ANALYSIS OF FIELD PROGRAMMABLE GATE ARRAY-BASED KALMAN FILTER ARCHITECTURES

by

Arvind Sudarsanam

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

DOCTOR OF PHILOSOPHY

in

Electrical Engineering

Approved:

Dr. Aravind Dasu Major Professor	Dr. Brandon Eames Committee Member
Dr. Edmund Spencer Committee Member	Dr. Stephen Allan Committee Member
Dr. David Geller Committee Member	Dr. Byron R. Burnham Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2010

Abstract

Analysis of Field Programmable Gate Array-Based Kalman Filter Architectures

by

Arvind Sudarsanam, Doctor of Philosophy

Utah State University, 2010

Major Professor: Dr. Aravind Dasu
Department: Electrical and Computer Engineering

A Field Programmable Gate Array (FPGA)-based Polymorphic Faddeev Systolic Array (PolyFSA) architecture is proposed to accelerate an Extended Kalman Filter (EKF) algorithm. A system architecture comprising a software processor as the host processor, a hardware controller, a cache-based memory sub-system, and the proposed PolyFSA as co-processor, is presented. PolyFSA-based system architecture is implemented on a Xilinx Virtex 4 family of FPGAs. Results indicate significant speed-ups for the proposed architecture when compared against a space-based software processor. This dissertation proposes a comprehensive architecture analysis that is comprised of (i) error analysis, (ii) performance analysis, and (iii) area analysis. Results are presented in the form of 2-D pareto plots (area versus error, area versus time) and a 3-D plot (area versus time versus error). These plots indicate area savings obtained by varying any design constraints for the PolyFSA architecture. The proposed performance model can be reused to estimate the execution time of EKF on other conventional hardware architectures. In this dissertation, the performance of the proposed PolyFSA is compared against the performance of two conventional hardware architectures. The proposed architecture outperforms the other two in most test cases.

(123 pages)

Dedicated to my mom and dad,
Rama and Sudarsanam.

Acknowledgments

I would like to thank my advisor, Dr. Aravind Dasu, for his continued guidance and support during the course of this dissertation. Dr. Dasu's vision and his constructive criticisms have helped me accomplish my goals. I would also like to thank Dr. Brandon Eames, Dr. Edmund Spencer, Dr. Stephen Allan, and Dr. David Geller for serving on my committee, and providing significant support. I thank Dr. Thomas Hauser for helping me to appreciate the application of FPGAs in the high-performance computing domain. I am grateful to my colleagues in the Reconfigurable Computing Group - Jonathan, Rob, Seth, Hari, Shant, Ram, Abe, Jeff, and Varun - for their interactions and all the brain-storming sessions. Also, special thanks to my friends Rohit, Shantanu, Netra, Prasad, Anand, Sai, Vignesh, Jaya, Smita, and the others who have made my stay here memorable. I would also like to acknowledge Starbridge Systems, Lockheed Martin, NASA, Micron, Intel, and the Department of ECE for having supported me financially at different times in my dissertation research.

Words cannot express my deepest gratitude to my family: my father Sudarsanam, mom Rama, brother Murali, and others who motivated me to take up this challenge and helped me complete it with their love and support.

Arvind Sudarsanam

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Spacecraft Navigation and Kalman Filters

Recent and future space missions involve complex objectives like exploration into deep space [1], interplanetary orbit determination [2], and asteroid rendezvous [3], which reduce the ability to constantly communicate between earth (ground stations) and spacecrafts. In such cases, spacecraft navigation and control systems are expected to be autonomous from time to time, while longer term objectives and commands can be sent from earth. Navigation algorithms involve determination of state (position, velocity, and attitude) of the spacecraft using external or internal measurements. Figure 1.1 shows a high-level description of the navigation and control system. It can be seen that the spacecraft state is hidden and is manifested in the form of observed measurements.

Due to stochastic nature of the overall system (caused by system errors and measurement errors), there is a need for an optimal stochastic state estimator filter. Kalman filters [4] are predominantly used in such spacecraft missions. Invented by R. E. Kalman, this filter estimates the current state of a system as a linear or nonlinear function of previous state estimate and reduces the error in estimation process by using a sequence of measurements. It involves a set of computationally complex linear algebra operations and the complexity is directly proportional to the number of states and number of measurements. Also, the complexity depends on the flavor of Kalman filter used. Extended Kalman Filters (EKF), that support nonlinear systems, are more compute-intensive than linear Kalman filters and are predominantly being used in ongoing missions.

The remainder of this chapter discusses the motivation behind the proposed research, thesis contributions, and an overview of this report.

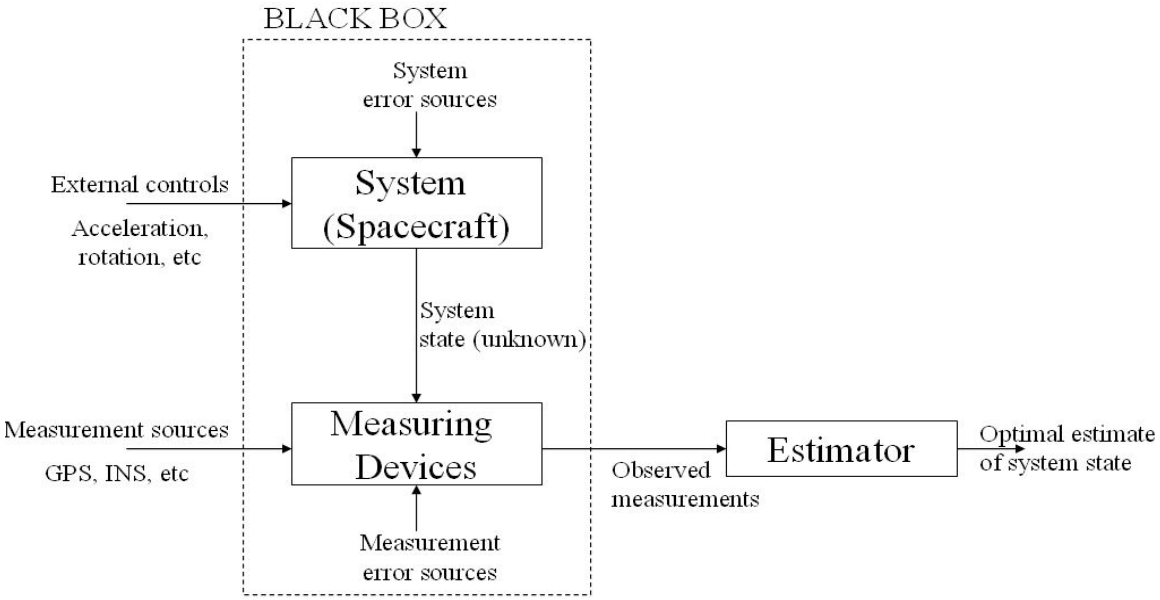


Fig. 1.1: Model of a navigation and control system.

1.2 Motivation and Thesis Contributions

In addition to autonomous navigation, space computers are required to perform a multitude of compute-intensive tasks that include event scheduling and processing of scientific data. During launch, re-entry and landing phases of the space missions, the navigation process becomes time-critical and execution of Kalman filters needs to obey tight timing constraints. During the remainder of the mission, the execution of Kalman filters is less critical and resources are required by other compute-intensive tasks. There is a need for a polymorphic architecture that can be reconfigured during run-time, so that a variable sub-section of the available computing resources can be dedicated to the processing of Kalman filters. For such requirements, Field Programmable Gate Arrays (FPGAs) are better equipped as target platforms than microprocessors and Application Specific Integrated Circuits (ASICs) since they have the electronic fabric to support polymorphic and run-time reconfigurable circuits and are capable of high speeds and real time performance.

In this dissertation, a 1-D Polymorphic Faddeev Systolic Array (PolyFSA) is proposed as the architectural template to accelerate the execution of the compute-intensive kernels

of Kalman filter. Number of nodes, internal design of nodes, inter-node communication, and communication between memory and nodes of the systolic array are some of the design parameters that can be configured. A polymorphic architectural template provides the spacecraft engineer with numerous design options, so that the engineer can program the template to realize an instance of the architecture (or multiple instances) that will meet the design goals in terms of time, area, power, and quantization error. While the spacecraft engineer is aware of his objectives, the means to achieve these objectives are unclear and complicated. There is a need for the architectural template to be accompanied by a comprehensive analysis of the various design options and the resulting design performance. This dissertation research presents such an analysis and results are presented in terms of 2-D (area versus error, area versus execution time) and 3-D (error versus execution time versus area) pareto curves. While the combinatorial analysis of the design goals is interesting to the spacecraft engineer, the design options may be hidden. Based on the requirements, the engineer can opt for a required design point in the pareto curves.

In the past decade, hardware developers have started focusing on variable precision arithmetic as an alternative to the IEEE-754 standard floating-point arithmetic and numerous other fixed-point arithmetic options. FPGAs provide an effective platform to evaluate such an option and various FPGA-based libraries are available to support the design of variable precision arithmetic operations. Conventional FPGA designs are analyzed in terms of three important factors: performance, area, and power. Use of variable precision arithmetic introduces yet another metric, Quantization error (or simply error). This error depends on two major factors: (i) reduction in precision, and (ii) application characteristics. During deep-space missions, there are several time intervals when the estimation using Kalman filters needs to be accurate (zero error in computations) and several time intervals when a specific amount of error can be tolerated. During the latter intervals, the polymorphic architecture may be reconfigured to operate using lower precision arithmetic, and any savings in resources can be re-used to target other applications. To derive an architecture with reduced precision, an application-specific approach is required to analyze the error

generated in the results of Kalman filter estimation due to reduced precision. The need for an application-specific approach is two-pronged.

- Kalman filters are error-correcting filters, and some of the quantization error may be inherently corrected.
- Specific low-complexity portions of Kalman filters are computed on a full-precision embedded processor. Such computations may compensate for some of the quantization error that is introduced by the low-precision computation units in the co-processor architecture.

For an overall analysis of the architecture, execution time and area also need to be estimated. Execution time is estimated using a simulation model of the architectural template. Area is estimated by identifying the functional units in the architecture and using the extensive Xilinx Core Generator library to obtain area requirement for a specific implementation of the functional unit, and eventually combining the area requirements of all functional units.

The following is a list of contributions of this dissertation research.

- An FPGA-based Polymorphic Faddeev Systolic Array (PolyFSA) architecture is proposed to accelerate the compute-intensive kernels of Kalman filters. This architecture acts as a co-processor to the embedded processor (PowerPC or MicroBlaze) and a pseudo-cache and hardware controller are used for communication and control. Results are provided to analyze the impact of such acceleration on overall performance and area requirements.
- Hierarchical analysis of the error introduced in results of Kalman filter computations due to reduction in precision is presented.
- A simulation model to estimate the overall execution time of the Kalman filter algorithm is proposed.
- Results of architecture analysis are presented in terms of pareto curves.

1.3 Overview of the Report

This dissertation report presents the derivation and analysis of a polymorphic systolic array architecture used for accelerating Kalman filters. Chapter 2 presents an overview of some of the fundamental concepts underlying the proposed research, namely Kalman filters, Systolic arrays, Faddeev algorithm, and FPGAs. Chapter 3 reviews the related work targeted towards acceleration of Kalman filters and linear algebra operations in general. This chapter also presents a comprehensive review of the literature in the domain of error and precision analysis. Furthermore, the chapter includes a survey of recent efforts towards performance and area modeling. Chapter 4 discusses the derivation of proposed PolyFSA and outlines the overall design methodology. Chapter 5 presents the proposed analysis of PolyFSA. Discussion of hierarchical error analysis for Kalman filter is followed by the discussion of the simulation model used to estimate performance. The chapter concludes by presenting the details of area analysis. Chapter 6 presents the results used to evaluate the proposed architecture. Also, a 3-D Pareto curve (area versus performance versus error) is presented. Chapter 7 concludes the report and provides directions for future research.

Chapter 2

Background

This chapter presents an overview of some of the fundamental concepts underlying the proposed research, namely Kalman filters, Systolic arrays, Faddeev algorithm, and FPGAs.

2.1 Kalman Filters

For a given physical system (e.g., spacecraft), optimal estimation of a state variable (e.g., spacecraft position) that cannot be measured directly is a complex problem and has been addressed by Carl Gauss, in the year 1795. He designed an estimator that is based on the least square error that is illustrated in fig. 2.1. In this figure, x is the value to be estimated using the measurement y.

Kalman filters were first proposed by Rudolf Kalman in the year 1960 [4] as optimal estimation filters for linear systems. The system is specified in the form of a mathematical model that can be used to represent all the deterministic, random, and time-variant properties of the system. In this system, the state variable is a vector that consists of (i) position of the spacecraft, (ii) velocity of the spacecraft, and (iii) attitude or orientation of the spacecraft (roll, pitch, and yaw). Measurements may include one or more of the following (i) measurements from Global Positioning System (GPS), (ii) measurements from star/sun sensors, and (iii) inertial navigation measurements. Figure 2.2 shows an example of a mathematical model for a spacecraft system. Based on the spacecraft dynamics and its mission objectives, space scientists develop a probabilistic model that represents the current state of a spacecraft in terms of previous state and current set of measurements. Kalman filters use this probabilistic model to determine the best possible estimate for the current state and also update the probabilistic model as new measurements are added. The process of state estimation without new measurements is termed the "predict" phase and

$$y = F(x) + \epsilon$$

$$|\epsilon|^2 = [y - F(x)][y - F(x)]^T$$

$$x_{est} = x \text{ for minimum } \epsilon$$

Fig. 2.1: Gaussian estimation that is based on least square error.

PROCESS MODEL : $x_k = Ax_{k-1} + Buk + w_{k-1}$

MEASUREMENT MODEL: $z_k = Hx_k + v_k$

x_k	→	State vector at time 'k'
x_{k-1}	→	State vector at time 'k-1'
uk	→	Input control vector at time 'k'
z_k	→	Measurement at time 'k'
A	→	System transfer model
B	→	Control system function model
H	→	Measurement transfer model

$w \sim N(0,Q)$

$v \sim N(0,R)$

Fig. 2.2: Mathematical model of spacecraft system.

the process of state estimation with new measurements is termed the "update" phase. Each phase is comprised of a task flow of linear algebra modules that operate on 1-D and/or 2-D data. Figure 2.3 shows this task flow for linear Kalman filter.

Spacecraft scientists tend to develop the mathematical model of the spacecraft using as many state and measurement variables as possible, in order to have total control over the spacecraft dynamics. During launch, re-entry, and landing phases of the space missions, execution of this mathematical model becomes time-critical and needs to obey tight timing constraints. Also, this model requires a computationally complex implementation of Kalman filters comprising of linear algebra modules like matrix inversion and matrix multiplication, and a sequential software-based solution may be too slow to realize such a real-time Kalman filter. There is an increasing need for fast and parallel implementations that can be used to accelerate the estimation process. Chapter 3 discusses the existing efforts towards realizing

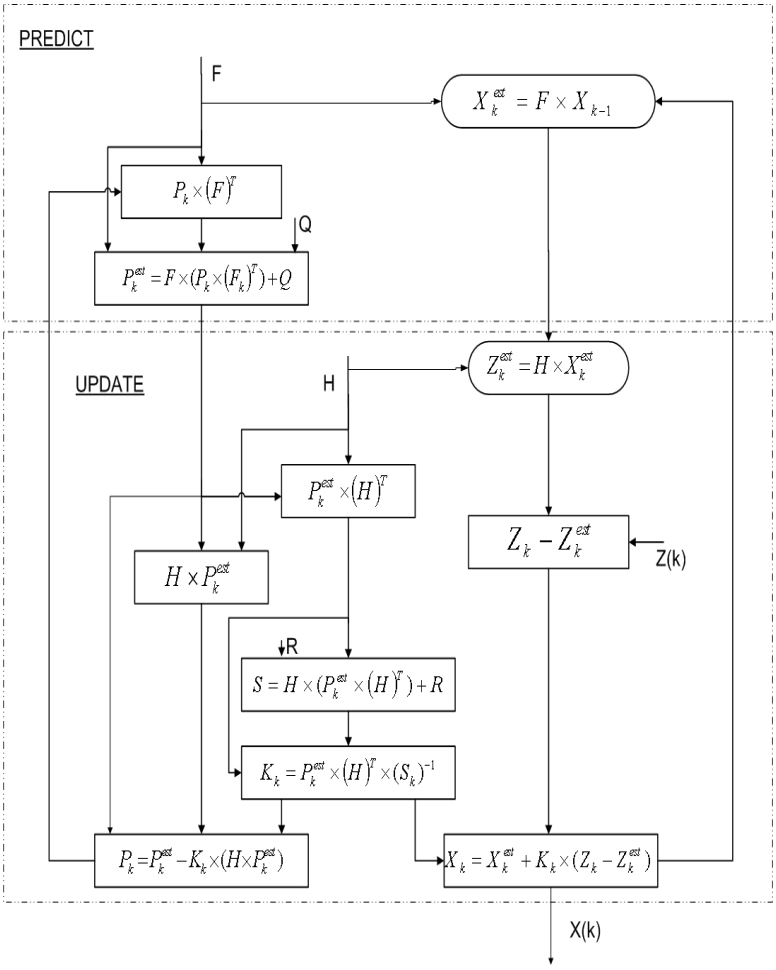


Fig. 2.3: Task flow of linear Kalman filter.

accelerators for Kalman filters and linear algebra operations, in general.

2.2 Systolic Arrays

In the year 1978, H. T. Kung and Charles E. Leiserson published the first paper describing systolic arrays [5]. As shown in fig. 2.4, systolic array is a pipelined network of Data Processing Units (DPU) with the following properties.

- Modularity and regularity - Each DPU performs a simple mathematical operation and all DPUs are inter-connected using a regular network pattern.

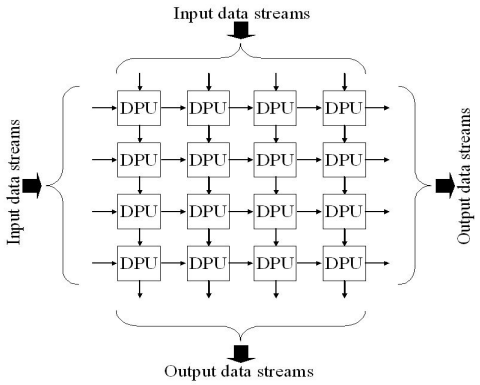


Fig. 2.4: Example of a 4x4 systolic array architecture.

- Spatial and temporal locality - A DPU communicates only with neighboring DPU. Data can be transferred from one DPU to another in one time unit.
- Synchronous - Data passes through the array in a rhythmic pattern synchronized by a global clock.
- Pipelined - Each data element is fetched once then "pumped" between DPUs.

A commonly used technique to realize systolic Kalman filters is based on the Faddeev algorithm [6]. An overview of this algorithm is presented in the next section.

2.3 The Faddeev Algorithm

The Faddeev algorithm is a popular method for computing the Schur complement (E) for a set of four input matrices (A, B, C, D) according to the following equation:

$$E = D + CA^{-1}B. \tag{2.1}$$

Given two or more matrices, any of the three matrix operations (matrix inverse, multiplication, and addition) or a combination of the three operations can be performed by appropriately assigning the given matrices to the input matrices of the Faddeev algorithm. For instance, to add two matrices X and Y, A must be assigned to an identity matrix, B

assigned to X, C assigned to an identity matrix and D assigned to Y. In this example, the Schur complement resolves to

$$E = Y + (I \times I^{-1}) \times X = Y + X. \tag{2.2}$$

The Schur complement can be implemented in a systolic array structure, which is comprised of two types of Processing Elements (PEs), namely boundary PE and internal PE. Figure 2.5 shows the different types of matrix operations that can be realized and fig. 2.6 shows the systolic array architecture to operate on matrices of different sizes. This systolic Kalman filter architecture is comprised of $2 \times (N^2)$ processing nodes where N is the order of the matrix. Existing research efforts [6–8] result in large hardware designs that may not fit in a single FPGA and fail to generate scalable designs.

2.4 Field Programmable Gate Arrays

A Field Programmable Gate Array (FPGA) is a general-purpose integrated circuit that can be configured by the designer rather than the device manufacturer. Unlike an Application Specific Integrated Circuit (ASIC), an FPGA can be reprogrammed, even after it has been deployed into a system. Figure 2.7 shows the top-level representation of an FPGA. An FPGA consists of the following blocks:

$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ Reduce to row echelon form and D-CA^{-1}B will result in the lower right corner

Operation	A	B	C	D
YW^{-1}X + Z	W	X	-Y	Z
X^{-1}	X	I	I	0
X*Y	I	X	-Y	0
X – Y	I	I	Y	X
X + Y	I	I	-X	Y

Fig. 2.5: Illustration of the Schur complement and the Faddeev algorithm.

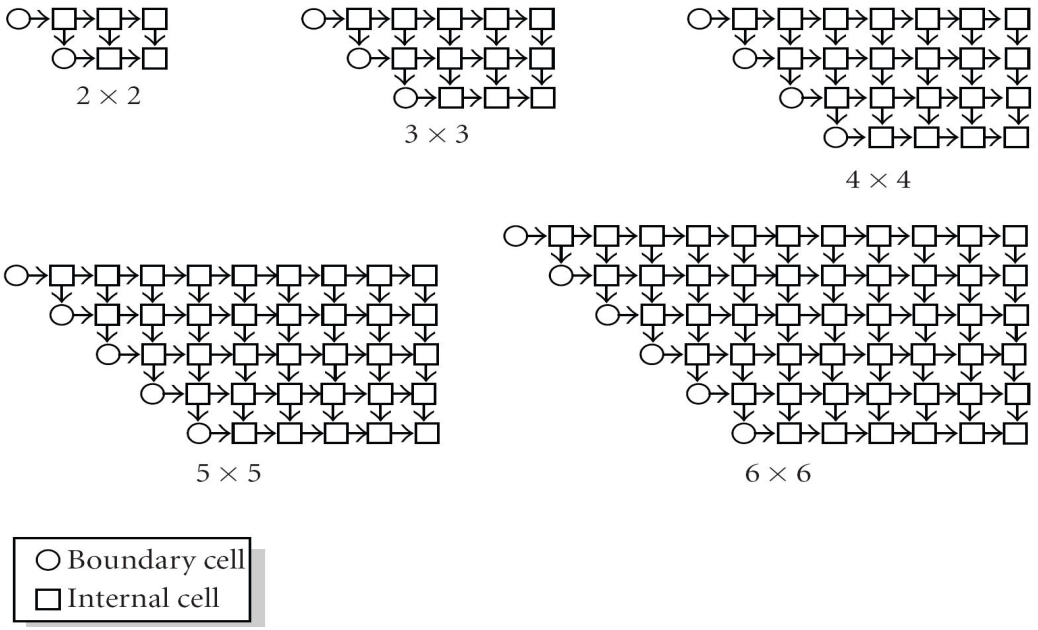


Fig. 2.6: Realization of the Faddeev algorithm using systolic arrays.

- Logic Elements (LE),
- Interconnects,
- Input/Output Blocks (IOB).

A Logic Element (LE) contains some lookup tables (LUT) and combinatorial logic. In this section, we will focus on a Xilinx FPGA. Figure 2.8 shows an LE that is found in a

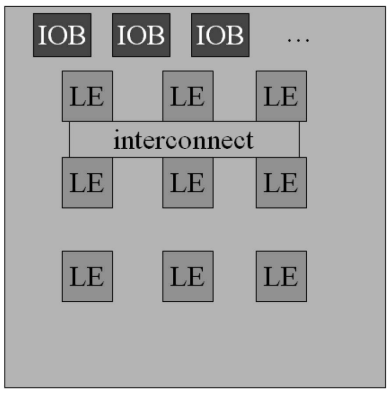


Fig. 2.7: Bird's eye view of an FPGA.

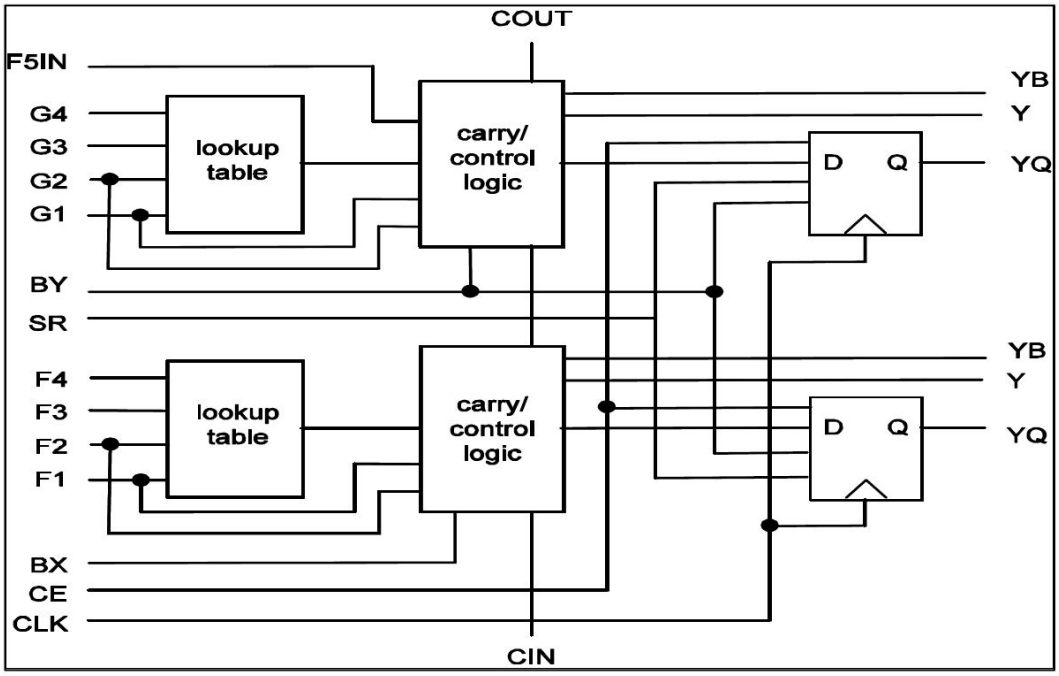


Fig. 2.8: Logic element for Xilinx Spartan FPGA.

Xilinx Spartan FPGA. The structure of the LE is similar across various families of Xilinx FPGAs. In this LE, two LUTs are available, each of which contain 4-bit wide input port and 1-bit output port. These LUTs can be configured either as a RAM unit, or as a Function generator, which is used to realize a truth table in hardware. Also, two carry-chain logic circuits are available and can be used to perform additions. Two registers are available that can be configured either as D-type Flip-Flops or as latches.

Interconnection architecture provides connections between the wiring channels and the logic elements. It is also used to connect any two wiring channels. Figure 2.9 shows the interconnection architecture available on Xilinx FPGAs. Four types of interconnects are:

- Local,
- General-purpose,
- Input/Output pin,
- Dedicated.

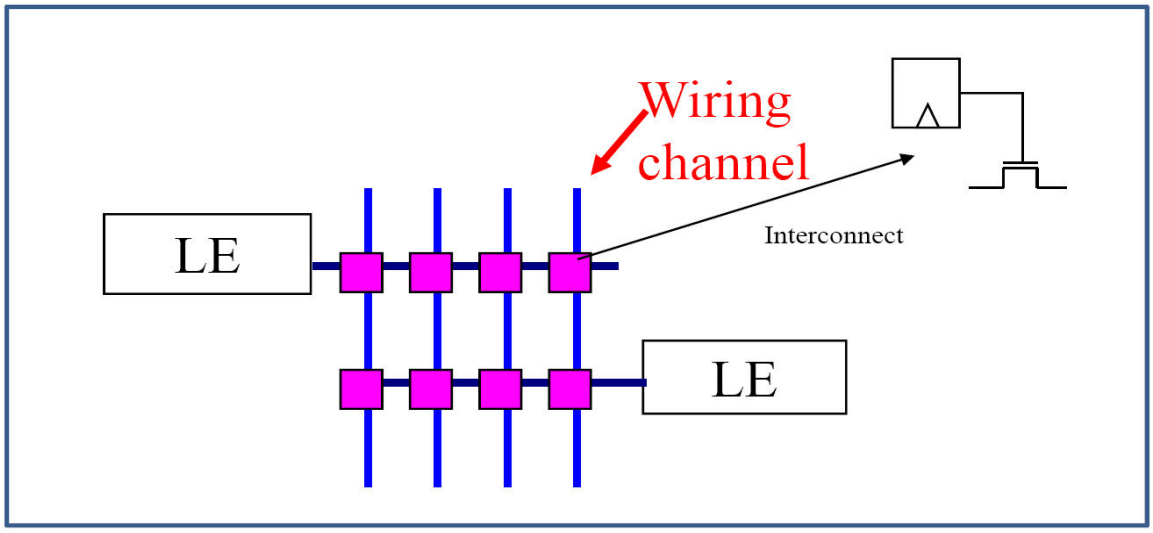


Fig. 2.9: Interconnection architecture in Xilinx Spartan FPGA.

Local interconnects are used to connect multiple LUTs and Flip-Flops (FF). It is the fastest programmable connection available in the FPGA. General-purpose interconnects are used to connect horizontal and vertical wiring channels. Dedicated interconnects provide direct LE-LE connectivity without any switches in the middle. I/O pin interconnects are used to connect the I/O pins with other logic internal to the FPGA. The amount of interconnect resources in FPGA is designed to be large enough to support the massive spatial parallelism that can be realized on an FPGA.

FPGA is programmed by downloading a configuration, called a *bitstream*, into its static on-chip configuration memory. This bitstream is the product of compilation tools that translate the high-level abstractions produced by a designer into something equivalent but low-level and executable. Tools used in the proposed research were developed by Xilinx. The design process involves a specification of the hardware circuit in a hardware description language, and the synthesis process breaks down this circuit into low-level gate representation. Xilinx mapping and place-and-route tools are used to generate the bitstream. The mapping process is done to map the gates generated by the synthesis process onto FPGA-based primitive circuit elements like LUTs and FFs. Place-and-route is done to place these primitive elements onto the actual FPGA floor plan and then wire them up

using the interconnect technology available on-chip.

Though FPGAs provide an attractive option for developing high-performance accelerator circuits, a major limiting factor in FPGA-based design is the run-time reconfiguration overhead. To alleviate this issue, FPGAs support some advanced features that include partial dynamic reconfiguration and partial dynamic relocation. A brief discussion of these features can be found in Chapter 4.

Chapter 3

Related Work

Proposed research is targeted towards developing and analyzing FPGA-based systolic array architectures for Kalman filters. Design parameters used in analyzing the architecture include, but are not limited to, precision (bit-width) of the floating-point Arithmetic Units (FPAU), latency of FPAUs, and number of Processing Elements (PEs) in the proposed architectures. Quality of the derived architecture can be measured in terms of the following: (i) performance in terms of overall execution time and maximum clock frequency, (ii) area or FPGA resource utilization, and (iii) error introduced in the results due to any reduction in precision. Other important factors that determine the quality of the architecture, but are not addressed in this research, include static and dynamic power and cost of the target FPGA. This chapter discusses the prior research efforts relevant to the scope of this dissertation research and is organized as follows. Section 3.1 discusses the research efforts undertaken towards the design of accelerator architectures for Kalman filters and other matrix-based operations. Section 3.2 discusses the efforts towards the analysis of error that is introduced in the design due to reduction in data precision. Section 3.3 discusses the modeling and analysis of performance and resource requirements.

3.1 Accelerator Architectures for Kalman Filters

As introduced in the previous chapter, Kalman filters were initially applicable to linear systems. Ongoing space missions use a modification of Kalman filters in order to appease the increasing demand for building nonlinear spacecraft models. Two variations of Kalman filters are used extensively: (i) Extended Kalman Filters (EKF), and (ii) Iterated Extended Kalman Filters (IEKF). Such Kalman filters, that approximate the nonlinear model into a linear model, are proven to be sub-optimal and may often result in numerical instability

within the mission duration, but are still the best-available option till date for autonomous estimation [9]. EKF and IEKF involve several irregular computations performed on regular data, as compared to the regular computations performed on regular data for linear Kalman filters. As a result, very few attempts have been made towards the hardware realization for these filters. Cardoso et al. [10] proposed a hardware software co-processor system to implement EKF on an FPGA. The designer performs the hardware software partitioning using some profiling information and an automated tool C2H [11] is used to generate hardware architectures for parts of the C source code as specified by the designer. Results show a speed-up of 4x over a software implementation.

This dissertation proposes to develop and analyze a polymorphic systolic array framework to accelerate EKF. While systolic arrays are well-equipped to accelerate any linear algebra operations, they possess specific limitations as a hardware co-processor, as listed below.

- Long design times and hardware design expertise are required to configure systolic arrays for a specific set of applications,
- Developing a memory support system to feed data in a specific pattern to such a co-processor is a tedious process.

There have been some efforts towards developing a specification language and an associated compiler for systolic arrays [12, 13]. However, the need for a customized specification language instead of ANSI C requires the designer to learn a new language.

In the previous chapter, the applicability of modeling the linear algebra kernels in Kalman filter using Faddeev algorithm was established. Existing research efforts [6–8] in developing systolic arrays to accelerate Faddeev algorithm fail to generate scalable designs and may result in large hardware designs that may not fit in a single FPGA (for large matrix sizes).

In addition to development of hardware architectures for Kalman filters, there has been considerable work done, in recent times, towards developing architectures for a specific linear algebra module. A discussion of some of these efforts is provided here.

El-Amawy [14] proposes a systolic array architecture consisting of $(2N^2 - N)$ PEs and claims to compute the inverse in $O(N)$ time, where N is the order of the matrix. However, there are no results to show that the large increase in area (for large values of N) is compensated by the benefits obtained in speed by this implementation.

Lau et al. [15] attempt to find the inverse of sparse, symmetric, and positive definite matrices using designs based on Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD) architectures. This method is limited to a very specific sub-set of matrices and not applicable for a generic matrix, and hence has limited practical utility. Edman and Owall [16] also targeted only triangular matrices.

Jang et al. [17] implement LU decomposition on Xilinx Virtex II FPGAs (XC2V1500), using a systolic array architecture consisting of 8/16 processing units. This work is extended to inversion and supports 16-bit fixed-point operations.

Daga et al. [18] propose a single and double precision floating-point LU decomposition implementation based on a systolic array architecture described by Jang et al. [17]. The systolic array architecture is a highly parallel realization and requires only a limited communication bandwidth. However, every element in the systolic array needs to have local memory and a control unit in addition to a computation unit, which adds significant overhead.

Wang and Ziavras [19] propose a novel algorithm to compute LU decomposition for sparse matrices. This algorithm partitions the matrix into smaller parts and computes LU decomposition for each of them. The algorithm to combine the results makes use of the fact that most of the sub-blocks of the matrix would be zero blocks. However, this method cannot be extended to find LU decomposition for dense matrices.

As the number of logic elements available on FPGAs increase, FPGA-based platforms are becoming more popular for use with linear algebra operations [20–22]. FPGA platforms offer either a distributed memory system or a shared memory system with large amounts of design flexibility. One such design, presented by Zhuo and Prasanna [22], utilizes FPGA-based architecture with the goal of minimizing power requirements.

By analyzing the existing efforts towards accelerating Kalman filters and linear algebra algorithms, it is observed that a plethora of work is done towards accelerating individual linear algebra operations. EKF consists of a varied set of linear algebra operations, and each of them need to be accelerated. Thus, it is necessary to realize a common architecture that can be used to accelerate all the operations. A systolic array architecture derived for Faddeev algorithm is an effective candidate for this requirement. It is observed that the existing research efforts fail to generate scalable designs that may not fit inside an FPGA. This dissertation proposes a scalable and polymorphic systolic array framework to accelerate EKF implemented using Faddeev algorithm. An extensive error, area, and performance analysis supports the efficient design of this architecture. Related work in architecture analysis is presented in the subsequent sections.

3.2 Error Analysis

In computer arithmetic, binary representation of a real number is required and this representation is an approximation of its real value. Such approximation is caused due to limitations in the number of binary digits used to represent the value and this limitation affects the range as well as precision of the binary representation. There are two types of binary representations that are primarily used: (i) fixed-point representation, and (ii) floating-point representation. In the following discussion, range is defined as the set of values that can be expressed using a particular representation. For instance, the range for real numbers is (-∞,+∞), and the range for IEEE-754 single precision floating-point representation is approximately (-2^{127},+2^{127}). Precision is used to define the smallest absolute difference between two numbers that are expressed using a particular representation. For instance, the precision for real numbers is infinite, and the precision for IEEE-754 single precision floating-point representation is 23. Limitations in the range may cause overflow errors and limitations in the precision may cause round-off errors.

A fixed-point representation can be defined as a representation where a fixed number of bits are allocated to represent the integer and fractional part of any real number. In this representation, range is defined using the number of bits used to represent the integer

parts and precision is defined using the number of bits used to represent the fractional parts. Arithmetic logic that is based on fixed-point representations can be easily realized by reusing the logic for integer arithmetic.

A floating-point representation is defined as a representation where the number of bits used to represent the integer and fractional parts are also part of the representation. Equation 5.1 shows the floating-point representation as conformed by IEEE.

$$\alpha = (-1)^{sign} \times 1.(mantissa) \times 2^{exponent}. \tag{3.1}$$

In this equation, sign is a single bit value. Mantissa has $nbits_m$ and exponent has $nbits_e$ bits. $nbits_m$ determines the precision of this representation and $nbits_e$ represents the range of numbers that can be represented. Single precision floating-point representation has one sign bit, followed by eight bits to represent a biased exponent (127+exponent), which is followed by 23 bits that represent the mantissa. The smallest number that can be represented using this representation is 2^{-151} and the range of values that can be represented using this representation is (-2^{127},$+2^{127}$). In the proposed research, this representation is termed as the benchmark floating-point representation. Floating-point arithmetic operations include addition, multiplication, and division. Each operation is characterized by the floating-point representation of its two inputs and one output. In the proposed research, floating-point representations of both the inputs and outputs are restricted to be the same and the set of arithmetic units is represented as add($nbits_m$,$nbits_e$), mul($nbits_m$, $nbits_e$), and div($nbits_m$, $nbits_e$). While additional logic is required to execute floating-point arithmetic operations, the range supported by floating-point representation is much larger than the range supported by fixed-point representation.

There has been some extensive research in the field of analyzing the effect of the data range and precision over the accuracy of the results. Prior to the standardization of floating-point arithmetic representation by IEEE, there were some efforts towards determining the precision and range and analyzing the error that is introduced. Richman [23] discusses the effect of tolerance in resultant error over the restrictions placed on the precision. In this

work, precision is expressed in terms of number of decimal places that is supported after the decimal point in a decimal representation of a floating-point number.

With the standardization of floating-point arithmetic, there has been limited work towards exploring the possibilities to vary data precision. Intel's MMX [24], HP MAX-2 [25], and SUN VIS [26] are some of the architectures that supported operations on sub-word data types. Mapping applications to such architectures required error analysis in an effort to use lower precision arithmetic operations that invariably had lower latency.

With the advent of FPGAs in various application domains over the past decade, there has been a considerable surge in efforts towards developing variable precision architectures. Lower precision architectures on FPGAs require lesser number of FPGA resources and are highly desirable. Efforts include error analysis for varying precision and range for various data types (fixed-point, floating-point, integer, etc). This section discusses the related work in this domain.

Initial efforts towards error analysis for variable precision architectures were targeted towards the fixed-point data representation [23, 27–34]. Constantinides [35–44] has also proposed numerous error analysis techniques in the domain of fixed-point representation. Many research efforts [45–52] target error analysis for floating-point representation. Also, a couple of efforts [53, 54] have specifically targeted matrix-based algorithms.

Richman [23] proposed one of the earliest techniques to analyze the error that can be tolerated in a rational expression when it is computed using variable precision arithmetic operations. Decimal notation was used to represent numbers and the number of digits after the decimal point represented the precision that is used to represent any number. Using interval arithmetic, the author transforms the tolerable error in the result into tolerable error in the inputs, thus attempting to determine the required precision.

Soderstrand and de la Serna [34] proposed one of the earliest techniques to automatically determine the error introduced in signal processing filters due to reduction in bit-width (precision) that is used to represent integer numbers. By analyzing the mathematical expressions that characterize the filters, the authors identified the operations that required

lesser bit-width. The designs were realized in earlier Xilinx FPGA devices that had limited support for variable precision arithmetic (options limited to 8-bit or 16-bit).

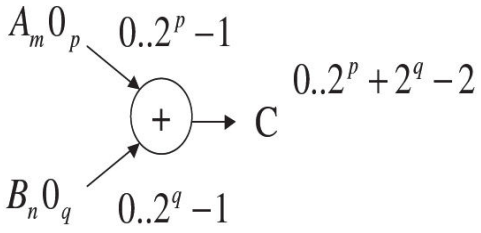
Nayak et al. [28] proposed an error analysis technique to improve the hardware designs generated by their Matlab to hardware compiler (MATCH) [55]. In this paper, error analysis is performed by varying the number of bits that are used to represent the integer and fractional parts of a fixed-point representation. The application is represented using a Static Single Assignment (SSA) based Data Flow Graph (DFG). Largest data range is computed at the output of every node of the DFG by propagating the inputs through the DFG. Number of bits required to represent the integer part of the real number that is generated at each node is computed using this data range. While number of bits to represent the integer parts of the output of different nodes can be different, the authors claim that the precision required at the outputs of different nodes is a constant. A mathematical formula is derived to determine the number of bits required to represent the fractional part of the output generated by all nodes. Results are provided to indicate a 5x savings in resources for an FPGA implementation of image processing applications.

While Nayak et al. [28] relied on simulation to determine the data ranges, Menard and Sentieys [31] proposed an analytical technique to evaluate the noise power at the output of an application for varying wordlengths (integer part + fractional part). Identical to the work by Nayak et al. [28], Menard and Sentieys [31] also represented the application as a DFG and proposed to vary the wordlength at the output of every node of the DFG. The authors identified three sources for quantization noise at the output: (i) noise transmitted from the inputs, (ii) noise introduced at each node, and (iii) noise introduced at the output due to quantization. Each of these sources were represented using specific parameters that depend on the set of wordlengths (at outputs of nodes) and can be used to calculate the output noise. By applying a threshold on the output noise, the set of wordlengths at all nodes were computed. Results are provided in terms of quantization noise versus the set of wordlengths. However, results in terms of resource savings are not provided.

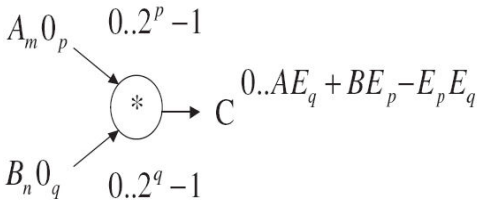
Chang and Hauck [27] proposed one of the earlier precision analysis techniques for

FPGA synthesis. In this paper, error models for fixed-point adder and multiplier units were proposed. Figure 3.1 illustrates their approach. Instead of using $m+p$ bits to represent A, only m bits were used. Maximum and average errors due to such modifications were modeled. For DFGs comprising of multiple operations, the error was propagated from the input to the output and the maximum and mean errors at the outputs was represented using a mathematical formula. Chang and Hauck also proposed a mathematical model to represent the resource requirements for adder and multiplier units in terms of their precision. This work initiated a trend in analyzing error and resource savings for FPGA-based variable precision architectures. However, generation of analytical models can be cumbersome for a complex DFG.

Chang and Hauck [33] further extended their work and presented some of their analysis in terms of a plot showing error versus resource savings. Such a plot is useful in terms of determining the design parameters for a particular threshold set on the error. Error versus resource savings plots have been presented for various applications including matrix multiplication, DCT, and CORDIC algorithms. One major drawback in this approach is



Error model of an adder



Error model of a multiplier

Fig. 3.1: Error models for fixed-point adder and multiplier units.

that error is not propagated from the input to the output. A normalized error is computed by finding the average of error introduced at each node of the DFG of the application. This error is not an exact indicator of the overall error introduced at the output due to quantization.

Lee et al. [29] proposed Minibit, a bitwidth optimization technique that uses the concept of affine arithmetic [56]. Authors proposed both range and precision analysis techniques. The authors proposed an interesting classification of error analysis techniques: static analysis versus dynamic analysis. Static analysis is performed using some analytical models and the characteristics of inputs are needed (possible range, expected precision, etc). In this analysis, the error estimates are mostly conservative. A wide range of possible error is presented. Dynamic analysis is performed using a simulation model that require input stimuli. While the error generated for a given input may be accurate, it may not be representative of the error generated for any input. Hence, a large number of input stimuli is required and this may result in long simulation times. Lee et al. [29] proposed a static error analysis that is based on affine arithmetic. Error metric was represented in terms of two terms: signal-to-noise ratio (an average value of error) and maximum absolute error bound. Resource estimates for Xilinx Virtex 4 FPGAs were modeled in terms of bit widths and results showed a 20% resource savings for an *ulp* [57] of 0.3. Here, *ulp* stands for unit of least precision, which is the least possible difference between any two real numbers represented using a particular precision and range.

Lee and Villasenor [32] proposed an alternate approach to analyze error due to variations in range and precision associated with every operation in the DFG. Range analysis was performed by computing the minimum and maximum values of the output of different nodes of the DFG, based on the range of inputs. This was determined by finding the roots of the derivative of the transfer function used to generate the output at every node. Such an approach requires the function to be continuous. Precision analysis was performed by generating an analytical model to represent the error at the output of the DFG. This model is comprised of the information on precision associated with every node of the DFG. Simu-

lated annealing techniques were used to determine the optimal values for precision at every node. Resource savings were demonstrated by using this approach for implementation on Xilinx Virtex 4 FPGAs.

Fiore [30] proposed an error analysis technique to determine wordlengths for signal processing applications. In this approach, the DFG was transformed into a flow graph with each node modeled as a transfer function and each arc associated with a wordlength. Effect of each individual wordlength on the overall error was presented in this paper. Results were presented in terms of a error versus resource savings.

Constantinides [35–44] has presented numerous efforts in the domain of synthesis of variable precision architectures for fixed-point representation and also published some papers [45] in the domain of synthesis of variable precision architectures for floating-point representation. We discuss some of his efforts here.

Constantinides et al. [36] proposed a paradigm for the design of systems involving multiple wordlengths. There have been numerous efforts towards the analysis of error that is introduced due to reduction in wordlengths at different nodes of a DFG. This effort by Constantinides et al. [36] differed from the other approaches, as the error analysis was now performed by reducing the wordlengths at different nodes of a system architecture. In this paper [36] and subsequent publications, Constantinides focused on the complexities involved in hardware synthesis of an architecture that uses operations with variable precision. Constantinides also provided a graphical interface of the system architecture to the user, thus allowing the user to modify the wordlength at multiple nodes and observe the error introduced at the output.

Constantinides et al. [37] proposed a technique to derive the optimal wordlength for each internal variable in a software design of a Digital Signal Processing (DSP) algorithm. Mixed Integer Linear Programming (MILP) solvers were used to determine the set of wordlengths for a given error threshold and the resource requirements were used to derive the cost function. Results have been shown for small circuits. The authors argued that minimizing the resource requirements for smaller logic is an essential task in the DSP

domain, as this domain consists of compute-intensive kernels that have limited arithmetic operations. Constantinides [38] also proved that the problem of optimal selection of multiple wordlengths is a NP-hard problem.

In subsequent years, Constantinides et al. [35] proposed an approach to optimally synthesize multiple wordlength architectures. In this paper, the application was modeled as a DFG with each node annotated with the precision information. This is in contrast with some of the contemporary approaches, where each edge was annotated with the wordlength information. This paper focused on the issues surrounding the scheduling and resource allocation for deriving an architecture for an annotated DFG. Both the problems were modeled using ILP with the resource requirement being the cost that needed to be minimized. Constraints included maximum permissible error and the data dependency constraints.

Constantinides proposed his latest efforts in the domain of synthesis of variable precision architectures [44]. In this paper, a set of kernels were identified for hardware acceleration and were analyzed to determine their maximum, minimum and mean wordlength requirements. A comprehensive set of results to illustrate the resource savings for integer arithmetic applications was provided for implementation on Xilinx Virtex 4 family of FPGAs.

Remainder of this section discusses the various efforts in the domain of error analysis for floating-point representation.

Gaffar et al. [46] proposed one of the earliest error analysis techniques targeting floating-point representation. Automatic differentiation was used to differentiate a program at runtime, given a specific set of input vectors. Partial derivatives of outputs for each of the inputs and any internal variable were derived. For a given output error, these derivatives were used to compute the permissible error in the inputs and internal variables, which, in turn, were used to derive the bit width requirement for these variables. Equation 3.2 represents this relation.

$$\Delta Y = \Delta X \times \frac{df(X)}{dX}. \tag{3.2}$$

Automated differentiation is a compute-intensive process. Authors attempted to limit the

design time by developing a specialized differentiation package. In this approach, chaining of error due to chaining of operations is ignored. The authors assumed that the reduction in precision of all variables affect the error at output. Error generated at output is assumed to be a sum of errors generated at output due to reduction in precision for the input and internal variables. Illustrative examples for a couple of signal processing applications were provided. Also, resource savings for implementation Xilinx Virtex II FPGA for several permissible errors were provided.

Chang and Hauck [48] proposed Precis, which is a design time precision analysis tool for floating-point applications. Similar to the multiple wordlength paradigm proposed by Constantinides et al. [36], this tool also assisted the developers to analyze the effect of precision on overall error and resource requirements. Support is provided for (i) simulation of error due to input stimuli provided by the developer, and (ii) determining the possible range for all variables based on the simulation results.

Strzodka and Goddeke [47] proposed an error analysis technique based on mixed-precision and targeted the implementation of Partial Differential Equation (PDE) solvers on FPGA devices. PDE solvers are iterative in nature and the algorithm converges towards the required result. By analyzing this algorithm, the authors deduced that the number of operations requiring full precision (of floating-point representation) are very few. Thus, using semantic knowledge of this algorithm, the authors split the algorithm into two parts: (i) a small set of computations that require full precision in the outer loop, and (ii) a compute-intensive inner loop that does not require full precision. The outer loop was implemented on a sequential processor (microblaze or PowerPC on the FPGA) and the compute-intensive inner loop is accelerated using the other FPGA hardware resources. This approach showcases a methodology that is effective for FPGA implementation of variable precision iterative solvers. However, this methodology cannot be extended to other application domains.

Sun et al. [50] proposed a mixed precision FPGA implementation of Lower-Upper (LU) decomposition based solver. An iterative technique was used to derive the lower and upper matrices and the approach seems similar to the work presented by Strzodka and

Goddeke [47]. A comprehensive analysis of resource savings on FPGA due to reduction in precision was presented. Results are shown for an implementation on the Cray XD-1 supercomputer.

Lee et al. [49] proposed the error and area analysis for using Floating-Point Arithmetic Units (FPAU) with reduced bit-widths. A novel representation of error associated with a particular precision (represented by *ulp*) and range (represented by the number of exponent bits) was proposed. This representation included Maximum Relative Representation Error (MRRE) and Average Relative Representation Error (ARRE). Output error was computed using these two error parameters. In this approach, the authors aimed to derive a single precision and range for a given output error threshold. However, obtaining multiple precisions for multiple operations has been proved to be a more viable approach to explore the design space [36]. Plots were shown for error versus number of fractional and exponential bits for each FPAU, and these results were interestingly similar to what we obtained. However, a more meaningful plot of error versus resource savings was not provided.

Constantinides et al. [45] proposed a parameterizable linear equation solver to be implemented in an FPGA. In contrast to the mixed precision approaches presented by Strzodka and Goddeke [47] and Sun et al. [50], Constantinides et al. proposed a single precision solver, and modeled the performance of the iterative solver in terms of time to converge to the solution. If the precision is low, the time to converge to the solution is expected to be high, and if the precision is high, the amount of parallelism that can be supported in hardware is low. In this paper, an error analysis technique was proposed to determine the optimal precision, and results were shown in terms of performance versus precision plots.

Jaiswal and Chandrachoodan [51] proposed an efficient FPGA implementation of a floating-point reciprocator. While maintaining the number of input and output bits, the authors aimed to reduce the resource requirements by modifying the internal design of the reciprocator such that a reduced bit-width is required at the internal stages of computation. Results were shown to indicate reduction in resources for supporting an output error of 1 *ulp* and 2 *ulp*. The authors have presented a similar approach to design an efficient double

precision floating-point multiplier in another publication [52].

In our research, we target the Kalman filter algorithm that involves linear algebra operations based on floating-point arithmetic. He et al. [53] and Irturk and Kastner [54] proposed variable precision architectures for linear algebra operations. He et al. [53] proposed a high-precision FPGA implementation of the linear algebra operations found in Basic Linear Algebra Sub-programs (BLAS) package. Resource reduction was obtained by computing specific portions of a linear algebra operation with a lower precision. For instance, summation of "n" floating-point numbers is illustrated in fig. 3.2 [53]. Error estimates for this approach were compared against the error estimates for fixed-point arithmetic and the numbers were found to be significantly lesser for the approach proposed by He et al. [53]. However, extra circuitry is required to align the matrix data before it is fed into the computation engine. Also, division is not supported.

Irturk and Kastner [54] proposed a scalable matrix inversion core using QR decomposition. In this paper, the author provided a comparative analysis of resource requirements and error of both variable precision floating- and fixed-point implementation of matrix in-

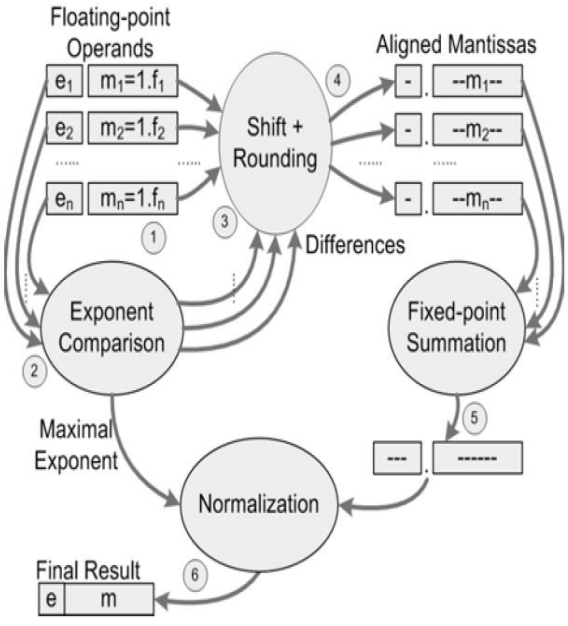


Fig. 3.2: Summation of "n" floating-point numbers.

version. Based on the results obtained, the authors proposed a fixed-point implementation and showed considerable savings in resources when compared with floating-point implementations associated with a similar error.

Constantinides et al. [58] proposed a novel data representation termed as dual fixed-point. This representation can dynamically support two different fixed-point representation and an additional bit is used to signify the selected fixed-point representation. Design of arithmetic operators for this data representation is also provided. Results indicated that the output error is comparable to the error introduced by floating-point representation, while the resource savings are significant.

Table 3.1 summarizes the efforts that have been discussed in this section. A set of characteristics to define the process of error analysis has been identified for this purpose and listed below.

- Static or dynamic analysis - Error analysis techniques can either be static or dynamic, as proposed by Lee et al. [29].
- Range or precision analysis - Either range or precision or both can be varied for analyzing error associated with real number-based computations.
- Floating- or fixed-point representation - Real numbers can be represented in binary arithmetic using two ways: (i) fixed-point, and (ii) floating-point. Error analysis technique targets one or both of these representations.
- FPGA as target - This characterizes whether the proposed error analysis targets FPGAs.
- Range of applications that are analyzed - Some of the work on error analysis propose an intermediate representation that can be used to represent any application and are characterized as general-purpose techniques. A few other approaches analyze a specific application or class of applications and propose error reduction techniques in addition to analysis techniques and are characterized accordingly.

Table 3.1: Classification of research efforts towards error analysis.

Related work	Static or Dynamic	Range or Precision	Data representation	FPGA	Range of applications
[23]	Static	Precision	fixed-point	No	General-purpose
[34]	Static	Precision	Integer	Yes	DSP
[28]	Both	Both	fixed-point	Yes	DSP and Image processing
[31]	Dynamic	Both	fixed-point	No	DSP
[33], [27]	Static	Precision	fixed-point	Yes	General-purpose
[29]	Static	Both	fixed-point	Yes	General-purpose
[32]	Both	Both	fixed-point	Yes	General-purpose
[30]	Static	Both	fixed-point	No	DSP
[35] - [44]	Both	Both	fixed-point	Yes	DSP
[46]	Dynamic	Precision	Float	Yes	General-purpose
[48]	Dynamic	Both	Float	Yes	General-purpose
[47]	Dynamic	Precision	Float	Yes	Iterative solvers
[50]	Dynamic	Precision	Float	Yes	Iterative solvers
[49]	Static	Both	Float	Yes	Neural networks
[45]	Static	Precision	Float	Yes	Iterative solvers

In this dissertation, a dynamic precision-based error analysis technique is proposed. Floating-point representation is used and the proposed approach targets FPGA, and is application specific.

3.3 Performance and Area Analysis

Any given application can be accelerated using numerous architectural options. While FPGAs provide an effective platform to realize and evaluate various hardware design options and obtain the performance and area associated with each design, the design time and complexity involved is considerable. Stringent constraints in design time and cost drastically reduce the number of designs that can be evaluated in order to realize the optimal architecture. Optimality of the architecture may be evaluated using (i) area requirements, or (ii) performance. In order to reduce design time, existing approaches propose the design of parameterizable architecture models that can be used to accelerate generic applications or a specific class of applications. Design parameters include (i) number of PEs, (ii) granularity of each PE, (iii) bit-width of the data path, (iv) memory size, (v) memory to PE

bandwidth, etc. Performance and area of the design depend on the design parameters and also the application characteristics. To determine the performance and area associated with a particular design option (defined using its parameters) that is used to accelerate a given application, we identify three possible options.

- Synthesis, Place, and Route on a target FPGA: This option is time-consuming.
- Development of a simulation model in software.
- Development of an analytical model and enhancing the model using the performance and area numbers for specific design options.

In this section, we discuss some of the existing techniques to estimate area and performance and emphasize on pre-synthesis estimation techniques. This section is concluded by providing a motivation for the approach adopted in proposed research.

Philippe et al. [59–62] proposed a fast estimation methodology that targeted FPGAs. In this approach, accuracy was compensated to achieve faster estimation times. Philippe et al. [61] proposed a Hierarchical Control Data Flow Graph (HCDFG) to represent a given application. Multiple control levels in the application were represented using multiple hierarchical levels in HCDFG. A 3-stage estimation process was adopted in this methodology, as listed below.

- Each node in the HCDFG is allocated a resource type.
- For a set of timing constraints, number of resources is estimated for each DFG in the HCDFG. For each DFG, number of resources required is now plotted against the number of clock cycles required to execute the DFG.
- A set of formulas is derived to combine the results for multiple DFGs in the HCDFG and derive the "number of resources" versus "number of clock cycles" plot for the entire HCDFG.

It can be observed that the accuracy of this approach depends on the number of data points that are provided in each of the plots. In this approach, the number of data points is limited, in order to perform fast estimation.

Philippe et al. [59] proposed a hierarchical modeling technique to model any reconfigurable architecture. This model proposed two types of elements: (i) functional elements, and (ii) hierarchical elements. Functional elements were used to model the functionality and granularity of the functional units available in the reconfigurable architecture. Hierarchical elements were used to model the communication between multiple functional elements. Performance and area costs were associated with each element. Simulation was required to estimate the area and performance of a particular application. A model for Xilinx Virtex II FPGA was presented in this paper. Philippe et al. [63] also presented the mapping of some applications on this architecture.

In another publication [60], Philippe et al. proposed an architecture model as an alternative to the hierarchical model presented in their prior work [59]. In this paper, a parameterizable architecture model was proposed. In this model, the number of functional units, number of memory units, and the bandwidth between functional and memory units can all be programmed to suit a particular application. Application was modeled using HCDFG, and area and performance were estimated in a technique similar to that presented in an earlier work [61]. In a related publication [60], memory costs and communication costs were also included in addition to the cost of functional units. Philippe et al. [62] also presented a model to estimate communication costs. Since communication costs in FPGAs are highly dependent on the placement and routing of various units, pre-synthesis modeling of such costs is a highly erroneous task and is not discussed further. Philippe et al. [64] presented a comprehensive analysis of the estimation process and accuracy of results was evaluated by comparing them against post-synthesis results. Also, estimation time was reduced by considering only the data points that fall on the pareto-optimal curve of each plot.

Agarwal et al. [65] proposed SimpleFit, which is a framework to explore the design

trade-offs in RAW architectures [66]. Raw Architecture Workstation (RAW) is a distributed multi-core processor network. Each core consists of a Reduced Instruction Set Computer (RISC) processor with instruction and data memory. A static network provides nearest neighbor communication. This paper identified two major design challenges: (i) granularity of each core, in terms of number of instructions that can be executed in parallel, and (ii) balancing of resources allocated to computation, memory, and communication, within each core. SimpleFit was composed of an architecture model and an application model. Cost of the architecture was estimated in terms of Static Random Access Memory (SRAM) bit equivalents (sbe) and performance was estimated in terms of number of machine cycles required to execute the application. Application was modeled as a task graph and each task was annotated with the following information: (i) number of operations to be performed, (ii) number of words to be communicated, and (iii) number of words that need to be stored. Architecture model for each core consisted of number of instruction pipelines and latency of each instruction, communication overheads and latencies, memory size available, etc. Cost of each core was computed based on some empirical data and statistics gathered from other superscalar architectures and was modeled as a polynomial expression. Analytical formulas based on the application and architecture parameters were used to estimate performance.

Nayak et al. [67] proposed area and delay estimators as part of the MATCH compiler framework [55]. In this framework, applications developed using Matlab are compiled to be accelerated using FPGAs. In the context of MATCH compiler, each functional unit is generated using a set of functional generators. For instance, the number of functional generators used to generate an n-bit integer adder is equal to "n." Post place-and-route results were used to determine the number of FPGA primitives used to derive one functional generator and this number is used to derive the estimation formulas for different functional units. Delay of functional units was estimated based on a formula that is derived using post place-and-route timing results.

Park et al. [68] proposed performance and area models for an architecture template that is used to accelerate image processing applications. Image processing applications can be

modified using various loop transformation techniques and the effect of each transformation is reflected in the image processing architecture template. This template can be programmed using the following parameters:

- Number of data paths,
- Number of external memories,
- Number of I/O ports,
- Number of FIFOs used as on-chip memories,
- Number of tapped delay lines used to exploit data reuse.

Performance was modeled on analytical formulas derived using the architectural parameters, application characteristics, and the loop transformation characteristics. Area was estimated using an analytical formula that is derived using experimental results. This approach was used to evaluate the impact of several loop transformations on overall performance.

Memik et al. [69] proposed an architecture template that used a streaming model to facilitate data transfers. In this paper, the authors estimated timing and area based on the characteristics of the functional and memory units, and also the units used for communication (FIFOs, multiplexers, etc). This approach is one of the few existing approaches that considered width of the data path as a design parameter in their architectural template. Application was modeled as streaming Data Flow Graph (sDFG) and each node in this sDFG was annotated with the requisite bit-width information. Multiplexer logic and size of FIFOs in the architecture were estimated by analyzing the sDFG of the application. This approach is limited to application that can be modeled as a streaming application.

Liu et al. [70] proposed an architecture template that is targeted to represent a hierarchical memory sub-system. The template presented in this paper represents a 2-stage memory sub-system (off-chip and on-chip). To use the on-chip memory efficiently, the authors aimed to maximize the opportunity to reuse data. While targeting loop-based application code, the authors identified that loop transformations often resulted in conflicting

impacts on the concurrency and data reuse opportunities associated with the transformed code. In this paper, the authors proposed a framework to evaluate multiple loop transformations and determine the optimal solution. Area was computed using the architectural parameters associated with on-chip memory and timing was computed using application characteristics and the two memory latencies (off-chip and on-chip).

Smith et al. [71] presented their research efforts towards floor planning for reconfigurable architectures. In this paper, the authors proposed an area and timing model to estimate the area and execution time associated with an architecture after it is floor planned onto the target FPGA. Estimated results were used to evaluate the floor plan. Area model was constructed using post-synthesis results and timing was computed by using estimated delay between two communicating modules. In the proposed research, we target estimation of area and timing at a much higher level of design abstraction (system level). Currently, we use floor planning to place the multiple nodes of the systolic array on the target FPGA. Due to the linear nature of the proposed systolic array, there is no freedom to place these nodes, and hence there is no need for a timing and area model at that level of design abstraction.

Rice and Kent [72] proposed an analysis of a parameterizable 1-D systolic array architecture. This paper classified hardware design approach as follows.

- Instance specific - One variation of a given application (for example, 8-point FFT) is analyzed thoroughly and one instance of the best possible architecture to accelerate this application is identified. Support for any other variation in the given application (for example, 16-point FFT) requires complete re-design.

- Parameter specific - Based on some parameter associated with the application (for example, n is a parameter for n-point FFT), an parameterized architecture template is developed. By modifying the architecture parameters, different variations of the application can be supported.

In this paper, a parameter specific systolic array architecture, with number of PEs as the variable parameter, was proposed. Variable number of PEs were used to support

acceleration of an application with different problem sizes and also to explore various levels of parallelization. The authors observed that increasing the number of PE resulted in a higher acceleration, but also resulted in a higher communication overhead (between memory and the array). A comprehensive timing analysis to identify the optimal number of PEs was presented.

Rashid et al. [73] proposed a design flow for FPGA/CPU/DSP-based heterogeneous platforms (hArtes). In this design flow, an application was mapped as a Task Flow Graph (TFG). Each task in the TFG was annotated with OpenMP pragmas that are used for application partitioning, profiling pragmas that are used to compute the execution time of the TFG, and hardware pragmas that are used to assign the task to a particular component of the target platform. Proposed tool flow allowed the designer to develop a task transformation. A particular task transformation was defined using three parts: (i) list of tasks that are suited for this transformation, (ii) a set of pre-conditions that determines the need to apply this transformation, and (iii) list of tasks after transformation. The author provided a cost estimation tool to determine the execution time of a particular task when it runs on the FPGA, CPU, or DSP. Our requirement is more fine-grained than the approach presented by Rashid et al. [73].

Givargis and Vahid [74] proposed a parameterized System-On-Chip (SoC) platform model. Parameters included frequency of operation of CPU, width of the data and address buses, cache characteristics, etc. In this paper, power and execution time were estimated for various sets of parameter values and for various applications. Results were provided in terms of 2-D pareto curves (time versus power).

By analyzing the existing work in this domain, it is observed that most of performance and area estimation approaches target a parameter specific architecture. Parameters include, but are not limited to, data path width, memory size, and number of PEs. In this dissertation, parameters are identified at a finer level of design granularity and such parameters include latency of floating-point arithmetic units, data path width of individual floating-point units, and input rate of floating-point units. A simulation model is proposed

to estimate execution time, instead of an analytical model, as formulating the run-time performance of the proposed design comprising of FIFOs and a systolic array is a tedious task. For estimating area, we store the area requirements for various implementation options and add up the area requirements of individual components to obtain the overall area.

Chapter 4

Polymorphic Faddeev Systolic Array Architecture

This chapter starts with an overview of Extended Kalman Filter (EKF). Computational requirements of EKF are presented. The Faddeev algorithm is used to process all the linear operations of EKF. Details of the Faddeev algorithm are followed by analysis of data flow in the Faddeev algorithm. Proposed techniques to map the Faddeev algorithm onto a 1-D Polymorphic Faddeev Systolic Array (PolyFSA) are presented. Proposed PolyFSA architecture is discussed in detail and the motivation to analyze the various architectural options to realize the FPGA design of PolyFSA is provided. This chapter is concluded by providing the details for mapping the PolyFSA architecture onto the target FPGA.

4.1 Extended Kalman Filter

The concept of Kalman filters was proposed by Rudolf Kalman [4] and is introduced in sec. 2.1. While the Kalman filter is applicable to a large number of linear systems, it cannot be used to accurately model the nonlinearity of complex real world problems, particularly for navigation/guidance of spacecrafts. Extended Kalman Filter (EKF) [75–78] is a variant of the linear Kalman filter that dynamically linearizes the nonlinear system equations to enable fast and accurate state estimation. Process of mapping the entire EKF onto an FPGA involves the following challenges.

- Portions of the algorithm are nonlinear in nature. Such nonlinear portions involve complex arithmetic that are difficult to be mapped onto hardware.
- Arithmetic logic in the nonlinear portions of the algorithm depends on the system and measurement models and can change completely from one pair of models to another, thus necessitating a new architecture to be designed for each specific problem and hence reducing the opportunities for design reuse.

To circumvent these issues, EKF is partitioned into linear (accelerated) and nonlinear (non-accelerated) portions that can be processed on a co-processor accelerator hardware and software, respectively.

Figure 4.1 illustrates all the functions present in EKF (in terms of linear algebra operations and nonlinear functions) and the data flow, and the "predict" and "update" stages are clearly delineated. Barnes [79] has described each variable in fig. 4.1 in detail. Each function in EKF is labeled using an index number and this number is required for fur-

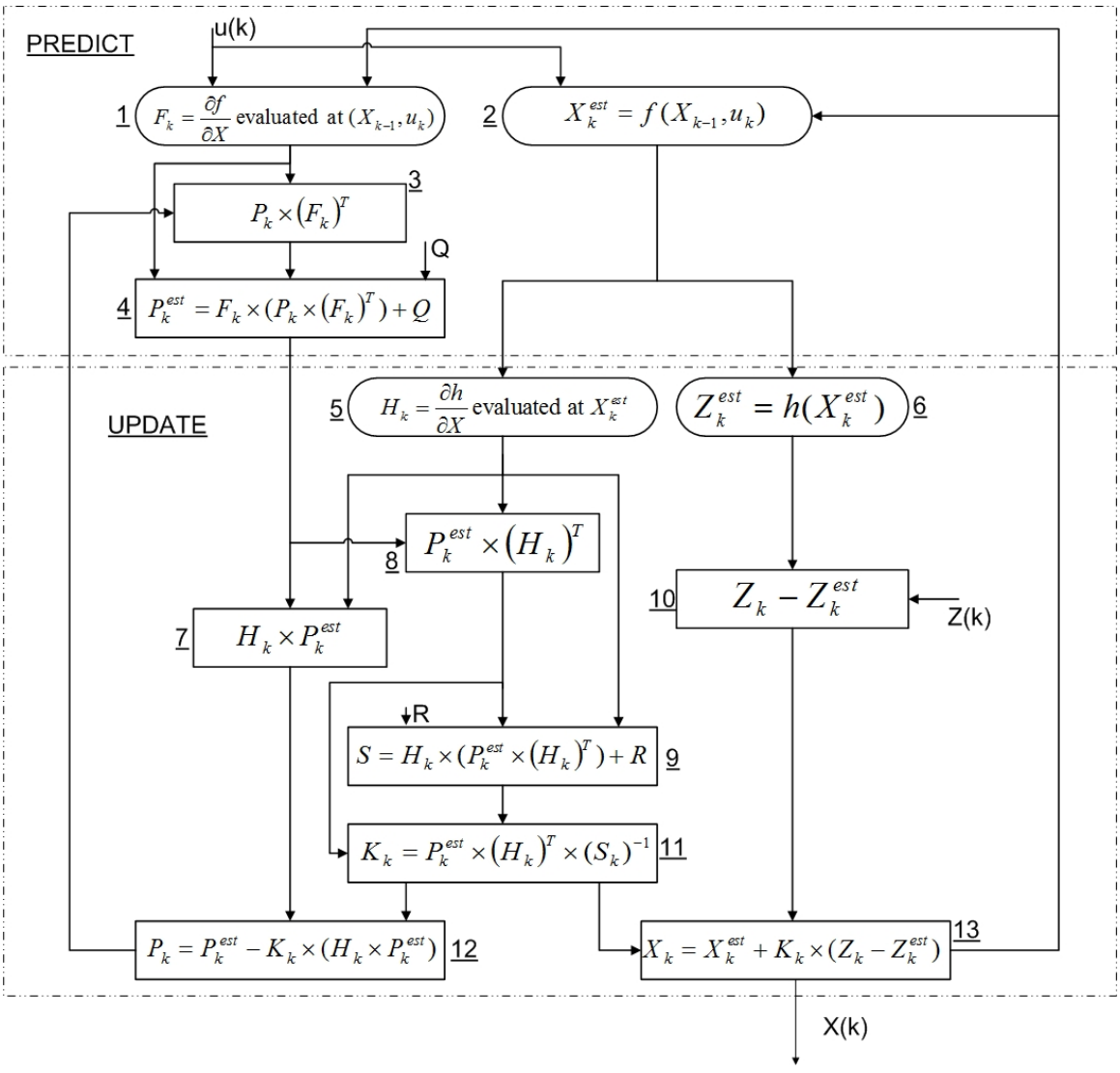


Fig. 4.1: Top-level flow of EKF algorithm.

ther discussions. Functions 1, 2, 5, and 6 are non-accelerated functions. Other functions are comprised of matrix operations that are consistent across all system and measurement models and vary only in size. These functions can be mapped onto a systolic array-based co-processor architecture using the Faddeev algorithm. Details of Faddeev algorithm are discussed in the next section. During run-time, execution of EKF algorithm is repeated for multiple iterations. During k^{th} iteration, a new measurement variable (Z_k) and a control variable (u_k) are available as input from the measurement system (GPS, INS, etc.) and the controller system (accelerator, etc.) of the spacecraft, respectively. At the end of each iteration, the state variable (X_k) and the covariance matrix (P_k) are updated. The total number of iterations is labeled as *nSteps*. The following section discusses the details of the Faddeev algorithm. In a typical EKF algorithm, the inputs from the controller are optional and are not included in this discussion. Also, number of states and number of measurements are represented by *NS* and *NM*, respectively.

4.2 Faddeev Algorithm - Overview and Analysis

Faddeev algorithm is a popular method for computing the Schur complement (E) for a set of four input matrices (A, B, C, D) according to the following equation:

$$E = D + CA^{-1}B. \tag{4.1}$$

Given two or more matrices, any of the three matrix operations (matrix inverse, multiplication, and addition) or a combination of the three operations can be performed by appropriately assigning the given matrices to the input matrices of the Faddeev algorithm. For instance, to add two matrices X and Y, A must be assigned to an identity matrix, B assigned to X, C assigned to an identity matrix, and D assigned to Y. In this example, the Schur complement resolves to

$$E = Y + (I \times I^{-1}) \times X = Y + X. \tag{4.2}$$

In the Faddeev algorithm, the input matrices are arranged in a specific format and the

resultant matrix is termed as the Faddeev matrix and is shown in eq. 4.3. $< N, M, P >$ are defined as the set of Faddeev parameters.

$$FaddeevMatrix = FM(N+M, N+P) = \begin{bmatrix} A(N,N) & B(N,P) \\ -C(M,N) & D(M,P) \end{bmatrix}, \tag{4.3}$$

where $\alpha(a, b)$ represents matrix α of size $a \times b$.

Table 4.1 shows different matrix operations that can be realized.

Following are the steps in the Faddeev algorithm.

- Step 1: Matrix C is annulled. Specifically, a specific multiple (W) of A is added to -C such that -C + WA = 0, meaning $W = CA^{-1}$.
- Step 2: B is multiplied by W and added to D.
- Step 3: Result in the lower right quadrant is $WB + D$, which expands to $CA^{-1}B + D$ (the Schur complement).

Due to its regularity, the Faddeev algorithm can be used to accelerate all the linear functions that are present in the EKF algorithm (refer to fig. 4.1). For each linear function, the Faddeev matrix and the set of Faddeev parameters can be derived. Table 4.2 shows the listing of derivations for all linear functions in EKF. If it is possible to derive a single systolic array architecture to accelerate a set of Faddeev algorithms with variable Faddeev parameters, then all the linear functions in EKF can be accelerated using this architecture. Since the Faddeev matrix is an input, it can be any 2-D set of numbers. Remainder of this

Table 4.1: Some matrix operations computed using the Faddeev algorithm.

Operation	A	B	C	D
$\alpha + \beta$	Identity matrix	Identity matrix	$-\beta$	α
$\alpha - \beta$	Identity matrix	Identity matrix	$-\beta$	α
$\alpha \times \beta$	Identity matrix	β	$-\alpha$	Zero matrix
α^{-1}	α	Identity matrix	-(Identity matrix)	Zero matrix
$\alpha + (\beta \times \partial^{-1})$	∂	Identity matrix	$-\beta$	α

Table 4.2: The Faddeev matrix and the Faddeev parameters for different linear operations in EKF.

Function index	Function	Faddeev matrix	M	N	P
3	$P_k^{temp} = P_k \times (F_k)^T$	$\begin{bmatrix} I & (F_k)^T \\ -P_k & 0 \end{bmatrix}$	NS	NS	NS
4	$P_k^{est} = F_k \times (P_k \times (F_k)^T) + Q$	$\begin{bmatrix} I & P_k \times (F_k)^T \\ -F_k & Q \end{bmatrix}$	NS	NS	NS
7	$P_k^{temp} = H_k \times P_k^{est}$	$\begin{bmatrix} I & P_k^{est} \\ -H_k & 0 \end{bmatrix}$	NS	NS	NM
8	$P_k^{est} \times (H_k)^T$	$\begin{bmatrix} I & (H_k)^T \\ -P_k^{est} & 0 \end{bmatrix}$	NM	NS	NS
9	$S = H_k \times (P_k^{est} \times (H_k)^T) + R$	$\begin{bmatrix} I & (P_k^{est} \times (H_k)^T \\ -H_k & R \end{bmatrix}$	NM	NS	NM
10	$Z_k - Z_k^{est}$	$\begin{bmatrix} I & I \\ Z_k^{est} & Z_k \end{bmatrix}$	NM	1	1
11	$K_k = P_k^{est} \times (H_k)^T \times (S_k)^{-1}$	$\begin{bmatrix} S_k & I \\ -P_k^{est} \times (H_k)^T & 0 \end{bmatrix}$	NS	NM	NS
12	$P_k = P_k^{est} - K_k \times (H_k \times P_k^{est})$	$\begin{bmatrix} I & H_k \times P_k^{est} \\ -K_k & P_k^{est} \end{bmatrix}$	NS	NM	NS
13	$X_k = X_k^{est} + K_k \times (Z_k - Z_k^{est})$	$\begin{bmatrix} I & Z_k - Z_k^{est} \\ -K_k & X_k^{est} \end{bmatrix}$	NS	NM	1

section discusses the data flow associated with the Faddeev algorithm. Understanding the data flow is critical to mapping any algorithm onto a systolic array.

Figure 4.2 illustrates the Data Flow Graph (DFG) of Faddeev algorithm for *M=N=P=3*. The DFG in fig. 4.2 is inherently a 3-D graph which has been flattened. An index is assigned to each node, for ease of discussion. In this figure, for sake of clarity, a broken line is used to indicate that the result of the boundary node is broadcasted to all the internal nodes in that row. Each node in this DFG is either a boundary node (square node) or an internal node (circular node). Based on the type of the node, the processing varies. Figures

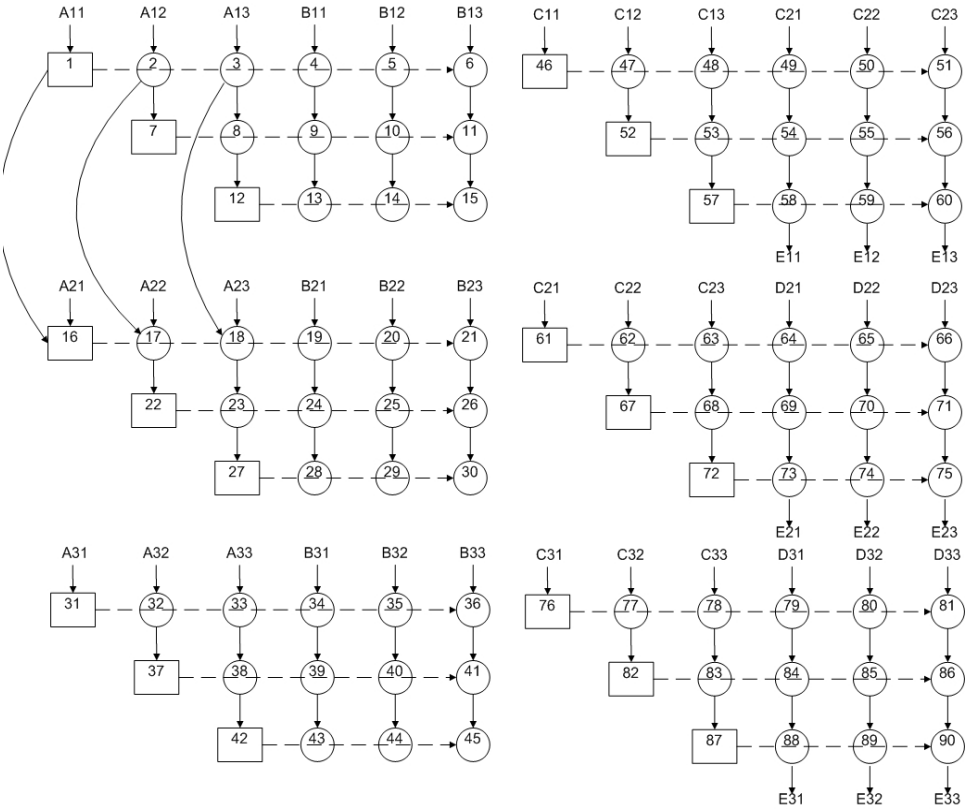


Fig. 4.2: A flattened 3-D DFG of Faddeev algorithm for M=N=P=3.

4.3 and 4.4 represent the data routing for i^{th} boundary and internal nodes, respectively. Functionality of each node is dependent on two factors: (i) input data (*In1* and *In2*), and (ii) location of the node in the DFG. Functionalities of the boundary and internal nodes are presented in Tables 4.3 and 4.4, respectively. In figs. 4.3 and 4.4, it is seen that multiplexers (MUX) are used to select between an input from the external world or the output from some other node. External inputs to the nodes in the Faddeev DFG are fed from the Faddeev matrix. In fig. 4.2, the inputs are specified as αij and this refers to $(i,j)^{th}$ element of the input matrix α in the Faddeev matrix (A,B,C, or D). Intermediate inputs are routed from multiple nodes.

Existing approaches to derive systolic array architectures for the Faddeev algorithm have tried to map this 3-D DFG to 2-D or 1-D arrays. A set of 2-D arrays for different Faddeev parameters is shown in fig. 4.5. It is observed that the number of Processing

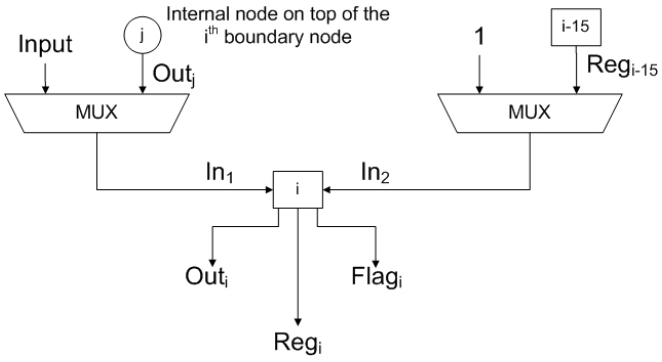


Fig. 4.3: Data flow inside the boundary node of Faddeev DFG.

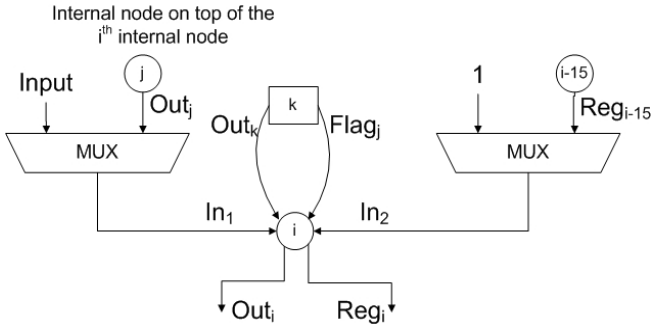


Fig. 4.4: Data flow inside the internal node of Faddeev DFG.

Elements (PEs) is set to be the same as the number of nodes in the DFG in the corresponding dimension. In some of the existing approaches, this 2-D array is further mapped onto a 1-D array. All these approaches result in large hardware designs that may not fit inside a single FPGA and also fail to generate scalable designs. An alternate approach is required to map the DFG onto a systolic array with the number of PEs set to be lesser than the number of nodes.

Mapping to systolic arrays for EKF algorithm has two major challenges.

Table 4.3: Operation of boundary nodes in EKF.

Matrix row	Case	Out_i	$Flag_i$	Reg_i
A/B (Node index < 46)	$\|In_1\| > \|In_2\|$	$-In_2/In_1$	1	In_1
	$\|In_1\| < \|In_2\|$	$-In_1/In_2$	0	In_2
C/D	All cases	$-In_1/In_2$	0	In_2

- EKF algorithm is comprised of multiple Faddeev algorithms with multiple sets of Faddeev parameters, with each algorithm resulting in a DFG of different number of nodes. Hence, a scalable systolic array is required.

- Data flow depends on the input and intermediate data values. Hence, the routing between PEs needs to be flexible.

The next section discusses the technique to map the Faddeev algorithm onto a scalable systolic array and discusses the systolic array architecture in detail.

Table 4.4: Operation of internal nodes in EKF.

$Flag_k$	Out_i	Reg_i
1	$Out_k*In_1+In_2$	In_1
0	$Out_k*In_2+In_1$	In_2

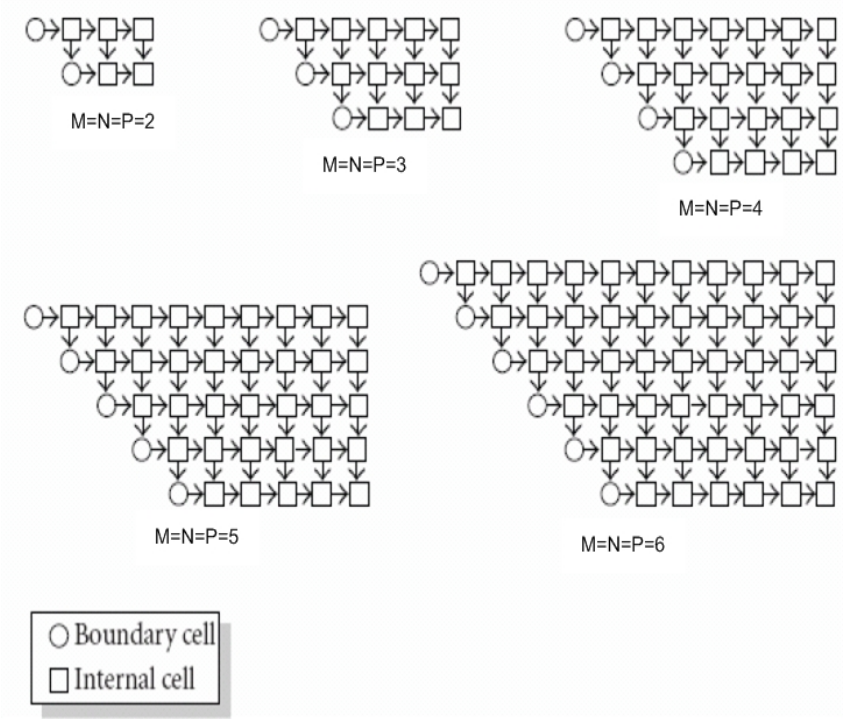


Fig. 4.5: 2-D systolic arrays for accelerating the Faddeev algorithm with various Faddeev parameters.

4.3 Mapping to Systolic Array

Based on the requirements of the Faddeev algorithm, a scalable systolic array is derived. As an initial step, each index in fig. 4.2 is transformed into a triplet $< i, j, k >$. DFG in fig. 4.2 can be described as a set of six trapezoids. In the triplet notation, i is used to indicate which trapezoid the node belongs to. j is used to indicate the row number inside each trapezoid. k indicates the location of the node in each row. For example, the node index 64 is transformed to $< 5, 1, 4 >$ and the node index 76 is transformed to $< 6, 1, 1 >$. This triplet notation is used in the following discussion. The Faddeev DFG is a 3-D graph with i, j, and k forming the three dimensions.

As a first step, the 3-D DFG is mapped to a 2-D systolic array along the "i" dimension. Hence, $< 1, j, k >$, $< 2, j, k >$, $< 3, j, k >$, $< 4, j, k >$,$< 5, j, k >$, and $< 6, j, k >$ are mapped to be executed on the same PE for all j and k. Figure 4.6(a) shows the 2-D array. The 2-D array is now mapped onto a 1-D array along the "k" dimension. As two types of nodes are found along the "k" dimension, support for both types is provided. There is one boundary node and this node is mapped onto a boundary PE and the multiple internal nodes are all mapped onto a single internal PE. Figure 4.6(b) shows the 1-D array. In this systolic array, all the nodes with index $< i, j, k >$ are executed on the j^{th} PE. Intermediate data between nodes $< i, j, k >$ and $< i, j+1, k >$ is sent from j^{th} PE to $(j+1)^{th}$ PE via a First-In-First-Out (FIFO) queue. To derive a scalable systolic array, the number of PEs should be independent of the number of nodes in the DFG. Data processing performed in the 1-D array shown in fig. 4.6(b) needs to be re-mapped to a systolic array with smaller number of PEs (nPE). Data flow is folded and re-mapped as shown in fig. 4.6(c). Figure 4.6(c) shows the framework of the proposed Polymorphic Faddeev Systolic Array (PolyFSA). In this figure, nPE is set to be 2. Now, each PE has two input FIFOs and two output FIFOs. Data are read from the input FIFOs in alternating fashion. Also, each PE is provided with ports to communicate with external circuitry.

As explained before, the Faddeev algorithm is comprised of complex data flow. This data flow is illustrated in fig. 4.2. Figures 4.3 and 4.4 showcase the data flow inside the

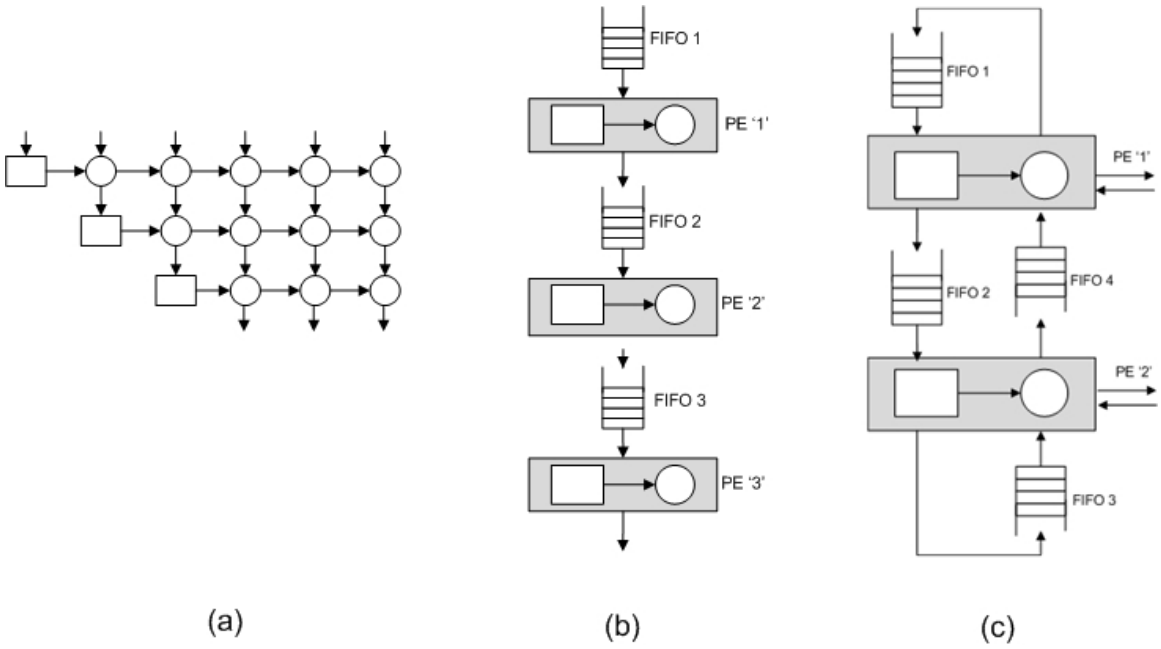


Fig. 4.6: Mapping the Faddeev algorithm to systolic array: (a) 2-D systolic array (simplistic view), (b) 1-D systolic array (without folding), (c) 1-D systolic array (with folding).

boundary and internal nodes, respectively. Tables 4.3 and 4.4 showcase the computations performed in the boundary and internal nodes, respectively. All the above information is used to derive the design of a single PE in PolyFSA. As seen in fig. 4.2, multiple types of data flow need to be supported. Three types of data flow are identified. These data flow types are listed below and the proposed method to support each data flow is also explained.

- In the proposed PolyFSA, nodes $< 1, j, k >$, $< 2, j, k >$, $< 3, j, k >$, $< 4, j, k >$,$< 5, j, k >$, and $< 6, j, k >$ are mapped to be executed on the same PE for all j and k. Intermediate data between nodes $< i, j, k >$ and $< i + 1, j, k >$ is stored in a memory location that is local to each PE. To ensure data flow, the intermediate data is tagged with its associated value of i. For instance, when the intermediate data between $< 5, j, k >$ and $< 6, j, k >$ is being written to the memory location, it is tagged with i=5. When node $< 6, j, k >$ needs to be executed, the data and tag is read from the memory location and the tag is checked. If the tag is not equal to 5, the execution is paused until the tag is updated.

- In the proposed PolyFSA, nodes $< i, j, 1 >$, $< i, j, 2 >$, $< i, j, 3 >$, $< i, j, 4 >$, $< i, j, 5 >$, and $< i, j, 6 >$ are mapped to be executed on the same PE for all i and j. Intermediate data between nodes $< i, j, 1 >$ and $< i, j, k >$ (for $2 \leq k \leq 6$ and for all i and j) is stored using a local memory location. To ensure data flow, the intermediate data is tagged with its associated value of i. For instance, when the intermediate data between $< 5, j, 1 >$ and $< 5, j, k >$ is being written to the memory location, it is tagged with i=5. When node $< 5, j, k >$ needs to be executed, the data and tag is read from the memory location and the tag is checked. If the tag is not equal to 5, the execution is paused until the tag is updated.

- In the proposed PolyFSA, $< i, j, k >$ and $< i, j + 1, k >$ are mapped to be executed on adjacent PEs for all i, j, and k. Intermediate data between nodes $< i, j, k >$ and $< i, j + 1, k >$ is sent from j^{th} PE to $(j + 1)^{th}$ PE via a First-In-First-Out (FIFO) queue. This data is tagged with all the three values (i, j, and k). These three values characterize the node that needs to be executed on the PE at a given instant. When data is read from the FIFO, the tags are also read and are used to verify if the data available in the local memory unit is valid.

A state machine is used to control the operation of a single PE in the proposed FSA. The associated state transition diagram is illustrated in fig. 4.7. Initially, the input FIFO is checked to see if it contains data by polling the *FifoEmpty* flag. If data is available, then a set of registers called *StartReg* is used to store the data. This data is comprised of the input to the PE and the tags (i, j, and k). Local memory is now checked to verify if all the inputs required to process the node $< i, j, k >$ are ready. A control variable termed as *flag* is used to perform this verification. If all the inputs are ready, then the computation is started. Based on the value of k, the boundary node computation or the internal node computation is started. After a certain number of control steps (depending on the latencies of the boundary unit and internal unit), the *done* signals are asserted and the appropriate memory locations and the output FIFOs are updated. Figures 4.8 and 4.9 illustrate the architectures of the boundary and internal PEs, respectively. In these architectures, a set

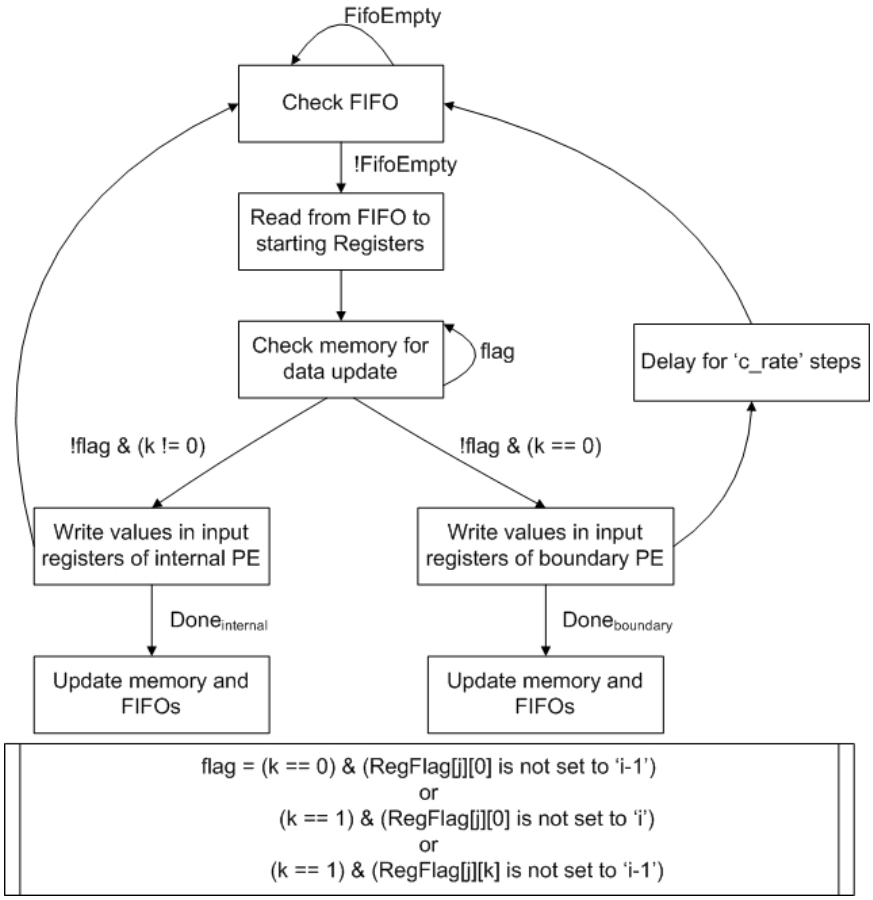


Fig. 4.7: State transition diagram used to control the operation of a single PE in PolyFSA.

of registers (*StartReg*s) are used to buffer the input data (*In1*) and the tags *(i,j,k)* until the other inputs become available. *ComputeFlag* unit is used to determine the value of *flag* variable. This unit is responsible to compute the memory locations from which the inputs are read. During every control step, the memory location is accessed and *flag* is computed. Once the *flag* is computed to be 0, then the values in *StartReg* (*In1*) and the memory locations (*In2* and *Out*) are written to the input registers of the boundary (or internal) compute unit. These input registers are labeled as *InReg*. It is observed that the boundary node computation does not require the data to be transmitted to the output FIFO.

Once outputs are available, both the local memory units are updated and data is written to output FIFO (in case of internal node computation). Data written to output FIFO consists of the output of the compute unit (*Out*) and the updated tags *(i,j,k)*.

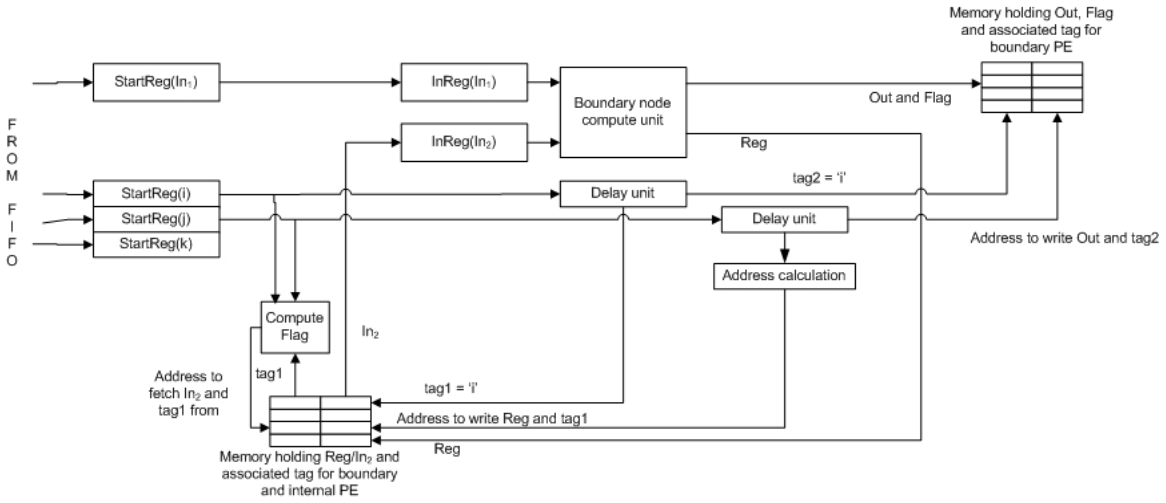


Fig. 4.8: Architecture details of boundary PE of PolyFSA.

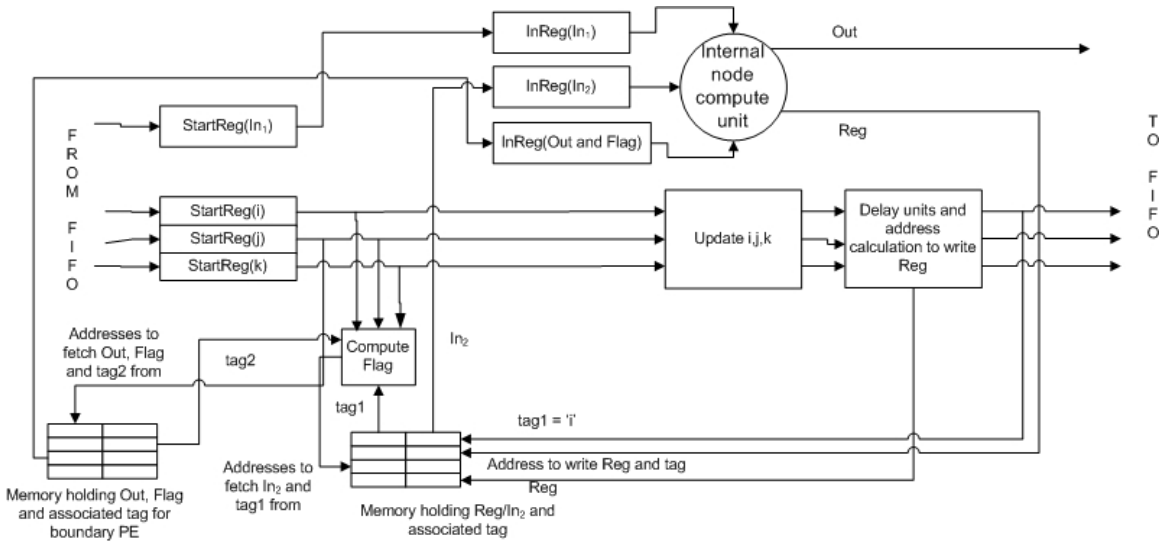


Fig. 4.9: Architecture details of internal PE of PolyFSA.

Final design of the single PE architecture is a combination of the two architectures illustrated in figs. 4.8 and 4.9, and the state machine derived using the state transition diagram in fig. 4.7. There is significant amount of overlap between the circuitry shown in figs. 4.8 and 4.9. Non-common parts are combined using multiplexers.

Compute unit for boundary node computation is comprised of a floating-point divider unit and the compute unit for the internal node computation consists of floating-point

adder and a floating-point multiplier units. Implementation of Floating-Point Arithmetic Units (FPAUs) requires considerable number of FPGA resources. It is observed that the FPAUs contribute to nearly 80% of the FPGA resource requirement of a single PE. Changes in the design of FPAUs affect the overall resource requirements by a significant amount. Following is a list of design parameters that affect the resource requirement of a single PE in the proposed design.

- Precision of the adder unit - This is represented by the number of bits used to represent the floating-point numbers that are fed as inputs to (and sent out as outputs from) the adder unit. Lesser the precision, lower is the resource requirement. However, reduction in precision results in erroneous results.
- Precision of the multiplier unit.
- Precision of the divider unit.
- Latency of the adder unit - This number indicates the number of pipeline stages in the adder unit. Lower the latency, lesser is the resource requirements, as the number of pipeline registers reduces. However, the maximum clock frequency at which the circuit can be operated is reduced with reduction in latency. This, in turn, decreases the overall performance.
- Latency of the multiplier unit.
- Latency of the divider unit.
- Input rate of the divider unit - This parameter indicates the number of clock cycles after which the next input can be fed into the divider unit. By increasing this number, the resource requirement can be reduced. However, the performance degrades.

In addition to these parameters, the choice of target FPGA and the choice of implementation (using dedicated resources like DSP48) also affect the overall resource requirements. In the proposed research, we emphasize on the parameters listed above. It is observed that

the efforts to reduce the resource requirements result in introduction of error into the results and also changes in overall performance. Chapter 6 provides a comprehensive analysis of error that is introduced in the results generated by EKF algorithm due to reduction in precision of FPAUs. Chapter 6 also presents a performance model and analyzes the effect of reduction in latencies and input rate on the overall performance. Section 6.3 presents an analysis of resource requirements for variations in the parameters listed above. The next section discusses the FPGA design of top-level system architecture that is used to execute the overall EKF algorithm and emphasizes on the design of PolyFSA.

4.4 FPGA Design using PDR

This section discusses the design of proposed Polymorphic Faddeev Systolic Array (PolyFSA) architecture that can be used to accelerate the EKF algorithm. This design is supported by a microprocessor, which could be either a MicroBlaze or a PowerPC. A data buffer and a hardware controller are used as interfaces between the PolyFSA and the microprocessor.

4.4.1 Top-Level System Architecture

Figure 4.10(a) illustrates the top-level system architecture for the proposed PolyFSA. This architecture consists of (i) microprocessor, (ii) co-processor (PolyFSA), and (iii) interfacing logic. Microprocessor can be implemented using the Xilinx soft-core MicroBlaze [80] processor with an internal floating-point unit and associated program/data memory. This microprocessor is used for performing three different tasks that are listed below.

- Computing portions of an algorithm that cannot be efficiently accelerated using the co-processor. Portions of EKF involve word-level operations involving complex arithmetic that are better suited to be implemented in software.
- Controlling and scheduling operations onto the co-processor.
- Hosting software necessary to support partial dynamic reconfiguration and relocation.

Proposed PolyFSA co-processor architecture consists of a set of PEs. In fig. 4.10(a), these PEs are labeled "FSA PE." Implementation of a PE using PDR techniques is discussed later in this section. In addition to PEs, a set of switch boxes is also available to route data between PEs. Figure 4.10(b) illustrates the design of a switch box. Each switch box consists of three multiplexers that can be programmed to allow routing along the east-west directions, east/west-north and loops (east-east or west-west). By controlling the reconfiguration of PEs and the multiplexers inside switch boxes, it is possible to dynamically scale the number of PEs dedicated to EKF algorithm in the proposed systolic array. Interfacing logic shown in fig. 4.10(a) consists of a controller and a data buffer. Controller logic performs two different tasks that are listed below.

- Receiving macro instructions from the microprocessor. By decoding these instructions, the controller generates appropriate signals to control the data buffer and PolyFSA. Macro instructions are used for (i) reading or writing data to the co-processor from the microprocessor, (ii) reading and writing data from the data buffer to the PolyFSA, (iii) programming the switch boxes, and (iv) resetting the co-processor.

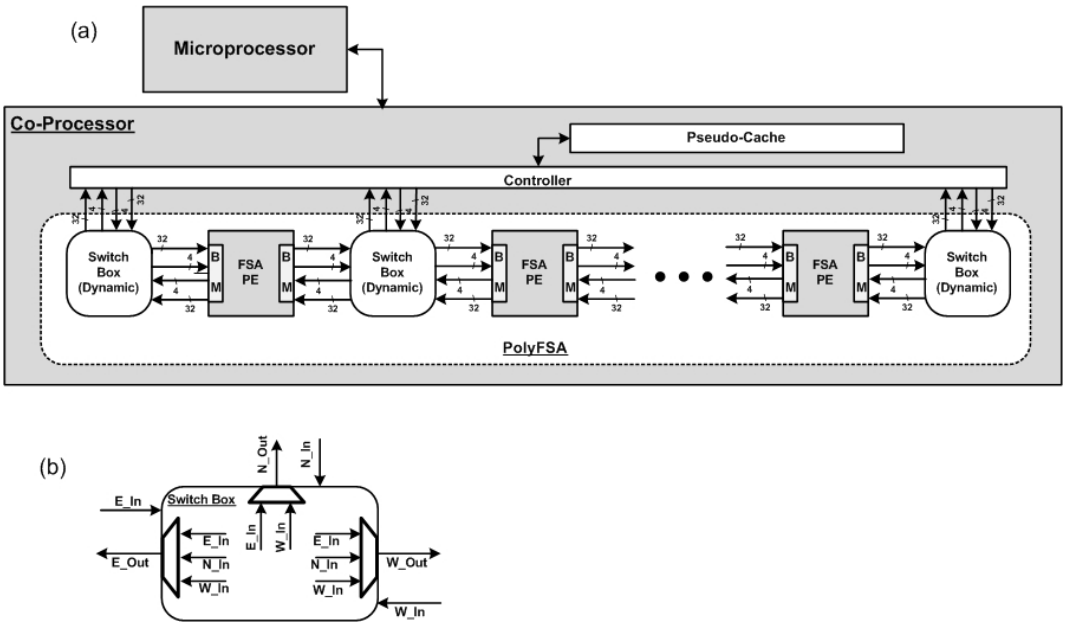


Fig. 4.10: (a) Top-level system architecture to accelerate EKF, (b) switch box design.

- Routing data between the microprocessor, data buffer, and PolyFSA. Special logic is available in the controller to read/write blocks of data from/to the data buffer.

Figure 4.10(a) shows the data buffer that can be selectively refreshed and provide low latency access to the co-processor. Size of the data buffer is determined by the number of available Block RAMs (BRAMs) and the problem size. It is possible that two different copies of the same data element are present in the microprocessor memory and the data buffer. A table on the microprocessor is designed to maintain dirty bits that represent the status of data elements in both memories. If data is made dirty by the microprocessor the corresponding data buffer memory locations are freed and the data is sent back to the co-processor only if it is required. If data is made dirty by the co-processor the copy in the cache is sent back to the microprocessor whenever it is required. This ensures data is only synchronized between the microprocessor and co-processor when necessary, thereby reducing data transfer time. Proposed implementation of the data buffer on the Xilinx Virtex 4 SX35 FPGA can be used to store 4K words, with 128 lines/blocks, and 32 words per block.

4.4.2 FPGA Implementation of PolyFSA Using Partial Reconfiguration Techniques

This section discusses the design technique to implement PolyFSA-based system architecture on a Xilinx Virtex 4 SX35 FPGA. This process can be extended to other FPGAs as well. Two design phases are discussed in the following sub-sections: (i) hardware design, and (ii) software design. FSA architecture is designed such that the number of PEs in the top-level systolic array framework can be modified by controlling the reconfiguration of PEs and the multiplexers inside switch boxes. Also, multiple versions of the PE designs with varying area and performance and error numbers can be implemented and the suitable version can be selected to be loaded during run-time. Remainder of this section discusses the design techniques used to develop the proposed PolyFSA architecture.

Hardware design

Figure 4.10(a) illustrates the top-level design of the proposed system architecture. This design consists of two types of regions.

- Partial reconfigurable region: Each FSA PE is placed inside a partial reconfigurable region or a socket. Each socket can be partially reconfigured (via Internal Configuration Access Port (ICAP)) to act as either one of the versions of FSA PE or act as a PE for some other application. Interface to static region is provided via four 32-bit buses that are dedicated for data transfers and four 4-bit buses that carry control information. Within a socket, asynchronous busmacros are inserted to realize these interfaces.

- Static region: Remainder of the system architecture design (other than PEs) is included in the static region. This region is configured once and cannot be dynamically reconfigured.

Layout of the floor plan for the system architecture is shown in fig. 4.11. It can be seen that the components of the static region (MicroBlaze, data buffer, controller, switch boxes, etc.) are distributed on the right side of the chip and the clock regions dedicated for partial reconfigurable regions are distributed on the left side of the chip (highlighted in white). Number of partial reconfigurable regions depends on three factors: (i) size of FPGA; (ii) size of the largest design that needs to fit inside this region, which in turn determines the size of the region; and (iii) placement of these regions on the FPGA. Techniques and tools used to place and route the design and their limitations are not discussed here.

Software design

In the proposed design, partial bitstreams are located in the external flash memory and the software processor is responsible for the reconfiguration process. Details of the software design can be found in the thesis report by Barnes [79].

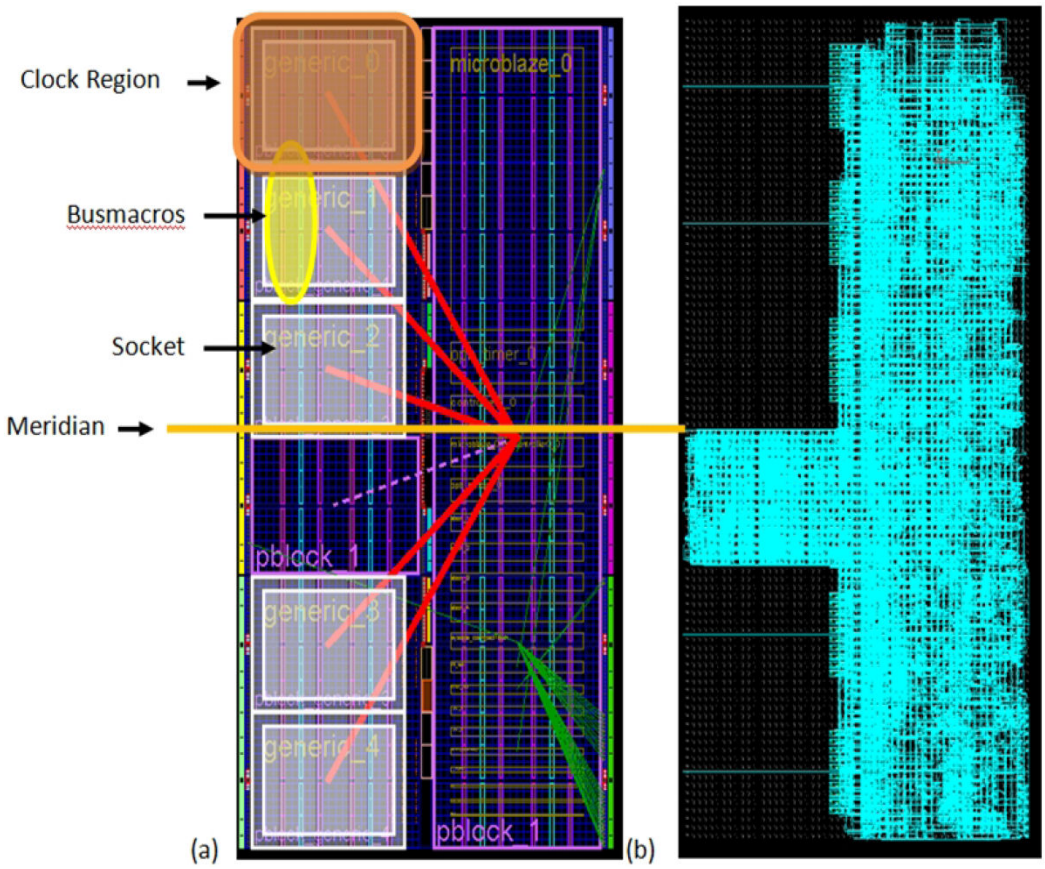


Fig. 4.11: (a) Placement of five partial reconfigurable regions in Virtex 4 SX35 FPGA, (b) static region.

4.5 Summary

This chapter discussed the derivation of the proposed PolyFSA for accelerating the EKF algorithm. A 1-D systolic array architecture has been derived. The remainder of this dissertation report proposes techniques to analyze this architecture and discusses the results of such analysis.

Chapter 5

Architectural Analysis

This chapter presents the architectural analysis of PolyFSA-based system architecture. A hierarchical error analysis is followed by the discussion of the simulation model used to estimate the performance. The chapter concludes by presenting an area analysis.

5.1 Error Analysis

This section provides a hierarchical discussion of the error introduced into the processing of Extended Kalman Filter (EKF) due to variations in data precision associated with floating-point arithmetic units. IEEE-754 standard for floating-point number representation is considered to be the fundamental representation. This section is organized as follows. Section 5.1.1 discusses the top-level flow of the proposed error analysis technique and also discusses the motivation behind our analysis. Section 5.1.2 analyses the errors introduced by individual arithmetic units with variable data precision. Section 5.1.3 provides a contextual overview of the Faddeev algorithm and discusses the errors introduced in the output of this algorithm due to variations in data precision of the three arithmetic units. Section 5.1.4 discusses EKF and showcases how the error varies over multiple iterations of this filter. Results of this analysis are presented in terms of a 2-D pareto curve (area versus error) that can be used to configure the proposed systolic array with a suitable data precision.

5.1.1 Motivation and Top-Level Flow of Proposed Error Analysis Technique

Kalman filters typically operate on 1-D vectors and 2-D matrices. Each of these matrices is made up of values that are derived using the position, velocity, acceleration, etc, of a spacecraft. Existing hardware or software designs used to execute EKF typically use the

benchmark floating-point representation. Proposed implementation is a scalable systolic array and is comprised of numerous floating-point arithmetic units. These units are highly expensive in terms of the number of resources required to realize them on an FPGA. Number of resources required to implement these units on a Xilinx Virtex 4 SX35 FPGA is shown in Table 5.1. Latency of each operation is set to be 8 clock cycles and no embedded units were used in this design. In this table, each unit is represented as operator($nbits_m$,$nbits_e$).

Each PE of the systolic array consists of an adder unit, a multiplier unit, a divider unit, and some control logic. This node utilizes nearly 15% of the overall FPGA resources and this limits the number of PEs that can be fit inside the FPGA. An obvious option to reduce the resource utilization is to reduce the number of bits used to represent the mantissa or exponent. Figure 5.1 shows a plot of number of LUTs and number of FFs required to realize the design of adder($nbits_m$,8) unit, for varying $nbits_m$ from 4 to 23 (supported by Xilinx floating-point core generator library). It is observed that the resource utilization can be reduced to half by reducing the $nbits_m$ to 13. Similar results can be observed for the multiplier and divider units as well. However, reduction in the precision of arithmetic units comes at the cost of increase in computational error. This error is found to vary based on the values of $nbits_m$ and $nbits_e$ for the different arithmetic units. Any attempt to reduce the precision of arithmetic units should be made only if the error that is introduced in the algorithm by this reduction is permissible. Thus, architecture exploration based on variation in data precision of arithmetic units requires a comprehensive analysis of the error introduced in the algorithm by such variation. Remainder of this section provides a discussion on the top-level flow of the proposed error analysis technique.

Table 5.1: Resource utilization for arithmetic units on Xilinx Virtex 4 SX35 FPGA.

	LUTs	Flip-Flops	Maximum Clock Frequency (MHz)
Xilinx Virtex 4 SX35	30720	30720	500
Adder(23,8)	565	420	243
Multiplier(23,8)	641	698	274
Divider(23,8)	824	1370	278

Figure 5.2 outlines the top-level flow of the proposed error analysis technique. Inputs to the EKF algorithm are generated using a random number generator. It is imperative that the inputs reflect the values that will be fed into the system during run-time. The

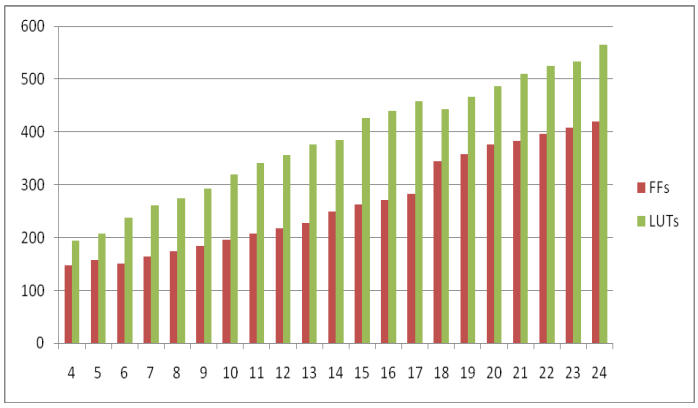


Fig. 5.1: Resource utilization for Adder($nbits_m$,8) for varying $nbits_m$.

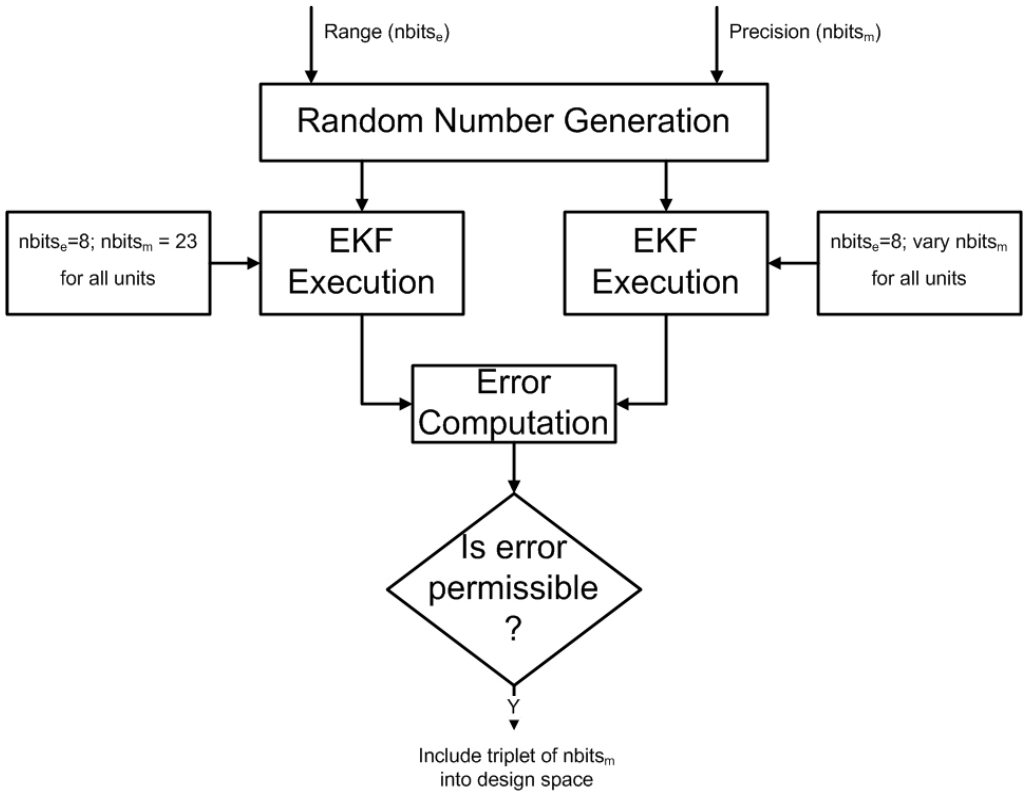


Fig. 5.2: Top-level flow of proposed error analysis.

random number generator is constrained by two factors: Range and Precision. Range dictates number of bits to be used for the exponent and precision dictates number of bits to be used for the mantissa. Both these factors can be varied by the user to guide the error analysis. Once the inputs are generated, they are fed to two variations of the EKF algorithm: (i) EKF algorithm with the arithmetic units characterized by the benchmark representation, and (ii) EKF algorithm with varying $nbits_m$ for the three arithmetic units. The three variables are represented henceforth as follows: *madd*, *mmul*, and *mdiv*. $nbits_e$ is not varied, because the area savings obtained by reducing it is negligible. In EKF algorithm, results are in the form of matrices. To derive the error percentage for a $M \times N$ matrix, the following formula is used.

$$Error = \left(\sum_{i=1}^{M} \sum_{j=1}^{N} \left| \frac{X_{bench}(i,j) - X_{vary}(i,j)}{X_{bench}(i,j)} \right| \right) \times \frac{100}{M \times N} \tag{5.1}$$

In the proposed error analysis, the inputs are generated using a pseudo random number generator. This analysis is iterated multiple times in order to generate the error for a wide range of values. Following statistical parameters are computed to represent the error. In the intermediate analysis presented in the following sections, only the mean error is presented.

$$Mean = \frac{\sum_{i=1}^{nIter} Error(i)}{nIter} \tag{5.2}$$

$$Variance = \frac{\sum_{i=1}^{nIter} (Error(i) - Mean)^2}{\sqrt{nIter}} \tag{5.3}$$

5.1.2 Error Introduced by Individual Arithmetic Units

This section documents the error caused by varying precision of individual arithmetic units. Effect of varying the range and precision of the inputs is also studied. Figure 5.3 shows the error caused by the arithmetic units. Range of inputs is set to be (-2^{16},$+2^{16}$) and precision of inputs is set to be 23 bits. It is observed that the error percentage is less than 1% for all the three arithmetic units when $nbits_m$ is greater than 8. From fig. 5.2,

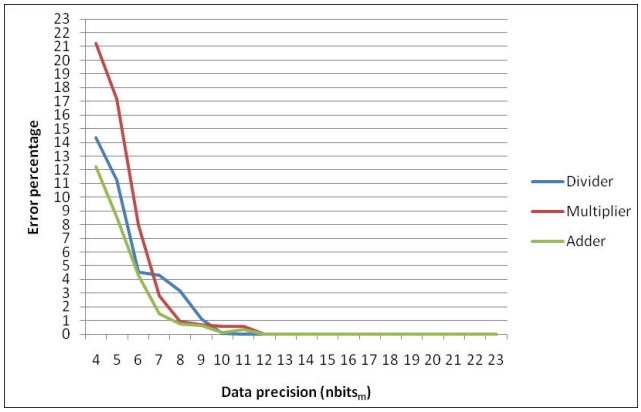


Fig. 5.3: Error percentage versus data precision for the three arithmetic units.

it can be observed that there is a possibility of reducing the resource requirements by 60% if an error of 1% is permissible. It is also observed that the error caused by multiplier is much higher than that of adder and divider. Also, the difference in error for varying input ranges is found to be insignificant.

5.1.3 Faddeev Algorithm and Associated Error Analysis

Previous section showcased the error caused by variation of precision for individual arithmetic units. In the Faddeev algorithm, a sequence of arithmetic operations is performed for generating results. For a given problem size (M=N=P), number of operations that need to be performed to generate a result is found to be $O(N^2)$. The error percentage is dependent on four variables: N, $madd$ (or $nbits_m$ for adder), $mmul$ (or $nbits_m$ for multiplier), and $mdiv$ (or $nbits_m$ for divider). Figure 5.4 showcases the effect of varying data precision of a single variable, while the other two are set to 23 (maximum) on the error. Error percentages are shown for multiple problem sizes. From these plots, following set of observations can be made.

- For most cases, the error increases with increase in problem size. This is due to the fact that more arithmetic operations are performed in a sequence.
- Overall error of the Faddeev algorithm is most affected by the reduction in precision of

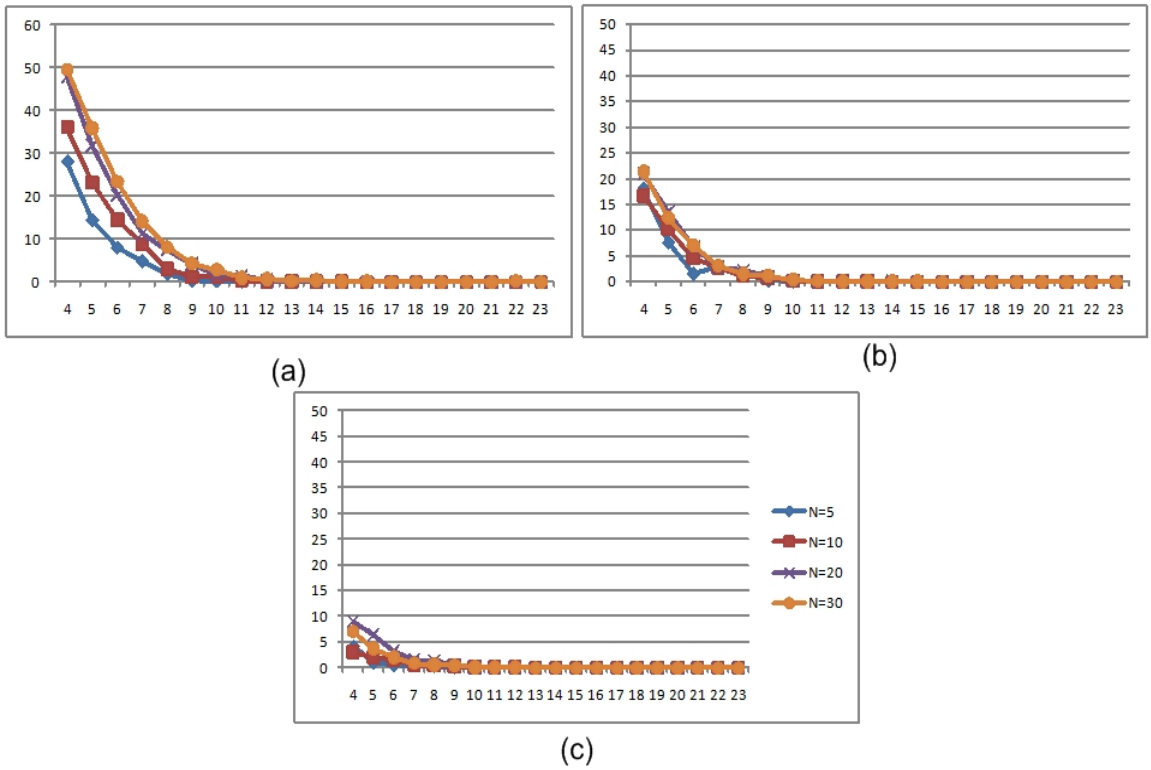


Fig. 5.4: Error percentage of result of the Faddeev algorithm for varying data precision of: (a) adder unit, (b) multiplier unit, and (c) divider unit.

the adder unit. A significant difference in the error caused by adder and error caused by other units is observed.

- Error caused by reduction in precision of the multiplier unit does not vary significantly for varying problem sizes.
- Overall error of the Faddeev algorithm is least affected by the reduction in precision of the divider unit.
- Error percentage is found to be less than 1% for a data precision equal to 10. Thus, reducing the precision of either adder or multiplier or divider unit, will result in an error percentage less than 1%.

Above observations require further analysis of the data flow inside the Faddeev algorithm. Each arithmetic operation with a reduced data precision introduces an error in the

result that it generates. An average of this error is showcased in fig. 5.3. We present the following analysis to study the effect of error introduced by the reduction in precision of individual arithmetic units on the overall error in the final results. As the DFG for Faddeev algorithm is complex and data flow depends on the value of the intermediate data elements, a set of smaller DFGs are used for this analysis. Results of this analysis are then used to explain the trend observed in the overall error of Faddeev algorithm.

Figure 5.5 showcases the sample DFGs used to analyze the effect of data flow on the overall error introduced in the final result. In this figure, In_i indicates an input with zero error. Out_i indicates either an intermediate result or the final result. DFGs shown in figs. 5.5(a) - 5.5(c) are used to analyze the effect of the error associated with the inputs of an arithmetic operation on the error associated with the output. In the DFG shown in fig. 5.5(a), both the inputs generating the final output are associated with zero error. In the DFG shown in fig. 5.5(b), one of the inputs generating the final output is associated with

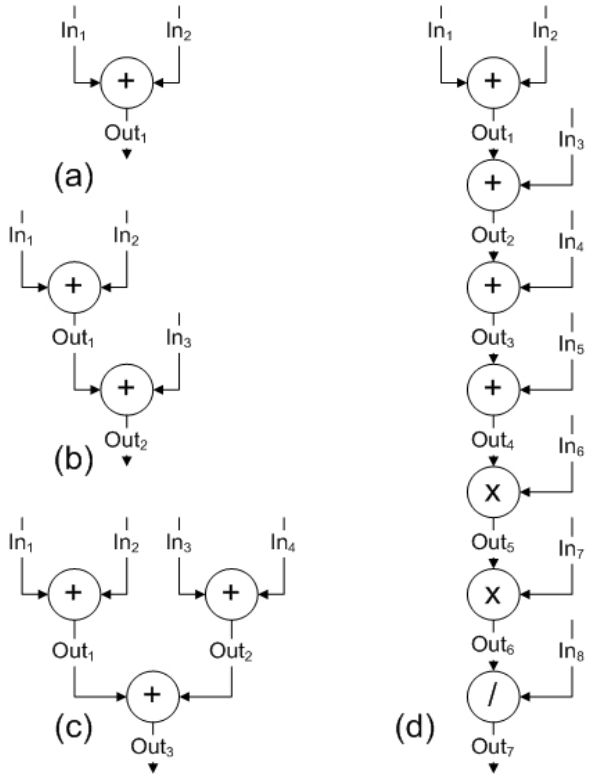


Fig. 5.5: Sample DFGs used for error analysis.

some error that is introduced by the earlier arithmetic operation. In fig. 5.5(c), both the inputs generating the final output are associated with some error that is introduced by the earlier arithmetic operations. Table 5.2 shows the error associated with the final outputs of the DFGs shown in figs. 5.5(a) - 5.5(c) for varying precisions (*madd*). From this table, it is observed that the error introduced in the final output increases with increase in the number of inputs that have some error associated with them. This analysis shows that the error introduced in the result of an arithmetic operation depends on three factors.

- Precision of the arithmetic unit performing the arithmetic operation (*madd*, *mmul* or *mdiv*).
- Error associated with the first input.
- Error associated with the second input.

DFG shown in fig. 5.5(d) is used to study the effect of the number of arithmetic operations on the error associated with the final result. In this DFG, it is observed that the data flow is such that the final result depends on all the intermediate results, and hence depends on the precision of all the arithmetic units. In this DFG, it is observed that there are four addition operations, two multiply operations, and a single division operation. Table 5.3 showcases the overall error associated with the error for different combinations of the precisions of adder, multiplier, and divider units performing those operations (*madd*,*mmul*,*mdiv*). It is observed that the increase in error is the largest for decrease in the precision of the

Table 5.2: Error (in percentage) associated with final output of DFGs shown in figs. 5.5(a) - 5.5(c).

Number of mantissa bits in adder (madd)	Error associated with final output of DFG shown in		
	Figure 5(a)	Figure 5(b)	Figure 5(c)
20	0.00007	0.000092	0.000105
16	0.001043	0.001512	0.002095
12	0.018413	0.025937	0.038739
8	0.285528	0.385217	0.504192
4	4.258781	5.925812	7.908808

adder unit, followed by the decrease in precision of the multiplier unit. From this analysis, we can conclude that the number of arithmetic operations that affect the final output also contribute to the error associated with the final output, in addition to the precision of the units performing those operations.

From the DFG of the Faddeev algorithm shown in fig. 4.2 (in Chapter 4), it is observed that the number of addition operations affecting the final result is much larger than the number of multiply operations, which in turn is much larger than the number of division operations. This is reflected in the error plots shown in fig. 5.4. In these plots, it is observed that the decrease in precision of the adder unit affected the error associated with the final result of the Faddeev algorithm to a much larger extent than the decrease in precision of the multiplier or the divider unit.

5.1.4 EKF and Associated Error Analysis

In Chapter 4, EKF algorithm is discussed in detail. It is shown that the EKF algorithm can be realized using multiple instances of the Faddeev algorithm. Problem size of the EKF algorithm is defined using four factors: (i) Number of state variables (NS), (ii) Number of measurement variables (NM), (iii) Number of control variables (NC), and (iv) Number of time steps ($nSteps$). EKF algorithm consists of two major sub-parts: (i) a set of nonlinear operations - These operations are executed on a soft processor with full precision and no error is introduced, and (ii) a set of Faddeev operations - These operations are executed on the proposed PolyFSA architecture whose precision ($madd$, $mmul$, $mdiv$) needs to be

Table 5.3: Error (in percentage) associated with output of DFG shown in fig. 5.5(d).

(madd,mmul,mdiv)	Error associated with final output of DFG shown in Figure 5(d)
(16,23,23)	0.002865
(12,23,23)	0.246204
(23,16,23)	0.002649
(23,12,23)	0.042768
(23,23,16)	0.000688
(23,23,12)	0.010761

determined and some error is introduced due to reduction in precision. From the error analysis for Faddeev algorithm presented in the prior section, it is observed that the reduction in precision for the addition operation results in the largest increase in error, followed by the multiply and the division operations. Whenever an option is available, we attempt to reduce the precision of division operation first, followed by the multiply and the addition operations. In this analysis, a system is developed with NS=10, NM=9, and NC=6 and EKF algorithm is applied on this system. This system is defined by Ronnback [78] and the nonlinear operations are developed accordingly.

In the EKF algorithm, *nSteps* can be varied and the overall error in the results of EKF may depend on *nSteps*. Table 5.4 illustrates this variation. Precision of all arithmetic units is set to 16. It is observed that a 1000x increase in *nSteps* results in 4x increase in the mean error. Thus, the effect of *nSteps* on the mean error is found to be negligible. In the remainder of the analysis, *nSteps* is assumed to be equal to 100.

In the proposed research, we aim to determine the set of precision parameters (*madd*,*mmul*,*mdiv*) that will result in a permissible error and also in maximum savings in resource utilization. So, a plot between error and maximum savings in resource utilization is useful to analyze the effect of reduction in precision on both error and overall resource utilization. Figure 5.6 illustrates this plot for both LUTs and FFs. *nSteps* is set to be 100 and *nIter* is set to be 10. Following variables are used to represent error.

$$Error = \frac{\sum_{i=1}^{NS} \left(X(23,23,23) - X(madd, mmul, mdiv)\right)}{NS}. \tag{5.4}$$

$$Mean = \frac{\sum_{i=1}^{nIter} \sum_{i=1}^{nSteps} Error(i,j)}{nIter \times nSteps}. \tag{5.5}$$

$$Variance = \frac{\sum_{i=1}^{nIter} \sum_{i=1}^{nSteps} \left(Error(i,j) - Mean\right)^2}{\sqrt{nIter \times nSteps}}. \tag{5.6}$$

The state variable (X) is updated during every time step of EKF algorithm. In the above equations, *nSteps* represents the total number of time steps in the EKF algorithm,

and *nIter* represents the total number of random iterations that the EKF algorithm is executed. Error is presented in terms of its mean and variance values in figs. 5.6(a) and 5.6(b), respectively. Error in computation of the state variable (X) is found to be large for a few iterations at the beginning of EKF algorithm. As EKF is a learning filter, initial errors are high and contribute to the overall error. In fig. 5.6, it is observed that a 50% reduction in area is obtained by allowing mean error and mean variance to be limited to 1%.

Table 5.4: EKF error analysis for varying number of time steps (precision of all arithmetic units is set to 16).

nSteps	Mean error (in percentage)	Variance error (in percentage)
10	0.0350	0.0000
100	0.0817	0.0006
1000	0.0862	0.0011
10000	0.1262	0.0647

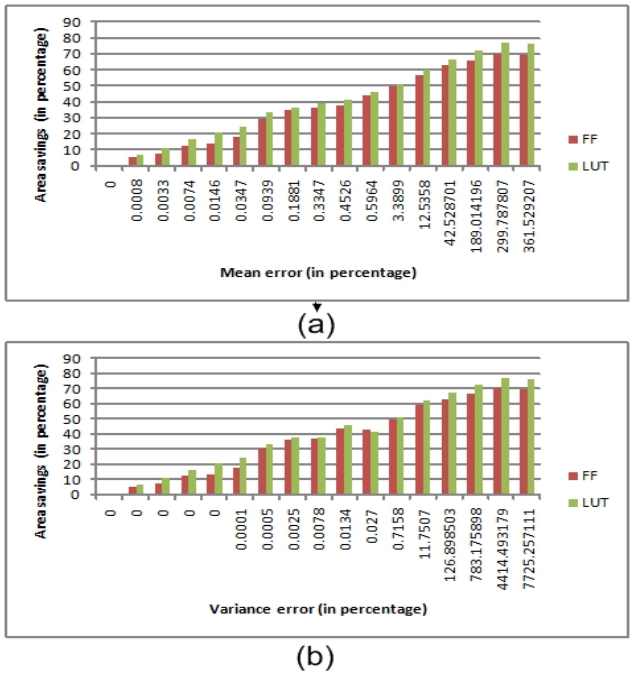


Fig. 5.6: Area savings versus different statistical error parameters: (a) mean error, (b) variance error.

5.1.5 Summary

From the hierarchical error analysis presented in this chapter, we can infer the following.

- Amongst individual arithmetic units, reduction in precision of multiplier unit results in the largest error, and the reduction in precision of divider unit results in the least error.
- In addition to precision of arithmetic units, the number of units with lower precision that affect the final result also have a significant impact on the error associated with the final result. This is reflected in the error analysis of Faddeev algorithm. Reduction in precision of the adder unit results in the largest error, and reduction in precision of divider unit results in the least error.
- From the error analysis of EKF algorithm, it is inferred that variations in number of time steps of EKF does not have a significant impact on the error. It is observed that a 50% reduction in area is obtained by allowing mean error and variance error to be limited to 1%.

In the proposed error analysis techniques, the system used is a real-life system. The system and measurement models were developed for an UAV [78]. However, real-life data sets are not available for the same. So, inputs for the proposed analysis are obtained using a random number generator. In order to mimic the real-life system, the input generation is constrained to obey the laws of physics (relationships between acceleration, velocity, and position are maintained). Also, the analysis has been performed for a wide range of inputs. A final point to note is that the precision analysis techniques have been developed in such a way that it can be re-executed for any test data and the plots can be regenerated for any test case.

5.2 Performance Analysis

This section discusses the effect of architectural parameters on the overall performance of the proposed PolyFSA-based system architecture. Performance is measured in terms of

the overall time taken (in microseconds) to execute a single iteration of EKF algorithm. This section is organized as follows. Section 5.2.1 discusses the motivation to perform performance analysis and presents the proposed performance model. Section 5.2.2 showcases the variations in performance based on Faddeev parameter, and number of PEs. Section 5.2.3 discusses the variations in execution time of Faddeev algorithm for varying latency of arithmetic units, and input rate of divider. Section 5.2.4 discusses the variations in execution time of overall EKF algorithm for varying latency of arithmetic units, and input rate of divider. Section 5.2.5 analyzes the variations in maximum clock frequency, and overall time taken (in microseconds) for varying architectural parameters. Section 5.2.6 summarizes this analysis.

5.2.1 Overview of Performance Model and Motivation for Performance Analysis

To predict the acceleration in performance of EKF algorithm on the proposed architecture for multiple problem sizes, variations in architecture parameters, and variation in number of PEs, a performance model is proposed. Overall execution time is expressed in terms of four individual timings that are listed below. Equation 5.7 shows the model for overall execution time (for any processor/co-processor architecture).

- T_{config} - Time required to set-up the architecture. This time is specific to the reconfiguration method, type of FPGA, and configuration bitstream size.
- $T_{microprocessor}$ - Time required to run the non-accelerated portions of the EKF algorithm on the microprocessor. This time is specific to the overlying application characteristics and is measured by profiling the application on the MicroBlaze.
- $T_{datatransfer}$ - Time required to transfer data between microprocessor memory and data buffer. This time is computed using (i) total amount of data transferred, and (ii) latency of data transfer between microprocessor and data buffer.

- $T_{co-processor}$ - Time required to run the accelerated portions of the algorithm on the co-processor. This time depends on the problem size, number of PEs dedicated to the EKF algorithm and architectural parameters (pipeline depth, etc).

$$T_{overall} = T_{config} + T_{microprocessor} + T_{datatransfer} + T_{co-processor} \tag{5.7}$$

Equation 5.8 is used to model the overall performance of EKF on the proposed system architecture.

$$T_{overall} = T_{config} + nSteps \times [T_{datatransferperstep} + \sum_{i\epsilon\{1,2,5,6\}} T_{mbprofile}(function_i) + \sum_{i\epsilon\{3,4,7,8,...,13\}} T_{cpsched}(function_i)] \tag{5.8}$$

In this equation, *nSteps* is number of iterations of EKF data flow (shown in fig. 4.1 in Chapter 4) executed before reconfiguring the system. $T_{datatransferperstep}$ is time required to transfer data between microprocessor and data buffer during a single iteration of EKF data flow. $T_{mbprofile}(function_i)$ is the time spent on the microprocessor to execute the i^{th} function (i is the index number of the function as shown in fig. 4.1). This is obtained by profiling the source code execution on a MicroBlaze. $T_{cpsched}(function_i)$ is the time spent on the co-processor (FSA) to execute the i^{th} function. This number is obtained by scheduling the Faddeev algorithm onto the co-processor architecture.

Table 5.5 lists the Faddeev parameters and the amount of data that is transferred between the microprocessor memory and the data buffer for all the accelerated functions. In this table, Number of States (NS) and Number of Measurements (NM) depend on the overlying application for which the EKF algorithm is used. $T_{datatransferperstep}$ (refer eq. 5.8) is computed by using the formula provided in eq. 5.9 and the data provided in Table 5.5. In this equation, NR represents total number of words read from the cache and NW represents total number of words written into the cache (for one iteration of EKF).

Table 5.5: Faddeev parameters and data transfer for accelerated functions in EKF.

Function Index	Faddeev Parameters	Data to be written to cache (*in words*)	Data to be read back from cache (*in words*)
(*i*)	*<N,M,P>*	(NW)	(NR)
3	*<NS,NS,NS>*	*NSxNS*	*0*
4	*<NS,NS,NS>*	*0*	*0*
7	*<NS,NS,NM>*	*NSxNM*	*0*
8	*<NM,NS,NS>*	*0*	*0*
9	*<NM,NS,NM>*	*0*	*0*
10	*<NM,1,1>*	*NM*	*0*
11	*<NS,NM,NS>*	*0*	*0*
12	*<NS,NM,NS>*	*0*	*0*
13	*<NS,NM,1>*	*0*	*NS*

$$T_{datatransferperstep} = T_{datareadperword} \times NR + T_{datawriteperword} \times NW \tag{5.9}$$

EKF algorithm consists of nine accelerated functions that can be defined as parameterized Faddeev algorithms (discussed in Chapter 4). Each of these functions can be defined by a set of Faddeev parameters $< M, N, P >$. Table 5.5 lists the set of Faddeev parameters for all $T_{cpsched}(function_i)$ (i ϵ $\{3, 4, 7, 8, 9, 10, 11, 12, 13\}$). Remainder of this section discusses the performance model of proposed Polymorphic Faddeev Systolic Array (PolyFSA) that will be utilized to predict the time taken to run a single instance of the Faddeev algorithm $T_{cpsched}(function_i)$ in eq. 5.8).

Each Processing Element (PE) in the proposed PolyFSA architecture consists of a mixture of pipelined and non-pipelined functional units with variable latencies and ordering of computation is complicated by the use of FIFOs in front of PEs. Use of FIFOs makes the task of predicting the run-time performance of the architecture a lot harder and it is

cumbersome to derive a mathematical formula to represent the performance.

There is a need for a scheduling algorithm that can predict the data flow through the FSA architecture during run-time and thus predict co-processor execution time. In this research, As-Soon-As-Possible (ASAP) scheduling algorithm is used due to the presence of FIFOs. Before scheduling, it is required that the Faddeev algorithm is represented in the form of a Data Flow Graph (DFG). Figure 5.7 presents the algorithm used to create a fully unrolled DFG for the Faddeev algorithm that is illustrated in fig. 4.2. Each node in the DFG is comprised of the following fields.

- *NodeType* - Specifies if the node is a boundary or internal node (as discussed in Chapter 4).
- *startTime* - Set for nodes that require an input from the data buffer. Data buffer access is sequential, thus certain nodes need to wait for their inputs before they can be executed.
- *ResType* - Specifies the PE on which the node is executed.
- *schedTime* - Set to -1. This value is updated with the time during which the node is scheduled.

Number of nodes in the DFG generated by this algorithm is computed as shown in eq. 5.10 and is found to be $O(N^3)$ (M=P=N). Number of edges is found to twice the number of nodes.

$$NumNodes = (N + P) \times [N \times (N + M) - N \times (N - 1)/2]. \tag{5.10}$$

Figure 5.8 presents the algorithm used to schedule the DFG created using the algorithm presented in fig. 5.7. PolyFSA architecture consists of R PEs, and each PE contains a boundary cell and an internal cell. A ready set is maintained for each of these $2R$ resources and is generated during every clock cycle. Ready set for a particular resource contains the set of nodes that can be scheduled onto that resource during a given clock cycle. Once

```
Algorithm CreateFaddeevDFG
Inputs:
1.    List of Faddeev parameters <N,M,P>
2.    Number of PEs <R>
3.    Latencies of boundary and internal nodes <LatB,LatI>
Output:
1.    A populated graph data structure in terms of list of nodes and edges
      a.    NodeList[MaxNodes] with each node containing the following fields
            {NodeType,startTime,ResType,schedTime}
      b.    EdgeList[MaxEdges] with each edge containing the following fields
            {SrcNode,DestNode}

Step 0: start = 0; count1 = 0; count2 = 0;
Step 1: for k = 1 to N+P do steps 2-8
Step 2: for i = 1 to N do steps 3-7
Step 3: for j = 1 to N+M-i do step 4-7
Step 4: NodeList[count1].NodeType = (j!=0)
Step 5: NodeList[count1].ResType = resAlloc[i]
Step 6:     if (i==0)
                NodeList[count1].startTime = start+j;
            else
                NodeList[count1].startTime = 0;
Step 7: NodeList[count1++].schedTime = -1;
Step 8: start = start + N + M
Step 9: count1 = 0;
Step 10: for k = 1 to N+P do steps 11-15
Step 11: for i = 1 to N do steps 12-15
Step 12: for j = 1 to N+M-i do step 13-14
Step 13: EdgeList[count2].src = count1 + j;
Step 14: EdgeList[count2++].dest = count1 + j + 1;
Step 15: Step 15: count1 = count1 + N + M - i;
Step 16: countsrc = 0;  countdest = (N+M);
Step 17: for k = 1 to N+P do steps 18-23
Step 18: for i = 1 to N-1 do steps 19-22
Step 19: for j = 1 to N+M-i do step 20-21
Step 20: EdgeList[count2].src = countsrc + j + 1;
Step 21: EdgeList[count2++].dest = countdest + j;
Step 22: countsrc = countsrc + N + M - i; countdest = countdest + N + M - i - 1;
Step 23: countsrc = countsrc + N + M - i; countdest = countdest + N + M;
```

Fig. 5.7: Algorithm for creating unrolled Data Flow Graph (DFG) for Faddeev algorithm. This DFG serves as input to ASAP scheduler outlined in fig. 5.8.

all the ready sets are generated, a candidate node from the ready set is selected randomly and scheduled. This operation is repeated for all the ready sets. Once a candidate node is scheduled, *startTimes* of some of its dependent nodes need to be modified. Two types of dependent nodes are found in the DFG and are listed below.

- Data dependent nodes - Nodes that need to wait for the candidate node's result to be available. Waiting time is equal to the latency of the resource on which the candidate node is executed.

```
Algorithm ASAPScheduleFaddeevDFG
Input:
1.    A populated graph data structure in terms of list of nodes and edges
      a.    NodeList[MaxNodes] with each node containing the following fields
            {NodeType,startTime,ResType,schedTime}
      b.    EdgeList[MaxEdges] with each edge containing the following fields
            {SrcNode,DestNode}
2.    Number of PEs <R>
3.    Latencies of boundary and internal nodes <LatB,LatI> and Input rate of divider (c_rate)
            LatB = LatDiv and LatI = LatMul + LatAdd
4.    Number of nodes and edges <nNodes,nEdges>
Output:
1.    Schedule length <SchedLength>

Step 0: schedCount = 0; currTime = 0;
Step 1: Repeat steps while (schedCount < nNodes)
Step 2: for I = 1 to nNodes do steps 3-9
Step 3: if ((NodeList[i].startTime <= currTime) AND (NodeList[i].schedTime = -1)) do steps 4-9
Step 4: Ready = 1;
Step 5: for j = 1 to nEdges do step 6
Step 6: if ((EdgeList[j].dest == i) AND (NodeList[EdgeList[j].src].schedTime == -1))
            Ready = 0;
Step 7: t = NodeList[i].ResType;
Step 8: if ((NodeList[i].NodeType == 0) AND (Ready == 1))
            Add node index 'i' to ready set of the boundary resource type 't'
Step 9: if ((NodeList[i].NodeType == 0) AND (Ready == 1))
            Add node index 'i' to ready set of the internal resource type 't'
Step 10: for i = 1 to R do steps 11-22
Step 11: if (ready set of boundary resource type 'i' is not empty) do steps 12-17
Step 12: t = A random node in ready set. Ready set is now emptied.
Step 13: NodeList[t].schedTime = currTime;
Step 14: for j = 1 to nEdges do step 15
Step 15: if ((EdgeList[j].src = t) AND (NodeList[edgelist[j].dest].startTime < currTime + LatB))
            NodeList[EdgeList[j].dest].startTime = currTime + LatB;
Step 16: for j = 1 to nNodes do step 17
Step 17: if ((NodeList[j].ResType = i) AND (NodeList[j].startTime < currTime + c_rate))
            NodeList[j].startTime = currTime + c_rate;
Step 18: if (ready set of internal resource type 'i' is not empty) do steps 19-22
Step 19: t = A random node in ready set. Ready set is now emptied.
Step 20: NodeList[t].schedTime = currTime;
Step 21: for j = 1 to nEdges do step 22
Step 22: if ((EdgeList[j].src = t) AND (NodeList[EdgeList[j].dest].startTime < currTime + LatI))
            NodeList[EdgeList[j].dest].startTime = currTime + LatI;
Step 23: SchedLength = currTime + LatI;
```

Fig. 5.8: Algorithm for scheduling the Faddeev algorithm Data Flow Graph (DFG) using ASAP scheduler.

- Resource dependent nodes - Nodes that share the resource with the candidate node and will need to wait for the candidate node to complete its execution. If the shared resource is an internal cell, then the waiting time is only a single clock cycle, as the resource is pipelined. If the shared resource is a boundary cell, then the waiting time is equal to the latency of the boundary cell, as the boundary cell is not pipelined.

Process of generating the ready sets and scheduling the candidate nodes is repeated till all the nodes in the unrolled DFG are scheduled. Complexity of algorithm is found

to be $O(N^6)$ (M=P=N). Once all $T_{cpsched}(function_i)$ (i ϵ $\{3, 4, 7, 8, 9, 10, 11, 12, 13\}$) are estimated, the values can be utilized to determine overall execution time of EKF on proposed architecture.

From the proposed performance model and the EKF algorithm characteristics, it is observed that the execution of Faddeev algorithm occupies the major portion of the overall execution time, assuming that reconfiguration is rarely performed. Sections 5.2.3 and 5.2.4 showcase the Faddeev algorithm execution time ($T_{Faddeev}$). Sections 5.2.5 and 5.2.6 analyze the performance of PolyFSA for the overall EKF algorithm. From the performance model, it is observed that $T_{Faddeev}$ depends on the following factors:

- Faddeev parameter *(M=N=P)*,
- Number of PEs in the PolyFSA (R),
- Latency of the divider unit ($LatDiv$),
- Latency of the adder unit ($LatAdd$),
- Latency of the multiplier unit ($LatMul$),
- Input rate of the divider unit (c_rate).

Variations of architectural parameters have a major impact on the area required to implement the PolyFSA architecture. Proposed performance analysis is required to study the variations in overall execution time and determine the architectural options that provide the best performance for a given area constraint. Remainder of this section discusses the effect of the factors listed above on $T_{Faddeev}$ and $T_{overall}$. Also, any modifications in the resource requirements are also discussed.

5.2.2 Variations in Overall Execution Time of Faddeev Algorithm (in Clock Cycles) for Varying Faddeev Parameter and Number of PEs

Faddeev parameter determines the size of the inputs that need to be processed and dictates the computational complexity of the Faddeev algorithm. Theoretical estimate for

computational complexity is $O(N^3)$, where N is the Faddeev parameter (for sake of clarity in discussion, M=N=P). Proposed performance model is used to estimate the time taken to execute an iteration of the Faddeev algorithm ($T_{Faddeev}$) for varying N ($1 \leq N \leq 10$) and varying R ($2 \leq R \leq 10$). In EKF algorithm, N is typically set to be equal to either the number of states or the number of measurements in the spacecraft navigation system and these numbers seldom exceed a value of 10. For the current implementations of proposed PolyFSA on Xilinx Virtex 4 family of FPGAs, it is possible to fit 5-10 PEs in the entire chip. Hence, R is limited to 10. Range of values for R and N can easily be increased in this experiment.

Schedules are generated for different values of R and N and the resulting plot of execution times is shown in fig. 5.9. In this analysis, the latencies of individual arithmetic units and the input rate of the divider are maintained at a value of 4. It is observed that the speed-up (when compared against R=2) is less than $R/2$ for most of the schedules. This is attributed to sequential memory accesses from/to the data buffer.

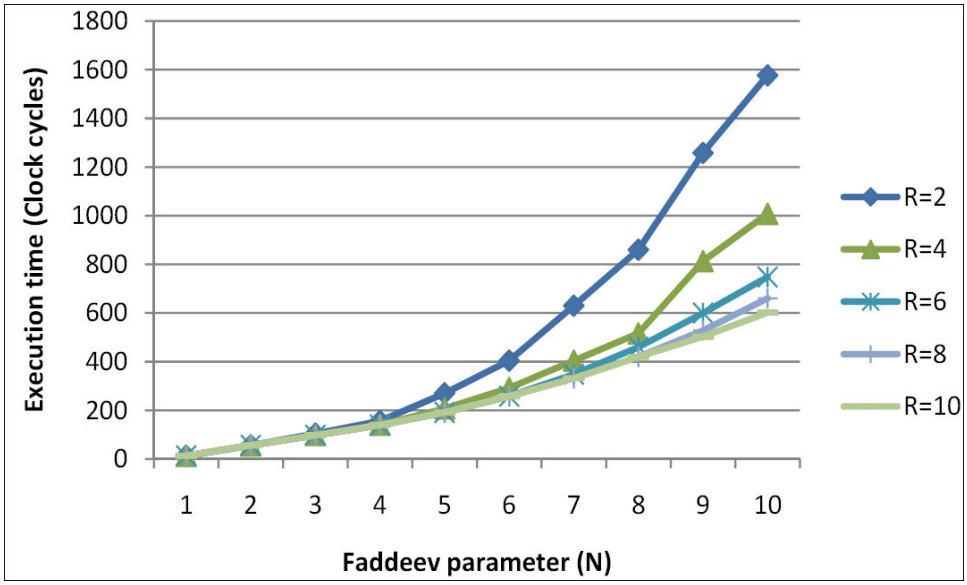


Fig. 5.9: Estimated performance of PolyFSA for varying problem sizes (M=N=P) and number of PEs (R). Timing is measured for a single Faddeev operation.

5.2.3 Variations in Overall Execution Time of Faddeev Algorithm (in Clock Cycles) for Varying Latency of Arithmetic Units and Input Rate of Divider

From fig. 4.2, it is observed that the Faddeev algorithm consists of two types of operations: (i) boundary node operation, and (ii) internal node operation. Computation performed inside a boundary node consists of a solitary division operation. Computation performed inside the internal node consists of a multiply operation followed by an addition operation. Critical path latency of the Faddeev DFG depends on the latency of the three arithmetic units and the number of boundary node and internal nodes in the critical path of the DFG. Critical path latency of any DFG is one of the major factors contributing to the overall execution time of the DFG. Figure 5.10 illustrates the effect of the latency of individual arithmetic units on the overall execution time ($T_{Faddeev}$). N and R are set to be equal to 10 and 5, respectively. For the plot showing the variation in performance-based on the latency of one arithmetic unit, the latencies of other two arithmetic units are set to be 4. Input rate of the divider is also set to be 4. In this plot, it is observed that the variations in latencies of adder and multiplier have an equal impact on $T_{Faddeev}$. Also, variation in the latency of adder (or multiplier) unit has a greater impact on $T_{Faddeev}$ than the variation in latency of divider unit. Number of internal nodes in the critical path of the Faddeev DFG is much larger than the number of boundary nodes. Hence, the latency of the internal unit (sum of the latencies of adder and multiplier unit) has a more pronounced effect on the overall execution time. From the plots shown in fig. 5.10, it is concluded that reduction in latencies of the adder and multiplier units helps to reduce the execution time and improve performance.

Figure 5.11 illustrates the variation in $T_{Faddeev}$ for varying input rate of the divider unit (c_rate). It is observed that there is minimal variation in $T_{Faddeev}$ for smaller values of c_rate ($c\_rate \leq 11$). For higher values, there is a significant variation in the performance for variation in c_rate. For this analysis, latency of all arithmetic units is set to be 4. N and R are set to be equal to 10 and 5, respectively. From this analysis, we conclude that

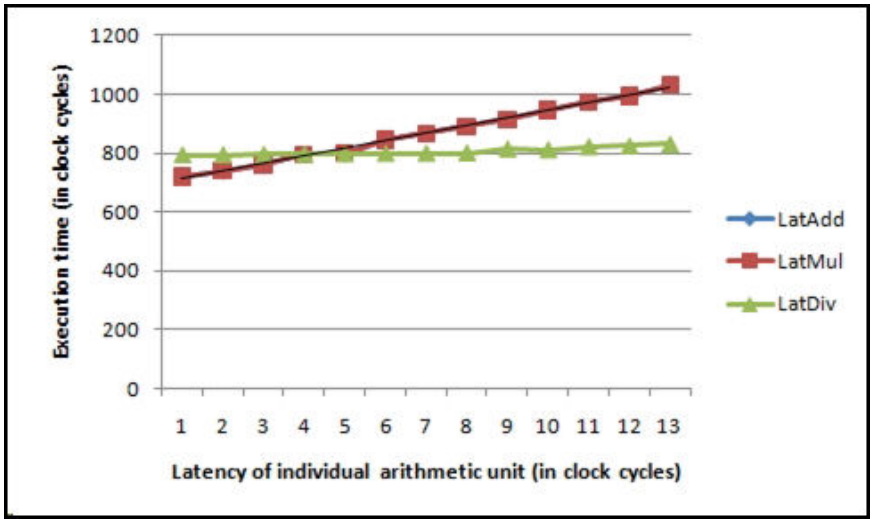


Fig. 5.10: Estimated performance of PolyFSA for varying latencies of individual arithmetic units.

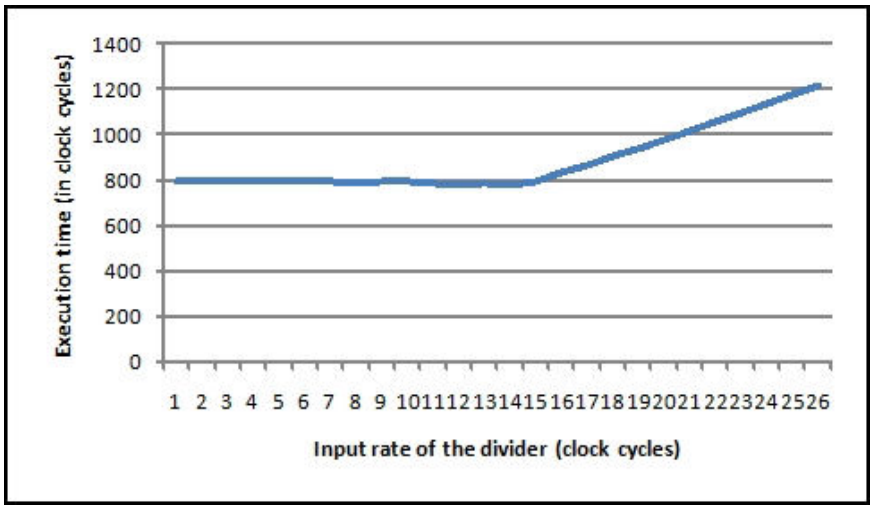


Fig. 5.11: Estimated performance of PolyFSA for varying input rate of divider unit.

reduction in c_rate results insignificant improvement in performance until c_rate becomes lesser than a value of 14.

5.2.4 Variations in Overall Execution Time of EKF (in Clock Cycles) for Varying Latency of Arithmetic Units and Input Rate of Divider

Ronnback [78] discusses a spacecraft navigation system with NS=10 and NM=9. Nonlinear functions are executed on the Microblaze and the overall execution time of the EKF algorithm is computed using eq. 5.8 for a single iteration and zero reconfiguration overhead. Figures 5.12 and 5.13 illustrate the variations in the overall execution time for variations in latencies of arithmetic units and variations in the input rate of the divider, respectively. These plots were generated in a fashion similar to the generation of the plots in figs. 5.10 and 5.11. A similar trend in variation of performance is noticed in both the plots. Total execution time of the nonlinear portion is found to be 4103 clock cycles (from profiling) and total time for data transfer is found to be 216 clock cycles (calculated using equation 5.9).

5.2.5 Variations in Overall Execution Time of EKF (in Microseconds) for Varying Latency of Arithmetic Units and Input Rate of Divider

Section 5.2.3 analyzes the performance of PolyFSA architecture in terms of number of clock cycles required to execute the EKF algorithm. To calculate the wall clock time, we need to analyze the maximum clock frequency at which the architecture can operate at.

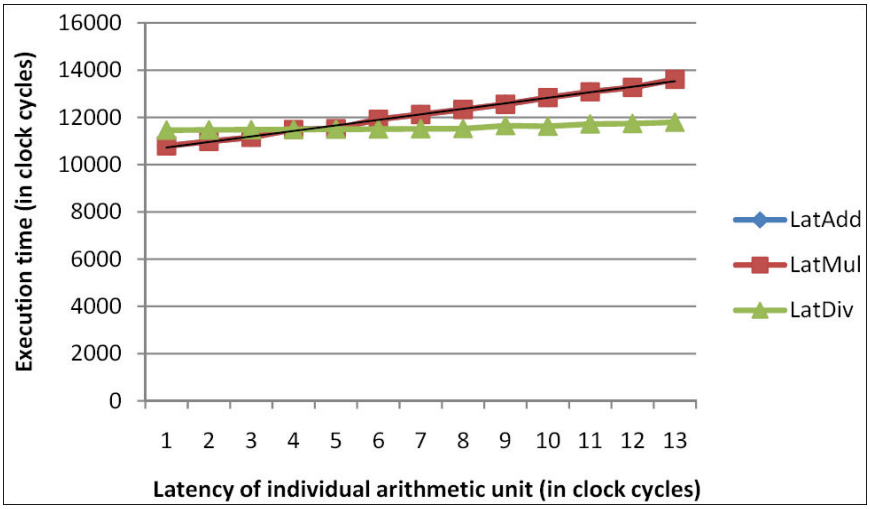


Fig. 5.12: Estimated performance of PolyFSA for varying latencies of individual arithmetic units.

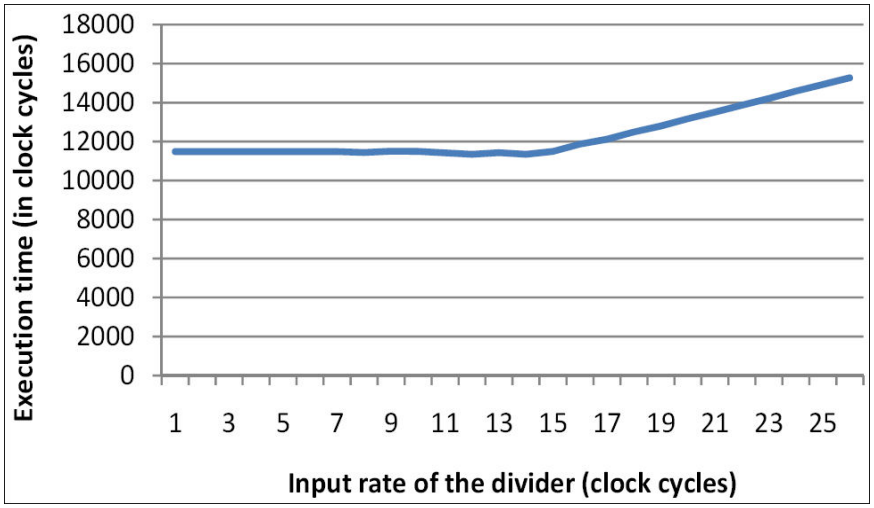


Fig. 5.13: Overall execution time of EKF for varying input rate of the divider unit.

Equation 5.8 is now modified to incorporate three additional parameters: (i) configuration clock frequency (F_{config}), (ii) maximum clock frequency for the microblaze ($F_{mb} = 150$ MHz for Virtex 4 SX35), and (iii) maximum clock frequency for the PolyFSA unit ($F_{PolyFSA}$). Equation 5.11 represents the total time taken (in microseconds) to execute the EKF algorithm and the clock frequency is represented in MHz. While F_{config} and F_{mb} are dictated by the FPGA device properties, $F_{PolyFSA}$ is a parameter that is affected by the choice of the arithmetic units selected to perform the operations. Remainder of the PolyFSA architecture is comprised of simple logic and can be assumed to run at high clock frequencies. Figure 5.14 illustrates the variations in maximum clock frequencies of individual arithmetic units for variations in their respective latencies. Figure 5.15 illustrates the variations in maximum clock frequency of divider unit for variations in its input rate. From fig. 5.14, it is observed that the maximum clock frequency of divider unit does not vary with variations in its latency.

$$T_{overall} = F_{config} \times T_{config} +$$
$$nSteps \times \left[\begin{array}{c} F_{mb} \times \left(T_{datatransferperstep} + \sum_{i \epsilon \{1,2,5,6\}} T_{mbprofile}(function_i) \right) \\ + F_{PolyFSA} \times \left(\sum_{i \epsilon \{3,4,7,8,\ldots,13\}} T_{cpsched}(function_i) \right) \end{array} \right] \quad (5.11)$$

In fig. 5.16, we illustrate the variations in overall time taken (in microseconds) for variations in latency of individual arithmetic units. This plot is obtained by using eq. 5.11. In Figure 5.17, we illustrate the variations in overall time taken (in microseconds) for variations in input rate of divider. In this analysis, $F_{PolyFSA}$ is defined as the minimum of the maximum clock frequencies at which the adder, multiplier, and the divider can operate. For example, we set *LatAdd*, *LatMul*, and *LatDiv* to 13, 8, and 8, respectively. This combination results in a PolyFSA architecture that can be run at a maximum clock frequency of 335.2 MHz. From figs. 5.16 and 5.17, it is observed that the best performance that can be achieved is equal to 53 microseconds (for $LatMul = 8$). In the proposed research, we aim to determine the set of latencies (*LatAdd*, *LatMul*, and *LatDiv*) that will result in the required performance and also translate into maximum savings in resource utilization.

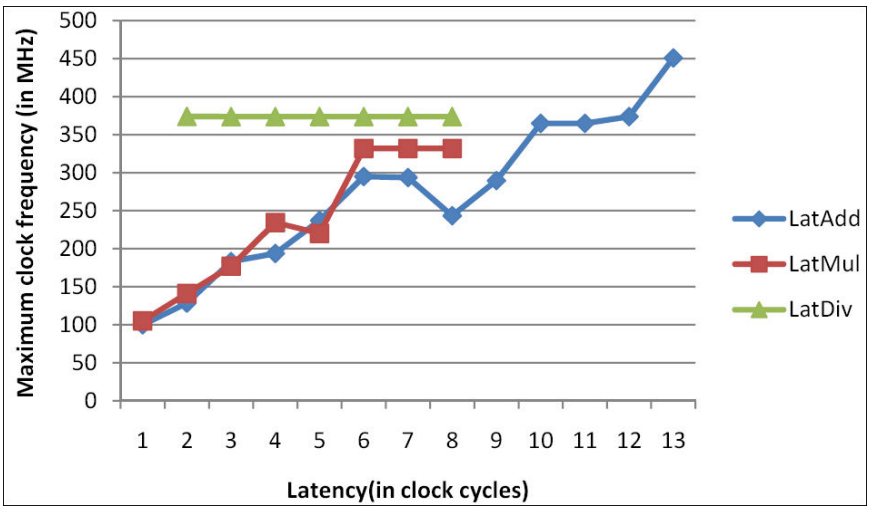


Fig. 5.14: Variations in maximum clock frequency of individual arithmetic units for variations in their respective latencies.

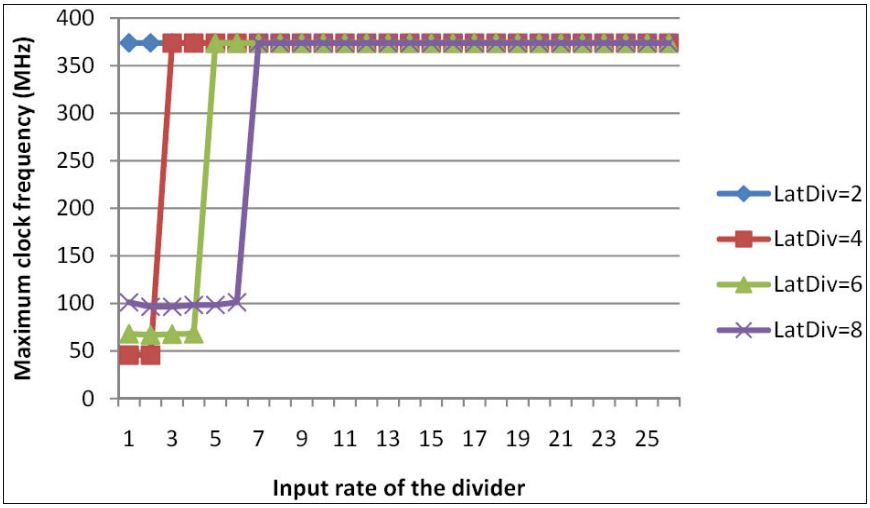


Fig. 5.15: Variations in maximum clock frequency of individual arithmetic units for variations in input rate of divider.

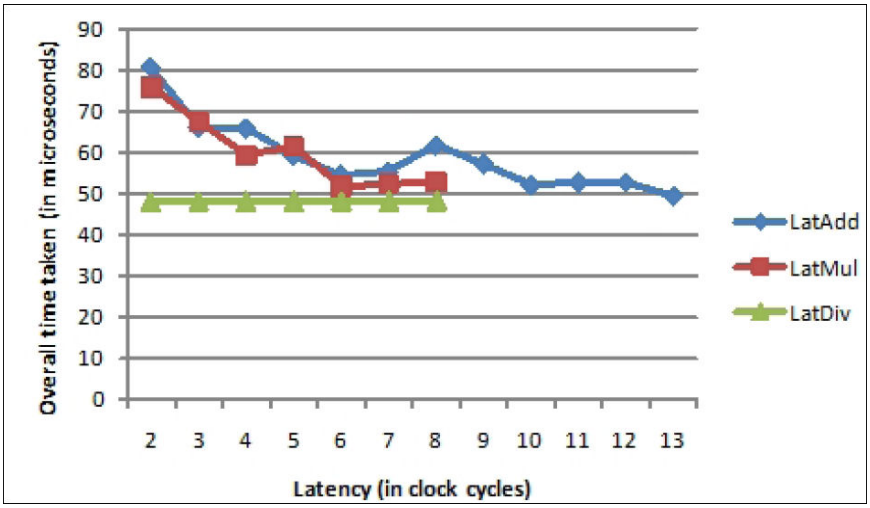


Fig. 5.16: Variations in overall time taken (in microseconds) for variations in latencies of individual arithmetic units.

So, a plot between overall time taken for EKF (in microseconds) and maximum savings in resource utilization is useful to analyze the effect of variation in latency on both performance and overall resource utilization. Figure 5.18 illustrates this plot for both LUTs and FFs.

5.2.6 Summary

From the performance model of EKF presented in this section, it is observed that the execution time of Faddeev algorithm ($T_{Faddeev}$) has a significant impact on the overall execution time of EKF. In this section, the effect of varying the problem size (in terms of Faddeev parameter) and varying architectural features of PolyFSA on $T_{Faddeev}$ is analyzed and results are presented. Two of the major architectural features that have the greatest impact on the performance are: (i) number of PEs in PolyFSA (R), and (ii) input rate of the divider unit. Also, variations in latency of the adder and multiplier units also have some impact on the overall performance. Variation of latency of divider unit does not have any impact on $T_{Faddeev}$.

5.3 Area Analysis

This section discusses the variations of area required by the proposed system-level architecture for accelerating EKF algorithm, when various architectural parameters are varied. In Chapter 4, we discussed the top-level system architecture in detail. There are two high-level features in this architecture that can be varied and analyzed for effects on area, performance, and error. These features are: (i) number of PEs (nPE) in the PolyFSA; (ii)

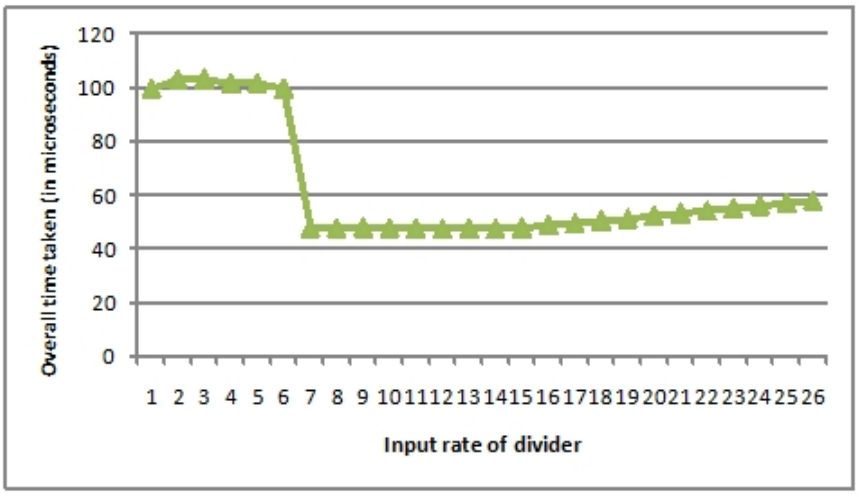


Fig. 5.17: Variations in overall time taken (in microseconds) for variations in input rate of divider.

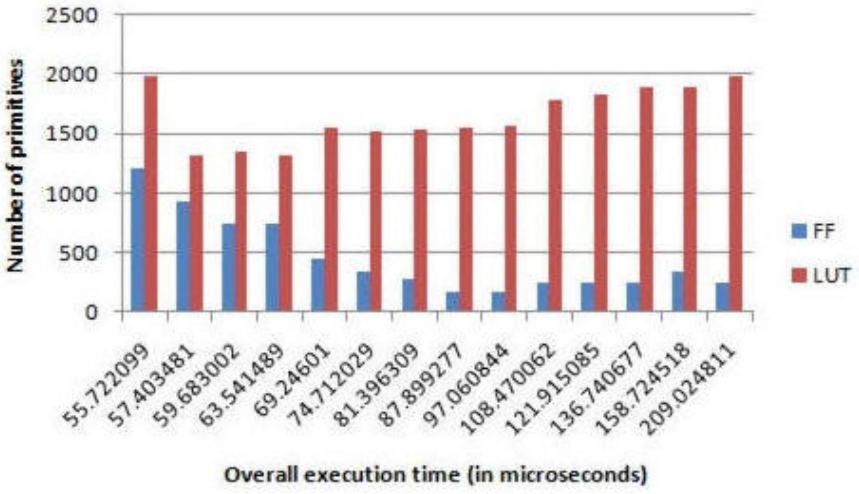


Fig. 5.18: Plot of area versus performance.

design of each PE, assuming all the PEs are identical. Design of each PE, in turn, depends on the following architectural parameters.

- Latency and precision of adder unit (LatAdd and madd);
- Latency and precision of multiplier unit (LatMul and mmul);
- Latency, precision and input rate of divider unit (LatDiv, mdiv and c_rate).

Figures 4.8 and 4.9 illustrate the design of the boundary PE and internal PE. It is observed that the arithmetic units form a major part of the overall PE design. Remainder of the design consists of a set of registers and control logic. Each PE also requires six Block RAMs (four for FIFOs and two for the local memories). Xilinx FPGAs typically contain nearly 200 Block RAMs, and hence the Block RAMs do not constrain the system design. In this analysis, the area requirement of a single PE is approximated to the sum of area requirements of all the three arithmetic units.

In this research, area is represented by two types of FPGA resources: Flip-Flops (FFs) and lookup tables (LUTs). Embedded resources are not considered in order to reduce the complexity of this analysis. All the numbers are obtained by developing the arithmetic unit using the floating-point Intellectual Property (IP) core library provided by Xilinx. Each

core is instantiated with the required architectural parameters and post-synthesis results are used to obtain the number of LUTs and FFs required.

Remainder of this section discusses the variation in area requirements of a single PE by varying the architectural parameters for a particular arithmetic unit. While varying a particular set of parameters, the other parameters are kept at *optimum* values that will result in zero error and fastest clock frequency. Following are the values for each parameter: (i) LatAdd = 13, (ii) LatMul = 11, (iii) LatDiv = 8, (iv) c_rate = 1, and (v) madd = mmul = mdiv = 23.

Figure 5.19 illustrates the variations in area for varying the architectural parameters of the adder unit. It is observed that the variations in latencies do not affect the area by a significant factor. From the two plots shown in this figure, it can be concluded that reduction in latency results in a large reduction in the number of FFs and reduction in precision results in a large reduction in the number of LUTs. A 50% reduction in number of FFs is observed for LatAdd equal to 7 (for madd $\leq$ 25) and a 50% reduction in number of LUTs is observed for madd $\leq$ 18 (for all LatAdd). Another interesting observation is that the number of LUTs decreases with increase in latency, albeit by a small amount. A similar trend is observed in variations of area for varying latency and precision of multiplier and divider. Plots for multiplier and divider are presented in figs. 5.20 and 5.21, respectively.

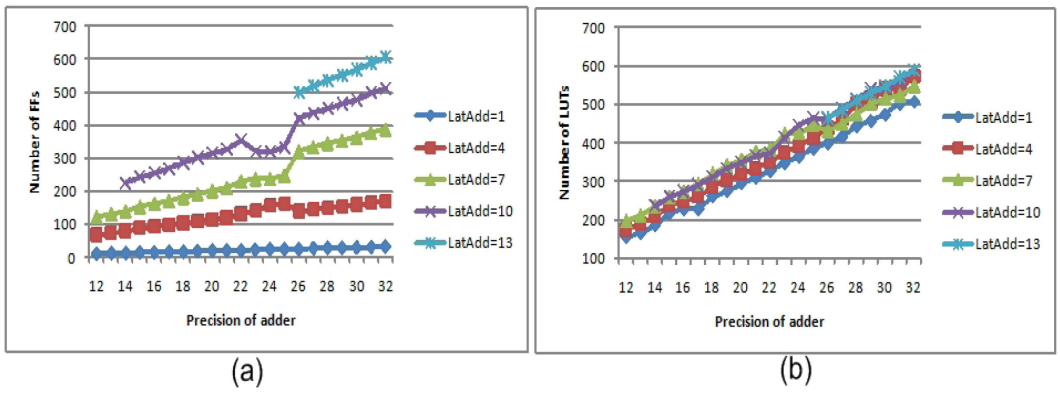


Fig. 5.19: Variation in area for varying architectural parameters of adder unit: (a) variation in number of FFs, (b) variation in number of LUTs.

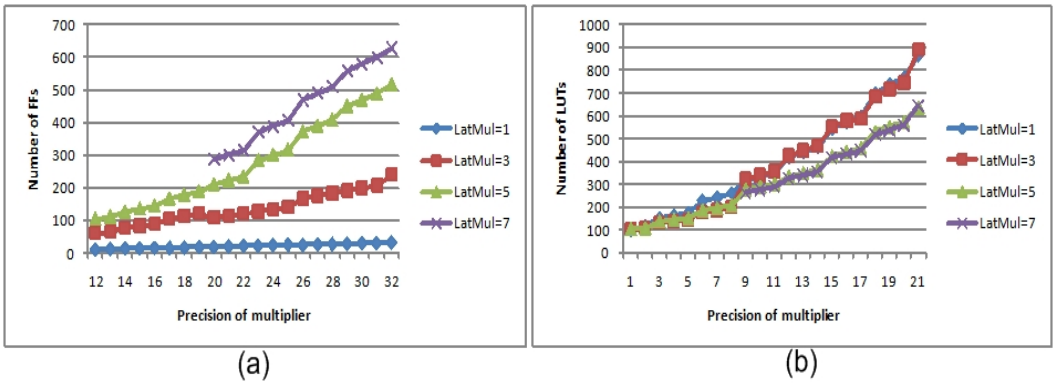


Fig. 5.20: Variation in area for varying architectural parameters of multiplier unit: (a) variation in number of FFs, (b) variation in number of LUTs.

In addition to latency and precision, divider unit has an additional architectural parameter that can be varied (c_rate). Logic governing the division operation consists of recurring sub-steps. Each stage in a pipelined divider unit can be used to operate on the same data for multiple clock cycles. The parameter that governs the number of clock cycles that each stage operates on a single data is labeled as c_rate. Figure 5.22 showcases the variations in the area required by the divider unit for varying c_rate. It is observed that there is a large reduction in the number of LUTs and FFs for increase in c_rate. Area reduces by a factor of 8, when c_rate increases from 1 to 7. Also, increase in c_rate beyond a value of 7 does not cause any variation in the area.

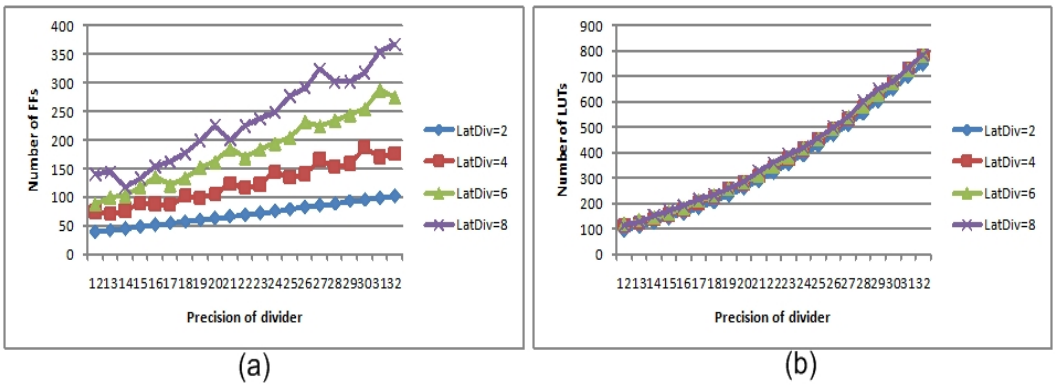


Fig. 5.21: Variation in area for varying architectural parameters of divider unit: (a) variation in number of FFs, (b) variation in number of LUTs.

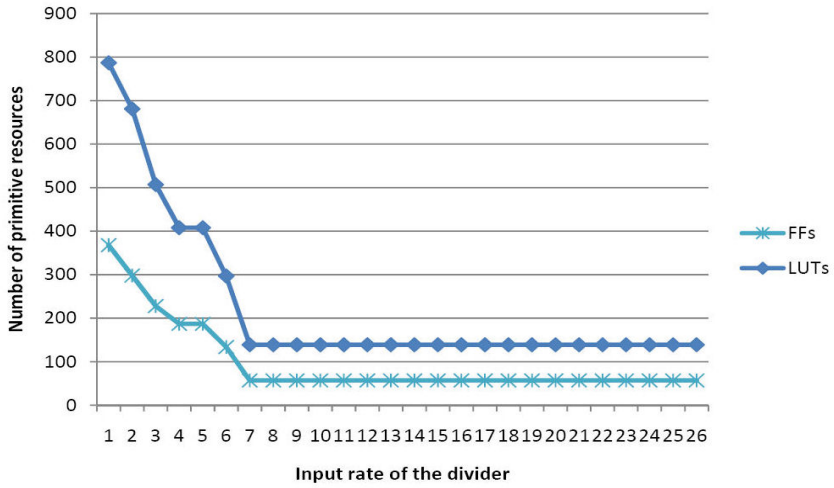


Fig. 5.22: Variation of area for varying input rate of the divider.

Chapter 6

Results and Analysis

This chapter presents the results that are used to evaluate the proposed PolyFSA architecture. In sec. 6.1, the proposed PolyFSA architecture along with the static design (host processor + controller + cache) is implemented, placed, and routed on a Xilinx Virtex 4 FPGA, and results are showcased. Section 6.2 uses the proposed performance model to compare the performance achieved using proposed PolyFSA against two other hardware accelerators. Section 6.3 presents the 3-D pareto curve (area versus performance versus error) obtained as an end-product of the proposed architectural analysis.

6.1 FPGA Implementation

Proposed PolyFSA-based system architecture is developed on Xilinx Virtex 4 SX35 ML-402 board and test cases are executed at a clock frequency of 100 MHz. Resource utilization for a single PE and the static region are presented in Table 6.1.

Performance of proposed design is compared to software implementations on a Virtutech Simics PowerPC 750 simulator [81] running at 150 MHz (equivalent to the embedded RAD750 used in many space applications). Test case for the EKF algorithm was developed for the example on an autonomous Unmanned Air Vehicle (UAV)-based space application [78] and related parameters are set as follows: (a) Number of States (NS) = 10, and

Table 6.1: Resource utilization for the static region and PolyFSA PE.

Design module	LUT	FF	XtremeDSP/DSP48E	BRAM (18Kb each)
Static Design (Microblaze+Cache+Controller)	6171	2822	7	86
PolyFSA PE	1185	1060	8	5
Available resources	30720	30720	192	192

(b) Number of Measurements (NM) = 9.

Figure 6.1 shows the overall execution time of both the algorithms on the proposed architecture and the software platform. Proposed architecture outperformed the software implementation by 4.18x. It is noted that the simulator for the PowerPC 750 is overly optimistic because it does not model memory latencies or cache performance. Therefore, performance numbers for actual execution on a PowerPC 750 device is expected to be worse, giving the proposed design an even better speedup.

Execution time for EKF is further analyzed. It was observed that 45% of the time is spent controlling accelerated functions, 25% is spent doing non-accelerated functions, and 29% is spent transferring data to or from the co-processor. It was also observed that 45% of the time is spent on the microprocessor and 55% on the accelerator.

Figure 6.2 shows the number of cycles taken to complete one iteration of the Faddeev algorithm on the PolyFSA for a varying number of PEs (1 to 5) and matrices A, B, C, D with equal dimensions N=M=P. Each of the PolyFSA architecture is synthesized, placed, and routed on the FPGA.

Power consumption is a major factor in spacecrafts and it is imperative to showcase

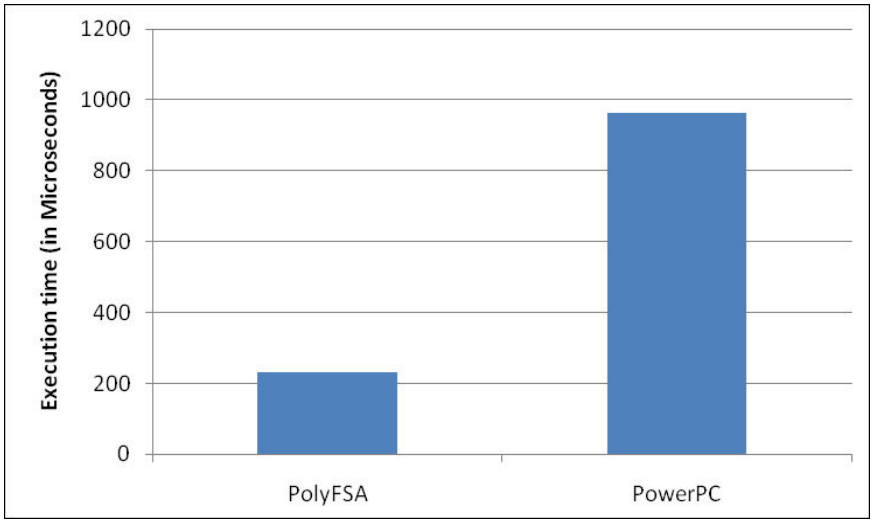


Fig. 6.1: Comparison of performance of proposed PolyFSA-based system architecture implemented on a FPGA against a software only implementation on a simulated PowerPC 750.

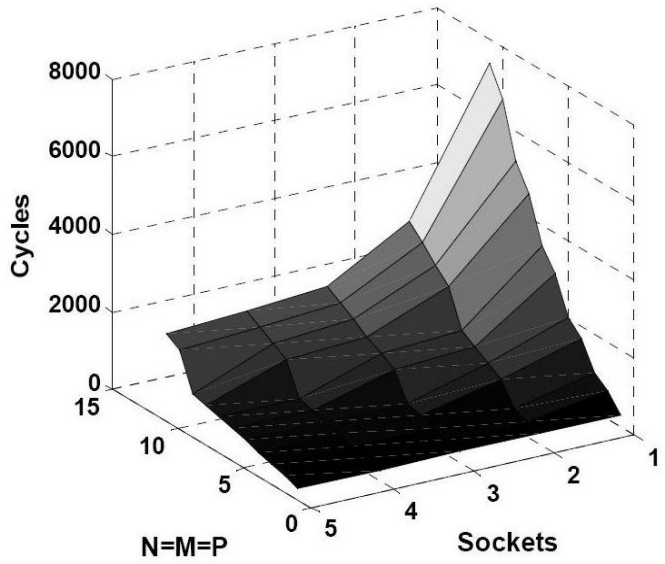


Fig. 6.2: Measured performance (in cycles) of PolyFSA for a varying Faddeev matrix size (N=M=P) and available sockets (R).

power consumed by EKF execution on PolyFSA. Table 6.2 presents the power consumption for various parts of the proposed design. These numbers are estimated using the XPower estimation tool provided by Xilinx. Overall power consumption is found to be around 1W, which is acceptable for on-board space computers.

6.2 Performance of EKF on PolyFSA Estimated Using the Analytical Model

In this section, performance of EKF on the proposed architecture is compared with other non-polymorphic implementations that are based on systolic arrays. Figure 6.3 illustrates two possible non-polymorphic hardware architectures.

- NonPolyArch_1: In this architecture, all the Processing Elements (PEs) are dedicated

Table 6.2: Static power consumption of individual modules estimated using XPower.

Design module	Power (mW)
Static Design (Microblaze+Cache+Controller)	954.8
FSA PE	38.1

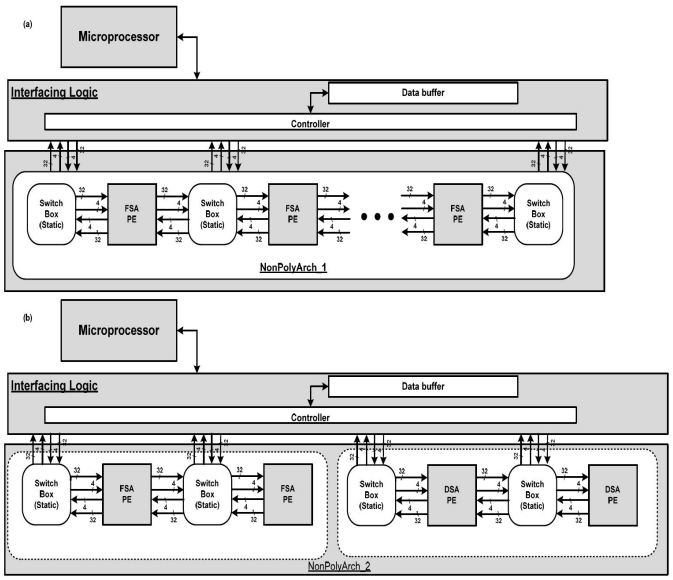


Fig. 6.3: (a) Top-level system architecture with proposed PolyFSA (shown in fig. 4.10) replaced by NonPolyArch_1; (b) Top-level system architecture with proposed PolyFSA replaced by NonPolyArch_2.

to perform the operations to accelerate Faddeev algorithm. All switch boxes are statically controlled. To accelerate a different application, the entire FPGA is reconfigured via the JTAG interface. In Chapter 4, polymorphic design of FSA PE is discussed. A single PE can be reconfigured using partial dynamic reconfiguration, thereby reducing T_{config}. However, the resource overhead associated with the proposed design reduces the number of PEs that can be fit into a particular FPGA, thereby reducing $T_{co-processor}$.

- NonPolyArch_2: In this architecture, the available FPGA resources are split equally between the two (or more) types of PEs that are required to accelerate EKF (FSA PE) and other applications. In Figure 6.3, two types of PEs are shown: (i) Faddeev Systolic Array PE, and (ii) Discrete Wavelet Transform Systolic Array (DSA) PE. Design using this architecture results in zero reconfiguration time. Trade-off is that the number of FSA PEs that can be fit into a particular FPGA is the lowest among the three architectures (thus, $T_{co-processor}$ is the highest). As the number of algorithms

Table 6.3: Resource set for FPGAs from Xilinx Virtex 4 and Virtex 5 families.

FPGA type	LUT	FF	XtremeDSP/DSP48E	BRAM (18Kb each)	Bitstream file size (bits)
Virtex 4 SX35	30720	30720	192	192	13705216
Virtex 4 SX55	49152	49152	512	320	22749184
Virtex 4 FX140	126336	126336	192	552	47857664
Virtex 4 LX200	178176	178176	96	336	51372032
Virtex 5 LX330T	207360	207360	192	648	82698240
Virtex 5 SX95T	58880	58880	640	488	35717120

that need to be accelerated increases, number of resources dedicated for accelerating each application reduces proportionally.

In sec. 5.2, a performance model to predict the execution time of EKF on the proposed system architecture is described in detail. It is noted that this system architecture can be realized using three types of co-processor architectures: (i) proposed PolyFSA architecture, (ii) NonPolyArch_1, and (iii) NonPolyArch_2. This section uses the proposed performance model to predict the execution time of EKF on proposed PolyFSA architecture implemented on a Xilinx FPGA and compare it against the execution times of EKF on NonPolyArch_1 and NonPolyArch_2 that are implemented on a same FPGA device. Six Xilinx FPGAs with varying sets of resources from Virtex 4 and Virtex 5 families are selected as target platforms. Table 6.3 shows the list of FPGAs, total resources available on the chip, and the size of bitstreams used to fully configure the FPGA. It is observed that some of the FPGAs are rich in LUT and FF count, but lack in XtremeDSP/DSP48E resources (e.g., Virtex 4 LX200). In contrast, some FPGAs are abundant in XtremeDSP/DSP48E count resources, but lack in FF and LUT count (e.g., Virtex 4 SX55).

Resource utilization for each PE and the static design is shown in Table 6.1. For all the three architectures, maximum number of PEs can be estimated for each FPGA and Table 6.4 lists these numbers. For obtaining these numbers, a set of approximations are considered while computing the total resource utilization and this set is listed below.

- Resource utilizations for PEs and static region are obtained by using Virtex 4 SX35 as the target FPGA platform. These numbers may be slightly different when the

Table 6.4: Maximum number of PEs that can be mapped onto different FPGAs for the three architectures.

FPGA type	Proposed PolyFSA	NonPolyArch_1	NonPolyArch_2
Virtex 4 SX35	5	20	10
Virtex 4 SX55	11	36	18
Virtex 4 FX140	20	23	11
Virtex 4 LX200	11	11	5
Virtex 5 LX330T	14	23	11
Virtex 5 SX95T	25	44	22

architectures are targeted for other FPGAs.

- Virtex 5 FPGAs have a different type of lookup tables (6-input LUTs) when compared to Virtex 4 FPGAs (4-input LUTs). So, the resource utilization for Virtex5 FPGAs may be different.

In this research, it is argued that the approximations for computing the total resource utilization are maintained across all three architecture evaluations, and hence the comparisons between the three architectures should be valid. Equation 5.11 is used to compute the overall execution time of EKF on the system architecture. Irrespective of the co-processor architecture, this equation remains valid. In this equation, it is observed that T_{config} occupies a significant portion of the overall execution time, if the architecture is reconfigured frequently (for lesser values of $nIter$). Figure 6.4 shows the comparison between the reconfiguration times for PolyFSA and NonPolyArch_1 for all the six FPGAs. In this analysis, NonPolyArch_2 is not considered because there is no reconfiguration involved ($T_{config} = 0$). It is observed that T_{config} is much smaller for the proposed PolyFSA architecture.

Figure 6.5 shows the overall execution time for EKF running on the three different architectures. This time does not include the microprocessor execution time. Execution time for non-accelerated functions on the microprocessor depends on two factors: (i) problem size (Number of States (NS) and Number of Measurements (NM)), and (ii) application characteristics that determine the sequence of computations found inside the non-accelerated functions. In the prior section, application characteristics for a particular problem size are presented and results are provided. Understanding the application characteristics for

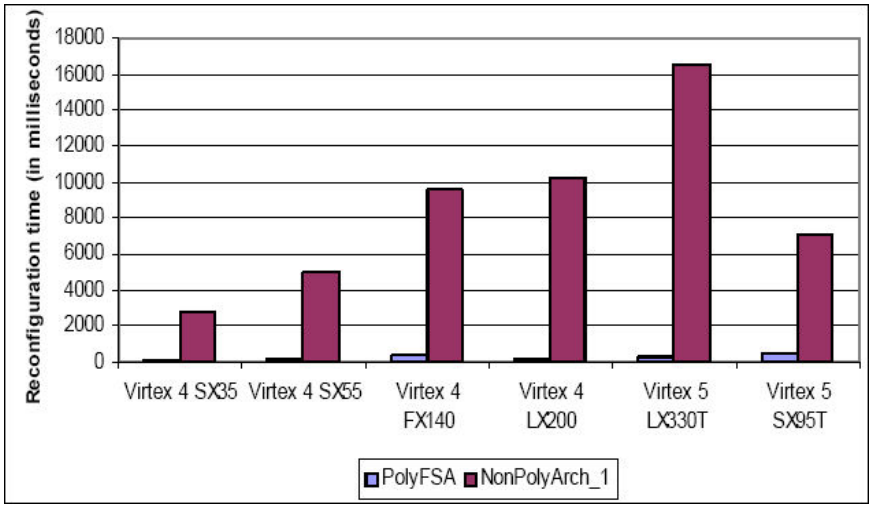
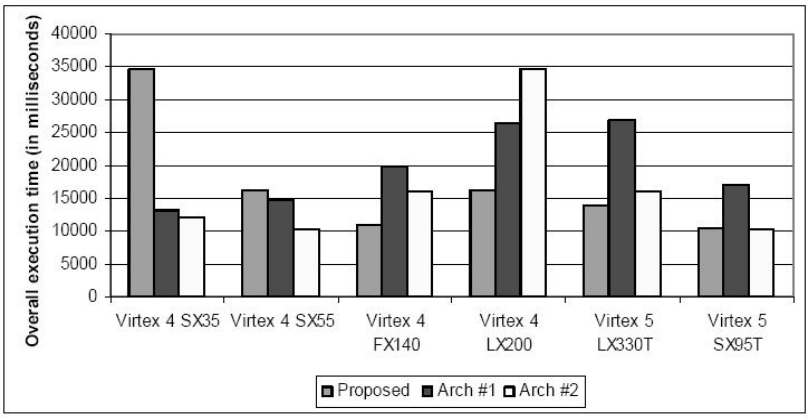


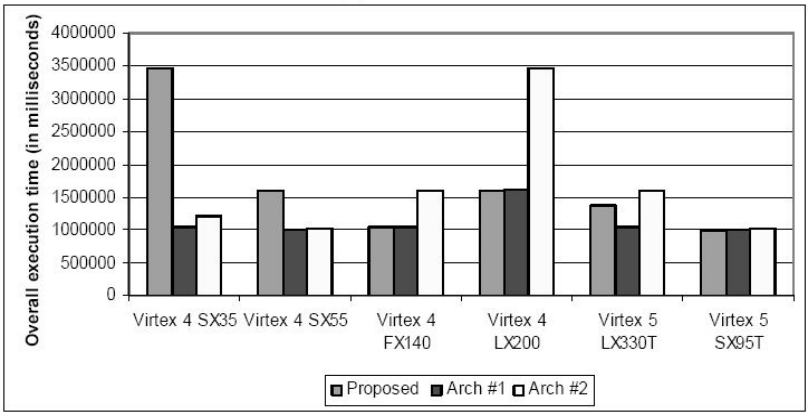
Fig. 6.4: Comparison of reconfiguration times between the proposed PolyFSA architecture and NonPolyArch_1.

multiple problem sizes and determining the microprocessor execution times is beyond the scope of this paper. It is however observed that the microprocessor execution time is the same for all three architectures and can be eliminated from the overall execution time while comparing the three architectures. In fig. 6.5, it is observed that the proposed PolyFSA architecture is outperformed by the other two architectures when the target FPGA platform is either Virtex 4 SX35 or Virtex 4 SX55. Reason behind this degradation in performance is that both the above mentioned FPGAs have limited real-estate (2-D area on the chip) and this limits the number of PEs that can be placed-and-routed (for the proposed PolyFSA architecture) on these FPGAs. For the non-polymorphic architectures, there is more freedom to place-and-route the PEs, thus resulting in a larger number of PEs being mapped onto the FPGA. For the other four FPGAs, it is observed that the proposed PolyFSA architecture outperforms both the non-polymorphic architectures for smaller number of iterations. However, NonPolyArch_1 outperforms the proposed architecture for some of the FPGAs, as the number of iterations is increased. As number of iterations is increased, the impact of reconfiguration time on the overall execution time recedes and co-processor execution time starts to dominate. It was noted earlier that more number of PEs can be mapped onto the target FPGA for NonPolyArch_1, so the co-processor execution time reduces, thereby

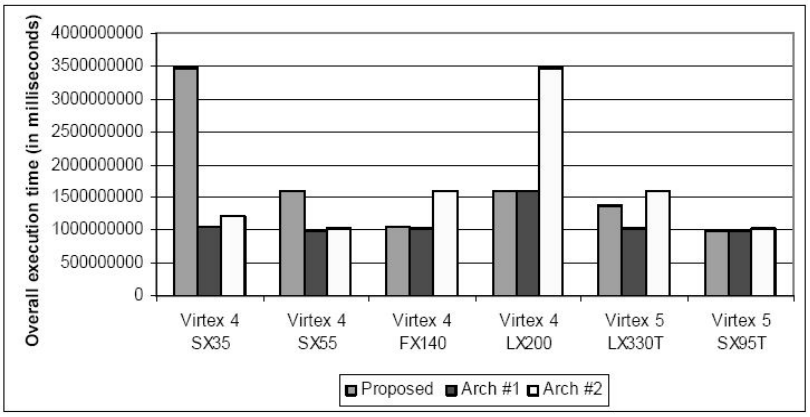
reducing the overall execution time.



(a) *nIter* = 10000



(b) *nIter* = 1000000



(c) *nIter* = 1000000000

Fig. 6.5: Predicted execution times for EKF on the proposed PolyFSA architecture and two non-polymorphic architectures. Results are presented for six different FPGAs and for three different number of iterations.

6.3 3-D Pareto Curve

This section discusses the 3-D pareto plots (area versus error versus execution time) that are obtained as the end product of the proposed architecture analysis. Figure 6.6 represents area in terms of FF usage and fig. 6.7 represents area in terms of LUT usage. In these figures, area corresponds to the area required to realize a single PE of the proposed PolyFSA. Error corresponds to the mean error observed in the computation of the state variable. Execution time corresponds to the overall execution time of EKF. EKF operates on the system that is illustrated in [78] (NS=9, NM=10, NC=6). It is observed that a large fraction of the data points correspond to an area savings of 50% or higher. Upon individual analysis of the two plots, it is observed that a large savings in FF usage can be obtained by allowing the execution of EKF to be slower and a large savings in LUT usage can be obtained by allowing a higher threshold for the error. It should be noted that the proposed architecture analysis is capable of generating a plot with a much higher resolution and a much bigger range than the plots that are presented here.

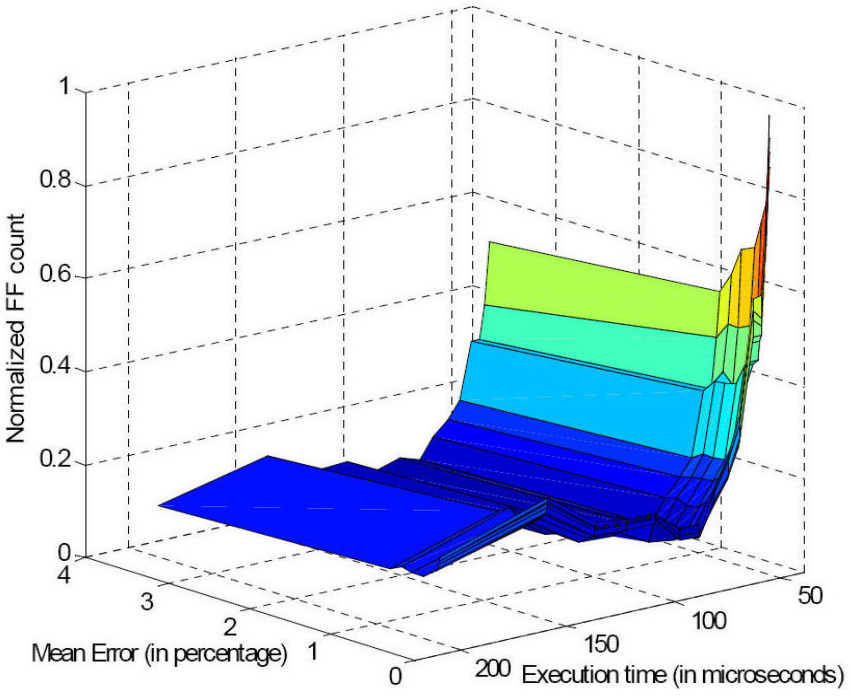


Fig. 6.6: 3-D pareto curve (area versus error versus execution time). In this plot, area is represented in terms of Flip-Flop usage.

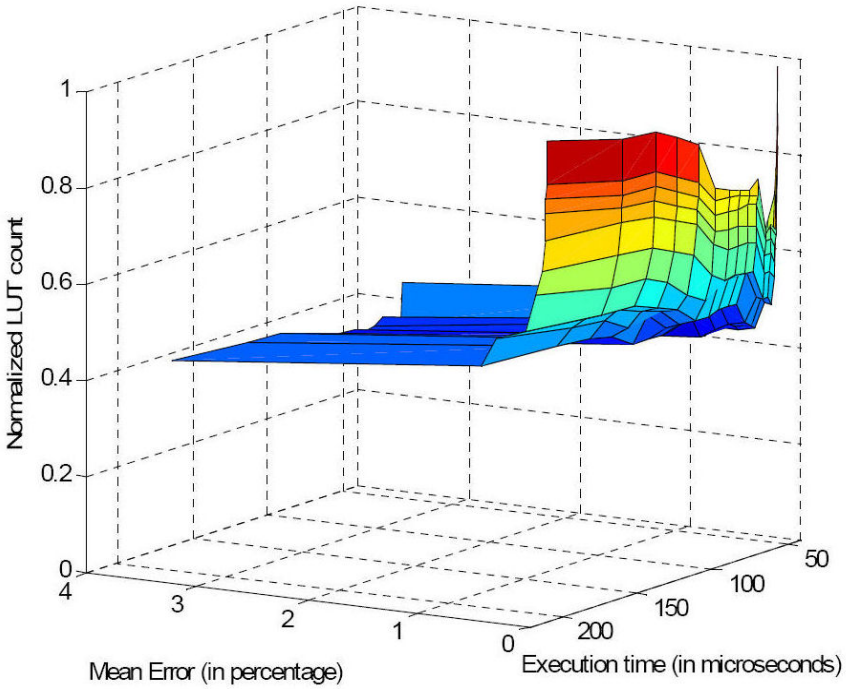


Fig. 6.7: 3-D pareto curve (area versus error versus execution time). In this plot, area is represented in terms of LUT usage.

Chapter 7

Conclusions and Future Work

Kalman filter is a compute-intensive algorithm comprising of multiple linear algebra operations. Based on the literature in the domain of designing accelerators for linear algebra operations and theoretical analysis, it is concluded that systolic arrays provide an effective framework to accelerate linear algebra operations. Transforming all the linear algebra operations into their equivalent Faddeev operations enabled us to design a single systolic array framework to accelerate the entire Kalman filter algorithm. Xilinx FPGAs are used as the target platforms.

In this research, we proposed a Polymorphic Faddeev Systolic Array (PolyFSA) that is used as a co-processor to accelerate all the linear algebra kernels. Proposed design is implemented on a Xilinx Virtex 4 SX35 ML-402 board using Partial Dynamic Reconfiguration (PDR)-based design techniques and test cases are executed at a clock frequency of 100 MHz. Timing results indicate that the proposed design outperforms one of the software processors (PowerPC 750) that is used in many space missions.

During any space mission, the design goals for the Kalman filter architecture, in terms of time, area, and error, may be different during different intervals. To support different design goals, proposed PolyFSA is designed such that the number of PEs in the systolic array that are used to accelerate Kalman filters can be varied during run-time. This variation is achieved by constraining each PE to a single Partial Reconfigurable Region (PRR or socket), and using PDR to provide the option to modify the functionality of a particular PRR during run-time. Results are provided to showcase the acceleration that is achieved for different number of PEs.

In addition to varying number of PEs dedicated to accelerating Kalman filters, it is identified that each PE of the PolyFSA can be redesigned so that it is possible to fit a

variable number of PEs into each PRR. This is achieved by varying the design parameters associated with the expensive (in terms of area) floating-point logic. Such parameters include latency, precision, and input data rate. In addition to resulting in multiple area requirements, variations in these parameters affect the error and execution time as well. A comprehensive architectural analysis is proposed and the results are presented in terms of 2-D Pareto plots (error versus area, performance (or time) versus area) and a 3-D plot (error versus area versus performance).

To analyze the PolyFSA-based system architecture, we proposed the following: (i) an application-specific error analysis to determine the effect of variations in data precision on the error introduced in the estimation of the state variable, (ii) a modified As-Soon-As-Possible (ASAP) scheduling algorithm to simulate the execution of Kalman filter on the system architecture and determine the overall execution time, and (iii) a simplified area model used to quickly estimate the overall area.

Results of error analysis showed that reduction in precision of the adder unit results in the largest error, and reduction in precision of divider unit results in the least error. Also, it is inferred that variations in number of time steps does not have a significant impact on the error. The 2-D pareto curve indicated that a 50% reduction in area can be obtained by allowing mean error and variance error to be limited to 1%.

From the performance model of EKF presented in this dissertation, it is observed that the execution time of Faddeev algorithm ($T_{Faddeev}$) has a significant impact on the overall execution time of Kalman filter. Two of the major architectural features that have the greatest impact on the performance are: (i) number of PEs in PolyFSA (R), and (ii) input rate of the divider unit. We also observed that variations in latency of the adder and multiplier units also have some impact on the overall performance. Variation of latency of divider unit does not have any impact on $T_{Faddeev}$. A 2-D pareto curve (area versus time) indicates that increase in overall execution time is not always accompanied by a reduction in area. While the number of LUTs remained almost the same for different execution times, number of FFs decreased significantly for increase in execution time.

Results of area analysis indicated the following.

- Reduction in latency of a floating-point unit results in a large reduction in the number of FFs and reduction in precision results in a large reduction in the number of LUTs.
- There is a large reduction in the number of LUTs and FFs for increase in c_rate. Area reduces by a factor of 8, when c_rate increases from 1 to 7. Also, increase in c_rate beyond a value of 7 does not cause any variation in the area.

The 3-D pareto curves (area versus performance versus error) are presented in Chapter 6. It is observed that a savings of 50% in both LUT and FF usage can be obtained for a large number of design options that result in a tolerable performance and error. These plots, in conjunction with the proposed PolyFSA architecture, enables the design engineer (not a VLSI architect) to select the optimal design option that also satisfy his requirements.

Remainder of this section outlines some of the possible future directions for the proposed research.

- Proposed PolyFSA architecture and architecture analysis is comprised of an application specific design methodology. In this design methodology, the hardware designer generates a polymorphic architecture template and derives a set of pareto curves based on the design goals. Application developer can use the template and the pareto curves to select a design option that are best-suited to meet his objectives. This methodology can be extended to support the design of hardware accelerators for any other application.
- Proposed architecture analysis supports three design goals: area, time, and error. Power consumption is an important design objective that needs to be added to this analysis, specifically for space-based applications. This involves development of power estimation models for variations in design parameters. Relationship between power, area, time, and error also need to be understood. A 4-D pareto curve will be the result of the modified architecture analysis.

- Proposed architecture analysis supports variations in the following design parameters for all the three floating-point units: precision, latency, and clock rate (only for divider). Other design parameters that may be varied are: (i) implementation type - whether the design uses embedded units (like DSP48), and (ii) FPGA family type. Variation in FPGA family type may be interesting in the case of power analysis.

References

[1] J. E. Riedel, S. Desai, D. Han, B. Kennedy, G. W. Null, S. P. Synnott, T. C. Wang, R. A. Werner, and E. B. Zamani, "Autonomous optical navigation (AutoNav) DS1 technology validation report," Jet Propulsion Laboratory, Technical Report, 2001.

[2] W. S. Chaer and J. Ghosh, "Hierarchical adaptive Kalman filtering for interplanetary orbit determination," *IEEE Transactions on Aerospace and Electronic Systems*, pp. 375–386, 1998.

[3] T. Misu and K. Ninomiya, "Optical guidance for autonomous landing of spacecraft," *IEEE Transactions on Aerospace and Electronic Systems*, pp. 459–473, 1999.

[4] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME Journal of Basic Engineering*, pp. 35–45, 1960.

[5] H. T. Kung and C. E. Leiserson, "Systolic arrays for VLSI," *Proceedings of the Sparse Matrix for the Society of Industrial and Applied Mathematics*, pp. 256–282, 1978.

[6] H.-G. Yeh, "Kalman filtering and systolic processors," *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 11, pp. 2139–2142, Apr. 1986.

[7] M. Lu, X. Qiao, and G. Chen, "A parallel square-root algorithm for modified extended Kalman filter," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 28, no. 1, pp. 153–163, Jan. 1992.

[8] P. Rao and M. Bayoumi, "An efficient VLSI implementation of real-time Kalman filter," *Proceedings of the IEEE International Symposium on Circuits and Systems*, vol. 3, pp. 2353–2356, May 1990.

[9] F. Busse and J. P. How, "Demonstration of adaptive Extended Kalman Filter for low earth orbit formation estimation using CDGPS," *Journal of The Institute of Navigation*, pp. 79–94, 2002.

[10] V. Bonato, R. Peron, D. F. Wolf, J. A. M. de Holanda, E. Marques, and J. M. P. Cardoso, "An FPGA implementation for a Kalman filter with application to mobile robotics," *Proceedings of the International Symposium on Industrial Embedded Systems*, pp. 148–155, July 2007.

[11] D. Lau, O. Pritchard, and P. Molson, "Automated generation of hardware accelerators with direct memory access from ANSI/ISO standard C functions," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 45–56, Apr. 2006.

[12] F. Wichmann, "An experimental parallelizing systolic compiler for regular programs," *Proceedings of the Programming Models for Massively Parallel Computers*, pp. 92–99, Sept. 1993.

[13] M. S. Lam, *A Systolic Array Optimizing Compiler*. Norwell, MA: Kluwer Academic Publishers, 1989.

[14] A. El-Amawy, "A systolic architecture for fast dense matrix inversion," *IEEE Transactions on Computer*, vol. 38, no. 3, pp. 449–455, Mar. 1989.

[15] K. Lau, M. Kumar, and R. Venkatesh, "Parallel matrix inversion techniques," *Proceedings of the IEEE Second International Conference on Algorithms and Architectures for Parallel Processing*, pp. 515–521, June 1996.

[16] F. Edman and V. Owall, "Implementation of a scalable matrix inversion architecture for triangular matrices," *Proceedings of the IEEE Conference on Personal, Indoor and Mobile Radio Communications*, vol. 3, pp. 2558–2562, Sept. 2003.

[17] J.-W. Jang, S. B. Choi, and V. K. Prasanna, "Energy- and time-efficient matrix multiplication on FPGAs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 11, pp. 1305–1319, 2005.

[18] V. Daga, G. Govindu, V. Prasanna, S. Gangadharapalli, and V. Sridhar, "Efficient floating-point based block LU decomposition on FPGAs," *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pp. 21–24, July 2004.

[19] X. Wang and S. Ziavras, "A configurable multiprocessor and dynamic load balancing for parallel LU factorization," *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, p. 234, Apr. 2004.

[20] G. Govindu, S. Choi, V. Prasanna, V. Daga, S. Gangadharpalli, and V. Sridhar, "A high-performance and energy-efficient architecture for floating-point based LU decomposition on FPGAs," *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, p. 149, Apr. 2004.

[21] X. Wang and S. Ziavras, "Performance optimization of an FPGA-based configurable multiprocessor for matrix operations," *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)*, pp. 303–306, Dec. 2003.

[22] L. Zhuo and V. Prasanna, "Scalable hybrid designs for linear algebra on reconfigurable computing systems," *Proceedings of the 12th International Conference on Parallel and Distributed Systems*, vol. 1, p. 9, July 2006.

[23] P. L. Richman, "Automatic error analysis for determining precision," *ACM Transactions on Communications*, vol. 15, no. 9, pp. 813–817, 1972.

[24] A. Peleg and U. Weiser, "MMX technology extension to the Intel architecture," *IEEE Micro*, vol. 16, no. 4, pp. 42–50, Aug. 1996.

[25] R. B. Lee, "Subword parallelism with MAX-2," *IEEE Micro*, vol. 16, no. 4, pp. 51–59, Aug. 1996.

[26] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu, and G. Zyner, "The visual instruction set (VIS) in UltraSPARC," *Digest of Papers of the Compcon Conference on Technologies for the Information Superhighway*, pp. 462–469, Mar. 1995.

[27] M. L. Chang and S. Hauck, "Variable precision analysis for FPGA synthesis," in *Proceedings of the Nasa Earth Science Technology Conference*, 2003.

[28] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee, "Precision and error analysis of MATLAB applications during automated hardware synthesis for FPGAs," *Proceedings of the conference on Design, Automation and Test in Europe*, pp. 722–728, Mar. 2001.

[29] D.-U. Lee, A. Gaffar, O. Mencer, and W. Luk, "Minibit: bit-width optimization via affine arithmetic," *Proceedings of the Design Automation Conference*, pp. 837–840, June 2005.

[30] P. Fiore, "Efficient approximate wordlength optimization," *IEEE Transactions on Computers*, vol. 57, no. 11, pp. 1561–1570, Nov. 2008.

[31] D. Menard and O. Sentieys, "Automatic evaluation of the accuracy of fixed-point algorithms," *Proceedings of the conference on Design, Automation and Test in Europe*, pp. 529–535, Mar. 2002.

[32] D.-U. Lee and J. Villasenor, "A bit-width optimization methodology for polynomial-based function evaluation," *IEEE Transactions on Computers*, vol. 56, no. 4, pp. 567–571, Apr. 2007.

[33] M. Chang and S. Hauck, "Automated least-significant bit datapath optimization for FPGAs," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 59–67, Apr. 2004.

[34] A. de la Serna and M. Soderstrand, "Trade-off between FPGA resource utilization and roundoff error in optimized CSD FIR digital filters," *Conference Record of the Twenty-Eighth Asilomar Conference on Signals, Systems and Computers*, vol. 1, pp. 187–191, Oct. 1994.

[35] G. A. Constantinides, P. Y. K. Cheung, and W. Luk, "Optimum and heuristic synthesis of multiple word-length architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 1, pp. 39–57, 2005.

[36] G. Constantinides, P. Cheung, and W. Luk, "The multiple wordlength paradigm," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 51–60, Apr. 2001.

[37] G. Constantinides, P. Cheung, and W. Luk, "Optimum wordlength allocation," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 219–228, Apr. 2002.

[38] G. A. Constantinides and G. J. Woeginger, "The complexity of multiple wordlength assignment," *Applied Mathematics Letters*, vol. 15, no. 2, pp. 137–140, Feb. 2002.

[39] G. Constantinides, P. Cheung, and W. Luk, "Multiple precision for resource minimization," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 307–308, Apr. 2000.

[40] D.-U. Lee, A. Gaffar, R. Cheung, O. Mencer, W. Luk, and G. Constantinides, "Accuracy-guaranteed bit-width optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1990–2000, Oct. 2006.

[41] G. Caffarena, G. Constantinides, P. Cheung, C. Carreras, and O. Nieto-Taladriz, "Optimal combined word-length allocation and architectural synthesis of digital signal processing circuits," *IEEE Transactions on Circuits and Systems*, vol. 53, no. 5, pp. 339–343, May 2006.

[42] S. Chan and K. Tsui, "Wordlength optimization of linear time-invariant systems with multiple outputs using geometric programming," *IEEE Transactions on Circuits and Systems*, vol. 54, no. 4, pp. 845–854, Apr. 2007.

[43] G. Constantinides, "Perturbation analysis for word-length optimization," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 81–90, Apr. 2003.

[44] A. Smith, G. A. Constantinides, and P. Y. K. Cheung, "An automated flow for arithmetic component generation in Field Programmable Gate Arrays," *ACM Transactions on Reconfigurable Technology and Systems*, pp. 1–20, 2009.

[45] A. Roldao-Lopes, A. Shahzad, G. A. Constantinides, and E. C. Kerrigan, "More flops or more precision? accuracy parameterizable linear equation solvers for model predictive control," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 209–216, Oct. 2009.

[46] A. Gaffar, O. Mencer, W. Luk, P. Cheung, and N. Shirazi, "Floating-point bitwidth analysis via automatic differentiation," *Proceedings of the IEEE International Conference on Field-Programmable Technology*, pp. 158–165, Dec. 2002.

[47] R. Strzodka and D. Goddeke, "Pipelined mixed precision algorithms on FPGAs for fast and accurate PDE solvers from low precision components," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 259–270, Apr. 2006.

[48] M. Chang and S. Hauck, "Precis: a design-time precision analysis tool," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 229–238, Apr. 2002.

[49] Y. Lee, Y. Choi, S.-B. Ko, and M. H. Lee, "Performance analysis of bit-width reduced floating-point arithmetic units in FPGAs: a case study of neural network-based face detector," *EURASIP Journal on Embedded Systems*, vol. 2009, pp. 1–11, 2009.

[50] J. Sun, G. D. Peterson, and O. O. Storaasli, "High-performance mixed-precision linear solver for FPGAs," *IEEE Transactions on Computers*, vol. 57, no. 12, pp. 1614–1623, 2008.

[51] M. K. Jaiswal and N. Chandrachoodan, "Efficient implementation of floating-point reciprocator on FPGA," *Proceedings of the 22nd International Conference on VLSI Design*, pp. 267–271, Jan. 2009.

[52] M. Jaiswal and N. Chandrachoodan, "Efficient implementation of IEEE double precision floating-point multiplier on FPGA," *Proceedings of the Third international Conference on Industrial and Information Systems*, pp. 1–4, Dec. 2008.

[53] C. He, G. Qin, R. E. Ewing, and W. Zhao, "High-precision BLAS on FPGA-enhanced computers," *Proceedings of the International conference on Engineering of Reconfigurable Systems and Algorithms*, pp. 107–116, 2007.

[54] S. M. Ali Irturk and R. Kastner, "An efficient FPGA implementation of scalable matrix inversion core using QR decomposition," University of California at San Diego, Technical Report, Mar. 2009.

[55] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky, "A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 39–48, Apr. 2000.

[56] L. H. de Figueiredo and J. Stolfi, "Affine arithmetic: Concepts and applications: Scan2002 international conference (guest editors: Rene alt and jean-luc lamotte)," *Numerical Algorithms*, vol. 37, no. 1-4, pp. 147+ [Online]. Available: `http://dx.doi.org/10.1023/B:NUMA.0000049462.70970.b6`.

[57] J.-M. Muller, "On the definition of ulp (x)," Institut National De Recherche En Informatique Et En Automatique, Technical Report, Feb. 2005.

[58] C. T. Ewe, P. Y. K. Cheung, and G. A. Constantinides, "Dual fixed-point: An efficient alternative to floating-point computation," *Proceedings of the International Conference on Field Programmable Logic*, pp. 200–208, Aug. 2004.

[59] L. Bossuet, G. Gogniat, J.-P. Diguet, and J.-L. Philippe, "A modeling method for reconfigurable architectures," *Proceedings of the IEEE International Workshop on System On a Chip*, pp. 170–180, June 2002.

[60] S. Bilavarn, G. Gogniat, J. Philippe, and L. Bossuet, "Fast prototyping of reconfigurable architectures from a C program," *Proceedings of the IEEE International Symposium on Circuits and Systems*, vol. 5, pp. 589–592, May 2003.

[61] S. Bilavarn, G. Gogniat, and J. Philippe, "Area time power estimation for FPGA based designs at a behavioral level," *Proceedings of the IEEE International Conference on Electronics, Circuits and Systems*, vol. 1, pp. 524–527, Dec. 2000.

[62] L. Bossuet, G. Gogniat, and J. L. Philippe, "Communication costs driven design space exploration for reconfigurable architectures," *Proceedings of the International Conference on Field Programmable Logic*, pp. 921–933, Sept. 2003.

[63] L. Bossuet, G. Gogniat, and J.-L. Philippe, "Generic design space exploration for reconfigurable architectures," *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, p. 163, Apr. 2005.

[64] S. Bilavarn, G. Gogniat, J.-L. Philippe, and L. Bossuet, "Design space pruning through early estimations of area/delay tradeoffs for FPGA implementations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1950–1968, Oct. 2006.

[65] C. A. Moritz, D. Yeung, and A. Agarwal, "SimpleFit: A framework for analyzing design trade-offs in raw architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 7, pp. 730–742, 2001.

[66] E. Waingold, M. Taylor, V. Sarkar, V. Lee, W. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktumi, R. Barua, J. Babb, S. Amarsinghe, and A. Agarwal, "Baring it all to software: The raw machine," Cambridge, MA, Technical Report, 1997.

[67] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee, "Accurate area and delay estimators for FPGAs," *Proceedings of the conference on Design, Automation and Test in Europe*, p. 862, Mar. 2002.

[68] J. Park, P. C. Member-Diniz, and K. R. Shesha Shayee, "Performance and area modeling of complete FPGA designs in the presence of loop transformations," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1420–1435, 2004.

[69] S. Memik, N. Bellas, and S. Mondal, "Presynthesis area estimation of reconfigurable streaming accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 11, pp. 2027–2038, Nov. 2008.

[70] Q. Liu, G. Constantinides, K. Masselos, and P. Cheung, "Combining data reuse with data-level parallelization for FPGA-targeted hardware compilation: A geometric programming framework," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 3, pp. 305–315, Mar. 2009.

[71] A. Smith, G. Constantinides, and P. Cheung, "Integrated floorplanning, module-selection, and architecture generation for reconfigurable devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 6, pp. 733–744, June 2008.

[72] J. Rice and K. Kent, "Case studies in determining the optimal field programmable gate array design for computing highly parallelisable problems," *IET Journal on Computers and Digital Techniques*, vol. 3, no. 3, pp. 247–258, May 2009.

[73] M. Rashid, F. Ferrandi, and K. Bertels, "hArtes design flow for heterogeneous platforms," *Proceedings of the 10th International Symposium on Quality of Electronic Design*, pp. 330–338, Mar. 2009.

[74] T. Givargis and F. Vahid, "Platune: a tuning framework for system-on-a-chip platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 11, pp. 1317–1327, Nov. 2002.

[75] L. Idkhajine, E. Monmasson, and A. Maalouf, "FPGA-based sensorless controller for synchronous machine using an Extended Kalman Filter," *Proceedings of the 13th European Conference on Power Electronics and Applications*, pp. 1–10, Sept. 2009.

[76] V. Bonato, E. Marques, and G. Constantinides, "A floating-point Extended Kalman Filter implementation for autonomous mobile robots," *Proceedings of the International Conference on Field Programmable Logic and Applications*, pp. 576–579, Aug. 2007.

[77] R. Baheti, D. O'Hallaron, and H. Itzkowitz, "Mapping Extended Kalman Filters onto linear arrays," *IEEE Transactions on Automatic Control*, vol. 35, no. 12, pp. 1310–1319, Dec. 1990.

[78] S. Ronnback, "Development of an INS/GPS navigation loop for UAV," Master's thesis, Lulea University of Technology, 2000.

[79] R. Barnes, "Dynamically reconfigurable systolic array accelerators: A case study with EKF and DWT algorithms," Master's thesis, Utah State University, 2008.

[80] *MicroBlaze Processor Reference Guide: Embedded Development Kit EDK 9.1i*, 7th ed., Xilinx Incorporated, 2007.

[81] *Simics/PM-PPC Target Guide*, Virtutech AB, 2006.

Vita

Arvind Sudarsanam

Education

- Doctor of Philosophy in Electrical and Computer Engineering, Utah State University, Logan, Utah, 2010
- Master of Science in Electrical Engineering, Arizona State University, Tempe, Arizona, 2004
- Bachelor of Engineering in Electronics and Instrumentation Engineering, Birla Institute of Technology and Science, Pilani, India, 2001

Published Journal Articles

- A. Dasu, A. Sudarsanam, and S. Panchanathan, "Design of Embedded Compute Intensive Processing Elements and their Scheduling in a Reconfigurable Environment", in the Canadian Journal of Electrical and Computer Engineering (CJECE), 2005
- T. Hauser, A. Dasu, A. Sudarsanam, and S. Young, "Performance of LU decomposition on a Multi-FPGA System Compared to a Low Power Commodity Microprocessor System", in the journal for Scalable computing: Practice and Experience, 2007, Vol 8, No 4
- J. Phillips, A. Sudarsanam, R. Kallam, J. Carver, and A. Dasu, "Methodology to Derive Polymorphic Soft-IP Cores for FPGAs", in the journal of IET Computers and Digital Techniques, 2008

- A. Sudarsanam, T. Hauser, A. Dasu, and S. Young, "A Power Efficient Linear Equation Solver on a Multi-FPGA Accelerator", in the International Journal of Computers and Applications, 2009

- A. Sudarsanam, R. Barnes, R. Kallam, J. Carver, and A. Dasu, "Dynamically Reconfigurable Systolic Array Accelerators: A Case Study with EKF and DWT Algorithms", in the journal of IET Computers and Digital Techniques, 2009

- A. Sudarsanam, A. Dasu, and K. Vaithianathan, "Analysis and Design of a Context Adaptable SAD/MSE Accelerator", in the International Journal of Reconfigurable Computing, 2009

- A. Sudarsanam, R. Kallam, and A. Dasu, "PRR-PRR Dynamic Relocation", in the IEEE Computer Architecture Letters, vol. 99, no. 2, Oct. 2009

Published Conference Papers

- A. Akoglu, A. Dasu, A. Sudarsanam, M. Srinivasan, and S. Panchanathan, "Pattern recognition tool to detect reconfigurable patterns in MPEG4 video processing", in the IEEE Proceedings of Parallel and Distributed Processing Symposium, 2002

- A. Sudarsanam, and S. Panchanathan, "Current Trends for Silicon and Embedded Computing Solutions for Automotive Applications", in the Proceedings of the convergence conference of SAE, Detroit, October, 2002

- A. Sudarsanam, A. Dasu, and S. Panchanathan, "Task Scheduling of Control Data Flow Graphs for Reconfigurable Architectures" in the Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, 2004

- A. Sudarsanam, M.Srinivasan, and S. Panchanathan, "Resource Estimation and Task Scheduling for Multithreaded Reconfigurable Architectures", in the Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS 2004)

- A. Sudarsanam, and S, Panchanathan, "Novel predicated data flow analysis based memory design for data and control intensive multimedia applications", in the Proceedings of the SPIE conference on Electronic Imaging, 2005

- A. Dasu, and A. Sudarsanam, "High Level - Application Analysis Techniques and Architectures - to Explore Design possibilities for Reduced Reconfiguration Area Overheads in FPGAs executing Compute Intensive Applications", in the Proceedings of the Reconfigurable Architectures Workshop (RAW), 2005

- A. Sudarsanam, and A. Dasu, "Implementation of Polymorphic Matrix Inversion using Viva", presented at the MAPLD conference, 2005

- A. Sudarsanam, and A. Dasu, "A Fast and Efficient FPGA-Based Implementation for Solving a System of Linear Interval Equations", in the IEEE Proceedings of the conference on Field Programmable Technology (FPT), 2005

- A. Sudarsanam, S. Young, T. Hauser, and A. Dasu, "Multi FPGA based High Performance LU Decomposition", in the Proceedings of the High Performance Embedded Computing workshop (HPEC), 2006

- S. Young, A. Sudarsanam, T. Hauser, and A. Dasu, "Memory Support Design for LU Decomposition on Starbridge Hypercomputer", in the IEEE Proceedings of the conference on Field Programmable Technology (FPT), 2006

- S. Chandrakar, A. Clements, A. Sudarsanam, and A. Dasu, "Memory Architecture Template for Fast Block Matching Algorithms on FPGAs", in the Proceedings of the Reconfigurable Architectures Workshop at the IEEE International Parallel and Distributed Processing Symposium, April 2010