ABSTRACT

Title of dissertation: MODEL-BASED HARDWARE DESIGN FOR IMAGE PROCESSING SYSTEMS

Mainak Sen, Doctor of Philosophy, 2006

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

Model-based design has been touted as the most viable design methodology of the future for the design of embedded hardware/software systems. Due to the large complexity of modern embedded systems, it is more and more error-prone to design systems without having a formal model to support and verify the application at design time. Also, formal models generally capture broad classes of applications, and thus any innovation on a modeling technique has the potential to enhance every individual application in the associated class. Often, a formal model captures the high-level abstraction of an application, which is lost in the final implementation, and thus modeling gives an effective platform to perform high-level design optimizations. Dataflow graphs have been widely used as formal models in the signal processing domain for a long time, and various commercial tools have adopted dataflow semantics for model-based design methodology.

In this thesis, we develop a new dataflow meta-modeling technique, called *homogeneous parameterized dataflow* (*HPDF*). HPDF is a meta-modeling technique in that it can be applied to a variety of underlying dataflow models of computation to enhance their expressive power, while maintaining much of the useful structure of the underlying models. HPDF addresses an important range of applications, especially in the image processing domain. We present various properties and capabilities of HPDF, including the notions of repetitions vector, valid schedule, derivation of looped schedules, single-rate equivalent graphs, and HPDF graph transformation methods. We also give three in-depth examples of complex systems that we have studied to demonstrate the capabilities of HPDF — a gesture recognition application, an image registration application, and a gait-DNA application. For hardware implementation, we target our applications onto Xilinx and Altera field programmable gate arrays (FPGAs), and we present results from the hardware mapping of the gesture recognition and the image registration application.

To build a foundation for further broadening the impact of HPDF modeling, we present initial work on applying cyclo-static dataflow as an intermediate representation for mapping MATLAB programs into hardware implementations. Because of the compatibility between cyclo-static dataflow and the HPDF meta-modeling approach, which we demonstrate in Chapter 3 of this thesis, this is an important first step to exploiting HPDF techniques in the context of MATLAB-to-hardware synthesis. In particular, we focus on relating cyclo-static dataflow to Compaan process networks, which is a variant of the Kahn process network model of computation that has been shown to be useful in representing concurrency in MATLAB programs.

In summary, this thesis develops a useful new meta-modeling approach for implementing an important class of image processing applications, and develops and extensively demonstrates a methodology for efficient hardware implementation from representations in the proposed new meta-model.

MODEL-BASED HARDWARE DESIGN FOR IMAGE PROCESSING SYSTEMS

by

Mainak Sen

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2006

DEDICATION

to my Mother, Father, Brother and Amrita

ACKNOWLEDGMENTS

I want to take this opportunity to thank my advisor, Professor Shuvra Bhattacharyya for all his support and encouragement throughout the course of my graduate studies. Looking back, I feel extremely lucky to have had him straight out of undergraduate from Jadavpur University as I was lost coming to a new country, and getting into unfamiliar surroundings. From that point on, whenever I have tried to set foot into something new, he has always given me the extra time for the learning curve. Without his support, I could not have taken some time out to have my own music band or lead the Electrical and Computer Engineering Graduate Student Association (ECEGSA).

I also want to thank Prof. Rama Chellappa for his support and ideas that has helped me in my Ph.D at various times. Prof. Wayne Wolf at Princeton University has provided us with applications to build our model on and has always obliged to all my requests. Prof. Raj Shekhar has been very supportive and also helped me with applications to test my model on. Prof. Ed Deprettere from Leiden University, The Netherlands has been very helpful with development of a strong collaboration between his research group and our DSPCAD group at UMD and it was a very rewarding experience to work with such an experienced professor in our field. Prof. Gang Qu, Prof. Andre Tits, Prof. Ankur Srivastava, and Prof. Richard La has lent me a patient ear specially towards the end of my Ph.D when time seems to be the flying faster than ever. Prof. Amitabh Varshney has been very kind by agreeing to be the Dean's representative for my committee.

It has been a pleasure to interact with many students in the DSPCAD group and the Embedded Systems Research Lab, including Ming-Yung, Vida, Dong-Ik, Nitin, Neal, Shahrooz, Sumit, Fuat, Sadagopan, Lin, Chung-Ching, Chia-Jui, Sankalita, Sebastian, Celine, and Ruirui.

Ivan and Fiorella needs special mention for working with me. It was a great experience working late nights and sharing our thoughts. The modeling and FPGA implementation of gesture recognition algorithm was the result of joint effort from the three of us. Modeling of image registration algorithm was a joint work with Yashwant and this work would not have been possible in such a short time without his efforts. I would also like to thank Todor Stefanov from Leiden University for our joint work on CPN and dataflow. I want to thank Dr. William Plishker for reviewing some of my papers and part of my thesis. My collaborations and discussions with all of them have strengthened this thesis and enriched my experience here.

I would also like to thank some of my friends here at UMD, many of whom I met for a relatively small period of time but they left a large footprint on my heart. Suvarcha has been a very good friend of mine for a long time and Ankush is one of the best people I have met in my life.

Finally and most importantly, I would like to thank my family for their love, support, and patience during these years while I focused on studies. Thanks to my mom, dad and brother for helping me aim high and provide support in every way possible. My late grandparents who loved me more than anyone else in the whole world. My nieces and nephews have been my source of fun and joy always. Last but certainly not the least, I would also like to thank Amrita for filling my life with laughter and joy and I look forward to spending the rest of my life with her.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

1.1. Overview

Model-based design is rapidly becoming a popular approach in design environments for embedded systems due to high level of complexity in these systems. In model-based design, design representations in terms of formal models of computation (MoC) are used to capture, analyze, simulate, and in some cases, optimize and implement the targeted applications. Detailed simulation of the whole system can cut down on costly changes late in the design phase, which are otherwise not visible at a component level. To overcome this fundamental problem, engineers are moving toward model-based design environments that encompass and support all the major phases of system development: design, simulation, code generation, verification, and implementation, as outlined above.

There are many model-based design tools available, both commercially and from academic sources, for embedded systems design — e.g., LabVIEW from National Instruments, Simulink® from Mathworks, and Ptolemy II from U.C. Berkeley [17], to name a few. However, designing a hardware system through systematic use of a formal model has only recently been emerging as an area of interest not only to the academic world, but also to commercial vendors.

We cite some of the industry efforts made towards model-based hardware code generation, however this list is not comprehensive. We also present some related work done in academia in Section 2.2. First, Synplicity — a leading provider in electronic design automation (EDA) tools, has a model-based hardware design tool named Synplify® DSP, which is built for digital signal processing (DSP) system designers who target field programmable gate arrays (FPGAs) and application specific integrated circuits (ASICs) for implementation of high-performance DSP designs. Synplify® DSP provides an environment for specifying a design at the algorithm level in MATLAB®/Simulink®, and mapping the algorithm representation into an RTL design without the need for multiple iterations between the DSP algorithm architect and the RTL hardware designer as in a traditional design cycle.

LabVIEW FPGA by National Instruments provides a graphical programming environment to define the logic in FPGA chips that are embedded across the family of National Instruments reconfigurable I/O hardware targets. Due to the high level of abstraction used in this tool, designer can design and implement FPGA-based designs without the knowledge of low-level hardware description languages.

Xilinx — a leading provider of FPGAs and related software tools, introduced System Generator™ for DSP, which is another effort to build a high-level tool for designing high-performance DSP systems using FPGAs. The tool provides abstractions that enable the development of parallel systems with advanced FPGAs, providing system modeling and automatic code generation from Simulink® and MATLAB®.

Altera, another leading maker of high-performance FPGAs, has a similar graphical tool for high performance DSP design for FPGAs. This tool, called Altera DSP Builder, is a block-based tool that interfaces between Quartus® II — which is the synthesis tool for Altera FPGAs — and MATLAB®/Simulink® tools.

Recently (September 2006), The Mathworks has introduced a new tool called the Simulink® HDL Coder. It is for system and hardware engineers, letting them adopt model-based design in their development processes for both hardware and software. The tool automatically generates synthesizable hardware description language (HDL) code from its own Simulink® and Stateflow® software. Simulink HDL Coder can produce target-independent Verilog and VHDL code and test benches for implementing and verifying ASICs and FPGAs and later synthesize for their target platforms. Figure 1.1 shows a snapshot of the working of HDL coder with a block-based design window in the middle, simulation results in software on the left, and generated HDL code on the right of the figure.

1.2. Contributions of this Thesis

Our work is similar in spirit to the recent industry trends described in the previous section. This similarity is in the sense that we have a similar motivation of proposing a methodology for generating FPGA implementations from high-level, block- and model-based design environments. However, our work is different in certain respects towards obtaining that goal. First, we employ dataflow models of computation, which are widely used in the DSP community, as the general formal modeling approach to capture high-level abstractions of applications, and we develop a framework through which various

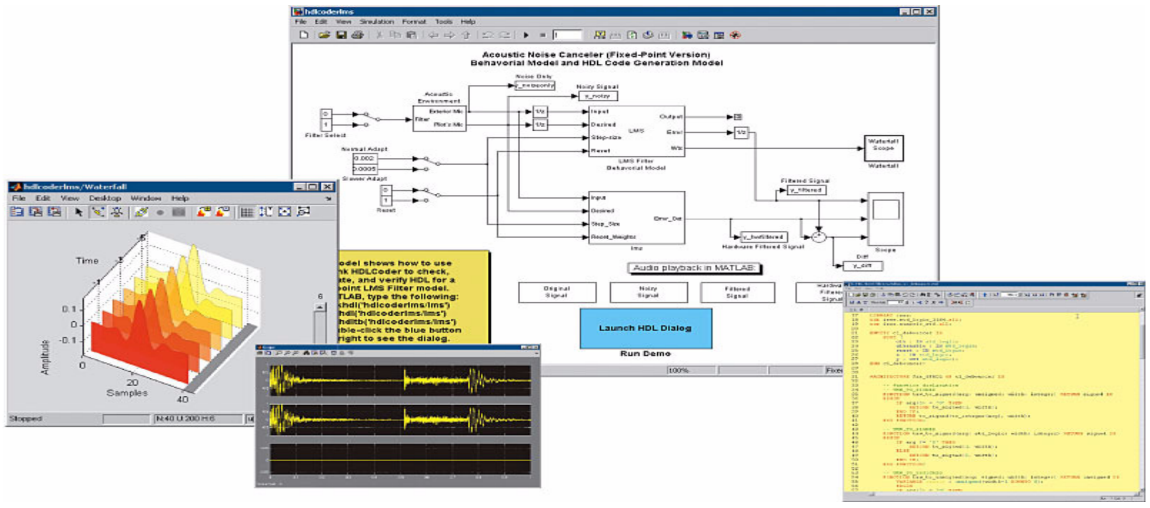

hdlcoderlms
Acoustic Noise Canceler (Fixed-Point Version)
Behavioral Model and HDL Code Generation Model
LMS Filter Behavioral Model
Audio playback in MATLAB
Launch HDL Dialog
Run Demo
hdlcoderlms/Waterfall
Time
Amplitude
Samples

Figure 1.1 Example working of Simulink HDL coder. Figure taken from the Mathworks website.

dataflow graph transformations can be made with high-performance, hardware synthesis in view. Second, we develop a specific dataflow modeling approach that targets a restricted but important class of image processing applications, and using our modeling approach, we develop methods for systematically exploiting properties in the targeted applications to streamline the analysis, synthesis, and optimization of hardware implementations. Thus, this thesis represents a novel convergence of methods involving dataflow modeling, image processing, and hardware implementation.

In the following sections, we elaborate further on the main contributions of this thesis.

1.2.1. Homogeneous Parameterized Dataflow (HPDF)

Static dataflow graphs, such as those based on synchronous dataflow (SDF) [31] or cyclo-static dataflow (CSDF) [6] principles, have been relatively well-studied in the literature. However, many modern signal processing applications are dynamic or data-dependent in nature to some extent, and cannot be fully modeled using static dataflow graphs. We have proposed a new dataflow modeling approach, called homogeneous parameterized dataflow (*HPDF*), that captures a subset of applications with a restricted form of dynamic data production and consumption behavior.

HPDF is a meta-model in that it can be applied to a variety of different underlying dataflow models of computation, such as synchronous dataflow or cyclo-static dataflow. When HPDF is applied to an underlying dataflow model, we refer to the underlying model as the *base model* to which HPDF is applied, and we say that HPDF is being applied "over" the base model (e.g., "HPDF over SDF" or "HPDF over CSDF"). The integration of the HPDF meta model with a base model generally results in a more powerful (more expressive) version of the base model that retains much of the intuitive structure and much of the useful analysis and optimization potential of the base model.

HPDF is intuitive and well-suited to many image processing applications. HPDF is similar to SDF in that it imposes significant restrictions on application structure, and has an inherent simplicity in its core semantics, but captures an important class of applications despite this simple and restrictive nature. to the These characteristics have contributed significantly to the intuitive appeal, general popularity, and utility of SDF, and it is therefore promising that HPDF exhibits this similarity.

In this thesis, we motivate and develop in detail the HPDF meta-modeling technique. We then define and explore useful properties of HPDF. To demonstrate the applicability and capabilities of HPDF, we have develop three in-depth case studies of important image processing applications. These applications include a gait-DNA application for load carrying event detection [42], a gesture recognition application [55], and a 3-D image registration application. For all the three applications, we first employ HPDF over SDF, and then refine our model using HPDF over CSDF so that the meta-modeling aspect of HPDF can be concretely demonstrated.

1.2.2. Graph Transformation - Node Unfolding

In this thesis, we propose a new dataflow graph transformation — *node unfolding*. Node unfolding can be used effectively to explore the design space starting with a dataflow graph representation of an application when the final implementation is targeted towards hardware, especially hardware in which area constraints are relevant. Node unfolding systematically replicates selected nodes in the dataflow graph.

We propose an algorithm to transform an input dataflow graph by repeated application of node unfolding for high-throughput implementation in hardware. The algorithm requires initial estimates of execution time and area for each node in the graph, and it applies maximum cycle mean (MCM) analysis for performance estimation. We also present a preliminary version of a Verilog code generator for dataflow graphs that can be used after we arrive at a suitably-transformed graph to generate HDL code.

1.2.3. HPDF to FPGA implementation

In this thesis, we present extensive demonstrations of FPGA implementations that are derived from HPDF-based representations. Such demonstrations are developed for two applications. First, we demonstrate the mapping of a gesture recognition application onto the Xilinx Multimedia and Microblaze board. This board features a Virtex II FPGA that supports 2 million gates. For the gesture recognition application, we also present an effective method for exploring trade-offs between different memory layout schemes, and we present a thorough floating point optimization study for the application. Next we present an HPDF-based mapping of a 3-D image registration algorithm onto an Altera StratixII FPGA. In the process, we present a study of performance and area trade-offs across a multitude of design points that correspond to various parallel implementations. These parallel implementations can be mapped naturally onto an FPGA from the high-level HPDF specification. We also present a dynamically reconfigurable architecture for the image registration algorithm, and we present a novel parameterization of such an architecture in terms of a metric that is based on the percentage of valid voxels (PVV) that are being considered at a given algorithm iteration. This metric and its utility are developed in detail in Chapter 6.

1.2.4. Dataflow as an Intermediate Representation for MATLAB Synthesis

Sequential, static affine nested loop programs can be automatically converted to input-output equivalent Kahn process network (KPN) specifications [28]. The correspondence between these classes of specifications have derived and exploited in depth by the Compaan project [28]. Such specialized KPNs are also called Compaan process networks (CPNs). Our studies, which have been developed in collaboration with the Compaan project team at Leiden University, have shown that CPNs form a special case of the cyclo-static dataflow graph (CSDF) model [6], but with additional non-dataflow properties that need to be associated to derive a comprehensive correspondence [14].

The dataflow interchange format (DIF) is a standard language for specifying mixed-grain dataflow models for DSP systems [25]. In this thesis, we extend DIF to capture the form of cyclo-static dataflow that emerges from CPNs. Both CPN and CSDF are relatively mature formal models and our preliminary study on their correspondence establishes the potential to exchange information between the two domains to improve the synthesis of hardware and software implementations from MATLAB programs. In particular, the correspondence that we developed provides a bridge between the path from MATLAB to CPNs, that has been developed in the Compaan project [28], with the paths from dataflow representations to hardware that are developed in efforts such as this thesis.

1.3. Outline of this thesis

The rest of this thesis is structured as follows — Chapter 2 gives background information on dataflow, and describes previous related work. Chapter 3 introduces the HPDF meta-model, proves formal properties in relation to this model, and provides a concrete example by modeling a gait-DNA application using the proposed techniques. Chapter 4 introduces a new graph transformation technique named “node unfolding”, its application to HPDF, use of this technique for design space exploration in high-level synthesis techniques for hardware synthesis from dataflow, and preliminary work developed towards designing a Verilog code generator for dataflow graphs. Chapter 5 describes a gesture recognition algorithm, and HPDF modeling and hardware implementation targeting a Xilinx Virtex II FPGA. Chapter 6 describes an image registration algorithm, its HPDF model, dataflow graph transformation based on the applied modeling, and FPGA implementation and comparison of various implementation points for the algorithm. Chapter 7 describes initial efforts towards integrating the Compaan intermediate representation framework with dataflow modeling, and develops extensions to the dataflow interchange format (DIF) [25] that achieve this integration. Chapter 8 gives concluding remarks, summarizes this thesis, and suggests useful directions for future work.

Chapter 2. Background and Related Work

In this chapter, we present some background to understand this thesis and also present some previous work and show their relation to this thesis.

2.1. Dataflow Graphs

In this section, we review background on synchronous dataflow, cyclo-static dataflow, and parameterized dataflow modeling, and describe various basic definitions related to signal-processing-oriented dataflow modeling. The concepts reviewed in this section will be applied later in the thesis in our formal development of HPDF.

2.1.1. Synchronous Dataflow

Synchronous dataflow (SDF) is a restricted form of dataflow in which data production and consumption rates of actor ports (inputs and outputs) are restricted to be constant values that are known at compile time [31]. This restriction enables static scheduling from SDF representations, and offers strong compile-time predictability properties, and powerful optimization techniques, such as joint minimization of program and data memory requirements [5]. These features come at the expense of limited expressive power, since SDF cannot model data production rates that vary dynamically or are otherwise unknown at compile time.

SDF is employed in a variety of widely-used commercial design tools, such as CoWare SPW, Agilent ADS, and National Instruments LabVIEW.

An example of a simple SDF graph is in shown in Figure 2.1. Here, each edge is annotated with the numbers of tokens that are produced by the source and sink actors of the edge. For example, actor A produces 3 tokens on edge (A, B) every time it is invoked, and each invocation of C results in 1 token being consumed from the edge (B, C). The description of $2D$ is given in Section 2.1.3.

SDF graphs — and signal-processing-oriented dataflow graphs in general — typically represent computations that are iterated infinitely or for indefinite time (e.g., due to the absence of apriori bounds on the durations of the input streams). Thus, each actor generally corresponds to an infinite number of invocations in an execution of the graph.

2.1.2. Cyclo-static dataflow

Cyclo-static dataflow (CSDF) is an extension of SDF where production and consumption rates of actors can vary as long as the variations take the form of periodic sequences that are known at compile time. Given a CSDF actor A, a finite sequence is associated with the production rate of each output edge, and with the consumption rate of each input edge. These sequences associated with inputs and outputs of A all have the same length, and correspond to a single period of the interface (data transfer) behavior of A. The ith element of each sequence corresponds to a distinct *phase* of execution for A.

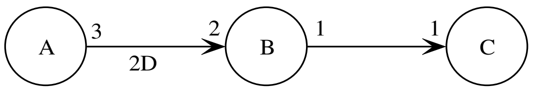


Figure 2.1 A simple SDF graph.

A simple example of a CSDF graph is presented in Figure 2.2. Each actor input/output is annotated with the associated sequence of production/consumption rates. For example, actor A has only one phase, and produces 3 tokens on edge (A, B) every time it executes. Actor B has two phases, and the amounts of tokens consumed and produced by successive invocations of B form the periodic patterns $(1, 1, 1, 1, \ldots)$, and $(1, 0, 1, 0, \ldots)$, respectively.

2.1.3. Scheduling Concepts

In this section, we review some basic SDF- and CSDF-related scheduling concepts and notations that are used throughout the rest of the thesis.

Dataflow notation. Given an edge e in a dataflow graph, the source and sink actors of e are denoted by $src(e)$ and $snk(e)$, respectively. Given an SDF edge e, the number of tokens produced on e by each invocation of $src(e)$ is denoted by $p(e)$, and the number of tokens consumed from e by each invocation of $snk(e)$ is denoted by $c(e)$. Given a CSDF actor A, the number of phases associated with A is denoted by $\phi(A)$. Given a CSDF edge e, the number of tokens produced by $src(e)$ onto e in the ith phase of $src(e)$ is denoted by $P(e, i)$, and similarly, the number of tokens consumed by $snk(e)$ from e in the ith phase of $snk(e)$ is defined by $C(e, i)$. Clearly, $P(e, i)$ is defined for $i = 1, 2, \ldots, \phi(src(e))$, and $C(e, i)$ is defined for $i = 1, 2, \ldots, \phi(snk(e))$.

Delays and buffer state. In dataflow models for signal processing, edges can have non-unity *delays* associated with them. One unit of delay is analogous to the z^{-1} operator

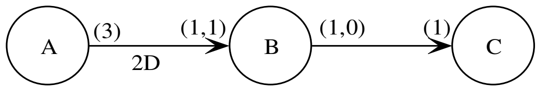


Figure 2.2 A simple CSDF graph.

in signal processing, and can be implemented by placing an initial token on the associated edge. The *buffer state* of an SDF graph at a given point in time t is an integer-vector b that is indexed by the graph edges such that $b(e)$ gives the number of tokens that reside on e at t. Since delays can be implemented as initial tokens, we define the initial buffer state of a graph to be $b(e) = delay(e)$, where $delay(e)$ denotes the delay on edge e.

Topology Matrix — The *topology matrix* (denoted by Γ) is used to represent the dataflow characteristics of an SDF graph G [31]. The rows of Γ are indexed by the edges in G, and the columns are indexed by the actors in G. The entries of Γ are defined by

$$\Gamma(e, A) = \begin{cases} p(e);\ \textit{if } A = \textit{src(e)} \\ -c(e);\ \textit{if } A = \textit{snk(e)} \\ 0;\ \textit{otherwise} \end{cases} \tag{2.1}$$

For example, the topology matrix for the SDF graph in Figure 2.1 can be written as

$$\Gamma = \begin{bmatrix} 3 & -2 & 0 \\ 0 & 1 & -1 \end{bmatrix}, \tag{2.2}$$

where the rows correspond to the edges (A, B) and (B, C), respectively, and the columns correspond to the actors A, B, and C, respectively.

The topology matrix for a CSDF graph is defined effectively by replacing $p(e)$ and $c(e)$ in (2.1) with the sums of $P(e, i)$ and $C(e, i)$, respectively, across all relevant phases. More precisely, the entries of the topology matrix Γ_{CSDF} for a CSDF graph can be expressed as

$$\Gamma_{CSDF}(e, A) = \begin{cases} \left(\sum_{i=1}^{\phi(A)} P(e, i)\right) & \textit{if } A = src(e) \\ -\left(\sum_{i=1}^{\phi(A)} C(e, i)\right) & \textit{if } A = snk(e) \\ 0 & \textit{otherwise} \end{cases} . \qquad (2.3)$$

Hence, the topology matrix for the CSDF graph in Figure 2.2 is the same as the topology matrix for the SDF graph in Figure 2.1.

Repetitions vector and *valid schedules* — The *repetitions vector* (usually denoted by q) for an SDF graph is a vector of co-prime positive integers that denotes the number of times that each actor in the graph is executed in a minimal valid schedule for the graph. A valid schedule S in turn is a finite sequence of actor invocations that fires each actor at-least once, does not deadlock (i.e., does not attempt to consume data from an empty buffer), and produces no net change in the buffer state of the graph (i.e., execution of S returns the graph to its initial buffer state). The repetitions vector, when it exists, can be determined by solving for the $n_a \times 1$ column vector x in the system of *balance equations* defined by

$$\Gamma x = 0 . \qquad (2.4)$$

In particular, the repetitions vector is defined to be the minimum positive integer solution to (2.4). It can be shown that such a unique minimum positive integer solution exists whenever (2.4) has a nontrivial solution [31].

For example, the repetitions vector for the SDF graph in Figure 2.1 is given by $q(A) = 2$, and $q(B) = q(C) = 3$.

An example of a valid schedule for this graph is given by $(BABABCCC)$.

The main practical significance of a valid schedule is that it can be iterated indefinitely to achieve unbounded-duration execution of the given graph with bounded buffer memory requirements.

For CSDF graph, the concept of a valid schedule is the same as that for an SDF graph; however, the process of computing the numbers of actor invocations involved in a valid schedule is slightly more involved. The system of balance equations for a CSDF graph is given by

$$\Gamma_{CSDF} x = 0, \tag{2.5}$$

where x again represents an $n_a \times 1$ column vector.

A solution to the CSDF balance equations, when it exists, gives the number of "actor periods" for each actor that is involved in an iteration of a valid schedule. Here, an actor period for actor A corresponds to execution of $\phi(A)$ successive invocations (phases) of A. The CSDF repetitions vector — which gives the number of actor invocations for each actor in a valid schedule — is thus obtained from

$$q_{CSDF}(A) = \tau(A)\phi(A) \text{ for all } A, \tag{2.6}$$

where τ is the minimum positive integer solution to (2.5).

For example, for the CSDF graph in Figure 2.2, we have $\tau(A) = 2$, $\tau(B) = \tau(C) = 3$, $q_{CSDF}(A) = 2$, $q_{CSDF}(B) = 6$, and $q_{CSDF}(C) = 3$. Furthermore, the schedule $(BBABBABBCCC)$ is a valid schedule for this CSDF graph.

Consistent graph — An SDF graph or CSDF graph is said to be *consistent* if it has a valid schedule. Intuitively, consistency in this context means that the balance equations ((2.4) or (2.5)) have a nontrivial solution, and all directed cycles in the graph have enough delays (initial tokens) to allow for deadlock-free execution [31].

The graphs in Figure 2.1 and Figure 2.2 are both consistent.

2.1.4. Parameterized Dataflow

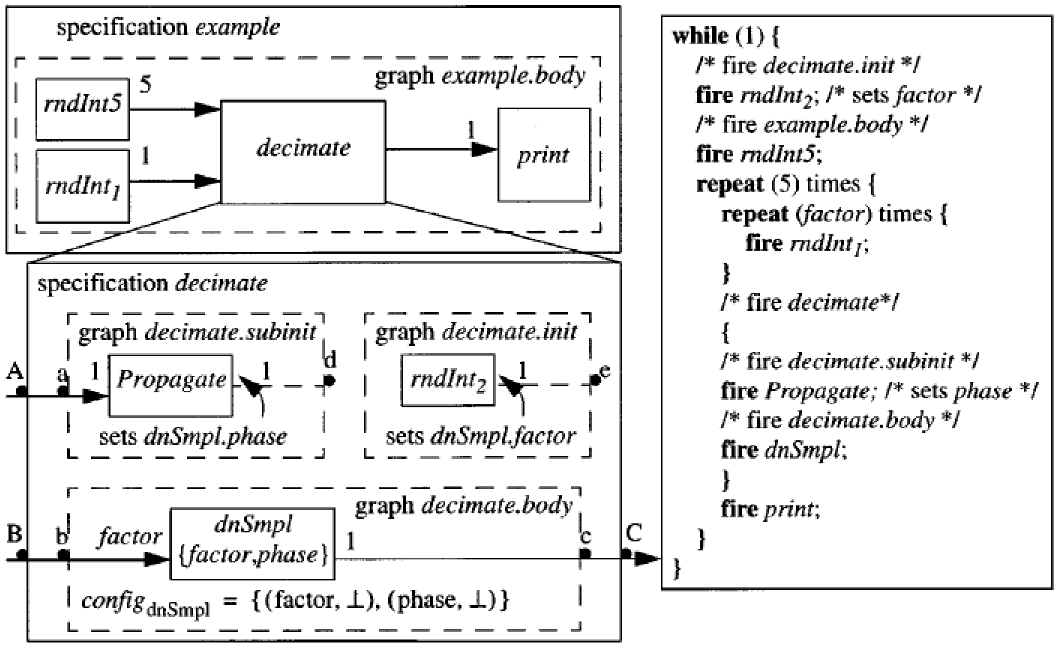
Parameterized dataflow [3] is a meta-modeling technique that can be used in conjunction with any dataflow model of computation that has a well-defined notion of a graph *iteration*. For example, in SDF and CSDF graphs, a graph iteration usually corresponds to execution of a valid schedule. When parameterized dataflow is applied to a dataflow model of computation D, the model D is called the *base* model, and the resulting integrated model can be viewed as a dynamically reconfigurable augmentation of D. Thus, parameterized dataflow provides for increased expressive power by allowing for run-time reconfigurability of actor and edge parameters in a certain structured way.

When parameterized dataflow is applied to SDF as the base model, the resulting model of computation is called parameterized synchronous dataflow (*PSDF*). An actor A in PSDF is characterized by a set of parameters ($params(A)$) that control the actor's functionality, including possibly its dataflow behavior. Each parameter is either assigned a value from a set of permissible values or left unspecified. These unspecified parameters are assigned values at run-time through a disciplined run-time reconfiguration mechanism. Techniques have been developed to execute PSDF graphs efficiently through carefully constructed quasi-static schedules [3].

Parameterized dataflow specifications are built up in a modular way in terms of hierarchical subsystems. Every subsystem is in general composed of three subgraphs, called the *init*, *subinit* and *body* graphs. New parameter values used during run-time reconfiguration are generally computed in the init and subinit graphs, and the values are propagated to the body graph, which represents the computational core of the associated parameterized dataflow subsystem. The init graph for a subsystem H is invoked at the beginning of each invocation of the (hierarchical) parent graph of H. In contrast, the sub-init graph is invoked at the beginning of each invocation of H itself, prior to execution of the body graph. Intuitively, reconfiguration of a body graph by the corresponding init graph occurs less frequently but is more flexible compared to reconfiguration by the sub-init graph [3].

2.1.5. Generic Model for Hierarchical Reconfiguration of Dataflow Graphs

Parameterization is a widely-used method to implement dynamic behavior of a dataflow graph. But a parameterized actor might also have a predetermined production and consumption rate. For example, an FIR filter might have its number of taps as a parameter, which does not affect the production consumption rate. In this thesis, we discuss parameters in the context of actors whose token production and consumption rates are a function



```
while (1) {
  /* fire decimate.init */
  fire rndInt2; /* sets factor */
  /* fire example.body */
  fire rndInt5;
  repeat (5) times {
    repeat (factor) times {
      fire rndInt1;
    }
    /* fire decimate*/
    {
    /* fire decimate.subinit */
    fire Propagate; /* sets phase */
    /* fire decimate.body */
    fire dnSmpl;
    }
    fire print;
  }
}
```

Figure 2.3 PSDF specification of a decimate actor that decimates by a different factor at each run (figure from [3]).

of these parameters. In [38], the authors develop a mathematical model to represent the reconfiguration of various types of dynamic dataflow graphs. The model allows reconfiguration at all levels of hierarchy. A hierarchical reconfiguration model is represented by a *containment tree,* which has a finite set of actors in it. Non-leaf nodes are composite actors and leaf elements are atomic actors. The behavior of a composite actor is given by the actors that are its direct children. Every actor has its own set of parameters which define its behavior and there is a one-to-one relation between the parameters and actors. Dependencies among parameters are expressed explicitly through a domain function and its value is constrained by a *constraint function*. A dependent parameter must at all times satisfy the constraint function to become *consistent*. An independent parameter has null in its domain function.

The authors introduce specific points in their model called *quiescent points*, which are constrained points in the execution model where change of parameter values are permitted. These points occur between firings and an actor cannot communicate or perform computation at these points. A *precedence relation* is set that performs partial ordering of quiescent points of all the actors. At each quiescent point, a set of independent parameters Q is chosen for reconfiguration and all the parameters dependent on Q are also reconfigured based on their initial and reconfigured values. Parameters that cannot be reconfigured or can be changed only at certain quiescent points are declared as *constant parameters*. A constant parameter can be forced to remain constant either during one particular execution of the model or over firings of the associated actor. To statically analyze the reconfiguration of a model, two methodologies have been suggested. Firstly, all the executions of the model are checked along with all possible reconfigurations and any invalid reconfiguration predicts invalidity of the model. Secondly, the authors suggest a least change context for every parameter p which is a conservative estimate of the actors affected by p. This helps in easy semantic constraint checking.

2.2. Related Work

2.2.1. Hardware from Formal Models

A number of studies have been undertaken in recent years on the design and implementation of multimedia applications on FPGAs using other formal or systematic approaches.

Streams-C [19] developed at the Los Alamos National Laboratory, USA provides compiler technology that maps high-level, *parallel C* language descriptions into circuit-level netlists targeted to FPGAs. To use Streams-C effectively, the programmer needs to have some application-specific hardware mapping expertise as well as expertise in parallel programming under the CSP (communicating sequential processes) model of computation [22]. Streams-C consists of a small number of libraries and intrinsic functions added to a subset of C that the user must use to derive synthesizable HDL code.

Handel-C [11] developed at the Oxford University, UK represents another important effort towards developing a hardware oriented C language. Handel-C is based on a subset of the ANSI C standard along with extensions that support a synchronous parallel mode of operation. It supports specification of width of variables, and consequently has strong bit manipulation capabilities. This language also conforms to the CSP model. A canny edge detector was designed in hardware in [37] using the Celoxica DK2 IDE tool — which is the development tool for Handel-C.

Match [1] or AccelFPGA as it is called now, generates VHDL or Verilog from an algorithm coded in MATLAB, a programming language that is widely used for prototyping image and video processing algorithms. AccelFPGA has various compiler directives that the designer can use to explore the design space for optimized hardware implementation. Loop unrolling, pipelining, and user-defined memory mapping are examples of implementation aspects that can be coordinated through AccelFPGA directives.

Compaan [28] is a another design tool for translating MATLAB programs into HDL for FPGA implementation. Compaan performs its translation through an intermediate representation that is based on the Kahn process network model of computation [27]. Compaan can either generate an embedded software code to run on the softcores (for example PowerPC on Virtex II Pro) or it can generate output in the form of executable Kahn Process Networks for another tool named Laura [56]. Laura accepts this specification and transforms the specification into design implementations described in synthesizable VHDL.

Rather than adapting a sequential programming language for hardware design, as the above-mentioned approaches do, our approach is based on concurrency exposed by the designer in representing the algorithm as a dataflow model. This is a useful approach for signal processing because the structure of signal processing applications in terms of its coarse-grain components (e.g., FIR filters, IIR filters, and FFT computations) often translates intuitively into concurrent specifications based on dataflow principles.

2.2.2. Hardware from SDF

A SDF based digital hardware design for embedded signal processing was addressed in [54]. Two techniques were presented for architecture generation — one is a general resource sharing technique for flexibility, and the other is a mapping of sequenced groups for compact communication and interconnect. The problem addressed was to find the minimum-cost hardware to meet the deadline for the time to execute all firings in the SDF dataflow (for this work, a firing precedence graph was constructed from the schedule of the SDF on which the hardware mapping was done). After the schedule of the SDF was generated, each individual firing (execution) of a node j in the graph was associated with a hardware cost C_j and execution time T_j which were estimates from the RTL (register transfer level) synthesis results of any standard synthesizer. Hence the goal was to minimize $\sum C_j$ for the schedule when $\sum T_j < T_{deadline}$. The two heuristics mentioned in the work, approach the same problem from the two opposite directions. In one approach, the authors start with maximum hardware — which is a separate hardware unit for every firing in the firing precedence graph and then try to cluster firings into shared hardware units until no more clustering can be done without violating the deadline. Clustering (which means shared resource allocation) was done based on the following criteria — two actors having the same firing (identical computation of the same SDF actor without any constraints on sequential firings) can be merged to execute on the same hardware; two actors having similar firings (similar computations, so that much of the execution unit can be the same hardware and no constraints on sequential firing) can be merged; actors differing only by a parameter can be merged too. If the inputs of the merged actors are from different actors, then a multiplexer is needed and controller logic needs to be added for correct execution. In the other approach, the whole graph is mapped onto a single hardware unit, which is capable of performing all the required functionalities — this would result in lowest area implementation as the standard hardware synthesizers are able to optimize the circuit the best. However, this would act like a uniprocessor with every node firing sequentially, thus not taking advantage of any parallelism that might be present in the application. However, there are various design point in between the two extreme implementations that can be explored using clustering or declustering techniques.

Our approach is different from the approach in [54], in respect that due to inherent simplicity of HPDF, we try to explore the architecture from the dataflow graph instead of the firing precedence graph and our approach can handle limited dynamicity in the application which the SDF based approach cannot. Also the clustering techniques are orthogonal to our approach and hence can be used to enhance our method of hardware development.

2.2.3. Image Processing in Hardware

Computer vision algorithms can be divided into three categories depending on the level of granularity at which they are specified — low level, intermediate level and high level algorithms. In the late 1980s and early 90s, a lot of work was done on image processing on hardware. Some of it was on homogeneous/heterogeneous architectures specially suited for image processing, and some of it was on specialized algorithms suited for hardware implementations. The general observation was that single instruction multiple data (SIMD) machines were good for exploiting fine-grained parallelism and multiple instruction multiple data (MIMD) machines were suitable for coarse-grained parallelism. In this section, we mention a few instances of the work done in the hardware architecture for computer vision domain. A more detailed description of all the architectures and some others not mentioned here can be found in [16], [40].

2.2.3.1. MESH Architecture

MESH is a SIMD 2D array architecture (array of processors) for computer vision. This architecture is suited to perform low and intermediate level vision algorithms. One major problem with such a major system is fault in final architecture due to fabrication variabilities. MESH has a fault-tolerant strategy to enhance yield and improve reliability. MESH has a hardware reconfiguration strategy to eliminate defective processors in combination with data reconfiguration to redistribute the problem over the working processors. We give an example of a *16* processing element MESH architecture in Figure 2.4.

2.2.3.2. Hypercube Architecture

A hypercube also describes a SIMD connection. It differs from a MESH connection in the way connections are made between different processing elements (PEs). A p-dimensional hypercube network connects 2^p processing elements. A hypercube network connects pairs of processing elements whose indices differ in exactly one bit when expressed in the binary representation. We represent the hypercube network in Figure 2.5.

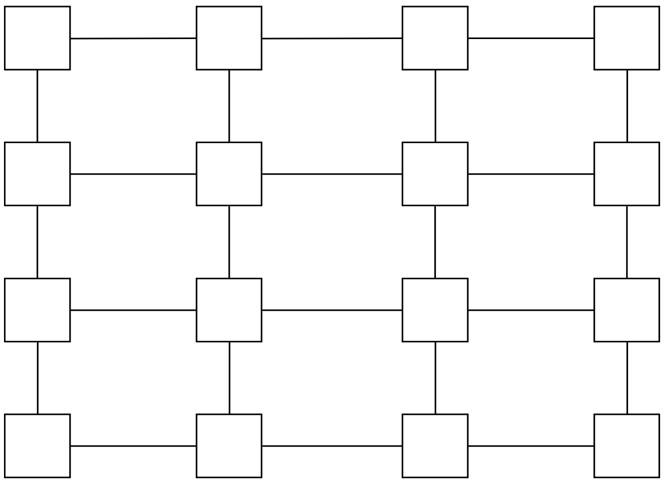


Figure 2.4 A 16 processing element (PE) MESH architecture where end-around connections are not shown.

2.2.3.3. NETRA

NETRA is a highly configureable architecture for image understanding. The topology of NETRA is recursive and hence easily scalable. It has a tree-type hierarchical architecture with leaf nodes consisting of small but powerful processor clusters connected by crossbar switches — the tree has distributing and scheduling processors that perform the task distribution. Each processor cluster has *16* to *64* processing elements with both shared and distributed memory. Each of the clusters can operate in SIMD mode, MIMD mode or systolic mode, and each processing element is a general purpose processor with high-speed floating point capabilities.

2.2.3.4. IUA (Image Understanding Architecture)

IUA is a 3-tier architecture with a dedicated architecture for each level of abstraction (low level, intermediate level and high level vision algorithms). Main processor languages were added with extensions to provide several levels of parallelism, each requiring a unique level of overhead.

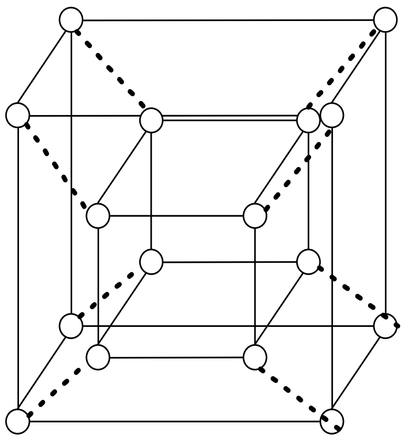


Figure 2.5 *16* processor hypercube architecture.

2.2.3.5. Warp

Warp was a medium grain systolic array machine built at Carnegie Melon University (CMU). There are a few versions of this machine — WWWarp consists of a linear array of *10* cells, each giving *10* MFLOPS with a total of *100* MFLOPS; PCWarp is an extension to WWWarp with capabilities of *160* MFLOPS and with larger cell data and program memory; iWarp (integrated Warp) which was a joint venture between CMU and Intel had capabilities of *16* MFLOPS per cell as a result of faster clock with a linear array of *72* cells, giving a total *1.152* GFLOPS.

Chapter 3. Homogeneous Parameterized Dataflow Graph (HPDF)

Real-time multimedia applications are an integral part of embedded systems technology. Modeling such applications using dataflow graphs can lead to useful formal properties, such as bounded memory requirements, and efficient synthesis solutions (e.g, see [4]). The synchronous dataflow (SDF) model for example has particularly strong compile time predictability properties [31]. However, this model is highly restrictive and cannot handle data-dependent execution of dataflow graph vertices (*actors*). There have been previous studies on extensions of SDF to provide for more flexible actor execution, including handling of such dynamic execution capabilities. For example, a cyclo-static dataflow (CSDF) [6] graph can accommodate multiphase actors with different consumption and production rates at the input and output, respectively, at different phases of iteration. This provides for more flexibility but does not permit data dependent production or consumption patterns. Another extension known as the token flow model [7] was proposed in which we can have dynamic actors where the number of data values (*tokens*) transferred across a graph edge may depend on the run-time value of a token that is received at a "control port" of an incident actor. A meta-modeling technique called parameterized dataflow [3] (PSDF) was proposed later in which dynamic dataflow capability was formulated in terms of run-time reconfiguration of actor and edge parameters. In this chapter, we present another model HPDF which can model certain restricted forms of dynamic dataflow very effectively and is more constrained compared to PSDF.

3.1. Model Definition

In this section, we first provide a more constrained definition of HPDF that was presented at some of our initial work. We present the characteristics of the actors, edges, and delay buffers in an HPDF graph.

An HPDF subsystem is homogeneous in two ways. First, unlike general SDF graphs and other multirate models, the top level actors in an HPDF subsystem execute at the same rate. Second, unlike the hierarchically-oriented parameterized dataflow semantics, reconfiguration across subsystems can be achieved without introducing hierarchy (i.e., reconfiguration across actors that are at the same level of the modeling hierarchy). Some dynamic applications are naturally non-hierarchical, and this kind of behavior can be modeled using HPDF without imposing "artificial" hierarchical structures that a parameterized dataflow representation would entail. At the same time, hierarchy can be used within the HPDF framework when it is desired.

HPDF is a meta modeling technique. Composite actors in an HPDF model can be refined using any dataflow modeling semantics that provide a well-defined notion of subsystem iteration. For example, the composite HPDF actor might have SDF, CSDF, PSDF or multi-dimensional SDF [32] actors as its constituent actors.

As with other many other dataflow models, such as SDF and CSDF, an HPDF edge e can have a non-negative integer delay $\delta(e)$ on it. This delay gives the number of initial data samples (tokens) on the edge. The stream of tokens that is passed across an edge needs markers of some kind to indicate the "packets" that correspond to each iteration of the producing and consuming actors. An end-of-packet marker is used for this purpose in our implementation.

Interface actors in HPDF can produce and consume arbitrary amounts of data, while the internal connections must, for fixed parameter values, obey the constraints imposed by the base model. An HPDF source actor in general has access to a variable number of tokens at its inputs, but it obeys the semantics of the associated base model on its output. Similarly, an HPDF sink actor obeys the semantics of its base model at the input but can produce a variable number of tokens on its output. HPDF source and sink actors can be used at subsystem interfaces to connect hierarchically to other forms of dataflow.

3.2. An Extended Model Definition

In this section, we present, a generalized form of our proposed homogeneous parameterized dataflow (HPDF) model of computation [44][46][20], which is an extension to the definition presented in Section 3.1. and we build on SDF scheduling fundamentals to present, a precise formalization static scheduling concepts for HPDF.

Like parameterized dataflow, HPDF is a meta-modeling technique that can be applied to different dataflow models, including SDF and CSDF. In our generalized form of HPDF, we restrict the homogeneity constraint so that it is required only for edges whose production or consumption rates involve parameter values that can vary dynamically (e.g., parameterized scalar rates in the case of HPDF-SDF or parameterized vector-rates in the case of HPDF-CSDF).

Henceforth in this thesis, by *HPDF* we mean the generalized form of HPDF that we develop in this section, as opposed to the original, more restricted, form introduced in Section 3.1. [44][46].

3.2.1. Definition of HPDF

An HPDF subsystem is homogeneous in the sense that parameterized edges (in particular, edges that are associated with dynamically variable parameter values) have identical rates of production and consumption for any given iteration of the underlying base model. Thus, the production rate associated with any given edge can change from one base model iteration to the next provided the consumption rate of that edge changes in exactly the same way.

For example, let e be an edge in an HPDF-SDF graph G — that is, a dataflow graph in which HPDF is applied to SDF as the base model. Furthermore, in a given execution of the G, let $p_{HPDF}(e, i)$ denote the (constant) production rate associated with e during the ith iteration of G, and similarly, let $c_{HPDF}(e, i)$ denote the consumption rate associated with e during the i iteration of G. Then the HPDF meta-model imposes the restriction that either 1) $p_{HPDF}(e, i)$ and $c_{HPDF}(e, i)$ remain constant for all i (although the constant value for $p_{HPDF}(e, i)$ may differ from the constant value for $c_{HPDF}(e, i)$), *or* 2) $p_{HPDF}(e, i) = c_{HPDF}(e, i)$ for all i. Here. condition 1 simply means that $p_{HPDF}(e, i)$ and $c_{HPDF}(e, i)$ are both independent of i. An edge that satisfies condition 2 but does not satisfy condition 1 is called a *dynamic edge* of the enclosing HPDF graph.

An example of an HPDF graph is shown in Figure 3.1. Here, the base model is SDF, and p is a symbolic placeholder for a parameter value that is not statically known and that

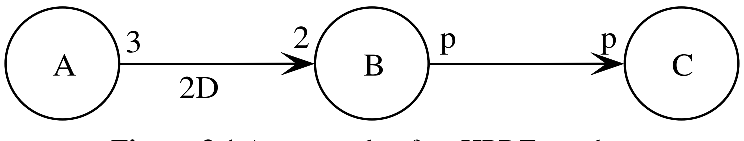


Figure 3.1 An example of an HPDF graph.

can vary dynamically from one base model iteration to the next. Thus, $p_{HPDF}((A, B), i) = 3$ for all i; $c_{HPDF}((A, B), i) = 2$ for all i; and

$p_{HPDF}(e, i) = c_{HPDF}(e, i) = p_i$ for all i,

where p_i represents the value of the parameterized expression p throughout the ith base model iteration and e represents the edge between B and C.

Interface actors in HPDF can produce and consume arbitrary amounts of data from interface edges — the constraints imposed by the HPDF meta-model in conjunction with the given base model need only be satisfied for the internal connections of an HPDF graph. Here, by an *interface actor* of an HPDF graph G, we mean an actor that is connected to one or more components that are outside of G, and by an *interface edge*, we mean an input or output edge of an interface actor that provides such an external connection. An HPDF source actor is an interface actor that has one or more input edges that are interface edges, but conforms to HPDF semantics on its output edges. Similarly, an HPDF sink actor conforms to HPDF on its input edges, and has one or more output edges that are interface outputs.

3.2.2. HPDF with CSDF as the Base Model

We now demonstrate the integration of CSDF base model semantics into the HPDF meta-modeling framework. This integration provides simultaneous application of the bounded memory, dynamic parameterization of HPDF and the finer granularity, phased decomposition of actor execution in CSDF.

As mentioned in Section 3.2.1., the homogeneity requirement in HPDF is in the sense that data transfer across a parameterized edge (production and consumption) must be equal (but not necessarily constant or statically-known) across corresponding invocations of the source and sink actors. In CSDF, a complete invocation of an actor involves execution of all of the phases in a fundamental period of the actor [6]. Integration of CSDF with HPDF allows the number of phases in a fundamental period to vary dynamically, and also allows the number of tokens produced or consumed in a given phase to vary dynamically. Such dynamic variation must adhere to the general HPDF constraint, however, that the total number of tokens produced by a source actor of a given parameterized edge in a given invocation (which, in the case of phased actors, means a given fundamental period) must equal the total number of tokens consumed by the sink in its corresponding invocation. Thus, for all positive n, the number of tokens produced by the nth complete invocation of a source actor must equal the number of tokens consumed by the nth complete invocation of the associated sink actor when they are connected by a parameterized edge.

For fundamental periods that involve dynamic token transfer, this can be accommodated by employing a special token that delimits the end of a fundamental period of a source actor. The source actor produces this special end-of-invocation (*EOI*) delimiter just after the end of each complete invocation. The HPDF restriction then requires the following.

Suppose that the sink actor of a dynamically parameterized HPDF edge e consumes the last token in its ith invocation (fundamental period of phases) at time $z_i(t)$. Then just after completing $\delta(e)$ more consumption operations after time $z_i(t)$, the sink actor will consume an EOI token, and it will not consume any EOI tokens before that. This pattern must hold for all positive integers i (i.e., all invocation indices); that is, after each complete sink invocation, the next EOI token is consumed after exactly $\delta(e)$ consumption operations. Furthermore no EOI token should be consumed during the first invocation $(i = 1)$ of the sink actor.

The above formulation is useful for precisely specifying how HPDF applies to dynamic parameterization of CSDF actors. The formulation can also be used to generate code for quasi-static schedules, and to verify consistency of HPDF specifications at run-time (i.e., to detect violations of HPDF behavior as soon as they occur).

3.3. Comparison of HPDF and PSDF

While HPDF employs parameterized actors and subsystems like PSDF, there are several distinguishing features of HPDF in relation to PSDF. For example, unlike PSDF, HPDF always executes in bounded memory whenever the component models execute in bounded memory. In contrast, some PSDF systems do not execute in bounded memory, and in general, a combination of static and run-time checks is need to ensure bounded memory operation for PSDF [5].

Also, as described in Section 3.2.1, we do not have to introduce hierarchy in HPDF to account for dynamic behavior of actors. For example, suppose that a dynamic source actor A produces n tokens that are consumed by the dynamic sink actor B. In PSDF, we need to have A and B in different subsystems; the body of A would set the parameter n, which will be a known quantity at that time, in the subinit of B (see Section 5.2.1 for a more detailed example). This hierarchy can be avoided in HPDF as we assume that data is produced and consumed in same-sized blocks. As we will describe further in Chapter 5, this simple form of dynamicity has many applications in image processing algorithms. It therefore deserves explicit, efficient support as provided by HPDF.

Also, unlike parameterized dataflow, the stream of tokens that is passed across a dynamic HPDF edge requires markers of some kind to delimit the “packets” that correspond to successive invocations of the producing/consuming actors. An end-of-packet marker is used for this purpose in our implementation.

In summary, compared to PSDF, HPDF provides for simpler (non-hierarchical) parameter reconfiguration, and for more powerful static analysis. In exchange for these features, HPDF is significantly more narrow in the scope of applications that it is suitable for. Intuitively, a parameterized multirate application cannot be modeled using HPDF. However, as we motivate in this thesis, HPDF is suitable for an important class of computer vision applications, and therefore it is a useful modeling approach to consider when developing embedded hardware and software for computer visions systems.

3.4. Scheduling of HPDF Graphs

3.4.1. Repetitions Vectors and Valid Schedules

When HPDF is applied to SDF or CSDF, the *topology matrix* Γ_{HPDF} for an HPDF graph can be defined in manner analogous to the definition of the topology matrix for its base model, with symbolic placeholders used to represent production rate values and consumption rate values that are not statically known. For HPDF-SDF, such a symbolic placeholder represents an unknown scalar value. For HPDF-CSDF, such a placeholder corresponds to symmetric production/consumption-rate tuples that are equal, but in general have variable lengths (numbers of phases), and variable values across CSDF iterations.

For example, the topology matrix for the HPDF graph in Figure 3.1 can be written as

$$\Gamma_{HPDF} = \begin{bmatrix} 3 & -2 & 0 \\ 0 & p & -p \end{bmatrix}.$$

Valid schedules and repetitions vectors can be defined for HPDF graphs in a manner similar to the corresponding concepts that are reviewed in Section 2.1.3. The *repetitions vector* denoted by q_{HPDF}, for an HPDF graph is a vector of co-prime integers that denotes the numbers of times the actors in the HPDF graph should be executed in a minimal base model iteration so that there is no resultant change in buffer state. For example, the repetitions vector for the HPDF graph in Figure 3.1 can be expressed as

$$q_{HPDF}(A) = 2, q_{HPDF}(B) = q_{HPDF}(C) = 3.$$

As in Chapter 2, we can derive a valid schedule from the repetitions vector for an HPDF graph. A valid schedule for Figure 3.1 is given by $(BABABCCC)$.

In Sections 3.4.2 and 3.4.3, we apply HPDF with SDF as the base model and show how existing methods for static scheduling that are based on SDF can be extended systematically to HPDF.

3.4.2. SDF Reductions of HPDF Graphs

Based on the concept of symbolic placeholders described in Sections 3.2.1 and 3.4.1, an HPDF graph contains a set of dynamic-parameter edges $e_1, e_2, \ldots, e_n$ such that the for each e_i, the production and consumption rate values of e_i are equal in any given base model iteration, and this common value of dynamically-varying production/consumption rate is represented by a symbolic placeholder p_i.

Given an HPDF-SDF graph G, the SDF reduction of G is an SDF graph G' that is derived by replacing each p_i associated with G by the constant production and consumption rate value of 1 for the associated edge. That is,

$$p(e_i) = c(e_i) = 1 \text{ for } i = 1, 2, \ldots, n.$$

For example, the SDF reduction of the HPDF graph of Figure 3.1 is the SDF graph in Figure 2.1.

SDF reductions are useful because important scheduling-related operations on HPDF-SDF graphs can be reduced to scheduling operations on the corresponding SDF reductions.

For example, the repetitions vector for HPDF-SDF graphs is well defined and has a similar interpretations as with SDF graphs — each element of the HPDF-SDF repetitions vector gives the number of times to execute the corresponding actor in a minimal valid schedule. Given an HPDF-SDF graph G, its repetitions vector can be derived simply by

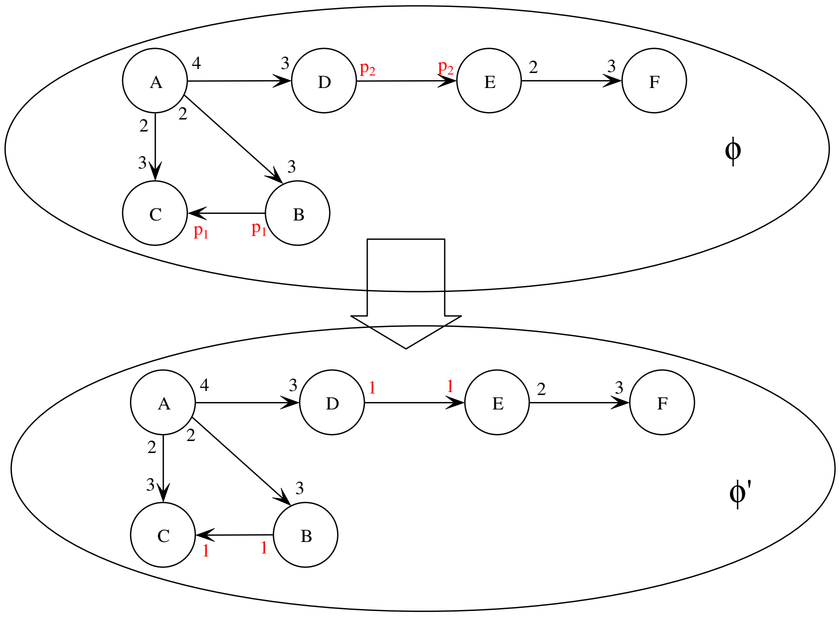


Figure 3.2 Converting ϕ to its SDF reduction ϕ' by replacing parameters p_1 and p_2 with the value 1.

determining the SDF reduction G' of G, and computing the repetitions vector of G'. In other words, the repetitions vector of an HPDF-SDF graph, when it exists, is equal to the repetitions vector of the corresponding SDF reduction. Intuitively, this is true because in terms of the balance equations, each dynamic edge e in an HPDF graph functions like an SDF edge that has equal-valued production and consumption rates — that is, the constraint imposed by such an edge on the balance equations is that the source and sink actor must execute the same number of times in a valid schedule.

For example, in Figure 3.2, ϕ is first converted to its equivalent SDF reduction ϕ'. The repetitions vector for ϕ can then be computed as the repetitions vector for ϕ', which is given by

$$q(A, B, C, D, E, F) = \begin{bmatrix} 9 & 6 & 6 & 12 & 12 & 8 \end{bmatrix}^T. \tag{3.1}$$

3.4.3. Looped Schedules

A *schedule* for a dataflow graph is a (finite or infinite) sequence of actor executions. To make schedules more compact (e.g., to reduce the code size when implementing schedules), it is useful to employ looped schedules, which employ looping constructs called *schedule loops* across regions of the schedules that involve repetitive execution schedules. A schedule loop is a parenthesized term of the form $(nI_1I_2\ldots I_k)$, where n is a positive integer, and each I_j, called an *iterand*, is either an actor or a (nested) schedule loop. For example, the schedule $(BBABBABBCCC)$ for Figure 2.1 can be expressed more compactly using schedule loops as $(2B)(2A(2B))(3C)$. Schedules such as this that employ looped notation are called *looped schedules*.

A *single appearance schedule (SAS)* of a dataflow graph is a looped schedule in which all actors appear only once. For example, the looped schedule $(2B)(2A(2B))(3C)$ for Figure 2.1 is not an SAS because it contains multiple appearances of B. On the other hand, $(2A)(3BC)$ and $(2A)(3B)(3C)$ are both valid (looped) schedules for Figure 2.1 that are also SASs.

Intuitively, a SAS provides minimized code-size for a software implementation of a dataflow graph. Acyclic, pairwise grouping of adjacent nodes (APGAN) [5] is a previously-developed algorithm that is useful for deriving SASs. Specifically, given a consistent, acyclic, and connected SDF graph, APGAN derives a SAS using heuristic techniques that minimize buffer memory requirements for the edges in the graph.

In this section, we show that with minor adaptations, APGAN can be applied to HPDF to derive valid SASs for HPDF-SDF graphs. We first review the original APGAN algorithm for SDF, and then show how the same basic approach can be applied to HPDF.

Suppose that we have a consistent, acyclic, and connected SDF graph G with k actors $v_1, v_2, \ldots, v_k$, and let the repetitions vector for G be $q_G = (n_1 \ldots n_k)^T$, with each ith element of this vector being in correspondence with actor v_i. We define the *repetition count* of an actor v_i to be n_i — i.e., the corresponding entry in the repetitions vector, and we represent this value also by $q_G(v_i)$. We also define

$$\rho_G(v_m, v_n) = gcd(q_G(v_m), q_G(v_n)), \tag{3.2}$$

where the *gcd* represents the *greatest common divisor* operator.

The basic idea in APGAN is that in each iteration, we try to hierarchically pair up two adjacent actors that have the maximum common repetition count — as defined by (3.2) — among all available adjacent actor pairs that can be clustered without introducing cycles in the graph. Here by *clustering* a subset Z, we mean grouping the actors in Z into a single hierarchical actor for the purposes of scheduling. This mechanism of clustering can be used to constrain subsequent scheduling steps so that the actors in Z are always scheduled together, as a single unit.

The clustering process of APGAN is repeated until there are no more actors available for pairing — that is, when the top-level of the clustered hierarchy consists of a single actor. In the resultant graph at each clustering iteration x, we represent the newly grouped actors by the new hierarchical actor Ω_x.

If the actors chosen by APGAN are denoted by (v_M, v_N) in a given clustering iteration, then for any edge e that has v_M or v_N as its source, e will be replaced by a modified version e' that has the following properties [5]:

$$src(e') = \Omega_x, snk(e') = snk(e), del(e') = del(e);$$

$$prd(e') = prd(e) \times (q_G(src(e)) / \rho_G(v_m, v_n)); \text{ and}$$

$$cns(e') = cns(e). \tag{3.3}$$

Similarly, for any edge e having v_M or v_N as its sink, e is replaced by a modified version e' that satisfies:

$$src(e') = src(e), snk(e') = \Omega_x, del(e') = del(e);$$

$$cns(e') = cns(e) \times (q_G(snk(e)) / \rho_G(v_m, v_n)); \text{ and}$$

$$prd(e') = prd(e). \tag{3.4}$$

Once the clustering process is complete, a schedule is constructed by recursively traversing the cluster hierarchy, and scheduling the clusters as they are traversed. The final schedule that results from this process is a SAS for the input SDF graph. An illustration of the operation of APGAN is shown in Figure 3.3.

To apply the concepts of APGAN to an HPDF graph, let us consider a parameterized edge with parameters p, where the source of that edge is B and the sink is C, and let the graph in Figure 3.4 represent an arbitrary iteration in APGAN. Here, dotted edges and vertices represent multiple copies — i.e., there can be multiple copies of G and multiple edges connecting them to or from B or C. The k_is are rates that are independent of p. Ω_i represents the grouped (hierarchical) actor that contains B, and Ω_i represents the hierarchical actor that contains C.

As discussed in Section 3.4.2., the repetitions vector for a graph such as ϕ_i is independent of p. Therefore, the production rate at the output edge of Ω_i will be (based on (3.3)) $p \times (q(B)/\rho_G(B, G_1))$, where $(q(B)/\rho_G(B, G_1))$ is independent of p and hence replaced by k_5. A similar argument can be used to show that k_6 is independent of p as well. By similar reasoning as in Section 3.4.2, it can be shown that the repetitions vector

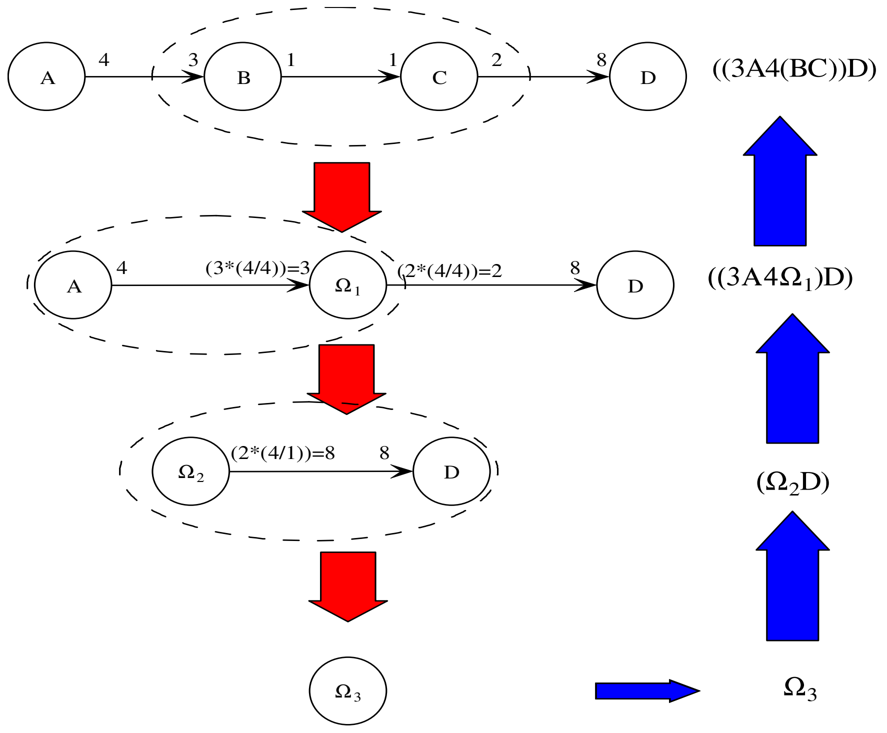


Figure 3.3 Example of working of *APGAN* and the resultant schedule.

for ϕ_j will be independent of p (by replacing p_l and $-p_l$ with $k_m p_l$ and $-k_n p_l$). Thus, the consumption rate on the incoming edge (from (3.4)) will be $k_3 \times (q(\Omega_i)/\rho_G(\Omega_i, \Omega_{i'}))$, which is clearly independent of p, and hence denoted by k_7 in Figure 3.4. Similar reasoning shows k_8 to be also independent of p.

After the clustering process of APGAN is complete, the final schedule is produced, as discussed earlier, by a recursive traversal of the cluster hierarchy. During the traversal of ϕ_j, in the schedule for graph ϕ_{j+1} (which is independent of p), $\Omega_{i''}$ will be replaced by $k_6\Omega_i k_5\Omega_{i'}$, which again is independent of p. This demonstrates that APGAN will produce a static schedule (i.e., a schedule that is independent of any dynamic edge parameter p) for an HPDF-SDF graph. Furthermore, as in the application of APGAN to SDF graphs, an SAS will be produced when APGAN is applied to an HPDF-SDF graph.

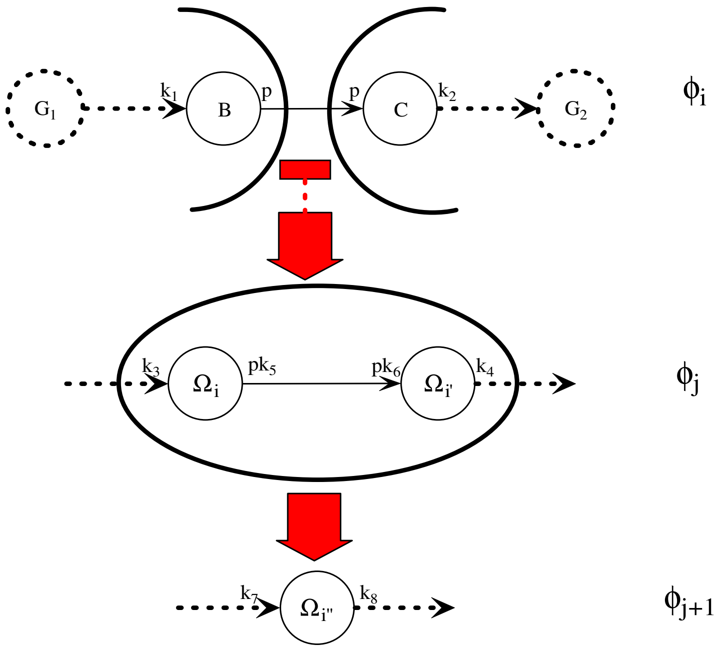


Figure 3.4 *APGAN* on an *HPDF* graph.

3.4.4. Single-rate Equivalents of HPDF Graphs

A single-rate dataflow (*SRDF*) graph is a dataflow graph in which all the sample rates (production and consumption rates on each edge) are same. A special case of SRDF is Homogeneous Synchronous Dataflow (*HSDF*) where all the sample rates are unity. Conversion to HSDF is a powerful technique because high-level performance estimations can be performed on an HSDF based on the following definition of *Maximum Cycle Mean* (*MCM*):

$$MCM(\Phi) = \max_{cycle C \in \Phi} \left\{ \frac{\sum_{v \in C} t(v)}{Delay(C)} \right\}$$

MCM puts a fundamental limit on the achievable throughput of a system [51]. We can apply the algorithm for converting an SDF graph to its equivalent HSDF [51] on an HPDF graph. The only difference the resultant HSDF would have, as compared to being generated from an SDF, is that all the nodes that were either sources or sinks of the HPDF, would be connected by a variable number of edges. We show the HSDF equivalent of the HPDF graph Φ in Figure 3.5. Note that the number of edges between $B(I)$ and $C(I)$ is p. However, this graph can still be used to calculate the MCM of Φ to derive the maximum achievable throughput for an HPDF graph. We will now demonstrate an interesting property — changing the parameter p can change the MCM value of Φ, hence the parameter p can be a determining factor in the throughput of the system.

Let us consider that the cycle in the HSDF with maximum mean is ζ. And let B and C be the source and sink of a parameterized edge with parameter p which have repetition counts of I (they will have the same repetition count due to homogeneity constraints).

Also note, that by construction of the HSDF, all the edges that have $B(i)\forall(i \in I)$ as the source will have $C(i)$ as the sink — we denote these edges as $e_{B(i),\, C(i)}$.

Let the cycle ζ involve at least an edge of type $e_{B(i),\, C(i)}$. Also, let the number of delays on the edge between B and C be η. If during execution, the parameter p assumes a value greater than η, then there would be $p - \eta$ delayless edges for each $B(i), C(i)$. So if $Delay(\zeta) = d$ for $p < \eta$, then for $p > \eta$, $Delay(\zeta) = d - 1$, resulting in an increased $MCM(\Phi)$.

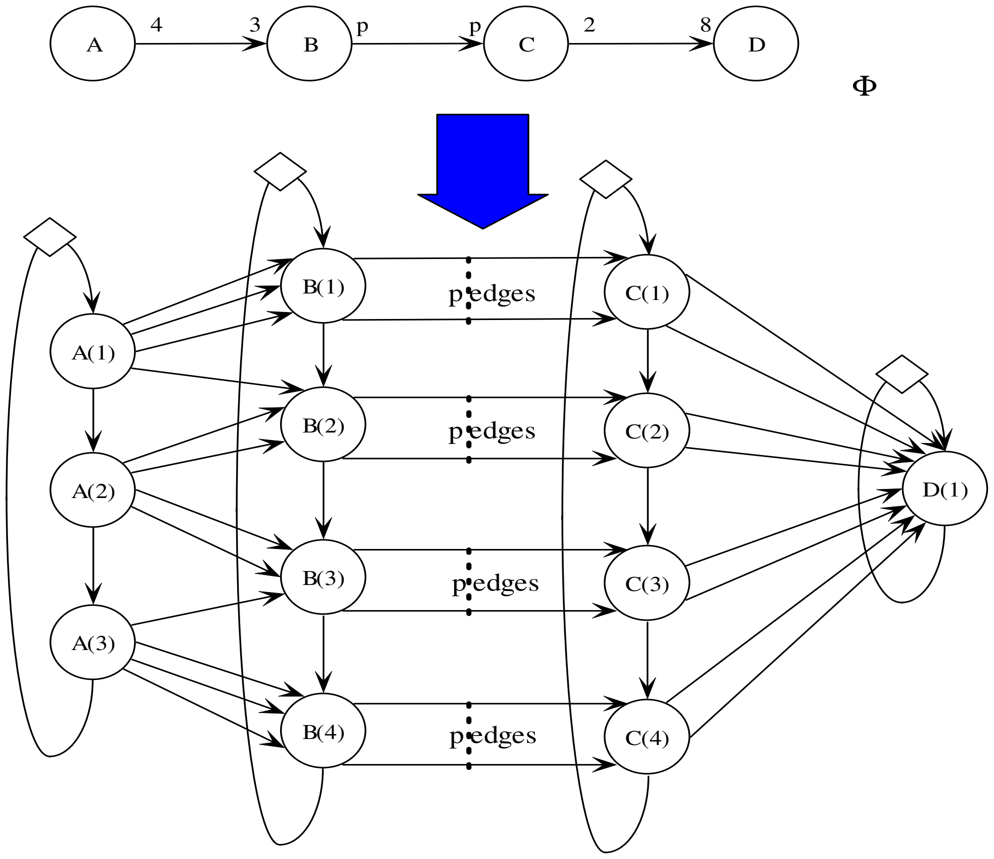


Figure 3.5 *HSDF* equivalent of an *HPDF* graph ϕ.

3.5. Modeling using HPDF

In this section, we first present a brief description of a “Gait-DNA” algorithm [42] for load carrying event detection, model the application using HPDF-SDF and then refine the model using HPDF-CSDF. We have also presented two additional examples of modeling using HPDF in Chapter 5 and in Chapter 6.

3.5.1. Gait-DNA algorithm for load carrying event detection

A Gait-DNA is an image processing algorithm which looks for periodic patterns corresponding to movement of wrists, elbows, knees and ankles in a sequence of image frames to represent the gait signature of a human being. Since the generated pattern from gait looks like a double helix in the DNA structure, hence the name. From the relative distortion in rate and period of the double helix pattern, a load carrying event detection formulation was proposed in [42].

The algorithm has 4 basic blocks at the top level - “Input”, “Slice Generator”, “Double Helix Signature (*DHS*) Extraction”, and “Classifier”. The actor “Input” can be any input device producing image frames of a certain size ($x \times y$ pixels) at a certain rate measured as frames per second (*fps*). “Slice Generator” stores a pre-defined number of collection of images — say t of them (we henceforth call this collection as an “input block”), and first finds “bounding box”-es which are the regions of interest for finding gait in all the frames. So a bounding box typically would enclose one human being, and there can be multiple of them in a frame corresponding to multiple human beings. Then it creates 4 y coordinates depending on the heights of the bounding boxes (or objects) which will correspond to the person’s elbow (y at 0.85 of object height), wrist (y at 0.75 of object height), knee (y at 0.35 of object height), and ankle (y at 0.25 of object height). The output of this actor is the value of the pixels at each of these y coordinates of each of the bounding boxes over the entire input block. So it effectively *slices* through the input block and creates 4 such slices for every human being. "DHS Extraction" performs 1-D curve approximation after dividing each slices into *strides* (one stride is one "8" like pattern in the double helix), and subsequently dividing each stride into 4 *quadrants* (each quadrant being one non-intersecting curve of the double helix pattern). It then extracts the average rate and amplitude of the approximated curves for each slice and outputs these information to the "Classifier". "Classifier" looks at the rate and frequency of each quadrant and after doing symmetry analysis classifies gait as either natural walking, carrying an object in one hand or carrying an object with both hands.

The experiment was done with frames for at least 2 seconds with 15 frames/sec. (total of 30 frames) and the size of the frames were 720×480 pixels.

3.5.2. HPDF-SDF modeling of Gait-DNA

We first model the Gait-DNA algorithm for load carrying event detection using HPDF-SDF as shown in Figure 3.6. "Input" (or actor A) produces p' tokens which is the number of image frames constituting one input block - this is a parameter, as the authors in [42] reported correct results with downsampling factor of up to 4 (for both the size and

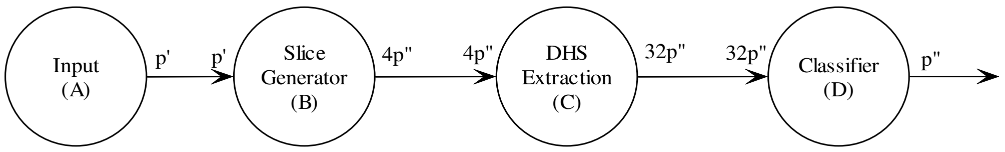


Figure 3.6 HPDF-SDF model of the Gait-DNA algorithm for load carrying event detection.

the frames per second) provided the bounding boxes are bigger than 40×60. "Slice Generator" (or actor B) consumes all the p' frames to produce $4p''$ tokens. p'' is the number of bounding boxes in the frames (which is a input-dependent parameter and can only be determined at run-time) and for each bounding box, 4 slices are generates as explained in Section 3.5.1. "DHS Extraction" (or actor C) generates 8 tokens for each of the $4p''$ slices (2 quantities — average rate and amplitude for each of 4 quadrants). "Classifier" consumes all the $32p''$ tokens and produces one token for each of the p'' bounding boxes which represents one of the three activities described earlier. The schedule of the graph would be:

$$ABCD. \tag{3.5}$$

3.5.3. HPDF-CSDF modeling of Gait-DNA

In this section, we further refine our model by using CSDF as the underlying model of our HPDF metamodel as shown in Figure 3.7. Parameters p', p'' represent the same parameters as described in Section 3.5.2.

A has the same behavior as its SDF counterpart, but now B has $1 + 4p''$ phases, where in the first phase, it consumes the input block with p' image frames, without producing any token. In the next $4p''$ phases, it produces a slice at each phase without consuming any data. C consumes one token (representing one slice) at every phase for $4p''$

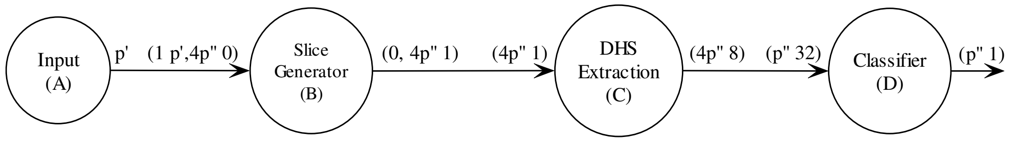


Figure 3.7 HPDF-CSDF model of the Gait DNA application for load carrying event detection.

phases of its execution and produces 8 tokens — 4 pairs of rate and amplitude. D has only p'' phases and it consumes 32 tokens corresponding to each bounding box to produce one token which specifies the class of activity. The schedule of the graph would be:

$$AB(p''\ (4\ BC)D)\,. \tag{3.6}$$

There are certain advantages of HPDF-CSDF over HPDF-SDF like lower buffer requirements. For example, looking at the edge between C and D, we see that buffer requirement in (3.5) is $32p''$ whereas the same edge has a buffer requirement of 32 in (3.6). Similarly for the edge between B and C, buffer requirement goes down from $4p''wt$ to wt where w and t are the width of the bounding box and number of frames in the input block respectively.

We have later modeled a single camera gesture recognition algorithm in HPDF-SDF and HPDF-CSDF. A comparative study of the two models can also be found in Chapter 5. We also present the model for the distributed gesture recognition algorithm in Chapter 5. Also Chapter 6 presents an image registration algorithm that was modeled using the HPDF-CSDF and some interesting transformations that were applied to the model.

Chapter 4. Transformations for HPDF Graphs

In this chapter, we describe an approach that we explored for low-power synthesis and optimization of digital signal, image, and video processing (DSP) applications. In particular, we consider the systematic exploitation of node unfolding (which as we explain in details later is possible due to data parallelism) across the operations of an application dataflow graph when synthesizing a dedicated hardware implementation. Data parallelism occurs commonly in DSP applications, and provides flexible opportunities to increase throughput or lower power consumption. Exploiting this parallelism in dedicated hardware implementation comes at the expense of increased resource requirements, which must be balanced carefully when applying the technique in a design tool. We propose a high level synthesis algorithm to determine the node unfolding factor for each computation, and based on the area and performance trade-off curve, design an efficient hardware representation of the dataflow graph. For performance estimation, our approach uses a cyclostatic dataflow intermediate representation of the hardware structure under synthesis. Then we apply an automatic hardware generation framework to build the actual circuit.

4.1. Motivation

High-level synthesis has been of primary importance in the field of DSP as area and power considerations are critical in the DSP domain. Design space exploration can be done effectively from a high level description as some inherent traits are more obvious in the high level abstraction and become obscure in the low level implementations. Dataflow has proven to be an attractive high-level computation model for programming DSP applications. A restricted version of dataflow, termed synchronous dataflow (SDF), that offers strong compile-time predictability properties, has been studied extensively in the DSP context [5][31] (also see Section 2.1.1 of this thesis for the definition of SDF). We have developed an algorithm and Verilog code generation framework for optimal application of data parallel hardware implementations to SDF graphs. Further, since $power \propto V^2 f$, where V is the operating voltage and f is the operating frequency, we can reduce both V and f by sacrificing the performance gain that our algorithm provides and still maintain existing performance but at a lower power. As an example, we consider a simple 3-tap FIR filter. Figure 4.1 shows a synchronous dataflow graph representation of such a filter.

Here, the inputs to each module consume one unit of data upon each invocation, and the modules produce one unit of data at the output. From this SDF graph representation of the filter, we can clearly see that data parallelism through replication of hardware blocks can be used for each of the modules to increase the throughput. The given dataflow graph provides enough information to derive a hardware implementation of the filter. But by analyzing the given dataflow, we can increase the throughput of the circuit by duplicating

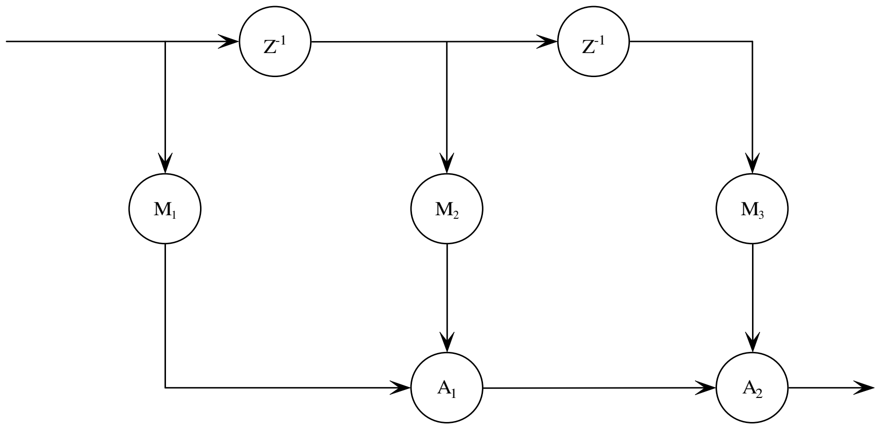


Figure 4.1 An SDF graph representation of a 3-tap FIR filter with production and consumption rates uniformly equal to one.

the multipliers and creating parallel datapaths to them. Figure 4.2 shows the 3-tap FIR filter of Figure 4.1 with node unfolding factors of $\langle 2, 2, 3\rangle$ and $\langle 1, 1\rangle$ for the multipliers and the adders respectively. The additional switches needed for sending data to multiple instances of the modules are also shown. This possibility of configuring a node unfolded (or replicated) hardware implementation results in a wide design space to probe around in order to maximize throughput or minimize power consumption.

The rest of this chapter is constructed as follows. In Section 4.2, we present the definition of node unfolding and present an algorithm to unfold a node n in a graph j times. In Section 4.3, we present how and when node unfolding can be applied to HPDF and some interesting results it gives. In Section 4.4, we present a formal synthesis problem statement as well as the optimality of the solution provided by the proposed systematic node unfolding algorithm. Section 4.5 provides the framework used for automatic hardware code generation from the optimally configured circuit given by our algorithm. Sec-

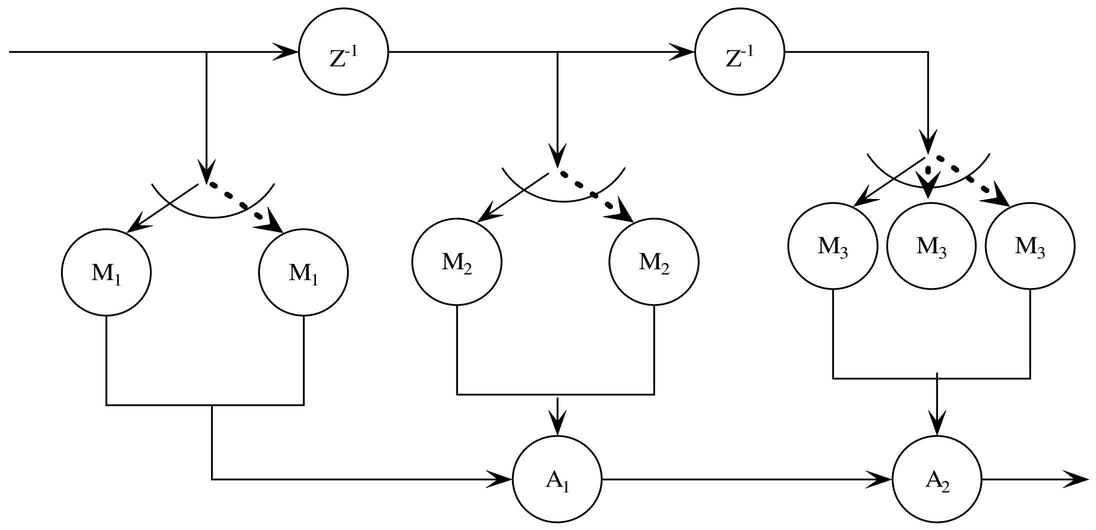


Figure 4.2 The 3-tap FIR filter shown in Figure 4.1 with different node unfolding factors for the multipliers. Switches are also shown as a means of implementation of node unfolding.

tion 4.6 provides results for some typical DSP subsystems. Section 4.7 discusses the implications of the results.

4.2. Node Unfolding

We define *node unfolding* as a technique, in which instead of unfolding the whole graph, the algorithm tries to unfold (replicate) nodes and edges that are either incoming and outgoing to or from the unfolded (replicated) node [39]. Switches which can naturally be represented as CSDF actors, are to be inserted at appropriate places to maintain correct functionality. We propose an algorithm to systematically unfold a node (n) j times in a dataflow graph in Figure 4.3. We show the concept of node unfolding through an example (Figure 4.4).

```
Place a switch $S_{in}$ on each edge incoming to $n$.
Place a switch $S_{out}$ on each edge outgoing from $n$.
Make $j$ copies of $n$ denoted by $n_i$, $1 \le i \le j$.
For each incoming edge to $n$, place one edge connecting $S_{in}$ and
$n_i$, for all $i$ where the production rate is of the form
$(0, 0, \ldots, c_n, \ldots)$ with $c_n$ being the consumption rate of $n$ and is
placed on phase $i$, total number of phases being $j$ and consump-
tion rate is $c_n$.
For each outgoing edge from $n$, place one edge between $n_i$ and
$S_{out}$, for all $i$ where the consumption rate is of the form
$(0, 0, \ldots, p_n, \ldots)$ with $p_n$ being the production rate of $n$ and is
placed on phase $i$, total number of phases being $j$ and produc-
tion rate is $p_n$.
Any delays on incoming edges to $n$ stays on the corresponding
edge now incoming to the switch $S_{in}$.
Any delays on outgoing edges to $n$ gets transferred to the cor-
responding edge now outgoing from the switch $S_{out}$.
A self-loop with delay $d$ on $n$ would get replicated on each
unfolded node $n_j$ with the same amount of delay $d$.
Consecutive switches on an edge can be replaced by one switch.
```

Figure 4.3 Algorithm for unfolding a node n, j times.

4.3. Node Unfolding on HPDF

Node unfolding can also be applied to HPDF systems whenever there is a data-rate mismatch — i.e, where data production and consumption rates for parameterized edges are not same, but differ by a constant factor k ($k > 1$)— say production rate is p and consumption rate is kp. In the original graph in Figure 4.5 (top), we see that the edge between C and D has such data-rate mismatches, with $k = 4$. The actor with a lower data-rate (or equivalently with a higher firing rate) of this edge can be unfolded k times using the algorithm described in Figure 4.3. The resultant graph is still HPDF but with a higher throughput but with an area overhead. We apply such a transformation and the resultant graph along with the original graph is shown in Figure 4.5. We also applied node unfolding on the HPDF graph of an image registration algorithm in Chapter 6 and presented a comprehensive study of area performance trade-off in Section 6.7.

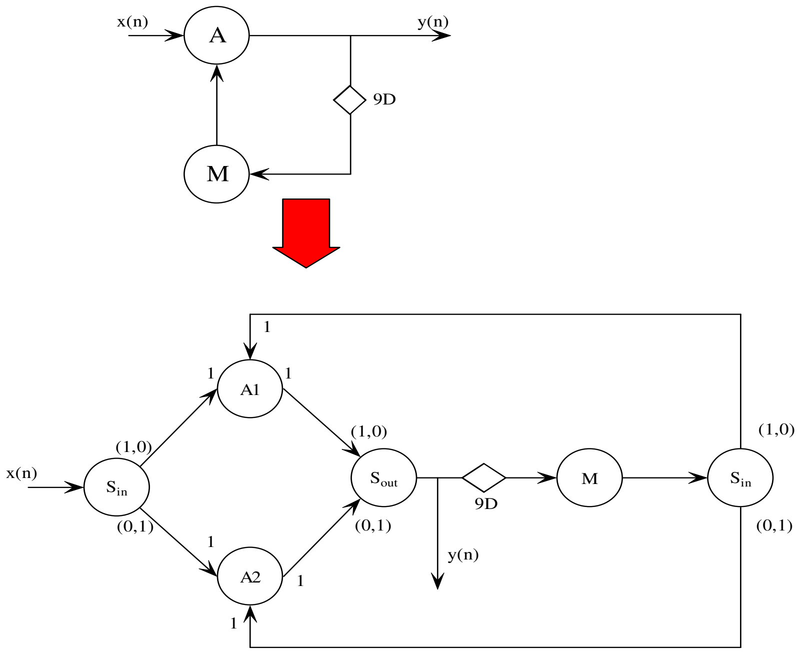


Figure 4.4 Illustration of the Node unfolding algorithm where A is unfolded twice.

4.4. High-level Synthesis Problem Statement and Systematic Node Unfolding Algorithm

In this section, we present the formal statement of the synthesis problem that we address, and present the algorithm developed to solve it. We also show that the algorithm has polynomial complexity and provides optimal synthesis results.

4.4.1. Problem statement

In this model, each functional module M (dataflow graph vertex) that has a node-unfolded implementation is characterized by an overhead factor, denoted v_M, which approximates the amount of additional functional resource area (or cost) required for each

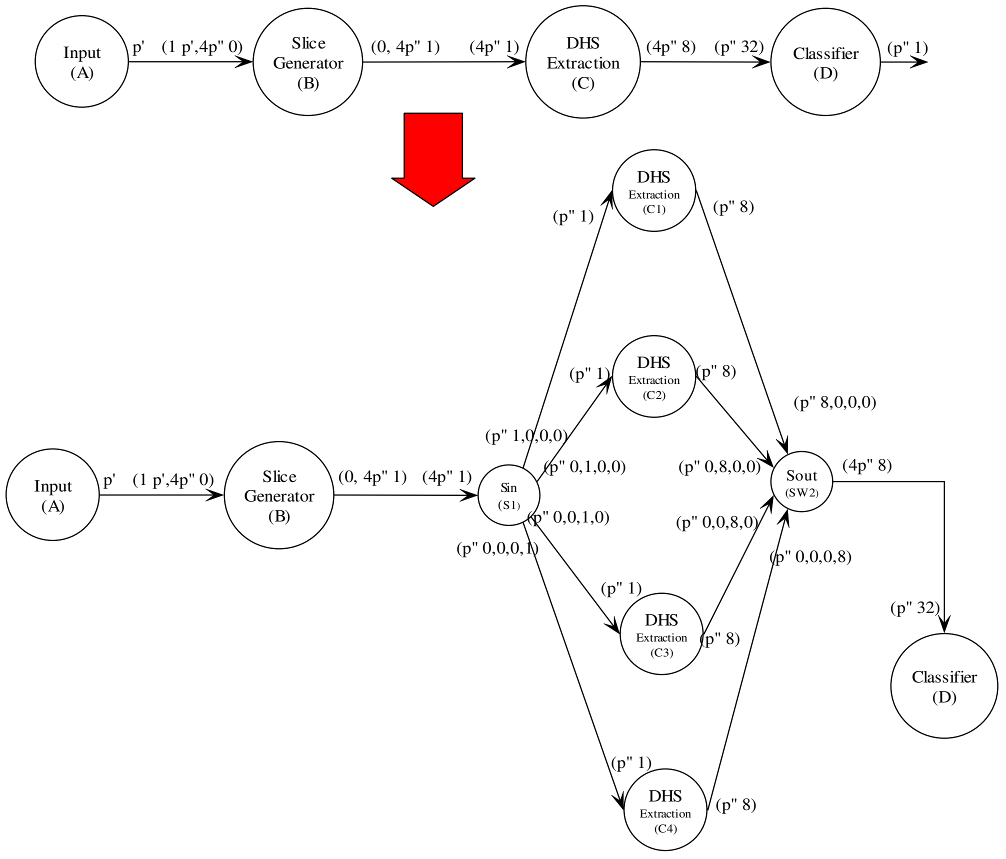


Figure 4.5 Example showing node unfolding being used on HPDF.

level of unfolding of the node. Specifically, an m-fold unfolding of a node M (an implementation with m parallel copies of the hardware block) is modeled as requiring a functional resource cost of

$$A_M + (m-1)v_M A_M = A_M(1 + (m-1)v_M), \tag{4.1}$$

where A_M is the cost of a single instance of module M (without application of node unfolding). Similarly, a module-independent area/cost switching overhead is used to model the switching area or communication cost required for the connection between an incoming (outgoing) data stream and an m-way parallel network of hardware modules operating at m times the data rate of the stream. Under this formulation, the data parallelism synthesis problem becomes one of determining a strategic mapping $\mu : V \to Z+$, where V denotes the set of application modules (dataflow graph vertices), $Z+$ denotes the set of positive integers, and $\mu(M)$ denotes the level of unfolding (the number of parallel instantiations) of module M. So, we are concerned with the constraints of area (cost), power consumption, and throughput, and the objective of data parallel synthesis is to achieve an optimal or near-optimal configuration μ that targets the relevant constraints and optimization criteria across these metrics.

4.4.2. Proposed Algorithmic Solution

We present the algorithm in Figure 4.6. The algorithm follows a greedy approach. At every iteration, it checks for the module which when duplicated gives the maximum performance benefit. The algorithm terminates when duplicating any hardware module violates the area constraint.

Data Structures Used:

```
list = queue of structures;
newlist = queue of structures;
structures = struct {
module_info;
number_of_copies_of_the_module;
}
```

Main algorithm:-

```
Form the newlist by enqueueing all the modules;
while(newlist_not_empty) {
    list = newlist;
    while(list_not_empty) {
            $Module_i$ = dequeue from the list;
            m = ($Module_i$ —>copies) ++;
            $A_{new} = v_i A_i + Area_{old}$;

            if ($A_{new}$ < Area available) {
                    Performance Analysis($Module_i$);
            }
    }

    get the module with **best** result;
    ($Module_i$ —>copies) ++ in newlist;
}
```

Performance Analysis:-

```
Form the corresponding Cyclostatic Dataflow(CSDF);
CSDF to Homogeneous Synchronous Dataflow(HSDF);
Maximum Cycle Mean (MCM) Analysis;
Store the result;
Enqueue_newlist($Module_i$, copies);
```

Figure 4.6 The algorithm used to get the data parallel factors for each module.

Performance benefit is measured by the function ‘Performance Analysis’ in Figure 4.6. The switching characteristics of any circuit is very aptly represented by the dataflow computational model known as Cyclo-static dataflow (CSDF) [6]. In this model, a module can have different phases in which it can consume and produce data at different rates. The initial dataflow along with the data parallel factors and switches can now be effectively represented by an equivalent CSDF graph. Performance of the resulting dataflow graph is measured by its throughput. This is done by first forming the equivalent Homogeneous SDF (HSDF) graph of the CSDF graph [6]. An HSDF graph is an SDF graph whose data production and consumption rates per firing are uniformly equal to one. Every module in the CSDF graph forms a cycle C in which the different elements in one cycle corresponds to different phases of the CSDF graph in its most simplified form. Let $W(v_i)$ be the execution time of each of the modules in one such cycle C_j. Then $\Sigma W(v_i)$ is the total weight of the cycle C_j. The mean cycle weight C_j in an HSDF graph is defined as

$$(\Sigma W(C_j))/(Delay(C_j)). \tag{4.2}$$

[51] where $Delay(C_j)$ is the total number of delay elements in C_j. The cycle with the maximum mean cycle weight is called the *critical cycle*; it gives the maximum achievable throughput for the graph.

Let T_i be the execution time for a module i and let the unfolding factor for this module be N_i. Then the throughput for module i is $(N_i/T_i) = P_i$. The throughput of the entire system is thus $Min(P_i)$ over all i. Also let B_i be the base area of i. Our objective is to maximize $Min(P_i)$ subject to the constraint $\Sigma f(N_i, B_i) \leq A_{max}$ where A_{max} is the maximum die-area on the chip available for hardware implementation. $f(N_i, B_i)$ is approximately $N_i B_i$ if the overhead for multiple hardware units is negligible.

In the greedy approach taken, we repeatedly select the bottleneck module i and increase its data parallel factor by one, provided area constraint is not violated. In effect, we expand the module i just enough so that it is no longer the bottleneck for the system. This greedy procedure results in optimal configurations; this can be seen from the following argument. If a module i that is the bottleneck has a current data parallel factor p_i, and only a data parallel factor of $p_i + k$ or more will remove it from being the bottleneck, then the algorithm will always choose i for the next $(k - 1)$ iterations (provided there is enough area). In other words, the algorithm always devotes available area toward improving the bottleneck module, which is the best that can be done under a given area constraint. Improving the performance of any non-bottleneck task cannot improve the overall throughput. The maximum number of times a module i is visited by the algorithm is $(A_{max})/B_i$ which is polynomial.

4.5. Automatic Verilog Code generation

After the synthesis algorithm provides the node unfolding factor vector, we simulate the actual hardware. For that we have developed an automatic Verilog code generation framework that is built on top of Ptolemy II [17], a design environment for modeling and design of heterogeneous embedded systems.

4.5.1. Motivation for code generation

To measure the effectiveness of our algorithm, we have performed area and power calculations on a number of circuits, which are presented in the results section. We synthesized the dataflow graphs in hardware to verify our results from the algorithm. Thus our results are backed by actual synthesis rather than software simulation.

4.5.2. Code generation methodologies

We have explored two different approaches for code generation. We either describe the Verilog code for a module as a congregation of functions it performs or we have a standard code library that implements the basic structure for that module. The only difference is in the granularity in which we confront the code generation problem.

The two different approaches were considered based upon flexibility and speed for code generation. If the user needs more customized code generation, then the functional description approach is more suited to his needs. But the user should have a sound knowledge on synthesizable code generation for the generated code to work correctly. As for code generation from the standard library, the user need not know the intricacies of code generation. A basic parameterized framework for a particular module is already provided in the library, the user needs to invoke it with the required parameters, one of them being the number of inputs to the module. For example, an adder can be a two bit adder, or any n-bit adder — and this parameter needs to be specified at the time of invocation. This is a very reasonable approach for code generation, and also we can generate area optimized code that is suitable for low power applications as the library modules are optimized. Overall, the library approach is easier and usually produces better code. We discuss this approach in more detail in the following section.

4.5.3. Library approach to code generation

From the input SDF graph, we extract all the modules needed for code generation. The only way to have a one-to-one correspondence between the module and the correct code from the library is to use a uniform nomenclature. For this purpose, we have used the intuitive names such as adder, delay, multiplier, etc. for the corresponding modules. After the modules are identified, we import the module definition from the library. The different modules are wired after the wiring pattern is extracted from the input SDF graph. Evidently, the wires are the edges in the graph. If the unfolding factor for a particular module is n, then the code for it is defined only once but instantiated n times. We add a switch to manage the data parallelization for the n instantiations. The generated code for the above mentioned 3-tap FIR is given in Figure 4.7 and 4.8. The adder module is the only complete module. The input, output and reg statements are omitted from the other module definitions for brevity.

The code generated is divided into synthesizable and verifiable parts. This feature is maintained by using the testbench approach for Verilog code generation. We generate two separate files, one file contains code for the system being designed, and the other file contains the test generator and the monitor. The first file contains the synthesizable part and

```
module adder(in1, in2, in3, out, clk);
    input [15:0] in1;
    input [15:0] in2;
    input [15:0] in3;
    input clk;
    output [15:0] out;
    reg [15:0] out;
  always @(posedge clk) begin
        out <= in1 + in2 + in3;
  end

endmodule

module multiplier(in1, in2, out, clk);
  always @(posedge clk) begin
        out <= in1 * in2;
  end
endmodule

module delay(in1, out, reset, clk);
  always @(clk or reset) begin
  if (reset == 1) begin
            out <= 0;
  end
  else if (clk == 1) begin
            out <= in1;
  end
    end
endmodule


  module top(in, clk, reset, out);
   assign param0 = `h0;
   assign param1 = `h1;
   assign param2 = `h2;

   adder a(w2, w4, w6, out, clk);
   multiplier m1(in, param0, w2, clk);
   multiplier m2(w3, param1, w4, clk);
   multiplier m3(w5, param2, w6, clk);
   delay d1(in, w3, reset, clk);
   delay d2(w3, w5, reset, clk);
 endmodule
```

Figure 4.7 Generated synthesizable Verilog code for the 3-tap FIR filter described in Figure 1.

the second file contains wires to input and output modules needed for verification. This approach is described in detail in [52]. Figure 4.7 shows only the synthesizable code for a 3-tap filter.

We also generate the code for a switch when we simulate the hardware for the graph shown in Figure 4.2. A simple $1 \rightarrow 2$ switch generated by our code generator is shown in Figure 4.8.

4.6. Results

We evaluated our algorithm on a number of typical DSP subsystems. We present the results of three such systems. The first one is a cascade of a simple adder (input node) and multiplier (output node). Simulation results from Synopsys Design Compiler [60] are shown in Table 4.1. The second circuit is a 3-tap FIR circuit shown in Figure 4.1. Third is

```
    module   switch(in1,   datainready1,   in2,
datainready2, reset, clk, dataoutready, out);
    always @(posedge clk) begin
       if (reset == 1) begin
              counter <= 0;
              datainready1 <= 0;
              datainready2 <= 0;
              dataoutready <= 0;
              end
       else if(counter == 0) begin
              counter <= counter + 1;
              out <= in1;
              dataoutready <= 0;
              datainready1 <= 1;
              datainready2 <= 0;
              end
       else if(counter == 1) begin
              counter <= counter + 1;
              out <= in2;
              dataoutready <= 1;
              datainready2 <= 1;
              datainready1 <= 0;
              end
       end
    endmodule
```

Figure 4.8 The example Verilog code of a simple $1 \rightarrow 2$ switch.

a second order IIR filter. We observe that the data parallel factors provided by our synthesis algorithm are supported by the data values produced by Design Compiler.

For the first circuit, our algorithm suggested $M = \langle 2 \rangle, A = \langle 1 \rangle$. The synthesized circuit gives maximum throughput for the same configuration under an area constraint of 60000 μcm^2. For the 3-tap FIR filter, the best performance is provided by $M_{1,2,3} = \langle 2, 2, 2 \rangle$, $A_{1,2} = \langle 1, 1 \rangle$ for $A_{max} = 130000 \mu cm^2$ which tallies with the output of our algorithm. The second row of Table 4.2 shows that even though the multiplier is the bottleneck, providing only one parallel data-path to one of the multipliers does not

Config	Area (μcm^2)	Dynamic Power (mW)	Critical Time (ns)
M=<1> A=<1>	27394	1.52	10.31
M=<1> A=<2>	33269	1.71	9.51
M=<2> A=<1>	51903	2.83	5.93

Table 4.1. Results of adder multiplier circuit from Synopsys

Config	Area (μcm^2)	Dynamic Power (mW)	Critical Time (ns)
$M_{1,2,3}$=<1,1,1> $A_{1,2}$=<1,1>	68632	3.89	10.68
$M_{1,2,3}$=<1,2,1> $A_{1,2}$=<1,1>	88266	4.62	10.68
$M_{1,2,3}$=<2,2,2> $A_{1,2}$=<1,1>	126634	6.08	5.88

Table 4.2. Results of a 3-tap FIR filter from Synopsys

decrease the critical path time, and accordingly, our algorithm does not choose this as an improved configuration. Even though the results shown here are for moderate-sized graphs with numbers of modules on the order of tens, the core algorithm is of low polynomial complexity, and therefore our approach can be expected to scale efficiently to larger systems.

4.7. Conclusion

The above tables show some of the possible configurations of the mentioned dataflow graphs that do not violate the given area constraints. It can be observed that in all of the above cases, the data parallel configuration suggested by our synthesis algorithm was the solution with the best performance. Power measurements are given as an added parameter to the problem.

Data parallelism for DSP hardware implementation is a well-known concept; the contribution of our work is in the full vertical integration of data-parallelism-based transformations with synchronous dataflow graph analysis, cyclostatic dataflow-based performance analysis, synthesizable Verilog code generation, and hardware synthesis using the

Config	Area (μcm^2)	Dynamic Power (mW)	Critical Time (ns)
$M_{1,2,3,4}$=<1,1,1,1> A_1=<1>	134812	1.9	7.63
$M_{1,2,3,4}$=<2,2,2,2> A_1=<1>	259309	3.5	4.09

Table 4.3. Results of a second order IIR filter from Synopsys

Synopsys Design Compiler. This integration provides a fully automated design flow that produces optimal exploitation of data parallelism for SDF-based designs.

Useful directions for further work include hardware synthesis from more general dataflow models, such as integer-controlled dataflow [8], and well-behaved dataflow [18]; and systematic integration with other flowgraph transformations for multi-objective synthesis.

Chapter 5. HPDF-based Analysis Case Study: Gesture Recognition

Computer vision methods based on real-time video analysis form a challenging and increasingly important domain for embedded system design. Due to the their data-intensive nature, hardware implementations for real-time video are often more desirable than corresponding software implementations despite the relatively longer and more complicated development processes associated with hardware implementation. The approach that we pursue in this thesis is based on direct representation by the designer of application concurrency using dataflow principles. Dataflow provides an application modeling paradigm that is well-suited to parallel processing (and to other forms of implementation streamlining) for digital signal processing (DSP) systems [51]. Dataflow is effective in many domains of DSP, including digital communications, radar, and video processing.

In this chapter, we use dataflow as a conceptual tool to be applied by the designer rather than as the core of an automated translation engine for generating HDL code. This combination of a domain-specific model of computation, and its use as a conceptual design tool rather than an automated one allows great flexibility in streamlining higher level steps in the design process for a particular application.

As an important front-end step in exploiting this flexibility, we employ HPDF (homogeneous parameterized dataflow) (See Chapter 3) semantics to represent the behavior of the target gesture recognition system. HPDF is a restricted form of dynamic dataflow, and is not supported directly by any existing synthesis tools. However, an HPDF-based modeling approach captures the high-level behavior of our gesture recognition application in a manner that is highly effective for design verification and efficient implementation. As our work in this chapter demonstrates, the HPDF-based representation is useful to the designer in structuring the design process and bridging the layers of algorithm and architecture, while HDL synthesis tools play the complementary role of bridging the architecture and the target platform.

This work was done in collaboration with Prof. Wayne Wolf's group at Princeton University. In particular, I would like to mention the help of Fiorella Haim and Ivan Corretjer from University of Maryland for the hardware implementation aspect of this work. [46][20][47].

5.1. Description of the algorithm

As a consequence of continually-improving CMOS technology, it is now possible to develop "smart camera" systems that not only capture images, but also process image frames in sophisticated ways to extract "meaning" from video streams. One important application of smart cameras is gesture recognition from video streams of human subjects. In the gesture recognition algorithm discussed in [55], for each image captured, real-time image processing is performed to identify and track human gestures. As the flow of images is increased, a higher level of reasoning about human gestures becomes possible. This type of processing occurs inside the smart camera system using advanced very large scale integration (VLSI) circuits for both low-level and high-level processing of the information contained in the images. Figure 5.1 gives an overview of the smart camera gesture recognition algorithm.

The functional blocks of particular interest in this chapter are the low-level processing components *Region*, *Contour*, *Ellipse*, and *Match* (within the dotted rectangle in Figure 5.1). Each of these blocks operate at the pixel level to identify and classify human body parts in the image, and are thus good candidates for implementation on a high performance field-programmable gate array (FPGA).

The computational core of the block diagram in Figure 5.1 can be converted from being an intuitive flow diagram to a precise behavioral representation through integration of HPDF modeling concepts. This exposes significant patterns of parallelism and of predictability, which together with application specific optimizations, help us to map the application efficiently into hardware.

The front-end processing is performed by region extraction (Region), which accepts a set of three images as inputs (we will refer to this set as an image-group from now on). The input images constituting the image-group are in the YC_rC_b color space in which Y represents the intensity and C_r, C_b represents the chrominance components of the image.

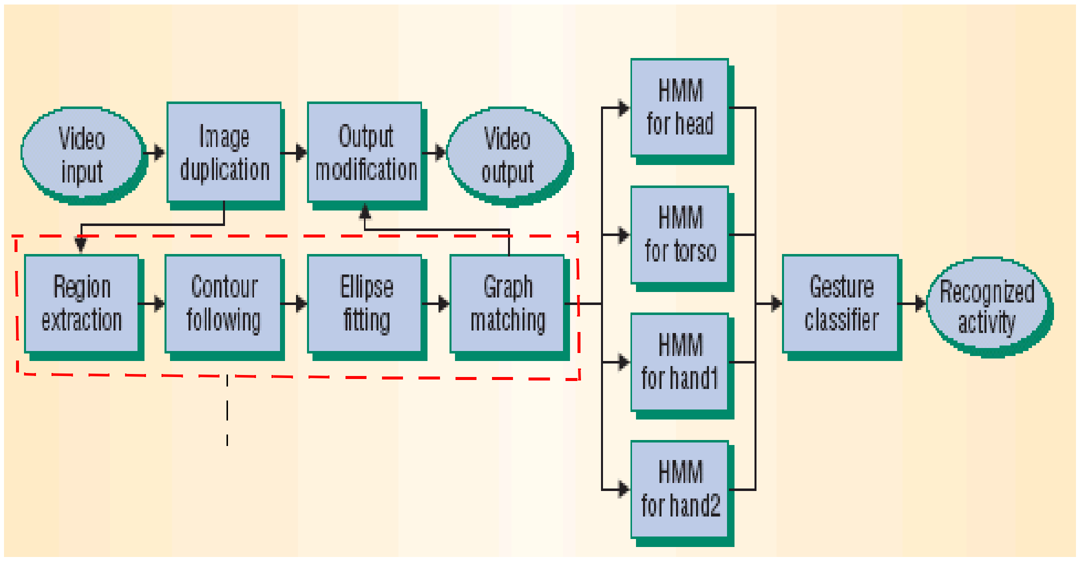


Figure 5.1 Block level representation of the smart camera algorithm [55].

In the current application input, chrominance components are downsampled by a factor of two. Thus, the three images in the image-group sent as input to Region extraction are:

- The Y component, (Image1 in Figure 5.6);
- the background (Image2 in Figure 5.6); and
- the downsampled C_r, C_b components together (Image3 in Figure 5.6).

The image with background regions is used in processing the other two images, which have foreground information as well. In one of the foreground images, the Region block marks areas that are of human skin-tones, and in the other, it marks areas that are of non-skin tone. Each of these sets of three images is independent of the next set of three, revealing image-level parallelism.

Additionally, modeling the algorithm with finer granularity (Section 5.2.3.) exposes that the set of three pixels from the corresponding coordinates in the images within an image-group are independent of any other set of pixels, leading to pixel-level parallelism. This has been verified by simulating the model for correct behavior. Furthermore, the operations performed are of similar complexity, suggesting that a synchronous pipeline implementation with little idle time between stages is possible.

After separating foreground regions into two images, each containing only skin and non-skin tone regions respectively, the next processing stage that occurs is contour following (Contour). Here, each image is scanned linearly pixel-by-pixel until one of the regions marked in the Region stage is encountered. For all regions in both images (i.e., regardless of skin or non-skin tone), the contour algorithm traces out the periphery of each region, and stores the (x, y) locations of the boundary pixels. In this way, the boundary pixels making up each region are grouped together in a list and passed to the next stage.

The ellipse fitting (Ellipse) functional block processes each of the contours of interest and characterizes their shapes through an ellipse-fitting algorithm. The process of ellipse fitting is imperfect and allows for tolerance in the deformations caused during image capture (such as objects obscuring portions of the image). At this stage, each contour is processed independently of the others, revealing contour-level parallelism.

Finally, the graph matching (Match) functional block waits until each contour is characterized by an ellipse before beginning its processing. The ellipses are then classified into head, torso, or hand regions based on several factors. The first stage attempts to identify the head ellipse, which allows the algorithm to gain a sense of where the other body parts should be located relative to the head. After classifying the head ellipse, the algorithm proceeds to find the torso ellipse. This is done by comparing the relative sizes and locations of ellipses adjacent to the head ellipse, and using the fact that the torso is usually larger by some proportion than other regions and that it is within the vicinity of the head. The conditions and values used to make these determinations are part of a piece wise quadratic Bayesian classifier that only requires the six characteristic parameters from each ellipse in the image [55].

5.2. Modeling the Single-Camera Gesture Recognition Algorithm

In this section, we model the gesture recognition algorithm using both PSDF and HPDF, and then show some application specific optimizations that are aided by the HPDF representation.

5.2.1. Modeling with PSDF

As mentioned in Section 2.1.4., PSDF imposes a hierarchy discipline. The gesture recognition algorithm is modeled using PSDF in Figure 5.2. At the uppermost level, the GesRecog subsystem has empty init and subinit graphs, and GesRecog.body is the body graph for the subsystem that has two hierarchical subsystems — H_E and H_M. The subsystems H_E and H_M in turn each have two input edges. On one of these edges, one token is consumed; this token provides the number of tokens (for example, the value of p_2 on the edge between C and H_E in Figure 5.2) that is to be consumed on the other edge, which is edge that contains the actual tokens that are to be processed.

The body graph of H_E has the actor E embedded inside. $H_E \cdot init$, which is called once per iteration of the GesRecog subsystem, has one actor in the graph. This actor sets the parameters $p_1 = p_2$ in the body graph. The $H_E \cdot subinit$ graph has one actor, which sets p_2 in $H_E \cdot body$ with the value sent by the actor C. D_1 is a dummy “gain” actor

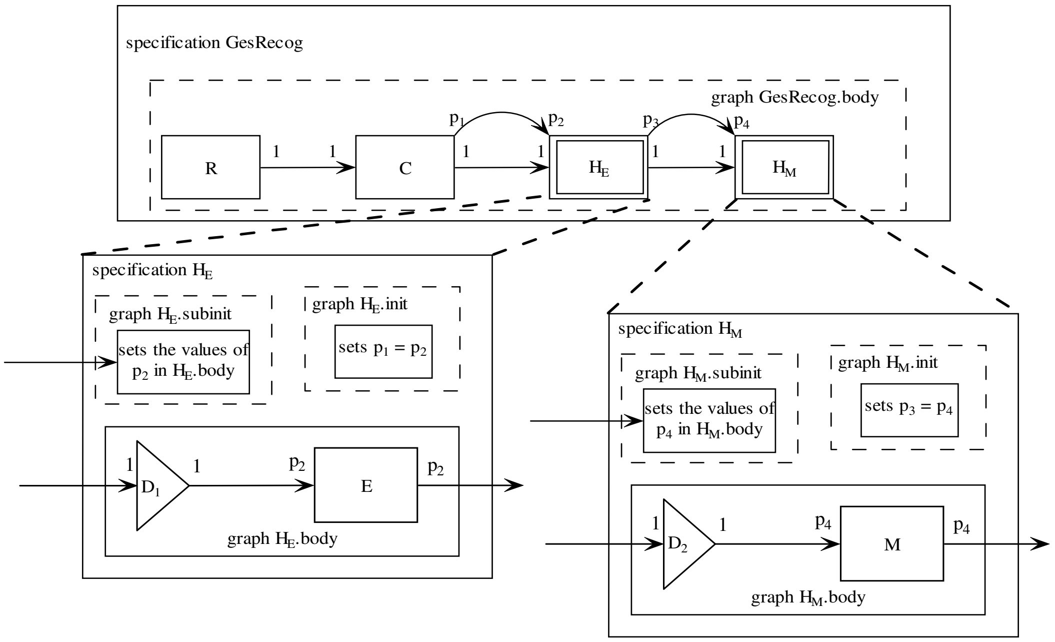


Figure 5.2 PSDF modeling of the Gesture Recognition application.

required so that the schedule in the body graph is p_2D_1E to accommodate for p_2 tokens as input to E. Analogous behavior is seen in $H_M \cdot init$, $H_M \cdot subinit$, and $H_M \cdot body$.

5.2.2. Modeling with HPDF over SDF

We prototyped an HPDF-based model of the gesture recognition algorithm in Ptolemy II [17], a widely-used software tool for experimenting with new models of computation and integrating different models of computation (See Figure 5.4). Here, we applied SDF as the base model to which the HPDF meta-model is applied. Our prototype was developed to validate our HPDF representation of the application, simulate its functional correctness, and provide a reference to guide the mapping of the application into hardware.

In the top-level, the HPDF application representation contains four hierarchical actors (actors that represent nested subsystems) — Region, Contour, Ellipse and Match — as shown in Figure 5.3. The symbols on the edges represent the numbers of data values produced and consumed on each execution of the actor. Here n and p are parameterized data transfer rates that are not known statically. Furthermore, the rates can vary during execution subject to certain technical restrictions that are imposed by the HPDF model, as described in Section 3.2.1.

5.2.3. Modeling with HPDF over CSDF

We have further refined our model for the gesture recognition algorithm using CSDF [20] as the base model for HPDF. Figure 5.5 shows that Region can be represented

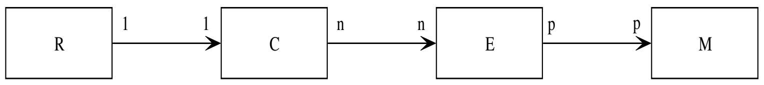


Figure 5.3 HPDF model of the application with parameterized token production and consumption rates, where R is Region, C is Contour, E is Ellipse, and M is Match.

as a CSDF subsystem with s phases, where s is the number of pixels in one input frame, and Region can work on a per-pixel basis (pixel level parallelism). On the other hand, Figure 5.5 suggests that Contour needs the whole image frame to start execution.

5.2.4. Modeling the actors

By examining the HPDF graph in conjunction with the intra-actor specifications (the actors were specified using Java in our Ptolemy II prototype), we derived a more detailed representation as a major step in our hardware mapping process. This representation is illustrated across Figure 5.6 and 5.7, which are lower level dataflow representations of Region and Contour respectively. Here, as with other dataflow diagrams, the round nodes (A, B, C, D, and E) represent computations, and the edges represent unidirectional data communication.

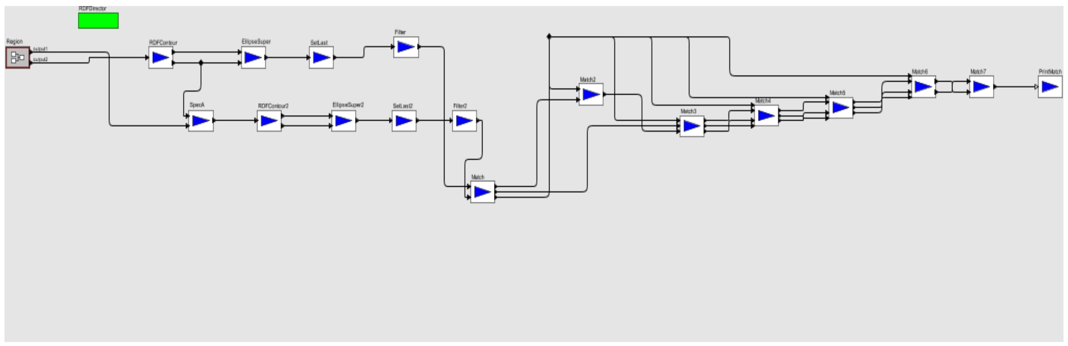


Figure 5.4 The HPDF graph of the application as shown in Figure 5.3 with flattened hierarchy for C, E, and M in Ptolemy II.

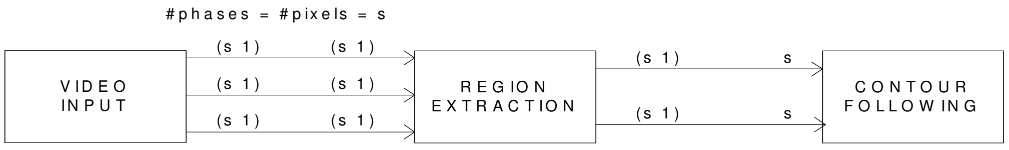


Figure 5.5 Model of the static part of the system

Figure 5.6 and 5.7 are created by hand while mapping Region and Contour to dataflow structures, and the actors A through E are each implemented in a few lines of Java code. These are more refined dataflow representations of the actors in the original HPDF representation. This kind of dataflow mapping from the corresponding application is a manual process, and depends on the expertise of the designer as well as the suitability of the form of dataflow that is being applied. In this particular case, the actors A to E represent the following operations ($ImageI$ here represents one pixel from the corresponding image and the algorithm runs for all the pixels in those images, $thold_i$ represents threshold values described in the algorithm):

A represents $abs(Image1 - Image2)$;

B represents $if(Image3 > thold_1)$;

C represents $if(((A) > thold_2) \wedge (thold_3 > Image1 > thold_4))$;

D represents $if(A > thold_5)$; and

E represents $\overline{C}D + C\overline{B}$.

The square nodes in Figure 5.6 represent image buffers or memory, and the diamond-shaped annotations on edges represent delays. The representation of Figure 5.6 reveals that even though buffers *Image1* and *Image3* are being read from and written into, the reading and writing occur in a mutually non-interfering way. Furthermore, separating the two buffers makes the four stage pipeline implementation a natural choice.

In Contour (Figure 5.7), the dotted edges represent *conditional data transfer.* In each such conditional edge, zero or one data item can be produced by the source actor depending on its input data. More specifically, in Figure 5.7 there will either be one data value produced on the edge between A and B or on the self looped edge, and the other edge will have zero data items produced. The representation of Figure 5.7 and its data transfer properties motivated us to map the associated functionality into a four stage, self-timed process.

5.3. Modeling the Distributed Gesture Recognition Algorithm

In this section, we provide the HPDF model for the distributed gesture recognition algorithm, which is an extension to the single-camera gesture recognition algorithm presented in Section 5.2.

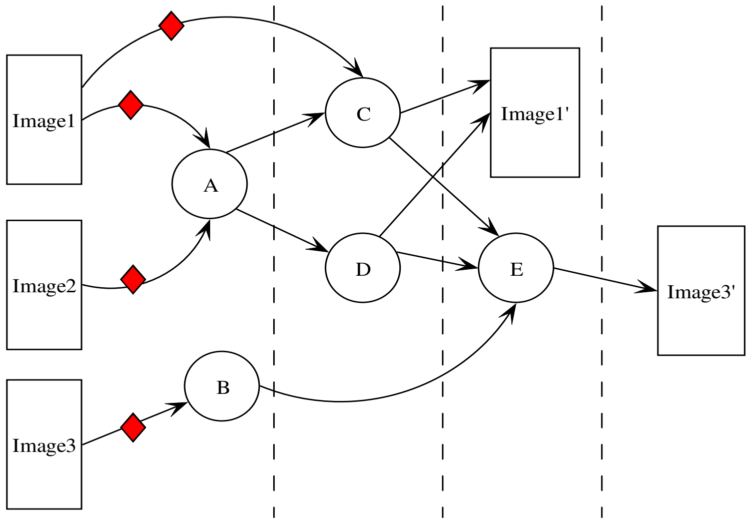


Figure 5.6 Region is shown to be broken into a four stage pipeline process.

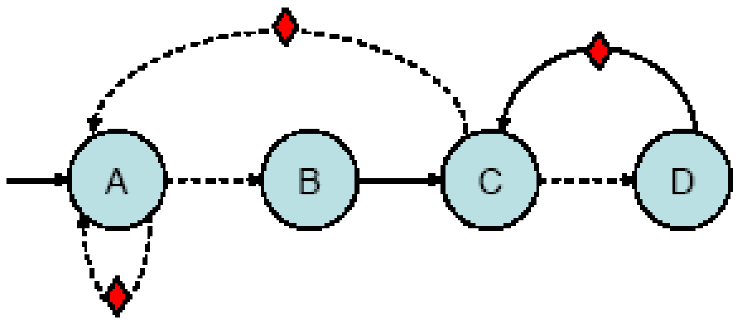


Figure 5.7 Contour is shown to have *conditional edges* and serial execution. This structure is implemented as a four-stage, self-timed process.

5.3.1. Description of the Distributed Gesture Recognition Algorithm

In the distributed version of the gesture recognition algorithm, a peer-to-peer algorithm was proposed in [34]. In the algorithm, two new terminologies were introduced. *Dominance* of a contour is determined by the number of pixels of the same contour captured by a camera — which means that when a contour is captured by more than one camera, after matching the corresponding contours across multiple cameras, the camera with the highest resolution of that contour is said to "dominate" that contour. An object is made of a few contours. *Ownership* of an object is determined by the largest size contour (for example a torso in a body part), so the camera which "dominates" the largest size contour is said to own the object.

With these terminologies in mind, a brief description of the distributed gesture recognition algorithm is presented. The algorithm assumes that the distributed cameras have some knowledge of the topology of the network — knowledge of the coordinates of their immediate neighbors would suffice. The algorithm also assumes that the regions of overlap in the frames of neighboring cameras are known beforehand.

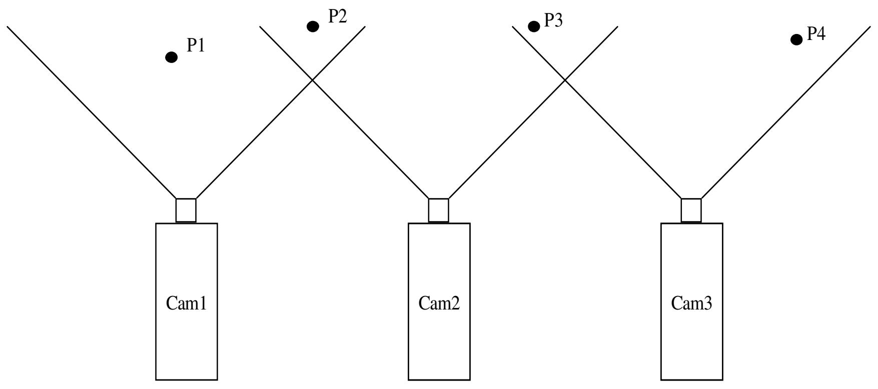


Figure 5.8 The geometry of the smart cameras in the multiple camera system.

The 'Contour' and 'Ellipse' has added logic to handle distributed gesture recognition. Depending on the load on the network of the cameras, 'Contour' or 'Ellipse' might decide to send data to its neighbors. The trade-off being amount of data being sent (more data to be sent in 'Contour' than in 'Ellipse') against better detection of 'dominance' and hence 'ownership' of objects. In addition to performing the operations explained in Section 5.1., 'Contour' (or 'Ellipse') check if the contour (or ellipse) detected lies inside or near the overlapping region with a neighboring camera. It then transmits all such contours (or ellipse) to its neighbors. Since coordinates of the neighboring cameras are known, the transferred information can be transformed to the destination coordinates. Once the ownership of an object is determined through dominance of the major part of the object (like dominance of the torso for a human body determines ownership of a human body), other cameras would just send data regarding other parts of the object and leave the application of the rest of the algorithm on the owner camera.

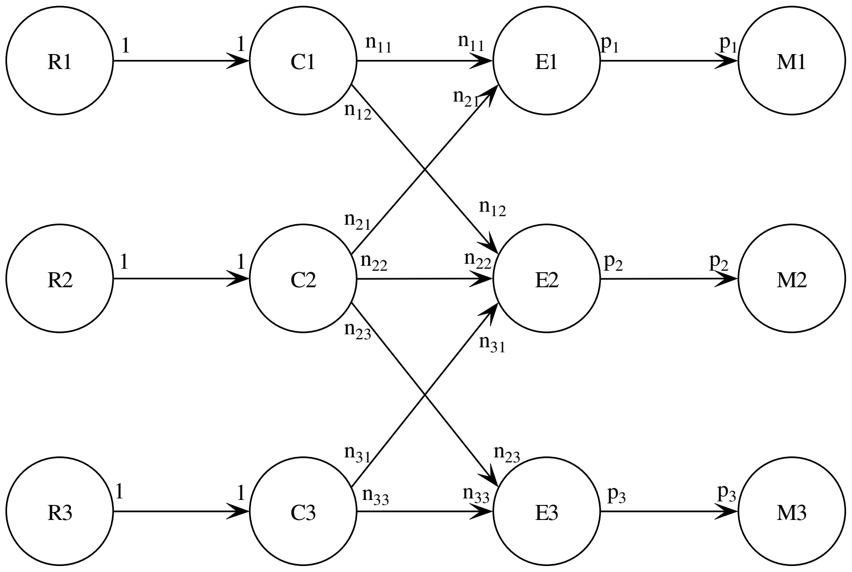


Figure 5.9 HPDF model of the distributed gesture recognition algorithm for the network in Figure 5.8, assuming contour is transmitted across cameras.

5.3.2. Application modeling

We apply HPDF to model the distributed gesture recognition algorithm. The geometry of the smart cameras is as shown in Figure 5.8 where *Cam1*, *Cam2*, *Cam3* are three representative cameras with overlapping field of views. $P1$, $P2$ etc. are objects in view, where $P1$ is entirely in the field of view of Cam1 and hence no data sharing is done between the cameras. However, $P2$, $P3$ are in the overlapping region of multiple cameras and hence sharing of information is essential. Figure 5.9 shows the HPDF over SDF model of the distributed smart camera network when transmission across cameras are done in contour level. n_{ij} in the figure represents n_{ij} contours are in or near the overlapping region between camera i to camera j and hence transmitted from C_i to E_j. Note that n_{ij} might not be equal to n_{ji} as some contours might not be inside the overlapping region, but close to it. Note that n_{ij} and p_is are parameters and equal (homogeneous) across an edge. Hence Figure 5.9 is the HPDF representation of the distributed gesture recognition algorithm. Contours in Figure 5.9 are modified version of contours in Figure 5.3, we take a closer look in Figure 5.10. C is Figure 5.10 is same as the contour in the

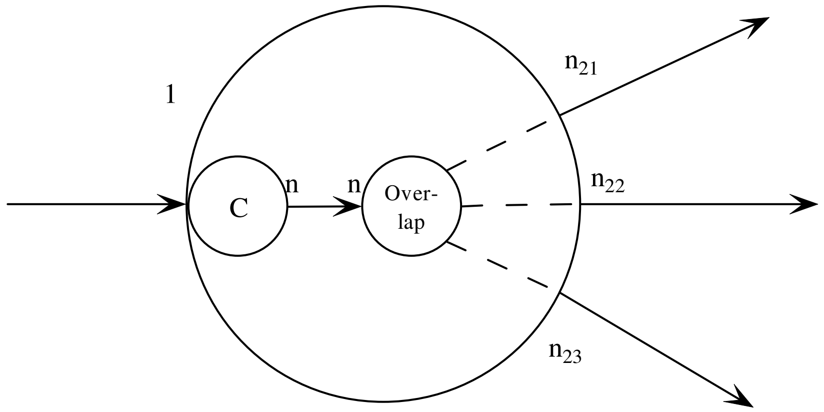


Figure 5.10 A closer look at the modified Contour actor in the distributed gesture recognition algorithm.

single camera algorithm, which is now followed by an “overlap” actor, which determines the number of contours that are in (or near) the overlap region with its neighbors. Hence, $n = n_{21} + n_{22} + n_{23}$.

We presented to HPDF modeling of the distributed gesture recognition algorithm to exemplify the robustness of HPDF modeling techniques over a wide variety of application. For the rest of the chapter, we discuss the single-camera gesture recognition algorithm and it is interchangeably used for “gesture recognition algorithm”.

5.4. From the Model to Hardware

Dataflow modeling of an application has been used extensively as an important step for verification, and for performing methodical software synthesis [17]. Hardware synthesis from SDF and closely related representations has also been explored (e.g., see [24, 45, 54]). In this section, we explore the hardware synthesis aspects for class of dynamic dataflow representations that can be modeled using HPDF. Compared to PSDF, HPDF can be more suited to intuitive, manual hardware mapping because of its non-hierarchical dynamic dataflow approach. For example, Figure 5.3 might suggest a power-aware self-timed architecture, where the different hardware modules hibernate and are occasionally awakened by the preceding module in the chain. Alternatively, it can also suggest a pipelined architecture with four stages for high performance. The designer can also suggest multiple instantiations of various modules based on applying principles of data parallelism on the dataflow graph [45]. Such application of data parallelism can systematically increase throughput without violating the dataflow constraints of the application. Hence, an HPDF model can suggest a range of useful architectures for an application, and thus aid the designer significantly in design space exploration.

In Region, the application level dataflow model (which shows pixel-level parallelism) in conjunction with actor level dataflow (which suggests a pipelined architecture) suggests that the pipeline stages should work on individual pixels and not on the whole frame for maximum throughput. On the other hand for Contour, a self-timed architecture that performs on the whole image was a natural choice.

In addition to dataflow modeling, we also applied some application specific transformations. For example, the Ellipse module utilizes floating-point operations to fit ellipses to the various contours. The original C code implementation uses a moment-based initialization procedure along with trigonometric and square root calculations. The initialization procedure computes the averages of the selected contour pixel locations and uses these averages to compute the various moments. The total computation cost is

$$5nC_+ + 6nC_- + 3nC_* + 5C_/,$$

where n is the number of pixels in the contour, and each term C_{OP} represents the cost of performing operation OP. In an effort to save hardware and reduce complexity, the following transformation was applied to simplify the hardware for calculating averages and moments:

$$mxx = \left(\sum_{i=1}^{n} \frac{(x_i - \bar{x})^2}{n} \right) \Rightarrow \left(\sum_{i=1}^{n} \frac{(x_i)^2}{n} - \overline{(x)^2} \right),$$

and similarly for mxy and myy. The computational cost after this transformation is:

$$5nC_+ + 3nC_* + 9C_/ + 3C_- + 3C_*.$$

Comparing this with the expression for the previous version of the algorithm, we observe a savings of $3nC_{-}$, which increases linearly with the number of contour pixels, at the expense of a fixed overhead $4C_{/} + 3C_{*}$. This amounts to a large overall savings for practical image sizes.

Further optimizations that were performed on the ellipse-fitting implementation included splitting the calculations into separate stages. This allowed for certain values (such as mxx, myy, mxy) to be computed in earlier stages and reused multiple times in later stages to remove unnecessary computations.

The characterization of ellipses in Match is accomplished in a serial manner, in particular, information about previously identified ellipses is used in the characterization of future ellipses. Our functional prototype of the matching process clearly showed this dependency of later stages on previous stages. The hardware implementation that we derived is similar to that of Contour, and employs a six-stage self-timed process to efficiently handle the less predictable communication behavior.

5.5. Experimental Setup

The target FPGA board chosen for this application is the multimedia and microblaze development board from Xilinx. The board can act as a platform to develop a wide variety of applications such as image processing and ASIC prototyping. It features the XC2V2000 device of the Virtex II family of FPGAs.

Some of the more important features of the board include the following.

- Five external, independent 512Kx36 bit ZBT RAMs
- A video encoder-decoder.

- An audio codec.
- Support for PAL/NTSC TV input/output.
- On-board ethernet support.
- An RS-232 port.
- Two PS-2 serial ports.
- A JTAG port.
- A System ACE-controller and Compact Flash storage device to program the FPGA.

5.5.1. ZBT Memory

One of the key features of this board is its set of five fully-independent banks of 512k x 32 ZBT RAM [58] with a maximum clock rate of 130 MHz. These memory devices support a 36-bit data bus, but pinout limitations on the FPGA prevent the use of the four parity bits. The banks operate completely independently of one another, as the control signals, address and data busses and clock are unique to each bank with no sharing of signals between the banks. The byte write capability is fully supported as it is the burst-mode, in which the sequence starts with an externally supplied address.

Due to the size of the images, we needed to store them using these external RAMs. A memory controller module was written in Verilog, simulated, synthesized, and downloaded onto the board. We then successfully integrated this module with the Region module.

5.5.2. RS-232

In order to communicate between the host PC and the board, we used the RS-232 protocol. We adapted an RS232 controller core with a wishbone interface [59] and configurable baud rate to write images from the PC to the memory. The board acts as a DCE device; we implemented the physical communication using a straight-through three wire cable (pins 2, 3 and 5) and used the Windows Hyperterminal utility to test it. This interface was integrated into the Region and Memory Controller modules and tested in the board.

Figure 5.11 illustrates the overall experimental setup, including the interactions between the PC and the multimedia board, and between the board and the HDL modules.

5.6. Design Trade-offs and Optimizations

There were various design decisions made during implementation of the algorithm, some of which were specific to the algorithm at hand. In this section, we explore in more detail the trade-offs that were present in the important design space associated with mem-

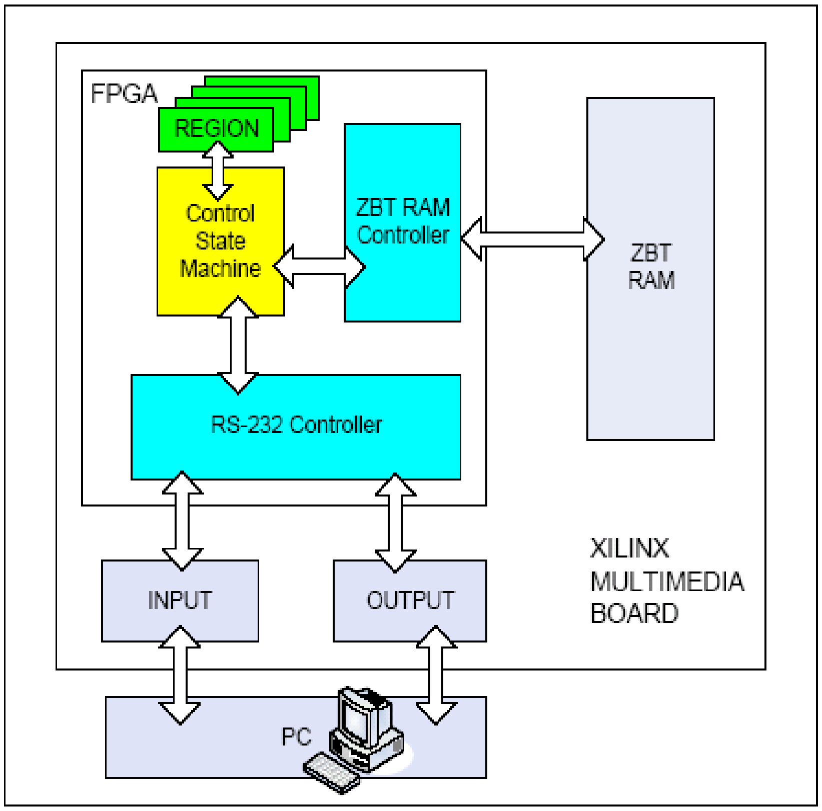


Figure 5.11 The overall setup showing interactions among various modules of our design and components of the multimedia board.

ory layout. We also present a step-by-step optimization that we performed on one of the design modules for reducing its resource requirements on the FPGA.

5.6.1. Memory Layout Trade-offs

The board memory resources are consumed by the storing of the images. Each of the 5 ZBT RAM banks can store 512 K words that are 32 bits long, for a total storage capacity of 10 Mbytes. Given that each pixel requires one byte of storage and that there are 384 x 240 pixels per image, 90 Kbytes of memory are required to store each image. The first module, Region, has 3 images as inputs, and 2 images as outputs. These two images are scanned serially in the second module, Contour. The total amount of memory needed for image storing is then 450 Kbytes, less than 5% of the external memory available on board. However, reorganization of the images in the memory can dramatically change the number of memory access cycles performed and the number of banks used. These trade-offs also affect the total power consumption.

Several strategies are possible for storing the images in the memory. The simplest one (Case 1) would be to store each of the five images in a different memory bank, using 90K addresses and the first byte of each word. In this way, the 5 images can be accessed in the same clock cycle (Figure 5.12a). However, we can minimize the number of memory banks used by exploiting the identical order in which the reading and writing of the images occurs (Case 2). Thus, we can store the images in only two blocks, using each of the bytes of a memory word for a different image, and still access all the images in the same clock cycle (Figure 5.12b).

On the other hand, a more efficient configuration in order to minimize the number of memory access cycles (Case 3) would be to store each image in a different bank, but using the four bytes of each memory word consecutively (Figure 5.12c). Other configurations are possible, for example, (Case 4) we can have two images per bank, storing 2 pixels of

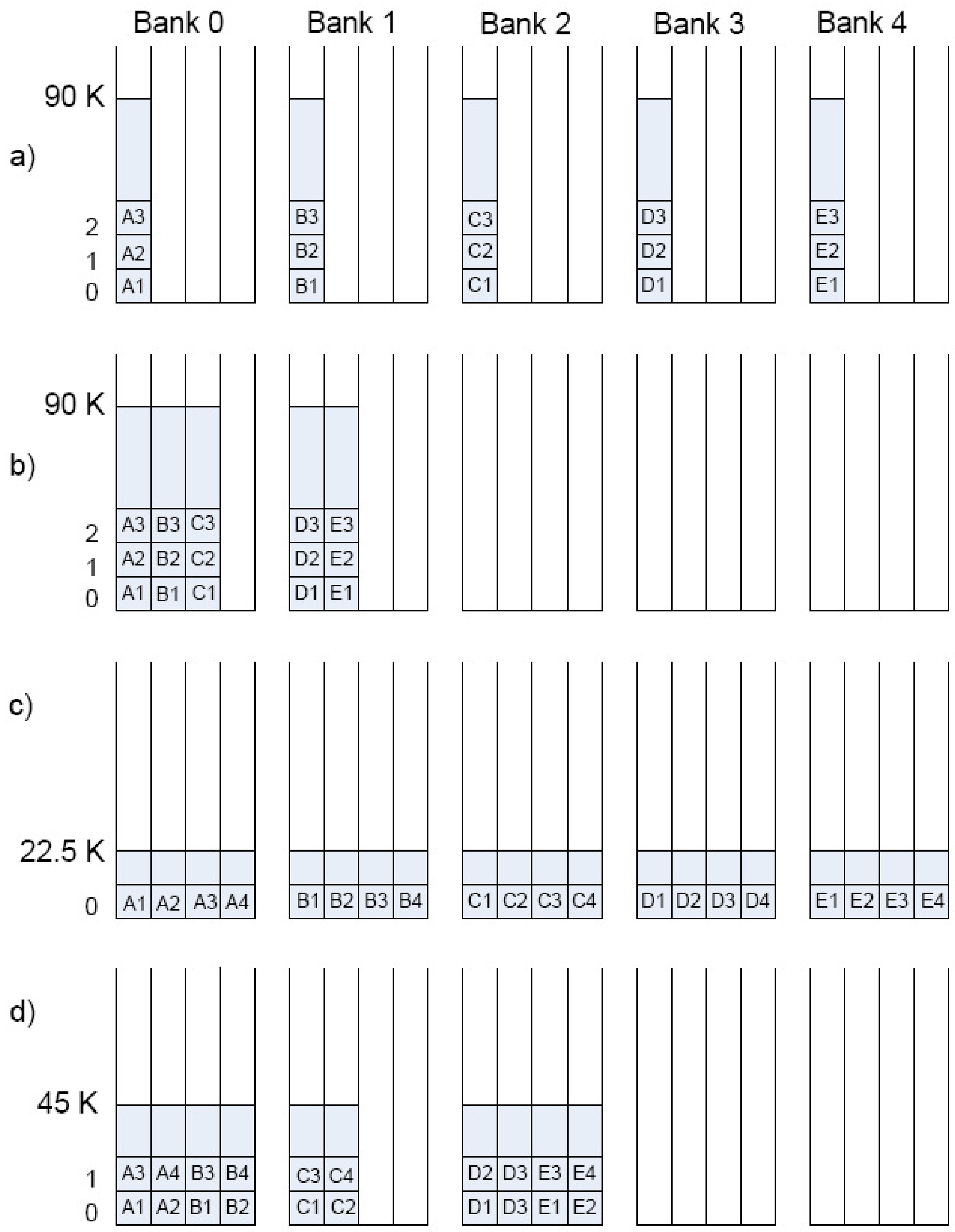


Figure 5.12 Image storage distribution. a) Case1: Each image in a separate bank using only the first byte of the first 90K words of the memory. b) Case2: Three images in bank 0 and two in bank. c) Case3: Each image in a separate bank but all four bytes used in each word, using 22.5K words. d) Case4: Images stored in three banks, each using 2 bytes of the first 45K words.

each image in the same word (Figure 5.12d). Table 5 1 summarizes the number of banks and memory access cycles needed for each of these configurations.

Case 3 appears to be the most efficient memory organization. Here, the time associated with reading and writing of the images is 69120 memory access cycles, and the total number of memory access cycles is also the lowest, 161280. This reduced number of memory access cycles suggests that power consumption will also be relatively low in this configuration. Figure 5.12 illustrates all of the cases discussed above.

5.6.2. Floating Point Optimizations

Floating-point operations are used throughout the implementation of the Ellipse and Match blocks. The Ellipse block processes the (x, y) location of every pixel that is along the border of a contour. From these locations, averages, moments, and rotation parameters are derived that characterize a fitted ellipse to the particular contour. An ellipse is uniquely defined by a set of five parameters — the center of the ellipse (*dxAvg*, *dyAvg*), its orientation (*rotX*) and the lengths of its major and minor axes (*aX*, *aY*) [26]. Here, the terms in the parenthesis are the abbreviations used in this thesis (See Figure 5.16).

Table 5 1. Comparison of different memory layout strategies.

Configuration	Banks Used	Read cycles-Region	Write cycles-Region	Read cycles-Contour	Total non-overlapping cycles	Total number of cycles
Case 1	5	92160X3	92160X2	184320X1	276480	645120
Case2	2	92160X1	92160X1	184320X1	276480	368640
Case3	5	23040X3	23040X2	46080X1	69120	161280
Case4	3	46080X2	46080X1	92160X1	138240	230400

Due to the non-uniform shapes of the contours, the ellipse fitting is imperfect and introduces some approximation error. By representing the parameters using floating point values, the approximations made have more precision than if integer values were used. To further motivate the need for floating point numbers, the Match block uses these approximations to classify each ellipse as a head, torso, or hand. To do so, the relative locations, sizes, and other parameters are processed to within some hard-coded tolerances for classification. As an example, the algorithm considers two ellipses within a distance Y of each other with one being around X times larger than the other to be classified as a head/torso pair. It is because of the approximations and tolerances used by the algorithm that floating-point representations are desirable, as they allow the algorithm to operate with imperfect information and still produce reasonable results

For our implementation, we used the IEEE 1076.3 Working Group floating-point packages, which are free and easily available from [57]. These packages have been under development for some time, have been tested by the IEEE Working Group, and are on a fast track to becoming IEEE standards. Efficient synthesis of floating point packages involved the evaluation of floating-point precision required by the smart camera algorithm. The C code version of the algorithm utilizes variables of type `double`, which represent 64-bit floating-point numbers. Utilizing the floating-point library mentioned before, we were able to vary the size of the floating-point numbers to see how the loss in precision affected the algorithm outputs as well as the area of the resulting synthesized design.

We reduced the number of bits used in the floating-point number representation and performed a series of simulations to determine the loss in accuracy relative to the original 64-bit algorithm. Figure 5.16 shows the resulting root-mean-square (RMS) error for various sizes of floating-point numbers. For the smart camera algorithm, we found that the range from 20 to 18 bit floating-point number representations gave sufficient accuracy, and any lower precision (such as 16-bit) caused a dramatic increase in the errors. The values that are most affected by the loss in precision are *rotX*, *aX*, and to some extent *aY*. These values depend on the computation of the *arctangent* function. As the precision is lowered, small variations cause large changes in the output of *arctangent*. The *dxAvg* and *dyAvg* parameters are not as affected by the loss in precision, as the only computations they require are addition and division.

Since the *arctangent* and *sqrt* functions have domains from ∞ to $-\infty$, and *sqrt* also has a range of ∞ to $-\infty$, theoretically the need might arise for expressing the whole real data set. The input image data set on which our experiment was performed was relatively small, and no prior knowledge was available of the range of values needed to be expressed for a new data set that the algorithm might be subjected to. Thus our choice of floating-point over fixed-point for implementation and simulations was motivated by the lack of a quantization error metric and lack of predictability of the input data set for the low-level processing of the gesture recognition algorithm. Also this low-level processing is a precursor to higher-level gesture recognition algorithms for which, to our knowledge, no prior metric has been investigated to determine how errors in low-level processing effect the ability of the higher level processing to correctly detect and process gestures. Through further simulation and analysis it may be possible to also determine suitable fixed-point precision, however, care must be taken to ensure reliable results especially for the *arctangent* function.

Table 5 2 presents the area in number of look-up tables required for each of the floating-point number representations. As expected, when we reduce the number of bits, the area of the resulting design decreases, but at the cost of lost precision.

The number of available LUTs in an FPGA varies heavily depending on the family of the FPGA and also on the specific devices within the family. For example, in the Virtex II family of the Xilinx FPGAs, the XC2V1000 contains 10,240 LUTs, the XC2V2000 contains 21,504 LUTs, and the XC2V8000 contains 93,184 LUTs. In the Xilinx Virtex II Pro family, the XC2VP7 contains 9,856 LUTs and XC2VP100 contains 88,192 LUTs (other intermediate devices in the family are omitted). In our experimental setup, we used the XC2V2000 FPGA, which did not have enough resources for us to implement Ellipse with the desired precision on the board (our current implementation involves 16-bit floating point numbers and additional optimizations) but a larger FPGA would have sufficed.

5.7. Results

In this section, we present some representative results from both software and hardware implementations of the gesture recognition algorithm.

Table 5 2. Synthesis results.

Number of bits	Area (in LUTs)
32-bit	110092
21-bit	54944
20-bit	46951
18-bit	41088
16-bit	23923

We developed a software implementation of the gesture recognition algorithm on a Texas Instruments (TI) programmable digital signal processor. We evaluated this implementation using the TI Code Composer Studio version 2 for the C'6xxx family of programmable DSP processors. The application, when implemented with our HPDF model, for a C64xx fixed-point DSP processor has a runtime of 21405671 cycles, and with a clock period of 40 ns, the execution time was calculated to be 0.86 sec. The scheduling overhead for the implementation is minimal, as the HPDF representation inherently leads to a highly streamlined quasi-static schedule. The worst-case buffer size for an image of 348 X 240 pixels was 184 kilobytes on the edge between Region and Contour, 642 Kb between Contour and Ellipse and 34 Kb between Ellipse and Match for at total of 860

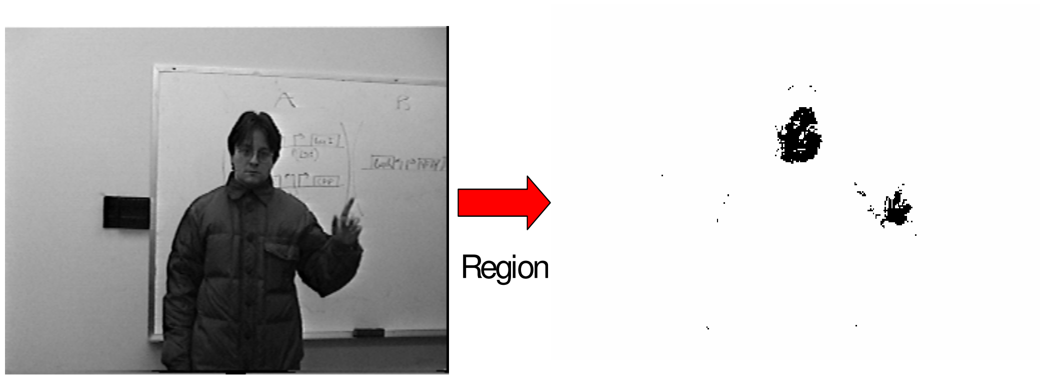


Figure 5.13 Our HDL representation of Region transforms the image on the left to the output on the right.

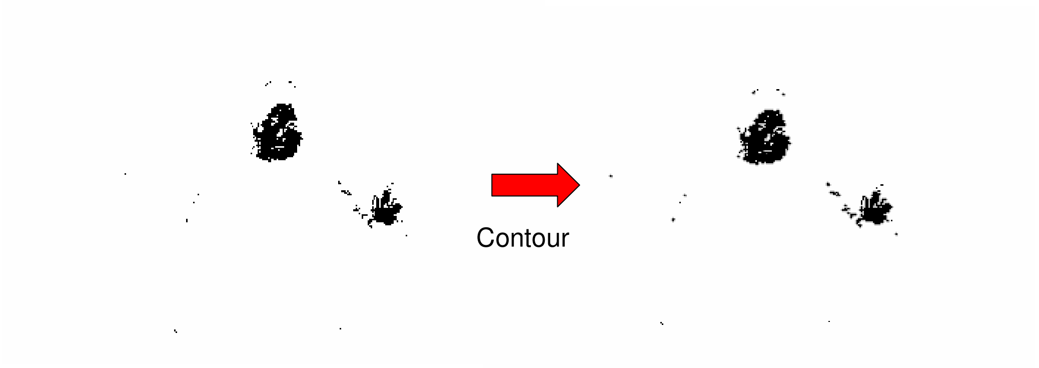


Figure 5.14 Actual transformation to the image done by HDL representation of Contour.

kilobytes. The original code (without modeling) had a run-time of 27741882 cycles, and with the same clock period of 40ns, the execution time was 1.11 sec. Thus, HPDF-based implementation improved the execution time by 23 percent.

To further take advantage of the parallelism exposed by HPDF modeling, we implemented both the Region and Contour functions in hardware. We used Modelsim XE II 5.8c for HDL simulation, Synplify Pro 7.7.1 for synthesis of the floating-point modules, and Xilinx ISE 6.2 for synthesis of non-floating-point modules, and for downloading the bitstream into the FPGA. Figure 5.13 and 5.14 show the outputs of the first two processing blocks (Region and Contour respectively) after they were implemented in HDL. Compar-

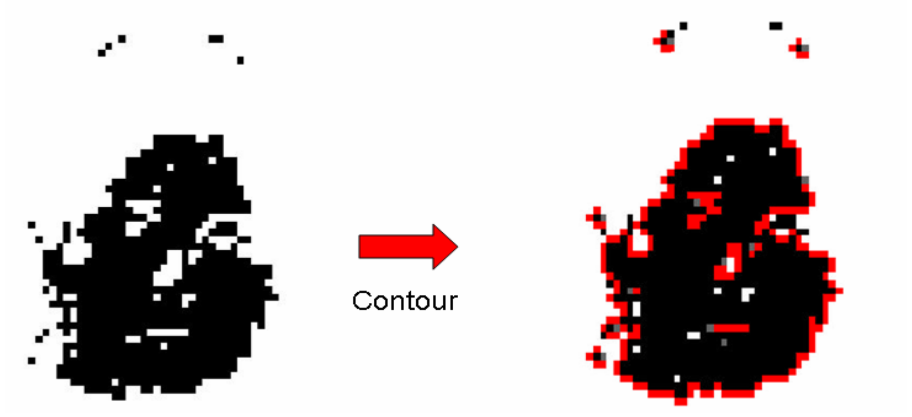


Figure 5.15 Part of Figure 5.14 zoomed-in and colored to show the effect of Contour.

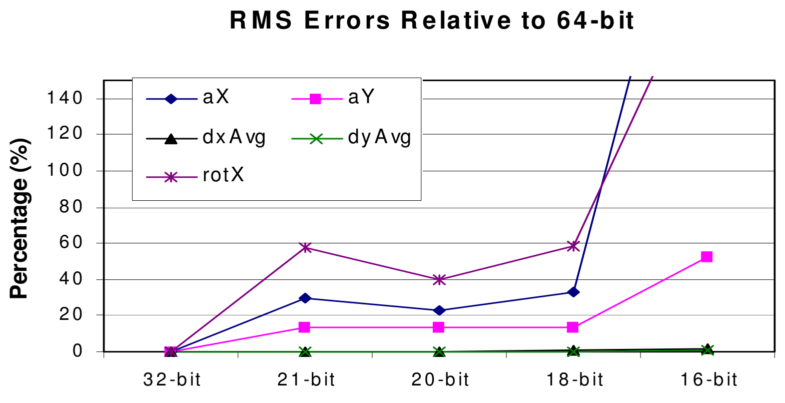


Figure 5.16 Comparison of percentage RMS error for different-length floating point representations, normalized to a 64-bit floating point representation.

ing these outputs with the outputs of the software implementation verified the correctness of the HDL modules.

5.8. Conclusion

In this chapter, we have developed homogeneous parameterized dataflow (HPDF), an efficient meta-modeling technique for capturing a commonly-occurring, restricted form of dynamic dataflow that is especially relevant to the computer vision domain. HPDF captures the inherent dataflow structure in such applications without going into more complicated hierarchical representations or into more general dynamic dataflow modeling approaches where key analysis and synthesis problems become impossible to solve exactly.

We have also developed and applied a novel design methodology for effective platform-specific FPGA implementation of computer vision applications based on the HPDF modeling technique. In particular, we have used HPDF to model a gesture recognition algorithm that exhibits dynamically-varying data production and consumption rates between certain pairs of key functional components.

The top-level HPDF model and subsequent intermediate representations that we derived from this model naturally suggested efficient hardware architectures for implementation of the main subsystems. The hardware description language (HDL) code for the four modules of the algorithm was developed following these suggested architectures. The modules were then verified for correctness, and synthesized to target a multimedia board from Xilinx. Memory management and floating point handling also played a major role in our design process. We explored various trade-offs in these dimensions and through the framework of our HPDF-based application representation, we integrated our findings seamlessly with the architectural decisions described above.

Chapter 6. HPDF-based Hardware Mapping Case Study: Image Registration

6.1. Introduction

Image registration is a fundamental requirement in medical imaging and an essential first step for meaningful multimodality image fusion and accurate serial image comparison. It is also a prerequisite for creating population-specific atlases and atlas-based segmentation. Despite the existence of powerful algorithms and clear evidence of clinical benefits they can bring, the clinical utilization of image registration remains limited. The slow speed (i.e., long execution time) of fully automatic image registration algorithms especially for 3D images has much do with this lack of clinical integration and routine use.

This chapter focuses on image registration algorithms that must be executed under real-time performance constraints. In some cases, for example, visual accessories in surgical applications must meet stringent performance criteria in order to provide adequate response and interactivity to surgeons. Hardware implementation is one way to speed-up applications over existing software implementations. However, designing hardware requires significantly higher turn-around time, and is more error prone compared to software implementation. Systematic methods based on precise application modeling abstractions and associated hardware mapping techniques are therefore desirable, since such methods make the design process more structured, while at the same time exposing opportunities for system-level performance optimization.

In this chapter, we develop such a structured design methodology in the context of image registration. Our approach starts with capturing the high level algorithm structure through a carefully-designed, coarse-grain dataflow model of computation. As a result, designers are exposed to various design points in the design space which represents an area-performance trade-off for different configurations of the circuit to customize their final implementation based on certain input characteristics that we define later. We also develop methods to analyze this dataflow representation to systematically provide a hardware implementation that dynamically optimizes its processing structure in response to the particular image registration scenario in which it operates.

Image registration algorithms have the potential to be mapped onto hardware for efficient execution. Fast Automatic Image Registration (FAIR) [10] is such an architecture proposed by Castro-Pareja et. al. for accelerated hardware implementation of rigid image registration. FAIR is optimized and fine tuned for the partial volume interpolation based image registration by means of pipelining, parallel memory access, and distributed processing. FAIR created a proof-of-concept implementation, and achieved greater than an order of magnitude speedup for registration of multimodality images (MR, CT and PET) of the human head, PET and CT images of the thorax and abdomen, and 3D ultrasound and SPECT images of the heart [50]. As a demonstration of single modality image registration, FAIR used the accelerated implementation also for registration of pre- and post-exercise 3D ultrasound images of the heart [50].

Several clinical applications to benefit from the proposed work include whole-body PET/CT registration [49], virtual colonoscopy [9] and image registration tasks involving pre- and intra-operative images in the context of image-guided surgeries [13]. The overall benefits will extend to numerous other applications being developed by researchers worldwide.

In this chapter, we build on our experience with architectures for image registration by developing and applying novel dataflow-based models and analysis methods of image registration applications. These methods provide a framework for mapping and high-level optimization of these applications onto embedded architectures. Using this framework, we evaluate trade-offs between different design points and propose a new dynamically reconfigurable architecture for image registration that optimizes its processing structure adaptively based on relevant characteristics of its input. This methodology is more generic and further low-level fine-tuning for specific applications can be performed on top of the implementation derived through the dataflow.

This work was a joint effort between us and the research group of Dr. Raj Shekhar at University of Maryland, Medical College. In particular, I would like to thank Yashwant Hemaraj for the simulation and initial synthesis of the Verilog code for the system.[21][48].

6.2. FPGA technology

In this work, we target our hardware optimization framework to an FPGA device, the Altera StratixII EP2S15F484C5. A major advantage of FPGA technology is the potential for dynamic reconfiguration of the processing structure. In the context of FPGA implementation, dataflow is especially useful because it effectively exposes application concurrency, and facilitates configuration of and mapping onto parallel resources. This opens up design space exploration opportunities for meeting different user constraints, and achieving different implementation trade-offs. However, streamlining the use of dataflow technology is challenging because it requires careful mapping of application characteristics into the graphical and actor-based modeling abstractions of dataflow, and because of the associated optimization issues, while exposed more effectively for signal processing applications compared to other modeling abstractions, are usually NP-complete to solve exactly [4]. This chapter addresses these challenges for the image registration domain.

6.3. Application Description

Image registration is the process of aligning two images that represent the same feature. So it can be thought of as a mapping function $F:I \rightarrow R$ that accepts an image to be mapped (also called the floating image I) and returns the image transformed such that it can map directly onto another image (also called the reference image R). Medical image registration concentrates on aligning two or more images that represent the same anatomy from different angles, obtained at different times, and/or using different imaging techniques. Image registration is a key feature for a variety of imaging techniques and there two main algorithmic approaches — *linear* and *elastic*. A linear transformation can be approximated by a combination of rotation, translation and scaling coefficients while an elastic approach is based on nonlinear continuous transformations, and is implemented by finding correlations among meshes of control points. Our study concentrates on the linear approach. As mentioned earlier, real-time image registration is essential in the medical field for enabling image-guided treatment procedures, and pre-operative treatment planning.

There are many approaches to 3D image registration [36]. But for hardware implementation a robust, accurate, flexible algorithm which does not require manual feedback is preferred. Algorithms based on voxel (a pixel in 3-d) similarity fulfill the above criteria better than feature-based approaches [23]. For the rest of the chapter, we use voxel which can be treated as the 3-d equivalent of a pixel. Of them, the most commonly used technique is image registration based on *mutual information* [41]. Mutual information (MI) methods have been shown to be robust and effective for multi-modal images.

6.3.1. MI-based Image Registration

Figure 6.1 represents the algorithmic flow of MI-based image registration. MI-based image registration relies on maximizing the mutual information between two images.

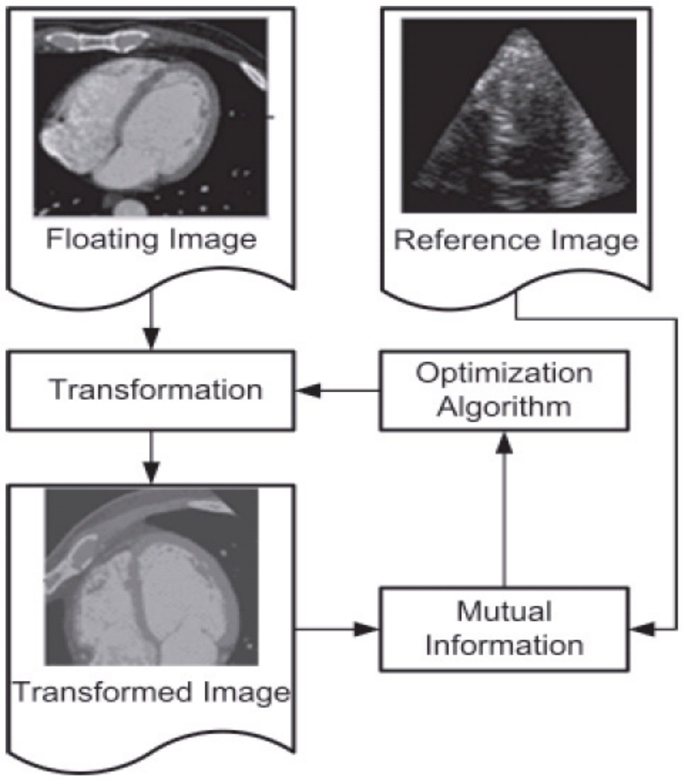


Figure 6.1 Mutual Information based Image Registration

Mutual information is a function of two 3-D images and a transformation between them. The transformation matrix contains the information about the rotation, scaling shear and translations that need to be applied to one of the images in order to map it completely to the other image so that a one-to-one correspondence is established between the coordinates of the images where they overlap. A cost function based on the mutual information is calculated from the individual and joint histograms. The transformation that maximizes the cost function is viewed as the optimum transformation. The goal of MI-based image registration is then to find this optimal transformation $\hat{T}$:

$$\hat{T} = argmax_T MI(RI(x, y, z), FI(T(x, y, z))),$$

Here, *RI* is the reference image, and *FI* is the floating image (the image that is being registered).

6.3.2. Computation of Mutual Information

Mutual information is calculated from individual and joint entropies using the following equations.

$$MI(RI, FI) = H(RI) + H(FI) - H(RI, FI),$$

$$H(FI) = -\Sigma p_{FI}(a) log p_{FI}(a), H(FI) = -\Sigma p_{FI}(a) log p_{FI}(a)$$

$$\text{and } H(RI, FI) = -\Sigma p_{RI, FI}(a, b) log p_{RI, FI}(a, b), \tag{6.1}$$

where $H(RI)$, $H(FI)$, $H(RI, FI)$ and $MI(RI, FI)$ are the reference image entropy, floating image entropy, joint entropy and mutual information between the two images for a given transformation.

The mutual histogram represents the joint intensity distribution. The joint voxel intensity probability, $p_{RI, FI}(a, b)$ is the probability of a voxel in the reference image having an intensity a and the corresponding voxel for a particular transformation T in the floating image having an intensity b, can be obtained from the mutual histogram of the two images.

The individual voxel intensity probabilities are the histograms of the reference and floating images in the region of overlap of the two images for the applied transformation. The individual histograms can be computed by taking the row sum and the column sum of the joint histogram.

The calculation of mutual information starts with the accumulation of the mutual histogram values to the mutual histogram memory while every coordinate is being transformed (MH update stage). This is followed by the MI calculation stage where the values stored in the mutual histogram memory are used to find the individual and joint entropies described above.

In the MH update stage, voxel coordinates are multiplied by the transformation matrix and the resultant coordinates obtained are used to update the joint histogram. Since the new coordinates do not always coincide with the location of a voxel in the reference image interpolation schemes need to be employed. In the trilinear interpolation scheme, the new value of the floating image $FI(x', y', z')$ is calculated based on the amount of offset the new coordinates (x', y', z') have from the nearest voxel position. However this scheme introduces a new value, which makes the MH sparse and hence ineffective in MI calculation. Maes et. al. [35] showed that the partial volume interpolation scheme does not cause such unpredictable variations in the MH values as the transformation matrix changes. This method accumulates the eight interpolation weights directly into the mutual histogram instead of calculating a resultant intensity level and increment that intensity level's MH count by one, as in trilinear interpolation. Thus the partial volume interpolation scheme ensures a smooth transition in the MH memory and hence causes smooth MI changes for various transformations applied.

Constructing the mutual histogram, the first step in mutual information calculation, involves performing partial volume interpolation n times, where n is less than or equal to the number of voxels in the reference image. The number of operations in the second step, the calculation of mutual information, is a function of the size of the mutual histogram. Since the size of the mutual histogram is less than the size of the image, it is the first part which is the performance bottleneck.

It has been shown that the size of the mutual histogram can be selected as 64×64 for 8 bit images. By doing so, we can obtain a very good density of MH values while at the same time preserving the variation along the different entries.

At current microprocessor speeds, the time of mutual histogram calculation for 3-D images is dominated almost exclusively by the memory access time. Around 25 memory accesses are needed to perform partial volume interpolation per voxel of the reference image: 1 to access the reference image voxel, 8 to access the 8-voxel neighborhood in the floating image and 16 accesses to the mutual histogram memory (8 reads to get the old value in the adder and 8 writes to write back the updated value after adding the weights). Accesses to the reference image are sequential and standard caching techniques can be effectively used. The mutual histogram memory has a small size and thus accesses to it also have high locality of reference. However, the floating image is accessed in a direction across the image that depends on the transformation being applied. Unless there is no rotation component, this direction is not parallel to the direction in which voxels are stored, hence accesses have poor locality and do not benefit from memory-burst accesses or memory-caching schemes.

Speedup of registration is achieved by identifying bottleneck areas and optimizing them in order to decrease the processing time. Speedup of the algorithm can be obtained by using pipelined architectures and also by using parallel architectures [10]. Since the majority of the registration execution time is spent on calculating the mutual histogram, accelerating mutual histogram calculation has been the focus of our work. The aim of this chapter is to use dataflow graphs to describe the inherent concurrency in applications, analyze the bottleneck areas and to use the dataflow graph transformations to exploit potential areas which can be parallelized.

6.3.3. Optimization

The image registration algorithm calculates the transformation matrix for which the mutual information between the images is maximum. Initially, a small number of test transformations are applied. The values of these transformations and the MI values are stored in an *optimizer*. The optimizer outputs the values of the new transformation depending on the values of the mutual histogram in the previous iterations. Optimization of the transformation parameters depends on the nature of the images and the amount of misalignment between the two images. Some methods, such as the downhill simplex method, provide faster convergence than the others. In the simplex method, in order to optimize a transformation with m parameters, the optimizer needs to store $(m + 1)$ previous values. There is a trade-off between the convergence time and the complexity of the optimizer.

6.4. Modeling using HPDF-CSDF

In this section, we construct a hierarchical dataflow representation of the MI-based image registration algorithms and we use the HPDF meta-modeling approach integrated with CSDF for modeling lower-level, multi-phase interactions between actors which was introduced in [20]. Figure 6.2 shows our top level HPDF model of the application. Here, "$(m+1)D$" represents $(m+1)$ units of *delay*; each unit of delay is analogous to the z^{-1} operator in signal processing, and is typically implemented by placing an initial data value on the corresponding dataflow edge. The MI actor consumes one data value (*token*) on every execution. This token contains co-ordinates of the reference image and the floating image. After s executions each consuming one token (coordinate values in this case), where s denotes the size of the image, the MI actor produces the entropy between the reference and floating images. This value is then sent to the optimizer as a single token.

The optimizer, which stores the previous $(m+1)$ values to perform a simplex optimization of an m-parameter transformation vector, sends m tokens to the MI actor. Since m can vary depending on the number of parameters used to represent the desired transfor-

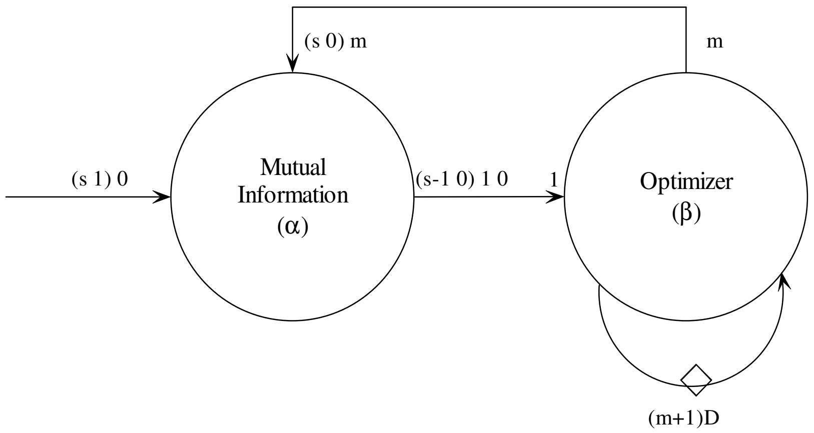


Figure 6.2 Top level model of image registration application.

mation, the associated edge represents a variable-rate edge of the HPDF graph. A valid schedule for this HPDF would be

$$(s\alpha)\beta\alpha. \tag{6.2}$$

The internal representation of the hierarchical MI actor is shown in Figure 6.3. Here, "Reference Image" (A) consumes one token (coordinates) and produces one token (intensity values at the input coordinates), and "Coordinate Transform" (B) produces one token, which represents the transformed coordinates. If this voxel is valid (i.e., the voxel coordinate falls within the floating image coordinates boundary), it is passed on to the "Weight Calculator" (D) and "Floating Image" (E).

Now since all voxels may not be valid, r tokens ($r \leq s$) are produced from the "Is Valid" (C) actor. This actor also produces r tokens on the edge that connects it to "MH

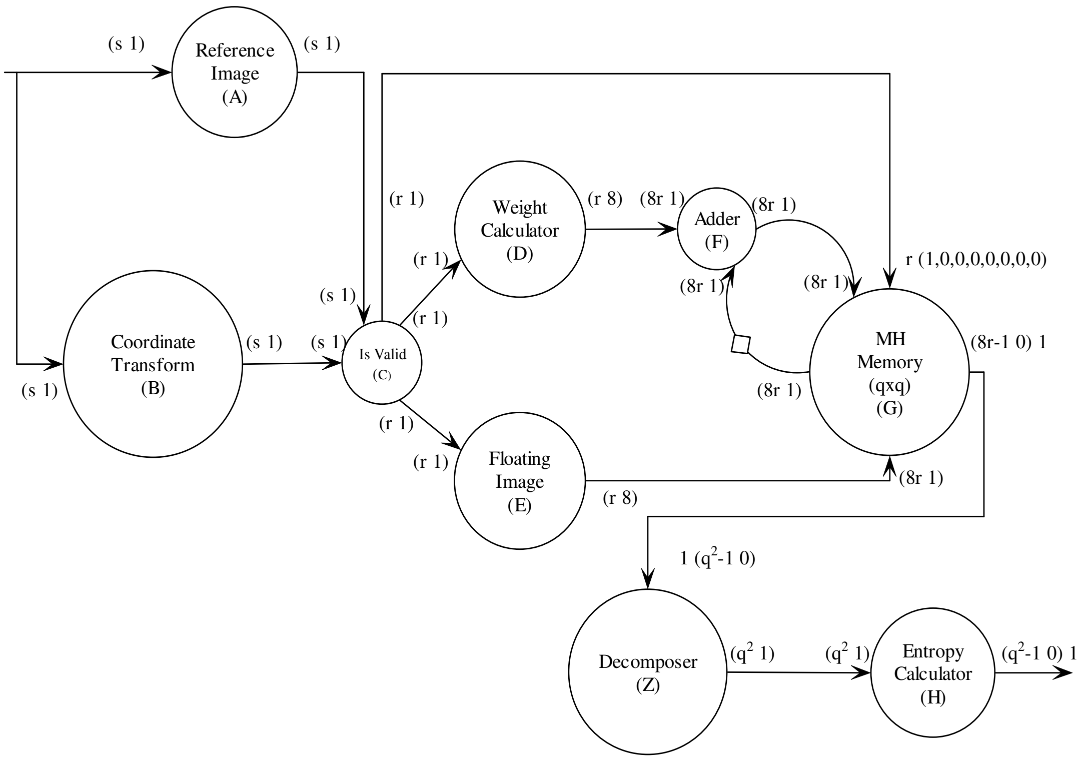


Figure 6.3 Dataflow model of Mutual Information subsystem.

Memory" (G) — specifically, it passes a token from "Reference Image" only if a valid voxel results from the transformation on input coordinates. For every input token in D and E, eight output tokens are produced on both the outgoing edges. The corresponding eight intensity locations in the "MH Memory" are updated based on the tokens produced by D.

After all coordinates are processed, which occurs during the the first $8r$ *phases* of the MH Memory actor or equivalently after s phases of the "Coordinate Transform" actor, one token of size $q \times q$ is sent to the "Decomposer", which in turn sends out $q \times q$ tokens to the "Entropy Calculator" (H) actor. H consumes all of these tokens, and produces a single token that contains the entropy value corresponding to the transformation applied based on equations given in (6.1). We added the "Decomposer" mainly for ease of representation of the application in dataflow and it was subsumed by "MH Memory" in the final hardware implementation. A valid schedule (ordering of execution) for the Mutual Information subsystem based on Figure 6.3 would be $(sABC)(rDE(8FG))(q^2ZH)$.

In this chapter, we describe our schedules as *looped schedules* which is a compact form of representing the execution order of actors and any generic looped schedule of the form $(nT_1T_2\ldots T_m)$ represents n successive repetitions of execution sequence $T_1T_2\ldots T_m$ where each T_i is either an actor or another looped schedule (to express nested looped schedules).

Looking more closely at "Coordinate Transform", we see that it has an additional input edge to it which takes in the initial m tokens from the "Optimizer" (β) but produces no output. Figure 6.3 only represents the steady-state behavior of Mutual Information subsystem for simplicity. Figure 6.4 represents the initialization and the steady-state behavior of Coordinate Transform - where the initial m tokens are used to calculate the new transformation matrix and hence it updates the values inside the actor without producing any data. Hence the schedule of the whole "Mutual Information" subsystem considering the initial and steady-state behavior of "Coordinate Transform" would be:

$$(mB)(sABC)(rDE(8FG))(q^2ZH). \tag{6.3}$$

Figure 6.5 shows the parameterized dataflow model of the "Entropy Calculator". "Row Sum" (I) executes once every time it gets one row (q elements) to produce one

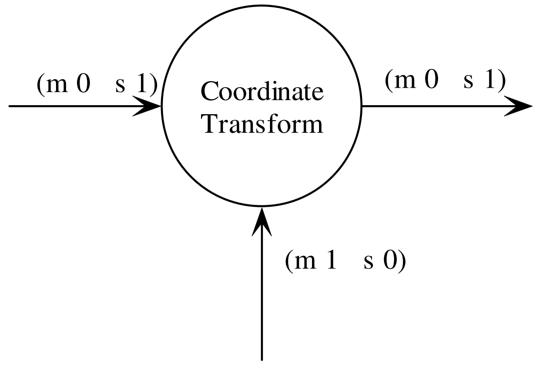


Figure 6.4 Initial and steady-state modeling of Coordinate Transform.

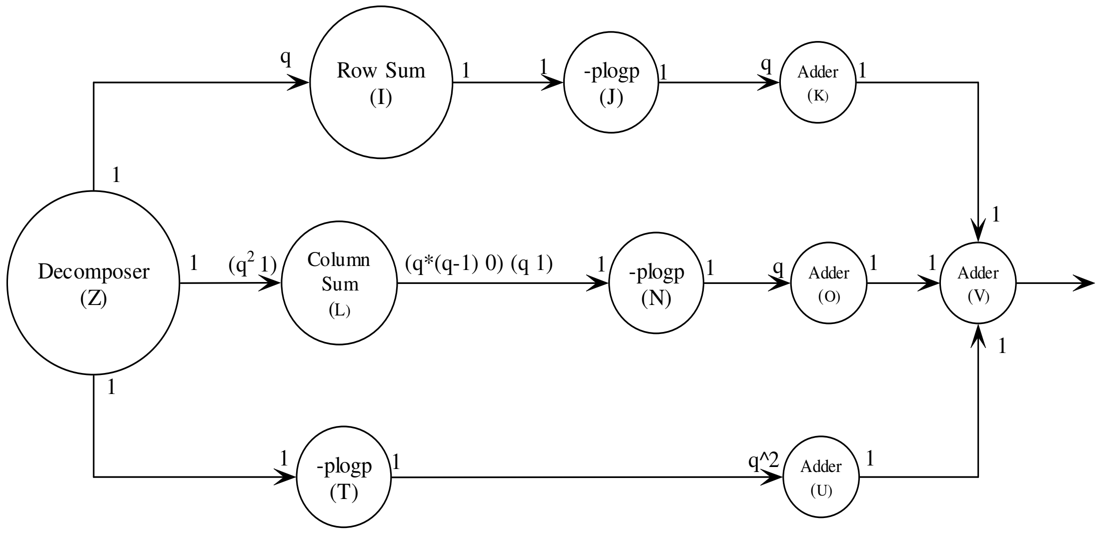


Figure 6.5 Parameterized "Entropy Calculator" where q depends on the number of bits used to represent each voxel in the input image, q is 64 for 8 bit images.

token but the “Column Sum” (L) can only produce an output for every input after it had already received $q \times (q-1)$ elements corresponding to $(q-1)$ rows. There are many valid schedules that can be proposed for Figure 6.5, but here we will try to derive any one valid schedule. Since a valid schedule for “Entropy Calculator” is quite complex, we derive it step-by-step - the graph has three distinct paths, the upper path (involving Z, I, J, K) would have a schedule $(q(qZ)IJ)K$, the middle path (involving Z, L, N, O) would have a schedule $(q(q-1)ZL)(qZLN)O$, and the lower part of the graph (involving Z, T, U) would have a schedule $(q^2ZT)U$. Combining these, a valid schedule for the “Entropy Calculator” subsystem can be:

$$(q-1(qZLT)IJ)(qZTLN)IJKOUV\,. \tag{6.4}$$

Modeling of “Entropy Calculator” exposes huge buffer overhead.

Combining (6.3) and (6.4), the schedule for the “Mutual Information” subsystem would be:

$$(mB)(sABC)(rDE(8FG))(q-1(qZLT)IJ)(qZTLN)IJKOUV \tag{6.5}$$

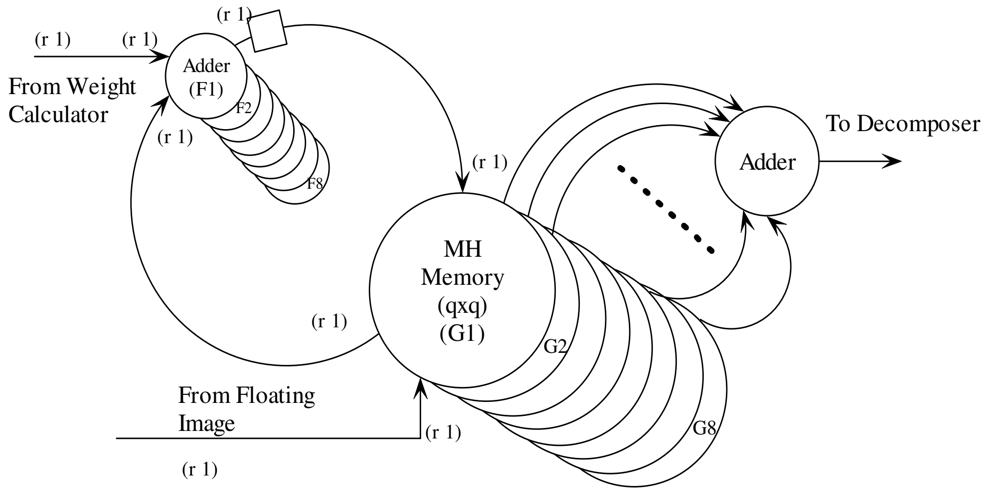


Figure 6.6 Parallel architecture for MH update exposing intra-voxel parallelism.

and taking (6.2) also into account, the schedule for the whole image registration algorithm can be derived by replacing α with (6.5).

Looped schedule is ideal for software code generation from a dataflow graph as every execution in the schedule can be replaced by a function call (or inline code) and corresponding loop index can be upper bounded by the iteration count (m, s, r, q in our case) to generate the executable code for the application [4].

Interestingly, the model shows potential for parallel hardware mapping at various levels of abstraction. For example, extensive “intra-voxel” (within the processing structure for a single voxel) parallelism is possible for the MH memory and adder. From Figure 6.3, we can see a data-rate mismatch between “Weight Calculator”, “Floating Image” and “Adder”, “MH Memory”. This naturally suggests a intra-voxel parallel architecture as shown in Figure 6.6 where multiple copies (eight in the illustration as the data-rates mismatched by a factor of eight) of an actor can be created for a parallel implementation. We also note, that resultant graph in Figure 6.6 becomes HPDF as all the parameterized actors now have the same production and consumption rates and hence fire at the same rate. The dataflow model also exposes inter-voxel parallelism, (as input actors A and B have s distinct phases where s is the number of voxels in the image) which leads to another set of useful parallel implementation considerations. We also develop an architecture in this which applies both intra- and inter-voxel parallelism, and balances these forms of parallelism adaptively based on input characteristics.

6.5. Actor Implementation

The lowest level (non-hierarchical) actors in our dataflow-based design are implemented in Verilog. As an illustration of Verilog-based actor in our design, Figure 6.7 shows the code corresponding to the Adder actor (F in Figure 6.3). An interesting point to note in this code example is that by analyzing the dataflow behavior, we can ensure that the interface code between the adder and the weight calculator places the correct weight at every clock cycle in the input buffer labeled 'weight'. This illustrates how using dataflow as a high-level modeling abstraction helps to structure the hardware implementation pro-

```
/* global definitions in top.v */
reg [imsize+fracwidth-1:0] mh [0:4096];
reg [imsize+fracwidth-1:0] edgeweights [0:numweights-1];

/*one example module */
module mhupdate
#(parameter imsize = 8,
parameter fracwidth = 8,
parameter numweights = 8,
parameter lognumweights = 3)
(input [imsize-1:0] rival,fival,
input [imsize+fracwidth-1:0] weight,
input clk);
reg [11:0] currval;
reg [lognumweights:0]counter;

always @(posedge resetall)
  counter <= 0;

always @(posedge clk)
  begin
    if(counter < numweights) begin
      mh[currval] <= mh[currval] + weight;
      currval[5:0] <= rival[imsize-1:imsize-6];
      currval[11:6] <= fival[imsize-1:imsize-6];
      counter <= counter + 1;
    end
    else
      counter <= 0;
  end
endmodule
```

Figure 6.7 Example code (partial) of the Adder from Figure 6.3

cess, and makes the hardware description language (HDL) code modular and reliable. Hence we had a one-to-one mapping in hardware from the dataflow graph except for the “Decomposer” which was subsumed inside “MH Memory” for efficient implementation.

6.6. Experimental Setup

We explored in detail the effect of having a parallel architecture on the application as suggested by the dataflow model. In our experimental setup, we varied the degree of parallelism and studied the relation between performance and area of the system. We also noted that the percentage of voxels that fall in the valid range after a transformation by the “Coordinate Transform” greatly influences the runtime of the algorithm. Hence we studied our system performance by varying percentage of valid voxels (PVV) for a given transformation.

6.6.1. Degree of Parallelism

When the “Floating Image” is provided with the base address in the floating image space, the actor generates the floating image values (corresponding to the neighborhoods) and provides it to the mutual histogram memory for updating the mutual histogram with the weights generated by the weight calculator actor. When we have just one set of actors (floating image, weight calculator and the mutual histogram memory), it takes eight firings of this set of actors (corresponding to the values of the eight neighborhood) before the next input can be processed by the coordinate transform actor. However if we have two copies of the above set of actors, then each set can process four neighborhoods each. Similarly if we have four (or eight) copies, then each set can process two (or one) neighborhood(s) each. This would mean that the number of firings of each set of actors becomes 4, *2* (or *1*) respectively. As updating the mutual histogram is a crucial part of the algorithm, such parallel execution should result in significant improvement of the whole application.

However the parallel configurations result in extra FPGA resources and extra external memory. Memory requirement also increases with increasing image size. In addition to this, there is a cost of interfacing these external memories that needs to be addressed. Each memory component comes with a latency that adds to the processing time.

6.6.2. Relationship between PVV and Performance

When the transformed coordinate falls in the valid region, there are eight firings of the actor set ("Adder", "MH Memory" in Figure 6.3). However when "Is valid" does not generate a signal (indicating the for the given input coordinates, the transformation produces coordinates outside of the valid coordinate boundary), the iteration of the graph stops for that input coordinates and the next token is processed by the coordinate transform actor indicating a new iteration. In our implementation, when an invalid voxel coordinate is generated for the first time, there is a two cycle penalty (as we have to propagate the invalid signal through "Weight Calculator" and "Adder"), however the penalty is only one clock cycle for every successive invalid signals (as now, we already have those two actors filled with the invalid signal).

We explore the performance area curve for different PVVs in Section 6.7.

6.7. Results

In this section, we present hardware synthesis results for various proposed configurations of the Image Registration application. The results are obtained using the QuartusII synthesis tool from Altera for the StratixII family of FPGA (StratixII EP2S15F484C5).

Table 6.1. presents the synthesis results for various configurations - the columns represent the number of parallel datapaths for MH Update actor and rows represent in order - external memory required for the system, logic circuitry used in the FPGA for the MI actor, DSP elements for the circuit for the MI actor, total number of ALUTs used in the FPGA for MI, and maximum frequency of operation of the circuit representing MI. We note that external memory increases with increasing data-paths due to multiple copies of "MH Memory".

Number of parallel datapaths	**1**	**2**	**4**	**8**
External Memory	256Kb	512Kb	1Mb	2Mb
LC Registers in FPGA	427	576	871	1463
DSP Elements	30	30	30	30
Total FPGA Area (number of ALUTs)	598	878	1439	2588
Max freq of operation (MHz)	74	72.2	74	70.1

Table 6.1. Synthesis results for the whole system for different configurations of the MH Update actor.

An adaptive logic module (ALM) is the basic building block of Altera StratixII FPGA. Each ALM contains a variety of LUT-based (look-up table) resources, two full adders, carry-chain segements, two flipflops and can be adaptively divided into two adaptive LUTs (ALUTs). Logic Cell (LC) registers represent the total number of registers used and ALUTs used represent the percentage of the available resources in the FPGA that is used. From Table 6.1. we can see that both of them increase as the number of data-paths are increase. However, the number of DSP elements used and the frequency of operation almost remains constant. Table 6.1. is independent of PVV as PVV only affects the runtime of the circuit.

Next, we simulated the performance of the various configurations of the circuit with four different PVVs as 100, 90, 50 and 10 in terms of number of clock cycles. We assumed that when PVV is low, invalid signals are contiguous and they are sparse when PVV is high. This has a bearing on the performance as mentioned in Section 6.6.2. Figure 6.8 shows the area (measured by the number of adaptive logic units in the circuit without considering the external memory) and performance (measured by the number of execution cycles) trade-off curve as we vary the number of parallel datapaths in the MH update actor

Number of parallel datapaths	**1**	**2**	**4**	**8**
Power for Logic (mW)	4	15	25	35
Dynamic power for FPGA (mW)	92	115	147	159

Table 6.2. Comparison of power consumption of circuit in different datapath configurations

and the PVV. ,The trend in Figure 6.8 reflects that number of clock cycles decrease with increasing amount of parallel data paths though the corresponding area increases but the relative change in number of clock cycles by increasing data-paths is also dependent on the PVV. Extending on this, a PVV-based dynamically-reconfigurable FPGA implementation is proposed in Section 6.7. For a more complete overview of the different configurations, we also present the full system area estimation with consideration for external memory in Figure 6.9.

We also measured the dynamic power of the FPGA for both the logic part and the full circuit including RAM I/O power, DSP blocks, and clocks (without considering external memory as that would depend on the physical board on which the application is finally implemented) as shown in Table 6.2. . As expected, we see an increase in the power consumption as the number of parallel data-paths increase.

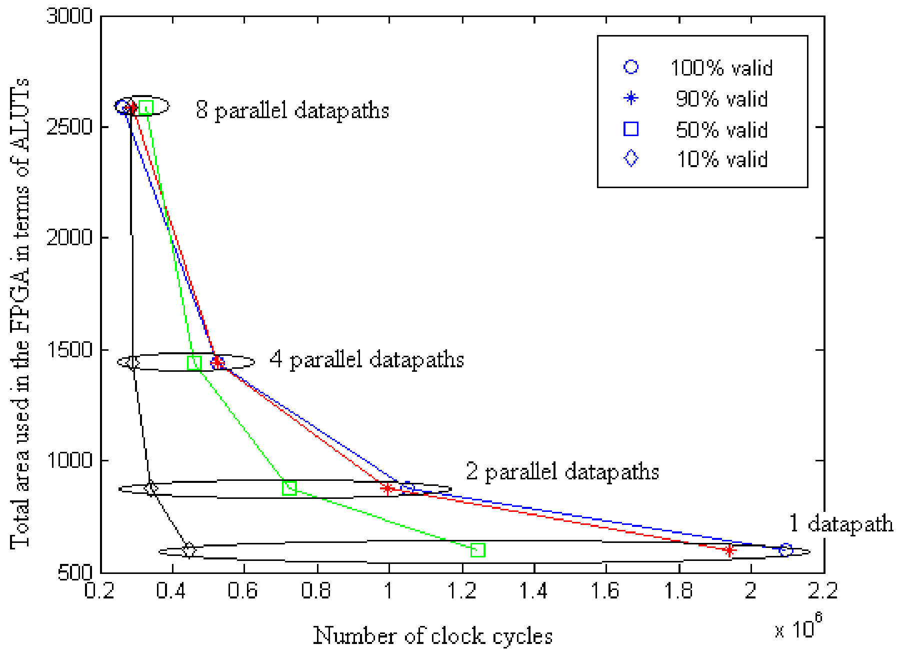


Figure 6.8 Area v/s clock cycles for different PVV for different number of datapaths.

6.8. Dynamic Reconfiguration

In this section, we compare multiple one voxel-one memory architecture against one voxel-eight memory architecure and elaborate on the dynamic reconfigurability of our proposed architecture and present results of our design. In other words, we compare inter-voxel parallelism against intra-voxel parallelism both of which were exposed by our data-flow-based design (Section 6.4).

Based on Section 6.7, we see that the PVV is input-dependent and as the PVV increases, the run-time increases and memory access becomes more of a bottleneck, and gradually, it becomes more performance-effective to trade-off inter-voxel parallelism in the architecture for intra-voxel parallelism in the form of multiple (parallel) memories that alleviate the memory bottleneck. This trend is demonstrated by the data in Table 6.3,

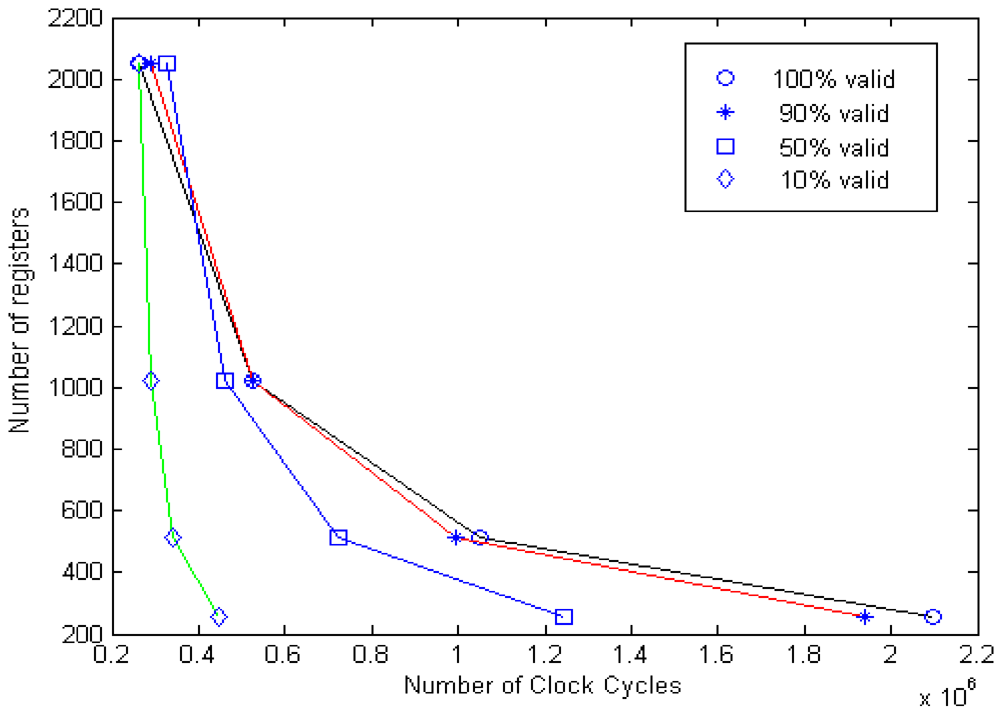


Figure 6.9 Whole system memory requirements in bytes including external memory.

which compares the performance, for different PVV values, of a 1 voxel-8 memory architecture (intra-voxel parallelism) to a 7 voxel architecture with 1 memory module per voxel (inter-voxel parallelism) architecture. The value of 7 is selected here because for the targeted FPGA device, the area of a 1 voxel-8 memory architecture is around 7 times that of a 1 voxel-1 memory architecture. The units of performance in Table 6.3 are nanoseconds per voxel per co-ordinate transform and the frequencies of operation of the different memory architectures vary between 70 MHz and 74 MHz for various configurations.

We note in Table 6.3, considering the area constraint, performance of 1 voxel-1 memory architecture is better than that of a 1 voxel-8 memory architecture, however this trend changes as the voxel validity percentage increases. Therefore, our image registration architecture monitors the PVV metric at run-time and dynamically reconfigures the architecture from inter-voxel parallelism mode to intra-voxel parallelism mode once the transition point of around 50% PVV is observed. In order to prevent rapid toggling between the two architectures (also known as thrashing) when PVV is close to 50%, users can select a threshold T so that architecture gets reconfigured when a $(50 - T)\%$ PVV state is fol-

Voxel Validity	Performance of 7 1 voxel-1 memory	Performance of 1 1voxel-8 memory
10%	6.39/7 = 0.91	2.54
50%	17.8/7= 2.54	2.91
90%	27.82/7 = 3.97	2.5
100%	30.08/7 = 4.29	2.33

Table 6.3. Comparison of intra- versus inter-voxel parallelism modes for different PVV values.

lowed by a ($50 + T$) % PVV state or vice versa. This can be viewed as a periodic (once per image), PVV-driven re-scaling of the subsystem shown in Figure 6.6. So in effect, our proposed architecture monitors the PVV at run-time, and dynamically reconfigures between inter-pixel and intra-pixel parallel architectures when the PVV crosses 50%.

Note that the optimal transition point is in general image-dependent, and our use of a fixed value of 50% as a transition point is therefore a heuristic approach. Dynamically determining the transition point is a useful topic for further investigation.

6.9. Conclusion

In this chapter, we have presented a dataflow-based design approach towards implementation of an image registration algorithm onto an FPGA. We have captured the inherent concurrency of the application at inter- and intra-voxel level by modeling it through the framework of homogeneous parameterized dataflow. We have also presented some dataflow motivated parallel architectures for image registration and presented a detailed study of area performance trade-off for these multiple architectures. Based on the results obtained, we have also presented the derivation and FPGA mapping of an architecture for dynamically-reconfigurable image registration. We have demonstrated the ability of the architecture to strategically adapt its parallel processing configuration in response to relevant image characteristics, and for this purpose we have formulated the PVV metric, which represents the percentage of valid voxels that results from a transformation on the given floating image.

Chapter 7. Intermediate Representations for MATLAB Synthesis

Specifying signal processing applications in terms of dataflow graphs [31] or process networks [33] is a common practise. This exposes inherent concurrency in the applications which otherwise is an extremely hard problem to solve starting from a sequential program. Synchronous dataflow (SDF) graphs, Cyclostatic dataflow (CSDF) graphs and Kahn process networks [27] (KPN) are natural choices for modeling static applications. Previous work showed that it is possible to generate a KPN from a sequential affine nested-loop program [28]. SDF and CSDF graphs can be analyzed for correctness, finite buffer sizes, and scheduling which cannot be done for a KPN graph. KPN graphs do not have a global schedule, and they synchronize by blocking reads. Due to the inherent unblocking writes that are part of the KPN specification, every edge on a KPN graph can potentially have an infinite buffer. In our recent work, [14], we showed that a KPN graph, which is input-output equivalent to a static affine nested loop program (and hence can be analyzed by a toolflow named Compaan as we describe later) is a special case of a CSDF graph whose production and consumption rates in each phase is either a *0* or a *1*. In this chapter, we give a brief description of the work presented in [14] and show our proposed extensions to a dataflow specification language (DIF) to capture the equivalent CSDF arising out of a KPN.

We are very greatful to Prof. Ed Deprettere and Dr. Todor Stefanov for their time and effort in collaborating with us for this work.

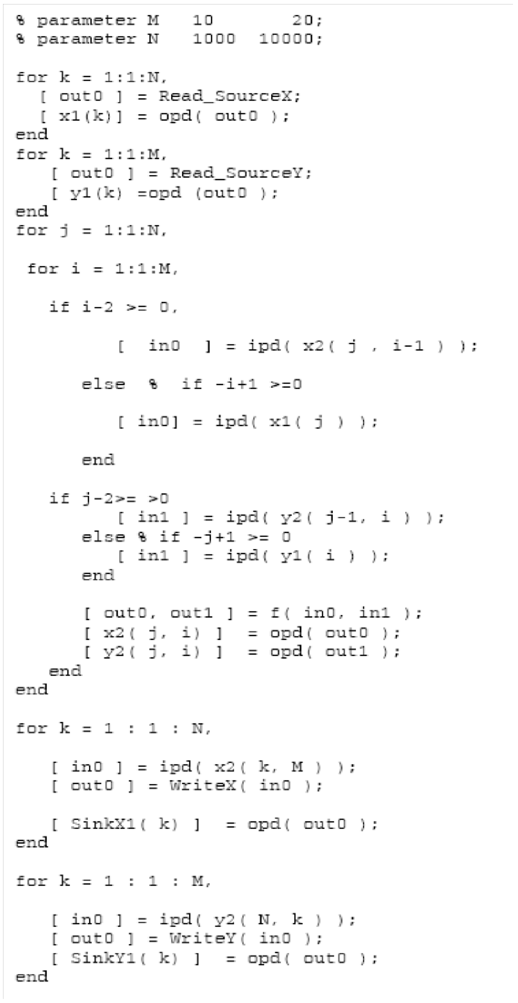
7.1. Introduction to Compaan

The behavior of signal processing applications is very often specified in terms of affine nested loop programs. An affine nested loop program is a nested loop program in which the loop boundaries, the conditions, and the variable indexing functions are affine functions of the loop iterators. An example of such a program is shown in Figure 7.1. Such programs can be automatically converted to input-output equivalent concurrent specifications where the underlying model of computation is Kahn Process Network (KPN) [12]. A Kahn process network is a network of processes that process communicate point-to-point over unbounded unidirectional FIFO-type buffered channels, and synchronize by means of blocking reads. A Kahn process network has neither a global memory nor a global

```
%parameter M 10 20;
%parameter N 1000 10000;

for k = 1:1:N,
   [x(k)] = Read_SourceX();
end

for k = 1:1:M,
   [y(k)] = Read_SourceY();
end

for j = 1:1:N,
   for i = 1:1:M,
      [x(j), y(i)] = f(x(j), y(i));
   end
end

for k = 1:1:N,
   [SinkX(k)] = WriteX(x(k));
end

for k = 1:1:M,
   [SinkY(k)] = WriteY(y(k));
end
```

Figure 7.1 An affine-nested loop program

scheduling policy [27]. Of course, KPNs that are derived from affine nested loop programs are special, and we call this special subclass Compaan process networks (CPN) because the first reported affine nested loop program to KPN translator was called Compaan translator [28]. The convertion of an affine nested loop program to a CPN goes in three steps. The first step is to derive a Single Assignment Program (SAP) version of the given affine nested loop program [30].The SAP for the affine nested loop program in Figure 7.1 is shown in Figure 7.2. In this program, the functions *ipd()* and *opd()* are the iden-



```
% parameter M   10       20;
% parameter N   1000  10000;

for k = 1:1:N,
  [ out0 ] = Read_SourceX;
  [ x1(k)] = opd( out0 );
end
for k = 1:1:M,
   [ out0 ] = Read_SourceY;
   [ y1(k) =opd (out0 );
end
for j = 1:1:N,

 for i = 1:1:M,

   if i-2 >= 0,

         [  in0  ] = ipd( x2( j , i-1 ) );

      else  %  if -i+1 >=0

         [ in0] = ipd( x1( j ) );

      end

   if j-2>= >0
         [ in1 ] = ipd( y2( j-1, i ) );
      else % if -j+1 >= 0
         [ in1 ] = ipd( y1( i ) );
      end

      [ out0, out1 ] = f( in0, in1 );
      [ x2( j, i) ]  = opd( out0 );
      [ y2( j, i) ]  = opd( out1 );
   end
end

for k = 1 : 1 : N,

   [ in0 ] = ipd( x2( k, M ) );
   [ out0 ] = WriteX( in0 );

   [ SinkX1( k) ]  = opd( out0 );
end

for k = 1 : 1 : M,

   [ in0 ] = ipd( y2( N, k ) );
   [ out0 ] = WriteY( in0 );
   [ SinkY1( k) ]  = opd( out0 );
end
```

Figure 7.2 The Single Assignment Program version of the Affine Nested Loop Program in Figure 7.1

tity function to bind input variables to arguments of the function *f()* and the results of the function *f()* respectively.

The second step is to convert the SAP to a Polyhedral Reduced Dependence Graph (PRDG) data structure which is a compact mathematical representation of the dependence graph counterpart of the SAP in terms of polyhedra and lattices [15][43]. In short, it is a graph $G = (\aleph, \xi)$ where nodes $N \in \aleph$, and edges $E \in \xi$ between output ports and input ports of Nodes. Please refer to [15][43] for a more detailed understanding. The equivalent PRDG is shown in Figure 7.3.

The third step is to derive the Compaan Process Network (CPN) from the PRDG which includes code generation for the processes, and linearization of the higher dimensional variable arrays [53]. Details of the process is in [53]. The topology of the derived CPN is the same as the topology of the originating PRDG, as shown in Figure 7.3. The code for Node ND_3 is shown in Figure 7.4. The nodes such generated from Compaan can also be represented using another model named as *Stream Based Function* (SBF) model [29]. The SBF model is a *fire-and-exit* model of computation that can be seen as a virtual

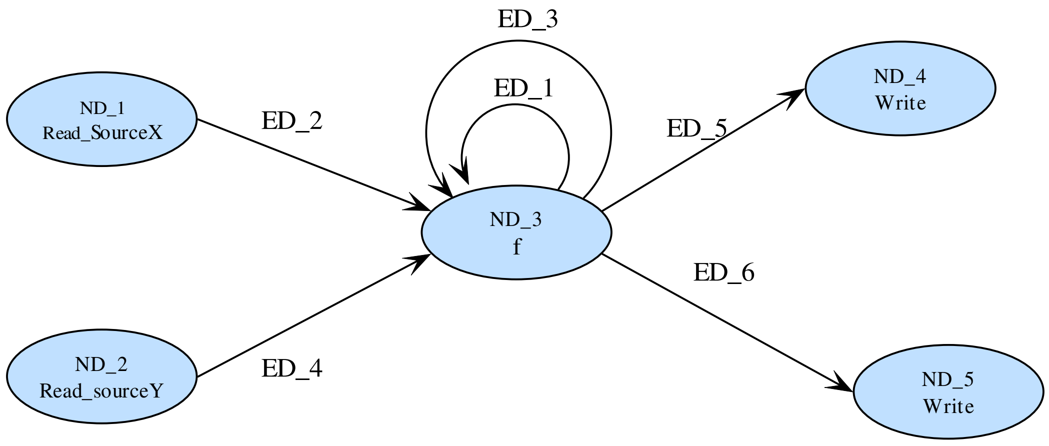


Figure 7.3 The PRDG corresponding to the SAP in Figure 7.2

procesor (VP) for subsequent implementations. The general structure of a SBF virtual processor is shown in Figure 7.5.

7.2. Relation between SBF and CSDF

In this section, we give a brief description of the relation between an SBF actor derived from a Compaan process network and a CSDF actor. The full extent of this work is presented in [14]. Though we were a part of this research, but the majority of this work was done in Leiden University, The Netherlands in collaboration with us. However, presentation of this part is essential towards understanding the next few sections of this chapter.

```
for j = 1:1:N,
  for i = 1:1:M,
    if i = 1,
      [in0] = Read(ED_2, token(r_2(j,i));
    else
      [in0] = Read(ED_1, token(r_1(j,i));
    end
    if j = 1,
      [in1] = Read(ED_4, token(r_4(j,i));
    else
      [in1] = Read(ED_3, token(r_3(j,i));
    end
    [out0, out1] = f(in0, in1);
    if i = M,
      [token(w_5(j,i))] = Write(ED_5, out0);
    else
      [token(w_1(j,i))] = Write(ED_1, out0);
    end
    if j= N,
      [token(w_6(j,i))] = Write(ED_6, out1);
    else
      [token(w_3(j,i))] = Write(ED_3, out1);
    end
  end
end
```

Figure 7.4 Example of the generated code for a Node. This node used here is *ND_3* in Figure 7.3

We show that SBF VPs which are nodes in the CPN can be converted to CSDF actors, and thus a CPN can be coverted to a CSDF graph.

Recall the definition of CSDF graph as given in Section 2.1.2, that actors have multiple phases. The operation of a CSDF actor with n edges (both input and output combined) is characterized by a function $F(z_0, z_1, \ldots, z_{n-1}, z_n)$ where z_0 is the phase argument, $z_1, \ldots, z_n$ are input and output arguments. Thus for the CSDF actor in Figure 7.6

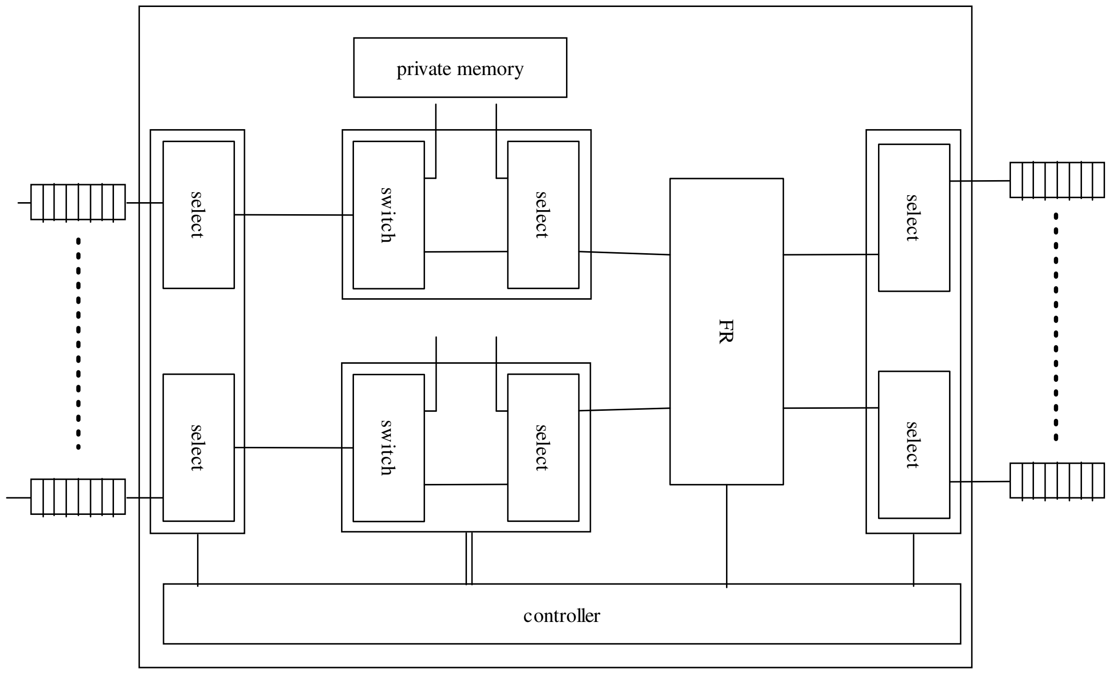


Figure 7.5 Strucure of a SBF Virtual Processor

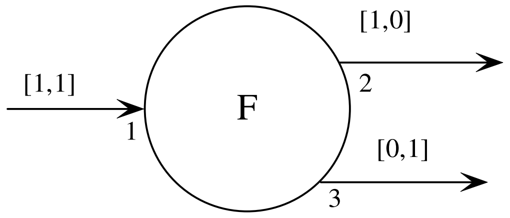


Figure 7.6 An example CSDF actor

$$\text{if } z_0 = 1\text{, then } [z_2, \perp] = f(z_1) \text{ and if } z_0 = 2\text{, then } [\perp, z_3] = f(z_1)\text{,} \qquad (7.1)$$

where $\perp$ represents a null token (or no token).

The core of a VP is a re-usable IP implementation of abstract symbolic functions in the underlying sequential program, such as *f()* in the program of Figure 7.1. The input and output arguments to and from this IP core are taken from and sent to input and output channels, respectively, possibly via the private memory, in case the consumption of tokens is in a different order than they are read from the channels (Figure 7.5). The controller selects the appropriate input and output channels, as well as the current function in the IP function repertoire, in case the IP core implements more than one function. Hence the input and output behavior of a VP can be expressed as phases (albeit possible long phases) corresponding to the order in which data is consumed and produced, and the functionality *f()* (or a combination of *f()'s* if there is a repertoir) can be the core functionality of the CSDF actor. In Figure 7.7, we show the virtual processor for *f()* as described in Figure 7.1 with explicit mention of read and write sequences on the input and output edges. We first derive the steady state behavior of the virtual processor and then show the resultant CSDF actor in Figure 7.8. We also notice that the tokens are read and written one at a time as in Process Networks, tokens are always read in multiples of one. As a result, the resultant CSDF from a SBF VP will always have a phase signature with each phase having a value of either a 0 or a 1. We have classified such CSDFs as *Binary Cyclo-static Dataflow* graphs or BCSDF. As described by (7.1), we can analogously characterize the CSDF actor F in Figure 7.8 by a function $F(X_0, X_1, X_2, X_3, X_4, X_5, X_6)$ in which X_0 is the phase argument with the help of Table 7. 1. FR in the table represents the Function Repertoire which in this case contains only one function *f()*. Figure 7.9 shows the SBF VP behavior that can be constructed back from Table 7. 1. We note that the generic expression — $[v, w] = f(x, y)$ can be used to represent the function *f()* in Figure 7.9 where each of v, w, x, y represent *4 groups:* $v = \{X_4, X_6\}$, $w = \{X_5\}$, $x = \{X_1, X_2\}$, $y = \{X_3\}$. The groups tell us to what exclusive channels the variables of the function can bind.

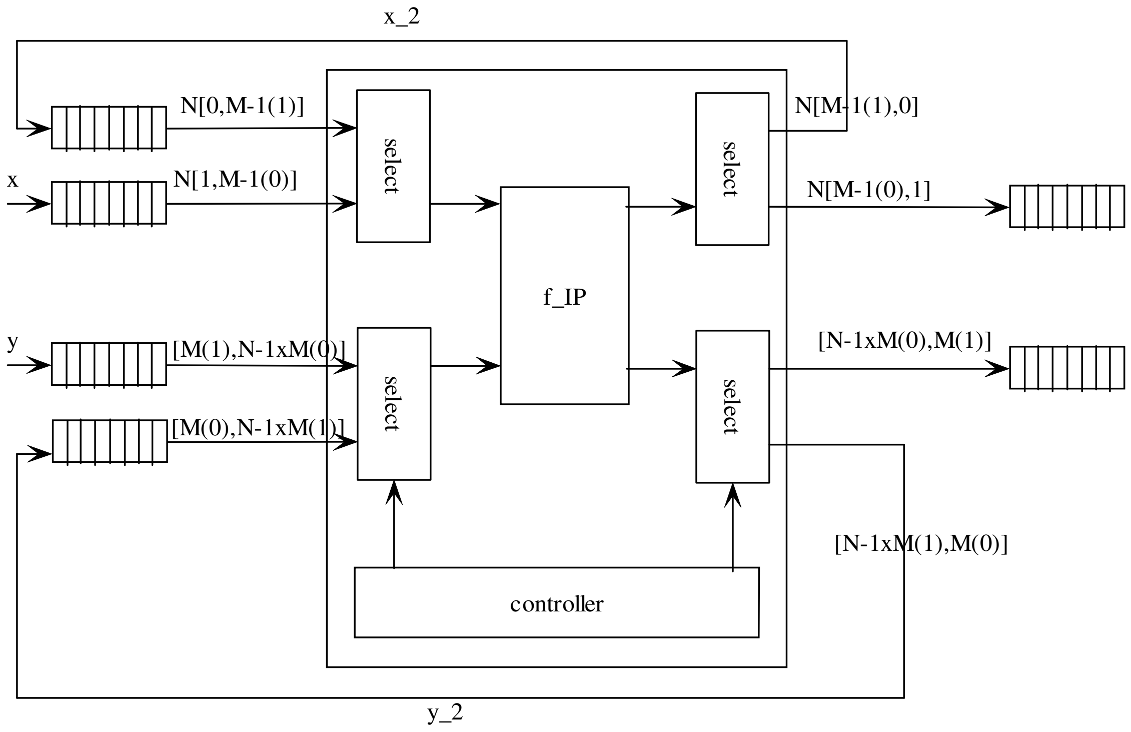


Figure 7.7 SBF virtual *f()* processor for the prgram in Figure 7.1 with explicit mention of read and write sequences.

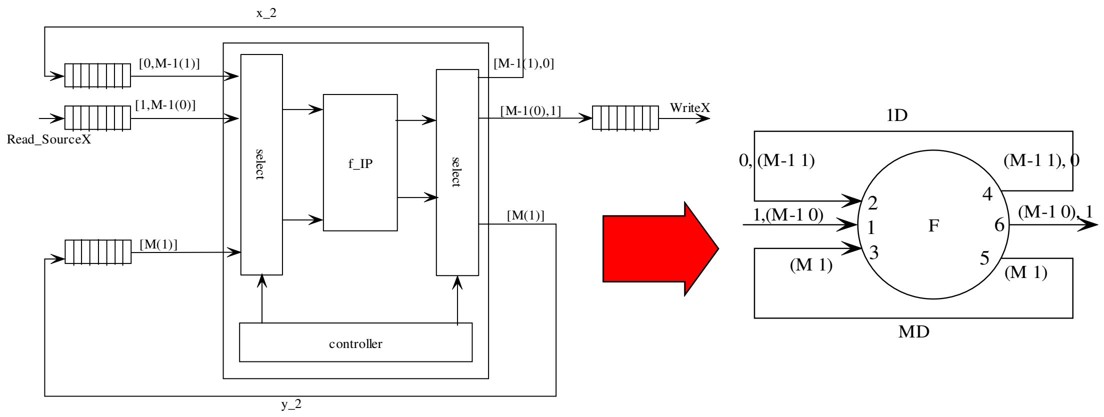


Figure 7.8 The steady-state equivalent virtual *f()* processor of Figure 7.7 on the left and its corresponding CSDF actor on the right.

7.3. CSDF to SBF

The conversion of a binary CSDF actor to a SBF virtual processor requires that the CSDF phase signatures and the function $F(X_0, X_1, \ldots, X_n)$ be converted to the selection of channels in the SBF virtual processor and the binding of the channel variables to the arguments of the functions in the function repertoire. We show the methodology for such a conversion through an example. For the more general methodology, please refer to [14]. The first step is to set up a table from a CSDF actor as we did in Table 7. 1 and extract the SBF VP specification from it. Thus let us consider the CSDF actor in Figure 7.10, and let

x_0	x_1	x_2	x_3	x_4	x_5	x_6	FR
1	1	0	1	1	1	0	f
2-(M-1)	0	1	1	1	1	0	f
M	0	1	1	0	1	1	f

Table 7. 1. Table represnting the function $F(X_0, X_1, X_2, X_3, X_4, X_5, X_6)$

```
if X_0 = 1,
   [X_4, X_5] = f(X_1, X_3);
else if X_0 = 2,3,...,M-1,
   [X_4, X_5] = f(X_2, X_3);
else if X_0 = M,
   [X_6, X_5] = f(X_2, X_3);
```

Figure 7.9 Construction of SBF VP back from Table 7. 1.

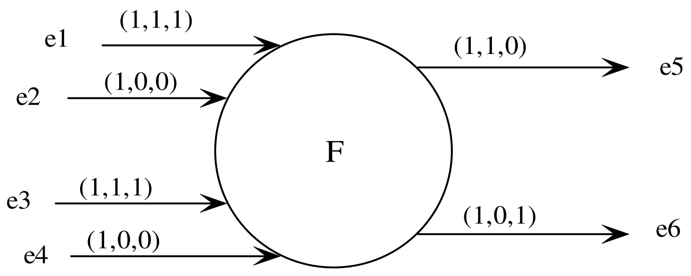


Figure 7.10 An example binary CSDF.

the actor F be characterized by a function $F(X_0, X_1, X_2, X_3, X_4, X_5, X_6)$ (as there are 4 input edges — e_1 through e_4, and 2 output edges — e_5 and e_6 with X_0 representing the phase variable) as in Figure 7.11. Now the corresponding table built from Figure 7.10 and Figure 7.11 would be as presented in Table 7. 2. For the first row, the functions *max* and *min* are $[X_5] = max(X_1, X_2)$ and $[X_6] = min(X_3, X_4)$, respectively. For the second row, we have $[X_5] = max(X_1, X_3)$ and for the last row, we have $[X_6] = min(X_1, X_3)$. We can also give for each phase signature the corresponding set of functions, one for each phase signature. Thus $e_1 \rightarrow \{max, max, min\}$, $e_2 \rightarrow \{max, \perp, \perp\}$, $e_3 \rightarrow \{min, max, min\}$, $e_4 \rightarrow \{min, \perp, \perp\}$, $e_5 \rightarrow \{max, max, \perp\}$, $e_2 \rightarrow \{min, \perp, max\}$. Finally, denoting by $[v_M] = max(x_M, y_M)$ and $[v_m] = min(x_m, y_m)$ the two core functions, the groups (selectors and distributors) will be as follows: $x_M = \{e_1\}$, $y_M = \{e_2, e_3\}$, $x_m = \{e_1, e_3\}$, $y_m = \{e_3, e_4\}$, $v_M = \{e_5\}$, and $v_m = \{e_6\}$. The equivalent SBF is presented in Figure 7.12 .

```
if X_0 = 1
   X_5 = max(X_1, X_2);
   X_6 = min(X_3, X_4);
else if X_0 = 2
   X_5 = max(X_1, X_3);
else if X_0 = 3
   X_6 = min(X_1, X_3);
```

Figure 7.11 Computational behavior of actor F in Figure 7.10 .

X_0	e_1	e_2	e_3	e_4	e_5	e_6	**FR**
1	1	1	1	1	1	1	{max,min}
2	1	0	1	0	1	0	max
3	1	0	1	0	0	1	min

Table 7. 2. Table representing the BCSDF actor in Figure 7.10.

A difference between a SBF process network and a CSDF graph is that the former has neither initial tokens nor termination tokens, while the latter does have initial and final tokens because it models the steady-state behavior of an algorithm. The SBF process network works on finite streams, and although the phase signatures seems to be very long and not repetitive, there are still core phase signatures that is periodically repeated for a finite number of periods. That number appears explicitly in the phase signature expressions and is merely an indication of the finiteness of the number of periods.

7.4. Introduction to DIF

The Dataflow Interchange Format (DIF) is a standard language to specify dataflow models for stream-oriented applications such as DSP, Image and video processing etc. DIF is built with the portability issue in mind. As in the dataflow domain, there is a lack of a standard vendor-independent language. DIF has helped in providing an expandable repos-

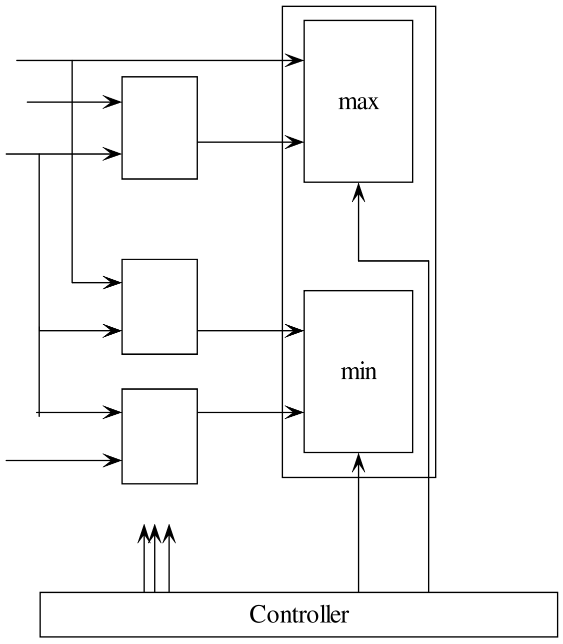


Figure 7.12 SBF VP for the actor in Figure 7.10

itory of dataflow models and techniques. Usually dataflow analysis and scheduling techniques require production rates, consumption rates, edge delays and various node and edge weight information. So detailed node characteristics is not important in many dataflow-based analyses. As a result, the initial version of DIF did not include actor-specific information as a part of the language specification. However, the computation and some other actor attributes are essential for implementation. In the later version of DIF (DIF 0.2), actor specific information was added to preserve an actor's functionality while importing and exporting between DIF and various design tools and also across design tools.

In this work, we used the extended functionality provided by DIF 0.2 to represent the dataflow graph as shown in Section 7.5.

7.5. Proposed extensions to DIF

We generate a complete CSDF graph from Figure 7.10 in Figure 7.13 for the purpose of illustration of the proposed extensions we made to DIF so that the CSDF generated from a SBF could be accurately captured.

We provide a brief explanation of Figure 7.14, for a more detailed explanation of the DIF language, please refer to [25]. The keyword *csdf* is used to describe the type of the graph *G1*. Next we describe the topology of the graph by the set of nodes and edges in graph *G1*. Each edge has a production and consumption rate, which for a CSDF can be an array of integers. Next the actor *F* is described in which the computation inside the actor is mentioned explicitly and the binding of edges with the variables for the actual function call (described by the computation) is done. Attribute is a keyword in DIF which is used to describe user defined attributes. An attribute *coreInputs* is used to express the grouping among the input edges. A blank left at the left hand side of the assignment operator means that the attribute described is an attribute of the csdf

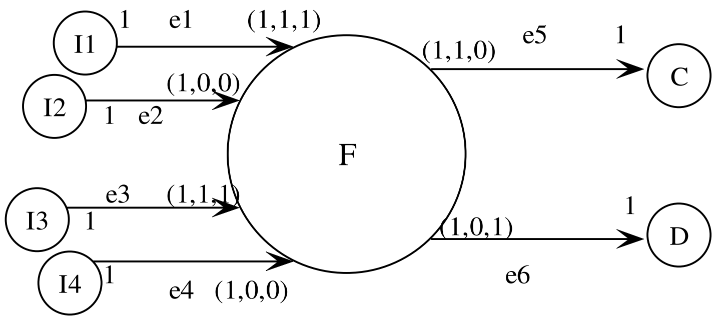


Figure 7.13 CSDF graph made from Figure 7.10 for the purpose of explaining the DIF generated from it.

```
csdf G1 {
  topology {
    nodes = I1, I2, I3, I4, F, C, D;
     edges = e1(I1, F), e2(I2, F), e3(I3, F), e4(I4, F), e5(F, C),
e6(F, D);
   }
  production {
    e1 = 1;
    e2 = 1;
    e3 = 1;
    e4 = 1;
    e5 = [1, 1, 0];
    e6 = [1, 0, 1];
   }
  consumption {
    e1 = [1, 1, 1];
    e2 = [1, 0, 0];
    e3 = [1, 1, 1];
    e4 = [1, 0, 0];
    e5 = 1;
    e6 = 1;
   }
  actor F {
    computation = "MaxMin";
    u = e1;
    v = e2;
    w = e3;
    x = e4;
    y = e5;
    z = e6;
   }
  attribute coreInputs {
    = "group1 = {e1}";
    = "group2 = {e2, e3}";
    = "group3 = {e3, e1}";
    = "group4 = {e4, e3}";
   }
  attribute coreOutputs {
    = "group5 = {e5}";
    = "group6 = {e6}";
   }
  attribute coreFunctions {
    = "maximum (2, 1)";
    = "minimum (2, 1)";
   }
```

```
     attribute coreSequences \{
           =  "(group5)  =  maximum(group1,  group2),  (group6)  =  mini-
    mum(group3, group4)";
        = "(group5) = maximum(group1, group2)";
        = "(group6) = minimum(group3, group4)";
      \}
    \}
```

Figure 7.14 DIF representation of the CSDF graph in Figure 7.13

graph - *G1*. Since this grouping information does not need to be parsed by the DIF parser, and is used only to extract relevant information for the equivalent SBF representation, it is expressed as a string. Similarly *coreOutputs* is an attribute for grouping the output edges. Attribute *coreFunctions* describe in more detail the computation for actor *F*, it describes the actual function calls made inside the computation of the actor *F* (if any) and the number of input and output arguments they have. Attribute *coreSequences* describe the function calls along with the input and output arguments (described as groups in *coreInputs* and *coreOutputs*) for each phase of the CSDF actor *F*. So in Figure 7.14, in phase 1, *F* takes in one input argument from *group1* of inputs and one input argument from *group2* of inputs, passes them to the *maximum* function and the output is produced on an edge in *group5*. In the same phase, it also takes in one input from *group3* and one from *group4* passes them onto the *minimum* function and produces an output on an edge from *group6*. However, in phase 2 and 3 of *F*, it only calls *maximum* and *minimum* as expressed in Figure 7.14. Along with the DIF specification, the functionality of *F* was expressed by a C function which is shown in Figure 7.15.

The function in Figure 7.15 describes the actual C code expressed by *F* which has the computation core expressed by *MaxMin* and the binding of the edges to variables in the function which is done in *actorF* block of Figure 7.14.

7.6. Binary CSDF

Binary CSDF is a restricted version of Cyclo-static dataflow where the production and consumption rates are constrained to be binary vectors, more specifically, the production and consumption rates can only be vectors of 0 or 1. A BCSDF is a natural outcome from a SBF as explained in Section 7.2. One of the major advantages of having a separate dataflow model for BCSDF is that the binary vectors can be very efficiently compacted as bit vectors, this is especially useful where the phases are long strings of 0's and 1's as we saw in the case of CSDFs arising out of SBFs.

```
void MaxMin (int *phase, float *u, float *v, float *w, float *x,
float *y, float *z) {
  if(*phase == 0) {
    y = maximum(u, v);
    z = minimum(w, x);
   }
  else if (*phase == 1) {
    y = maximum(u, w);
   }
  else if(*phase == 2) {
    z = minimum(u, w);
   }
}

float *maximum(float *a, float *b) {
  if(*a >= *b) return a;
  return b;
}

float *minimum(float *a, float *b) {
  if(*a <= *b) return a;
  return b;
}
```

Figure 7.15 C addition to the DIF representation of Figure 7.14 to represent the CSDF graph of Figure 7.13

Chapter 8. Conclusions and Future Work

8.1. Conclusion

In this thesis, we developed a new dataflow meta-modeling technique, called homogeneous parameterized dataflow (HPDF). HPDF is a meta-modeling technique in that it can be applied to a variety of underlying dataflow models of computation to enhance their expressive power, and we gave examples through the use of HPDF over two static dataflow models — synchronous dataflow (SDF) and cyclo-static dataflow (CSDF) which allowed us to extend SDF and CSDF to be used for dynamic data-dependent applications as well. We also demonstrated that using HPDF as a metamodel allows us to retain much of the useful structure of the underlying models. We also believe that HPDF can be easily extended to other underlying dataflow models which have a well-defined notion of graph iteration. We also presented various properties and capabilities of HPDF — we defined the notion of repetitions vector, valid schedule for HPDF and proved that an existing algorithm (APGAN) can be used to derive efficient looped schedule for HPDF. We also gave a framework for derivation of a single-rate equivalent for HPDF. We presented three in-depth examples of image processing applications where we have applied our HPDF modeling technique to expose inherent parallelism. We presented the models for a gesture recognition application, an image registration application, and a Gait-DNA application. With the advent of model-based hardware generation, which is aspiring to be the bridging gap between hardware and software designers, our work is one of the first dynamic dataflow-based approach towards hardware synthesis. Use of static dataflow is quite limited to complex modern applications, thus our effort is a significant improvement over existing SDF based hardware code generation. Traditional handwritten hardware codes will almost always outperform hardware generated through an automated tool, but with more and more complex systems, hand-written hardware generation is becoming more time-consuming and error prone. Our work provides a framework that software designers should abide by during the design of their algorithms so that efficient hardware can be generated at a later stage. We believe without such a formal framework, hardware design either will be of poor quality, or a hardware designer will have to manually reorganize the algorithm to generate efficient hardware.

We presented a dataflow graph transformation technique — node unfolding, which is an effective technique for design space exploration with the goal of maximizing throughput when the final implementation is targeted towards hardware. We also presented a preliminary demonstration of a verilog code generator from dataflow graphs that combined with the graph transformation techniques can serve as an effective tool for model-based hardware code generation. Such a high-level transformation takes advantage of application properties that a modern synthesizer cannot take because these high-level graph properties get obscured at the implementation level. Graph transformations along with standard synthesizer optimization techniques present the opportunities for a better overall system.

We have presented in this thesis an extensive demonstration of FPGA implementations from the HPDF model of applications for two applications. We also presented how the HPDF model exposes inherent parallelism that might otherwise be obscured by the implementation details. We also demonstrated the usefulness of our modeling technique by exploring the various design points early in the design phase. We targeted the gesture recognition application to a Xilinx Multimedia and Microblaze board with a Virtex II FPGA on board and presented trade-offs between different memory layout schemes for image storage. We also presented a HPDF-based mapping of a 3-D image registration application onto an Altera StratixII FPGA. We explored the area-performance trade-offs between different design points representing different degrees of parallelism exposed by HPDF. We also presented a dynamically reconfigurable architecture for the image registration algorithm which based on some input characteristics we defined, will change its architecture to arrive at a good area-performance design point. Our FPGA implementations not only exemplified the efficacy of our model and architecture exploration techniques, they also behave as a prototype implementation which can be transformed into ASIC designs for the final implementation of a system. Also FPGA served as the ideal platform for our dynamic reconfiguration technique for image registration.

We also presented based on previous work (Compaan) done at Leiden University, The Netherlands, that sequential affine nested loop MATLAB programs can be expressed as a special case of cyclo-static dataflow graph (CSDF), but with additional non-dataflow properties required to derive a comprehensive correspondence. We proposed extensions to the dataflow interchange format (DIF) to capture the additional information that Compaan produces through its intermediate CSDF representation.

We illustrated some additional interesting results that were done in the Appendices that could be further explored in future.

8.2. Future Work

We believe that the work presented in this thesis is one of possible large impact as model-based hardware design is gaining momentum in industry right now. This work introduces a framework instead of an automated tool for hardware code generation. Most of the current tools come with a predefined set of libraries, which are not adequate for new and better algorithms being constantly developed by the algorithm development community. Hence, the current tools can only be used for certain “basic” algorithms. Since it is not feasible to ask an algorithm developer to use only a certain number of pre-defined blocks for his/her design but at the same time, an algorithm developed with no hardware implementation in mind is in most cases going to be a suboptimal design for hardware, hence we think that our framework can be used as a guideline for algorithm developers (without being too restrictive for new algorithm development) for efficient hardware implementation at a later stage. That said, we think that our framework can also be used as the backbone for an automated hardware code generation tool at a later stage. Since such a tool will be based on dataflow, dataflow related formal properties and graph transformation techniques (such as node unfolding) can be systematically used in addition to ad-hoc optimizations that the current synthesis tools provide for target specific hardware code generation.

We think that the preliminary work done on proving that Compaan Process Networks are a special case of Cyclo-static dataflow is a major step towards providing a common platform through with the two formal models can exchange information. Both the models are extremely rich and well studied and a wealth of work has been done on both of them, however, each one has some fundamental limitations that the other model does not.

For example, static buffer management cannot be guaranteed through Compaan Process Networks and CPNs do not have static schedules which are basic properties of static dataflow and also some forms of dynamic dataflow (HPDF for example). Dataflow on the other hand cannot be used very effectively to model extensive control dependent applications. Thus when an application can be modeled with either of them, exchange of such information will add valuable information towards more efficient implementation either in software or in hardware. There exists a toolflow of Compaan and Laura which starting from a (restricted) MATLAB code builds a multi-processor (multiple virtual processors) implementation on FPGAs through the use of Process Networks as the formal model. On the other hand, dataflow graph is extremely well suited for software synthesis of DSP applications. Hence we think that our preliminary work when fully explored can lead to a very efficient MATLAB to KPN to CSDF to software code synthesis on one hand, and DSP application to CSDF to KPN to code generation for FPGA as shown in Figure 8.1.

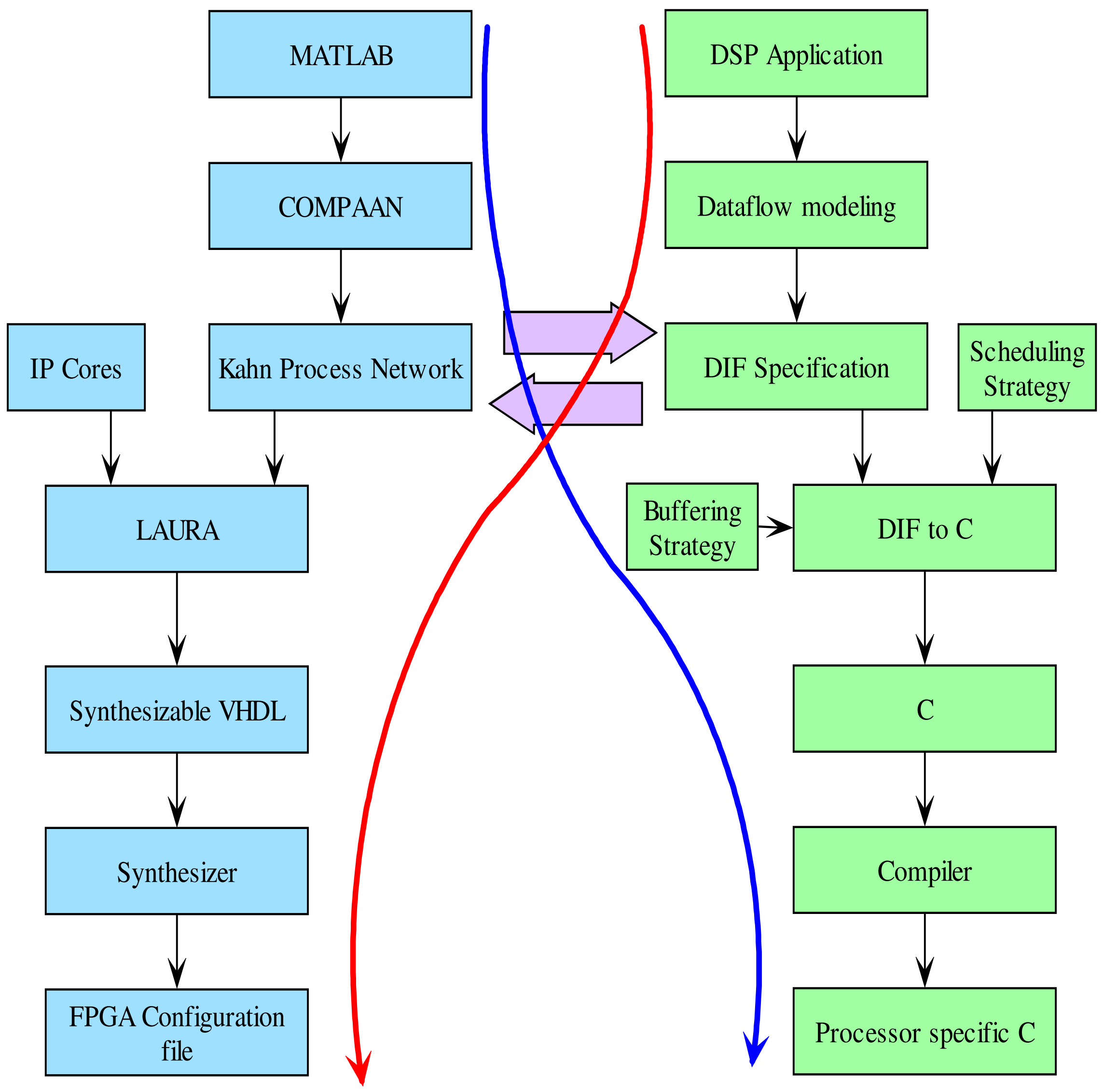


Figure 8.1 The two existing flows can be merged using our proposed intermediate representation as shown by the two curved lines.

Appendix A

A.1. Modeling of SOBEL

The Sobel algorithm was modeled by hand - this is an application that has been explored in details at Leiden. The application can be implemented in two different ways depending on the way the input image is scanned. Both the implementations were modeled by hand using dataflow graph, in specific CSDF was used as the model.

The model without data behavior has the following structure, the exact dataflow model depends on the way the image is scanned and will be represented later in the section.

A.2. Description of the application

The algorithm has five distinct processing blocks, a block which represents the input which in this case is a streaming input from which a 3×3 window is sent to the next block which does edge detection based on the SOBEL algorithm. These 3×3 pixels are

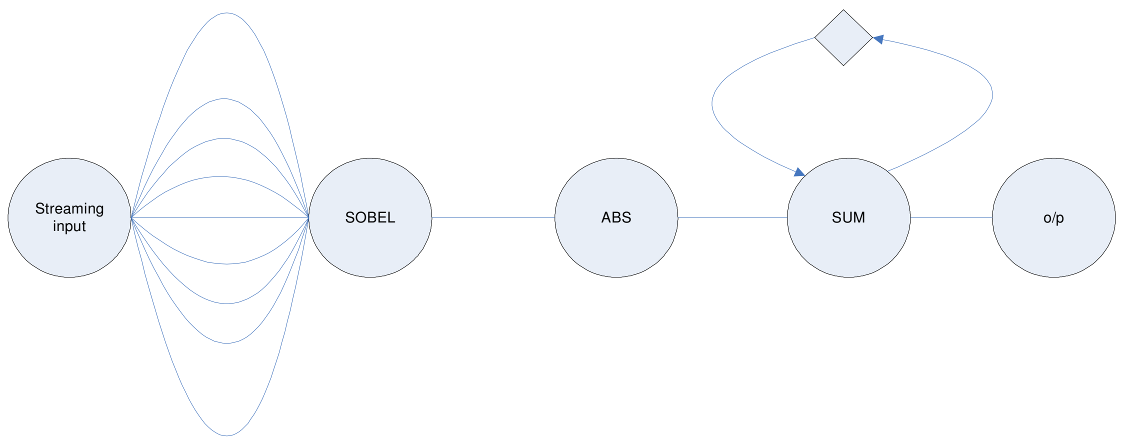


Figure A.1 The flowchart of the application.

shown by the nine edges, the point to be noted is that since the input is streaming, and pixels come in row by row, so input modeling is very important to accurately reflect the behavior between the first two blocks. Output of SOBEL is sent through an absolute value operator and it is sent to an accumulator. The accumulator adds up the output of ABS for each column and outputs one number per column, so the output of the algorithm is one row of numbers.

A.3. Two different implementations

The algorithm can be implemented in two different ways, the 3×3 window mentioned in Section A.1 can either slide horizontally or it can slide vertically, the difference is noted in the way the inputs are handled and the way accumulator treats the incoming data. Though the output result will be the same in both the cases, a static analysis provides different buffer requirements on the various edges which provides an interesting topic for exploration.

A.4. Input modeling

Input modeling is of prime importance in this application as the input is streaming and it comes in a rowwise manner. However, the next processing block - SOBEL needs the inputs in the form of a 3×3 window, so there is a lot of internal storing that happens on the edges. The generic technique used for storing the intermediate data is to copy the data as many times as needed on the appropriate edges at the appropriate time instant for future use. For a more detailed explanation, please see Figure A.2 which shows the buffer arrangement for the nine different edges when horizontal sliding is applied on the image for an image of size 4×6. Two different techniques are applied in the case of a horizontal scanning and vertical scanning which are explained in the next two sections.

A.5. Horizontal scanning

If the input image is m pixels wide and n pixels high, then generalizing Figure A.2, we get that the firing rules on the edges between Streaming Input and SOBEL for a horizontal scanning should be :

- $([n-2]([m-2]\ 1\ 2\ 0))(2m\ 0)$

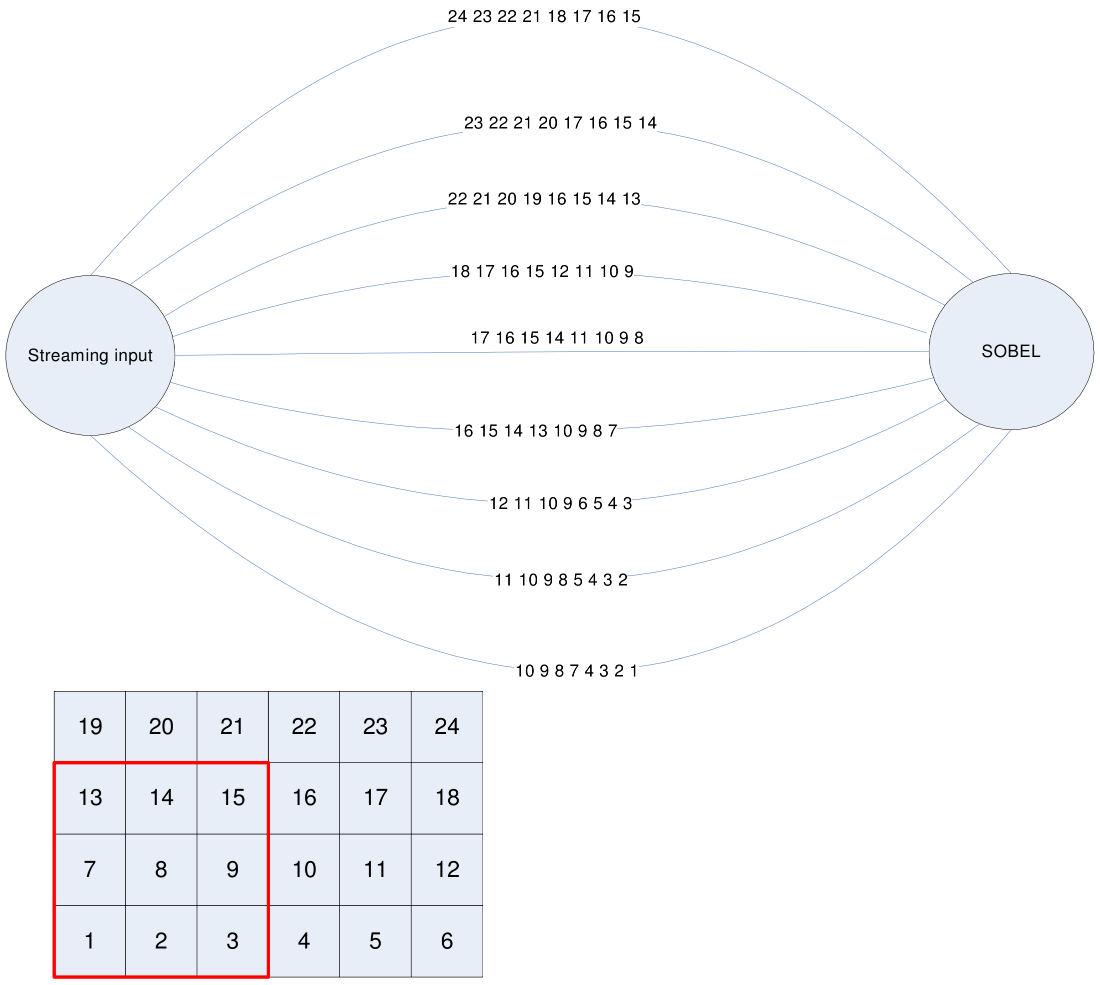


Figure A.2 An example showing the pixels to be stored on different edges where m = 6 and n = 4.

- $1\ 0([n-2]([m-2]\ 1\ 2\ 0))([2m-1]\ 0)$
- $2\ 0([n-2]([m-2]\ 1\ 2\ 0))([2m-2]\ 0)$
- $(m\ 0)([n-2]([m-2]\ 1\ 2\ 0))(m\ 0)$
- $(m+1\ 0)([n-2]([m-2]\ 1\ 2\ 0))(m-1\ 0)$
- $(m+2\ 0)([n-2]([m-2]\ 1\ 2\ 0))(m-2\ 0)$
- $(2m\ 0)([n-2]([m-2]\ 1\ 2\ 0))$
- $(2m+1\ 0)([n-3]([m-2]\ 1\ 2\ 0))([m-2]1)1\ 0$
- $(2m+2\ 0)([n-3]([m-2]\ 1\ 2\ 0))([m-2]1)$

in that order going from the top edge downwards.

A.6. Vertical Scanning

For vertical scanning the edges should have the following phases : (it can be easily derived from the horizontal scanning by interchaning m and n).

- $([m-2]([n-2]\ 1\ 2\ 0))(2n\ 0)$

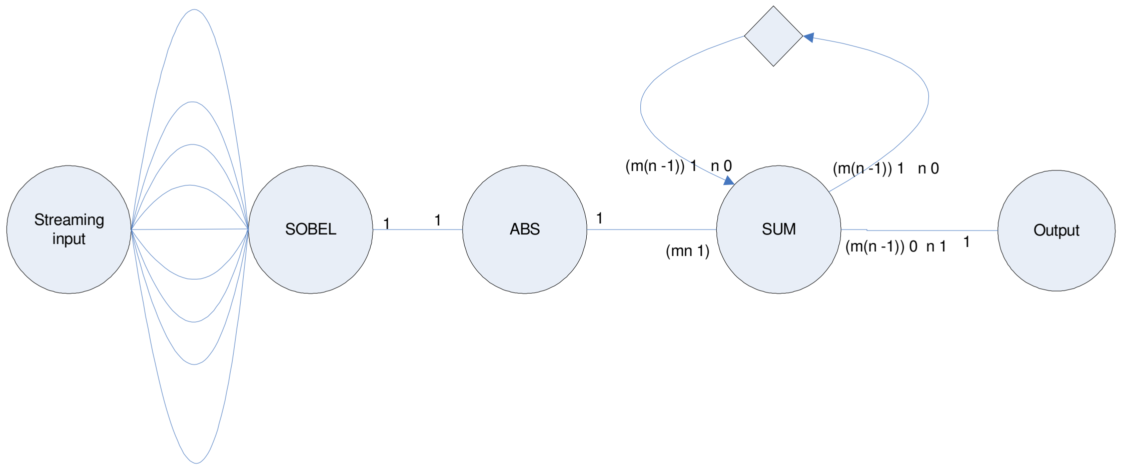


Figure A.3 CSDF modeling of the model of the application with horizontal scanning

- $1\ 0([m-2]([n-2]\ 1\ 2\ 0))(2n-1\ 0)$
- $2\ 0([m-2]([n-2]\ 1\ 2\ 0))(2n-2\ 0)$
- $(n\ 0)([m-2]([n-2]\ 1\ 2\ 0))(n\ 0)$
- $(n+1\ 0)([m-2]([n-2]\ 1\ 2\ 0))(n-1\ 0)$
- $(n+2\ 0)([m-2]([n-2]\ 1\ 2\ 0))(n-2\ 0)$
- $(2n\ 0)([m-2]([n-2]\ 1\ 2\ 0))$
- $(2n+1\ 0)([m-3]([n-2]\ 1\ 2\ 0))([n-2]1)1\ 0$
- $(2n+2\ 0)([m-3]([n-2]\ 1\ 2\ 0))([n-2]1)$

A.7. Schedule and buffer calculations

We calculate a valid schedule for the application for both type of scanning assuming it to run sequentially on a single processor.

A.7.1. Horizontal scanning

One of the possible schedules for the horizontal scanning could be :

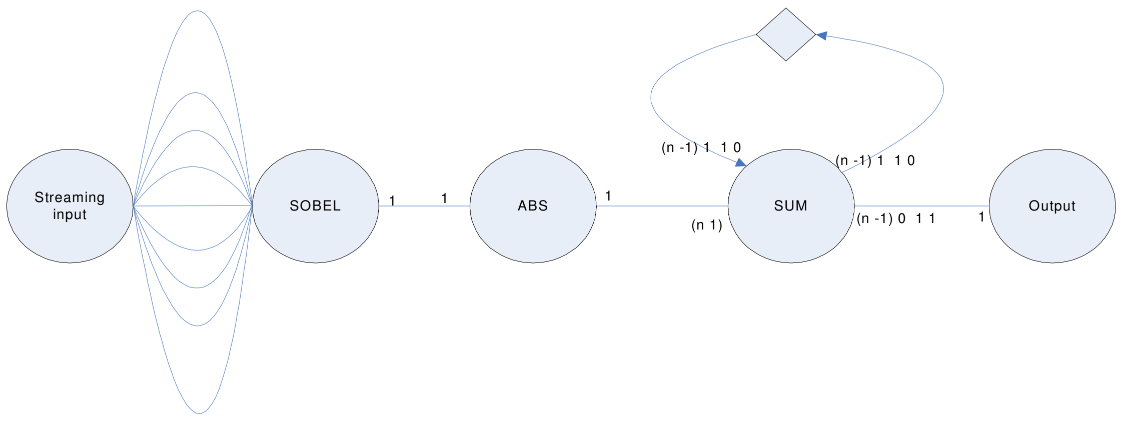


Figure A.4 CSDF modeling of the application for vertical scanning. The buffer is initialized to zero for each iteration of the graph.

$([2m+2]\ A)([(n-3)(m-2)]ABCD)(mABCDE)$

The buffer is calculated with the following logic, maximum buffer is needed on an edge when a process fires the maximum number of times before the following actor on the edge fires once. With that logic, the maximum amount of buffer needed is calculated as follows:

max A-B (2m+3) AB => buffer size is (2m+3) + (2m+2) + (2m+1) + (m+3) + (m+2) + (m+1) + 3 + 2 + 1 = 9m + 18

max B-C BC => buffer size is 1

max C-D CD => buffer size is 1

max D-D cannot be calculated such

max D-E (n-3)(m-2) + 1 D E => buffer size (n-3)(m-2) + 1

A.7.2. Vertical scanning

One of the possible schedules for the vertical scanning could be :

$([2n+2]A)([m-2]([n-2]ABCD)E)$

The maximum amount of buffer needed is calculated as follows:

max A-B (2n+3) AB => buffer size is (2n+3) + (2n+2) + (2n+1) + (n+3) + (n+2) + (n+1) + 3 + 2 + 1 = 9n + 18

max B-C BC => buffer size is 1

max C-D CD => buffer size is 1

max D-D cannot be calculated such

max D-E (n-2)D E => buffer size (n-2)

Appendix B

B.1. MJPEG modeling in CSDF

We present a simplified version of the Motion-JPEG(MJPEG) algorithm in Fig. A.5 where the image size in pixels is $64 \times N \times M$ and M is the number of vertical 8×8 pixel blocks and N is the number of horizontal pixel blocks — M, N both being parameters. In the application, the image is worked upon by 4 set of parallel hardware and depending on how the partioning is done, we can get different CSDFs. For example. if the partitioning of the image is done as shown in Fig. A.6 and its corresponding CSDF is shown in Fig. A.7. In Fig. A.5, DCT represents Discrete Cosine Transform, Q represents the quantizer, VLE represents the Variable length encoder.

This example is mainly provided as in many image processing algorithms, we see a similar ordering of pixels (or blocks in image) in which input is required. Since this order-

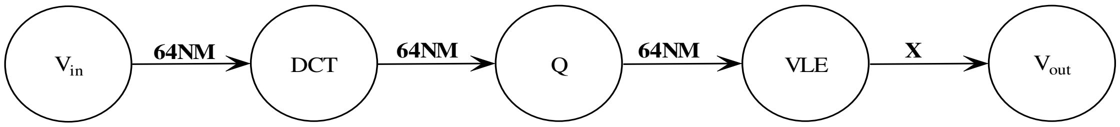


Figure A.5 A simplied version of MJPEG algorithm in dataflow representation.

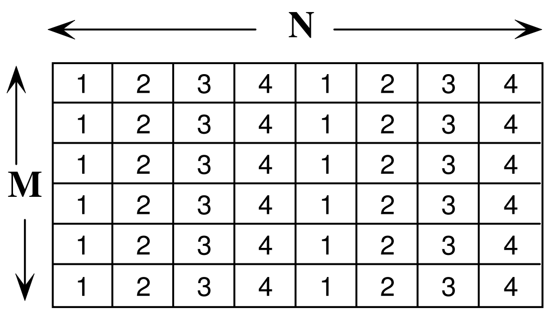


Figure A.6 4 way image partioning for MJPEG where each numbered block is 8×8.

ing is not the same as the input ordering, the CSDF exposes huge buffer overhead on the edges which can also be seen intuitively as the equivalent of a reorder buffer.

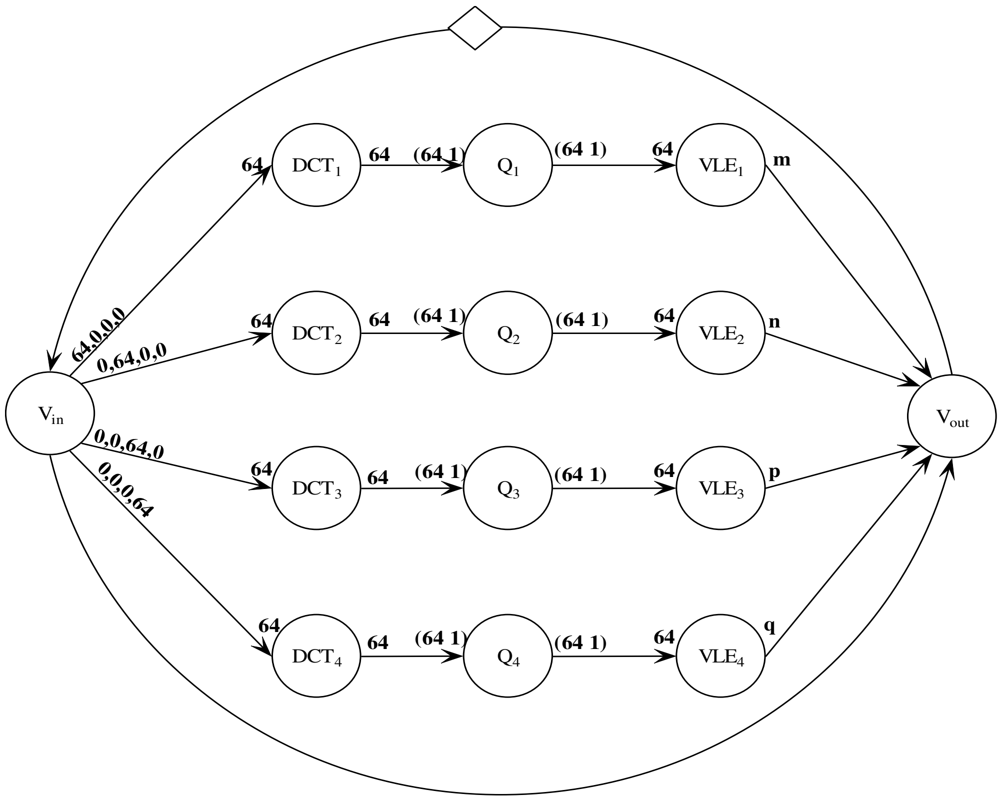


Figure A.7 CSDF representation of the MJPEG considering a partition as shown in Fig. A.6.

BIBLIOGRAPHY

[1] P. Banerjee, D. Bagchi, M. Haldar, A. Nayak, V. Kim, and R. Uribe, *Automatic Conversion of Floating-point MATLAB Programs into Fixed-point FPGA based Hardware Design*, Proceedings of the 41st Annual Conference on Design Automation, pp. 484-487, 2004.

[2] B. Bhattacharya and S. S. Bhattacharyya. Quasi-static scheduling of reconfigurable dataflow graphs for DSP systems. In *Proceedings of the International Workshop on Rapid System Prototyping*, pages 84-89, Paris, France, June 2000.

[3] B. Bhattacharya, S. S. Bhattacharyya. Parameterized Dataflow Modeling for DSP Systems. *IEEE Transactions on Signal Processing*. 49(10):2408-2410, October 2001.

[4] S. S. Bhattacharyya, R. Leupers, and P. Marwedel. Software synthesis and code generation for DSP. *IEEE Transactions on Circuits and Systems — II: Analog and Digital Signal Processing*, 47(9):849-875, September 2000.

[5] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations. *Journal of Design Automation for Embedded Systems*, 2(1):33-60, January 1997.

[6] G Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-Static Dataflow. *IEEE Transactions on Signal Processing*. Vol 44, No 2, February 1996.

[7] J. T. Buck. A Dynamic Dataflow Model Suitable for Efficient Mixed Hardware and Software Implementations of DSP Applications. *Proceedings of the 3rd international workshop on Hardware/software co-design.* Pages 165-172, 1994.

[8] J. T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control systems. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems and Computers,* pages 508-513, October 1994.

[9] C. R. Castro-Pareja, B. Daly, and R. Shekhar. Elastic registration using 3D chainmail. In *Proc. of SPIE (Medical Imaging)*, 2006.

[10] C. Castro-Pareja, J M. Jagadeesh, and R. Shekhar. FAIR: A hardware architecture for real-time 3-d image registration. *IEEE Trans. on Information Technology in Biomedicine*, 7(4):426-434, 2003.

[11] S. Chappell, C. Sullivan. Handel-C for co-processing & co-design of Field Programmable System on Chip. White paper, Sept 2002.

[12] P. Clauss and V. Loechner. Parametric analysis of polyhedral iteration spaces. In *IEEE International Conference on Application Specific Array Processors,* ASAP'96, Chicago, Illinois, August 1996.

[13] O. Dandekar, V. Walimbe, K. Siddiqui, and R. Shekhar. Image registration accuracy with low-dose CT: How low can we go? In *Proc. of IEEE International Symposium on Biomedical Imaging*, pages 502-505, 2006.

[14] E. F. Deprettere, T. Stefanov, S. S. Bhattacharyya, M. Sen. Affine nested loop programs and their binary cyclo-static dataflow counterparts. In *Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors*, pages 186-190, Steamboat Springs, Colorado, September 2006.

[15] E. F. Deprettere, E. Rijpkema, and B. Kienhuis. Translating imperative affine nested loop programs to process network. In E. F. Deprettere, J. Teich, and V. Vassiliadis, editors, *Embedded Processor Design Challenges, LNCS 2268*, pages 89-111. Springer, Berlin, 2002.

[16] Edited by P. M. Dew, R. A. Earnshaw, T. R. Heywood. “Parallel processing for Computer Vision and Display” *Addison-Wesley publishing Company.* 1989.

[17] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE,* January 2003.

[18] G. R. Gao, R. Govindarajan, and P. Panangaden. Well-behaved programs for DSP computation. In *Proceedings of International Conference on Acoustics, Speech, and Signal Processing,* March 1992.

[19] M. Gokhale, J. Stone, J. Arnold, M. Kalinowski. Stream-oriented FPGA computing in the Streams-C High Level Language. In IEEE international symposium on Field-Programmable Custom Computing Machines.

[20] F. Haim, M. Sen, D. Ko, S. S. Bhattacharyya, and W. Wolf. Mapping multimedia applications onto configurable hardware with parameterized cyclo-static dataflow graphs. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages III-1052-III-1055, May 2006.

[21] Y. Hemaraj, M. Sen, S. Bhattacharyya, R. Shekhar, "Model-based Mapping of Image Registration Applications onto Configurable Hardware", In *Proceedings of Asilomar Conference on Signals, Systems, and Computers,* Pacific Grove, CA, 2006.

[22] C. A. R. Hoare. Communicating Sequential Processes. *Prentice Hall, 1985*.

[23] M. Holden, D. Hill, E. Denton, J. Jarosz, T. Cox, T. Rohlfing, J. Goodey, and D. Hawkes. Voxel similarity measures for 3D serial MR brain image registration. *IEEE Trans. on Medical Imaging*, pages 94-102, 2000.

[24] J. Horstmannshoff, T. Grötker, and H. Meyr. Mapping multirate dataflow to complex RT level hardware models. In *Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors*, pages 283-293, July 1997.

[25] C. Hsu and S. S. Bhattacharyya. Dataflow interchange format version 0.2. *Technical Report UMIACS-TR-2004-66*, Institute for Advanced Computer Studies, University of Maryland at College Park, November 2004. Also Computer Science Technical Report CS-TR-4624.

[26] A. K. Jain. Fundamentals of Digital Image Processing. *Prentice Hall,* 1989.

[27] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress*, 1974.

[28] B. Kienhuis, E. Rijpkema, and E. F. Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. *8th International Workshop on Hardware/Software Codesign (CODES'00),* May 2000, San Diego. CA.

[29] B. Keinhuis, and E. F. Deprettere. Modeling stram based applications using the SBF model of computation. *Journal of VLSI Signal Processing Systems,* 34(3):291-299, July 2003.

[30] B. Kienhuis. MatParser: An array dataflow analysis compiler. Technical report, University of California at Berkeley, 2000. UCB/ERL M00/9.

[31] E. Lee and D. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, pages 55–64, September 1987.

[32] E. A. Lee. Multidimensional Streams Rooted in Dataflow. *Proceedings of the IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism,* January 1993.

[33] E. Lee and T. Parks. Dataflow process networks. *Proceedings of the IEEE,* vol. 83, no.5, pp. 773-799, May 1995.

[34] C. H. Lin, T. Lv, W. Wolf, I. B. Ozer. A Peer-to-peer architecture for distributed real-time gesture recognition. *IEEE International Conference on Multimedia and* Expo (ICME 2004).

[35] F. Maes, D. Vandermeulen and P. Suetens, Medical image registration using mutual information, *Proc. IEEE* 19, 1699 (2003).

[36] J. B. Maintz and M. Viergever, "A survey of medical image registration" *Med. Image Anal.,*vol 2, no. 1, pp. 1—36, 1998.

[37] V. Muthukumar, D.V. Rao. Image processing algorithms on reconfigurable architecture using HandelC. *Proceedings of Digital System Design (DSD), 2004*. pages:218 - 226.

[38] S. Neuendorffer, E. Lee. Hierarchical reconfiguration of dataflow models. *Conference on Formal Methods and Models for Codesign (MEMOCODE)*, June 22-25, 2004.

[39] K. K. Parhi. High-level algorithm and architecture transformations for DSP synthesis. *Journal of VLSI Signal Processing,* 9, 121-143 (1995).

[40] Edited by V. K. Prasanna Kumar. Parallel architectures and algorithms for Image Understanding. *Academic Press Inc.* 1991.

[41] J. P. W. Pluim, J. B. A. Maintz, M. A. Viergever, Mutual Information based Registration of Medical Images: A Survey. *IEEE Trans on Medical Imaging*, 2003.

[42] Y. Ran, R. Challappa, Q. Zheng. A compact charachterization of human gait and acitivities. In Preperation.

[43] E. Rijpkema. Modeling task level parallelism in piece-wise regular programs. PhD thesis, Leiden Institute of Advanced Computer Sciences, Leiden University, The Netherlands, September, 2002.

[44] M. Sen, S. S. Bhattacharyya, T. Lv, W. Wolf. Modeling Image Processing Systems with Homogeneous Parameterized Dataflow Graphs. *ICASSP 2005*.

[45] M. Sen, S. S. Bhattacharyya. Systematic exploitation of data parallelism in hardware synthesis of DSP applications. In Proceedings of the *International Conference on Acoustics, Speech, and Signal Processing,* pages V-229-V-232, Montreal, Canada, May 2004.

[46] M. Sen, I. Corretjer, F. Haim, S. Saha, J. Schlessman, T. Lv, S. S. Bhattacharyya, W. Wolf, Dataflow-based Mapping of Computer Vision Algorithms onto FPGAs. *EURASIP Journal on Embedded Systems*, accepted for publication.

[47] M. Sen, I. Corretjer, F. Haim, S. Saha, S. Bhattacharyya, J. Schlessman, W. Wolf, "Computer Vision on FPGAs: Design Methodology and its Application to Gesture

Recognition", *IEEE Workshop on Embedded Computer Vision, (ECV 2005)*, New York, USA.

[48] M.Sen, Y. Hemaraj, S. Bhattacharyya, R. Shekhar, "Reconfigurable Image Registration on FPGA platforms", In *Proceedings of Biomedical Circuits and Systems* (BioCAS'06), London, UK, Nov 2006.

[49] R. Shekhar, V. Walimbe, S. Raja, V. Zagrodsky, M. Kanvinde, G. Wu, and B. Bybel. Automated three-dimensional elastic registration of whole-body PET and CT from separate or combined scanners. *Journal of Nuclear Medicine*, 46(9):1488-1496, 2005.

[50] R. Shekhar R, V. Zagrodsky, C. R. Castro-Pareja, V. Walimbe, and J. M. Jagadeesh. High-speed registration of three- and four-dimensional medical images by using voxel similarity. *RadioGraphics*, 23(6):1673-1681, 2003.

[51] S. Sriram and S. S. Bhattacharyya. Embedded Multiprocessors: Scheduling and Synchronization. *Marcel Dekker, Inc., 2000.*

[52] D. E. Thomas, P. Moorby. The Verilog hardware description language. *Kluwer Academic Publisher,* Nowell Massachusetts 1991.

[53] A. Turjan, B. Kienhuis, and E Deprettere. Realization of the extended linearization model. *Marcel Dekker, Inc. Domain specific processors: Systems, Architectures, Modeling and Simulation edition, 2003.*

[54] M. C. Williamson and E. A. Lee. Synthesis of parallel hardware implementations from synchronous dataflow graph specifications. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, 1996.

[55] W. Wolf, B. Ozer, T. Lv. Smart cameras as embedded systems. *IEEE Computer Magazine* Vol 35, Iss 9, Sept 2002, Pages 48-53.

[56] Claudiu Zissulescu, Todor Stefanov, Bart Kienhuis, and Ed Deprettere, "LAURA: Leiden Architecture Research and Exploration Tool," in *Proc. 13th Int. Conference on Field Programmable Logic and Applications (FPL'03)*, 2003.

[57] IEEE Working Group, http://www.eda.org/vhdl-200x/vhdl-200x-ft/packages/files.html.

[58] Data-sheet for ZBT memory,http://www.samsung.com/Products/Semiconductor/SRAM/SyncSRAM/NtRAM_FT_n_PP/16Mbit/K7N163631B/ds_k7n16xx31b_rev04.pdf

[59] OpenCores Organization, “WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores”, revision B.3, September 2002, www.opencores.org.

[60] Synopsys Design Compiler User Manual, *Synopsys.*