triggers as soon as there is data in input buffer. However a slot can only be programmed by some external entity such as general purpose processor shown by the block μP in figure 1.2. For example presently ARM processor configures a slot on the MARS MPSoC. Therefore a slot is not data-driven.

1.3 Problem Description

It is desired to multiplex maximum applications on FLORA. Given a set of applications to be implemented on FLORA it is required to find the optimal configurations for the input applications. An analytical method is required to exploit flexibility of FLORA configurations and also to guarantee performance requirement of each application. There are two problems in applying such an analytical method.

1. SDR applications that we consider consists of tasks partly mapped to FLORA hardware blocks and partly mapped to other parts of MARS MPSoC such as ARM processor. Moreover there is a hierarchy in scheduling. Within the same application tasks mapped to hardware blocks in FLORA which are collectively configured as a FLORA configuration are scheduled internally with self-timed schedule (tasks are activated as soon as data is available) that is because FLORA hardware blocks are data-driven. Whereas a FLORA configuration itself is activated by ARM and so are tasks mapped to ARM processor. We explain in chapter 2 that we use SDF model of computation to model SDR applications. Figure 1.8 shows Digital Video Broadcasting -Terrestrial (DVB-T) standard expressed in an SDF graph. Actors $DI1, DP, V, DI2, RS, DS$ are mapped to FLORA hardware blocks. Whereas So, $Sy + Re$ and Si are mapped on ARM processor.
2. There are finitely many ways to configure FLORA hardware blocks for the same application. It is required to find configurations that satisfy throughput and buffer constraints for a given set of applications and reduce the scheduling load on ARM processor. FLORA consists of diverse hardware

blocks for decoding common kernels used in SDR applications. FLORA has RDMA and WDMA blocks to transfer data in and out of FLORA respectively. FLORA has a configurable flexible datapath so that the RDMA block and WDMA block can serve any hardware block within FLORA. Therefore there are finitely many possible ways to group tasks mapped to hardware blocks in FLORA as a FLORA configuration. Smaller groups mean fewer blocks are used in a FLORA configuration which enables pipelining via buffering in memories outside FLORA. However smaller configurations may incur excess read/write overhead for channels in and out of FLORA thus reducing throughput. Bigger groups means more blocks are used in a FLORA configuration which incurs less read/write overhead but pipelining is limited due to small buffers within FLORA and non-preemptive nature of a FLORA configuration. Therefore there is a trade-off between read/write overhead and pipelining for FLORA configurations. Both of these factors affect throughput.

Therefore there are two challenges - being able to model hierarchical schedule for applications mapped to MARS MPSoC and choosing the best configuration for FLORA among available possibilities.

With a simple example we shall illustrate here why we have multiple configuration options for FLORA and why is it difficult to choose the best configuration. Consider a simple application shown in figure 1.3. *So* and *Si* are source and sink actors. *A,B,C* are actors mapped to FLORA blocks.

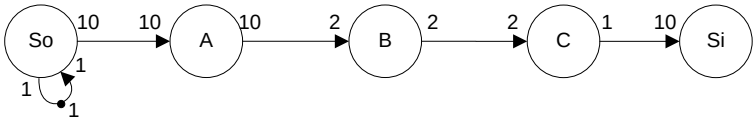


Figure 1.3: *A simple application SDF graph.*

A possibility of configuration is to combine actors *A,B,C* as a single configuration. As we shall explain in chapter 4, to tackle both the problems mentioned earlier, we model FLORA configuration as a clustered actor. The clustered actor represents an atomic firing of a number of iterations of the sub-graph which is clustered. The resulting clustered graph for the biggest configuration option is shown in figure 1.4. In figure 1.4, the SDF graph on top is the implementation SDF graph for FLORA with the specified buffers and execution times. For fetching data inside FLORA and transferring data out of FLORA, RDMA R_1 and WDMA W_1 are inserted. The self-timed schedule for this sub-graph is shown below the sub-graph. Assuming FLORA is configured for one iteration of this sub-graph, the total time to execute one iteration is 41. Thus the resulting clustered graph is shown below the self-timed schedule with a clustered actor R_1ABCW_1. The throughput of SDF graph can be calculated by number of source data units processed per unit time. The Maximum Cycle Mean (mcm) of the clustered SDF graph is 41. Therefore the throughput of this configuration is 10 data units per 41 time units that is $10/41 = 0.244$.

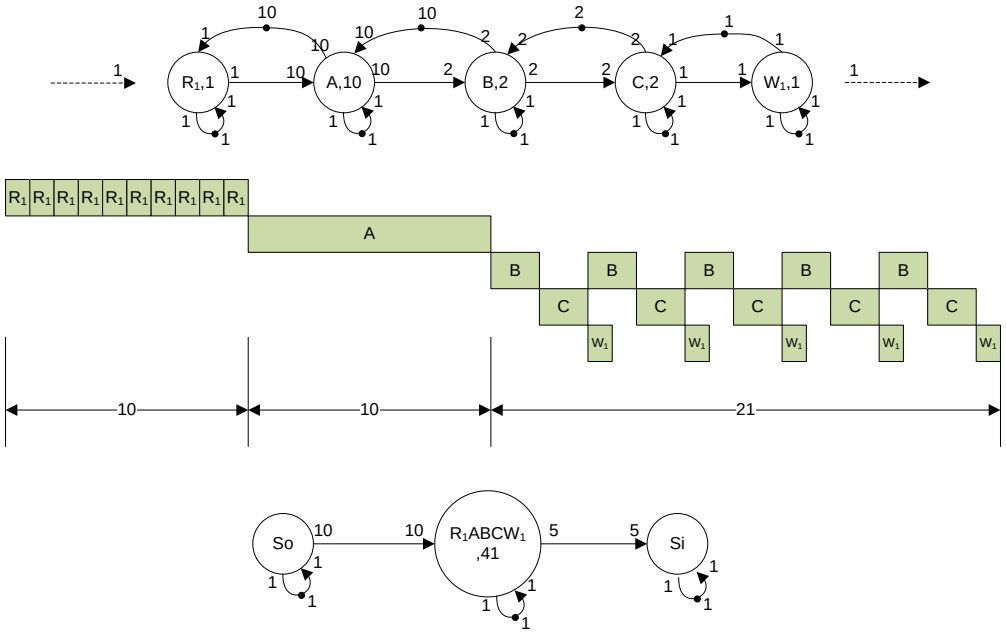


Figure 1.4: *Clustering option 1. The top SDF graph shows the sub-graph mapped to FLORA with DMAs, buffers and execution times. The self-timed schedule for the sub-graph is shown below the sub-graph. The resulting clustered SDF graph is shown at the bottom.*

Another configuration possibility is to split ABC into A and BC. This possibility is shown in figure 1.5. The respective two sub-graphs are shown above in the figure 1.5. The resulting clustered graph with execution times equivalent to one firing of individual sub-graph is shown below in the figure. The throughput of this configuration is $10/30 = 0.333$. The throughput improves due to pipelining with buffering in external memories.

A third configuration possibility is to split ABC in A, B and C. This possibility is shown in figure 1.6. The respective sub-graphs are shown at the top. We assume we have only two sets of DMAs. Therefore configuration B and configuration C share R_2 and W_2. To model the sharing of resources in round-robin modeling we add execution times of actors sharing resources. Therefore in the bottom SDF graph the execution time of actors R_2BW_2 and R_2CW_2 are redefined as 43 (22+21). The throughput of this configuration option is $10/43 = 0.233$. The throughput reduces due to sharing of DMAs.

Figure 1.4, 1.5, 1.6 had number of clusters as 1,2,3 respectively. The throughput performance of these options is shown in figure 1.7. Thus we see with different clusterings we get different throughput. It is desired to choose configurations which give maximum throughput or allow multiplexing maximum applications.

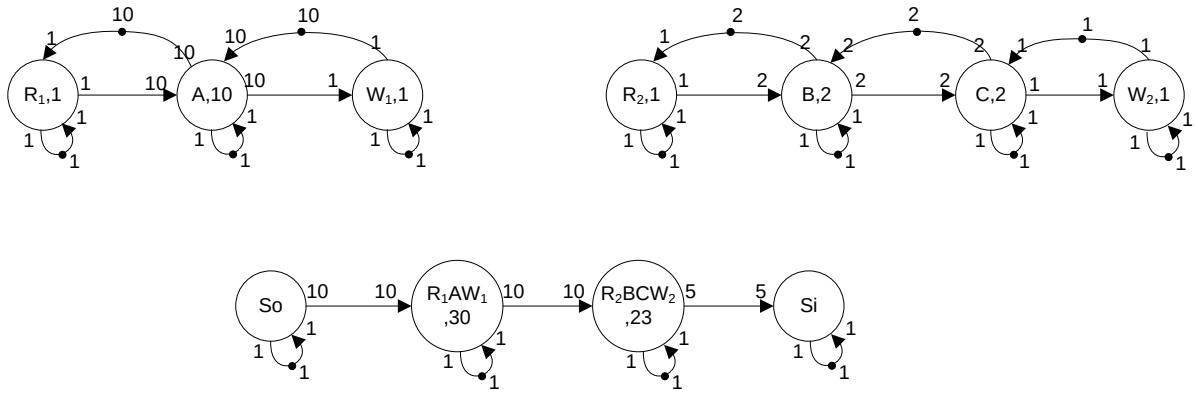


Figure 1.5: *Clustering option 2. We have two FLORA configurations-* A *and* BC*. The respective sub-graphs are shown at the top. The resulting clustered SDF graph is shown below.*

As we can observe in figure 1.7, a configuration option that gives maximum throughput is not simply the biggest or the smallest but it can be in between. Therefore in general it is difficult to find configuration option for FLORA to multiplex maximum radio applications.

We shall illustrate the mentioned problems by explaining two configuration possibilities for a real application of DVB-T standard shown in figure 1.8.

Configuration option 1

Figure 1.9 explains configuration option 1. Figure 1.9(a) shows the original SDF graph of application with nodes (also called as actors in dataflow terminology) in orange are mapped to hardware blocks in FLORA. The other actors are mapped on ARM processor. Hardware blocks in FLORA cannot read data from memories outside FLORA without Direct Memory Access (DMA) blocks. Therefore it is required to insert DMA actors in the application SDF graph to enable using FLORA hardware blocks. Here we have freedom to insert RDMA and WDMA blocks at different possible locations. Figure 1.9(b) shows possibility 1 which identifies configuration option 1. *W1,W2* are WDMA actors. *DI1,DI2* are actors mapped to De-Interleaver which has its own RDMA. Therefore no RDMA actors are inserted for *DI1*. There are two problems with SDF graph in figure 1.9(b). First, actors shown in orange in SDF graph shown in figure 1.9(b) are scheduled differently than the rest of the actors. Secondly for SDF graph shown in figure 1.9(b) we have two FLORA configurations (collective configuration of *DI1+DP+V+W1* and collective configuration of *DI2+RS+DS+W2*) which are non-preemptible. In section 3.2 we shall explain we tackle both of these problems by modeling FLORA configuration as a clustered atomic SDF actor. Atomicity of SDF actor captures non-preemptibility and it abstracts from the

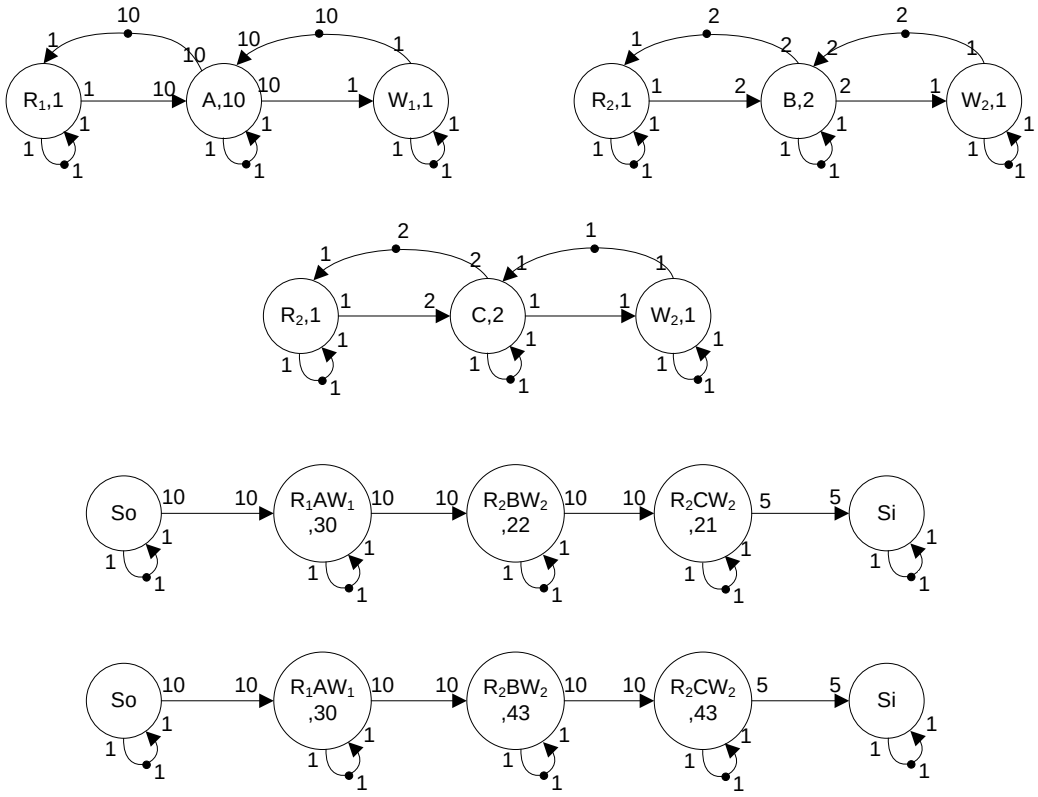


Figure 1.6: *Clustering option 3.* A*,* B *and* C *are configured separately.* B *and* C *share DMAs. Therefore the resulting clustered SDF graph is redefined with new execution times in the bottom most SDF graph.*

schedule of the sub-graph within the clustered atomic actor. The resulting SDF graph is shown in figure 1.9(c). Figure 1.9(c) will now be useful in evaluating performance of configuration option 1.

Configuration option 1 shown in figure 1.9(c) has two big configurations - *(DI1+DP+V+W1)*) and *(DI2+RS+DS+W2)*. Since a FLORA configuration is non-preemptible, all the blocks in a FLORA configuration have to wait until the complete configuration finishes and cannot be configured and triggered for a different configuration. Therefore these blocks cannot be used for pipelining different iterations of the whole SDF graph or for actors from other applications.

Configuration option 2

FLORA configuration option 2 is illustrated in figure 1.10. Here we split *(DI1+DP+V+W1)* into *(DI1)* and *(R1,DP,V,W1)*. Thus the total number of configurations are 3 - *(DI1)*, *(R1,DP,V,W1)* and *(DI2+RS+DS+W2)*, this can

Figure 1.7: *Throughput vs. Number of Clusters for simple example shown in figure 1.3.*

be seen in figure 1.10(c). All the edges in 1.10(c) are stored in memories outside of FLORA. Hence we assume we have sufficient buffer space. Therefore because of splitting configuration and available buffer space it is now possible to configure hardware block De-Interleaver for firing of *DI1* for next iteration of SDF graph or possibly to use it for firing of other actors (mapped to *De-Interleaver*) from other SDF graphs.

On the other hand by making smaller configurations, we include more DMA transfers within every FLORA configuration thereby increasing the read/write overhead. For SDR applications where big chunks of data are processed, this could be a bottleneck. For example in figure 1.10(c) we add the overhead of *R1* in the configuration *(R1+DP+V+W1)*.

For the present version of FLORA, configuration option 1 utilizes FLORA for 29% (ratio of required throughput to maximum obtainable throughput). Therefore the remaining 71% is slack which can be utilized for multiplexing more SDR applications. Configuration option 2 utilizes FLORA for 34%. Therefore we see for DVB-T standard the read/write overhead is significant and configuration option 2 provides less slack for other SDR applications.

Therefore depending on the SDR application bigger configurations could be suitable where read/write overhead is significant whereas if read/write overhead is not significant than smaller configurations could be more useful to allow more

pipelining for multiplexing more SDR applications.

1.4 Contributions

An analytical method is required to exploit flexibility of FLORA configurations and also to guarantee performance requirement of each application mapped on FLORA. In this thesis such an analytical method is proposed based on clustering of SDF actors. The contributions of the thesis are as follows:

- A definition of clustering for the SDF model of computation is presented where clustered sub-graphs are modeled as a single SDF actor. This definition enables following advantages
  - Each clustered actor represents a FLORA configuration. Thus the definition enables capturing the flexibility offered in configurable hardware such as FLORA in a formal model (SDF).
  - Analyzing the trade-off between read-write overhead and constrained pipelining for FLORA configurations. The communication cost of channels (or edges in SDF graphs) are modeled with the help of RDMA and WDMA actors. The modeling of finite non-zero communication overhead is crucial for finding the optimal clustering options for MPSoC with DMAs.
  - Modeling of hierarchical scheduling for a heterogeneous MPSoC. Sub-graphs within a clustered actor can be scheduled differently than the parent clustered graph which contains mixture of clustered and unclustered actors. For instance hardware blocks in FLORA are data driven, therefore sub-graphs within a clustered actor are self-timed scheduled. Whereas clustered graphs containing actors mapped to different components of MARS MPSoC (memories, ARM processor, FLORA) are scheduled using static-assignment (Round-Robin) strategy.
- For the proposed definition of clustering, exact and heuristic algorithms are presented to find the optimal clusterings in multiple applications to satisfy throughput constraint of each application. An optimal clustering is an option which satisfies throughput and buffer constraints and also minimizes scheduling load on the entity which configures and triggers FLORA (ARM processor on MARS MPSoC).
- A condition has been presented to avoid clusters which will result in deadlock. Exact and efficient algorithms to check for this condition are also presented.

1.5 Overview

This document has been organized as follows.

In chapter 2 we explain the SDF model of computation and the associated definitions. The notations and definitions defined in this chapter are used throughout the rest of the report.

In chapter 3 we describe the challenges and details of our problem domain. We clarify our goal, assumptions and the problem statement of this thesis. At the end of chapter we state the related work.

In chapter 4 we define a clustering function which given an input SDF graph and a set of nodes to be clustered, produces a clustered SDF graph as output. We give a condition to check for deadlock-free clusters. We guarantee by giving proofs that the clustered graphs generated by the defined clustering function is consistent and deadlock free.

In chapter 5 we present an algorithm to check for deadlock -free clusters. We also present an exact and a heuristic clustering algorithm to find optimal clusterings in given input SDF graphs. We describe experiments conducted to evaluate performance of the clustering algorithms.

In chapter 6 we summarize the work of this thesis. We propose a generalized application of clustering definition presented in this thesis for scheduling problems in MPSoC domain. We conclude by stating possible extensions and improvements for the work presented in this thesis.

Appendix A describes architecture of MARS MPSoC. Appendix B describes the architecture of FLORA in detail.

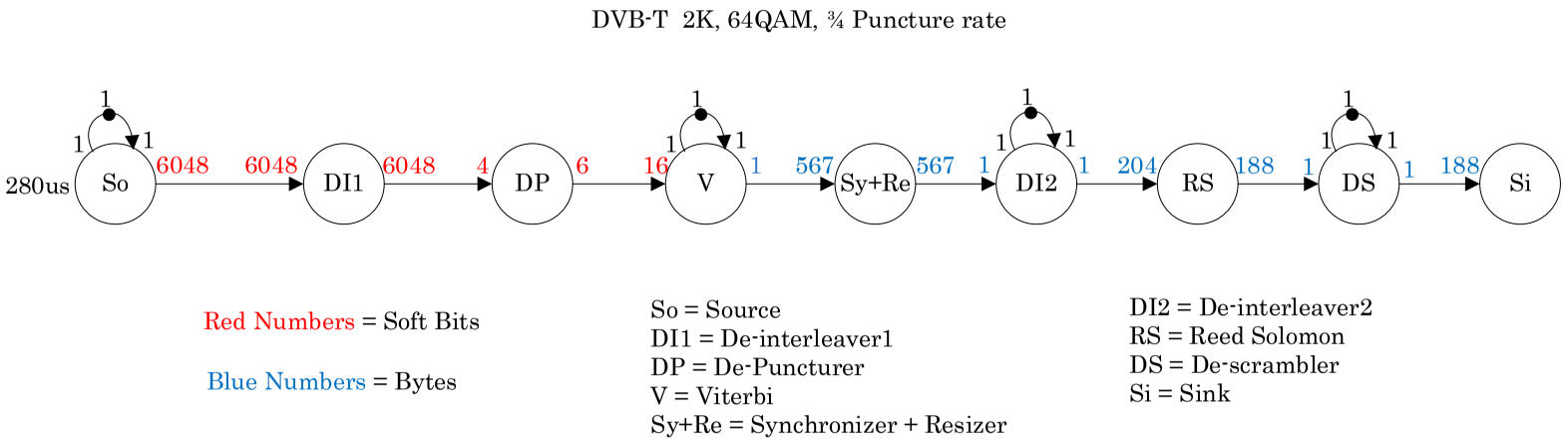


Figure 1.8: *SDF graph model of outer receiver of DVB-T standard 2K mode, 64QAM, 3/4 Puncture Rate*

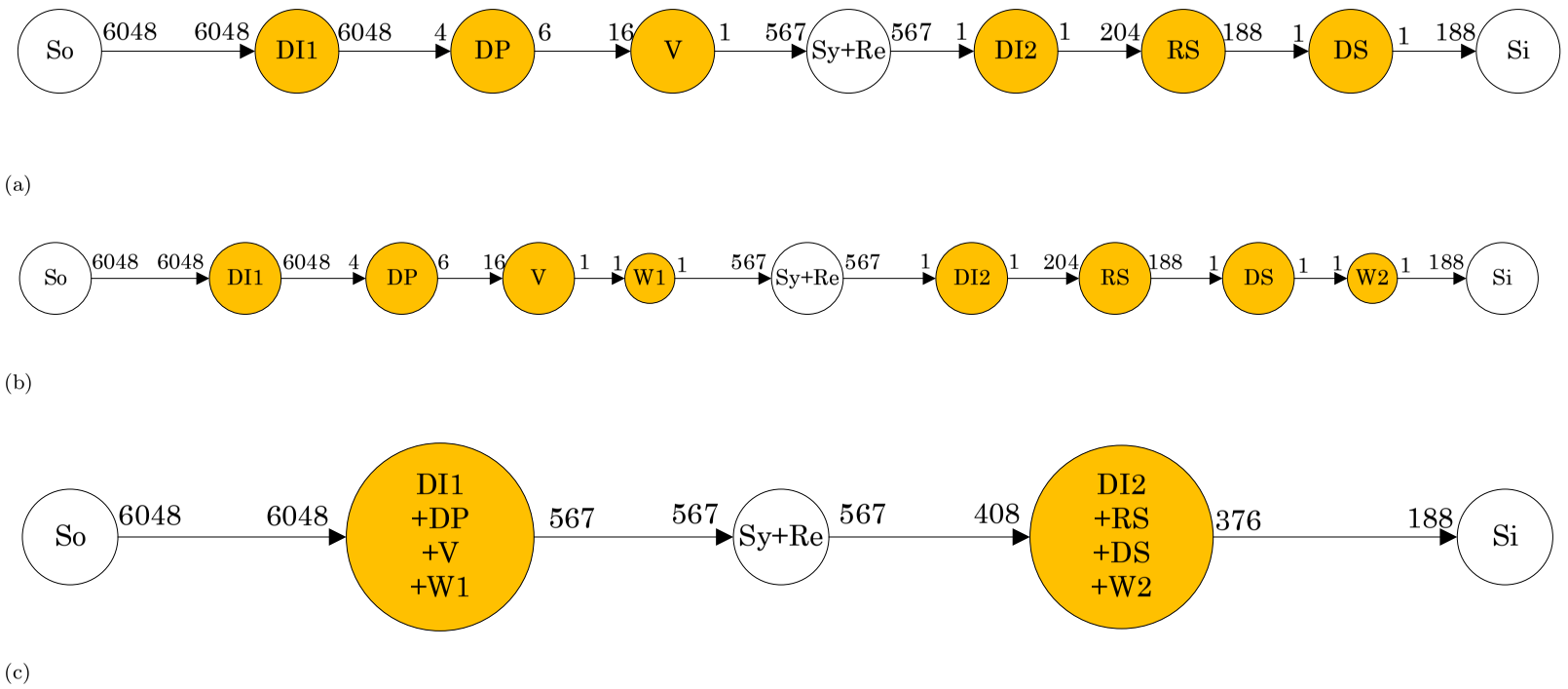


Figure 1.9: *(a) Application SDF graph (assuming implicit self-edges). (b) Application SDF graph with DMA actors. (c) Modeling of configuration 1 obtained from (b).*

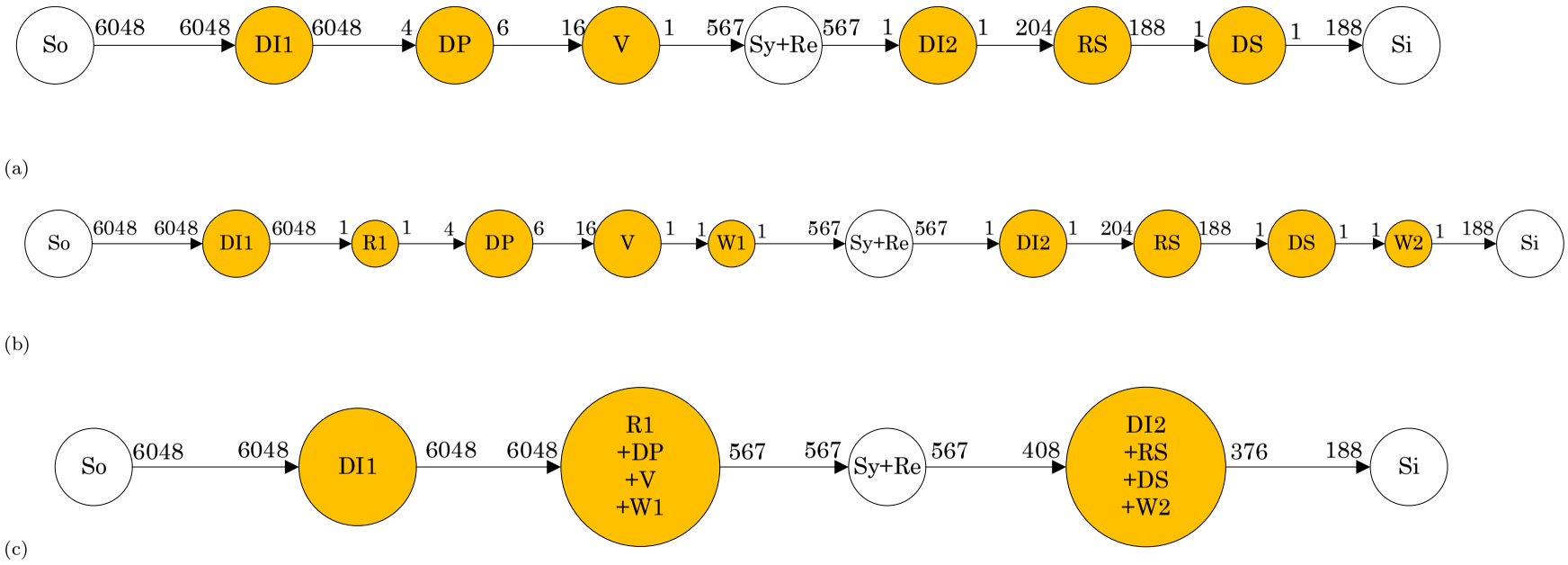


Figure 1.10: *(a) Application SDF graph (assuming implicit self-edges). (b) Application SDF graph with DMA actors. (c) Modeling of configuration 2 obtained from (b).*

Chapter 2

Dataflow Preliminaries

Software Defined Radio (SDR) applications demand certain performance guarantees. Therefore for programming embedded systems for such applications a model-driven approach is preferred [1, 3, 18] so that we can guarantee predictable performance at design stage. Literature suggests dataflow model of computation are well suited for streaming applications (applications which can iterate forever) such as multimedia applications. Different dataflow models offer different expressibility, analyzability, succinctness [14]. The simplest model is Homogeneous Synchronous Data Flow (HSDF), with highest analyzability but poor expressibility. SDF graphs are just sufficient to express SDR applications and still possess a useful level of analyzability. Therefore we use SDF as a model of computation to model SDR. To be more specific we shall use SDF to model only the outer receiver part of a SDR receiver, since design-flow of this part of MPSoC is of interest to us.

In this chapter we describe the Synchronous Data Flow (SDF) model of computation and related definitions. In section 2.2 we describe Homogeneous Synchronous Dataflow Graphs. In section 2.3 we describe the modeling of throughput constraint for SDR application. The notations and definitions defined in this chapter are used throughout the rest of the thesis report.

2.1 Synchronous Data Flow Graphs

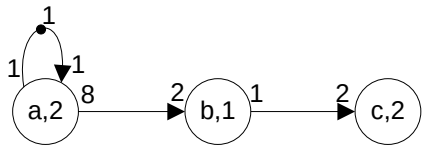


Figure 2.1: *An example of SDF graph*

SDF was first introduced in [11]. Figure 2.1 shows an example of a SDF graph. The nodes *a,b,c* are called as actors which are connected to each other by dependency edges also called as channels. Since we may deal with multigraphs, where multiple edges between two actors are possible, we define actors

as collection of ports. So that an edge can be uniquely defined as a tuple of (outputPort,inputPort). We define an actor as follows.

Definition 1. *(Actor)* Assume P is set of *Ports*. An actor a is a tuple (I, O) consisting of $I \subseteq P$ input ports denoted by $I(a)$ and set of $O \subseteq P$ output ports denoted by $O(a)$ with $I \cap O = \phi$. q^v denotes that q is a port of actor v. A port can belong to only one and unique actor.

A token is a container in which fixed amount of data can be stored. The black dot on the self-loop edge of actor a in figure 2.1 is a token. With each port is a rate associated with it which gives the amount of tokens consumed on an input port or the amount of tokens produced on an output port. In figure 2.1, the numbers shown nearby the head and tail of an edge are the rates associated with those ports. By definition actors consume tokens from all incoming edges via input ports and produce tokens on all outgoing edges via output ports *atomically* when they *fire*. An actor can *fire* only when there are sufficient tokens as required by the respective input ports on all incoming edges. *Firing* of an actor takes finite time called as response time. Response time is defined as the difference between finish and start of a firing of an actor. In figure 2.1, the numbers inside the actors give us the response time of the respective actor. For many reasons we sometimes need initial tokens to be placed on some edges. In figure 2.1 the self-loop edge of actor a has one initial token placed on it. Without this token graph would deadlock.

Synchronous Data Flow are also called as Static/Multirate Data Flow. The word static refers to the fact that the port rates and the response times remain the same for all possible iterations of a SDF graph. Multirate refers to the fact that ports have different rates relative to each other. In fact the static nature of SDF makes it very analyzable and multirate property gives its expressibility. We can now define functions for a SDF graph giving the static initial tokens, response times, production and consumption rates. We summarize these concepts in our definition of SDF graph in definition 2.

Let $\mathbb{N}_0 = \{0, 1, 2..\}$ be set of natural numbers and $\mathbb{N} = \mathbb{N}_0 \setminus \{0\}$.

Definition 2. *(SDF graph)* A *SDF graph* is a directed graph $G(V, E, \delta, \varphi, \pi, \mu)$. V consists of finite set of actors connected by set of edges $E = \{(p^s, q^d) | (s \in V) \wedge (d \in V)\}$. All ports of all actors are connected to precisely one edge. $\delta : E \to \mathbb{N}_0$ gives initial tokens placed on every edge in E. The response time $\varphi : V \to \mathbb{N}_0$ gives the difference between finish and start time of a firing of every actor in V. For every edge $e = (p^s, q^d) \in E$ the function $\pi : E \to \mathbb{N}_0$ gives the number of tokens produced by actor s on port p and $\mu : E \to \mathbb{N}_0$ gives the number of tokens consumed by actor d on port q.

2.2 Homogeneous Synchronous Dataflow Graph

SDF graphs are useful because they express only true data dependencies, thus allowing maximum parallelism. For instance in example of figure 2.1 the SDF

graph only says for 1 firing of actor a, b can fire 4 times and c can fire 2 times. With this information the 4 firings of b and 2 firings of c can take place in parallel. This becomes clear in the equivalent HSDF or also called as Single Rate Data Flow (SRDF) of the SDF graph. The equivalent HSDF graph for example in figure 2.1 is shown in figure 2.2.

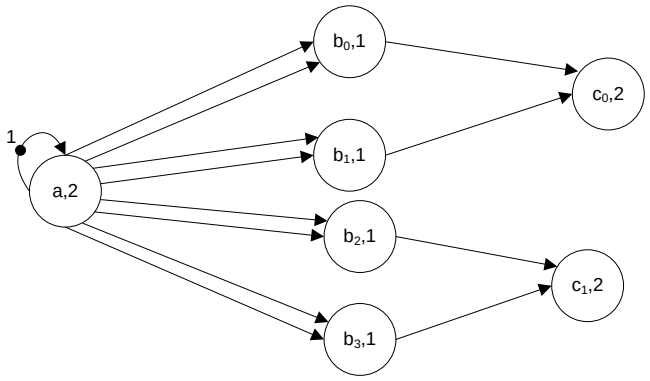


Figure 2.2: *Equivalent HSDF graph of example in figure 2.1*

In HSDF all actors produce or consume only single tokens on all edges. Therefore we do not show port rates in figure 2.2. Every SDF graph can be converted into its equivalent HSDF graph as described in [13]. The parallelism available in any application becomes more visible in its equivalent HSDF graph.

Actors are usually used to model tasks or functions in an application. With actors, we completely abstract from the functionality of task and simply consider the amount of tokens they consume (possibly zero) or they produce (possibly zero) and this production and consumption is atomic.

Consistency and Repetition Vector

Deadlock and Consistency are two important properties of SDF graphs. Inconsistent SDF graph will require either unbounded memory or may deadlock. When a SDF graph deadlocks no actor is able to fire either due to inconsistency or due to insufficient initial tokens. A SDF graph is *consistent* if it has non trivial *repetition vector*. We define them in definition 3.

Definition 3. *(Repetition Vector,Consistency)* A *repetition vector* γ of a SDF graph $G(V, E, \delta, \varphi, \pi, \mu)$ is a function $\gamma : V \to \mathbb{N}_0$ such that $\forall e = (p^s, q^d) \in E : \gamma(s) \times \pi(e) = \gamma(d) \times \mu(e)$. A *repetition vector* is called non-trivial if and only if $\forall v \in V : \gamma(v) > 0$. A SDF graph is *consistent* if and only if it has a non-trivial *repetition vector*. For a *consistent* SDF graph there is a unique smallest non-trivial *repetition vector* called as *the repetition vector* of a SDF graph.

For example, the repetition vector of SDF graph in figure 2.1 is $\{\gamma(a) = 1, \gamma(b) = 4, \gamma(c) = 2\}$. Hence the SDF graph is consistent.

Throughput

Throughput of an SDF graph can be computed with mcm analysis on its equivalent HSDF graph [13]. For a strongly connected HSDF graph, we can compute a cycle mean as follows

$$cm(c) = \frac{\sum_{v \in V(c)} \varphi(v)}{\sum_{e \in E(c)} \delta(e)}$$

where *V(c)* is the set of actors traversed by simple cycle c and *E(c)* is the set of edges traversed by simple cycle c. A simple cycle is a cycle that traverses actors maximally once. The *mcm* of the equivalent HSDF graph is given by

Definition 4. *(mcm)* For an HSDF graph $mcm : G \rightarrow \mathbb{N}_0$ is defined as

$$mcm(G) = \max_{c \in C(G)} cm(c)$$

Where *C(G)* is the set of all simple cycles in HSDF graph G.

The maximum attainable throughput of the HSDF graph is $\frac{1}{mcm}$.

2.3 Throughput Constraint for SDR applications

Throughput specified as $\frac{1}{mcm}$ gives the rate at which a SDF graph iterates (that is all the actors in an SDF graph fire as many times as required by the repetition vector). This is the computed throughput. For checking throughput constraint, we compare the computed throughput with a specified value by the application. For practical radio applications throughput constraint is usually specified as the minimum rate at which a source or sink actor should fire. For example in figure 2.3 we show an example of the DVB-T standard. The SDF graph in figure 2.3 models the outer receiver part of the standard. The standard requires that input should be periodically accepted for every $280\mu s$. We model this constraint using a source actor *So* with a response time of $280\mu s$ as shown in figure 2.3. For radio applications, firing of source actor represents periodic input. Therefore the firings of source actor are never concurrent. Therefore we show a self-edge on the source actor.

To check throughput constraint for the SDF graph in figure 2.3 is to check if actor *So* fires periodically with period $280\mu s$. Depending on the production and consumption rate, the *mcm* of an SDF graph may cover multiple firings of source actor. What is necessary is that all the firings of source actor take periodically with period of $280\mu s$.

Therefore we specify throughput constraint for SDR application as

$$mcm(G) = \gamma(S_o) \times \varphi(S_o)$$

Where *So* is the source actor of SDF graph G.

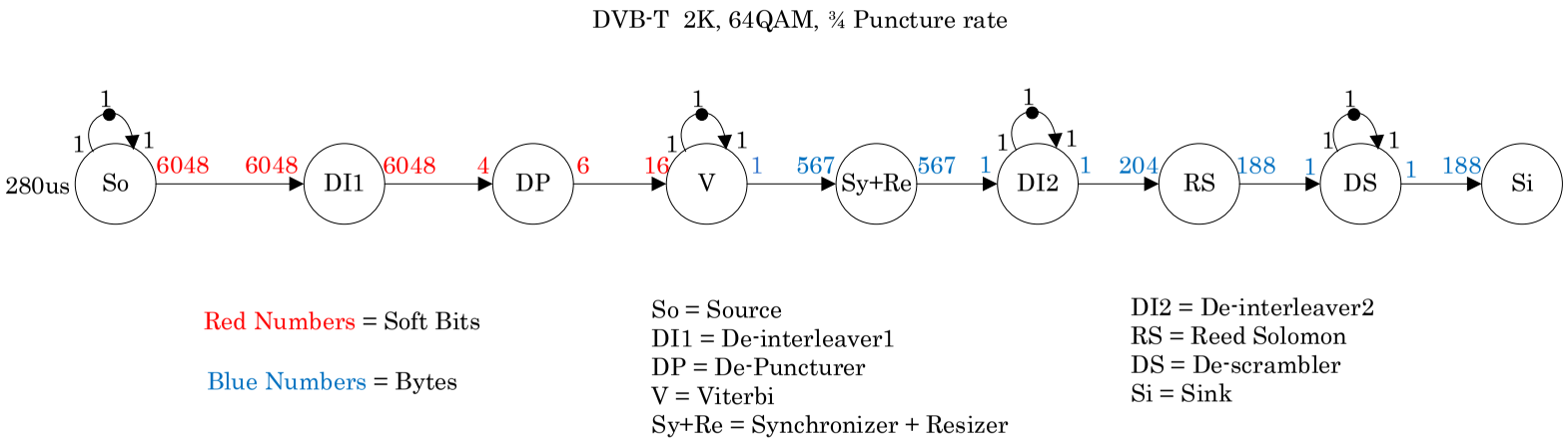


Figure 2.3: *SDF graph model of outer receiver of DVB-T standard 2K mode, 64QAM, 3/4 Puncture Rate*

Chapter 3

Problem Formulation

In was discussed in chapter 1 that FLORA can be configured in different possible ways for the same application. Each configuration gives different throughput. An analytical method is required to model and analyze the effect of different configurations for FLORA. This problem is addressed in this thesis. In this chapter we elaborate the addressed problem. We translate the details of our problem in the dataflow model of computation so that we can solve the problem formally. In section 3.2 the identified challenges are descried. In section 3.3 we explain the assumptions of the thesis. In section 3.4 we present the formal problem statement of the thesis. At the end of the chapter in section 3.5 we mention the related work.

3.1 Goal

Configurability of FLORA hardware blocks and configurability of the connection matrix in FLORA enables us to support multiple applications representing different standards simultaneously. We wish to exploit the flexibility offered by FLORA hardware accelerator to support multiple SDR applications and at the same time guarantee performance requirements of each standard. Therefore the goal of the thesis defined as follows - *Given a number of applications to be implemented on FLORA, to find the optimal configurations for FLORA such that throughput constraint of each application is satisfied.* An optimal configuration satisfies throughput and buffer constraints and minimizes scheduling load on ARM processor.

Before describing the addressed problem of this thesis, we describe in the following section the challenges in implementing SDR applications to an MPSoC.

3.2 Challenges

While porting SDR applications to MARS MPSoC, we identify following aspects where we have room for improvement.

1. **Scheduling** tasks of application onto resources of MARS platform
2. Obtaining FLORA **configurations**.

We shall explain these two points in the following sections in detail.

Scheduling

In this section we shall explain what is meant by finding a schedule. Given an application model consisting of tasks (also called as actors in dataflow terms) and a number of available resources (on the MPSoC platform), finding a schedule involves taking following decisions. We need to decide on which resource an actor should be executed, the order in which actors should be fired and the exact invocation time of each actor. For each of these decisions we have in general two choices. We can decide them at compile time (during design phase) or during run-time. Taking decisions at run-time gives some dynamism and flexibility. On the other hand dynamism reduces predictability and it implies certain run-time overhead. This run-time overhead may reduce the maximum obtainable performance. The opposite holds for compile-time decisions. Flexibility is reduced since we make one-time decisions. But we can guarantee predictable performance, we can also obtain the maximum possible performance. Based on the choice of compile-time or run-time decision for the above mentioned three aspects, we can have different scheduling strategies (see [1,13]). These are mentioned in table 3.1.

Scheduling strategy	Assignment	Ordering	Invocation Time
Fully-static	Compile-time	Compile-time	Compile-time
Static-Order	Compile-time	Compile-time	Run-time
Static-Assignment	Compile-time	Run-time	Run-time
Fully-dynamic	Run-time	Run-time	Run-time

Table 3.1: *Comparison of scheduling strategies [1].*

We use static-assignment scheduling strategy for MARS MPSoC. In section 3.3 we explain our reasoning behind choosing static-assignment schedule.

FLORA Configuration Possibilities

We have the configurable hardware accelerator - FLORA. Therefore scheduling a SDF graph application on FLORA means configuring FLORA hardware blocks as per the scheduling strategy for every present SDF actor present in the application model. Programming FLORA means programming a FLORA slot as was explained in section 1.2. Configuring a FLORA slot consists of

- Selecting and configuring FLORA hardware blocks
- Configuring RDMA and WDMA with appropriate source and sink.

Thus to find FLORA configurations for an application is equivalent to answer the questions - How many slots should be formed to completely map an application on FLORA? And more importantly which FLORA blocks should be selected with what type of configuration for each of these slots? Such configurations gives us freedom to choose a number of hardware blocks to be programmed

together in a slot. Before we elaborate the necessity, significance and implications of such freedom, we first formalize this idea of freedom for SDF graph.

When we program a number of hardware blocks in FLORA together in a complete configuration as a slot, such a slot can be triggered when input data is available, the slot cannot be pre-empted, execution of slot is atomic that is we cannot re-trigger hardware blocks of a slot in execution for next set of input data. Once output data is transferred to an output First In First Out (FIFO), we can re-trigger this slot for next iteration. Now this behaviour of a slot is in direct correlation with the definition of an actor explained in chapter 2. Therefore we define a slot of FLORA as a composite actor. The composition refers to the grouping of actors in the original SDF graph, but this newly defined composite actor has same semantics as that of a SDF actor defined in definition 1. We call the process of grouping a number of actors into a new actor as *clustering*. We shall formalize clustering in chapter 4.

With the concept of *clustering* we have expressed the flexibility available in programming FLORA purely in semantics of SDF. This shall help us in developing analytical methods with SDF to find the best configuration possibilities for FLORA.

Before going to problem statement we shall enumerate the assumptions of the thesis in the following section.

3.3 Assumptions

In this section we explain the assumptions in our thesis. For multiplexing maximum possible SDR applications on FLORA and the MPSoC platform we are required to find the optimal mapping, optimal schedule and also the optimal clustering. This is a multi-level problem. Moreover each level is inter-dependent, that is decisions at a level affect decision at other levels, hence we cannot find solution to each level in isolation. Therefore we make some simplifying but reasonable assumptions. We first explain the problem at each level and then our simplified assumptions.

Mapping

Tasks mappable to FLORA hardware blocks can be mapped to those blocks, but if we are short of hardware resources in FLORA then we can also map the excess tasks on a general purpose processor(s). Thus we get multiple combinations. To simplify the complete problem we assume we have only one general purpose processor (presently ARM processor). Since tasks mappable to FLORA are fastest executed on FLORA, we assume fixed mapping of tasks. Tasks are mapped to the respective FLORA hardware block if such a block exists in FLORA or to ARM processor otherwise.

Scheduling

Of the different scheduling strategies mentioned in table 3.1 we cannot have a fully-static schedule for FLORA because we cannot have static invocation times. Hardware blocks in flora are self-timed triggered, that is they trigger themselves as soon as data arrives. This is only true within a FLORA slot. A configuration is programmed by an external processor such as ARM and this is event triggered based on the output of inner receiver.

Static-order is a promising choice, since it is possible to find the best ordering of tasks at compile time. To be able to compute a static-order schedule for multiple applications sharing resources, it is necessary that the tasks in multiple applications are synchronized. Without guaranteed synchronization of multiple applications, there is possibility of relative deviation in arrival times of tasks in different applications. But because we have a static-order schedule, the required change in ordering due to deviation does not take place. Thus applications may either deadlock due to insufficient buffers thus breaking throughput constraint or could result in buffer-overflow resulting in loss of data. For radio applications we cannot guarantee synchronization. A radio application may deviate in its arrival times independently of the other radio application. Therefore we do not use static-order schedule. Thus we make a reasonable conclusion to design a static-assignment schedule. In our static-assignment schedule ordering is decided at run-time with round-robin policy.

Scheduling Load

The thesis problem is framed as an optimization problem. FLORA is a class of hardware which has to be configured and triggered by an external hardware unit(s) (presently ARM processor in our MPSoC). This incurs a scheduling load on this external unit. This scheduling load depending on the application and FLORA configuration can be very high. But the external hardware unit may have other tasks to do as well. Hence it is not desired to load the external hardware unit only for the sake of configuring/triggering FLORA.

Therefore our optimization criteria is to minimize the scheduling load on hardware unit used for configuring/triggering FLORA. This criteria is relevant as long as configurable hardware such as FLORA does not have the capacity to configure and trigger themselves for any arbitrary tasks allotted to them.

3.4 Formal Problem Statement

In previous sections we identified challenges in our problem domain. They concern decisions on mapping, scheduling and FLORA configurations. The focus of this thesis is the problem of finding optimal configurations for FLORA. We have explained the assumptions of the thesis for the aspects of mapping and scheduling in section 3.3.

Revisiting our **assumptions** we assume following points are given:

1. SDF graphs for each application.
2. Assignment of each actor in the SDF graphs onto the resources on MARS platform which includes FLORA hardware units.
3. Worst Case Execution Time of each actor for all SDF graphs on the respective hardware unit to which it is mapped.

These details are accepted via an eXtensible Markup Language (XML) file. These details are specified in chapter 5. Our input, analysis and output are completely on SDF model of computation. Therefore the **output** is as follows:

1. Clustered SDF graphs for each respective input application SDF graph. Where each clustered actor in the output SDF graphs represent a complete FLORA configuration.
2. Buffer sizes for each channel in the clustered SDF graphs

These details are outputted as a normal text file with all required details.

Let S is the set of input SDF graphs representing SDR applications. S' is the set of clustered SDF graphs. V_{src} is the specified source actor for each input application graph g. $Buffer : G \rightarrow \mathbb{N}_0$ gives the total buffer space required for an SDF graph. B is the maximum available total buffer capacity for all SDF graphs. For an SDF graph g representing an SDR application the throughput constraint is stated as $mcm(g) = \gamma(V_{src}) \times \varphi(V_{src})$ where γ is the *Repetition Vector* and φ gives the response time of an actor. $ASL(g)$ gives the absolute scheduling load of an SDF graph g and it is defined in definition 9 as $ASL(g) = |\gamma|/mcm(g)$ where $|\gamma| = \sum_{v \in V} \gamma(v)$. SDF model and other related definitions are explained in chapter 2. The throughput constraint is explained in section 2.3. Then the problem can be formally stated as follows:

Input $S = \{g_1, g_2, ...g_n\}$
Output $S' = \{g'_1, g'_2, ...g'_n\}$

Constraints

1. *Throughput* $\forall g \in S' : mcm(g) = \gamma(v_{src}) \times \varphi(v_{src})$
2. *Buffer* $\sum_{g \in S'} Buffer(g) \leq B$

Minimize $\sum_{g \in S'} ASL(g)$

We enumerate the related work in literature in the following section.

3.5 Related Work

Use of dataflow models of computation for scheduling a MPSoC system have been extensively studied in the literature. Among different aspects of dataflow models, work on scheduling and clustering for SDF are relevant to us.

Given a HSDF graph, computing a fully static schedule for a finite number of processors for a throughput constraint is NP-Complete [2, 11]. Further given a SDF graph, its equivalent HSDF graph can have exponential number of nodes. Therefore in-complete heuristic approaches have been proposed. Lee et.al in [11] propose a heuristic approach based on optimal unfolding of SDF graph to find a Periodic Admissible Sequential Schedule (PASS). S.Stuijk et.al propose a heuristic approach in [15] which works directly on a SDF graph without expanding to an equivalent HSDF graph. Bonfietti et.al. in [2] propose an exact algorithm to find a fully static schedule for HSDF. Static assignment scheduling and resource contention has been studied in depth by A.Kumar in [10].

Given a dataflow model, transforming the original model into a new one by grouping or clustering nodes has also been studied for various motivations as follows.

Book of [13] discusses three clustering algorithms for obtaining a multiprocessor schedule such that the makepsan of the obtained schedule is minimal. These three algorithms consider scheduling a single application represented in HSDF graph on a finite number of processors. In each of the discussed algorithms a cluster of nodes represents a group of tasks scheduled on a single processor. Each edge has an associated inter-processor-communication(IPC)cost. The underlying idea behind clustering in these three algorithms is that the IPC cost for edges within a cluster are zeroed to analyze the performance of schedule.

J.L Pino et.al in [12] present a hierarchical multiprocessor scheduling approach using clustering. For obtaining a multiprocessor schedule SDF graph is translated into its equivalent HSDF or Acyclic Precedence Graph (APG). This translation can give rise to exponential increase in number of nodes in the equivalent HSDF or APG. The goal of [12] is to reduce number of nodes in an SDF graph using their clustering approach so as to reduce complexity of scheduling the SDF graph. Not all clusterings are valid, an arbitrary cluster can deadlock. A theorem is presented in [12] which explain four conditions which guarantee deadlock-free clusters.

S.Bhattacharya et.al in [4] consider a dynamic data flow graph to model the application. Their goal of clustering is to replace SDF sub-graphs appearing in the dynamic data flow graph by a single dynamic data flow actor such that the global performance in terms of latency and throughput is optimized. A quasi-static schedule is then obtained for the clustered dynamic dataflow actor. Thus they try to exploit advantages of static model of computation in the application graph which comprises of more general model of computations.

Masters thesis in [19] is about an earlier version of FLORA hardware. It has similar goal as ours. They find largest configurations for FLORA for multiple applications while meeting throughput constraint for each application. They do not have an optimality criteria.

Comparison with Previous Work

State-of-art work on clustering has been discussed in [4, 12, 13]. Work in [19] is also on FLORA. Therefore we compare our work with [4, 12, 13, 19].

1. We have presented a method to compute optimal clusterings for multiple applications on multiple resources to satisfy throughput constraint of each application. Our proposed method works on SDF model. We model the communication overhead of channels and its effect on throughout of the application with the help of SDF actors (modeling DMAs)which are inserted at the appropriate positions in an SDF graph for every sub-graph to be clustered. Using SDF actors gives better approximation of the communication overhead due to DMAs. Whereas clustering algorithms presented in [13] consider a single application on multiple resources, they consider HSDF model and they consider communication cost by attaching constant weights to edges.

2. Clustering approach in [12] considers a single application in SDF model. They also consider communication cost by attaching constant weights to edges in a Directed Acyclic Graph obtained from SDF graph.

3. We present exact and heuristic algorithms to find the optimal partitioning (clusters that cover the entire application graph) of applications graphs. Algorithms to find clusterings are not mentioned in [12] and [4].

4. While traversing the search space to find the optimal clustering, clusters which introduce deadlock should be avoided. The paper of [12] presents a theorem which specifies four conditions for SDF graphs which allow checking for deadlock-free clusters. No algorithms are presented in [12] to check their four conditions. We present a single condition on equivalent APG which covers the four conditions presented in [12]. We present exact and efficient algorithms to check for deadlock-free clusters using our condition.

5. Clustering defined in [12] and [4] considers only connected actors. We do not limit our search space with this constraint. In our exhaustive search approach all valid clustering options (those which do not result in deadlock) are tried which can also comprise of unconnected SDF actors.

6. Masters thesis of [19] is based on an earlier version of FLORA and they also find FLORA configurations for multiple application to satisfy throughput constraint of each application. Our work differs with their approach as follows.

    - Figure 3.1 shows the architecture of FLORA used in [19]. Architecture of FLORA that we use is shown in figure B.1 and B.2. FLORA in [19] had no RDMA (data enters only via De-Interleaver) and one WDMA. Whereas our version of FLORA has one RDMA and two WDMAs (and one De-Interleaver with its own RDMA and WDMA).
    - Due to the fact that no separate RDMA is present and only one WDMA is available, the possible FLORA configurations in [19] are limited. Moreover the approach presented in [19] always results in one clustering option (the largest possible cluster with actors mapped to FLORA hardware blocks). Therefore approach presented in [19]

completely excludes multiple possibilities of clustering that exist in any future version of FLORA.

- We exploit the fact that the present FLORA has RDMA and WDMA blocks. With availability of RDMA and WDMA all possibilities of clustering exist in an SDF graph. Unlike [19] we consider all available possibilities.
- Work in [19] does not consider any optimization criteria. We take scheduling load on the ARM processor as an optimization criteria.

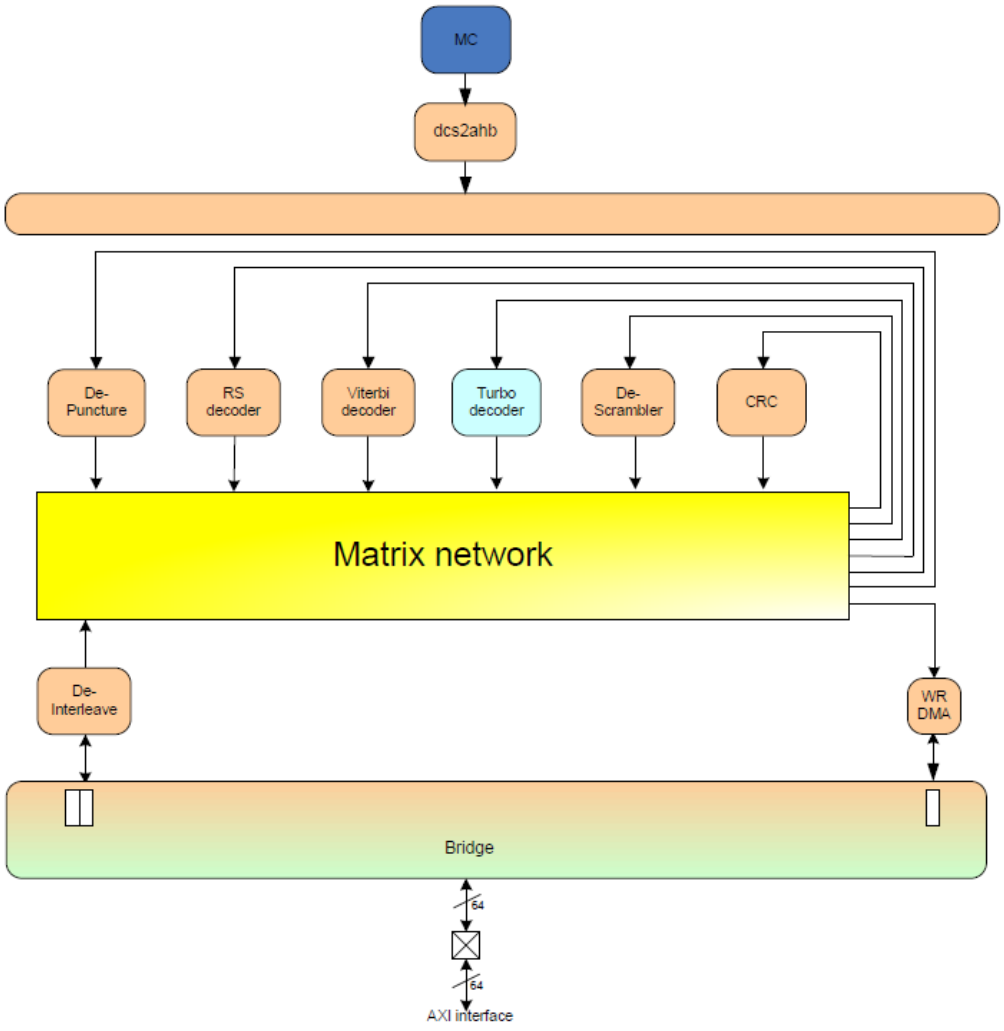


Figure 3.1: *FLORA architecture used in [19].*

Chapter 4

Clustering Definition

For obtaining optimal configurations for FLORA, a method is proposed in this thesis where a FLORA configuration is modeled as a clustered actor formed by clustering actors in an SDF graph. A definition of clustering is proposed in this chapter wherein we cluster sub-graph(s) in an input SDF graph into a single actor called as *composite* actor. The new graph containing the newly defined composite actor is also a pure SDF actor therefore it is atomic. Such a definition achieves two important objectives.

1. The atomicity of the composite actor models the non-preemptive nature of the FLORA configuration.

2. The hierarchical composition separates clustered sub-graph(s) from the parent graph containing composite actors. This allows separate analysis of sub-graph to be clustered from analysis of the parent graph. For instance it allows us to model self-timed schedule of actors mapped to FLORA hardware blocks. Whereas the clustered graph is scheduled with static-assignment (round-robin modeling) scheduling strategy.

In following section we describe our concept of clustering for SDF graphs. In section 4.2 we describe the clustering function in detail. In section 4.4 we describe that arbitrary clusters can result in deadlock. We immediately present a condition to prevent such clusterings. In section 4.5 we give two theorems with their proofs which guarantee consistency and deadlock freedom for clustered graphs obtained by the clustering function explained in section 4.3. In section 4.6 we elaborate the effects of clustering on throughput, buffers and scheduling load.

4.1 Clustering

In our concept of clustering, sub-graph(s) in an input SDF graph are clustered into a single actor called as *composite* actor. This means that the *composite* actor represents an atomic firing of k iterations of sub-graph(s). For a formal and generic definition of clustering we define a clustering function which accepts input and produces output as follows:

Input

1. $G(V, E, \delta, \varphi, \pi, \mu)$

2. $V_s \subset V$

3. $k \in \mathbb{N}$ where $\mathbb{N} = \{1, 2, 3 \ldots\}$

Output

1. $G'(V', E', \delta, \varphi', \pi', \mu')$

G is an input SDF graph. Definition of SDF graph is explained in chapter 2. $V_s \subset V$ is set of nodes to be clustered. From V_s sub-graph of sub-graph**s** are formed which are clustered. k is the number of sub-graph iterations that are clustered. G' is the clustered SDF graph with new set of nodes, edges, response times, production rates and consumption rates.

$V_s \subset V$ as an input parameter of clustering function can be any possible subset of V. In the clustering function from V_s we form a connected sub-graph. A sub-graph is created by removing all nodes in the original SDF graph which do not belong to V_s and by removing all edges whose source or sink node is not a node from V_s. However V_s may also consist of disconnected sub-graphs. For example in figure 4.2 $V_s = \{b, c\}$ forms only one connected sub-graph shown in figure 4.4. On the other hand for SDF graph shown in figure 4.1 for $V_s = \{b, c, e\}$ we get two maximal (largest possible) sub-graphs, a sub-graph formed by $\{b, c\}$ which is same as shown in figure 4.4 and the other sub-graph consists of a single actor e with its self-edge.

Clustering disjoint sub-graphs without assigning appropriate token rates to the composite actor may lead to inconsistent SDF graph. Therefore we define two clustering functions. The first clustering function Φ_c accepts V_s which only contains one **connected** sub-graph and second clustering function Φ_g which accepts any V_s which may contain disjoint sub-graphs as well. The main difference between Φ_g and Φ_c is that the token rates in Φ_g are constrained to guarantee consistency for the resulting clustered SDF graph irrespective of disjoint sub-graphs.

Φ_c and Φ_g are explained in detail in the following sections.

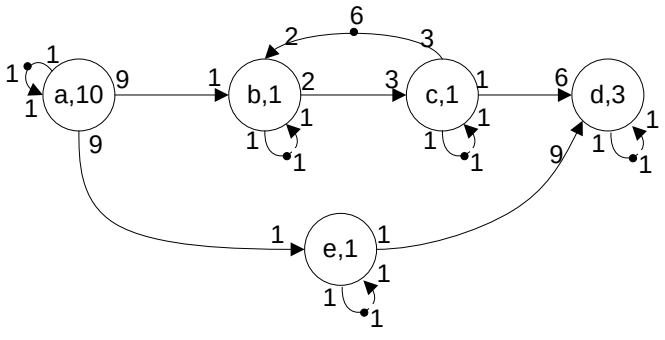


Figure 4.1: *An example SDF graph with branches.*

4.2 Clustering Function (Φ_c) for a single sub-graph

This clustering function Φ_c has input and output as explained in section 4.1. The important requirement for Φ_c clustering function is that $V_s \subset V$ consists of a single connected sub-graph. Clustering function is explained here informally. The formal definition is presented in definition 6.

The output of clustering function Φ_c is a clustered SDF graph $G'(V', E', \delta, \varphi', \pi', \mu')$. Thus clustering produces a new SDF graph. To define clustering function it is required to define every component of the new SDF graph namely – set of new nodes (V'), set of new edges (E'). set of new response times (φ'), set of new production rates (π') and set of new consumption rates (μ'). The set of initial tokens δ for edges are kept same. We elaborate each component in the following sections.

For illustration we consider an example SDF graph shown in figure 4.2. Consider figure 4.2 shows the input SDF graph $G(V, E, \delta, \varphi, \pi, \mu)$. The set of nodes to be clustered are $V_s = \{b, c\}$. We illustrate the effect of different k in following sections.

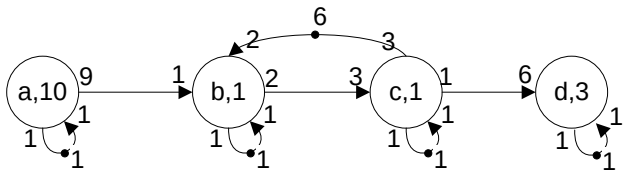


Figure 4.2: *An example SDF graph for illustrating clustering.*

New set of Nodes and Edges

The set of nodes to be clustered are replaced by a new node. The new node is called as *composite actor* v_c. Therefore the new set of nodes is obtained by removing all nodes in V_s and adding a new node $\{v_c\}$. This is formally defined as $V' = V \setminus V_s \cup \{v_c\}$. For our example in figure 4.2 the new set of edges are $V' = \{a, b, c, d\} \setminus \{b, c\} \cup \{v_c\} = \{a, v_c, d\}$.

On similar lines the set of new edges E' is defined by removing set of edges E_s from original set of edges E and adding new set of edges E_c. Therefore $E' = E \setminus E_s \cup E_c$. E_s is the set of edges in E which have either source or sink in V_s. For example in figure 4.2 E_s has all the edges except the self-edges of a and d. E_c is the set of constructed edges to connect v_c to remaining graph. It is constructed using only the nodes in V'. E_c consists of outgoing edges from v_c and incoming edges to v_c. Outgoing edge – for every edge in E where the source is a node from V_s and sink is a node $v \in V'$, an edge is created in E_c with source as v_c and sink as $v \in V'$. Incoming edge – for every edge in E where the source is a node from $v \in V'$ and sink is a node from V_s, an edge is created in E_c with source as v and sink as v_c. v_c has input and output ports which are created for every incoming and outgoing edges in E_c respectively. Additionally

a self-edge is added for v_c in E_c with one initial token to limit one firing of v_c at a time. The resulting graph is shown in figure 4.3.

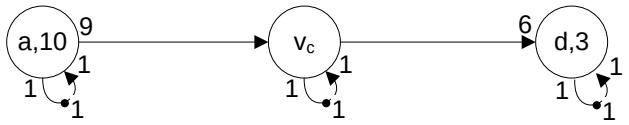


Figure 4.3: *Resulting SDF graph after defining new set of nodes and edges.*

Token production and consumption rates and the response times of actor v_c are missing in figure 4.3. They are explained in following section.

New Production Rates(π') and Consumption Rates(μ')

For every newly constructed edge in E_c the production rate and the consumption rate have to be defined. For the remaining edges in E' the production and consumption rates remain the same. The new production and consumption rates depend on the input parameter k. k is the number of sub-graph iterations we wish to cluster.

Composite actor v_c represents an atomic firing of k iterations of the sub-graph formed by actors in V_s. Therefore v_c consumes on each of its input ports as many tokens as are required to fire k iterations of sub-graph formed by V_s and produces tokens on each of its output port as many tokens as produced by k iterations of sub-graph formed by V_s. Therefore to compute new token rates a sub-graph is created from set of nodes to be clustered V_s. A sub-graph is created by removing all nodes in the original SDF graph which do not belong to V_s and by removing all edges whose source or sink node is not a node from V_s. The sub-graph formed by $V_s = \{b, c\}$ is shown in figure 4.4.

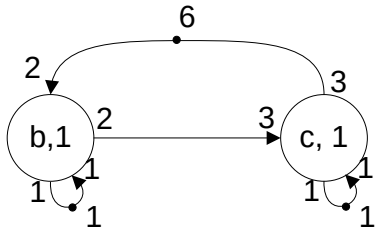


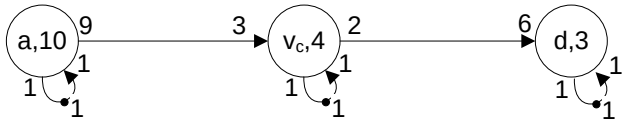
Figure 4.4: *Sub-graph of SDF graph shown in figure 4.2 formed by set of nodes* $\{b,c\}$.

We call the sub-graph shown in figure 4.4 G_{SG}. The repetition vector for G_{SG} is $\{\gamma_{G_{SG}}(b) = 3, \gamma_{G_{SG}}(c) = 2\}$. Therefore the minimum token consumption rate for newly constructed incoming edge $e' = (a, v_c)$ for $k = 1$ (minimum value of k) is $\mu'(e') = \gamma_{G_{SG}}(b) \times \mu(e) = 3 \times 1 = 3$ where $e = (a, b)$ is the corresponding

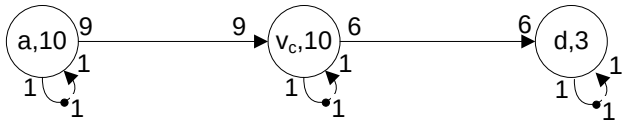
edge for e' in E. The production rate for newly created edge $e'' = (v_c, d)$ is $\pi'(e'') = \gamma_{G_{SG}}(c) \times \pi(e) = 2 \times 1 = 2$ where $e = (c, d)$ is the corresponding edge for e'' in E.

In this way the clustered SDF graph for $k = 1$ is shown in figure 4.5(a) and for $k = 3$ the clustered SDF graph is shown in figure 4.5(b).

Therefore for any value of $k \in \mathbb{N}$ the new consumption rate for all newly created incoming edges $e' \in E_c$ is $\mu'(e') = k \times \gamma_{G_{SG}}(v) \times \mu(e)$ where $v \in V_s$ is the corresponding sink node in V_s and e is the corresponding edge in E. The new production rate for all newly created outgoing edges $e' \in E_c$ is $\pi'(e') = k \times \gamma_{G_{SG}}(v) \times \pi(e)$ where $v \in V_s$ is the corresponding source node in V_s and e is the corresponding edge in E.



(a)



(b)

Figure 4.5: *(a) Resulting clustered SDF graph for input SDF graph shown in figure 4.2 for $V_s = \{b, c\}$, $k = 1$.(b) Resulting clustered SDF graph for input SDF graph shown in figure 4.2 for $V_s = \{b, c\}$, $k = 3$.*

New Response Times(φ')

The response times of all actors in G' remain same except for actors in V_s that is $\varphi'(v) = \varphi(v)$ where $v \in V \setminus V_s$. Actors in V_s will be removed and they will be replaced by new composite actor v_c. Therefore only the response time of composite actor v_c has to be defined.

The response time of actor v_c depends on the input parameter k. Since v_c represents an atomic firing of k iterations of sub-graph formed by V_s, the response time of composite actor v_c is the amount of time spent in k iterations of sub-graph formed by V_s.

To compute the amount of time spent for a number of iterations of a sub-graph, we need to decide the schedule for the SDF graph to be clustered. FLORA blocks are data-driven, that is the blocks fire as soon as data is available to them and there is enough space in the output FIFO. Therefore we use *self-timed* schedule for the SDF graph to be clustered. In self-timed schedule,

actors fire as soon as they can. To model the back-pressure due to finite output FIFOs the input SDF graph must be modeled with back edges with appropriate initial tokens.

Thus the response time of the composite actor depends on the scheduling strategy we choose for the SDF sub-graph to be clustered. We present a formulation for the response time of composite actor for *self-timed* schedule.

SDF actors in a self-timed schedule are pipelined [13, 14]. It is proven in [5] that for every consistent and strongly connected SDF graph in a self-timed schedule there is a transient phase followed by a periodic phase.

For every firing of composite actor, the sub-graph it represents, is fired from its initial state. Therefore the response time of the composite actor is the time the sub-graph spends in transient phase + the time sub-graph spends in k periodic iterations. Inspired by latency-rate server modeling for dataflow [21] we say that the time spent in the transient phase of a graph is modeled by θ and the time spent in periodic phase is modeled by ρ. This is illustrated in figure 4.6.

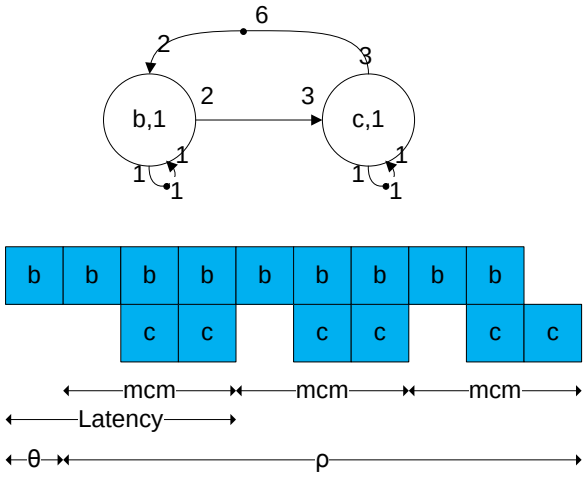


Figure 4.6: *Self-timed schedule for three iterations of sub-graph.*

We use function defined in [6] to compute latency between SDF nodes in a self-timed schedule. $latency(G, v_i, v_j)$ gives latency between nodes v_i and v_j in the SDF graph G. Using this function, for the sub-graph obtained from the original SDF graph we find the maximum latency which is the time spent to cover all firings of actors in the sub-graph as required by the repetition vector of the sub-graph. In figure 4.6, the maximum latency is 4 which covers 3 firings of b and 2 firings of c. For brevity we refer to this maximum latency as $latency(G)$ where G is the sub-graph obtained from V_s. Similarly $mcm(G)$ gives the Maximum Cycle Mean of an SDF graph for a self-timed schedule. Then the

latency- server variables θ and ρ for the sub-graph G_{SG} can be found as follows.

$$\theta = latency(G_{SG}) - mcm(G_{SG}) \tag{4.1}$$
$$\rho = mcm(G_{SG}) \times k \tag{4.2}$$

We define a new function τ to compute the response time of an SDF graph for self-timed schedule in definition 5.

Definition 5. *(Response Time of a Graph) Response Time of a Graph* is a function $\tau(G, k) = \theta + \rho$, where G is an SDF graph, $k \in \mathbb{N}$ is the number of iterations of SDF graph G, $\theta = latency(G) - mcm(G)$ and $\rho = mcm(G) \times k$. $latency(G)$ is the time spent for all firings of all actors in G as required by the repetition vector $\gamma(G)$.

Therefore the response time of composite actor is $\varphi'(v_c) = \tau(G_{SG}, k)$ where G_{SG} is the sub-graph formed with set of nodes V_s and k is the number of iterations of the sub-graph SDF graph G_{SG} that we specify.

For illustration,for SDF graph G shown in figure 4.2 clustered for $V_s = \{b, c\}$, $k = 1$, the response time of v_c is

$$\begin{aligned} latency(G) &= 4 \\ mcm(G) &= 3 \\ \theta &= latency(G) - mcm(G) = 1 \\ \rho &= k \times mcm(G) = 1 \times 3 = 3 \\ \varphi'(v_c) &= \theta + \rho = 1 + 3 = 4 \end{aligned}$$

For $k = 3$ we have

$$\begin{aligned} \theta &= latency(G) - mcm(G) = 1 \\ \rho &= k \times mcm(G) = 3 \times 3 = 9 \\ \varphi'(v_c) &= \theta + \rho = 1 + 9 = 10 \end{aligned}$$

The resulting clustered SDF graphs for $k = 1$ and $k = 3$ are shown in figure 4.5(a) and 4.5(b) respectively.

Set of Initial Tokens

In an SDF graph with cycles initial tokens on edges are essential to initiate transition of the graph. Without sufficient initial tokens graph deadlocks. Initial tokens denote different meaning in different context. Initial tokens placed on the back edges from consumer to producer denote capacity of buffer between producer and consumer. We have freedom to change amount of initial tokens for such edges. However if the original SDF representation of an algorithm

has initial tokens on some edges, they denote amount of data tokens used in a loop. Usually such tokens have important meaning and changing the amount of initial tokens in such cases may give a different meaning to the original algorithm. Therefore we do not change the set of initial tokens when constructing a clustered graph.

It is possible to shift the initial tokens from an edge to a consecutive edge(s) with appropriate pre-processing before firing the SDF graph. For example consider an SDF graph shown in figure 4.7(a). If we consider that actor a is fired one time before initiating the SDF graph then the produced 4 tokens from actor a can be placed in on the edge *(a,b)*. The new SDF graph is shown in figure 4.7(b).

(a)

(b)

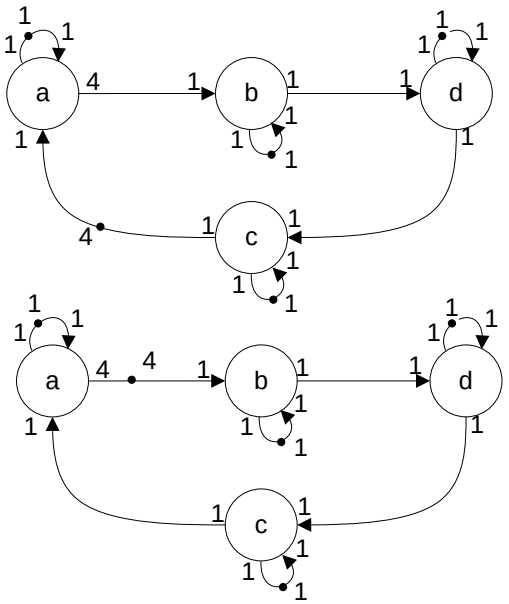


Figure 4.7: *(a) An example of SDF graph with a cycle. (b) A new SDF graph after pre-firing actor a and then placing the produced tokens on edge (a,b).*

Shifting of initial tokens enables some clustering options which were before not possible and disables some other clustering options. For example in figure 4.7(a) cluster *(c,a)* results in deadlock. Whereas *(c,a)* is a valid clustering option in figure 4.7(b). However cluster *(a,b)* now results in deadlock in figure 4.7(b) which was a deadlock-free cluster in figure 4.7(a). Shifting of initial tokens gives additional freedom in clustering. Therefore to restrict the clustering space we do not consider shifting initial tokens.

Formal Definition of Clustering Function Φ_c

Clustering function results in a new SDF graph. In the previous sections construction of each component of the clustered SDF graph was described. We summarize the description formally in definition 6.

Definition 6. *(Clustering Function (Φ_c) for single sub-graph)* Given an SDF graph $G(V, E, \delta, \varphi, \pi, \mu)$, a set of actors $V_s \subseteq V$ to be clustered and $k \in \mathbb{N}$, clustering function produces a new SDF graph $\Phi_c(G, V_s, k) = G'$ where $G'(V', E', \delta, \varphi', \pi', \mu')$ is the clustered SDF graph. G' is obtained by replacing V_s by a new actor v_c called as a composite actor. $V' = V \setminus V_s \cup \{v_c\}$. $E' = E \setminus E_s \cup E_c$ where $E_s = \{(p^s, q^d) \in E | (s \in V_s) \vee (d \in V_s)\}$ is the set of old edges to be removed and $E_c = \{(p^s, q^d) | (s \in V') \wedge (d \in V') \wedge (s = v_c \Rightarrow ((p^v, q^d) \in E \wedge (v \in V_s))) \wedge (d = v_c \Rightarrow ((p^s, q^v) \in E \wedge (v \in V_s)))\}$ is the set of new constructed edges. For all actors except v_c that is $\{v \in V' \setminus v_c\}$, the response times are $\varphi'(v) = \varphi(v)$. For the composite actor v_c, $\varphi'(v_c) = \tau(G_{SG}, k)$ where τ is a function which gives response time of an SDF graph defined in definition 5, G_{SG} is the connected sub-graph formed with set of nodes V_s. For all edges except the new constructed edges $\{e \in E' \setminus E_c\}$, the production and consumption rates are $\pi'(e) = \pi(e)$ and $\mu'(e) = \mu(e)$. For all new outgoing edges from v_c, $\{e' = (p^{v_c}, q^d) \in E_c\}$, $\pi'(e') = k \times \gamma_{G_{SG}} \times \pi(e)$ where $e = (p^v, q^d) \in E \wedge v \in V_s$. For all new incoming edges to v_c, $\{e' = (p^s, q^{v_c}) \in E_c\}$, $\mu'(e') = k \times \gamma_{G_{SG}} \times \mu(e)$ where $e = (p^s, q^v) \in E \wedge v \in V_s$.

4.3 Clustering Function (Φ_g) for sub-graphs

Clustering function we explained in section 4.2 accepts three input parameters, an input SDF graph $G(V, E, \delta, \varphi, \pi, \mu)$, a set of actors $V_s \subseteq V$ to be clustered and $k \in \mathbb{N}$ is the number of sub-graph iterations. As was explained in section 4.1, $V_s \subset V$ as an input parameter can be any subset of V. Therefore V_s may also consist of disjoint sub-graphs. For SDF graph shown in figure 4.1 for $V_s = \{b, c, e\}$ we get two maximal (largest possible) sub-graphs shown in figure 4.9.

For arbitrary $V_s \subset V$ and $k \in \mathbb{N}$ we may generate an inconsistent clustered SDF graph. For SDF graph shown in figure 4.1, for $k = 1$ and $V_s = \{b, c, e\}$ the resulting clustered SDF graph is shown in figure 4.8.

Therefore to guarantee consistent clustered graphs we need to bound V_s and/or k. In our new definition of clustering function we guarantee consistency by restricting k. Thus all possible subset of actors V_s can be tried and therefore all possible configurations of FLORA can be tried. Instead of keeping k variable (by accepting it as input parameter) for each maximal sub-graph formed by V_s we choose value of k to be equal to as many iterations of sub-graph as possible with firings of actor in sub-graph specified by the repetition vector of the input graph. Therefore for input SDF graph $G(V, E, \delta, \varphi, \pi, \mu)$ and any subset of actors $V_s \subset V$ to be clustered we compute k for each sub-graph as follows.

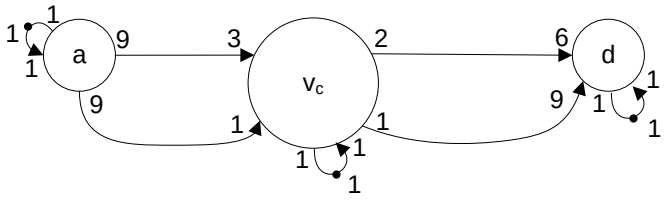


Figure 4.8: *Resulting inconsistent clustered SDF graph for input SDF graph shown in figure 4.1 for* $V_s = \{b, c, e\}$ *and* $k = 1$.

Let SG be the set of all maximal connected components that can be formed from nodes in V_s. Then for each sub-graph, $g \in SG$, $k = \frac{\gamma(v)}{\gamma_g(v)}$ where γ is the repetition vector of input SDF graph, γ_g is the repetition vector of sub-graph g and $v \in V_s$.

For example, for input SDF graph shown in figure 4.1 to cluster $V_s = \{b, c, e\}$ we have two sub-graphs formed from V_s shown in figure 4.9.

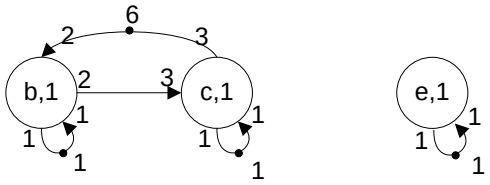


Figure 4.9: *Resulting sub-graphs for input SDF graph shown in figure 4.1 for* $V_s = \{b, c, e\}$

For input SDF graph shown in figure 4.1, repetition vector is $\gamma = \{\gamma(a) = 1, \gamma(b) = 9, \gamma(c) = 6, \gamma(d) = 1, \gamma(e) = 9\}$. For sub-graph G_1 formed by $\{b, c\}$, $\gamma_{G_1} = \{\gamma(b) = 3, \gamma(c) = 2\}$. Therefore for sub-graph G_1, $k = \frac{9}{3} = 3$. Using this k, the new rates for incoming edge to actor b and the outgoing edge from actor c can be computed. For sub-graph G_2 formed by $\{e\}$, $\gamma_{G_2} = \{\gamma(e) = 1\}$. Therefore for sub-graph G_2, $k = \frac{9}{1} = 9$.

For computing the response time of clustered actor v_c, the maximum of the response time of all sub-graphs is taken. That is $\varphi'(v_c) = \max_{g \in SG} \tau(g, K(g))$ where SG is the set of maximal sub-graphs formed by V_s and $K(g) = \frac{\gamma(v)}{\gamma_g(v)}$, $v \in V_s$ gives the number of iterations for each sub-graph $g \in SG$.

Thus for SDF graph shown in figure 4.1, the resulting clustered SDF graph is shown in figure 4.10. The new response time of actor v_c is $max(9, 10) = 10$.

We summarize the definition of this general clustering function Φ_g in definition 7.

Definition 7. *(General Clustering Function (*Φ_g*) for sub-graphs)* Given an

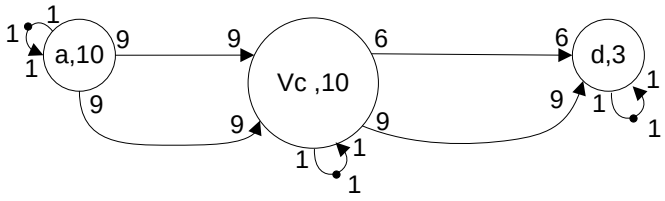


Figure 4.10: *Resulting consistent SDF graph for input SDF graph shown in 4.1 for* $V_s = \{b, c, e\}$

SDF graph $G(V, E, \delta, \varphi, \pi, \mu)$, a set of actors $V_s \subseteq V$ to be clustered, general clustering function produces a new SDF graph $\Phi_g(G, V_s) = G'$ where $G'(V', E', \delta, \varphi', \pi', \mu')$ is the clustered SDF graph. G' is obtained by replacing V_s by a new actor v_c called as a composite actor. $V' = V \backslash V_s \cup \{v_c\}$. $E' = E \backslash E_s \cup E_c$ where $E_s = \{(p^s, q^d) \in E | (s \in V_s) \vee (d \in V_s)\}$ is the set of old edges to be removed and $E_c = \{(p^s, q^d) | (s \in V') \wedge (d \in V') \wedge (s = v_c \Rightarrow ((p^v, q^d) \in E \wedge (v \in V_s))) \wedge (d = v_c \Rightarrow ((p^s, q^v) \in E \wedge (v \in V_s)))\}$ is the set of new constructed edges. For all actors except v_c that is $\{v \in V' \setminus v_c\}$, the response times are $\varphi'(v) = \varphi(v)$. For the composite actor v_c, $\varphi'(v_c) = \max_{g \in SG} \tau(g, K(g))$ where SG is the set of maximal sub-graphs formed by V_s and $K(g) = \frac{\gamma(v)}{\gamma_g(v)}$, $v \in V_s$ gives the number of iterations for each sub-graph $g \in SG$ and τ is a function which gives response time of an SDF graph defined in definition 5. For all edges except the new constructed edges $\{e \in E' \setminus E_c\}$, the production and consumption rates are $\pi'(e) = \pi(e)$ and $\mu'(e) = \mu(e)$. For all new outgoing edges from v_c, $\{e' = (p^{v_c}, q^d) \in E_c\}$, $\pi'(e') = K(g) \times \gamma_g \times \pi(e)$ where $e = (p^v, q^d) \in E \wedge v \in V_s$. For all new incoming edges to v_c, $\{e' = (p^s, q^{v_c}) \in E_c\}$, $\mu'(e') = K(g) \times \gamma_g \times \mu(e)$ where $e = (p^s, q^v) \in E \wedge v \in V_s$.

4.4 Deadlock - Free Clusters

Clustering actors in an SDF graph may result in an SDF graph which deadlocks. An SDF graph does not deadlock if its equivalent *Precedence Graph* is *acyclic* [12]. Conversely an SDF graph which does deadlock must have at least one cycle in its equivalent *Precedence Graph*. Equivalent precedence graph for an SDF graph is obtained by removing all the edges in the equivalent HSDF graph which have at least one initial token [13].

Consider an SDFgraph shown in figure 4.11(a). Clustering $\{b, c\}$ results in an SDF graph shown in figure 4.11(b). This SDF graph deadlocks because there is a cycle between actor *bc* and *d* which is seen in its equivalent precedence graph in figure 4.11(c) and also in 4.11(b).

To avoid clustering options for an SDF graph $G(V, E, \delta, \varphi, \pi, \mu)$ that lead to deadlock it is necessary to check if clustering set of actors $V_s \subset V$ in a single actor v_c creates a cycle in the precedence graph of clustered SDF graph. For

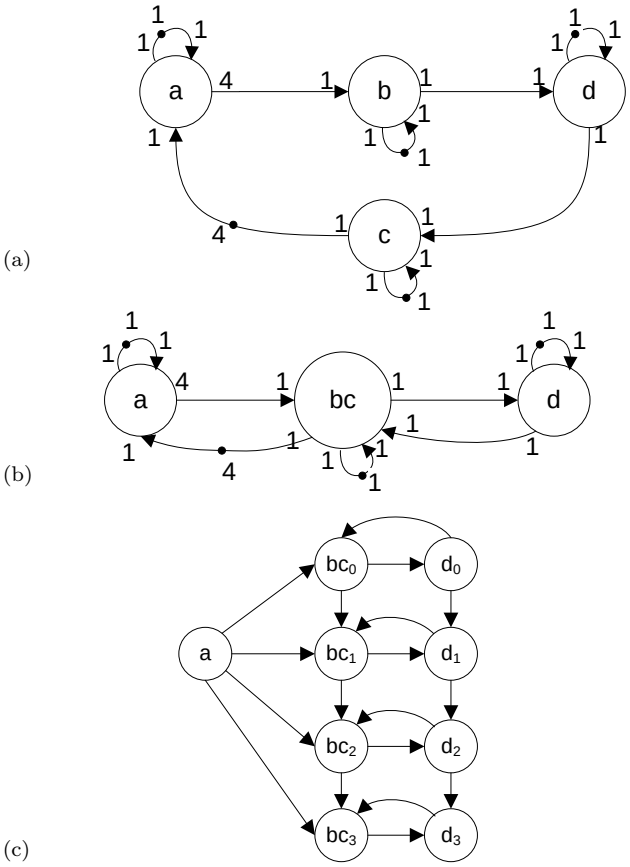


Figure 4.11: *(a) An example of SDF graph. (b) Resulting SDF graph after clustering $\{b, c\}$ which deadlocks.(c) Equivalent Precedence Graph for clustered SDF graph in (b).*

different set of actors to be clustered $V_s \subset V$ we get different clustering options. Therefore for each given set $V_s \subset V$ we need to check for deadlock.

In the input SDF graph if there is a path from any node in V_s to any other node in V_s via a node outside V_s then a cycle will be created in the resulting clustered SDF graph with the clustered actor v_c. Therefore to check for a clustering option $V_s \subset V$ that leads to deadlock we state a condition on the input SDF graph in definition 8.

Definition 8. *(Condition for checking deadlock)* Given an SDF graph $G(V, E, \delta, \varphi, \pi, \mu)$ and a set of actors $V_s \subset V$ to be clustered, the resulting SDF graph is deadlock free if no path $(v_i, \ldots, v_k, \ldots, v_j)$ exists in the equivalent precedence graph of G where $(v_i \in V_{pgs}) \wedge (v_j \in V_{pgs}) \wedge (v_k \notin V_{pgs})$ where V_{pgs} is the corresponding set of actors for V_s in the equivalent precedence graph of G.

For illustration consider SDF graph shown in figure 4.11(a). Its equivalent precedence graph is shown in figure 4.12. For the case when we wish to cluster $V_s = \{b, c\}$ we see in figure 4.12 that there are multiple paths from b_i to c_i ($i = \{0, 1, 2, 3\}$) via nodes d_i and $d_i \notin V_{pgs}$. Therefore clustering V_s will result in deadlock. This holds as illustrated in figure 4.11.

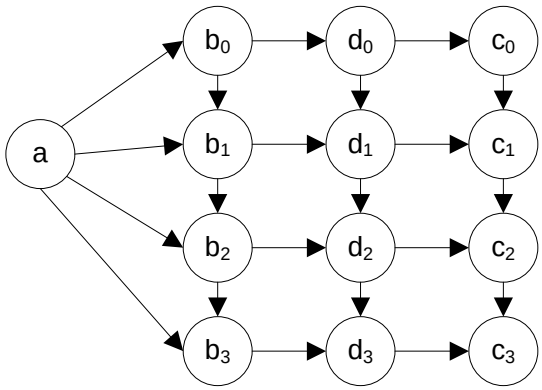


Figure 4.12: *Equivalent precedence graph for SDF graph shown in figure 4.11(a).*

4.5 Properties of Clustering Function

It is important that the clustering function defined in definition 7 results in consistent and deadlock-free SDF graphs so that the clustered graphs are practically useful. In this section we present theorems guaranteeing consistency and deadlock freedom along with their proofs.

4.5.1 Preservation of Consistency

Theorem 1. Given an SDF graph $G(V, E, \delta, \varphi, \pi, \mu)$ and a set of actors $V_s \subset V$ to be clustered, if G is consistent than the clustered graph $\Phi_g(G, V_s) = G'$ is also consistent.

Proof. If $G(V, E, \delta, \varphi, \pi, \mu)$ is consistent than the repetition vector γ of G has non-zero elements. That is $\forall v \in V : \gamma(v) > 0$. To prove that $G'(V', E', \delta, \varphi', \pi', \mu')$ is also consistent we must show $\forall v \in V' : \gamma'(v) > 0$.

We shall derive γ' for G' using steps as defined by the clustering function Φ_g in definition 7 and prove that $\forall v \in V' : \gamma'(v) > 0$. In the proof, to derive γ' we shall be required to derive $\gamma'(v_c)$. To find $\gamma'(v_c)$ we can evaluate either an incoming edge to v_c or an outgoing edge from v_c. We use an incoming edge in the proof. γ_{SG} in the proof indicates the repetition vector of sub-graph formed

by $V_s \subset V$. For simplicity we consider an edge to be a tuple $e = (s, d)$ where s is the source actor and d is the destination actor.

1	$\forall v \in V : \gamma(v) > 0$	Given
2	$\gamma(V) = \{\gamma(v) \mid v \in V\}$	Given
3	$V' = V \setminus V_s \cup \{v_c\}$	Clustering Function
4	$\forall e = (s, v_c) \wedge (s \in V' \setminus \{v_c\}) \wedge (d \in V_s) : \pi'(e) = \pi(e)$	Clustering Function
5	$\forall e = (s, v_c) \wedge (s \in V' \setminus \{v_c\}) \wedge (d \in V_s) : \mu'(e) = K(d) \cdot \mu(e) \cdot \gamma_{SG}(d)$	Clustering Function
6	$K(v) = \{\frac{\gamma(v)}{\gamma_{SG}(v)} \mid v \in V_s\}$	Clustering Function
7	$\gamma'(V') = \{\gamma(v) \mid v \in V \setminus V_s \cup \{v_c\}\}$	Using 2,3
8	$\gamma'(V') = \{\gamma(v) \mid v \in V \setminus V_s\} \cup \{\gamma(v) \mid v \in \{v_c\}\}$	Property of $\cup$
9	$\gamma'(V') = \{\gamma(v) \mid v \in V\} \setminus \{\gamma(v) \mid v \in V_s\} \cup \{\gamma(v) \mid v \in \{v_c\}\}$	Property of $\setminus$
10	$\gamma'(V') = \{\gamma(v) \mid v \in V\} \setminus \{\gamma(v) \mid v \in V_s\} \cup \gamma(v_c)$	Single Element
11	$\forall e = (s, d) \wedge (s \in V) \wedge (d \in V) : \gamma(s) \cdot \pi(e) = \gamma(d) \cdot \mu(e)$	Definition of γ for G
12	$\quad \forall e = (s, v_c) \wedge (s \in V' \setminus \{v_c\}) \wedge (v_c \in V')$	Incoming edge to v_c
13	$\quad \gamma'(s) \cdot \pi'(e) = \gamma'(v_c) \cdot \mu'(e)$	Definition of γ for G'
14	$\quad \gamma(s) \cdot \pi'(e) = \gamma'(v_c) \cdot \mu'(e)$	Using 10,13
15	$\quad \gamma(s) \cdot \pi(e) = \gamma'(v_c) \cdot \mu'(e)$	Using 4,14
16	$\quad \gamma(s) \cdot \pi(e) = \gamma'(v_c) \cdot K(d) \cdot \mu(e) \cdot \gamma_{SG}(d)$	Using 5,15
17	$\quad \gamma(s) \cdot \pi(e) = \gamma'(v_c) \cdot \frac{\gamma(d)}{\gamma_{SG}(d)} \cdot \mu(e) \cdot \gamma_{SG}(d)$	Using 6,16
18	$\quad \gamma(s) \cdot \pi(e) = \gamma'(v_c) \cdot \gamma(d) \cdot \mu(e)$	
19	$\quad \gamma(s) \cdot \pi(e) = \gamma'(v_c) \cdot \gamma(s) \cdot \pi(e)$	Using 11,18
20	$\quad \gamma'(v_c) = 1$	
21	$\forall v \in V' : \gamma'(v) > 0$	Using 1,2,10,20

□

4.5.2 Deadlock Freedom

Theorem 2. Given an SDF graph $G(V, E, \delta, \varphi, \pi, \mu)$ and a set of actors $V_s \subset V$ to be clustered the resulting clustered graph $\Phi_g(G, V_s) = G'$ is deadlock-free if G is deadlock-free and the condition specified in definition 8 is satisfied.

Proof. The clustered SDF graph G' will deadlock if there is a cycle in its equivalent precedence graph. Such a cycle was created either after clustering process or it existed in the input SDF graph G before clustering.

If it existed before clustering, the condition that G should be deadlock-free is dissatisfied and hence theorem holds.

For the case when a cycle was created after clustering process, we need to show that if condition in definition 8 is satisfied then no cycle is created in the equivalent precedence graph of clustered graph G'.

Clustering set of actors $V_s \subset V$ results in a single actor v_c as defined in definition 7. Therefore all edges going out from $v \in V_s$ in G originate from v_c in G' and all edges coming to $v \in V_s$ terminate on v_c in G'. Therefore for the context of paths we can say there is an equivalence between actors in G and G' and also in their equivalent precedence graphs. That is $\forall v \in V_{pgs} : v_c \equiv v$ where V_{pgs} is the corresponding set of actors for V_s in the equivalent precedence graph. On the other hand set of paths are transformed in a new modified set of paths with $\overleftrightarrow{\Phi}_g(P) \vDash P'$, where P is the set of paths in precedence graph of G and P' is the set of paths in equivalent precedence graph of G'.

Let condition defined in 8 hold.

1	$\nexists p = (v_i, ...v_k, ..v_j) \wedge (p \in P) : (v_i \in V_{pgs}) \wedge (v_j \in V_{pgs}) \wedge (v_k \notin V_{pgs})$	Assumption
2	$\forall v \in V_{pgs} : v_c \equiv v$	Clustering Equivalence
3	$\overleftrightarrow{\Phi}_g(P) \vDash P'$	Clustering Transformation
4	$\nexists p = (v_c, ...v_k, ..v_c) \wedge (p \in P') : (v_c \in V_{pgs}) \wedge (v_c \in V_{pgs}) \wedge (v_k \notin V_{pgs})$	Using 2,3 for v_i, v_j
5	$\nexists p = (v_c, ...v_k, ..v_c) \wedge (p \in P') : (v_c \in V_{pgs}) \wedge (v_k \notin V_{pgs})$	$a \wedge a \equiv a$
6	$\nexists p = (v_c, ...v_k, ..v_c) \wedge (p \in P') : (v_c \in V_{pgs}) \wedge (v_k \notin V_s)$	$V_{pgs} \equiv V_s$

The derived statement above in 5 says, in the set of all paths in the equivalent precedence graph of clustered SDF graph G' no path exists from v_c to v_c via a node which is not from V_s (that is any node in V' other than v_c). Which means after clustering transformation no cycle was created in set of all paths in equivalent precedence graph of G' containing v_c. Therefore G' is deadlock free. □

4.6 Effects of Clustering

Clustering of SDF actors in a single SDF actor affects obtainable throughput, buffer requirement and the scheduling load to schedule the SDF graph. In this section we explain the significance of clustering. We detail the effects of clustering on throughput, scheduling load and buffers.

Clustering and throughput

Clustering actors into a single actor affects throughput. For the case where we ignore the time spent in reading/writing FIFO in an SDF graph, clustering may only decrease the throughput. This happens because we constrain the pipelining

possibilities in the clustered actor. This is illustrated in figure 4.13. Figure 4.13(a) shows two SDF graphs with their self-timed schedule shown below. Their throughput constraints are defined by the inverse of the time period of their source actors (So_1 and So_2). Actors A and C are mapped to resource P1 and actors B and D are mapped to resource P2. If we let the resources be shared via round-robin scheduler we get an execution shown in 4.13(b). We see that their throughput constraints are satisfied. Now we cluster the actors A and B and actors C and D. The token rates are adjusted to let as many firings of A and B take place as there are in the repetition vector ($\{\gamma(So_1) = 1, \gamma(A) = 2, \gamma(B) = 1\}$) and ($\{\gamma(So_2) = 1, \gamma(C) = 2, \gamma(D) = 3\}$). Since we use round-robin scheduler we sum the execution time of actors on shared resources. Therefore actors AB and CD have execution time 7. The clustered graph is shown in figure 4.13(c). We see that we either break the throughput or the SDF graph shall take unbounded memory for execution. Hence we have allowed boundless FIFO in figure 4.13(c). This tells us we should avoid clustering to allow maximum possible throughput.

On the other hand, if we consider that finite time is spent in reading/writing FIFOs, then clustering can improve the throughput. This is illustrated in figure 4.14. Figure 4.14(a) shows an example of SDF graph without read/write overhead. We can model read/write overheads in different ways such as introducing a RDMA and WDMA actors on every edge (this gives better approximation of the overhead). But for illustration we assume a constant read/write overhead for all actors. Say every actor takes 0.5 time units to read and write. We accommodate the read/write overhead in the firing duration of each actor. Thus we see in figure 4.14(b) we have added 1 to every actor (ignoring the source actor). The self-timed execution is also shown thereby. The dark-green region shows the time spent in reading/writing. We see that the bottle neck is execution of 4 Bs with *mcm* of 8. This breaks the throughput constraint. But when we cluster actors B and C in figure 4.14(c), the bottleneck reduces to execution of composite actor BC with firing duration of 7, thus throughput constraint is met.

Clustering and scheduling load

For every model of computation we can have a program or schedule. For example, for turing machines such as processors we have machine code stored in memory. For dataflow models we can have different types of schedules explained in table 3.1. If it is a static schedule we can store such a schedule. We can say such code or a schedule has a *load* associated with it. For stored programs in a memory, the code size can be inferred as the load. For a periodic schedule the length of the schedule for a period can be inferred as its load. For run-time schedules or fully-dynamic schedules the switching over-head can be inferred as a *scheduling load*. For FLORA as explained in section 3.1 such a *scheduling load* is relevant to us.

Since we use SDF as a model of computation, we wish to measure scheduling load for a SDF graph. There are numerous ways to measure scheduling load. A

simple measure is number of firings per unit time. We define this measure as follows.

Definition 9. *(Absolute Scheduling Load)* Absolute Scheduling Load of a SDF graph $G(V, E, \delta, \varphi, \pi, \mu)$ is defined as $ASL(G) = |\gamma|/mcm(G)$ where $|\gamma| = \sum_{v \in V} \gamma(v)$.

Clustering always decreases scheduling load. This is illustrated in figure 4.15. Figure 4.15(a) shows the original SDF graph with repetition vector $\gamma = \{1, 4, 2\}$ and mcm=4. Therefore Absolute Scheduling Load (ASL)=1.75. In figure 4.15(b) we have clustered 4 Bs and in figure 4.15(c) we have clustered 4 Bs and 2 Cs to form actor BC. We see that ASL reduces from 1.75 to 1 to 0.333.

Clustering and buffer requirement

Finite buffers are represented with back edges from consumer to producer actors with initial tokens placed on these back edges [14]. Buffer sizing affects throughput [14]. Clustering also affects the buffer requirement. This is so because clustered actors get new firing durations, new consumption rates and new production rates which means the waiting time of tokens in FIFO can change. But unlike scheduling load, clustering does not affect buffer requirement monotonously. In figure 4.15 we show buffers for maximum throughput. As we cluster 4 Bs in figure 4.15(b), buffer capacity required changes from 20 to 24. And in figure 4.15(c) buffer capacity required is again 20. Even though actors B and C are clustered, we say that actor BC has internally a buffer of size 4 which is also accounted for.

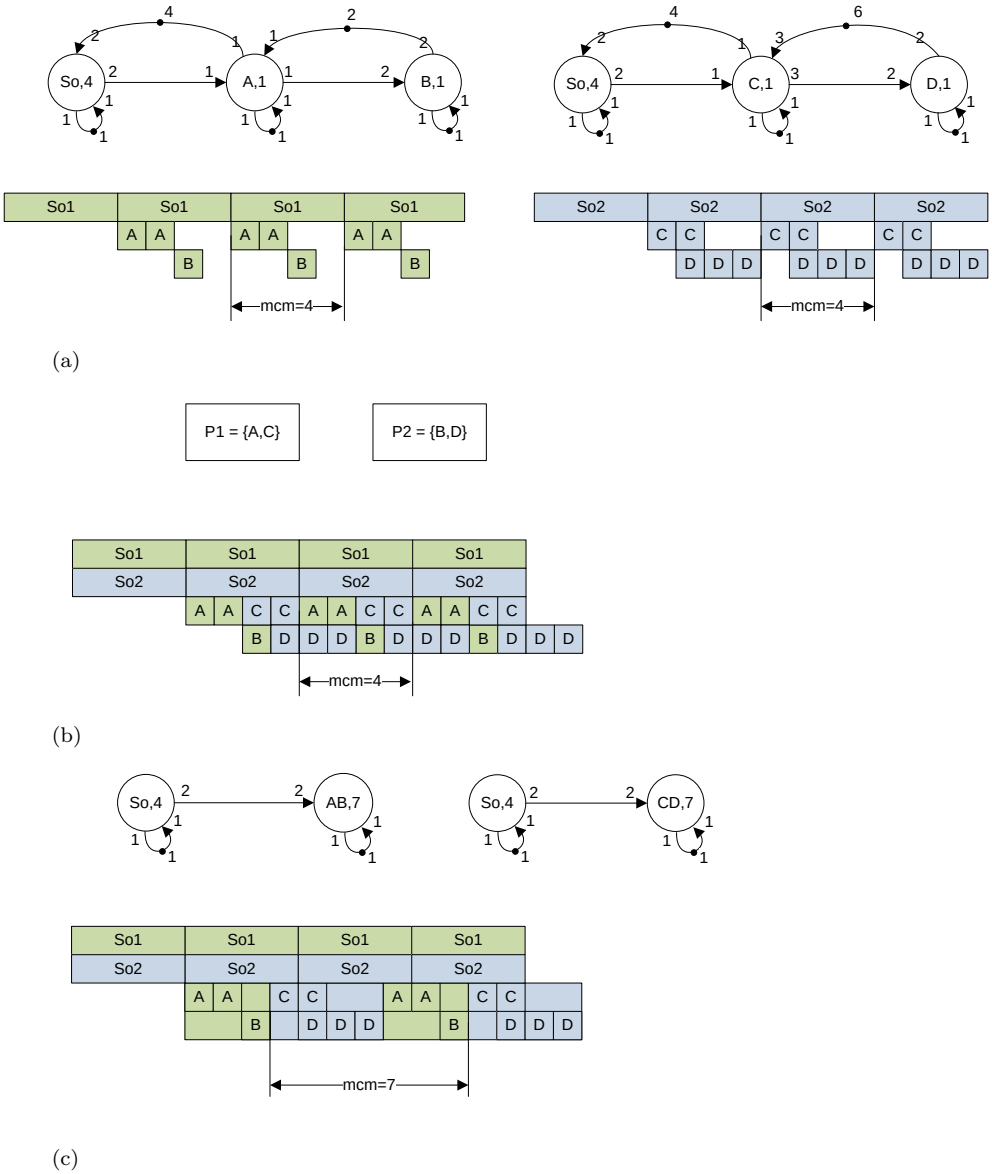
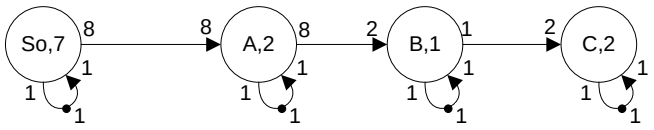
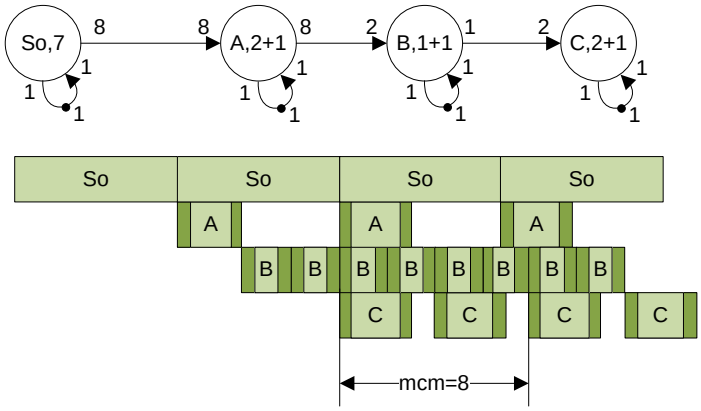


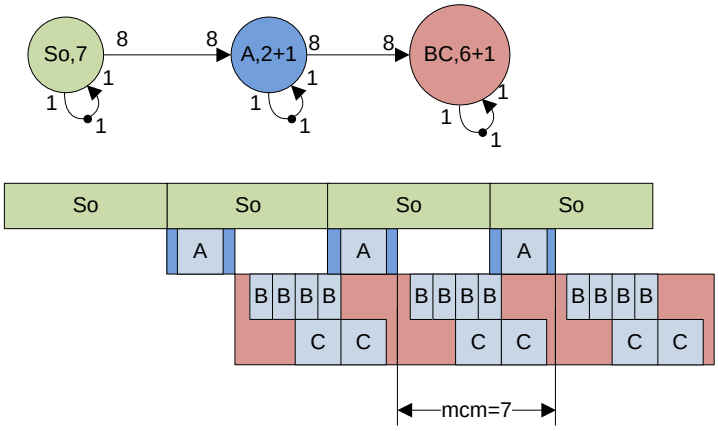
Figure 4.13: *Clustering may reduce throughput. (a) Two SDF graphs with their schedules shown below. Both have throughput constraint of 1/4. (b) A combined schedule on shared resources. A,C are mapped to resource P1 and B,D are mapped to resource P2. Throughput constraints of both SDF graphs are still satisfied. (c) A,B are clustered to form actor AB. C,D are clustered to form CD. Throughput constraint of both SDF graphs break.*



(a)

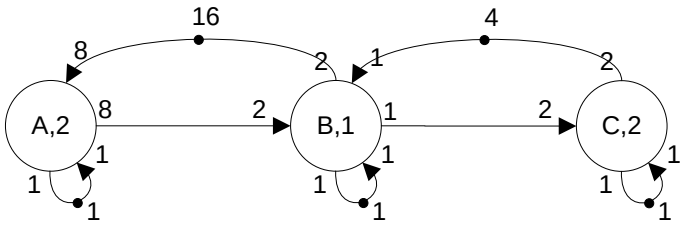


(b)

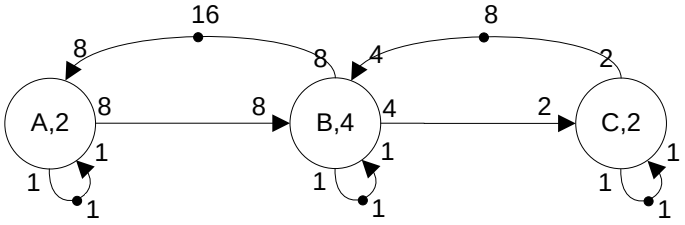


(c)

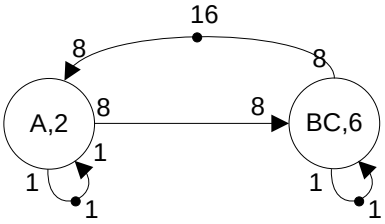
Figure 4.14: *Clustering can improve throughput when FIFOs have finite read/write overhead. (a) An example SDF graph without read/write overhead.(b) SDF graph with a constant read/write overhead of 0.5 time units to read and write. Throughput constraint is not met.(c) Actors B and C are clustered. Throughput constraint is met.*



(a)



(b)



(c)

Figure 4.15: *Clustering always reduces scheduling load. Clustering affects buffer requirement. (a) Absolute Scheduling Load(ASL)=*$\frac{1+4+2}{4} = 1.75$*. Buffer capacity required 16+4=20.(b) ASL=*$\frac{1+1+2}{4} = 1$*. Buffer capacity required 16+8=24.(c) ASL=*$\frac{1+1}{6} = 0.333$*. Buffer capacity required 16+4=20.*

Chapter 5

Clustering Algorithms

In the previous chapter we described our clustering function which given an input SDF graph and a set of nodes to be clustered produces a clustered SDF graph. Using this clustering function in this chapter we present an exact exhaustive search algorithm and a heuristic algorithm. In section 5.1 we describe algorithms for checking deadlock. In section 5.2 we describe the exact algorithm and the heuristic algorithm to find optimal clusterings for input SDF graphs. We describe the details of the implemented algorithm in section 5.3. In section 5.4 we describe experiments to evaluate performance of the clustering algorithms.

5.1 Algorithms For Checking Deadlock-free Clusters

In section 4.4 it was explained that given a set of nodes $V_s \subset V$ to be clustered in an SDF graph $G(V, E, \delta, \varphi, \pi, \mu)$ the resulting clustered graph may deadlock. To prevent such clustering options a condition was presented in definition 8. A naive algorithm implementing condition specified in definition 8 is presented in algorithm 1. The algorithms returns *valid* if clustering set of nodes $V_s \subset V$ does not lead to a deadlock. Otherwise *invalid* is returned.

Algorithm 1 checks for all paths between all nodes in V_{pgs} where V_{pgs} is the set of nodes corresponding to V_s in the equivalent precedence graph of the input SDF graph. The equivalent precedence graph for an SDF graph is obtained by removing edges in the equivalent HSDF graph which have at least one initial token [13]. Therefore the equivalent precedence graph may contain exponential number of nodes and edges and therefore exponential number of paths. For example consider SDF graph shown in figure 5.1(a). Its precedence graph is shown in figure 5.1(b). For checking clustering option $V_s = \{b, c\}$ there are 26 possible paths from node b_i to c_i, $i \in \{0, 1, 2, 3\}$ in the precedence graph shown in 5.1(b). Therefore algorithm 1 can have exponential complexity in size of repetition vector of input graph. For practical SDR applications the complexity of algorithm 1 is unacceptable. Therefore we propose an efficient algorithm as follows.

input : $G(V, E, \delta, \varphi, \pi, \mu), V_s \in V$
output: $Valid|Invalid$

Obtain Precedence Graph G_{pg} for G
V_{pgs} are the corresponding nodes for V_s in G_{pg}
for *all* $v_i, v_j \in V_{pgs}$ **do**
  for *all paths in* G_{pg} *between* $v_i, v_j \in V_{pgs}$ **do**
    if *path contains* $v_k \notin V_{pgs}$ **then**
      return *Invalid*
    end
  end
end
return *Valid*

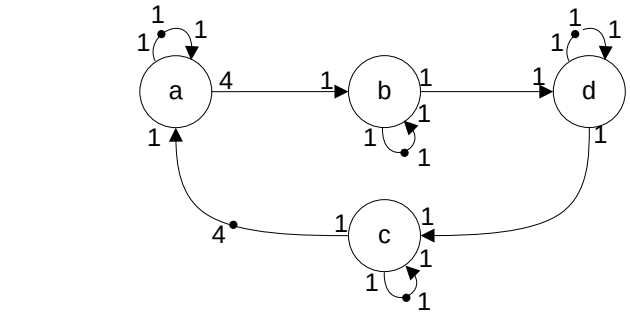
Algorithm 1: *Algorithm to check for deadlock-free clusters.*

input : $G(V, E, \delta, \varphi, \pi, \mu), V_s \in V$
output: $Valid|Invalid$

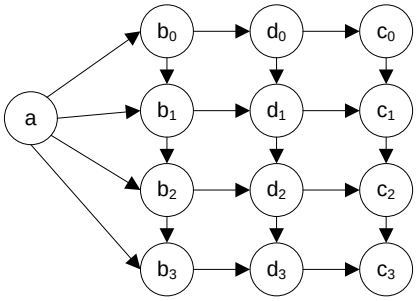
// Construct G_{SPG} as follows
Clone $G(V, E)$ into $G_{SPG}(V', E')$
for *each* $e = (s, d) \in E'$ **do**
  if $\delta(e) \geq \gamma(d) \cdot \mu(e)$ **then**
    remove (e)
  end
end
// Check for deadlock in G_{SPG} as follows
for *all* $v_i, v_j \in V_s$ **do**
  for *all paths in* G_{SPG} *between* $v_i, v_j \in V_s$ **do**
    if *path contains* $v_k \notin V_s$ **then**
      return *Invalid*
    end
  end
end
return *Valid*

Algorithm 2: *Efficient Algorithm to check for deadlock-free clusters.*

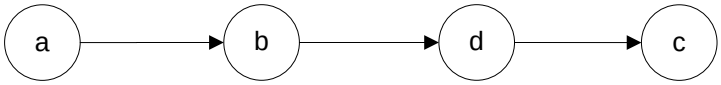
Paths in the equivalent HSDF are redundant as the equivalent HSDF graph contains replicated nodes and paths. Therefore we propose to construct a precedence graph directly from an SDF graph skipping the intermediate transformation to equivalent HSDF graph. We do so by removing edges from the input SDF graph which have initial tokens. However number of initial tokens are important. Removing an edge which has insufficient amount of tokens may validate a clustering option which will deadlock. Therefore we remove an edge $e = (s, d) \in E$ if it has as many tokens as required by the destination actor of the edge. The number of tokens required by destination actor is the product of the consumption rate of the destination actor times its repetition vector, that is



(a)



(b)



(c)

Figure 5.1: *(a) An example of SDF graph. (b) Equivalent Precedence Graph for SDF graph in (a). (c) A precedence graph constructed directly from SDF graph.*

$\delta(e) \geq \gamma(d) \cdot \mu(e)$. For example in figure 5.1(a) the precedence graph constructed directly from SDF graph is shown in figure 5.1(c). Note that only information of nodes and edges are required to check for deadlock. Therefore rates, response times are omitted while constructing graph in figure 5.1(c). Figure 5.1(c) has only one path between node b and c via node $d \notin V_s$ compared to 26 paths in figure 5.1(b). In this way we avoid searching exponential number of paths. The efficient algorithm is presented algorithm 2.

5.2 Clustering Algorithms

In chapter 4 definition for clustering function (Φ_g) was presented. Given an input SDF graph and a set of nodes to be clustered, clustering function Φ_g returns a clustered graph. In this section we present clustering algorithms for the problem statement mentioned in section 3.4. All the clustering algorithms accept a set of input SDF graphs and output corresponding set of optimal clustered SDF graphs using the clustering function Φ_g. A clustered graph is optimal if it satisfies the throughput and buffer constraints and minimizes absolute scheduling load (defined in definition 9).

5.2.1 Exhaustive Search Approach

Given a set of SDR applications optimal configurations for FLORA are computed at compile time. Therefore an exact clustering algorithm is desired. In exhaustive search approach all the possible clustering options are searched to find the exact solution. For the clustering function $\Phi_g(G, V_s) = G'$, $G(V, E, \delta, \varphi, \pi, \mu)$ is the input SDF graph, $V_s \subset V$ is the set of actors to be clustered and G' is the resulting clustered SDF graph. To try out all clustering options for G is to try all partitions of set of actors V. For example consider SDF graph in figure 5.1(a). The set of actors are $V = \{a, b, c, d\}$. The list of all partitions for set V is presented in table 5.1. The number of all possible non-empty partitions of a set is given by Bell Number or Sterling Numbers of the second kind [1]. To obtain list of all partitions, a recursive procedure is presented in algorithm 3.

1	[[a] [b] [c] [d]]
2	[[a,b] [c] [d]]
3	[[b] [a,c] [d]]
4	[[b] [c] [a,d]]
5	[[a] [b,c] [d]]
6	[[a,b,c] [d]]
7	[[b,c] [a,d]]
8	[[a] [c] [b,d]]
9	[[a,c] [b,d]]
10	[[c] [a,b,d]]
11	[[a] [b] [c,d]]
12	[[a,b] [c,d]]
13	[[b] [a,c,d]]
14	[[a] [b,c,d]]
15	[[a,b,c,d]]

Table 5.1: *List of all partitions of set* $V = \{a, b, c, d\}$

[1] `http://en.wikipedia.org/wiki/Bell_number`, `http://en.wikipedia.org/wiki/Stirling_numbers_of_the_second_kind`

```
   input  : Biggest Partition. Example {{a,b,c,d}}
   output: List of all partitions
 1 // An example of element is {a,b}
 2 // An example of partition for set {a,b,c,d} is {{a,c},{b,d}}

 3 // An example of list of partition is
       {{{a,b,c,d}},{{a,c},{b,d}} }
 4 // partition.n denotes nth element in a partition
 5 ListOfPartition ObtainListOfPartition(partition P)
 6 begin
 7     LP = new(ListOfPartition)
 8     if P.size() == 0 then
 9         return LP
10     end
11     if P.size() == 1 then
12         LP.push(P)
13         return LP
14     end
15     if P.size() == 2 then
16         LP.push(P)
17         LP.push({P.1},{P.2})
18         return LP
19     end
20     front = P.popfront()
21     SUB_LP = ObtainListOfPartition(P)
22     for each partition P' ∈ SUB_PL do
23         LP.push({front,P'})
24         for each element e ∈ P' do
25             e.push(front)
26             LP.push(P')
27         end
28     end
29     return LP
30 end
```

Algorithm 3: *A recursive procedure to obtain list of all partitions.*

For every partition the clustering function Φ_g is called as many times as there are clusterings (elements of a partition). For example in option 12 shown in table 5.1 we have two clusterings $\{a, b\}$ and $\{c, d\}$. Therefore for option 12, the clustering function Φ_g is called twice with $V_s = \{a, b\}$ and $V_s = \{c, d\}$.

Naive Exhaustive Search Algorithm

A naive exhaustive search based algorithm is presented in algorithm 4. Algorithm 4 is called naive because it is unoptimized.

input : $S = \{g_1, g_2, ...g_n\}$
output: $S' = \{g'_1, g'_2, ...g'_n\} | \perp$
constraints: $\forall g \in S' : mcm(g) = \gamma(v_{src}) \times \varphi(v_{src})$
$\sum_{g \in S'} Buffer(g) \leq B$
minimize: $\sum_{g \in S'} ASL(g)$

```
1  For each g ∈ S obtain all partitions of set V using algorithm 3
2  For each list of partitions for each g ∈ S filter invalid clusters
3  for all valid partitions of S do
4  |  for each g_i ∈ S do
5  |  |  for each member of partition V_s do
6  |  |  |  Insert RDMA and WDMA actors in g_i
7  |  |  |  V_sd = V_s ∪ V_d where V_d contains set of DMA actors
8  |  |  |  g'_i = Φ_g(g_i, V_sd)
9  |  |  end
10 |  end
11 |  A solution set is S' = {g'_1, g'_2, ...g'_n}
12 |  Model each graph g' ∈ S' with Round-Robin
13 |  For each g' ∈ S' Compute Buffers and Throughput
14 |  if all constraints are met then
15 |  |  Select S' as a valid solution
16 |  end
17 end
18 if no valid solutions are found then
19 |  return ⊥
20 end
21 For all valid solutions choose a solution S' with minimum ASL
22 return S'
```

Algorithm 4: *Naïve Exhaustive Search Clustering Algorithm.*

In **step 1** we obtain all partitions of set of actors for all SDF graphs. For obtaining list of partitions, algorithm 3 is used.

In **step 2** the obtained list of all partitions for all graphs is filtered for invalid clusters. We specify following two conditions to remove a partition.

1. A partition is discarded if it contains a group combining FLORA actor(s) and at least one external actor (an actor ported to ARM or other component such as SDRAM). Such an option is not useful because we are looking for FLORA configurations which consists of only FLORA actors.

2. A partition is discarded if it has at least one cluster which leads to deadlock. Algorithm 2 is used to check if a clustering option leads to deadlock.

After filtering invalid partitions, all remaining valid partitions are evaluated in **step 3**.

In **step 5-9** all the clusterings V_s (group of actors) in a partition are clustered using clustering function Φ_g to obtain the set of clustered SDF graphs S'.

In **step 6** for each clustering V_s RDMA actors are inserted in the input SDF graph for every incoming edge to the actors in V_s and WDMA actors are inserted for every outgoing edge from actors in V_s. This enables data transfer in and out of FLORA. Since RDMA and WDMA actors are part of FLORA configuration, they are added in V_s in **step 7**.

After correctly constructing set of all clustered SDF graphs S', each graph in S' is modeled with *Round-Robin* schedule in **step 12**. In round-robin modeling the response times for each actor in each graph is redefined as follows:

For all graphs, for each actor v its response time is redefined as $\varphi(v) = \varphi(v) + \sum_{w \in W} \varphi(w)$ where W is the set of actors from all SDF graphs in S' that share resources with actor v.

In **step 13** for each clustered graph in S' the required buffer capacity for each edge and the throughput for each graph is computed. A solution set S' is selected if it satisfies the buffer and throughput constraints. At the end of the algorithm in **step 21** a solution with lowest absolute scheduling load is returned.

Optimized Exhaustive Search Algorithm

Exhaustive search algorithm presented in algorithm 4 can be optimized. The optimized algorithm is presented in algorithm 5.

The first optimization is to merge the three loops in **step 1,2,3** in algorithm 4. The loops consist of generating a partition, filtering a partition and clustering the graph based on the partition. These three steps can be merged as presented in the optimized algorithm 5.

The second optimization is as follows. Buffer capacity for each channel in an SDF graph for a throughput constraint is computed using algorithm mentioned in [16]. It can have exponential complexity depending upon the number of available options for buffer size distribution for all edges. For some cases this step takes a long time. Therefore this step is conditionally executed by performing a simple throughput check for the case of infinite buffers. For the case of infinite buffers, due to self-edges throughput of an SDF graph is limited by the maximum response time of actors in the SDF graph. If a solution (set of clustered graph) fails to satisfy throughput constraint for infinite buffers it will also definitely fail for any buffer capacity of finite size. The simple throughput construct check is implemented as follows.

After round-robin modeling if for each actor v in each graph, $\varphi(v) > \varphi(v_{src}) \cdot \gamma(v_{src})$ where v_{src} is the source actor for each SDF graph, than the step for computing buffers and throughput is skipped and next partition option is tried. $\varphi(v_{src}) \cdot \gamma(v_{src})$ is the throughput constraint for SDR applications as explained in section 2.3.

input : $S = \{g_1, g_2, ...g_n\}$
output: $S' = \{g'_1, g'_2, ...g'_n\}|\perp$
constraints: $\forall g \in S' : mcm(g) = \gamma(v_{src}) \times \varphi(v_{src})$
$\sum_{g \in S'} Buffer(g) \leq B$
minimize: $\sum_{g \in S'} ASL(g)$

```
1  while all solutions have not been tried do
2  |  // Generate a solution
3  |  For each g ∈ S generate partitions of set V
4  |  if each partition for each g ∈ S is valid then
5  |  |  for each g_i ∈ S do
6  |  |  |  for each member of partition V_s do
7  |  |  |  |  Insert RDMA and WDMA actors in g_i
8  |  |  |  |  V_sd = V_s ∪ V_d where V_d contains set of DMA actors
9  |  |  |  |  g'_i = Φ_g(g_i, V_sd)
10 |  |  |  end
11 |  |  end
12 |  |  A solution set is S' = {g'_1, g'_2, ...g'_n}
13 |  |  Model each graph g' ∈ S' with Round-Robin
14 |  |  if simple throughput check does not fail then
15 |  |  |  For each g' ∈ S' Compute Buffers and Throughput
16 |  |  |  if all constraints are met then
17 |  |  |  |  Select S' as a valid solution
18 |  |  |  end
19 |  |  end
20 |  end
21 end
22 if no valid solutions are found then
23 |  return ⊥
24 end
25 For all valid solutions choose a solution S' with minimum ASL
26 return S'
```

Algorithm 5: *Optimized Exhaustive Search Clustering Algorithm.*

5.2.2 Heuristic Approach

The largest FLORA configurations give the lowest scheduling load (larger configuration means more blocks are grouped together within FLORA and blocks within FLORA trigger themselves as soon as data arrives, thus reducing the scheduling load on ARM). Moreover for FLORA the larger configurations usually also satisfy throughput constraint (since larger configurations reduce read/write overhead for data transfer in and out of FLORA). Therefore a heuristic algorithm is proposed as follows.

After obtaining list of all partitions for all graphs, the list can be ordered from the largest partitions to the smallest partitions or they can also be generated is such order. The first solution which satisfies all constraints is chosen as the final optimal solution. Since the list of partitions were ordered, the first feasible solution may also have the lowest scheduling load. This algorithm is presented in algorithm 6. The algorithm shall keep searching until a valid solution is found. Therefore the worst case running time of algorithm 6 is same as the worst case running time of the exhaustive search algorithm 4.

In general the solution obtained by algorithm 6 may not be optimal. However for SDR applications the deviation is negligible as observed in section 5.4.

input : $S = \{g_1, g_2, ...g_n\}$
output: $S' = \{g'_1, g'_2, ...g'_n\} | \perp$
constraints: $\forall g \in S' : mcm(g) = \gamma(v_{src}) \times \varphi(v_{src})$
$\sum_{g \in S'} Buffer(g) \leq B$
minimize: $\sum_{g \in S'} ASL(g)$

```
1  For each g ∈ S obtain all partitions of set V
2  For each list of partitions for each g ∈ S filter invalid clusters
3  Sort list of partitions for each g ∈ S in descending order of partition size
4  for all valid partitions of S do
5  |  for each g_i ∈ S do
6  |  |  for each member of partition V_s with |V_s| > 1 do
7  |  |  |  Insert RDMA and WDMA actors in g_i
8  |  |  |  V_sd = V_s ∪ V_d where V_d contains set of DMA actors
9  |  |  |  g'_i = Φ_g(g_i, V_sd)
10 |  |  end
11 |  end
12 |  A solution set is S' = {g'_1, g'_2, ...g'_n}
13 |  Model each graph g' ∈ S' with Round-Robin
14 |  For each g' ∈ S' Compute Buffers and Throughput
15 |  if all constraints are met then
16 |  |  return S'
17 |  end
18 end
19 if no valid solutions are found then
20 |  return ⊥
21 end
```

Algorithm 6: *Stop-at-first-solution heuristic based Clustering Algorithm.*

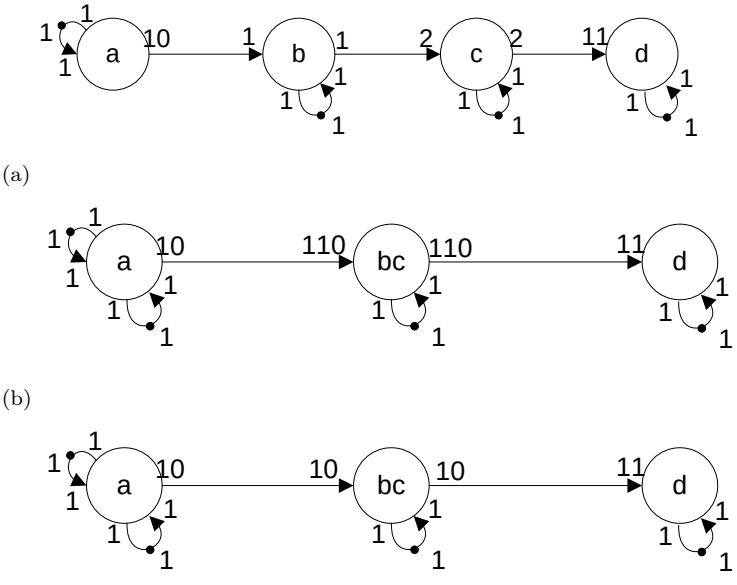
5.3 Implementation of Clustering Algorithm

We described clustering algorithms in pseudo code for applications represented in SDF graphs in general in the previous section. The details of implementation of clustering algorithms for SDR applications and FLORA are presented in this section.

- A *cluster-analysis-sdf* tool is developed in C++ using classes of **SDF3**. SDF3 is a collection of tools for generating, transforming and analyzing HSDF, SDF graphs [17].
- SDF graphs representing SDR applications are taken as input in an XML file. A separate XML file is used to provide details of FLORA platform such as execution time of DMA blocks. The output consisting of optimal clusterings for each application, buffer capacity for each edge, the total required buffer capacity and scheduling load is outputted in a text file.
- To check for deadlock the efficient algorithm (algorithm 2) is implemented.
- Two versions of clustering algorithms are implemented, naive exhaustive search algorithm (algorithm 4) and the stop-at-first-solution heuristic algorithm (algorithm 6).
- For the implementation we do not use the value of k (number of sub-graph iterations to be clustered) specified in definition 7. It was explained in section 4.3 that for our definition of generalized clustering we restrict k to deliver consistent graphs. The value of k was defined as $k = \frac{\gamma(v)}{\gamma_g(v)}$ where γ is repetition vector of input graph and γ_g is the repetition vector of sub-graph formed by the actors to be clustered. However for SDF graphs with prime rates, the repetition vector explodes based on least common multiple of the token rates. Thus the consumption rate(s), the production rate(s) and the response time of the clustered actor is also set to a corresponding high value. For example consider SDF graph in figure 5.2(a). The resulting clustered SDF graph using the clustering function Φ_g defined in definition 7 is shown in figure 5.2(b). In figure 5.2(b) we see that the new consumption and production rate is 110 which results in a buffer capacity requirement of incoming and outgoing edge to be at least 110. Similarly the response time of clustered actor *bc* explodes to cover as many sub-graph iterations as specified by k.

  For practical SDR applications which have uneven prime rates using the value of k specified in definition 7 results in impractical buffer requirements and very long response times of FLORA configurations. Therefore we propose a value of k which results in practical buffers and response times.

  We let as many iterations of sub-graph be clustered as many as are possible in a single firing of the source actor v_{src} of every application. Thus the



(a)

(b)

(c)

Figure 5.2: *(a) An example of SDF graph with repetition vector* $\gamma = \{11, 110, 55, 10\}$. *(b) Resulting clustered SDF graph using clustering function* Φ_g *defined in definition 7 for* $V_s = \{b, c\}$. *(c) Resulting clustered SDF graph for* k *defined in equation 5.1.*

new value of k is

$$k = \left\lfloor \frac{\gamma(v)}{\gamma(v_{src})} \right\rfloor \tag{5.1}$$

Where $v \in V_s$, V_s is the set of actors to be clustered and v_{src} is the source actor of application. The resulting clustered graph using the new practical value of k specified in equation 5.1 is shown in figure 5.2(c). In general the new practical k specified in equation 5.1 may result in inconsistent graphs. However for SDR applications which do not have multiple paths, no inconsistent graphs are generated.

5.4 Run-Time Evaluation of Algorithms

FLORA has been designed for fast decoding of SDR applications. Therefore for evaluating performance of proposed clustering algorithms, the clustering algorithms were experimented with SDR applications. In table 5.2 we summarize

the results for experiments with different modes of DVB-T, DVB-SH and XM-T standards [2].

For every input a detailed output consisting of optimal configurations, clustered graphs, required buffer capacity for each edge, total buffer capacity, and total scheduling load were generated. However for brevity we only capture performance parameters in table 5.2. In table 5.2, column 2 contains the set of applications which were given as input. Column 3 mentions if the clustering algorithm was able to find a solution for which throughput and buffer constraints were satisfied. Column 4 mentions the number of clustering options which were generated for the given set of applications. Column 5 gives the total number of HSDF actors present for each given application. Column 6 gives FLORA utilization which is the percentage of time FLORA is busy on average and it is computed by taking the ratio of throughput constraint of an application to the maximum attainable throughput (without source actor). Thus utilization above 100% means throughput constraint was not met. Column 7 shows the total time spent in generating clustered graphs in the execution of clustering algorithm. Column 8 shows the total time spent in computing buffers in the complete execution of clustering algorithm. Column 6-9 capture performance of exhaustive search algorithm 4. Column 10-13 capture performance of heuristic algorithm 6. From the experiments following inferences were observed.

- The experiments were done incrementally so as to fit maximum possible applications. Single applications of DVB-T and DVB-SH standards were tried in input set 1 to 4 in table 5.2. Based on these observations two applications were fed as input in input set 5,6. Input set 5 failed to meet throughput constraint by a margin of 4%. Accordingly DVB-T mode was reduced from 64QAM (Quadrature Amplitude Modulation) to 4QAM which is a change of 36288 Bytes per heart beat(time period of SDR application) to 12096 Bytes per heart beat. Thus reducing throughput requirement of input 6. Solution for input 6 was obtained with a slack of 6.5%. Thus the clustering algorithm proves helpful in finding optimal configurations for FLORA for a given set of applications.

- For the mentioned SDR applications in table 5.2 the optimal solutions given by exhaustive search algorithm and stop-at-first-solution heuristic algorithm matched. This is due to the fact that SDR applications consist of actors with an extremely wide range of token granularity. Therefore smaller configurations incur huge read/write overhead due to data-transfer in and out of FLORA which breaks the throughput constraint. Thus the bigger configurations (with more FLORA blocks configured together) reduce the read/write overhead by making maximum use of datapath within FLORA and also incur a lower scheduling load. In heuristic algorithm the clustering options are ordered from biggest to smallest, therefore for SDR

[2]The experiments were conducted on a PC with Intel Core 2 Duo processor (2 Cores on 3.06GHz), 4GB of main memory and Linux OS

applications usually the first solution which satisfies constraints is also the optimal solution.

- For positive solutions, the running time of stop-at-first-solution heuristic algorithm is negligible compared to the running time of exhaustive search algorithm. For negative solutions, the worst case running time of heuristic algorithm is same as the worst case running time of the exhaustive search algorithm. Therefore for input set 5 in table 5.2 where no solutions could be found, the running time of heuristic algorithm and exhaustive search algorithm was observed to be same.

- For all set of inputs the total time spent in generating clustered graphs is consistent and in the of range of milliseconds to few minutes. This is due to the fact that generating clustered graphs involves two parts - first part is removing/adding nodes and edges which is trivial. The second part computes execution time of the clustered actor which involves computing throughput and latency of sub-graphs to be clustered. Since the sub-graphs are smaller in size (with respect to number of nodes and edges) compared to the original graph, the second part also takes acceptable time.

- On the other hand the total time spent in computing buffers is inconsistent varying from milliseconds to hours and depends on the input SDF graphs. The algorithm used in SDF3 to compute buffers explores all the options of buffer distribution for all edges until the throughput constraint is satisfied [16]. This algorithm has exponential run-time complexity. Therefore for some cases of SDR applications the algorithm to compute buffers did not terminate in an acceptable time (an hour). Such cases are omitted in table 5.2.

- In table 5.2, the input set in 6 uses the same DVB-T mode specified in input set 3. However due to the bottleneck of computing buffers the SDF graph in 3 was simplified with respect to the token rates to reduce the complexity of computing buffers for evaluating input set 6. Therefore the application in input set 6 is called DVB-T'-8K-4QAM.

- In round-robin modeling for SDF graphs, response times of actors are added if they share a resource to capture worst case response time. In worst case actors share the resource for equal share of time. However for the average case the obtainable response time is much lower. For example SDR applications with different heart beats (period of input) share the resources with different frequencies. However for such cases the clustering algorithm makes use of worst case response time and therefore throughput constraint of applications are not met. Such cases were excluded in table 5.2. For making use of average case response time either static-order scheduling or a probabilistic model of resource sharing was necessary. Static-order scheduling was not possible under the assumption

of unsynchronized applications. Probabilistic model is not used in this thesis.

					Exhaustive Search				Heuristic Search			
No.	Application(s)	Const -raints Satis- fied	Cluste -ring Op- tions	HSDF actors	FLORA Uti- liza- tion	Cluste -ring Time	Buffer Com- puta- tion Time	Total Time	FLORA Uti- liza- tion	Cluste -ring Time	Buffer Com- puta- tion Time	Total Time
1	DVBT-2K-64QAM	Yes	32	2514	29.70%	3.27s	3ms	3.273s	29.70%	41ms	1ms	42ms
2	DVBT-8K-64QAM	Yes	32	8751	28.90%	6.72s	64ms	6.73s	28.90%	54ms	4ms	58ms
3	DVBSH-8K-4QAM	Yes	16	484157	56.97%	38.4s	25.3min	26min	56.97%	2.4s	100ms	2.5s
4	DVBSH-8K- 16QAM	Yes	16	7685	95.34%	4.7s	89ms	4.71s	95.34%	48ms	4ms	52ms
5	DVBT-8K-64QAM +DVBSH-8K- 4QAM	NO	32×16 =512	7685 +8751	104.2%	3.34min	120ms	3.36min	104.2%	3.34min	120ms	3.36min
6	DVBT'-8K-4QAM +DVBSH-8K- 4QAM	Yes	32×16 =512	5024 +7685	93.51%	2.7min	36s	2.76min	93.51%	1.98s	1ms	1.98s
7	XM-T	Yes	8	483783	6.71%	2.7s	24ms	2.7s	6.71%	0.9s	1ms	0.9s

Table 5.2: *Summary of results for experiments with different modes of DVB-T, DVB-SH and XM-T standards.*

Chapter 6

Conclusion

Integration of digital wireless communication and embedded systems has facilitated a wide range of wireless services. These are represented under a common framework of Software Defined Radio (SDR). SDR applications are firm real time streaming applications. Due to the wide range of variety in supported services, each stage in digital baseband processing of SDR application has different requirements for flexibility and performance. To suit the requirements of each stage a heterogeneous MPSoC platform has been designed. For fast channel decoding of SDR streams FLORA hardware accelerator has been designed as part of the heterogeneous MPSoC. FLORA consists of diverse hardware blocks for decoding common kernels used in SDR applications. FLORA enables high performance with hardware implementation but it is also configurable to support parametric kernels and multiple standards. An analytical method is required to exploit the flexibility offered by FLORA and also to guarantee the throughput requirement of each SDR application ported to FLORA. This problem is addressed in this thesis.

In section 6.1 we draw conclusions from the presented work in this thesis. We conclude this chapter in section 6.2 where we list the shortcomings of this thesis and propose future work which can improve the work presented in this thesis. The proposed analytical method in this thesis is based on clustering for SDF model of computation. Hence apart from FLORA, the analytical method is also applicable for a range of scheduling problems in MPSoC domain. We illustrate the general application of clustering in section 6.2.1.

6.1 Conclusion

This thesis presents an analytical method based on clustering of SDF actors which helps to analyze trade-off between read/write overhead and pipelining for FLORA and it also helps to capture the hierarchical schedule of actors within FLORA and outside FLORA for the same application described as SDF graph. The conclusions from this are as follows

- For configurable hardware such as FLORA we identified challenges in configuring FLORA for multiple applications. FLORA is designed to be a configurable hardware so as to support multiple SDR applications. However due to configurable hardware blocks and configurable datapath within FLORA we have many configuring options for FLORA and each option

provides different throughput. Bigger configurations have longer execution times due to small buffers within FLORA and hence can be a bottleneck. Smaller configurations allow pipelining by buffering in external memories, however they also add read/write overhead which can also reduce the throughput. Thus for supporting multiple applications it is not immediately clear which configuration options satisfy throughput requirements for multiple applications.

- We showed that SDF model can be used to evaluate the optimal configuration option for FLORA that satisfies throughput constraint of each application. We model a FLORA configuration as a clustered SDF actor. Such modeling helps model the atomicity of a FLORA configuration and also we can abstract from the schedule of actors mapped to FLORA thus modeling the hierarchical schedule for a radio application mapped to MARS MPSoC
- An exhaustive algorithm was presented which evaluates all the clustering options and a heuristic algorithm was presented which evaluates clustering options starting with the biggest partition and stops at the first option which satisfies throughput constraint.
- It was found out that for present FLORA with an assumption of single RDMA and WDMA, the biggest configurations perform best. Therefore the heuristic algorithm which starts with the biggest configurations performs as good as the exhaustive algorithm.
- The algorithms were evaluated with real SDR applications. The exhaustive search algorithm finds solution within an hour and the heuristic algorithm finds a solution within few minutes. It was observed that there are two bottlenecks in both the algorithms. Firstly the total number of clustering options rise more than exponentially with number of SDF actors in the input SDF graphs. Secondly the step which computes buffer capacities for clustered SDF graphs can also take exponential time depending upon the input SDF graphs.
- The proposed clustering algorithm in this thesis can be extended for multiple DMAs and can be used for any hardware accelerator which adheres to the FLORA template described in chapter 1.

6.2 Future Work

The work in this thesis can be extended as follows

- The exhaustive clustering algorithm is based on obtaining all partitions of set of nodes. For a set with n elements the total number of partitions is in the order of $O(n) < \left(\frac{n}{\log(n)}\right)^n$ [1]. Therefore for input SDF graphs with

[1] http://en.wikipedia.org/wiki/Bell_number

more than 14 actors, the running time of exhaustive clustering algorithm is unacceptable (hours). A better technique can be implemented to explore only useful partitions.

- We have assumed only one RDMA and WDMA block is present in the FLORA hardware accelerator. Thus all the RDMA and WDMA actors inserted for every clustering option share one RDMA and one WDMA block. For a finite number of available DMAs, assigning DMAs for every communication edge in an SDF graph to maximize parallel communication is a non-trivial problem. This problem can be solved and integrated in the present clustering method.

- Cyclo Static Data Flow (CSDF) model of computation offers more expressibility and can capture some applications details more eloquently than SDF model of computation. Therefore the proposed clustering method in this thesis can be extended for CSDF model of computation.

- A bottleneck in the proposed clustering algorithms is computation of buffer capacities for channels in the clustered SDF graphs. Currently the algorithm used from SDF3 toolset computes all possible buffer distribution for all channels until the throughput constraint is satisfied. For some SDF graphs representing SDR applications, this algorithm takes unacceptable time to terminate. Therefore for computing buffer capacities an efficient algorithm could be used.

- For scheduling multiple clustered SDF graphs static-assignment scheduling (round-robin) was used. In round-robin modeling for input SDF graphs with unequal heart beats(time period of input) response times of all actors which share resources are added. Due to this throughput constraint of SDF graphs with lower heart beat is not met. This makes clustering algorithm inapplicable for SDF graphs with unequal heart beats. Therefore a better scheduling strategy is required for scheduling multiple SDF graphs.

6.2.1 General Application of Clustering for SDF

The clustering method proposed in this thesis is based on SDF model of computation. SDF model of computation has been used to address scheduling problems for streaming applications [1, 2, 10, 11, 13, 14]. Therefore the proposed clustering method can be extended for addressing scheduling problems in the MPSoC domain. Specifically for scheduling problems where it is required to schedule multiple applications with following assumptions

- Applications have real time constraints

- Communication cost cannot be ignored

For illustration consider a generic MPSoC template shown in figure 6.1. It consists of four tiles. Every processor P_i has a DMA D_i and cache C_i of its own.

M is a shared memory. All tiles are connected by a Network -on-Chip (NoC). Consider a streaming application which can be modeled as an SDF graph shown in figure 6.2. For simplicity only tasks (nodes) and dependencies (edges) have been shown. The application has real-time constraints and the communication cost of NoC is not negligible.

Since number of tasks are more than the number of processors, more than one task will be scheduled on a processor. Such a mapping decision can be modeled with the clustering method proposed in this thesis. Every such mapping decision has an impact on throughput of application. The impact can be analyzed with two aspects, the communication cost offered by the mapping decision and the execution time of the clustered composite tasks. For example consider tasks b,c,d in figure 6.2. Assume that for application shown in figure 6.2, tasks b,c,d are the bottleneck for throughput. We illustrate two cases with their analysis.

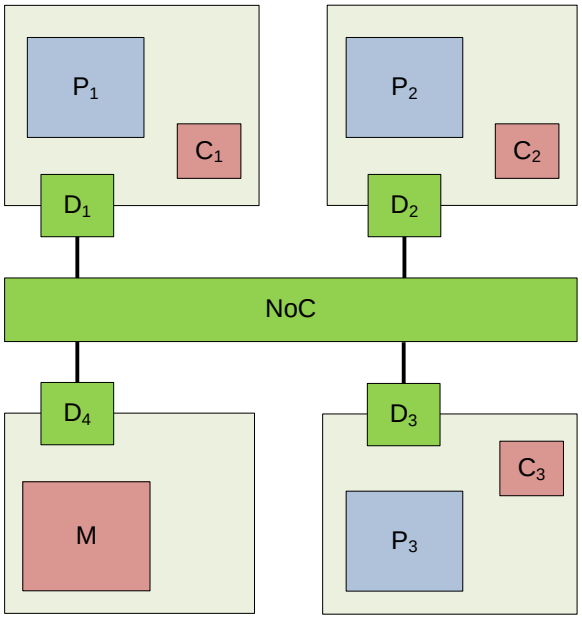


Figure 6.1: *An MPSoC Template.*

Case I Lets say tasks b,c,d are mapped to processor P_1. The communication cost in the proposed method is modeled with DMA actors. For simplicity let $T_R(e)$ be the time spent in reading data on an edge e and $T_W(e)$ be the time spent in writing data on an edge e. Assume a non-preemptive static schedule is designed for processor P_1. Since there is only one processor, tasks b,c,d will run sequentially run on processor P_1. Edges e_2, e_3, e_4 are stored in cache C_1 therefore communication cost of these edges will be neglected. Therefore the throughput of the composite actor bcd is

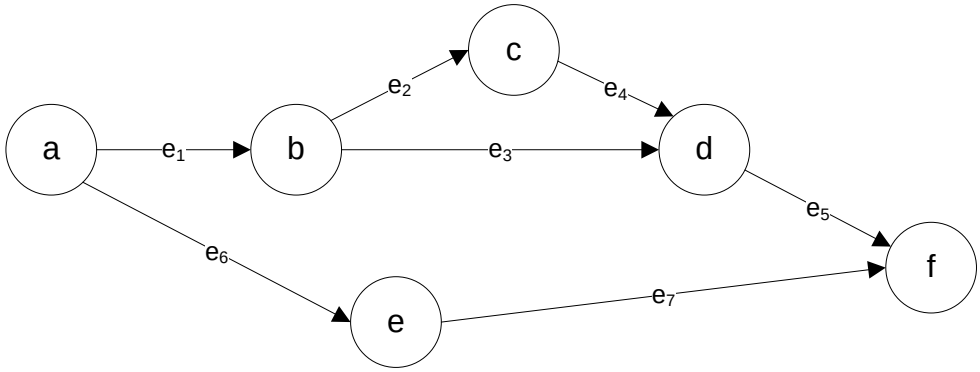


Figure 6.2: *An example of application.*

$$Thput = \frac{1}{T_R(e_1) + T(b) + T(c) + T(d) + T_W(e_5)}$$

Case II We now map task b, c, d on three processors P_1, P_2, P_3 respectively. Since edges e_2, e_3, e_4 will now be stored in the external memory M, the throughput of application will be

$$Thput = \max\left(\frac{1}{T_R(e_1) + T(b) + \max\left(T_W(e_2), T_W(e_3)\right)}, \frac{1}{T_R(e_2) + T(c) + T_W(e_4)}, \frac{1}{\max\left(T_R(e_3), T_R(e_4)\right) + T(d) + T_W(e_5)}\right)$$

Case I has low communication overhead but the execution time of tasks b,c,d get added up due to sequential execution and can be a bottleneck. Case II offers pipelining between actors b,c,d but the communication cost of edges e_2, e_3, e_4 can be a bottleneck. Therefore to decide the best mapping decision we can model mapping of tasks to a processor as clustering of the SDF actors in a single SDF actor. Then the proposed clustering algorithms can analyze all possibilities of clustering and find the mapping which can satisfy the throughput constraint.

In the proposed clustering method all sub-graphs are scheduled with self-timed schedule (actors fire as soon as they can) and the parent clustered graph containing the clustered actors is scheduled with static-assignment schedule (round-robin). However in our clustering approach the clustered actor is treated as an atomic SDF actor. Therefore sub-graphs to be clustered are scheduled independently than the parent clustered graph. It is possible to further generalize, that is each sub-graph and the parent graph can be scheduled independently with different scheduling strategies. Therefore in the above example every processor can use different scheduling strategy.

In this way the proposed clustering method for SDF model of computation can be extended for scheduling problems in MPSoC domain.

Appendices

Appendix A

MARS MPSoC Platform

NXP has developed an MPSoC platform called as MARS for the baseband processing of SDR applications. This platform was designed not only keeping in mind the performance requirements of SDR applications but also to serve multiple standard multiple applications. Figure A.1 shows a basic block digram of the MARS MPSoC. In the following paragraph we relate blocks in figure A.1 to the blocks in the baseband processing of SDR explained in section 1.1 and figure 1.1.

The transceiver block serves the function of AFE. The DFE is a hardware block designed to serve the function of DFE. VDSP1 and VDSP2 are Vector Digital Signal Processors which perform the task of inner receiver. ARM processor and FLORA are used to implement the outer receiver or the codec stage. FLORA can only decode, it is not used for encoding.

All the blocks are connected via a set of buses shown in black lines. The decoded data is transmitted to a host for processing of the MAC layer via the Universal Serial Bus (USB) . Apart from internal Static Random Access Memory(s) (SRAM), MARS chip also has a connection to an external Synchronous Dynamic Random Access Memory (SDRAM) which has larger storage capacity. UPL block is used for Up-Link that is to transmit data. GPIO is General Purpose Input/Output.

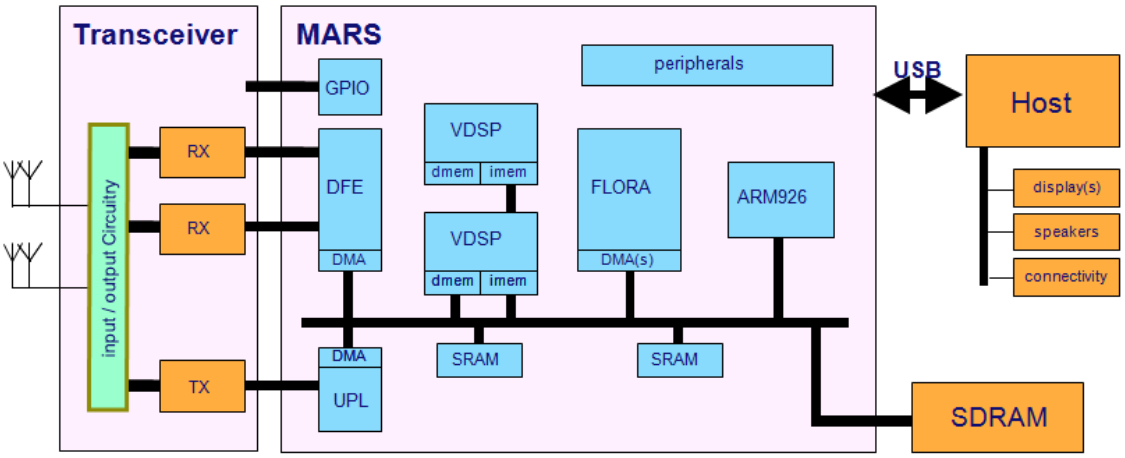


Figure A.1: *MARS MPSoC Platform*

Appendix B

FLORA

As explained in section 1.1 codec stage is best implemented on a configurable hardware for high performance. We have FLORA as a hardware accelerator designed for fast forward error correction in the outer receiver. Its design is as follows.

1. Figure B.1 shows block diagram of FLORA. It consists diverse hardware blocks. Each block is capable of processing data for a specific decoding technique, those mentioned in table 1.1. But there are multiple blocks for same functions. For example as we can see in figure B.1, we have two de-puncture units, two de-scramblers, etc. Multiple blocks allow parallel processing of multiple radio streams.

2. Although hardware blocks process only fixed functions, they can process family of the same function. They can be configured for different set of equations, for different input/output sizes. Each block has two sets of registers for storing two configurations. We can program a set of registers while other set is running.

3. All blocks are non-preemptible. Prempting by re-configuring the configuration which is still running will corrupt the data.

4. Different blocks work on different granularity ranging from few bits to thousands of bytes. But majority of blocks work on bits.

5. Data transfer on the external shared bus is costly. Therefore FLORA has its own internal set of buses to allow passing data from a block to another block. Keeping in mind the most common order of functions used in wireless standards, a set of datapaths have been created using a configurable bus matrix shown in figure B.1. The matrix is illustrated in figure B.2. We can see each block can accept data from one of the preceding blocks. Each block has a multiplexer to choose the block from which it can accept data.

6. All blocks have input and output buffers. But some blocks have big buffers (De-Interleaver has 1Mbit buffer), while some have small buffers in order of few bits (De-puncture, Viterbit, De-Scrambler, etc.)

7. All blocks are data driven. That is once configured a block triggers as soon as there is data in input buffer. After processing it sends data to the

input buffer of next block connected to its output which triggers the next block and so on. Thus by configuring a set of blocks on a datapath we can decode the data stream as specified by the standard.

8. Blocks within FLORA are data driven, but data has to be fed in from outside of FLORA via a RDMA and has to be taken out of FLORA via a WDMA. Thus from RDMA to set of connected and configured blocks to WDMA we form a unidirectional datapath.

9. Therefore, once we have decided on the set of blocks to be programmed, a necessary and complete configuration of FLORA implies

   - configuration of each block in FLORA for the mathematical equation, parameters and input/output size.
   - configuration of input multiplexers of each block - that is configuring the datapath
   - configuration of RDMA and WDMA with the appropriate data source and sinks addresses and sizes.

   Such a combined complete configuration is called as a *slot*.

10. Blocks within FLORA are data-driven. However a slot can only be programmed by some external entity such as general purpose processor. For example presently ARM processor configures a slot. And therefore a slot is not data-driven.

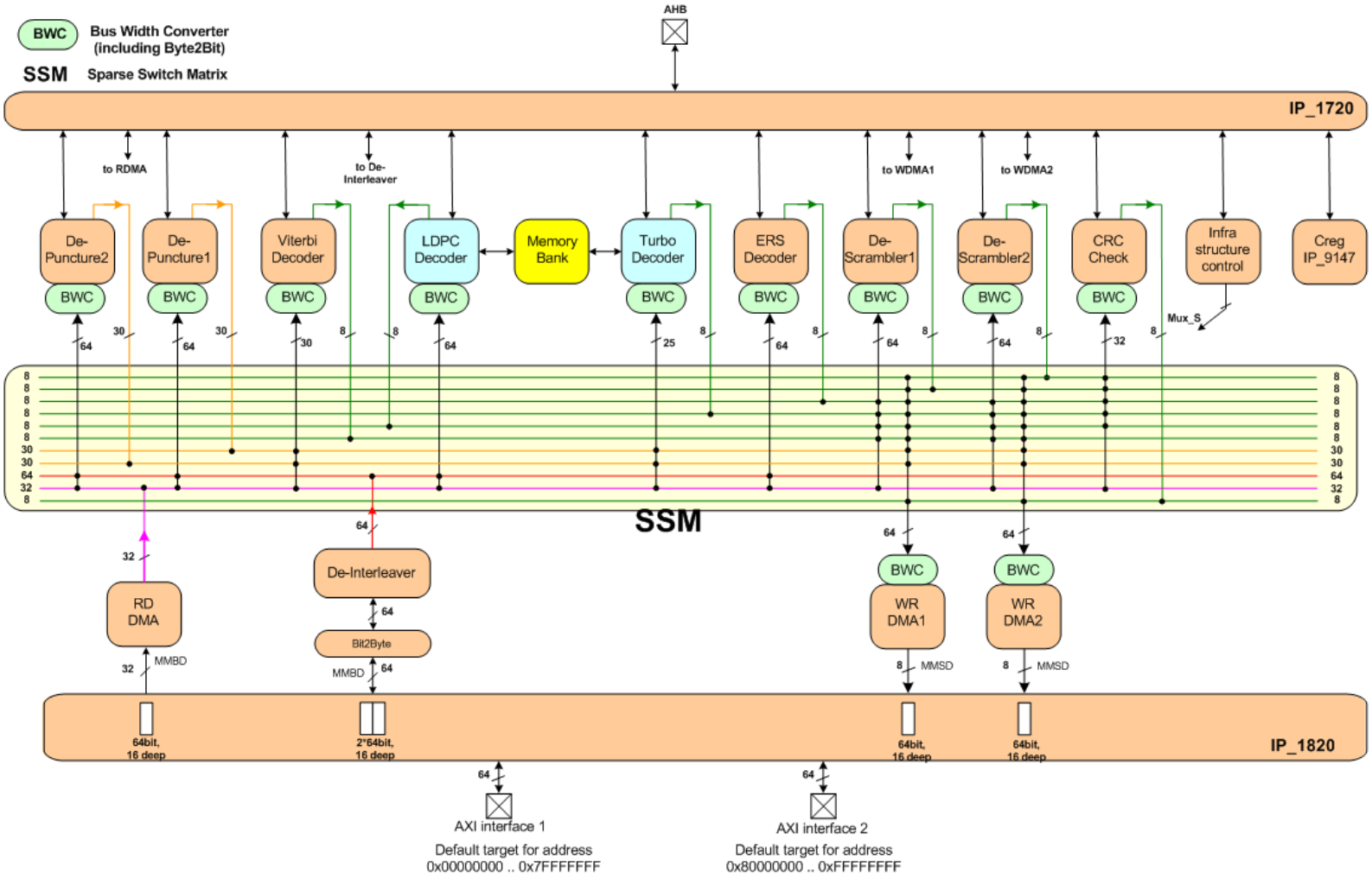


Figure B.1: *Block Diagram of FLORA*

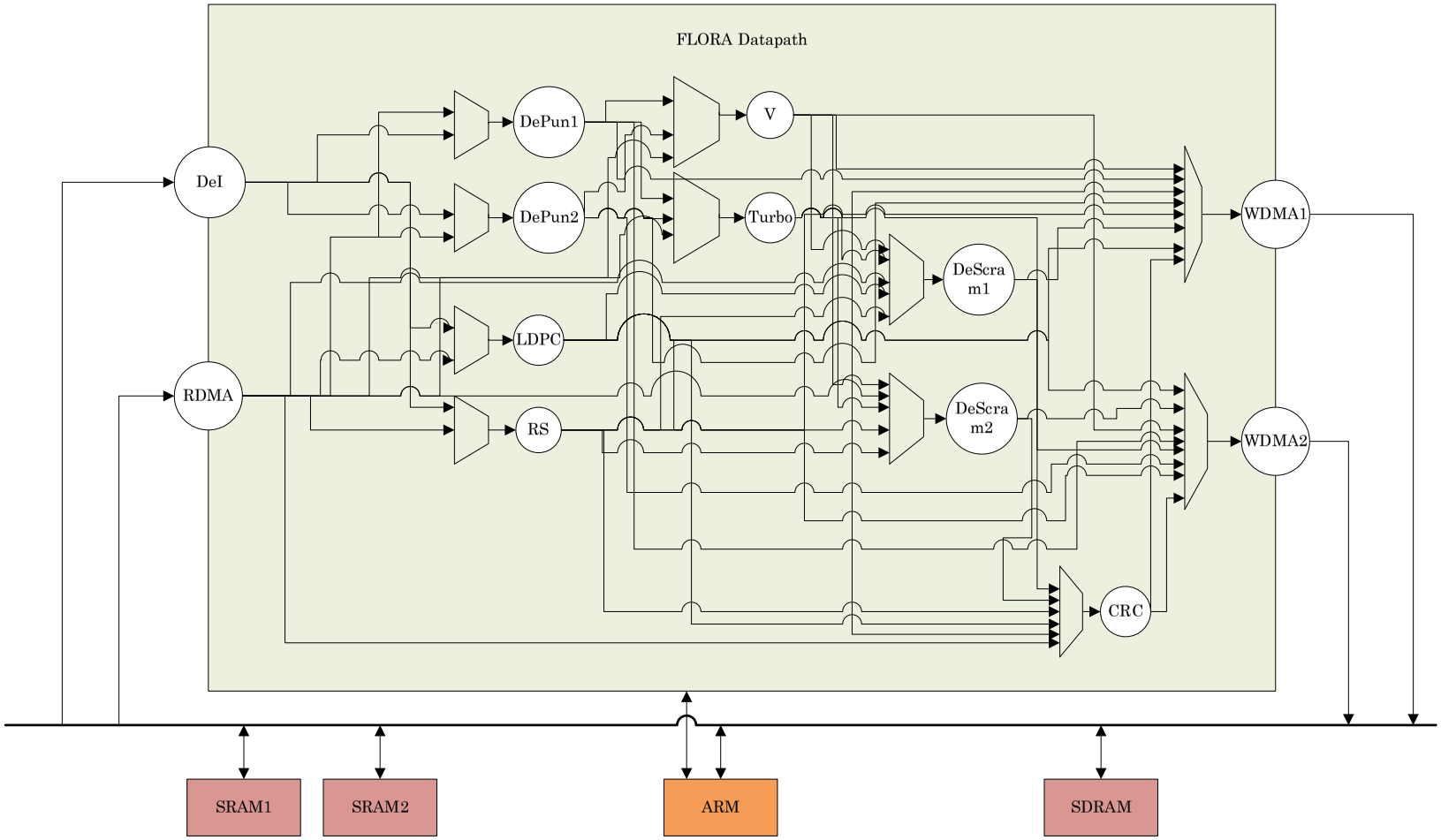


Figure B.2: *Schematic diagram of FLORA datapath and interface to external components*

Appendix C

List of Symbols

$\mathbb{N}_0$	Set of Natural numbers $\mathbb{N}_0 = \{0, 1, 2..\}$
$\mathbb{N}$	$\mathbb{N} = \mathbb{N}_0 \setminus \{0\}$
V	Set of nodes
E	Set of edges
δ	Initial tokens function $\delta : E \to \mathbb{N}_0$
φ	Response time function $\varphi : V \to \mathbb{N}_0$
π	Production rate function $\pi : E \to \mathbb{N}_0$
μ	Consumption rate function $\mu : E \to \mathbb{N}_0$
γ	Repetition Vector function $\gamma : V \to \mathbb{N}_0$
γ_G	Repetition Vector function of an SDF graph G
mcm	Maximum Cycle Mean function for SDF graph $mcm : G \to \mathbb{N}_0$
Φ_c	Clustering function for a connected SDF sub-graph defined in definition 6
Φ_g	General Clustering function for an SDF sub-graph(s) defined in definition 7
θ	Duration of transient phase of a self-timed schedule of an SDF graph defined in equation 4.1
ρ	Duration of periodic phase of a self-timed schedule of an SDF graph defined in equation 4.1

Glossary

AFE Analog Front End. 2, 3, 68

APG Acyclic Precedence Graph. 24, 25

ASL Absolute Scheduling Load. 43

ATSC Advanced Television Systems Committee. 2

CSDF Cyclo Static Data Flow. 65

DAB Digital Audio Broadcasting. 2

DFE Digital Front End. 3, 68

DMA Direct Memory Access. 7, 25, 56, 61, 63, 65

DSP Digital Signal Processor. 3, 4

DVB Digital Video Broadcasting. 2

DVB-T Digital Video Broadcasting -Terrestrial. 6, 7, 11, 17, 18

FEC Forward Error Correction. 4

FIFO First In First Out. 21, 31, 32, 41–43

FLORA FLexible Outer Receiver Architecture. ii, 2, 5–10, 19–27, 31, 35, 42, 50, 52–54, 56–58, 61, 62, 65, 68, 70–73

GSM Global System for Mobile Communications. 2

HSDF Homogeneous Synchronous Data Flow. 14, 16, 17, 24, 25, 37, 47, 48, 56, 58

ISDB Integrated Services Digital Broadcasting. 2

LTE Long Term Evolution (3GPP 4G technology). 2

MAC Media Access Control. 3, 68

mcm Maximum Cycle Mean. 17, 43

MPSoC Multi-Processor System on Chip. 1, 2, 6, 9, 10, 14, 19–23, 61–64, 68

RDMA Read Direct Memory Access. 5–7, 9, 20, 25, 26, 42, 53, 61, 65, 71

SDF Synchronous Data Flow. ii, 6–9, 14–18, 20, 21, 23–43, 47–51, 53, 56, 59, 61–66

SDR Software Defined Radio. ii, 1–7, 14, 17, 19, 21, 23, 47, 50, 53, 55–59, 61, 62, 66, 68

SRDF Single Rate Data Flow. 16

WDMA Write Direct Memory Access. 5–7, 9, 20, 25, 26, 42, 53, 61, 65, 71

WiFi Wireless Fidelity (IEEE 802.11). 2

XM Satellite radio. 2

XML eXtensible Markup Language. 23, 56

Bibliography

[1] Bekooij, M., Moreira, O., Poplavko, P., Mesman, B., Pastrnak, M., and Meerbergen, J. V. Predictable embedded multiprocessor system design. In *In Proc. Intl Workshop on Software and Compilers for Embedded Systems (SCOPES), LNCS 3199* (2004), Springer.

[2] Bonfietti, A., Lombardi, M., Milano, M., and Benini, L. Throughput constraint for synchronous data flow graphs. In *Proceedings of the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems* (Berlin, Heidelberg, 2009), CPAIOR '09, Springer-Verlag, pp. 26–40.

[3] Buck, J., Ha, S., Lee, E. A., and Messerschmitt, D. G. Ptolemy: A framework for simulating and prototyping heterogeneous systems, 1992.

[4] Falk, J., Keinert, J., Haubelt, C., Teich, J., and Bhattacharyya, S. S. A generalized static data flow clustering algorithm for mpsoc scheduling of multimedia applications. In *Proceedings of the 8th ACM international conference on Embedded software* (New York, NY, USA, 2008), EMSOFT '08, ACM, pp. 189–198.

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

[6] Ghamarian, A., Stuijk, S., Basten, T., Geilen, M., and Theelen, B. Latency minimization for synchronous data flow graphs. *Digital Systems Design, Euromicro Symposium on 0* (2007), 189–196.

[7] Haykin, S., and Moher, M. *Modern Wireless Communications*. Prentice Hall, 2004.

[8] Heath, S. *Embedded System Design*, second ed. Newnes, 2002.

[9] Hentschel, T., Henker, M., and Fettweis, G. The digital front-end of software radio terminals. *Personal Communications, IEEE 6*, 4 (aug 1999), 40 –46.

[10] Kumar, A. *Analysis, Design and Management of Multimedia Multiprocessor Systems*. PhD thesis, Eindhoven University of Technology, 2009.

[11] Lee, E., and Messerschmitt, D. Synchronous data flow. *Proceedings of the IEEE 75*, 9 (sept. 1987), 1235 – 1245.

[12] Pino, J. L., Bhattacharyya, S. S., and Lee, E. A. A hierarchical multiprocessor scheduling system for dsp applications. In *In Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers* (1995), pp. 122–126.

[13] Sriram, S., and Bhattacharyya, S. S. *Embedded Multiprocessors: Scheduling and Synchronization*, 1st ed. Marcel Dekker, Inc., New York, NY, USA, 2000.

[14] Stuijk, S. *Predicatable Mapping of Streaming Applications on Multiprocessors*. PhD thesis, Eindhoven University of Technology, 2007.

[15] Stuijk, S., Basten, T., Geilen, M., and Corporaal, H. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE* (june 2007), pp. 777 –782.

[16] Stuijk, S., Geilen, M., and Basten, T. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Design Automation Conference, 2006 43rd ACM/IEEE* (0-0 2006), pp. 899 –904.

[17] Stuijk, S., Geilen, M., and Basten, T. SDF^3: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings* (June 2006), IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 276–278.

[18] Stuijk, S., Geilen, M., and Basten, T. A predictable multiprocessor design flow for streaming applications with dynamic behaviour. In *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on* (sept. 2010), pp. 548 –555.

[19] Tong, W. Multi-standard multi-channel channel decoder architecture for mobile applications. Master's thesis, Eindhoven University of Technology, 2009.

[20] Wiggers, M., Bekooij, M., Jansen, P., and Smit, G. Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis* (New York, NY, USA, 2006), CODES+ISSS '06, ACM, pp. 10–15.

[21] Wiggers, M. H., Bekooij, M. J. G., and Smit, G. J. M. Modelling run-time arbitration by latency-rate servers in dataflow graphs, 2007.

[22] Wireless Innovation Forum. *SDRF Cognitive Radio Definitions*. http://www.sdrforum.org/pages/documentLibrary/documents/SDRF-06-R-0011-V1_0_0.pdf.