ABSTRACT

Title of Document: **DESIGN METHODOLOGY FOR EMBEDDED COMPUTER VISION SYSTEMS**

Sankalita Saha, Doctor of Philosophy, 2007

Directed By: Professor Shuvra S.Bhattacharyya
Department of Electrical and Computer Engineering

Computer vision has emerged as one of the most popular domains of embedded applications. Though various new powerful embedded platforms to support such applications have emerged in recent years, there is a distinct lack of efficient domain-specific synthesis techniques for optimized implementation of such systems. In this thesis, four different aspects that contribute to efficient design and synthesis of such systems are explored:

(1) Graph Transformations: Dataflow modeling is widely used in digital signal processing (DSP) systems. However, support for dynamic behavior in such systems exists mainly at the modeling level and there is a lack of optimized synthesis techniques for these models. New transformation techniques for efficient system-on-chip (SoC) design methods are proposed and implemented for cyclo-static dataflow and its parameterized version (parameterized cyclo-static dataflow) — two powerful models that allow dynamic reconfigurability and phased behavior in DSP systems.

(2) Design Space Exploration: The broad range of target platforms along with the

complexity of applications provide a vast design space, calling for efficient tools to explore this space and produce effective design choices. A novel architectural level design methodology based on a formalism called multirate synchronization graphs is presented along with methods for performance evaluation.

(3) Multiprocessor Communication Interface: Efficient code synthesis for emerging new parallel architectures is an important and sparsely-explored problem. A widely-encountered problem in this regard is efficient communication between processors running different sub-systems. A widely used tool in the domain of general-purpose multiprocessor clusters is MPI (Message Passing Interface). However, this does not scale well for embedded DSP systems. A new, powerful and highly optimized communication interface for multiprocessor signal processing systems is presented in this work that is based on the integration of relevant properties of MPI with dataflow semantics.

(4) Parameterized Design Framework for Particle Filters: Particle filter systems constitute an important class of applications used in a wide number of fields. An efficient design and implementation framework for such systems has been implemented based on the observation that a large number of such applications exhibit similar properties. The key properties of such applications are identified and parameterized appropriately to realize different systems that represent useful trade-off points in the space of possible implementations.

DESIGN METHODOLOGY FOR EMBEDDED COMPUTER VISION SYSTEMS

By

Sankalita Saha

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
2007

Advisory Committee:

Professor Shuvra S. Bhattacharyya, Chairman/Advisor
Professor Rama Chellappa
Professor Manoj Franklin
Professor Ramani Duraiswami
Dr. Neal K. Bambha

Dedication

To my parents, sister and to Bhaskar

Acknowledgements

First of all, I would like to thank my advisor, Dr. Shuvra S. Bhattacharyya for his invaluable guidance and support throughout my PhD work and also his academic and professional guidance throughout the program. He provided me with enormous freedom in choosing the direction of my research while providing the right insight at the right time to ensure that I did not get lost. His valuable advice in approaching a research problem in different ways as well as patience and understanding ensured that my experience was both learning and enjoyable.

I would also like to thank my dissertation committee members, Dr. Rama Chellappa, Dr. Manoj Franklin, Dr. Neal K. Bambha and Dr. Ramani Duraiswami for their cooperation and support. I would specially like to thank Dr. Bambha for his help and collaboration in my work that resulted in an important chapter of this thesis.

The research reported in this thesis was supported by grant number 0325119 from the U.S. National Science Foundation.

A big thanks to all the members and friends of the DSPCAD group at University of Maryland who have greatly aided me in this research, especially to Dr. Vida Kianzad, Dr. Ming-Yung Ko, Dr. Dong-Ik Ko, Dr. Jerry Hsu, Dr. Mainak Sen, Chung-Ching Shen, Nimish Sane, Sebastian Puthenpurayil, Ruirui Gu and Hojin Kee. A special thanks to Vida, Ming-Yung and Dong-Ik for their encouragement and guidance in the initial years of my PhD work and Nimish for his help in the last few days.

My deepest gratitude goes to my parents, Prof. R. K. Saha and Ratna Saha, for their unconditional encouragement, support and love and without whose initiation I would not have pursued this studies in the first place and my sister Swatilekha for

always believing in me. I would like to thank my friends, Paramita, Rachana, Shailesh, Sibasish, Rajarshi, Ayush, Ayan, Abhishek, Ankush, Brinda, Surupa, Sumesh, Vishal, Gaurav, Ashok, Rania, Supratik, Uma, Ritaja, Indrajit, Subhamoy, Anyesha, Divya, Mithun, Mrinmoy, Bhaskar, Sunitha, Anand, Vivek and all those that I fail to mention, who made my experience as a graduate student a truly enjoyable one.

Last but definitely not the least, I would like to thank my husband and best friend Bhaskar — my greatest source of support, joy and happiness — for being there with me at all times.

Table of Contents

List of Figures

List of Tables

Chapter 1 : Introduction

Embedded computer vision systems have become common occurrences in our day-to-day consumer lives with the advent of cell-phones, PDAs, cameras, portable game systems, and so on. The complexity of such embedded systems is expected to rise even further as consumers demand more functionality out of such devices. To support such complex systems, new heterogeneous multiprocessor System-on-Chip (SoC) platforms have already emerged in the market. There is no dearth of work regarding suitable architectures for such applications which vary, from dedicated and programmable to configurable processors such as programmable DSP, ASIC, FPGA (Field Programmable Gate Array) subsystems and their combinations as demonstrated by these platforms. However, performance and resource constraints combined with the complexity of the platforms as well as the applications have made the job of the system designer more difficult than ever before. The lack of software and tool support for these architectures is huge and is now widely recognized as one of the most challenging and important areas in the field of embedded processing systems. In this thesis, some of the system design and implementation problems for such systems have been addressed.

The problem of implementing computer vision applications on multiprocessor platforms consists of several sub-problems. In this work, the problem has been categorized into different subtasks, each of which is vast and rich with complex, multifaceted problems, and can very well be considered a separate field of study. However, in this work the approach is streamlined to focus on dataflow techniques. Dataflow is widely

considered as one of the most natural models for representation of signal processing systems. Along with modeling and specification, abundant work exists on optimized techniques for implementation of dataflow-specified DSP applications on embedded systems. Thus, by concentrating on this important modeling technique, various existing tools can be successfully leveraged to address the more demanding and pertinent problems in order to make implementations of computer vision systems more efficient, cost-effective, and predictable. The focus in this thesis is more from the system synthesis perspective and the next sections elaborate on the specific problems addressed.

1.1 Contributions of the Thesis

1.1.1 Modeling and Synthesis

A suitable model to specify an application is an extremely important first step towards an efficient implementation. Dataflow is an established standard for modeling signal processing applications. Amongst the various dataflow models, synchronous dataflow (SDF) has been the most widely used model, but it suffers from limited expressivity. There have been numerous efforts to develop models to overcome this. Though most of these models provide more flexibility, there has not been much effort to develop optimized synthesis techniques for them. In this work, new synthesis techniques for a popular variant of synchronous dataflow — cyclo-static dataflow (CSDF) and its parameterized version, parameterized cyclo-static dataflow (PCSDF) — along with the first in-depth analysis of parameterized cyclo-static dataflow graphs have

been proposed.

The class of synthesis techniques addressed in this contribution are transformation techniques and associated scheduling algorithms that are important in many contexts of System-on-Chip (SoC) implementation, particularly in the domain of signal processing. Most previous work on dataflow graph transformations has focused on SDF, and closely-related graph representations, including single-rate and homogeneous synchronous dataflow. However, modern SoC applications are often not fully amenable to SDF. This is because SDF has limited expressive power and cannot capture dynamically-varying patterns of data production and consumption between computational modules, as well as dynamic modes of operations involving different modules and interconnections. In this work, methods of dataflow transformation for CSDF graphs and its parameterized version i.e., parameterized cyclo-static dataflow (PCSDF) graphs — a powerful new form of reconfigurable dataflow graph modeling — that derives its properties from parameterized dataflow and CSDF are presented. The analysis and algorithms developed demonstrate the high expressive power, and potential for efficient memory management, low latency operation, and improved hardware utilization ability of parameterized cyclo-static dataflow in the design and implementation of signal processing SoCs.

1.1.2 Design Space Exploration

Embedded computer vision applications are characterized by increased functionality, and hence increased design complexity and processing requirements. The resulting design spaces are vast. As a result, designers are typically able to evaluate only

small subsets of architectural solutions, partitionings, and mappings of the system functionalities. A more comprehensive design space exploration enables designers to select higher quality solutions and provides substantial savings on the overall cost of the system. However, such exploration is not often practiced today since there is a lack of efficient methodologies and design tools to facilitate them.

In this thesis, an architectural-level design methodology that provides means for such comprehensive design space exploration is described. The methodology is demonstrated by implementation of two applications — an embedded face detection system and a 3D facial pose tracking system. The target platforms for this study includes a reconfigurable system on chip, a multiprocessor system, and a programmable digital signal processor (PDSP) system. Models for performance estimation are presented and validated with experimental values obtained from implementing the systems on different hardware and software platforms. The modeling approach is efficient, accurate, and intuitive for designers to work with. Using this approach, it is shown how a wide range of design options can be selected that trade-off various architectural features.

1.1.3 Multiprocessor Communication Interface

Parallelization of embedded software is often desirable for power/performance-related considerations for computation-intensive applications that frequently occur in the signal-processing domain. Although hardware support for parallel computation is increasingly available in embedded processing platforms, there is a distinct lack of effective software support. One of the most widely known efforts in support for parallel software is the message passing interface (MPI). However, MPI suffers from sev-

eral drawbacks with regards to customization towards specialized parallel processing contexts, and performance degradation for communication-intensive applications.

In this thesis, a new interface called the signal passing interface (SPI) is presented. SPI is targeted toward signal processing applications and addresses the limitations of MPI for this important domain of embedded software by integrating relevant properties of MPI and coarse-grain synchronous dataflow modeling. SPI is much easier and more intuitive to use, and due to its careful specialization, more performance-efficient for the targeted application domain.

This interface also provides support for dynamic dataflow behavior — not supported by synchronous dataflow modeling — by integration of the concept of variable token sizes (VTSs). Further optimization by providing methods for resynchronization is integrated into SPI as well. Two library implementations — software-based and FPGA-based — have been created. Details of the experiments with different signal processing applications have been presented.

1.1.4 Parameterized Design Framework for Particle Filter Systems

Particle filtering methods are being increasingly used in a variety of computer vision applications. For example, in smart camera systems, application of particle filters is being explored extensively for tracking objects. Most particle filters involve vast amounts of computational complexity, thereby intensifying the challenges faced in their real-time, embedded implementation. However, many of these applications share common characteristics, and the same system design can be reused by identifying key system parameters and varying them appropriately. Here, a novel SoC archi-

tecture involving parallel processing elements for a class of particle filters is presented. Along with this system architecture, a parameterized design framework is presented to enable fast and efficient reuse of the architecture with minimal re-design effort for a wide range of particle filtering applications.

Using this framework, different design options for implementing three different particle filtering applications on FPGAs are explored. The first two applications involve particle filters with one-dimensional models, and are used to demonstrate the key features of the framework while the third is a 3D facial pose tracking system for videos. In this multi-dimensional particle filtering application, the proposed architecture is extended with models for hardware/software co-design so that limited resources can be utilized most effectively. The experiments demonstrate that the framework is easy and intuitive to use, and promotes efficient design and implementation.

1.2 Outline of Thesis

The rest of the thesis is organized as follows: in chapter 2 the dataflow modeling paradigm and related work in the context of the contributions developed in this thesis are presented. In chapter 3, detailed analysis of parameterized cyclo-static dataflow along with high-level transformation and synthesis techniques for cyclo-static dataflow and parameterized cyclo-static dataflow are presented. Chapter 4 deals with the novel design space exploration methodologies developed as part of this thesis. In chapter 5, the new communication interface (Signal Passing Interface) is presented. In chapter 6, a parameterized design framework for SoC implementation of particle filter systems is presented. Finally, in chapter 7 conclusions and future work are presented.

Chapter 2 : Dataflow Modeling and Related Work

2.1 Introduction

Dataflow graphs is one of the most natural and intuitive modeling paradigm for DSP systems. In the dataflow modeling paradigm, the computational behavior of a system is represented as a directed graph $G = (V, E)$. A vertex or node in this graph $v \in V$ represents a computational module or a hierarchically nested subgraph and is called an *actor*. A directed edge $e \in E$ represents a FIFO buffer from a source actor $src(e)$ to its sink actor $snk(e)$. An edge e can have a non-negative integer delay $del(e)$ associated with it which specifies the number of initial data values (tokens) on edge e before execution of the graph. Dataflow graphs use a data-driven execution model, thus an actor v can execute (fire) only when it has sufficient numbers of data values (tokens) on all of its input edges. On firing, v consumes certain number of tokens from its input edges and executes based on its functionality to produce certain number of tokens on its output edges.

Of all the dataflow models, synchronous dataflow (SDF) proposed by Lee and Messerschmitt [48] has emerged as the most popular model, mainly due to its compile-time predictability. However, it lacks significantly in terms of expressivity. On the other hand, there is dynamic dataflow (DDF) modeling which supports arbitrary data dependent behavior using non-SDF actors with unknown token production/consumption at compile-time. DDF offers high expressivity — allowing modeling of conditionals, iterations, and recursion — but allows very little compile-time analysis and

optimizations. All the other models lie between these two extremes of modeling techniques to broaden the range of applications that can be represented while maintaining the compile-time predictability properties (like in SDF) as much as possible. In the following sections first a detailed discussion of the new models relevant to the work described in this thesis along with SDF is presented. This is followed by detailed review of related work for each of the major contributions in this thesis arranged in order of the chapters.

2.1.1 Synchronous Dataflow (SDF)

E. A. Lee and D.G. Messerschmitt had proposed the synchronous dataflow (SDF) model [48] (figure 1). In SDF the number of tokens produced ($prd(e)$) and consumed ($cns(e)$) by each actor within a dataflow model is a constant known at compile time. The static properties of SDF offer potential for thorough optimization, and effective optimization techniques have been developed in the contexts of data memory minimization [1], joint minimization of code and data [9], high-throughput block processing [64], multiprocessor scheduling [37] and a variety of other objectives. But it suffers from limited expressive power, and consequently, various alternative modeling techniques have been pursued to increase the expressivity of SDF while maintaining much or all of its potential for static analysis.

The first step towards synthesizing code from a dataflow graph is the generation of a schedule. A schedule can be static, dynamic or a combination of both. An SDF

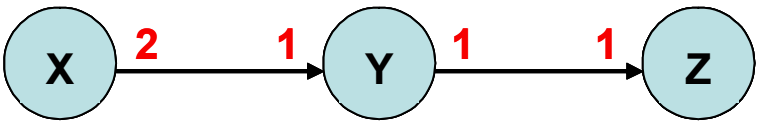

Figure 1. Example of an SDF graph.

graph $G = (V, E)$ has a valid schedule if it is free from deadlock and is sample rate consistent i.e., it has a periodic schedule that fires each actor at least once and produces no net change in the number of tokens on each edge [48]. In more precise terms, G is sample rate consistent if there is a positive integer solution to the balance equation

$$\forall e \in E, prd(e) \times \Gamma[src(e)] = cns(e) \times \Gamma[snk(e)] \tag{2.1}$$

where $src(e)$ and $snk(e)$ represent the source and sink actors respectively for edge e. When it exists, the minimum positive integer solution for the vector Γ is called the repetitions vector of G, and is denoted by q_G. For each actor v, $q_G[v]$ is referred to as the repetition count of v. Thus, a valid minimal periodic schedule (which is abbreviated as schedule hereafter in this thesis) is a sequence of actor firings in which each actor v is fired $q_G[v]$ times, where the firing sequence obeys the data-driven properties of an SDF graph.

To provide for more memory-efficient storage of schedules, actor firing sequences can be represented through looping constructs [7]. For this purpose, a schedule loop, $L = (nT_1T_2\ldots T_m)$, is defined as the successive n repetitions of the invocation sequence $T_1T_2\ldots T_m$, where each T_i is either an actor firing or a (nested) schedule loop. A looped schedule $S = L_1L_2\ldots L_m$ is an SDF schedule that is expressed in terms of the schedule loop notation. If every actor appears only once in S, it is called a single appearance schedule (SAS), otherwise, S is called a multiple appearance schedule (MAS).

2.1.2 Cyclo-Static DataFlow (CSDF)

A widely used model that tries to maintain static properties of SDF while providing increased support for expressivity is cyclo-static dataflow (CSDF) where token production and consumption can vary between actor firings as long as the variation forms a certain type of periodic pattern. Thus, over each iteration of a dataflow graph, actors under CSDF semantics can have different production and consumption rates in a cyclic and periodic pattern. The benefits that CSDF offers over SDF include increased flexibility in compactly and efficiently representing interaction between actors, decreased buffer memory requirements for some applications, and more opportunities for behavioral optimizations like constant propagation and dead code elimination [13].

CSDF, increases the expressivity of dataflow model at the expense of complicating the scheduling problem. However, since many computer vision applications show behavior that can be successfully captured by CSDF graphs [24], it is imperative to look into efficient scheduling techniques for them.

2.1.3 Parameterized DataFlow

The other significant modeling technique in this domain is the parameterized dataflow, which was introduced in [6]. A parameterized dataflow modeling emphasizes a hierarchical modeling of a dataflow and allows a subsystem's behavior to be controlled by a set of parameters. These parameters can change at runtime by allowing the subsystem behavior to vary dynamically. Parameters can control the functional behaviors of subsystems as well as the token flow behavior of a dataflow graph. In this, the model can have different parameter configurations at each iteration of a graph.

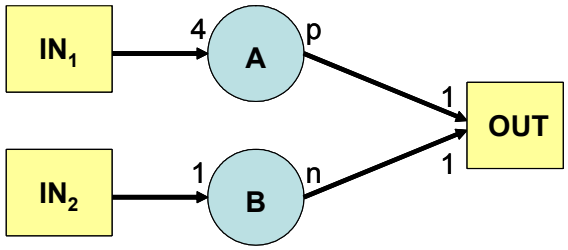

Figure 2. Example of a PSDF system.

Parameterized dataflow is a meta modeling technique. Thus when applied to SDF (PSDF), it extends SDF to allow runtime reconfiguration of parameter configurations for actors and edges in certain fixed ways. A PSDF specification consists of three distinct graphs: the init graph, the subinit graph and the body graph. Intuitively, the body graph models the main functional behavior of the subsystem, whereas the init and subinit graphs control the behavior of the body graph by appropriately configuring the body graph parameters. The init graph is invoked prior to each invocation of the associated (hierarchical) parent subsystem while the subinit graph is invoked prior to each invocation of the associated body subsystem, thus allowing for two distinct reconfiguration of controls. Figure 2 shows an example of PSDF graph. Parent A has three sub graphs. $A.subinit$ graph sets parameters of the body graph before the associated body graph is fired. PSDF increases the expressivity by adopting parameterized modeling, and exploits a quasi-static schedule.

Though parameterized dataflow is an attractive extensions to the synchronous dataflow technique, it has not achieved as widespread popularity as SDF due to lack of elegant scheduling strategies.

2.1.4 Other Dynamic Models

Boolean dataflow [15], also is an important modeling technique in which, the number of tokens produced or consumed at an edge is either fixed, or is a two-valued function of a control token present at a control terminal of the same actor. Multidimensional synchronous dataflow (MDSDF) [56] had been developed as an extension of SDF to better accommodate multidimensional representation. In MDSDF graphs, actors produce and consume M-dimensional data. For example, 2DSDF is very suitable for modeling image processing systems where actors process images and 2-dimensional data. Other examples of efforts in this area include well-behaved dataflow [31] and windowed synchronous dataflow [41].

2.2 Related Work

2.2.1 Modeling and Transformations

Appropriate modeling is crucial for any efficient and accurate solution. Associated with this step are transformations that are extremely beneficial for a final optimized implementation. High-level transformations is an effective technique for increasing optimizations in the final implementation. These techniques involve transforming a given description of the system to another description that is closer to the optimal. Though traditional focus for efficient embedded processor implementation has been on optimizing code-generation techniques and hence relevant compiler technology, high-level transformations have started gaining importance because of their inherent portability and resultant boost in performance when applied appropriately

([29], [50]). The nature of transformations can be of various kinds such as algorithmic or architectural [59], source-to-source [30] and have been studied in various contexts such as VLSI synthesis [18], DSP software synthesis [78], fault detection in parallel system [33] and so on.

While most efforts for the general case have concentrated on loop transformations [63], array manipulations ([30], [50]) minimizing syntactic variances [19] and energy-aware scheduling and transformations [50], in the domain of DSP software synthesis the efforts have encompassed various other techniques in addition to these, such as retiming [49], vectorization [64] and clustering ([38], [69], [62]). Clustering is an important technique used in the context of dataflow modeling for DSP applications for efficient scheduling and code generation for programmable DSPs (PDSPs). There have also been efforts in developing transformations involving memory organization for efficient SoC synthesis [28].

Dataflow transformation using clustering has been studied mainly for multiprocessor implementation ([26], [22]). It has been applied for uniprocessor implementation as well leading to lesser context-switching overheads [27]. However, most of the efforts have been limited to SDF and closely related SSDF (Single-rate SDF) and HSDF (Homogeneous SDF) graphs which do not allow dynamic reconfigurability. However, with increase in the design complexities, the need for models that capture dynamic behavior of components and their efficient implementation is becoming a necessity. An approach to handle dynamic behavior (dynamic stream processing) has been proposed by Geilen and Basten in [32]. In [24], the authors present a new approach to express and analyze implementation specific aspects in CSDF graphs — a

primary focus of this work — for computer vision applications with concentration only on the channel/edge implementation.

However, further development of sophisticated high-level transformation tools for these dynamic dataflow models with a focus on exploiting their specific properties — for example fine grain modeling capability of CSDF — is still required to enable efficient use of these modeling techniques in practical implementations. In this thesis, we address this gap.

The first scheduling strategy for CSDF graph — uniprocessor scheduling — was proposed by Bilsen et al. [27]. The same authors formulated the minimum repetition count for each actor in the graph [12]. Their scheduling strategy is based on a greedy heuristic that proceeds by adding one node at a time to the existing schedule; the node selected adds the minimum cost to the existing cost of the schedule. This yields a minimum buffer schedule, but the schedule is not a single appearance schedule. Also, the time complexity of the algorithm is very high. Another possible method is by decomposing a CSDF graph into an SDF graph [60]. The drawbacks of this method are (a) it is not always possible to transform the CSDF graph into a deadlock-free SDF graph and (b) it does not exploit the versatility of CSDF which can result in a better schedule.

Compared to cyclo-static dataflow, the only scheduling strategy for parameterized dataflow is the quasi-static scheduling method [7]. In a quasi-static schedule some actor firing decisions are made at run-time, but only where absolutely necessary.

In this work, novel transformation techniques and scheduling strategies for CSDF and parameterized CSDF are presented to address the aforementioned gap.

2.2.2 Architectural Design Space Exploration

With an increase in the complexity and performance requirements of applications, designers have resorted to looking beyond pure hardware or software solutions. As a result of which the total design space for a system has vastly increased, calling for systematic methods to explore this space and utilize the available resources optimally. An efficient design space exploration tool can dramatically impact the area, performance, and power consumption of the resulting systems. An optimized implementation on such hybrid platforms requires sophisticated techniques. These techniques start with the specification and design requirements of the system, and search the solution space to find an (or a set of) implementation(s) that meet(s) a single optimization goal or a set of goals. The problem consists of three major steps, i) resource (computation/communication) allocation, ii) assignment and mapping of the system functionality onto the allocated resources, and iii) scheduling and ordering of the assigned functions on their respective resources. Addressing each of these steps is an intractable problem, and most of the existing techniques are simulation-based heuristics.

Some recent studies have started to use formal models of computation as an input to the partitioning problem to allow the exploration of much larger solution spaces [77], and to use such models for design space exploration and optimization [40]. A methodology for system level design space exploration is presented in [4], but the focus is purely on partitioning and deriving system specifications from functional description of the application. Peixoto et al. gave a comprehensive framework for algorithmic and design space exploration along with definitions for several system-level metrics [61]. But the methodology presented is complex and does not address the

special needs of embedded systems. Kwon et al. proposed a design exploration framework that make estimations about performance and cost [45], but the performance estimation technique is based on instruction set simulation of architectures which restricts the platforms to which it can be applied. Miramond et al. also proposed methodologies for design space exploration as well as performance estimations but they restrict themselves to purely reconfigurable architectures [51].

There is little work that treats the problem of embedded system design with stringent, integrated attention to data accesses, interprocessor communication, and synchronization, as well as resource assignment and task scheduling. Such an integrated approach is especially important in real-time computer vision systems, where large volumes of data must be handled with predictable performance. Exploration tools that make early predictions of performance or area would be extremely beneficial in reducing redundant search space. A high-level tool that enables easy yet robust design space exploration and performance estimation is presented in this work.

2.2.3 Multiprocessor Communication Interface

With an increasing shift towards heterogeneous hardware/software platforms, the need for improved parallel software has been increasing correspondingly. To this end, the most popular programming paradigm supported by most multiprocessor machines has been the message passing interface (MPI) [25]. Due to various drawbacks in MPI, however, alternative protocols have been proposed, such as message passing using the parallel vector machine (PVM) [73], extensions to the C language in unified parallel C (UPC) [17], and in recent years, OpenMP [22] for shared memory architectures. How-

ever, all of these are software techniques that target general-purpose applications, and are not tuned towards the signal processing domain.

In terms of hardware, a wide variety of architectures for optimized implementation of communication networks have been explored as well. An efficient communication mechanism for MIMD (Multiple Instruction Multiple Data) processors was proposed in [26]. An architecture that integrates communication and computation in hardware is shown in [68]. A system-oriented approach as shown in [76] where communication for the Eindhoven multiprocessor system (EMPS) is presented that utilizes a truly distributed interrupt mechanism available to operating system primitives for message passing and remote procedure calls. A review of various practical hardware implementations of communication mechanisms is provided by Henry in [36]. Henry also proposes a novel mechanism that directly couples hardware with applications by bypassing intermediate operating system handling [36]. In recent times, specialized interfaces and middleware for signal processing applications have come to focus as well (e.g., see [20], [35]).

However, the above body of work does not address the need for a standardized interface that is portable over different platforms, while retaining application-domain-specific optimizations especially in the domain of signal processing that comprises of computation intensive applications. In this work, this gap is addressed systematically through a novel communication interface that integrates application-specific streamlining with a system-level approach to yield an intuitive and efficient inter-processor communication system for computation-intensive computer vision applications.

2.2.4 Parameterized Design Framework for Particle Filter Systems

Real-time implementation of particle filters presents a two-dimensional problem of efficient memory management, as well as high-speed processing. Various modifications to particle filtering techniques have been proposed to meet the above requirements [44]. Since most of the targeted applications involve extensive computation, parallelization, multiprocessor implementation of particle filters is an important option to examine. A generic system architecture for particle filters has been proposed by Bashi et al.[5]. However, this work mainly concentrates at the algorithmic level. In [67], a shared memory multiprocessor implementation of a particle-filter-based 3D facial pose tracking algorithm was developed. This implementation was shown to provide significant performance gains.

A major bottleneck in full parallelization of particle filters is resampling. Various efforts have been made on distributed resampling techniques (e.g., see [70]). However, the need to look into hardware solutions for efficient real-time implementation still exists. Architectural design and efficient memory management schemes for particle filter implementations on hardware are discussed in [3], [14] and [70]. A low-power analog particle filter implementation has been described in [74]. Mixed mode implementations — that is, partially-analog and partially-digital realizations — have also been explored. In such mixed-mode approaches, the analog components are used for the non-linear computations that are involved in particle filtering [75]. Hendeby et al. explore the use of graphics processing units to implement a parallel particle filtering system [34]. In [39], the authors provide a scheme for reconfigurable particle filtering, where two particle filtering algorithms are implemented on the same platform, and the

system can be configured to use any one of them by switching mechanisms.

The above body of work focuses mainly on implementing individual applications and optimizing various subsystems for an efficient implementation in such specialized particle filtering contexts. There is therefore a lack of a general methodology for comprehensive design space exploration of complete particle filtering systems with attention to the interactions among the various processing subsystems. Consequently, various design-space trade-offs are left unexplored. In the work presented in this thesis, a comprehensive methodology to facilitate the such design space exploration is presented.

Chapter 3 : Graph Transformation in Cyclo-Static Dataflow Representations

3.1 Introduction

Implementation of DSP applications on embedded multiprocessor systems is a multi-faceted and multiobjective optimization problem. Such systems consist of a combination of multiple programmable digital signal processors (PDSPs), application-specific hardware components and of late FPGAs. With increasing levels of integration, it is now feasible to integrate such heterogeneous systems entirely on a single chip. Such powerful capabilities along with increase in the complexities of the applications have resulted in a challenging task for the designer.

The typical design flow for such systems consists of several steps each of which can qualify as an independent problem. The focus in this chapter is on the first step — modeling and associated transformations at this step. Appropriate modeling is crucial for any efficient and accurate solution. Associated with this step are transformations that are extremely beneficial for a final optimized implementation. The nature of transformations can be of various kinds such as algorithmic or architectural or even source-to-source. The concentration in this work is on an important domain of transformations called *dataflow transformations* that arise in the context of dataflow modeling, which is a popular and widely-used modeling tool.

Of the various different modeling paradigms, dataflow is considered to be one of the most intuitive modeling paradigm for signal processing applications. Along with

modeling and specification, abundant work exists on optimized techniques for implementation of dataflow specified DSP applications on embedded systems. High-level dataflow graph transformations for efficient implementations have been explored in the past but have been limited mainly to synchronous dataflow (SDF) and its related graph representations namely single-rate (SSDF) and homogenous SDF (HSDF) graphs. However, such dataflow graphs suffer from lack of sufficient expressibility especially for systems involving dynamic interactions between different computational modules. Cyclostatic dataflow (CSDF) is one of the powerful models that maintains compile-time predictability while providing support for dynamic interaction. CSDF also is a preferred model for computer vision applications and can capture the nuances of these systems in a better fashion compared to other models supporting dynamic behavior [24].

In this chapter, transformation techniques for this important and promising dataflow model are explored. Also the first in-depth analysis of parameterized cyclostatic dataflow (PCSDF) graphs which emerged as a new powerful form of reconfigurable dataflow modeling is presented [66]. New transformation techniques for CSDF [66] as well as PCSDF graphs are presented. The techniques and analysis have been developed for the general case without any specialization for any special class of applications. To verify their properties and capabilities, the techniques were applied to an acoustic data compression system. The experiments and results demonstrate that appropriate transformations applied to high-level dataflow graphs lead to more optimized implementations.

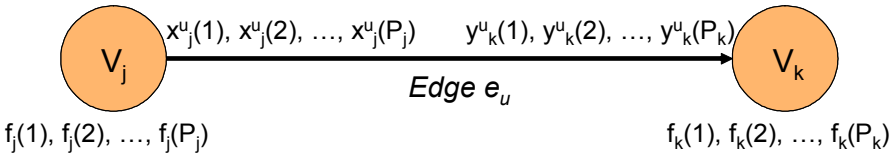

Figure 3. Example of a 2-actor CSDF graph.

3.1.1 Cyclo-static Dataflow

One of the most popular extension of SDF that allows dynamic flexibility in the graph is CSDF introduced by Bilsen et al [12]. In CSDF, token production and consumption can vary between actor firings as long as the variation forms a certain type of periodic pattern. Each time an actor is fired, a different piece of code called a phase is executed.

Formally, in CSDF, every actor V_j has an underlying execution sequence $f_j(1), f_j(2), \ldots, f_j(P_j)$ of length P_j, where $f_j(i)$ is called a phase. Effectively, the nth time actor v_j is fired, it executes the code of function $f_j((n-1) mod P_j + 1)$. As a consequence, in a CSDF graph production and consumption rates also follow periodic sequences. The amount of data (number of tokens) produced by actor v_j on edge e_u, is represented as a sequence of constant integers $[x_j^u(1), x_j^u(2), \ldots, x_j^u(P_j)]$. The nth time that actor is executed, it produces tokens $x_j^u((n-1) mod P_j + 1)$ on edge e_u. Representation of the amount of data consumed by actor v_k, is completely analogous. The firing rule of a cyclo-static actor is evaluated as "true" (i.e., the actor is enabled for execution) for its nth firing if and only if every input edge e_u to actor v_k contains at least $y_k^u((n-1) mod P_k + 1)$ tokens. In figure 3, the semantics of the cyclo-static dataflow model are illustrated.

3.1.2 Parameterized SDF

Parameterized dataflow is a metamodeling technique that allows dynamic behavior by allowing parameterization of various properties of the actors and edges — including dataflow properties — of the underlying dataflow graph. Parameterized dataflow has been studied so far primarily in the context of its application to synchronous dataflow. The resulting model of computation – i.e., the model that results from integrating parameterized dataflow metamodeling with synchronous dataflow as the "base model" – is called parameterized synchronous dataflow (PSDF).

A PSDF graph is composed of PSDF actors and PSDF edges. A PSDF actor is characterized by a set of parameters that can control the actor's functionality, as well as the actor's dataflow behavior (number of tokens consumed and produced) at its input and output ports. An application designer determines a configuration of a PSDF actor A by assigning values to the parameters of A, where each parameter is either assigned a value from an associated set or is left unspecified. Parameter values that are left unspecified in this way are assigned values at run-time, and such parameter value assignments can in general change dynamically during execution of the graph. Like a PSDF actor, a PSDF edge also has associated notions of parameterization and configuration. For practical implementations, an upper bound is specified on the number of tokens produced and consumed onto the edge, and the number of delay tokens (i.e., the number of data values that reside on the edge initially) residing on the edge, whenever these quantities are left statically unspecified.

A PSDF specification, also called a PSDF subsystem comprises of three distinct

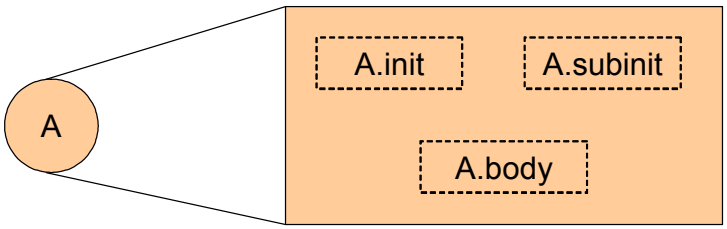

Figure 4. Example of a PSDF subsystem.

PSDF graphs: the *init* graph; the *subinit* graph and the *body* graph. Intuitively, the body graph models the main functional behavior of the specification, whereas the *init* and *subinit* graphs control the behavior of the body graph by appropriately configuring the body graph parameters. Figure 4 shows a simple PSDF subsystem. Complete details of the syntax and semantics of the parameterized dataflow modeling can be found in [6].

3.2 Parameterized CSDF (PCSDF)

As mentioned earlier, parameterized dataflow is a metamodeling technique and thus can be applied to any underlying graph that has a well-defined notion of graph iteration. In this section the special class of graphs formed by applying parameterization to CSDF graphs is investigated. Till date, most work done on parameterization of dataflow has been restricted to SDF. But CSDF offers several advantages compared to SDF such as flexibility in compactly and efficiently representing interaction between actors, decreased buffer memory requirements for some applications, and more opportunities for behavioral optimizations like constant propagation and dead code elimination. Thus, parameterization of CSDF graphs creates a new class of graphs that combines powerful optimization capabilities with strong expressibility properties.

The syntax for PCSDF graphs is similar to PSDF. Thus, a PCSDF graph consists of PCSDF actors and PCSDF edges. But unlike a PSDF actor, a PCSDF actor's functionality is not parameterized directly. Its functionality varies cyclically and it is this cyclic pattern that is parameterized. This applies to the actor's dataflow behavior i.e., production and consumption patterns as well, which vary cyclically and may be parameterized. There are two fundamental parameters in a PCSDF actor's dataflow properties: the period of the cycle and the data rates. Dynamic behavior of the graph can be modeled by allowing parameterization of either of these two or their combination. Thus, it is possible to have a behavior in which the period is fixed for a single iteration in which the data rates vary over periods, while the periods themselves vary over iterations.

The semantics for PCSDF graph is also similar to a PSDF graph, and hence a PCSDF specification is done using a PCSDF subsystem comprising of *init*, *subinit* and *body* graphs. An example of a PCSDF subsystem is shown in figure 5.

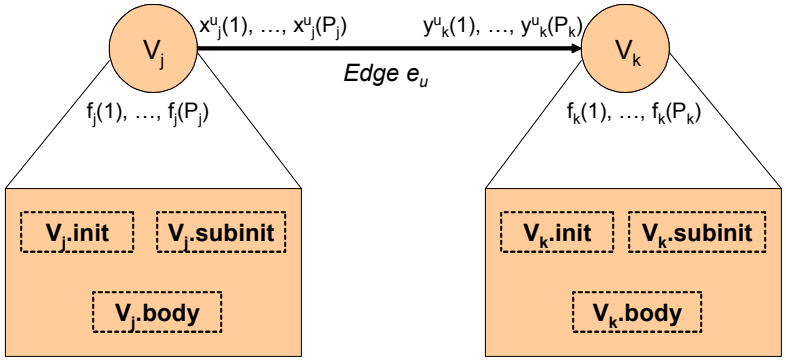

Figure 5. Example of a PCSDF system.

3.3 Dataflow Clustering Transformations

The first step towards implementing a dataflow-based system is construction of a *schedule*. Dataflow transformations may be applied before creating a schedule, as well as during the process of constructing a schedule. Of the various possible scheduling strategies the focus here is on *looped schedules* [8] and *parameterized looped schedules* [7], [43], which construct schedules in terms of static and dynamic looping constructs, respectively. These types of schedules combine the advantages of efficient looping facilities in programmable digital signal processors [46], low complexity storage and manipulation of schedule information [43], [54], and potential for extensive analysis and optimization [54].

Dataflow graph transformation is an effective technique to produce high-performance DSP software as well as hardware/software solutions. The main objective of such techniques is to create from a given dataflow graph, a functionally equivalent dataflow-graph with improved characteristics with regards to modeling and/or implementation. As discussed by Zivojnovic et al [78], there are mainly five different kind of transformations that can be applied to dataflow graphs: *retiming*, *unfolding*, *vectorization*, *clustering* and *node/edge-set extension/reduction*. Of the above five forms of transformations, *clustering* is especially important in the derivation of schedules from high-level dataflow representations and is the main focus in this work.

Clustering transformation can be of various types for example task graph clustering [69], clustering in dataflow graphs [10] and etc. The general definition of clustering is “a transformation in which a set of nodes is combined into a single node”. Task graph clustering refers to grouping of basic tasks into subsets that are to be executed

on the same processor in a multiprocessor environment and is typically the first step in scheduling. A clustering transformation for a dataflow graph is one that replaces a set of multiple actors in the graph with a single actor, thereby constraining subsequent synthesis steps to view the chosen set of actors as a single "unit" for some relevant purpose. When clustering a set A of actors, characterizations for production and consumption rates of edges at the interface of a cluster are derived based on how much data is transferred with respect to the interfaces in one execution of the subsystem input/output port functionality associated with A. A formal development of clustering for SDF graphs is provided in [62].

In this section a new form of clustering called "phase clustering" in the context of CSDF and PCSDF graphs is introduced. The rest of this section is organized as follows. First clustering techniques for SDF graphs are discussed, followed by that of PSDF and then introduce "phase clustering" for CSDF and PCSDF graphs.

3.3.1 Clustering in SDF and PSDF graphs

An important clustering algorithm for SDF graphs is the APGAN (Acyclic Pairwise Grouping of Adjacent Nodes) [10] algorithm in which hierarchies of clusters are constructed in a bottom-up fashion by repeatedly selecting and clustering adjacent pairs of actors. The selection process is performed in a heuristic fashion based on clustering cost functions that are geared towards the key implementation objectives. Scheduling can then be performed on the clustered actors. The focus here is on single appearance schedule (SAS) only, in particular the APGAN approach.

Since in a PSDF graph, the production and consumption rates of actors are variable and generally not known at compile time, it is neither possible to construct static

schedules, nor develop exact characterizations of dataflow behavior at cluster interfaces. However, clustering is an effective tool for transforming PSDF graphs in situations where the characteristics of clusters can be expressed in terms of subsystem parameters. Fortunately, through careful analysis, this can be done for many practical PSDF systems.

One such example is P-APGAN for acyclic graphs formed by extending APGAN. In a PSDF graph, it is not possible to select an adjacent pair of actors based on the repetition count as done in APGAN for SDF, since the repetition count is symbolic. Thus, a heuristic is used in which preference is given to edges for which gcd (greatest common divisor) is known which, in the order of decreased importance, are as follows:

1) single-rate edges with $p(e) = c(e)$.

2) SDF edges within the graph so that their $p(e)$ and $c(e)$ values are known at compile time.

3) edges whose gcd may be known by user assertions.

For each clustered subgraph, a looped schedule is generated symbolically in terms of the dynamic parameters associated with the subgraph. This allows the subgraph schedule to adapt at run-time in correspondence with changes in the parameter values. After the cluster hierarchy is constructed, a schedule for the overall PSDF graph is derived by recursive traversal of the cluster hierarchy and subsets of the schedules associated with each clustered subgraph [18].

3.3.2 Clustering in CSDF graphs and PCSDF graphs

In this section, a new *multirate* clustering transformation for CSDF graphs is introduced in detail. Multirate clustering is useful for decomposing complex, high-

level dataflow designs for scheduling especially for multiprocessor systems and was first introduced in the context of SDF graphs [62]. However, multirate clustering has been mainly studied in the context of SDF graphs and in this section a framework to extend multirate clustering to CSDF graphs is presented.

We define the *data rate signature (DRS)* for an edge to be the production/consumption rate tuple $x_j^u(1), x_j^u(2), \ldots, x_j^u(X_j)$ for an edge e^u of a CSDF actor A_j with X_j phases. Each such DRS is broken down into repeating sub-regions or sub-DRSs $s_j^u(1), s_j^u(2), \ldots, s_j^u(m)$ such that each $s_j^u(i)$ covers exactly k tokens. The new DRS resulting from the combination of the sub-DRSs is denoted by DRS_{phased}. Thus, DRS_{phased} is a DRS that generally consists of fewer phases compared to the original DRS. Furthermore, the data rate value associated with every phase is a constant k, which is called the $subDRSrate$. This grouping of phases can be viewed as imposing an SDF abstraction of finest possible granularity on top of the underlying CSDF input/output port.

Consider for example the edge e shown in figure 6 with DRS $(1, 0, 1, 2, 0, 1, 1, 0, 2)$. This can be decomposed into (s_1, s_2, s_3, s_4) where $s_1 = (1, 0, 1)$, $s_2 = (2, 0)$, $s_3 = (1, 1, 0)$ and $s_4 = (2)$. Note that such a decomposition is not unique because any trailing 0 of an intermediate s_i can be moved to the beginning of s_{i+1}. For

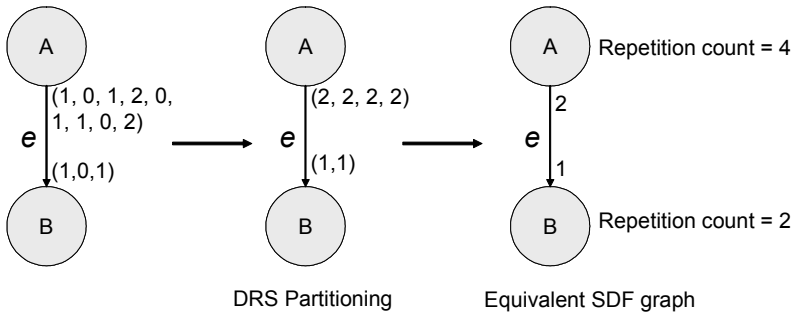

Figure 6. Example of DRS partitioning for a 2-actor CSDF graph.

conciseness, the restriction that trailing 0s with respect to any intermediate s_i are always included in s_i itself is imposed. Exploring the broader design space allowed by more flexible placement of such trailing 0s is a topic for further study.

As implied above, such a decomposition effectively creates an SDF abstraction of the input/output port that the DRS is associated with, but at a finer granularity compared to the trivial SDF abstraction that results from collapsing all the phases into one monolithic actor invocation. However, in general it is not always possible to find a valid decomposition that is of finer granularity than the trivial one — in these cases, the decomposition technique here degenerates to the trivial SDF abstraction. For the case of binary CSDF [23](a binary CSDF graph has only 1s and 0s in the DRSs), it is generally always possible to obtain finer granularity decompositions compared to the trivial form (that is, whenever the sum of token rates in a DRS exceeds 1). This general process of creating sub-DRSs is called "DRS Partitioning" (DRSP).

Once a valid DRSP of the graph is formed, a resulting SDF graph is obtained, that is, a graph with constant, scalar production and consumption rates (figure 6). Thus, now various SDF-oriented strategies for clustering and scheduling, such as APGAN, can be considered. However, after scheduling, a reverse decomposition of production and consumption rates needs to be performed for actors that have undergone DRSP. This reverse decomposition process is denoted as *reverse* DRSP (RDRSP). Based on these concepts, a top level transformation for CSDF graphs can be expressed algorithmically (algorithm 1).

A simple algorithm for DRSP for a single DRS is presented in algorithm 2. In the formulation of the algorithm, the following two observations are used:

Algorithm 1. DRSP transformation algorithm.

DRSP transformation (Φ_G)
$\Phi_{DRSP} \leftarrow$ DRSP (Φ_G)
Schedule $S_{DRS} \leftarrow$ APGAN (Φ_{DRSP})
Schedule S $\leftarrow$ RDRSP (S_{DRS})

- The $subDRSrate$ for a valid sub-DRS partitioning of a DRS S cannot be less than the maximum production/consumption rate in S.
- The $subDRSrate$ for a valid sub-DRS partitioning of a DRS S is a factor of the sum of the production/consumption rates of S.

Let the original DRS be stored in an array S and have a length N. The algorithm takes S as input, examines whether DRSP can be performed, and if so returns DRS_{phased}. The algorithm looks as follows:

Algorithm 2. DRS Algorithm.

```
DRSP SINGLE (DRS S,N)
max ← findmax(S,N)
sum ← findsum(S,N)
(factors,m) ← factors(sum)
for i ← 0 to m {
if (factors(i) >= max) {
      start ← factors(i)
      break
     }
}
for f ← start to factors(m) {
     (DRSphased, phases) ← partition(S, f)
     if (DRSphased ! = NULL)
           break
}
return (DRSphased)
```

In this pseudocode block, the function $findmax(A, n)$ finds the maximum element of an array A of length n; $findsum(A, n)$ finds the sum of the elements of an array A of length n; and $factors(M)$ returns an array containing the factors of M. The function

$partition(A, s)$ takes as input an array A; determines whether it is possible to split the array into sub-sets, each of whose sum of elements is s; and if so, returns an array containing $phases$ elements, each having value s. Note that $phases$ can be 1, denoting the trivial SDF abstraction. If such a split of the input array cannot be performed, $partition(A, s)$ returns an empty array. The complexity of this algorithm is dictated by the *for* loop, which attempts to find a valid partition. The complexity of the computation performed by this loop is $O(m)$ for m factors. The complexity is also dictated by the function $partition(A, s)$ which is $O(n)$, where n is length of A. The overall complexity is therefore $O(mn)$.

The PCSDF graph transformation builds on the CSDF graph transformation. The clustering is done by creating a static CSDF once the parameter values are known at run-time. A parameterized transformation similar to P-APGAN would be a useful and better approach to transforming PCSDF graph and is an important direction for future work

3.4 Experiments

To demonstrate the capabilities of the transformation techniques discussed in the sections above, the methodologies were applied to a acoustic data compression algorithm. The application inherently involves dynamic production and consumption rates. However, it may be forced to behave statically in exchange of higher design flexibility. In this section the experiments of modeling and clustering transformations using SDF, PSDF and PCSDF and the related trade-offs are presented.

3.4.1 Acoustic Data Compression (ADC)

The basic components of the LPC consist of framing, predictor coefficient and coding. For the process of framing, signals generated for a particular duration of time contain size L samples, and such L sample signals are divided into frames of size N. Each frame is subjected to the linear predictor which generates the predictor coefficients. These coefficients are used to determine the predicted value of the sample signal. The error e is determined and the error e and the predictor coefficients for each frame N are encoded using a coding algorithm.

Mathematically a linear predictor can be represented as follows. $s[n]$ represents the input sample data. $x[n]$ represents the data for a frame. $a[k]$ represents the predictor coefficients of a frame. The predictor is modeled as an Autoregressive (AR) process. The relation between the predictor coefficients and the input data is given by the AR model by the following equation of the input data

$$Predicted(i) = \sum_{i=1}^{k} a_i \times x_{n-i} \tag{3.2}$$

The prediction error is given as

$$Error(i) = x[i] - predicted(i) \tag{3.3}$$

3.4.1.1 SDF modeling and transformation of ADC

Given a sample input data $s[n]$ of size L the framing computes the optimum frame size N of the input data. Each frame is then passed to a linear predictor which computes the Acoustic Data Compression (ADC) predictor coefficients a. The size of

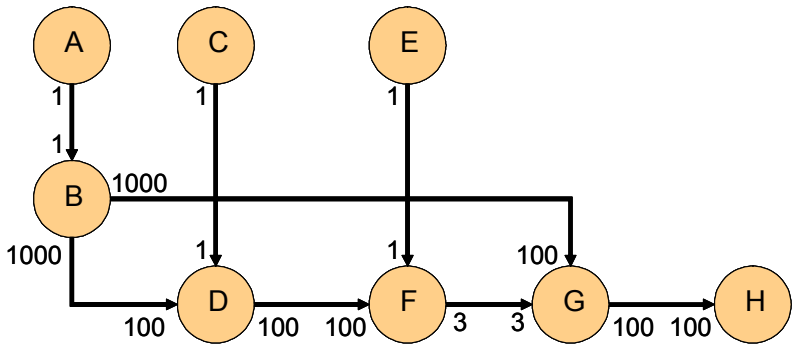

Figure 7. SDF model of ADC system.

the prediction coefficients depends on the order M. In SDF model the frame size N and model order M of the predictor is known beforehand. The detailed block diagram of SDF model is given in figure 7.

In this figure L denotes the size of the input data sample and N denotes the frame size. L and N are fixed at compile time. Based on the size of L and N a valid schedule can be generated. The computational blocks are represented by actors A to H. A detailed description of the actors in this graph is as follows

- Actor Buffer Size (A); This actor sets the sample size L of the input data
- Actor Input Data (B). This actor reads a file and stores the data in a buffer
- Actor FFT Size (C). This actor calculates the size of FFT based on the frame size of the samples
- Actor FFT (D). This actor implements Fast Fourier Transform on the input samples.
- Actor model (E). This actor determines the fixed model order for the frames.
- Actor LU Decompose (F).This actor implements the LU Decompose algorithm for determining the predictor coefficients.
- Actor arError (E). This actor generates the error on the samples using the AR

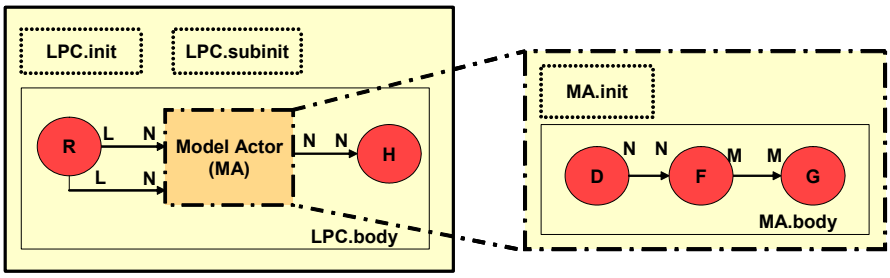

Figure 8. PSDF modeling of ADC system.

model.

- Actor QxE (*H*). This actor implements the Huffman Coding on the error samples.

The clustering/transformation and scheduling for this SDF graph was based on the APGAN strategy and for $L = 100$ and $N = 10$ generated the schedule given by $AB(10CDEFGH)$. As may be observed, the schedule is a looped schedule and also a *SAS*.

3.4.1.2 PSDF modeling and transformation of ADC

A block diagram of the PSDF model is given in figure 8. In this, LPC.init sets the segment size L while LPC.subinit sets the frame size N. LPC.body implements the body graph — i.e., the core, parameterized computation of the compression system. Actor *R* reads a segment of input data and stores it in a buffer while Model Actor (MA) is a hierarchical actor that implements the model order for each frame. More specifically, *MA.init* sets the model order for each frame and *MA.body* is an associated body graph that contains actors *D, F, G.* Actors *D, F, G and H* have the same functionality as described in the SDF-based model.

The P-APGAN scheduling strategy [7] was applied manually to derive the schedule the for the above PSDF model of the LPC system and thus did not involve any call

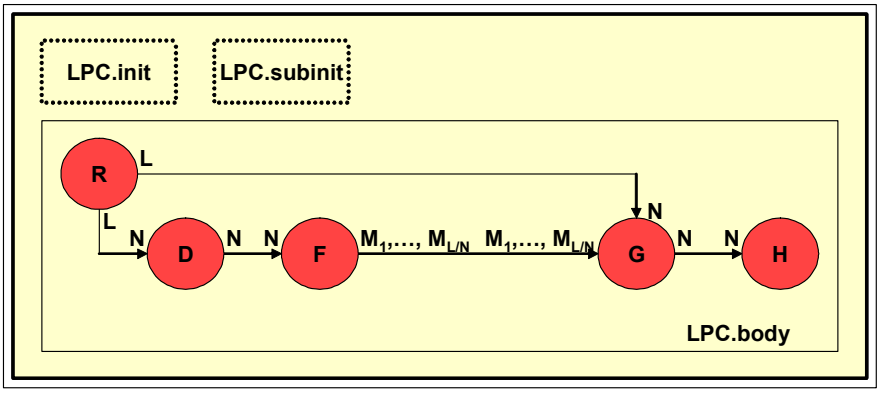

Figure 9. PCSDF modeling of ADC system.

to APGAN as in the DRSP transformation for CSDF. The resulting schedule is $R((L/N)MA)H$, Note that this is a parameterized schedule with the iteration count quotient L/N varying dynamically as the application executes. As the underlying parameter values L and N change at run-time, the parameterized schedule naturally adjusts the actor execution sequence to be consistent with the new parameter values.

3.4.1.3 PCSDF modeling and transformation of ADC

The overall block diagram of PCSDF model is given in figure 9. A detailed description of the various actors in the PCSDF model is as follows:

- *LPC.init*: This actor sets the samples size L.
- *LPC.subinit*: This sets the frame size N and the model order M_1 to $M_{L/N}$ for each frame.
- *LPC.body*: This actor implements the body graph which has actors *RI* explained below and *F*, *D*, *G* and *H* explained in the SDF model.
- Actor *RI*: This actor reads the input data from a file

The body graph models the main behavior of the subsystem. The *lpc.init* and *lpc.subint* graph controls the parameters for the *lpc.body* graph. The *lpc.init* graph ini-

tializes the sample size L. The *lpc.init* graph calls the *lpc.subinit* graph to determine the frame size N. The sample size and frame size are passed as a parameter to the body graph.

This model, based on reconfigurable data flow graphs, naturally accommodates adaptive applications such as the framework of energy-adaptive compression used here and allows variable amounts of data to be produced and consumed by functional blocks.

The transformation of the PCSDF graph was done by creating a CSDF version (on which the DRSP transformation was applied) after reading in the parameters L and N with values 100 and 36 respectively and is given by $RI(2DFGH)$.

3.5 Results

In this section comparisons of the various performance indices are presented to study the trade-offs involved in the different modeling and transformation techniques which include code size, data size and cpu execution cycles. The system was implemented using the simulation environment provided Texas Instrument's Code Composer Studio (version 2). The target PDSP platform was the TMSC320c64xx processor with instruction cycle time of 40ns.

Figure 10 shows the comparison between the code size and data size for the implementation of the ADC system using SDF, PSDF and the PCSDF implementations.

As observed from the graph, the SDF implementation yields the least code and

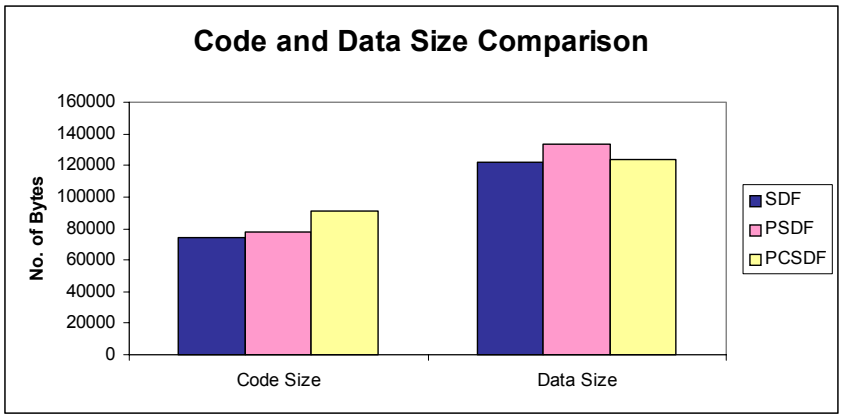

Figure 10. Code size comparison for ADC system.

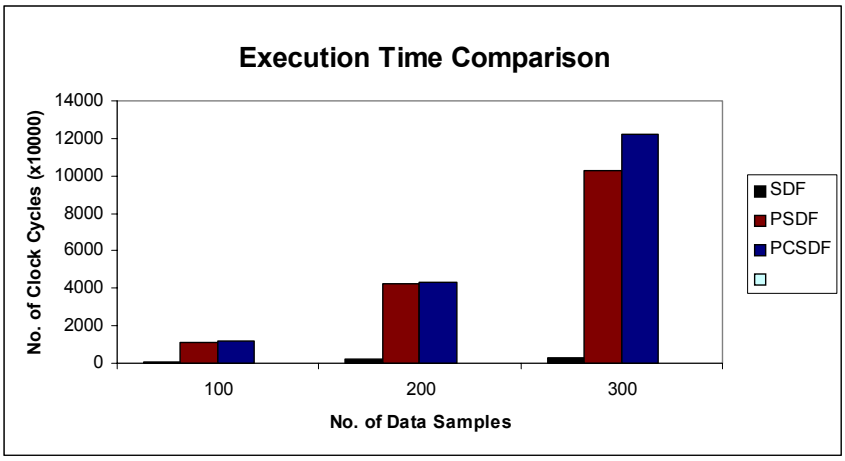

Figure 11. Comparison of execution time for different data sample size for ADC system.

data size. However, this is at the cost of reduced expressibility and lack of dynamic behavior in the system. Allowing dynamic reconfigurability leads to run-time overheads as shown by the PSDF and the PCSDF implementations. However, the PCSDF implementation results in significantly less data size compared to the PSDF system. The PSDF implementation yields a better implementation since the run-time overhead of scheduling is much less compared to that required in the PCSDF implementation that involves DRS partitioning.

Figure 11 shows the comparison of CPU execution cycles for the different imple-

mentations. The higher execution times for the PSDF and PCSDF models compared to SDF is due to the run-time overhead of the scheduling strategies, which is not present for the SDF model. The PCSDF and PSDF systems show comparable performance despite the overhead the PCSDF system incurs due to run-time invocation of APGAN. This is due to a two-fold effect: the PSDF-based system incurs overheads due to repeated invocation of the init graph of the *MA* actor, and the quasi-static schedule for PSDF involves 2 for loops compared to the schedule for the PCSDF system that has only 1 for (all of which depend on the number of samples L). However, for a very high value of L, the dynamic scheduler execution time overshadows this gain and the overall execution time is higher than that of the PSDF system.

It may be observed from these results, that PCSDF yields a much more efficient implementation compared to PSDF while allowing dynamic reconfigurability which demonstrates the capabilities of PCSDF and its associated transformation technique as a powerful modeling tool.

Chapter 4 : Architectural Design Space Exploration

4.1 Introduction

The number of platforms available for the implementation of computer vision applications is vast and varied and the applications are characterized by multifaceted functionalities along with a multi-dimensional optimization space comprising of performance, power consumption, memory size and also area associated with the implementation resulting in an immense and complex design space of which designers are typically able to evaluate only small subsets of architectural solutions, partitionings, and mappings of the system functionalities. A comprehensive design space exploration will enable designers to select higher quality solutions and provide substantial savings on the overall cost of the system.

An architectural level design methodology that provides means for such comprehensive design space exploration along with models for performance estimation is described in this chapter. The methodology differs from the existing ones in its simple and intuitive approach — it exploits the concept of synchronization between processors, a function which is essential in any implementation, with simple extensions. In other words, instead of building a separate formal method a functionality, which is inevitable in such design and implementation, is reused for exploration purposes. Its efficiency and accuracy in how a wide range of design options can be selected that trade off various architectural features is demonstrated by experimental results from two significant test applications.

Other than the contribution stated above, this model also makes useful contributions in the following ways:

- It demonstrates the first formulation and use of multirate synchronization graphs, which is shown to be useful for compactly representing repetitive patterns in the execution of a multiprocessor image processing system.
- It integrates aspects of ordered transaction execution with more conventional, self-timed execution in a flexible and seamless way to demonstrate a new class of hybrid self-timed/ ordered transaction designs. It also demonstrates how the synchronization graph modeling methodology can be used to unify analysis across the entire spectrum of systems encompassing pure self-timed execution, pure ordered transaction execution, and the set of hybrid self-timed/ordered transaction possibilities that exist between these extremes.
- It shows how our multirate synchronization graph approach, together with hybrid self-timed/ordered transaction scheduling, can be used to effectively design systems involving extensive multi-dimensional processing where previous development of synchronization graph modeling has focused primarily on single-dimensional, signal processing systems.

This chapter is organized as follows. First, a brief overview of the key concepts behind the methodology, i.e., self-timed execution and ordered transaction strategy for dataflow graphs and multirate synchronization graph, is given. Next, the applications and target architecture used for experiments are briefly described. The new methodology is then described in the context of one of these applications. Finally, the experimental results are given. For both the applications used as benchmarks, the system

architecture consists of an array of special processing elements along with special interfaces for I/O and/or external memory systems.

4.2 Modeling Approach Overview

Mapping an SDF-based application to a multiprocessor architecture includes i) assignment of actors to processors, ii) ordering the actors that are assigned to each processor, and iii) determining precisely when each actor should commence execution. For this purpose the focus is on the self-timed scheduling strategy and the closely-related ordered transaction strategy [71].

In self-timed scheduling, each processor executes the tasks assigned to it in a fixed order that is specified at compile time. Before executing an actor, a processor waits for the data needed by that actor to become available. Thus, processors are required to perform run-time synchronization when they communicate data. This provides robustness when the execution times of tasks are not known precisely or when they may exhibit occasional deviations from their compile-time estimates and it also eliminates the need for global clocks.

The ordered transaction method is similar to the self-timed method, but it also adds the constraint that a global, linear ordering of the interprocessor communication operations (communication actors) is determined at compile time, and enforced at run-time. The linear ordering imposed is called the transaction order of the associated multiprocessor implementation. Enforcing of the transaction order eliminates the need for run-time synchronization and bus arbitration, and also enhances predictability.

The synchronization graph G_s [71] is used to model the self-timed execution

of a given parallel schedule for an iterative dataflow graph. Given a self-timed multiprocessor schedule for graph G, G_s can be derived by instantiating a vertex for each task, connecting an edge from each task to the task that succeeds it on the same processor, and adding an edge that has unit delay from the last task on each processor to the first task on the same processor. Each edge (v_j, v_i) in G_s is called a *synchronization edge*, and represents the synchronization constraint:

$$\forall k, start(v_i, k) \geq end(v_j, k - delay(v_j, v_i)), \tag{4.1}$$

where $start(v_j, k)$ and $end(v_j, k)$, respectively, represent the time at which invocation k of actor v_j begins execution and completes execution, and $delay(e)$ represents the delay associated with edge e.

Once G_S for a system is constructed, the *maximum cycle mean* (*MCM*) of the graph is used for performance analysis. The MCM is defined by,

$$MCM(G_s) = \max_{CycleCinG_s} \left\{ \frac{\sum_{v \in C} t(v)}{Delay(C)} \right\} . \tag{4.2}$$

where $Delay(C)$ denotes the sum of the edge delays over all edges in cycle C.

4.3 Application Overview

4.3.1 Face Detection based on Shape Operator

There are several approaches that make use of shape and/or intensity distribu-

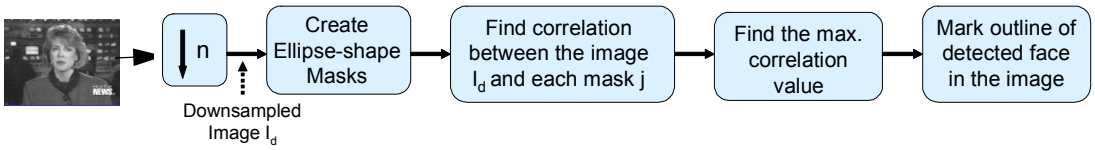

Figure 12. The flow of face detection algorithm.

tion on the face. A shape-based approach as proposed by Moon et al. [52] is used. A face is assumed to be an ellipse. This method models the cross-section of the shape (ellipse) boundary as a step function. The operator for detecting faces is derived by extending the DODE (double exponential) filter along the boundary of the ellipse. The probability of the presence of a face at a given position is estimated by accumulating the filter responses at the centre of the ellipse.

At the implementation level, this reduces to finding out correlations between a set of ellipse shaped masks with the image in which a face is to be detected. Figure 12 shows the complete flow of the employed face detection algorithm.

4.3.2 3D Facial Pose Tracking in Video

The aim in facial pose tracking is to recover the 3D configuration of a face in each frame of a video. The 3D configuration consists of 3 translation parameters and 3 orientation parameters that correspond to the yaw, pitch and roll of the face. The 3D tracking algorithm considered uses the particle filtering technique along with geometric modeling [2]. The complete algorithmic flow is given in figure 13.

There are three main aspects that capture the 3D tracking system. The first is the model to represent the facial structure. The second is the feature vector used. The third is the tracking framework used. A model attempts to approximate the shape of the object to be tracked in the video. In this application, a cylinder with an elliptical

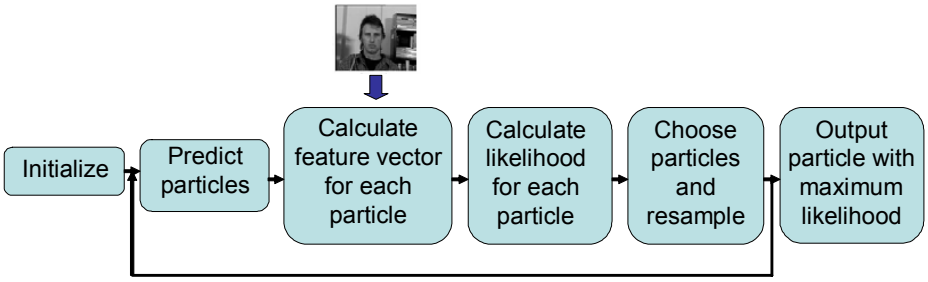

Figure 13. The flow of 3D facial pose tracking algorithm.

cross-section is chosen as a model to represent the 3D structure of face.

The feature vector represents characteristics from the image. A rectangular grid superimposed around the curved surface of the elliptical cylinder is used for this purpose: the mean intensity for each of the visible grids/cells forms the feature vector. Given the current configuration, the grids can be projected onto the image frame and the mean can be computed for each of them.

For the tracking framework, i.e., estimating the configuration or pose of the moving face in each frame of a given video, a particle filter based technique is used. As mentioned before, the motion of the face is characterized by 3 translation and 3 orientation parameters. For each new image frame read in from the camera, multiple predictions for these parameters are made where each prediction is a particle. The feature vector is then extracted for each particle. The particle that yields the best likelihood value gives the position of the face in the frame. The number of particles to be used in the system is decided by the user and is constant for one application. For updating the particles for the next frame, a small number of particles from the current frame are chosen based on their likelihood values and re-sampled.

4.4 Target Architecture Overview

The face detection system was implemented on a reconfigurable system on chip, while the 3D object tracking system was implemented on a multiprocessor system and a PDSP-based system. The reconfigurable system on chip considered is Xilinx's ML310 board, which contains a Virtex II Pro FPGA device. This board supports both hardware (FPGA-based) and software (PowerPC-based) design. It also includes on-chip and off-chip memory resources. The multiprocessor system considered is a shared memory system, specifically the Sunfire 6800 containing 24 SUN UltraSparc III machines running at 750 MHz and using 72GB of RAM was used. The PDSP system considered is the TMS320C64xx series from Texas Instruments.

4.5 Design Space Exploration for Face Detection System

Figure 14 shows the implementation, transaction/execution order and synchronization model of the system. The various actors in this figure are explained below:

- $MR_{i,j}$ (Mask Read): This actor represents the reading of mask i to processing element j, where i varies from 1 to m and j varies from 1 to n. This process takes t_{MR} time units.
- *MTC* (Mask Transfer Controller): The masks are stored in the external memory, the MTC controls the reading to the dedicated block RAMs (BRAMs) for each PE (processing element). The MTC conducts mask transfers one-at-a-time according to a repeating, pre-determined sequence in a fashion analogous to that of the ordered trans-

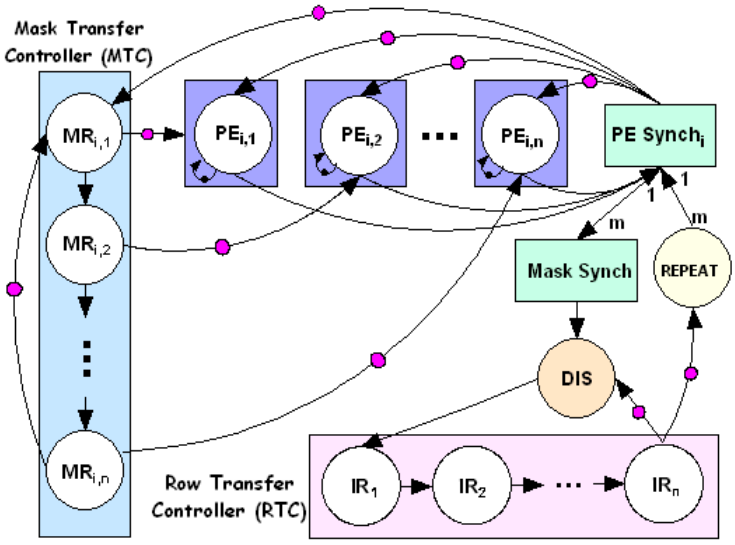

Figure 14. Multirate synchronization graph for face detection system.

action strategy.

- $PE_{i,j}$ (Processing Element): This actor represents the processing of the i th mask by PE_j, which takes t_{PE} time units.
- *DIS* (Downsampled Image Source): This actor represents the downsampling of an image stripe whose execution time is t_{DIS} time units.
- *IR* (Image Read): This actor represents the reading of the downsampled stripe into the BRAMs one row at a time with execution time t_{IR} units.
- *REPEAT*: This actor is a conceptual actor that ensures that exactly m mask sets are processed for each new row of image data. No data actually needs to be replicated by the *REPEAT* actor; the required functionality can be achieved through simple, low-overhead synchronization and buffer management methods.
- $PESynch_i$: This actor represents the synchronization unit that synchronizes the start of the i th iteration of the PEs with the reading of mask set. This unit receives data from the PEs and the *REPEAT* actor. The messages from the *PE*s confirm that they have completed the processing of one mask. These messages are sent at the end of

each iteration. The production rate of m and consumption rate of 1 (shown on the (*REPEAT*, $PESynch_i$) edge) indicate that the *PEsynch* unit has to execute m times before the REPEAT actor is invoked again.

- *Mask Synch*: This actor is again a conceptual actor, and need not be mapped directly to hardware. It represents the synchronization between m executions of the *PE Synch* actor and the corresponding execution of the *DIS* actor. Therefore, for each execution of the *DIS* actor, the *PE Synch* executes m times.

Unlike conventional ordered transaction implementation, however, a transaction ordering approach is not used in this design for all dataflow communications in the enclosing system. A self-timed model is used to coordinate interaction between the *MTC* and the *PE* cluster. Before reading a new set of masks, the controller must synchronize with the PEs to make sure that they have finished processing of the current masks. This synchronization process is represented as the edge directed from the *PE Synch* actor to the starting actor of the *MTC* block. The edge delay connecting $MR_{i,n}$ to $MR_{i,1}$ represents the initially-available mask data from pre-loading the first set of masks for a new image to their associated BRAMs.

A properly-constructed ("consistent"), multirate SDF graph unfolds unambiguously into a homogeneous SDF (HSDF) graph [71], which in general leads to an expansion of the dataflow representation. An example of an SDF-to-HSDF transformation is given in figure 15 for $m = 3$ and $n = 2$. For performance analysis, it is necessary to reason in terms of directed cycles in the HSDF representation. Table 1 tabulates several of the different classes of cycles, along with a description and the *MCM* for each class.

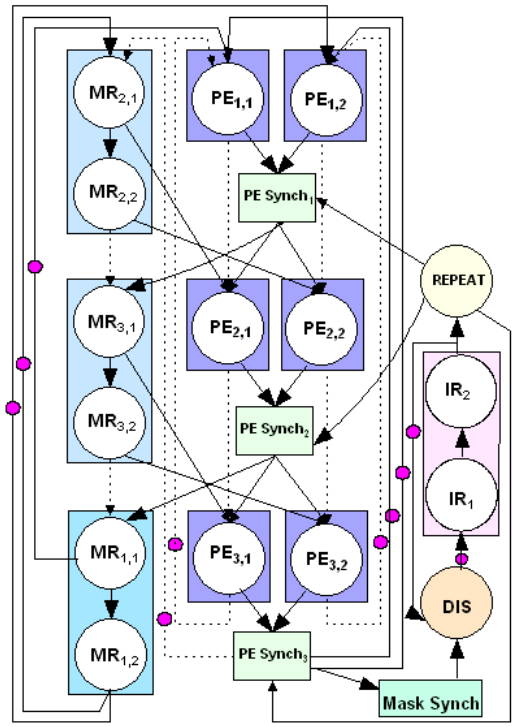

Figure 15. Example of an unfolded HSDF graph for face detection system. Here m = 3,n = 2.

As can be seen from table 1, the overall system performance is a function of m, n, t_{PE}, t_{MR} and t_{DIS}. The number of masks and the mask sizes were fixed; the face detection algorithm is shape-based and hence the face is modeled as an ellipse. To handle variability in size of the faces several elliptical masks of varying sizes were created and used which resulted in a large mask set and a large number of correlation computations. To reduce this mask set the following valid assumptions were made. The number of masks required is bounded by the possible ellipticity of faces and by the size of the image. The target application was a smart-camera based vision system which imposed constraint on the size of the images as images shot by a given camera can be assumed to be fixed.

Given the above assumptions m and t_{MR} were constant. t_{PE} is a function of several parameters but only the following parameters were considered: (a) degree of

Table 1. Cycle mean expressions for the multirate synchronization graph for the face detection system.

No	Description of Class of cycle	Cycle	Cycle Mean Expression
1	Reading of masks	$MR_{2,1} \rightarrow MR_{2,2} \ldots \rightarrow MR_{2,n} \rightarrow MR_{3,1} \ldots \rightarrow MR_{1,1} \ldots \rightarrow MR_{1,n} \rightarrow MR_{2,1}$	$\frac{m \times n \times t(MR)}{D_0}$
2	Reading of image rows	$DIS \rightarrow IR_i \rightarrow DIS$	$\frac{t(DIS)+n \times t(IR)}{D_1 + D_2}$
3	Reading of image rows with synchronizations	$DIS \rightarrow IR_i \rightarrow REPEAT \rightarrow PESynch \rightarrow MaskSynch \rightarrow DIS$	$\frac{t(DIS)+n \times t(IR)+3}{D_1}$
4	PEs synchronization	$PESynch_m \rightarrow PE_{1,i} \rightarrow PE_{2,i} \ldots \rightarrow PE_{m,i} \rightarrow PESynch_m$ or $PESynch_m \rightarrow PE_{1,i} \rightarrow PESynch_1 \rightarrow PE_{2,i} \ldots \rightarrow PESynch_m$	$\frac{m \times (t(PE)+1)}{D_1}$
5	Synchronization of PEs and reading of masks	$PESynch_m \rightarrow MR_{2,1} \ldots \rightarrow MR_{3,1} \ldots \rightarrow MR_{3,n} \ldots \rightarrow MR_{1,1} \rightarrow \ldots MR_{1,n} \rightarrow PE_{1,i} \rightarrow PE_{2,i} \ldots \rightarrow PESynch_m$ or $PESynch_m \rightarrow MR_{2,1} \ldots \rightarrow MR_{3,1}\ MR_{3,n} \ldots \rightarrow MR_{1,1} \rightarrow \ldots\ MR_{1,n} \rightarrow PE_{1,i} \rightarrow PESynch_1 \rightarrow PE_{2,i} \ldots \rightarrow PESynch_m$	$\frac{m \times n \times t(MR)+m \times (t(PE)+1)}{D_3+D_4}$

fine grain parallelism (i.e., how many simultaneous operations can be performed within each PE), (b) image size, and (c) the resolution (number of rows/columns considered) at which the image is compared with the mask. t_{DIS} is a function of the frame sizes. To keep the design space manageable, the frame size was fixed and handled as stripes (frame size = 240×320, stripe size = 65×160), and the number of PEs (i.e., n), the steps - the granularity at which the image is correlated with a mask, and the fine-grain parallelism were varied up to the permissible HW limits.

The execution times were dictated/obtained by multiplying the number of execution cycles for each node by the inverse of the clock frequency, which is 125 MHz for the given board. The delays were given by the number of cycles required for the initial values to load from the source actor to the destination actor in the synchronization graph. From this, the cycle means for the stated classes of cycles were obtained, which yielded the throughput as the minimum inverse of the cycle mean over all possi-

ble cycles,

$$Throughput = \frac{1}{MCM} \ . \tag{4.3}$$

There were restrictions imposed by the target platform as well. The number of PEs that may be implemented was limited by the area constraints. The board has 136 BRAMs present of which few were allocated for the use of other modules such as the down-sampling unit, the PowerPC, and so on. Each PE required 8 BRAMs, which set an upper bound of number of PEs to 15. Also, parallelization was possible within each PE: since the multiplications required for the calculation of each correlation value are independent of each other, more than one multiplication could be performed at the same instant. The number of multipliers available on the board limited this parallelization: there were 136 multipliers on board, which limited the number of PEs to 13. The I/O buffers present on the board also imposed serious restrictions on the number of PEs. Since each PE communicates with the down-sampling unit, the BRAMs, the external DDR SDRAM memory controller, and the output interface, it has a significant number of I/O ports — this limited the maximum number of PEs that could be practically implemented to 6. The execution times based on MCM expressions were obtained as shown in table 2 with parameters bounded by the above analysis.

Table 2. Execution times for 1 frame for different design parameters for face detection system.

n	degree of parallelism	Estimation(ms)		Experimental(ms)	
		steps = 2	steps = 4	steps = 2	steps = 4
6	0	451	136	697	205
1	20	168	62	227	79
2	10	170	62	227	79
3	6	180	65	249	85
4	5	176	63	227	79
5	4	173	63	227	79
6	3	184	66	249	85

4.6 Experimental Results

4.6.1 Face Detection System

The results are presented in Table 2. Here, n is the number of PEs, "degree of parallelism" is the number of additional multiplications done simultaneously in each PE, and "steps" is the granularity at which masks are correlated with the image. There are differences in the estimated and experimental values in the table. In our model, for simplicity a small constant value for all synchronization actors was assumed, which is a major reason for the difference. The area-performance trade-off curve is shown in figure 16, where area is quantified by the number of PEs and performance by the execution times.

The maximum frame rate of applications in security and video surveillance toward which this work is targeted is 30 frames per second (fps). With the current board and available hardware resources, the implementation achieves a maximum frame rate of 12 fps. This entails the discarding of every two out of three frames at

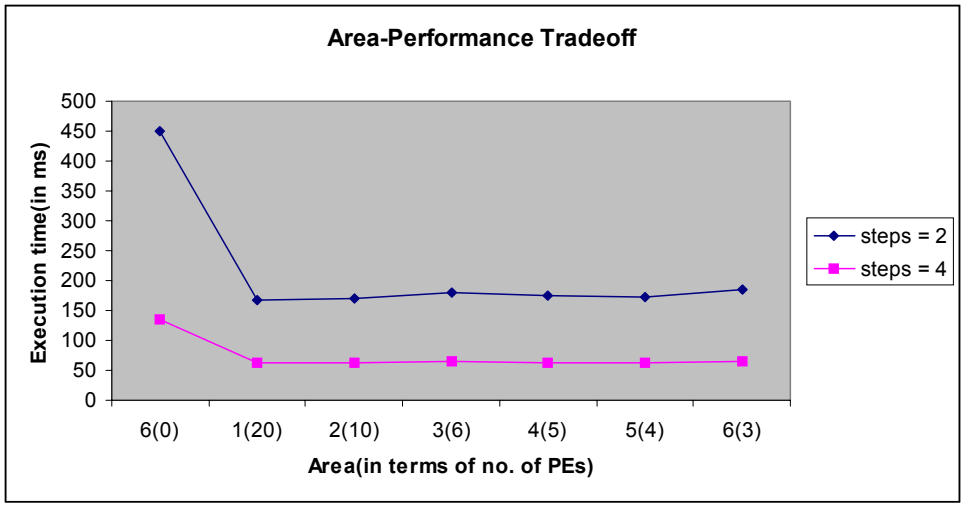

Figure 16. Area-performance trade-off results for face detection system. Area is in terms of the number of PEs and the degree of parallelism within each PE. Performance is measured by execution time.

most, which can often be tolerated for such applications.

4.6.2 3D Facial Pose Tracking System

In table 3 a comparison between the performance values predicted by the MCM expressions and obtained from actual expressions are presented while the area performance trade-off curve is given in figure 17. As mentioned earlier, a shared memory multiprocessor system was used as the target platform. The OpenMP parallel pro-

Table 3. Execution times for 1 frame for different design parameters for 3D facial pose tracking system implemented on shared memory multiprocessor system.

N_p	Estimation (ms)				Experimental (ms)			
	n = 1	n = 2	n = 4	n = 8	n = 1	n = 2	n = 4	n = 8
100	0.072	0.045	0.031	0.024	0.072	0.048	0.037	0.030
200	0.127	0.072	0.045	0.031	0.135	0.080	0.059	0.048
300	0.181	0.099	0.058	0.038	0.188	0.120	0.080	0.060
500	0.291	0.154	0.086	0.052	0.310	0.186	0.117	0.095
1000	0.564	0.291	0.154	0.086	0.611	0.346	0.229	0.172

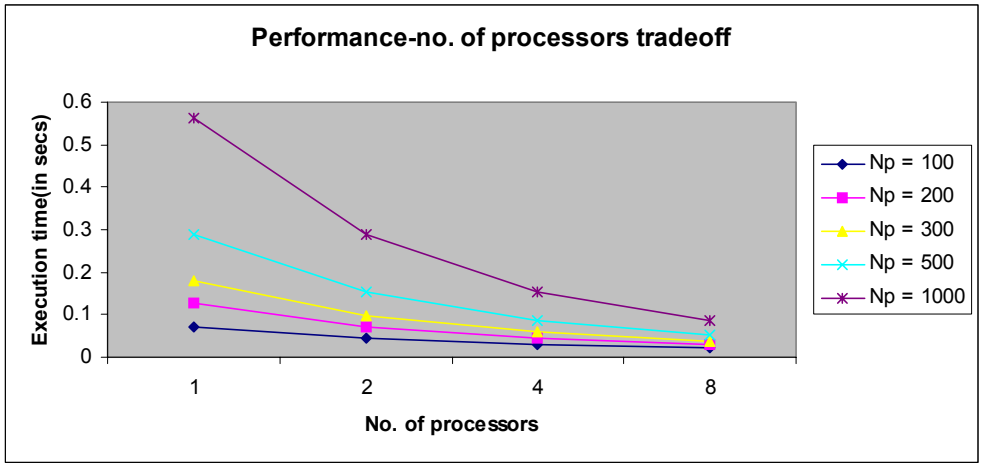

Figure 17. Performance-no. of processors trade-off results for 3D facial pose tracking system. Performance is measured by execution time.

gramming model was used for this implementation. The estimated and experimental values match very closely for lower numbers of threads, but for larger numbers of threads, the estimations are not as good. The reason behind this is once again the assumption of negligible execution time of synchronization actors for this system. The synchronization actors try to capture the time to schedule and synchronize the threads, which is difficult to model and estimate for the given system and does not remain valid for larger number of threads. For large numbers of threads, there is a significant overhead for scheduling the threads as well as associated synchronizations. The maximum frame rate obtained for this application is 33 fps for 100 particles using 8 threads, which is higher than the required 30 fps typical for such applications.

The PDSP system simulation was done using Code Composer Studio (Version 2) from Texas Instruments. The TMS320c64xx processor series was used with an instruction cycle time of 40ns. The sizes of the RAMs used were 320KB and 1.6MB for instruction and data, respectively. The main objective behind this experiment was

Table 4. Execution times for 1 frame for different design parameters for 3D facial pose tracking system implemented on a PDSP.

No. of particles	Estimation (s)	Experimental (s)
50	4.17	4.23
100	7.09	7.21
150	10.02	10.2
200	12.94	13.21
250	15.87	16.21

to study the implications of embedding such an application in a PDSP platform. It was observed that the memory requirement is higher than other typical PDSP applications. Also, without parallelization, such implementations are far from meeting the required frame rate. Synchronization graph modeling — though trivial in this case because the target was a uniprocessor platform — was carried out to estimate the execution times. The results are shown in table 4; the estimated results match very well with the experimental results.

For the face detection system, a maximum frame rate of 12fps was obtained for the given ML310 target board. This indicated that increased resources or a tolerance for periodically dropping frames is required to achieve the target frame rate of 30fps. For the 3D facial pose tracking system, the PDSP implementation showed that without parallelization, reasonable performance cannot be achieved. The multiprocessor implementation gave, in exchange for significantly increased cost, very good performance results. The multiprocessor implementation achieved the target frame rate of 30 fps for many cases, and in the best case, gave an overachieving frame rate of 33 fps.

Chapter 5 : Multiprocessor Communication Interface for Signal Processing System

5.1 Introduction

The complexity of embedded signal processing applications is increasing rapidly, intensifying the need for multiprocessor implementation to meet execution time constraints. A class of important System-on-Chip (SoC) architectures is emerging that provides advanced support for such embedded applications. These architectures are composed of heterogeneous subsystems, including CPUs, embedded memory, memory interfaces, and specialized I/O interfaces, along with application-specific intellectual property (IP) cores. However, the growing complexity of both the applications and the SoC platforms has made the job of design and implementation more difficult. One of the key challenges in this regard is the issue of communication between the different heterogeneous processing units.

In the domain of general-purpose processing, the most widely-known endeavor in this regard is the message passing interface (MPI) protocol. The main advantage of MPI is that it is portable — MPI has been implemented for a wide-range of architectures, and each implementation is typically optimized for the hardware on which it runs. Although MPI provides for various features and various forms of flexibility, it has several drawbacks especially in the context of embedded processing systems. First, since it is designed for general-purpose multiprocessor applications, MPI cannot leverage optimizations obtained by exploiting characteristics specific to an application

domain. Also, MPI is mainly intended for computer clusters, and does not scale well for system-on-chip (SoC) platforms.

The focus in this work is at multiple levels, including the development of an efficient protocol and communication interface, and the derivation of an efficient implementation of this interface for generic signal processing applications. As mentioned earlier, computer vision applications — an important class of applications in the domain of signal processing — are characterized by their computational intensity and distributed signal processing properties, and their real-time requirements have started creating a strong demand for sophisticated multiprocessor implementation.

The new communication interface presented, called the signal passing interface (SPI) attempts to address this demand by integrating relevant properties of two different yet important paradigms in this context — dataflow and the message passing interface (MPI). SPI is targeted towards signal processing applications, and through careful specialization, it is streamlined for embedded implementation in this domain. SPI attempts to overcome the overheads incurred by the use of MPI for signal processing applications by carefully integrating concepts of MPI with coarse-grain dataflow modeling and useful properties of interprocessor communication (IPC) that arise in this dataflow context. The resulting integration provides a new, more intuitive and easy-to-use paradigm for multiprocessor implementation of signal processing applications.

Of the various signal-processing-oriented dataflow models of computation, synchronous dataflow (SDF) is a particularly popular one because of its static nature, which leads to compile-time predictability and potential for extensive analysis and optimization. Hence, the underlying dataflow model in SPI is SDF. However, SDF

does not allow dynamic rates of data transaction between different subsystems in a dataflow graph, thereby limiting its applicability. A significant number of signal processing applications do not conform to the rigid dataflow structure imposed by SDF semantics. For example, consider a multiple camera object tracking system. In such a setup, each camera typically communicates with other cameras at run-time to exchange and update data. The values of this data and more importantly the overall volume of the data cannot be determined statically since they depend on the current video input. Thus, to implement an effective communication interface for such domains, it is imperative to allow some amount of dynamic behavior, and hence to look beyond conventional SDF for the proposed interface.

In this work, the concept of variable token size (VTS) — earlier supported implicitly by various forms of dynamic dataflow — is presented as an explicit modeling tool to apply efficient and intuitive SDF techniques to certain kinds of dynamic dataflow behavior. This concept is integrated into SPI to enable efficient handling of dynamic behavior in subsystems that need such flexibility, while static-dataflow-oriented subsystems are handled using conventional, fixed-token-size SDF semantics.

A significant motivation for using SDF in SPI is to exploit the powerful optimization techniques that are enabled by design and synthesis using SDF. One such technique is resynchronization. Resynchronization is used in the context of multiprocessor implementation of SDF graphs for reducing synchronization overhead between processors. Until now, this technique has been mainly used for shared-memory systems. Through the SPI framework, the concept of resynchronization is extended to distributed memory systems, and corresponding optimizations are incorporated into SPI.

An interface definition is incomplete without an optimized implementation. Two SPI library implementations have been developed. An SPI library consists of special communication-related, functional modules (communication actors) that take care of message passing and implementing the various synchronization and buffer management protocols for correct functioning of an SPI-based system. These special modules ensure that the communication part of a system is completely separated from the computation part.

5.2 Dataflow-based Modeling of Signal Processing Systems

When dataflow is used to model signal processing applications for parallel computation, four important classes of multiprocessor implementation can be realized: fully static, self-timed, static assignment and fully dynamic methods [47]. Among these classes, the self-timed method is often the most attractive option for embedded multiprocessors due to its ability to exploit the relatively high degree of compile-time predictability in signal processing applications, while also being robust with regards to actor execution times that are not exactly known or that may exhibit occasional variations [71]. For this reason, the SPI methodology is targeted to the self-timed scheduling model, although adaptations of the methodology to other scheduling models is feasible, and is an interesting topic for further investigation.

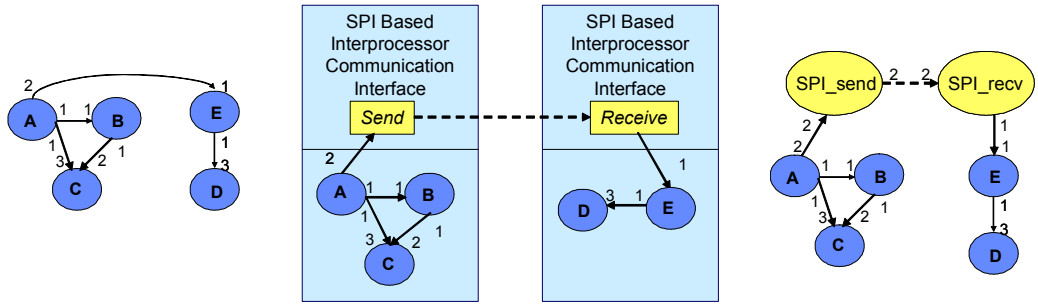

Figure 18. Model for implementing the SDF graph on the left on a 2-processor system and the new SDF graph.

5.3 The Signal Passing Interface

For a given dataflow graph, SPI inserts a pair of special actors (called SPI actors) for sending and receiving associated IPC data whenever an edge exists between actors that are assigned to two different processors, as shown in figure 18.

The static properties of SDF together with self-timed scheduling provide important opportunities for streamlining of interprocessor communication. This leads to the following differences between SPI and MPI implementations:

- In MPI, a *send* message contains the identifier (ID) of the destination actor. In SPI, instead, the message contains the ID of the corresponding edge of the graph.
- In the preliminary, static version of SPI, there is no need to specify the size of the buffer associated with a message, as in MPI. Once a valid schedule is generated for the graph, all the edge buffer sizes are known, and hence, the edge ID can be used to determine the buffer size. Although buffer overflow conditions may occur if one of the processors produces data at a faster rate than the corresponding consumer

processor consumes it, these conditions can systematically be eliminated in SPI through appropriate buffer synchronization protocols.

- SPI allows only non-blocking calls as opposed to provisions in MPI. Any buffer overwriting or race conditions are avoided by the self-timed execution model, which enforces that 1) any actor can begin execution only after all its input edges have sufficient data, and 2) each actor is implemented on one processor only, and not distributed across multiple processors.

5.3.1 Variable Token Size (VTS) Model

As mentioned earlier, a significant limitation of SDF is that it does not allow dynamic variation in token size. In this section, the concept of variable token size (VTS) is proposed to deal with dynamically varying data rates encountered in various signal processing applications.

In *dynamic* dataflow, the production/consumption rates of an actor may change at run time depending on current or previous values of its input data. Dynamic dataflow originates at dynamic ports. An actor is called dynamic if at least one of its ports is dynamic. General dynamic dataflow, with no apriori information about the dynamic behavior, has high overhead and low predictability for synthesis of efficient real-time implementations, since, for example, this requires fully dynamic memory management

Our approach to providing for a significant degree of dynamic dataflow is the use of variable token sizes, while maintaining static production/consumption rates of actors in terms of the numbers of tokens that are produced or consumed. Furthermore, to enable static memory allocation, it is required that an upper bound on the token size is specified for each dynamic port. VTS provides a mechanism to “re-pack” tokens in

such a way that the new ("packed") tokens flow at static rates, in situations where the underlying raw ("unpacked") tokens were flowing at dynamic rates.

Consider actors A and B in figure 19 The production rate of edge (A, B) varies dynamically but has an upper bound of 10. Similarly, the consumption rate varies with an upper limit of 8. These varying rates can be captured using variable token sizes. Thus, A has a production rate of 1 with a token size of x and B has a consumption rate of 1 with token size of y, where x has an upper bound of 10 and y has an upper bound of 8. If by application of the above principle to all possible edges, a consistent graph is obtained, then bounded memory for all the edge buffers can be guaranteed. Such a conversion is denoted as *VTS conversion* of the original dataflow graph.

An upper bound on the total size of the *packed* tokens on an edge e is required to ensure bounded buffer memory using VTS. This may be computed as follows. Let $c_{sdf}(e)$ be an SDF buffer bound of e — i.e., an upper bound on the buffer size of e in terms of the maximum number tokens that coexist on e at any given time. $c_{sdf}(e)$ can be computed using any of the existing techniques for computing SDF buffer bounds (e.g., see [71], [72]). $c_{sdf}(e)$ is computed on the graph after VTS conversion, so it is computed on a pure SDF graph. The total size of *packed* tokens $c(e)$ is then given as

$$c(e) = c_{sdf}(e) \times b_{max}(e), \tag{5.1}$$

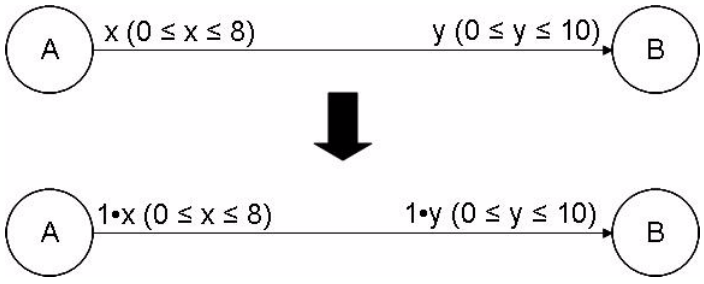

Figure 19. SDF graph with dynamic data rates and corresponding VTS conversion.

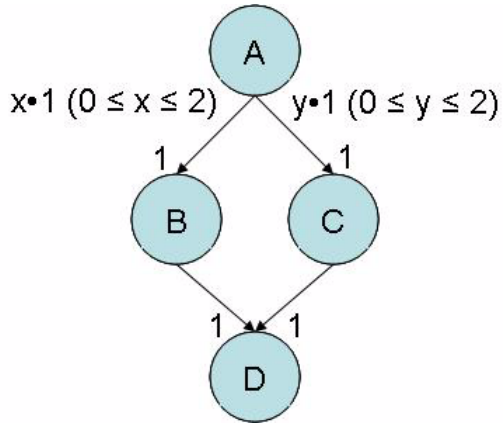

Figure 20. Example showing how VTS may cause unbounded buffer memory requirements.

where $b_{max}(e)$ is the maximum number of bytes in a *packed* token associated with e in the VTS conversion. In the application of VTS, it is required that the bound $b_{max}(e)$ exists and is known in advance. When the bound exists, it can be determined from any available bound on the maximum variable data rate for a port (e.g., the bound on x in figure 20) times the maximum number of bytes in a single raw (unpacked) token for the port.

The IPC buffer bounds, as computed in [71], now gets modified to

$$B(e) = (\rho(src(e), snk(e)) + delay(e)) \times c(e), \tag{5.2}$$

where $B(e)$ is the upper bound on the IPC buffer size, ρ_G is the total delay on a minimum delay path directed from $src(e)$ to $snk(e)$, $delay(e)$ is the initial delay on edge e and $c(e)$ is as defined in equation (5.1).

In terms of implementation of VTS in the context of SPI, there have to be provisions for notifying the receiving actor of the token size of the current tokens being sent. This can be done either by transmitting a token size in the header or by using a special delimiter that is then used by the receiver to determine the length of the mes-

sage. The most efficient method to use is dependent on the implementation platform. For example, if the final target is an FPGA (as in this case), using a delimiter can be expensive as it would then involve extra operations on the receiver side to determine the length of the message. Thus, sending the size using a field in the header of the message is much more efficient.

Note that VTS conversion of a graph requires that actors operate in terms of packed tokens, and this mode of operation can in general result in actors that have unbounded storage requirements for their internal state. Thus, to ensure overall bounded memory for an application, it must be ensured that the actors themselves operate within bounded memory (e.g., by disallowing dynamic memory allocation within actor implementations). The preliminary experiments, however, indicate that in practice, VTS conversion can be performed for useful applications in conjunction with bounded memory actor implementation, and therefore the technique can be an important technique to consider when encountering dynamic dataflow behavior.

It may be noted that the VTS approach developed in this work differs from the models supporting dynamic behavior discussed in 2.1.3 and 2.1.4 by providing an explicit way to allow dynamic token transfer rates within the SDF framework. Bounded dynamic dataflow (BDDF), introduced by Pankert et al. [58], allows arbitrary data rates as long as an upper bound is specified for the data rate of each dynamic port. The application of VTS shown here is similar to BDDF in the sense that it requires bounds on dynamic behavior. However, in this approach, dynamic behavior is captured by varying token sizes instead of varying data rates — i.e., repacking of tokens at dynamic-rate ports to enforce SDF behavior. Thus, in contrast with BDDF,

SDF-based analysis techniques can be applied to our VTS-based system representations.

5.4 Synchronization Graph Modeling

The SPI methodology uses the synchronization graph model, which is a graph-theoretic model for analyzing the performance and synchronization structure of a self-timed multiprocessor implementation [71]. In this section, a precise definition of the synchronization graph for SPI in the context of distributed memory systems is developed, along with the associated resynchronization technique to reduce synchronization overhead.

5.4.1 Synchronization and SPI

The *SPI_BBS* and *SPI_UBS* protocols derived earlier in the context of shared memory systems provide buffer synchronizations between the sender and receiver side. Here BBS and UBS stand for bounded buffer and unbounded buffer synchronization, respectively. These protocols were modified slightly for better optimization in the context of SPI for distributed memory systems to lead to *SPI_BBS* and *SPI_UBS* protocols.

Let S_a and R_a be actors in the original dataflow graph, while S and R be the corresponding interprocessor communication actors inserted by SPI based on the given self-timed schedule. Both S and R maintain a write pointer (wr) that is shared with S_a and R_a, respectively. R also maintains a read pointer (rd) that is shared with R_a.

If it can be guaranteed that a buffer will not exceed a predetermined size, then the

SPI_BBS protocol is used. Let the maximum buffer size on both S and R be B_{bb}. The protocol operates as follows:

- S_a writes a token to the buffer, and increments the write pointer as $wr = (wr + k) mod B_{bb}$, where k is the total number of tokens received.
- Once the number of tokens written into the write buffer is equal to the production rate of the corresponding actor, S sends a message to R containing the new data tokens and message header.
- Upon receiving the message, R modifies its buffer, sets the write pointer accordingly, and sends an acknowledgement to S.
- R_a reads a token when $rd \neq wr$. After reading a token, R_a modifies the read pointer as $rd = (rd + k) mod B_{bb}$.

The *SPI_UBS* protocol is used when it cannot be guaranteed statically that an IPC buffer will not overflow through any admissible sequence of send/receive operations on the buffer. In this protocol, an additional counter of unread tokens (ur) is maintained; this counter is also shared between each interprocessor edge actor and its associated SPI actor. Let the buffer size on both S and R be B_{ub}. The protocol operates as follows:

- Before S_a writes to the buffer it checks if $ur \neq B_{ub}$. If this inequality holds, then it writes to the buffer and increments the write pointer as in SPI-BBS, and it also increments the count ur.
- Then S sends a message to R from offset rd.
- When S receives an acknowledgment for the send message, it decrements the count of unread tokens ur.

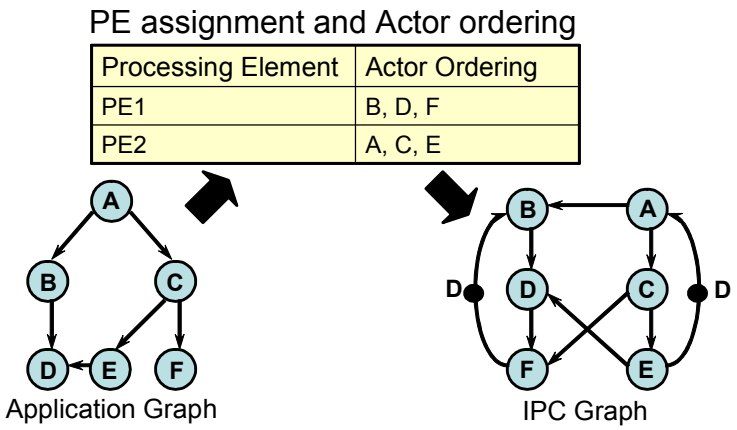

Figure 21. Example of derivation of an IPC graph from an application graph.

• Upon receiving the message, R checks if $ur \neq B_{ub}$. If this inequality holds then it modifies its buffer, sets the write pointer accordingly, and increments the value of ur. However, R does not send an acknowledgement to S immediately.

• R_a reads a token when $ur \neq 0$. It reads the token from offset rd and modifies the read pointer accordingly. The value of the count ur is decremented and the acknowledgement for the send from S is sent.

Note that the protocols remain invariant in case any of the interprocessor edges uses VTS, since that is already encapsulated in the header of the message.

5.4.2 Synchronization Graph

Given the dataflow graph for an application G and a multiprocessor schedule for G, we derive a data structure called the IPC graph G_{ipc} by instantiating a vertex for each task, connecting an edge from each task to the task that succeeds it on the same processor, and adding an edge that has unit delay from the last task on each processor to the first task on the same processor. Also, for each edge (x, y) in G that connects tasks that execute on different processors, an IPC edge is instantiated in G_{ipc} from to x to y.

Figure 21 shows an application graph and how the corresponding IPC graph is derived using processor assignment/actor ordering. An IPC edge in G_{ipc} represents both data communication and synchronization functions. The synchronization graph G_S, derived from G_{ipc}, shows synchronization constraints only. Initially G_S is identical to G_{ipc}. However, resynchronization modifies the synchronization graph by adding and deleting synchronization edges.

The synchronization graph construction depends on the underlying target platform (e.g., shared memory [71] or domain specific [65]). Thus, the construction of such a graph is a general part of SPI methodology.

5.4.3 Resynchronization

The process of adding one or more new synchronization edges and removing any redundant synchronization edges that result is called *resynchronization*. Resynchronization exploits the well-known observation that in a given multiprocessor implementation, certain synchronization operations may be redundant in the sense that their associated sequencing requirements are ensured by other synchronizations in the system. The goal of resynchronization is to introduce new synchronizations in such a way that the number of additional synchronizations that become redundant exceeds the number of new synchronizations that are added, and thus the net synchronization cost is reduced.

Figure 22 (adapted from [71]) illustrates how this concept can be used to reduce the total number of synchronizations in a multiprocessor implementation. Here, the dashed edges represent synchronization edges. Observe that if the new synchroniza-

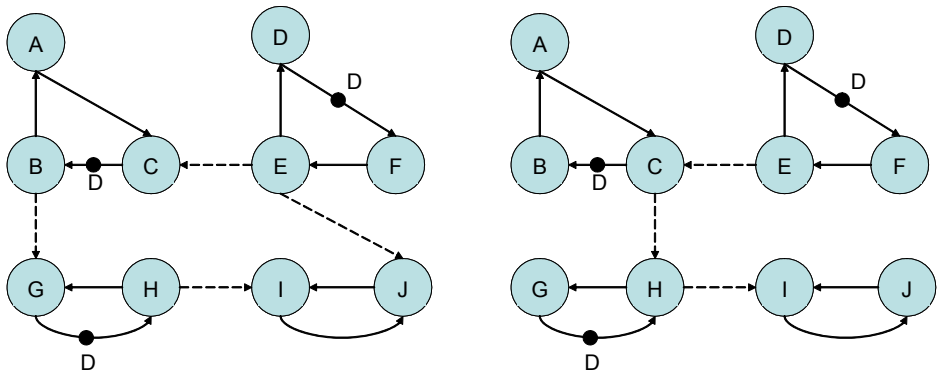

Figure 22. An example of resynchronization.

tion edge $d_0(C, H)$ is inserted, then two of the original synchronization edges — (B, G) and (E, J) — become redundant and can be removed

SPI-based implementation of a system that uses the *SPI_UBS* protocol can result in multiple redundant acknowledgements which increases the synchronization overhead. These overheads can be removed by careful and systematic application of resynchronization to the complete system. Thus, resynchronization for SPI based implementation involves removal of redundant acknowledgement edges for SPI actors. In the HDL-based SPI library implementation developed in this work, the corresponding actors do not implement synchronization acknowledgments, thereby implementing the *SPI_BBS* protocol. This technique is further illustrated using practical applications in sections 5.5.3 and 5.5.4.

5.5 Experiments

Experiments were carried out with three applications: a face detection application, a speech compression using linear predictive coding (LPC) and particle filter based fault

diagnosis system. In this section, details of the SPI library implementation and details of the system design and implementation of the applications using SPI are presented.

5.5.1 SPI Library Implementation

To accommodate static as well as dynamic behavior, we propose a two-phase SPI interface consisting of *SPI_static* and *SPI_dynamic* components. *SPI_static* handles the static communication part — i.e., communication between subsystems whose behavior is determined before run-time. For edges that exhibit dynamic communication behavior, *SPI_dynamic* is used. However, there are limitations to the range of dynamic behavior that can be encompassed. The most important restriction is that the communication edges should be known before run-time and they cannot vary at run-time. Once this is fixed, varying data rates can be supported by using VTS, given an upper bound on the variation, as described in section 5.3.1.

The first library implemented was in software and used an underlying MPI layer for communication. A new FPGA library for SPI was developed later using the Xilinx System Generator. *SPI_init*, *SPI_send* and *SPI_receive* actors for both *SPI_static* and *SPI_dynamic* were implemented. The message header for *SPI_static* consists of the ID of the interprocessor edge only while that of *SPI_dynamic* also contains the message size. Note that for *SPI_dynamic,* the need for this header element can be eliminated by using an appropriate delimiter. However, for FPGA implementation, use of such header space is relatively inexpensive and is more efficient. Also, in the targeted implementations, the message datatype for all communication edges is known at compile-time, and hence need not be included in the message header.

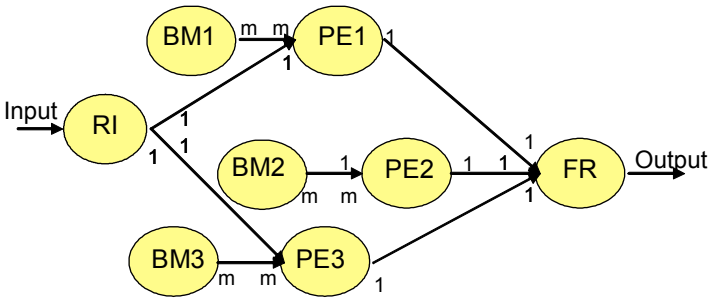

Figure 23. Coarse-grain dataflow model for face detection system.

5.5.2 Face Detection

Computer vision applications such as face detection are good candidates for parallelization as they are computation-intensive, and at the same time, inherently parallelizable. In this work, a shape-based approach proposed by Moon et al. [52] is used. Details of this application can be found in 4.3.1 Profiling results show that the mask correlation operation is computationally the most expensive. However, this operation is parallelizable. The mask set can be divided into subsets and each subset can be handled by a single processor independent from the rest.

A coarse-grain dataflow model of the resulting face detection system is shown in figure 23 for the case of 3 processors for mask correlation with corresponding actor to processor assignment shown in figure 24. Besides using multiple processors for the

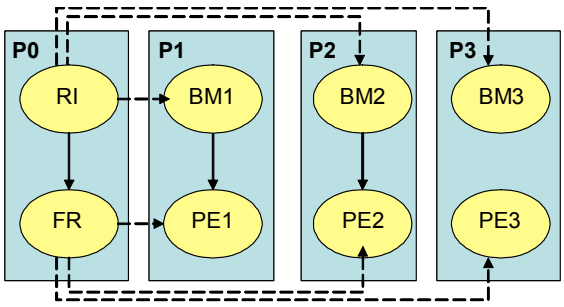

Figure 24. Actor to processor assignment for face detection system.

correlation operation, we use a separate processor to handle the required I/O operations. The actors in the coarse-grain dataflow model are explained below:

- Actor RI: Reads Image I and downsamples it
- Actor BM_i: Creates the mask set for PE_i
- Actor PE_i: Computes correlation for mask set BM_i and image I and finds the local best match
- Actor FR: Finalize results by finding the best match amongst all the local matches and marking the outline

This system comprises of static operations only and hence does not require VTS or *SPI_dynamic*. It was implemented using the software SPI library.

5.5.3 Speech Compression

The acoustic data compression (ADC) algorithm that was used is LPC (linear predictive coding). The details of this application along with the SDF model are in section 3.4.1.

Most of the actors have high computational intensity and the FPGA hardware resources were not enough to fit a multiprocessor version of the whole system. Thus, for this application, only the parallelization of the error generation actor (D) in hardware is explored, while the rest of the actors were implemented in software Thus, this experiment of SPI is in the context of an overall hardware/software co-design solution.

Actor E requires the input samples as well as the predictor coefficients for error computation. Since the number of coefficients, that depend on the model order M and the size of the input frame, are not known before run-time, this leads to dynamic data

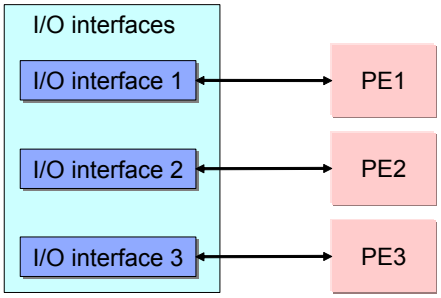

Figure 25. 3-PE architecture for error generation actor of speech compression application.

transactions and use of *SPI_dynamic*. For n processing elements (PEs) computing the errors in parallel, each PE computes $\lceil N/n \rceil$ error values.

The system architecture for a 3-PE implementation of the system is shown in figure 25. Each PE i performs error calculation for the part of the frame that is sent to it. Note that for the parallel version of error calculation, the input frame is split into over-lapping sections and each PE finds out the error values corresponding to $\lceil N/n \rceil$ such sections. The I/O interface for a given PE sends the predictor coefficients and the input frame subsections to the PE and receives the computed error values. Figure 26 shows the synchronization graph before and after resynchronization is performed. Here, the dashed edges represent synchronization edges. Note that the synchronization graph is

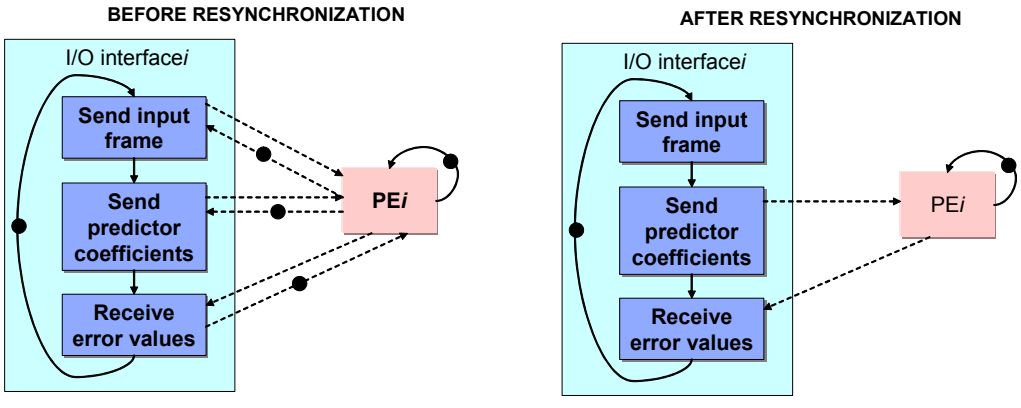

Figure 26. Resynchronization for 3-PE implementation of error generation actor of speech compression application.

constructed by considering appropriate *SPI_send* and *SPI_receive* actors jointly with their corresponding dataflow actors. These SPI actors are not shown in the illustrated synchronization graphs for purposes of clarity and since the main aim of the graph is to illustrate the synchronization operations.

5.5.4 Particle Filter based Fault Detection

The last application used for experiment is a particle filter based system for tracking crack failure length in the blades of a turbine engine [57]. In this application, particle filtering techniques are used to track crack faults in the blades of a turbine engine [57]. Particle filter provides a method for recursively estimating the unknown state X_t, from a collection of noisy observations Y_t. The state parameters to be estimated are dependent on the exact tracking problem being tackled. For example, in this application the state represents the fault dimension i.e, crack size. The state transition function characterizes the state evolution which in this case is the fault growth model given by

$$X_t = X_{t-1} + \frac{1}{(X^5{}_{t-1} + X^4{}_{t-1} + X^3{}_{t-1} + X^2{}_{t-1} + 1)} + W_t, \tag{5.3}$$

where W_t is zero-mean Gaussian white noise with variances 10 and 1, respectively. The state is updated based on received observation corrupted by noise.

$$Y_t = X_t + V_t, \tag{5.4}$$

where W_t and V_t are zero-mean Gaussian white noise with variances 10.0 and 1.0, respectively.

An important step in the practical implementation of a particle filter is resampling. Resampling is the act of redrawing particles from the same density such that the

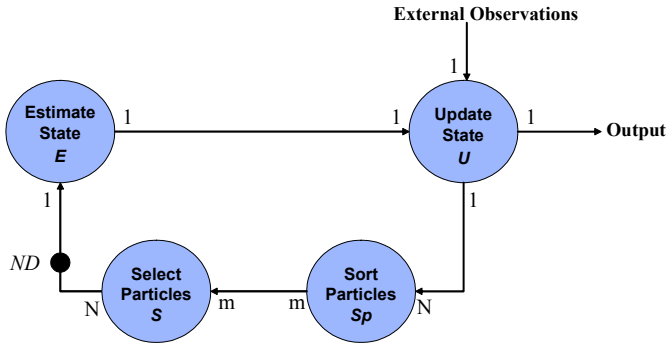

Figure 27. SDF modeling of fault-detection system.

weights of each particle is approximately equal. Resampling introduces dynamic behavior in a particle filter based application and is illustrated in detail later.

5.5.4.1 SDF modeling of fault detection system

Figure 27 shows the SDF model for the fault detection system. The description of the constituent actors are given below:

- Estimate State (*E*): This actor estimates the state given a particle by using the state equation for a given iteration. Thus, given N particles, it makes N estimates.
- Likelihood Estimate (*L*): This actor uses current noisy observation to calculate the likelihood estimates of the output given the current estimates.
- Update State (*U*): This actor uses the likelihood values calculated by *L* to update the current state estimates and the particle weights.
- Sort Particles (Sp): This actor sorts the particles according to their weight and passes the best m particles for resampling in the next step.
- Select Particles (*S*): This actor given the current particles and their likelihood values resamples them to choose the particles to be propagated to the next iteration.

The delay in the edge between actors *S* and *E* reflect the initialization condition:

the particle filter has to be initialized with N particles all having equal weight, where N is the total number of particles.

All the steps in a particle filter can be completely parallelized except the resampling step. Distributed resampling algorithms have been developed that parallelize the load to a certain extent. In the implementation, the particles are equally distributed among processing elements (PEs) after which all the steps execute in parallel and communicate only during resampling. Thus for N particles (N is typically large and for this system can vary from 50 to 300) and n PEs, each PE handles $\lceil N/n \rceil$ particles in each iteration.

In the distributed resampling scheme utilized here, first *local resampling* is performed in a PE, in which replication factors for the particles of a given PE are calculated locally. Then in the next step, called *intra-resampling*, excess new particle values are communicated to the other PEs to ensure that all PEs have the same number of particles for the following iteration. Thus, in this case all communication between PEs is not-deterministic.

Figure 28 shows the system architecture for a 2-PE implementation of the system. The resampling step is split into three steps — (1) the initial step of calculating a partial sum and communicating to other PEs; (2) local resampling; and (3) intra-resampling. There are two messages passed between the PEs: the first one is to exchange local sums, and the second one is to exchange particles. In terms of SPI modes, the first message has static and known length and hence *SPI_static* is used. On the other hand, the second message has dynamic length, which is decided at every iteration, and hence *SPI_dynamic* is used. The synchronization graphs before and after resynchroni-

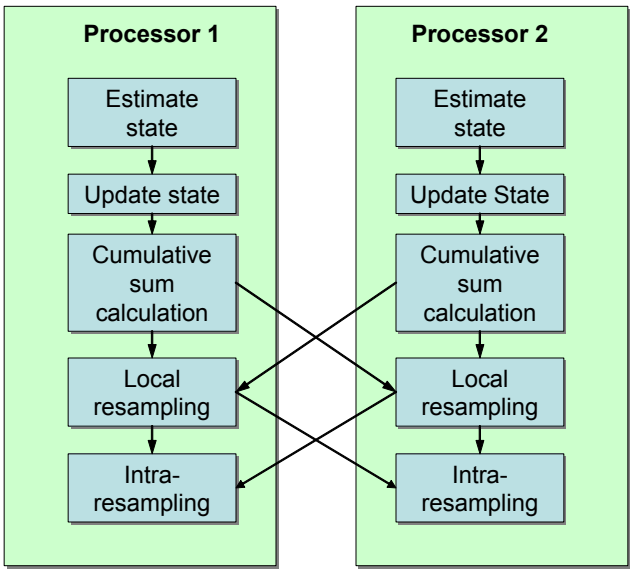

Figure 28. 2-PE architecture for fault detection system.

zation is performed are shown in figure 29, where again the dashed edges represent synchronization edges.

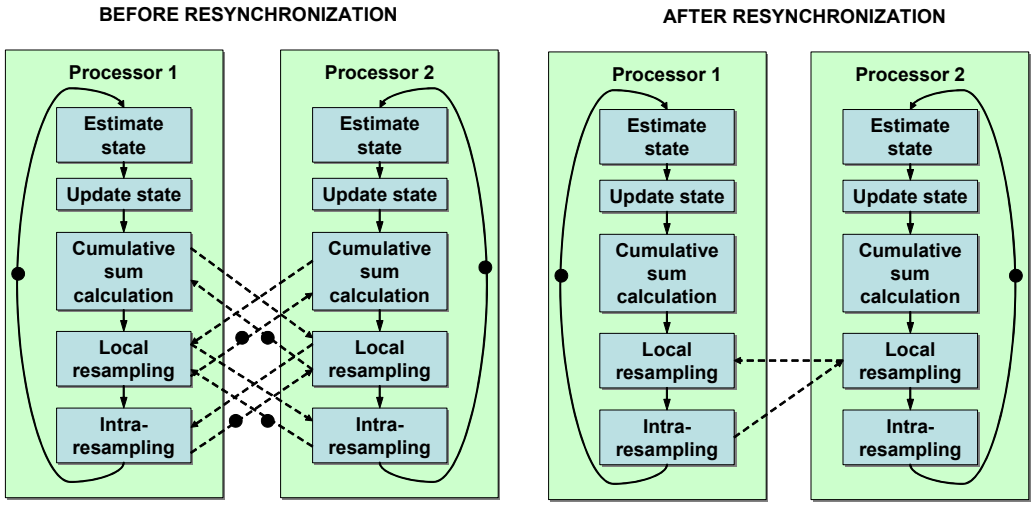

Figure 29. Resynchronization for 2-PE implementation of fault detection system.

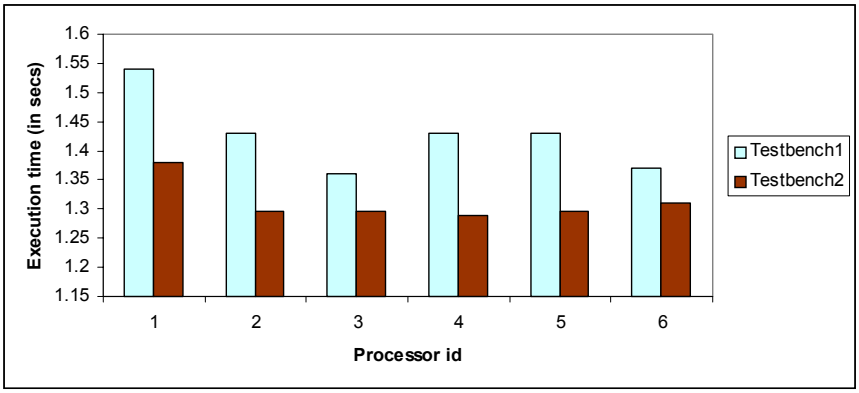

Figure 30. Execution time results for face detection system.

5.6 Results

The software implementation for the face detection system was targeted towards a multiprocessor platform consisting of a combination of IBM Netfinity and Dell Poweredge hardware nodes: each node was a dual-processor PIII-550 (2xPIII-550Mhz) with 1GB memory and 2x18GB of disk capacity. Each node had a 1Gigabit link to the other nodes. Two test benches involving 126 and 114 masks, each of size 121x191 and 127x183 pixels, and image of size 128x192 pixels were used. The execution time results for the two testbenches are shown in figure 30.

The hardware implementations that were experimented with were designed

Table 5. FPGA resource requirements for a 4-PE implementation of error generation module for speech compression system.

	Slices	Slice FFs	4 input LUTs	Block RAMs
Full system	2.63%	1.88%	2.15%	8.33%
SPI library (relative to full system)	11.88%	12.5%	13.94%	50%

using Xilinx System Generator version 9.1 and synthesized using Xilinx ISE 9.2. The target device family was Virtex 4 with a speed grade -10. Although the FPGA board could support a clock frequency of 500 MHz, this frequency could not be attained in most cases.

Figure 31 and table 6 show the performance results that were obtained for the error calculation module of the speech compression system and the particle filter based failure prognosis system. For both cases, n represents the number of PEs used. $n = 1$ implies the serial implementation. As may be observed from these figures, parallelization leads to significant improvement in performance in these experiments.

Table 5 shows the FPGA area requirements for a 4 PE implementation of the speech compression system along with the SPI library resource requirements relative to the full system, while table 7 shows the same for 2 PE implementation of the fault detection system. The computational requirement for the fault detection system was relatively high and hence only 2PEs could be accommodated. The SPI library area sizes are small in both cases — although for the first case, the relative size is larger than the second one, because of the overall smaller size of the full system.

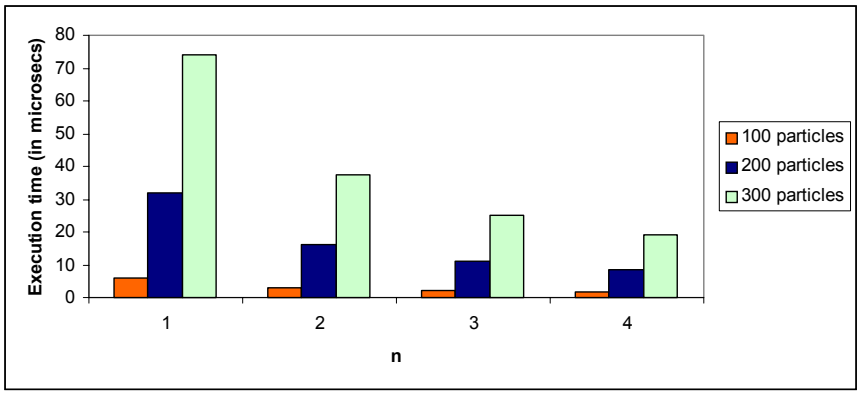

Figure 31. Performance results for error generation module for speech compression system.

Table 6. Performance results for fault detection system.

No. of Particles (N)	Execution time (in μsecs)	
	n = 1	n = 2
50	30	20.33
100	50	30.27
200	90	50.25

Table 7. FPGA resource requirements for 2 PE implementation of fault detection system.

	Slices	**Slice FFs**	**4 input LUTs**	**Block RAMs**	**DSP 48s**
Full system	90.74 %	47.52%	65.48%	18.23%	56.25%
SPI library (relative to full system)	0.2%	0.08%	0.27%	11.43%	0%

Chapter 6 : Parameterized Design Framework for Particle Filter Systems

6.1 Introduction

Particle filtering is an emerging and powerful methodology for sequential signal processing with a wide range of applications in science and engineering. Researchers from a variety of fields ranging from signal processing to statistics and econometrics use particle filters because of their potential for coping with difficult nonlinear and/or non-Gaussian noise problems. Particle filters are based on the idea of approximating the probability density functions (PDFs) of the state of a dynamic model by random samples (particles) with associated weights and propagating them across iterations based on the probabilistic model of the state update and the measurements. But use of particle filters in real-time systems has been limited due to their computational complexity. A particle filter typically involves several complex mathematical operations that are invoked at every iteration of the filter, as well as a large number of particles, which in turn results in huge memory requirements. A possible solution for real-time implementation of such systems is parallelization and the use of multiprocessor systems; but this is also restricted because of the presence of an unavoidable computing step (resampling), which is serial in nature, and therefore difficult to parallelize. This suggests the need for exploration of customized solutions.

Design and implementation of a generic yet highly optimized architecture for all particle filter based systems is not possible because of the wide range of applications

to which particle filtering techniques are applied currently and may be applied in the future. But, there are many applications that share similarities as far as particle filtering is concerned. A generic architectural framework that can be suitably and easily reconfigured for such applications would be of significant utility. Such an architecture could be highly optimized as well because of the potential for streamlining based on a given set of particle filtering features.

The emerging class of architectures comprising of heterogeneous SoCs provides advanced support for embedded signal processing applications. Such architectures include CPUs, embedded memory, memory interfaces, and specialized I/O interfaces, along with domain-specific IP cores. However, the increased functionality of such architectures leads to increase in design and implementation complexity. Examples of such heterogeneous processing platforms are platform FPGAs.

In this chapter, an SoC architecture involving parallel processing units for tracking applications using particle filters is proposed. The architecture utilizes specialized hardware elements as well as special soft cores. Additionally, a novel parameterized design framework to implement particle-filter-based applications on platform FPGAs is presented. The aim of this framework is to enable comprehensive design space exploration of complete particle filtering systems with attention to the interaction between the various processing subsystems and different particle filtering parameter configurations that may be used across different applications. Specifically, the following objectives have been addressed:

- A novel architectural design framework for implementing a class of particle filters on reconfigurable system-on-chips.

- Robust design space exploration using the design framework
- Exploration of hardware/software codesign and implementation, and analyzing trade-offs associated with partitioning and mapping.

6.2 System Design Framework

Particle filters provide a method for recursively estimating the unknown state X_t, from a collection of noisy observations. The state parameters to be estimated are dependent on the exact problem being considered. The state transition and observation models are given by

$$\textit{State Transition Model}\ \ X_t = F(X_{t-1}, W_t), \text{ and} \tag{6.1}$$

$$\textit{Observation Model}\ \ Y_t = G(X_t, V_t) \tag{6.2}$$

where W_t is the system noise and V_t is the observation noise. X_t represents the dynamically evolving state of the system, and Y_t is the observation vector of the system, which is corrupted by the measurement noise V_t at instant t. The particle filter estimates the state of the system X_t and updates it based on the received, corrupted observations.

As shown in any figure 32 particle filter based system essentially consists of the following three computational steps:

- *Sampling*: In this step samples (particles) of the unknown state are generated based on the given sampling function. These samples provide an estimate of the current state of the system and also propagate the particles from previous time instant to

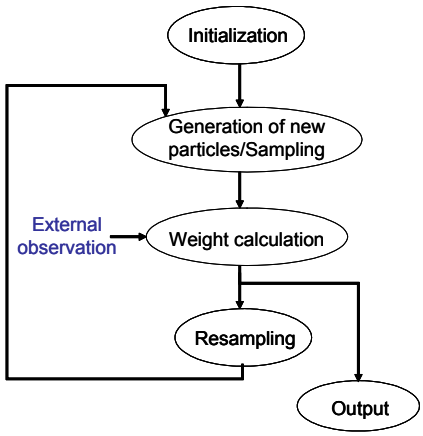

Figure 32. Particle filtering algorithm.

current.

- *Weight Calculation*: Based on the observations an importance weight is assigned to each particle.

- *Resampling*: This step involves the act of redrawing particles from the same probability density based on some function of the particle weights such that the weight of each new particle is approximately equal. Resampling is a very important step in a particle filter and without this step a particle filter is highly likely to degenerate, i.e., after a few iterations all the weights will go to zero except the weight of one particle.

While the sampling and the weight calculation steps are strongly dependent on the application, various standard methods of resampling exist and may be chosen based on their suitability for a given system. Also, sampling and weight calculation are generally the most computationally intensive and involve complex computations such as transcendental, trigonometric and exponential functions.

6.2.1 Architecture Overview

The architecture proposed in this thesis for particle filters is based on the computational framework described above. The wide range of applications to which particle

filtering techniques are applied prohibits the focus on a generic system architecture suitable for all applications. However, since there exist wide ranges of applications that use the same particle filtering algorithm with different state models, it is possible to develop a generic architecture for a subset of applications and streamline it for specific applications in this subset. The goal of this framework is to provide the user a systematic approach for such streamlining — with the ability to explore the various design trade-offs between area and execution speed — and provide the capability to implement a wide range of applications with significantly reduced re-design efforts.

To achieve this, first a system architecture is devised that is based on the use of parallel processing elements to achieve as much performance improvement using parallelization as possible. A comprehensive design framework is required for efficiently mapping the applications to this architecture and then finally onto the implementation platform. For this, a parameterized design framework is proposed. The fundamental idea being dividing the overall system into small parameterized subsystems. Each such subsystem can then be modified to the needs of a wide range of applications, as well as to final target constraints by setting appropriate parameters, such as the memory size, and the number of particles.

An overview of a two-processing-element configuration of such an architecture is given in figure 33. The framework essentially consists of an array of processing elements (PEs), and a resampling unit, along with a set of parameterized interfaces. A PE consists of three units, a PEcore, a weight calculation unit (WU), and a noise generator

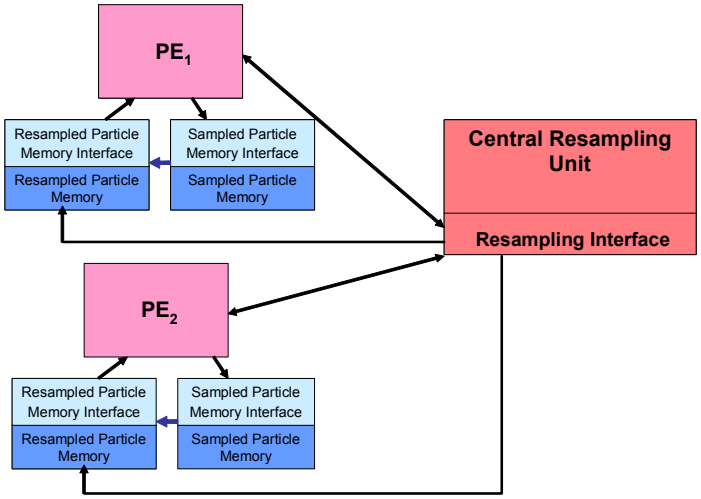

Figure 33. Distributed particle filter architecture.

as shown in figure 34. Each of these units can operate independently of changes in functionality of the other units. However, the interaction between various units can change with the variation in the functionality of any one unit. These changes are handled by the interfaces so that the individual streamlined units need not be redesigned, which would require significant effort. The PEcores perform the sampling operation, while a separate weight calculation unit (WU) is used for calculating the weights. The PEcore as well as the WU interact with memory banks whose sizes are dependent on system parameters. The interfaces provide parameterized interaction with memory

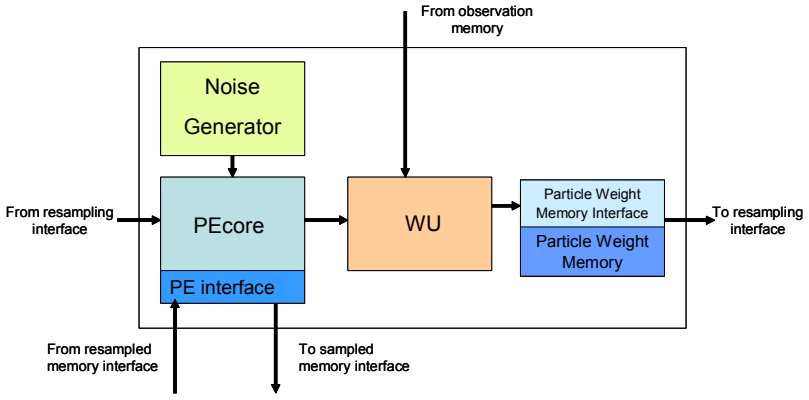

Figure 34. Single PE architecture.

banks and the resampling interface (where required), and perform synchronization operations. The individual units can be composed as specialized hardware modules or as software modules that are to be executed on embedded processors in the target platform.

6.2.2 Design Framework

In this section the details of the design framework and the parameterizations that can be employed based on restrictions imposed by the available implementation resources are presented. Figure 35 shows the overall design framework. Xilinx System Generator and the Xilinx EDK were used for design and functional verification of the hardware and processor (for software modules), and the Xilinx ISE tool-set for synthesis of the hardware modules. Xilinx System Generator provides a hardware library that consists of various architectural units, such as RAMs and adders, for modular design. It allows the use of custom Verilog or VHDL modules for system design. In addition one may also configure and use soft core modules such MicroBlaze or PowerPC mod-

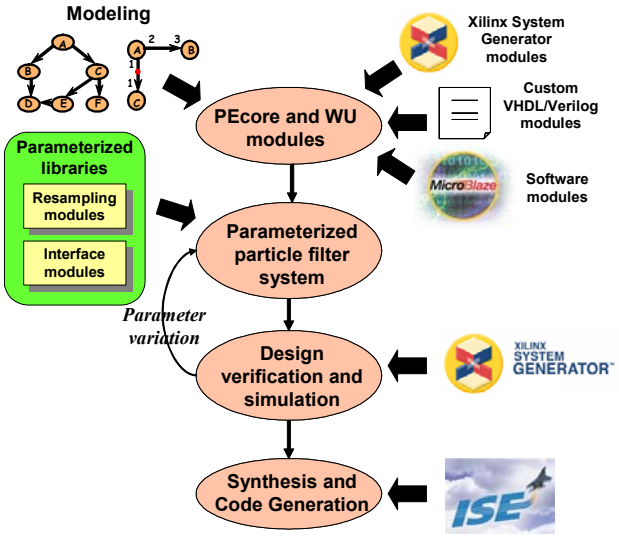

Figure 35. Parameterized design framework.

ules to implement software modules.

As mentioned in section 6.2.1, multiple processing elements (PEs) for the sampling and weight calculation step are used. Within a given PE, further pipelining can generally be used, but the degree to which pipelining can be employed is strongly dependent on the characteristics of the targeted application. The sampling and weight calculation operations involve complex mathematical operations, and thus impose restrictions on the number of PEs that can be implemented. The number of particles handled by each PE is

$$p = \lceil P/N \rceil, \tag{6.3}$$

where $\lceil x \rceil$ denotes the smallest integer that is greater than or equal to the real number x; P is the number of particles; and N is the number of PEs. In general, the sampling step can be implemented in hardware. However, since the weight calculation or "weight update" (WU) step for some applications may involve many complex mathematical functions, it may not always be possible to accommodate multiple WU units. In such cases, the most complex part is moved to the resampling unit. The WU unit computes an intermediate result that is then sent to the central resampling unit, which computes the final value before resampling. Since the resampling unit is serialized, it accommodates only one unit for computing the complex operations.

A straightforward memory management scheme for particle storage and updating is used. Three memory banks or buffers are used for each PE for storing (1) sampled particles, (2) particle weights, and (3) resampled particles. Since the number of memory banks that are available on a given platform is limited,

$$N \leq (M/3), \tag{6.4}$$

where M is the number of memory banks available on the targeted FPGA board. The area consumed by the associated memory banks directly depends on P and N. The observation data is stored in a shared memory between clusters of PEs. The memory interface for this buffer handles the read requests from the PEs. The reading from this memory for the ith operation can be overlapped with either the resampling step of the $(i-1)$th operation, the sampling step of the ith operation, or both. However, if the system throughput is greater than or equal to the observation input rate, this interface becomes trivial as only a single buffer is required. Note that in the case the WU unit is partially integrated with the resampling unit the particle weight memory stores the intermediate weight.

There are seven main interfaces corresponding to the operations of (1) observation data reading, (2) sampled particle memory interfacing, (3) resampled particle memory interfacing, (4) particle weight memory interfacing and (5) resampling unit interfacing. Among these, the reading of observation data is not dependent on N or P, while the rest are dependent on N and P. The resampling unit varies based on the resampling scheme being used and is functionally independent from the rest of the units. It is triggered when all the P particles have been processed for a given iteration.

The resampling interface consists of a global address generator and a local address generator. The global address generator generates addresses for P particles and depends on P. These addresses are routed to individual PEs by the local address generator, which, thus, depends on both P and N. In this framework, systematic resampling has been used. However, this can be easily replaced with other sequential resa-

mpling mechanisms. Systematic resampling is often a preferred method due to its computational simplicity and good empirical performance. When the PEs carry out partial weight calculation, the remaining weight calculation is carried out in the resampling unit and hence the corresponding module is integrated. A library of these parameterized interfaces and resampling schemes are created using a combination of Xilinx System Generator hardware components and custom HDL modules.

The execution time for resampling directly depends on P and is constant over all iterations. Thus, the total execution time (in terms of clock cycles) for one iteration is,

$$T = T_{PEcore} + L_{resampling} + L_{WU}, \tag{6.5}$$

where $L_{resampling}$ is the latency due to the resampling unit, L_{WU} is the latency induced by the WU unit, and T_{PEcore} is the execution time of PEcore, which for a fully pipelined PEcore is given as

$$T_{PEcore} = L_{PEcore} + \lceil P/N \rceil, \tag{6.6}$$

When partial weight calculation is performed in the PE, Equation (5) is modified to

$$T = T_{PEcore} + L_{resampling} + L_{WU1}, \tag{6.7}$$

For systematic resampling, the latency for complete hardware implementation is given by [3]:

$$L_{resampling} = 2 \times P - 1, \tag{6.8}$$

This signifies that the latency of the resampling unit increases directly with an

increase in number of particles, and thus the latency will generally become a bottleneck for applications requiring very high P. The total resampling time is given by

$$T = L_{resampling} + L_{WU2} + L_{interface}, \tag{6.9}$$

where, L_{WU2} is non-zero only when partial weight calculation is done in the PEs and depends on the complexity of the computation. For software implementation of the resampling module — as explored in one of the implementations in this work — $L_{resampling}$ provides the latency due to interfacing between hardware modules and EDK processor and depends on both N and P. For the first processing iteration, any initial latency that exists should be added to the latency model of (6). Such initial latency may exist, for example, because of startup time associated with the noise generator.

6.3 Experiments

In this section, implementations for three different particle filter problems using the proposed architectural framework are demonstrated along with corresponding experimental results. First, the basic framework is illustrated by means of two particle filter systems using underlying one-dimensional models. The results of the implementations show that the block RAMs (BRAMs) were not fully utilized for these designs, which indicates that systems with multi-dimensional models can be supported as well. The next application explored is based on such a multi-dimensional model. This application is a 3D facial pose tracking system in video. Based on the proposed design methodology, details of this application are provided, along with partitioning and map-

ping results.

The three systems were designed and synthesized using Xilinx System Generator 8.2, Xilinx EDK 8.2, and Xilinx ISE 9.1. For the first two applications, the target device family was the Xilinx Virtex-4SX series. Although the FPGA board used in the experiments could support a clock frequency of 500 MHz, this frequency could not be attained in most cases. For the third application, the Video Starter Kit (ML 402) from Xilinx was utilized that provides advanced support for video and imaging applications. By varying key parameters appropriately, different implementations were obtained and various design options were explored.

6.3.1 Uni-variate Non-stationary Growth Model

The first application explored is an example of a one-dimensional non-linear system (typically studied in the context of stochastic systems) [16]. The state transition and observation models are as follows

$$X_t = 0.5 \times X_{t-1} + \frac{25 \times X_{t-1}}{1 + X^2_{t-1}} + 8 \times \cos(1.2 \times (t-1)) + W_t, \tag{6.10}$$

$$\text{and } Y = X^2_t/20 + V_t, \tag{6.11}$$

where W_t and V_t, are zero-mean Gaussian white noise with variances 10 and 1, respectively. The execution of the PEs and the resampling units is fully pipelined.

The above equations were mapped to appropriate Xilinx System Generator computation blocks to build the PEcore and the WU. The noise generation was performed using Xilinx's Gaussian white noise generator. This noise generator needs only periodic resetting to provide continuous output, thus the PE interface did not have to send

requests for data. However, the initial latency of the generator is 10 cycles, which is present only for the first iteration. Additionally, Xilinx's lookup-table-based cosine generators were used. These are both fast and inexpensive (area-efficient) compared to standard CORDIC cosine generators. Fully-pipelined multipliers and dividers were employed. In the design, the WU uses an exponential calculation unit that uses a combination of a look-up table and a polynomial approximation method. Uniform random number generation for resampling is done using multiple-bit, leap-forward linear feedback shift registers (LFSRs) [21]. Parameterized interfaces were used to build the interconnections between the various subsystems.

6.3.2 Fault Detection System

This practical particle filtering application is adapted from [57], where particle filtering is used to track crack faults in the blades of a turbine engine and has been described earlier in section 5.5.4. The design units from 6.3.1 were reused appropriately to create the implementation for this system.

6.3.3 3D facial pose tracking in video

For this system described earlier in section 4.3.2, the computational complexity of the application made implementation of the whole system on hardware unfeasible. Hence, an embedded processor to implement software modules was used as well. Thus, partitioning and mapping decisions were required to appropriately identify the units of the system to be moved into hardware and software modules on the platform. This was done based on profiling of a MATLAB-based software prototype.

6.3.3.1 Partitioning and Mapping

The initial algorithm was developed in MATLAB and hence the MATLAB profiler was used to derive a distribution of execution times across the various functional sub-systems. The MATLAB profiler provides information about individual function execution times along with the total execution times and the number of calls made to each function. The profiling results for individual execution times are shown in figure 36. In this figure, the execution time for a function is the total time spent in the function for a single execution of the overall program.

As one can observe from the figure, the "extract features" function, which extracts the feature vector, contributes the most to the overall execution time. This function is a part of the weight calculation step of the overall particle-filter. Thus, it is necessary to speed up this unit as much as possible. Computationally, this function consists of computing the mean pixel value of the grids/cells. This can be done in parallel since the computation for one grid/cell does not depend on the rest. However, if multiple units for this function are created to exploit this parallelism, the number of units is restricted

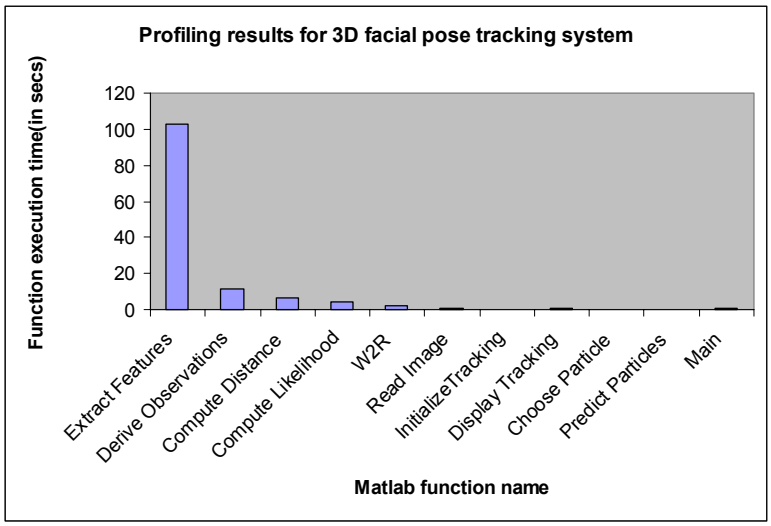

Figure 36. MATLAB profiler result for the 3D facial pose tracking system.

by the number of RAMs present in the target platform, since each of these parallel units requires a copy of the image. Shared memory can be used, but the number of read ports for such memories in FPGAs are limited, and hence, again multiple copies of the image would be required for maximal parallel data access. In the implementations, dual port RAMs have been used to enable limited sharing.

For this application, the WU unit is split up into hardware and software sub-units to enhance performance within given resource constraints. The “likelihood calculation” is moved to software since it involves complex math functions. Moving this unit to software, saves resources which can then be exploited for the parallel hardware implementation of the “extract features” function. Also, since the resampling function cannot benefit significantly from customized hardware realization, it is moved into software.

In this implementation rotation effects were ignored due to resource constraints. Taking such effects into account provides more tracking accuracy, but requires complex trigonometric and matrix manipulation functions. Given the targeted system architecture, which involves multiple PEs parallelized over the set of particles, including rotation effects would translate to providing multiple instantiations of the required mathematical manipulation units (for the trigonometric and matrix manipulations). These multiple instantiations can be supported logically as an extension of the system architecture presented in this work, but they would exceed the resources available on the targeted FPGA device. For future FPGA device families that provide more resources, incorporating rotation effects into the design framework presented here is a useful direction for further work.

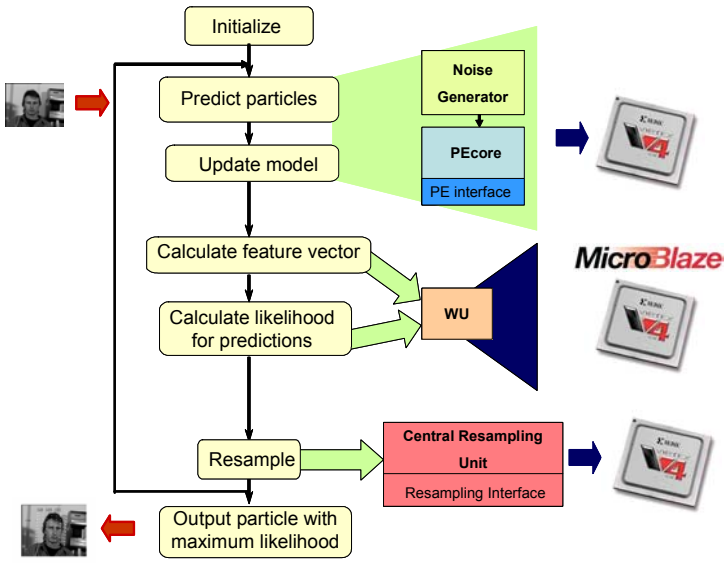

Figure 37. Mapping of subsystems of 3D facial pose tracking system onto hardware and software modules.

6.3.3.2 Implementation

The system architecture that was employed for this application is the same as that shown in figure 33. As mentioned earlier, the video starter kit (ML 402) was used to implement this application. The FPGA family supported by this board is the Xilinx Virtex-4 SX. The board also includes a video input/output daughter card and a CMOS image sensor camera. Xilinx's EDK version 8.2 along with Xilinx System Generator version 8.2 was used to create the MicroBlaze processor (clock frequency 100 MHz) for the software module implementation. The final mapping of the various subsystems is shown in figure 37

In this implementation, an important parameter is the number of RAMs (M) present in the board as that decides the amount of parallelization that can be achieved for the "extract features" function as well as the total number of PEs N.

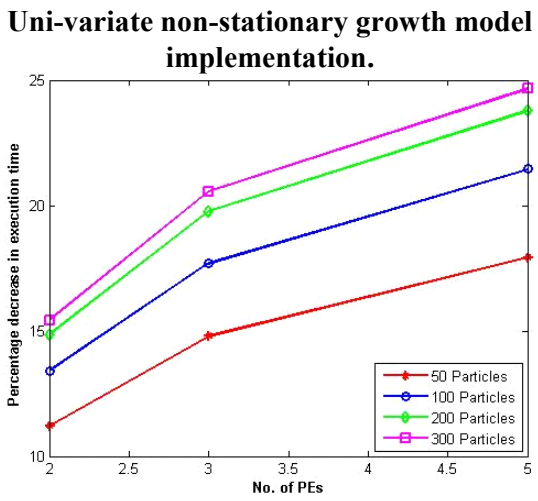

Figure 38. Percentage decrease in execution time (1 iteration) for uni-variate non-stationary growth model implementation.

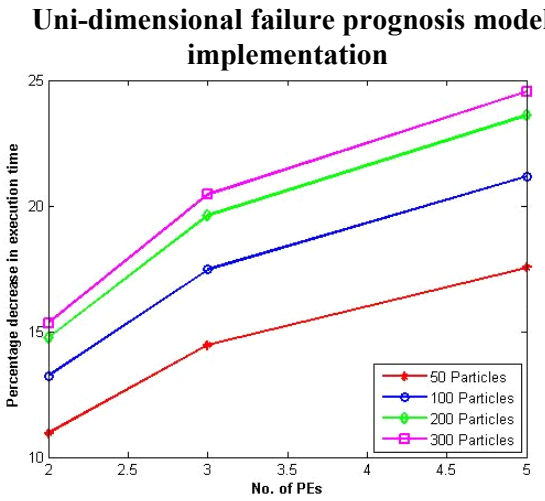

Figure 39. Percentage decrease in execution time (1 iteration) fault detection system implementation.

6.4 Results

The percentage decreases in execution times compared to serial execution are shown in figures 38 and 39 for the various design cases for the first two applications. The results shown are for one iteration at steady state — i.e., not the first iteration,

Table 8. FPGA resource utilization for uni-variate non-stationary growth model implementation.

Uni-variate non-stationary growth model implementation.

No. of PEs	Slices	Slice Flip-Flops	4 input LUTs	DSP48s	BRAMs
2	29.17%	8.54%	19.91%	43.23%	15.63%
3	42.25%	12.56%	28.6%	60.93%	23.44%
5	68.97%	20.61%	45.73%	96.35%	39.06%

where there is additional latency due to the Gaussian white noise generator. The corresponding resource utilizations of the two implementations are shown in tables 8 and 9. The block RAM (BRAM) memory banks available for the Virtex 4 device family are each of size 18Kb, which is much higher than what is required for any of the implementations. Increasing P affects only the required memory bank sizes, thus the resource utilization remains the same for different numbers of particles. However, for applications with larger memory requirements, this would not be the case. Note that the execution times for both of the applications are similar because the latencies of the PEs are relatively small compared to the latency induced by P.

For the 3D facial pose tracking implementation the FPGA resources allowed the implementation of only a 2PE system, the resource requirements and execution results (per frame) for varying values of P are shown in tables 10 and 11 respectively.

Table 9. FPGA resource utilization for fault detection system implementation.

Uni-dimensional failure prognosis model implementation

No. of PEs	Slices	Slice Flip-Flops	4 input LUTs	DSP48s	BRAMs
2	19.81%	6.13%	14.27%	43.23%	15.63%
3	28.39%	8.94%	20.18%	60.93%	23.44%
5	45.57%	14.57%	31.98%	96.35%	39.06%

Table 10. FPGA resource utilization for 3D facial pose tracking system implementation.

No. of PEs	Slices	Slice Flip-Flops	4 input LUTs	DSP48s	BRAMs
2	87.57%	42.26%	41.03%	66.67%	96.35%

Table 11. Execution time (per frame) variation with total particles for a 2PE implementation of 3D facial pose tracking system .

No. of particles (N)	Execution time per frame (in ms)
50	36.855
100	73.72
200	143.94
300	215.33

Chapter 7 : Conclusions and Future Work

7.1 Conclusion

In this thesis, various design tools and methodologies to aid in the design of a complex embedded implementation of computer vision systems have been presented. Experiments with a wide range of applications have demonstrated that the tools are simple yet powerful, and help to improve the efficiency of targeted implementations. Exploration on a variety of platforms has been carried out to demonstrate the robustness of the tools.

In chapter 3, high-level transformation techniques for an important and popular computation model for DSP applications — dataflow graphs — were explored. Traditional transformations have been limited largely to SDF modeling only. However, SDF modeling is restricted in terms of limited expressibility as it does not allow dynamic reconfigurability in applications. Of the various modeling tools that are based on SDF with enhanced capabilities to allow such dynamic behavior, two very important ones namely PSDF and CSDF were explored. Details of a new transformation technique for PCSDF graphs have been presented. Experimental results with typical signal processing application demonstrated that PCSDF modeling based implementation using the new transformation technique results in a much more optimized realization of dynamic systems compared to SDF and PSDF.

A simple, intuitive and robust architectural design space exploration methodology has been presented in chapter 4. The methodology generalizes the synchronization

graph modeling technique and applies it as a conceptual tool for design. The technique considers synchronization graph modeling for multidimensional signals, and integrates the methods of self-timed and ordered-transaction scheduling. The practical utility of the methodology has been shown by applying it to two computer vision applications — face detection and 3D facial pose tracking — and a variety of target platforms.

In chapter 5 a new, flexible and optimized communication interface (SPI) has been presented for multiprocessor signal processing systems. SPI achieves significant streamlining and associated advantages by integrating with the MPI framework properties of dataflow graph modeling, synchronous dataflow analysis and self-timed scheduling.

The novel concept of applying variable token sizes for dynamic data-rate transactions between different actors in a data flow graph has been presented, analyzed, and demonstrated. Given an upper bound on the associated data rate variation, the scheme can handle dynamic data-rate behavior through an enclosing framework of efficient, SDF-based analysis. Capability to support dynamic data rates between different subsystems for SPI has been added by thorough integration of this concept into the SPI framework. Resynchronization for distributed embedded systems in the context of SPI has also been explored, and related synchronization optimizations have been added to the interface.

Two communication libraries for SPI have been created and demonstrated on two useful signal processing applications. This work demonstrates the capability of SPI to provide a standard, low-cost, and modular message passing interface with careful

streamlining for signal processing applications, and with efficient separation between communication and computation for easier development of embedded multiprocessor implementations.

Chapter 6 presents a new methodology for design, modeling and exploration of particle filters on reconfigurable System-on-chips (SoCs). The methodology uses the notion of parameterization to provide a useful tool for. evaluating multiple design alternatives, and exploring the associated trade-offs in an efficient and intuitive manner. It also provides scope for implementing a wide range of applications with minimal redesign effort between different applications. For all the applications that we examined, the execution speed was determined mainly by the number of particles, and thus, the latency of the resampling unit played a significant role in determining the overall execution time. Although multiple expensive (area consuming) computational units were used, the area constraint imposed by the target platform was met in each case.

7.2 Future Work

Further exploration is possible for the tools presented in this thesis. In this section, some of the possibilities have been outlined.

Exploration of quasi-static scheduling strategies as well as other scheduling algorithms for multiprocessor implementation would be a valuable addition to the transformation and scheduling strategies for the CSDF and PCSDF graphs developed in this thesis (chapter 3). The current work utilizes a DRSP transformation that results in a single transformed actor. Extending DRSP transformation such that multiple new transformed actors are created is also possible and can lead to interesting directions for

multiprocessor implementation.

For the design space exploration methodology outlined in chapter 4, experiments with other platforms as well as more applications can be carried out to verify the robustness of this methodology. Also, it was observed that lack of proper modeling of synchronization actors resulted in discrepancies between estimated and experimental values, therefore developing better models to capture the synchronization aspects of targeted systems is an important direction for future work.

Platform-specific libraries for SPI would aid in increasing the utility of this communication interface. Also, a standalone development of a software implementation of the interface would be very useful. In addition, since the interface uses SDF as an underlying model, various other optimizations can be integrated with the methodology in a manner similar to the integration of resynchronization, as shown in chapter 5.

The range of applications currently handled by the parameterized design framework shown in chapter 6 can be further extended. Also, more in-depth analysis of parameterizable features of other applications would aid in obtaining an optimized implementation.

BIBLIOGRAPHY

[1] M. Ade, R. Lauwereins, and J. A. Peperstraete, “Data memory minimization for synchronous data flow graphs emulated on DSP-FPGA targets,” In Proceedings of Design Automation Conference, June 1994.

[2] G. Aggarwal, A. Veeraraghavan, and R. Chellappa. 3D Facial pose tracking in uncalibrated videos. In *Proc. of Intl. Conf. on Pattern Recognition and Machine Intelligence (PReMI)*, 2005.

[3] A. Athalye, M. Bolic, S. Hong and P. M. Djuric. Generic hardware architectures for sampling and resampling in particle filters. In *EURASIP Journal on Applied Signal Processing*, Vol. 2005, pp.2888-2902.

[4] M. Auguin, L. Capella, F. Cuesta and E. Gresset. CODEF: a system level design space exploration tool. In *Proc. of IEEE International Conference on Acoustics, Speech, and Signal Processing*, Volume 2, 7-11 May 2001 Page(s):1145 - 1148 vol.2.

[5] A. S. Bashi, V. P. Jilkov, X. R. Li and H. Chen. Distributed implementations of particle filters. In *Proc. Sixth International Conference of Information Fusion*, New Orleans, pp. 1164-1171.

[6] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling of DSP systems. In *Proc. of the International Conference on Acoustics, Speech, and Signal Processing,* pages 1948-1951, Istanbul, Turkey, June 2000

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

[8] S. S. Bhattacharyya and E. A. Lee. Looped schedules for dataflow descriptions of multirate signal processing algorithms. *Jnl. of Formal Methods in System Design*, pages. 183–205, Dec. 1994.

[9] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Optimal parenthesization of lexical orderings for DSP block diagrams. In *Proc. of the International Workshop on VLSI Signal Processing*, October 1995. Sakai, Osaka, Japan, pages 177-186.

[10] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Boston, MA: Kluwer, 1996.

[11] 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.

[12] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Static scheduling of multi-rate and cyclo-static DSP applications. In *Workshop on VLSI Signal Processing*, 1994, pages. 137-146.

[13] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-Static dataflow. In *IEEE Transactions on Signal Processing*, Vol. 44, No.2, February 1996, pages 397-408.

[14] M. Bolic. Architectures for efficient implementation of particle filters. *Ph.D. dissertation*, Stony Brook University, New York, Aug 2004.

[15] J. T. Buck and E. A. Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *Proc. of Intl. Conf. on Acoustics, Speech, Signal Processing*, April 1993.

[16] B. P Carlin, N. G. Polson and D. S Stoffer. A Monte-Carlo approach to nonnormal and nonlinear state space modelling. *Journal of American Statistical Association*, 1992, pages 493-500.

[17] William W. Carlson, et al. Introduction to UPC and language specification. *CCS-TR-99-157*.

[18] R. Camposano. From behavior to structure: High-level synthesis. In *Proc. of IEEE Design and Test in Europe*, Oct. 1990, vol. 7, pages 8-19.

[19] V. Chaiyakul, D. D Gajski and L. Ramachandran. High-level transformations for minimizing syntactic variances. In *Proc. of 30th Conference on Design Automation*, Jun. 1993, pages 413-418.

[20] A. Chirila-Rus, K. Denolf, B. Vanhoof, P. R. Schumacher and K. A. Vissers. Communication primitives driven hardware design and test methodology applied on complex video applications. In *Proc. of IEEE Intl. Wkshp. on Rapid System Prototyping*, 2005, pages 246-249.

[21] P. P. Chu and R. E. Jones. Design techniques of FPGA based random number generator. In *Proc. Military and Aerospace Applications of Programmable Devices and Technologies Conference*, The Johns Hopkins University- Applied Physics Laboratory, Sept. 1999.

[22] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1), 1998, pages 46-55.

[23] E. F. Deprettere, T. Stefanov, S. S. Bhattacharyya, and M. Sen. Affine nested loop programs and their binary cyclo-static dataflow counterparts. In *Proc. of the Intl. Conf. on Application Specific Systems, Architectures, and Processors*, Steamboat Springs, Colorado, Sept. 2006.

[24] K. Denolf, M. Bekooij, J. Cockx, D. Verkest and H. Corporaal. Exploiting the expressiveness of cyclo-static dataflow to model multimedia implementations. *EURASIP Journal on Advances in Signal Processing*, 2007, Article ID 84078, 14 pages, doi:10.1155/2007/84078.

[25] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker. A message passing standard for MPP and workstations. *Communications of the ACM*, 39(7), 1996, page 84-90.

[26] M. Dubois and F. A. Briggs. Efficient interprocessor communication for MIMD multiprocessor systems. In *Proc. of the 8th annual symposium on Computer Architecture*, 1981, pages 187 - 196.

[27] M. Engels, G. Bilsen, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow: model and implementation. In *Asilomar Conf. on signals, Systems and Computers*, Pacific Grove, California, October 1994.

[28] S. Fischaber, R. Woods and J. McAllister. SoC Memory hierarchy derivation from dataflow Graphs. In *Proc. of the IEEE Workshop on Signal Processing Systems*, Oct. 2007.

[29] B. Franke and M. O'Boyle. An empirical evaluation of high level transformations for embedded processors. In *Proc. of Intl. Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, Nov. 2001

[30] B. Franke and M. O'Boyle. Array recovery and high-level transformations for DSP applications. In *ACM TECS*, May 2003, vol. 2, pages 132–162.

[31] G. R. Gao, R. Govindarajan, and P. Panangaden. Well-behaved dataflow for DSP computation. In *Proc. of IEEE Intl. Conference on Acoustics, Speech, Signal Processing*, San Francisco, March, 1992.

[32] M. Geilen and T. Basten. Reactive process networks. In *Proc. of the International Workshop on Embedded Software*, Sept. 2004, pp. 137-146.

[33] C. Gong, R. Melhem, and R. Gupta. Loop transformations for fault detection in regular loops on massively parallel system. *IEEE Trans. on Parallel and Distributed Systems*, Dec. 1996, vol. 7, pages 1238 - 1249.

[34] G. Hendeby, J. Hol, R. Karlsson and F. Gustafsson. A graphics processing unit implementation of the particle filter. In *Proc. of EUSIPCO*, 2007.

[35] T. Henriksson and P. van der Wolf. TTL hardware interface: a high-level interface for streaming multiprocessor architectures. In *Proc. of the IEEE Wkshp. on Embedded Systems for Real-Time Multimedia*, pages 127-132, Seoul, Korea, October 2006.

[36] D. S. Henry. Hardware mechanisms for efficient interprocessor communication. *PhD Thesis*, MIT, 1996.

[37] P. Hoang and J. Rabaey. A compiler for multiprocessor DSP implementation. In *Proc. of Intl. Conf. on Acoustics, Speech, Signal Processing*, March 1992.

[38] P. D. Hoang and J. M. Rabaey. Scheduling of DSP programs onto multiprocessors for maximum throughput. *IEEE Trans. on Acoustics, Speech, Signal Processing*, June 1993, vol. 41, p.ages 2225–2235.

[39] S. Hong, X. Liang, P. M. Djuric. Reconfigurable particle filter design using dataflow structure translation. In *Proc. of IEEE Workshop on Signal Processing Systems*, 2004, pages 325-330.

[40] I. Karkowski and H.Corporaal. Design space exploration algorithm for heterogeneous multi-processor embedded system design. In *Proc.of Design Automation Conference*, 15-19 Jun 1998, pages 82 - 87

[41] J. Keinert, C. Haubelt, J. Teich. Modeling and analysis of windowed synchronous algorithms. In *Proc. of the 31st International Conference on Acoustics, Speech, and Signal Processing*, Toulouse (France) May 14-19, 2006

[42] M. Ko, P. K. Murthy, and S. S. Bhattacharyya. Compact procedural implementation in DSP software synthesis through recursive graph decomposition. In P*roc. of the Intl. Workshop on Software and Compilers for Embedded Systems*, Amsterdam, The Netherlands, September 2004, pages 47-61.

[43] M. Ko, C. Zissulescu, S. Puthenpurayil, S. S. Bhattacharyya, B. Kienhuis, and E. Deprettere. Parameterized looped schedules for compact representation of execution sequences. In *Proc. of the Intl. Conf. on Application Specific Systems, Architectures, and Processors*, Steamboat Springs, Colorado, Sept. 2006.

[44] C. Kwok, D. Fox and M. Meila. Real-time particle filters. *Proc. of the IEEE*, Vol. 92, March 2004, pages 469 - 484

[45] S. Kwon, C. Lee, S. Kim, Y. Yi and S. Ha. Fast design space exploration framework with an efficient performance estimation technique. In *Proc. of 2nd Workshop on Embedded Systems for Real-Time Multimedia*, 2004, pages 27 - 32.

[46] P. Lapsley, J. Bier, A. Shoham, and E. A. Lee. *DSP Processor Fundamentals*. Berkeley Design Technology, Inc., 1994.

[47] E. A. Lee and S. Ha. Scheduling strategies for multiprocessor real-time DSP. In *Proc. of IEEE Global Telecommunications Conf. and Exhibition*, pages 1279–1283, Nov 1989.

[48] E.A. Lee and D.G. Messerschmitt. Static scheduling of synchronous dataflow programs for digital signal processing. *IEEE Transactions on Computers*,Vol. C-36, No. 2, February, 1987.

[49] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing synchronous circuitry by retiming. In *Proc. of CalTech Conf. on VLSI*, 1983.

[50] P. Marwedel. Embedded software: how to make it efficient. In *Proc. of the Euromico Symp. on Digital System Design*, Sept. 2002, pages 201– 207.

[51] B. Miramond and J-M. Delosme. Design space exploration for dynamically reconfigurable architectures. In *Proc. of Design Automation and Test in Europe*, 2005, pages. 366-371.

[52] H. Moon, R. Chellappa, and A. Rosenfeld,. Optimal edge-based shape detection. *IEEE Transaction on Image Processing*, Vol. 11, pages 1209-1227, 2002.

[53] P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee. Joint minimization of code and data for synchronous dataflow programs. *Journal of Formal Methods in System Design*, Vol. 11(1), pages 41-70, July 1997.

[54] P. K. Murthy and S. S. Bhattacharyya. Shared buffer implementations of signal processing systems using lifetime analysis techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 20(2), pages 177-198, February 2001.

[55] P. K. Murthy and S. S. Bhattacharyya. *Memory management for synthesis of DSP software*. CRC Press, 2006.

[56] P.K. Murthy and E.A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, Vol. 50, no. 8, pages 2064-2079, July 2002.

[57] M. Orchard, B. Wu and G. Vachtsevanos. A particle filter framework for failure prognosis. In *Proc. WTC2005 World Tribology Congress III*. Washington D.C., Sept., 2005.

[58] M. Pankert, O. Mauss, S. Ritz, H. Meyr. Dynamic data flow and control flow in high level DSP code synthesis. In *Proc. of the IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, Vol. 2, pages 449-452, Adelaide, Australia, April, 1994.

[59] K. K. Parhi. High-level algorithm and architecture transformations for DSP synthesis. *Journal of VLSI Signal Processing*, Jan. 1995.

[60] T. M. Parks, J. L. Pino, and E. A. Lee. A comparison of synchronous and cyclo-static sataflow. In *Proc. of IEEE Asilomar Conf. on Signals, Systems, and Computers*, Pacific Grove, CA, Oct. 29 - Nov. 1, 1995.

[61] H. P. Peixoto and M. F. Jacome. Algorithm and architecture-level design space exploration using hierarchical data flows. In *Proc. of IEEE Intl. Conference on Application-Specific Systems, Architectures and Processors*, 14-16 July 1997, pages 272 - 282.

[62] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee. A hierarchical multiprocessor scheduling system for DSP applications. In *Proc. of the IEEE Asilomar Conf. on Signals, Systems, and Computers*, Nov. 1995, vol.1, pages 122–126.

[63] M. Rim and R. Jain. Valid transformations: a new class of loop transformations for high-level synthesis and pipelined scheduling applications. *IEEE Trans. on Parallel and Distributed Systems*, April 1996, vol. 7, pages 399-410.

[64] S. Ritz, M. Pankert, V. Zivojnovic, and H. Meyr. Optimum vectorization of scalable synchronous dataflow graphs. In *Proc. of Intl. Conf. on Application-Specific Army Processors*, 1993, pages 285-296.

[65] S. Saha, V. Kianzad, J. Schessman, G. Aggarwal, S. S. Bhattacharyya, W. Wolf and R. Chellappa. An architectural level design methodology for smart camera applications. To appear in *Intl. Journal of Embedded Systems, Invited paper for Special Issue on Optimizations for DSP and Embedded Systems*.

[66] S. Saha, S. Puthenpurayil, and S. S. Bhattacharyya. Dataflow transformations in high-level DSP system design. In *Proc. of the Intl. Symp. on System-on-Chip*, pages 131-136, Tampere, Finland, November 2006.

[67] S. Saha, C. Shen, C. Hsu, A. Veeraraghavan, A. Sussman, and S. S. Bhattacharyya. Model-based OpenMP implementation of a 3D facial pose tracking system. In *Proc. of the Workshop on Parallel and Distributed Multimedia*, pages 66-73, Columbus, Aug. 2006.

[68] S. Sakai, H. Matsuoka, Y. Kodama, M. Sato, A. Shaw, H. Hirono, K. Okamoto and T. Yokota. RICA: Reduced Interprocessor-Communication Architecture -conceptand mechanisms. In *Proc. of the Fifth IEEE Symposium on Parallel and Distributed Processing*, Dec. 1-4, 1993, pages 122 - 125.

[69] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, 1989.

[70] A. C. Sankaranarayanan, R. Chellappa and A. Srivastava. Algorithmic and architectural design methodology for particle filters in hardware. In *Proc. of IEEE International Conference on VLSI in Computers and Processors*, Oct.2005, pages 275 - 280.

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

[72] S. Stuijk, M. Geilen, and T. Basten. Exploring tradeoffs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Proc. of the Design Automation Conference,* July 2006.

[73] V.S. Sunderam, G.A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: evolution, experiences, and trends. *Parallel Computing*, Vol. 20(4), 1994, pages 531-545.

[74] R. Velmurugan, S. Subramanian, V. Cevher, D. Abramson, K. M. Odame, J. D. Gray, H.-J. Lo, J. H. McClellan and D. V. Anderson. On low-power analog implementations of particle filters for target trackin. In *Proc. of EUSIPCO* 2006, Italy, Sept.

[75] R. Velmurugan, S. Subramanian, V. Cevher, J. H. McClellan and D. V. Anderson. Mixed-mode implementation of particle filters. In *Proc. of IEEE Pacific Rim Conference on Communications, Computers and Signal Processing,* Aug. 2007, pages 617-620.

[76] A. J. van der Wal and G. J. W. Dijk. Efficient interprocessor sommunication in a tightly-coupled homogeneous multiprocessor system. In *Proc. of Second IEEE Wkshp. on Future Trends of Distributed Computing Systems*, 1990, pages 362 - 368.

[77] D. Ziegenbein, R. Ernest, K. Richter, J. Teich, L. Thiele. Combining multiple models of computation for scheduling and allocation. In *Proc. of Codes/CASHE 1998*, pages. 9-13.

[78] V. Zivojnovic, S. Ritz and H. Meyr. High performance DSP software using data-flow graph transformations. In *Proc. of the IEEE Asilomar Conf. on Signals, Systems, and Computers*, 1994.