5-1-2009

Dynamically Reconfigurable Systolic Array Accelerators: A Case Study with Extended Kalman Filter and Discrete Wavelet Transform Algorithms

Robert C. Barnes
Utah State University

DYNAMICALLY RECONFIGURABLE SYSTOLIC ARRAY ACCELERATORS: A CASE STUDY WITH EXTENDED KALMAN FILTER AND DISCRETE WAVELET TRANSFORM ALGORITHMS

by

Robert C. Barnes

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

of

MASTER OF SCIENCE

in

Computer Engineering

Approved:

Dr. Aravind R. Dasu Major Professor	Dr. Brandon Eames Committee Member
Dr. Paul Israelsen Committee Member	Dr. Byron R. Burnham Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2008

Abstract

Dynamically Reconfigurable Systolic Array Accelerators: A Case Study with Extended Kalman Filter and Discrete Wavelet Transform Algorithms

by

Robert C. Barnes, Master of Science

Utah State University, 2008

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

Field programmable grid arrays (FPGA) are increasingly being adopted as the primary on-board computing system for autonomous deep space vehicles. There is a need to support several complex applications for navigation and image processing in a rapidly responsive on-board FPGA-based computer. This requires exploring and combining several design concepts such as systolic arrays, hardware-software partitioning, and partial dynamic reconfiguration. A microprocessor/co-processor design that can accelerate two single precision floating-point algorithms, extended Kalman filter and a discrete wavelet transform, is presented. This research makes three key contributions. (i) A polymorphic systolic array framework comprising of reconfigurable partial region-based sockets to accelerate algorithms amenable to being mapped onto linear systolic arrays. When implemented on a low end Xilinx Virtex4 SX35 FPGA the design provides a speedup of at least 4.18x and 6.61x over a state of the art microprocessor used in spacecraft systems for the extended Kalman filter and discrete wavelet transform algorithms, respectively. (ii) Switchboxes to enable communication between static and partial reconfigurable regions and a simple protocol to enable schedule changes when a socket's contents are dynamically reconfigured to alter the concurrency of the participating systolic arrays. (iii) A hybrid partial dynamic

reconfiguration method that combines Xilinx early access partial reconfiguration, on-chip bitstream decompression, and bitstream relocation to enable fast scaling of systolic arrays on the PolySAF. This technique provided a 2.7x improvement in reconfiguration time compared to an off-chip partial reconfiguration technique that used a Flash card on the FPGA board, and a 44% improvement in BRAM usage compared to not using compression.

(64 pages)

For my son Logan.
I hope the completion of my degree will give me the means to provide you with more opportunities and experiences.

Acknowledgments

I would like to thank my advisor, Dr. Aravind Dasu, for investing in me and for his guidance throughout my research. I would also like to thank my committee members, Dr. Brandon Eames and Dr. Paul Israelsen.

I would like to thank my peers, Hari, Arvind, Jonathan, Varun, Madhu, and Shant, for their help. In particular, I acknowledge Ramachandra Kallam and Jeff Carver for their important contributions to my research in the area of bitstream compression and bitstream relocation, respectively.

Last, but not least, I would like to thank my beautiful wife for her patience and support.

Robert C. Barnes

Contents

Page

Abstract . . . . . **iii**

Acknowledgments . . . . . **vi**

List of Tables . . . . . **ix**

List of Figures . . . . . **x**

Acronyms . . . . . **xi**

1 Introduction **1**

2 Background and Related Work **5**
2.1 Spacecraft Navigation 5
2.1.1 The Kalman Filter 5
2.1.2 Extended Kalman Filter 5
2.1.3 KF and EKF Hardware Architectures 7
2.1.4 The Faddeev Algorithm 9
2.2 Discrete Wavelet Transform 10
2.3 Systolic Arrays 10
2.4 Reconfigurable Architectures 11
2.5 Partial Dynamic Reconfiguration 12
2.5.1 Partial Bitstream Compression and Decompression 12
2.5.2 Partial Bitstream Relocation 14

3 Proposed Design **15**
3.1 System Architecture 15
3.1.1 Microprocessor 15
3.1.2 Pseudo-Cache 15
3.1.3 Software Interface 16
3.1.4 PolySAF 18
3.1.5 Programmable Switchboxes 19
3.1.6 Scaling 19
3.2 Application of the EKF 20
3.2.1 Hardware-Software Partitioning 20
3.2.2 Faddeev Systolic Array 21
3.3 Application of the DWT 27
3.4 Hybrid PDR 27

4 Results and Analysis **30**
4.1 FloorPlan 30
4.2 Performance of the FSA and DSA on the PolySAF 31
4.3 Performance of the EKF and DWT (FPGA vs. PowerPC) 32
4.4 Analysis of PDR 33
4.5 Area Analysis 35

5 Conclusions and Future Work **37**
5.1 Conclusions 37
5.2 Future Work 38
5.2.1 PolySAF 38
5.2.2 FSA 39
5.2.3 DSA 39
5.2.4 Hybrid PDR 39

References **41**

Appendices **46**
Appendix A Code Snippets 47
A.1 Schur Complement 47
A.2 Boundary Cell 49
Appendix B Design Flow 51
B.1 EDK Flow 51
B.2 PlanAhead Flow 51
B.3 Flow Automation 52

List of Tables

Table		Page
2.1	Matrix operations mapped to the Faddeev algorithm	9
3.1	Co-processor instruction bitmap	17
3.2	List of co-processor instructions	17
3.3	Switchbox routing configurations	19
3.4	Boundary cell.	23
3.5	Internal cell.	24
4.1	Reconfiguration labels	35
4.2	PE resource utilization.	36
4.3	Static PE device utilization.	36

List of Figures

Figure Page

2.1 Kalman filter process cycle. 6

2.2 DWT algorithm 11

2.3 Configuration layout of the Virtex 4 FPGA. 13

3.1 System architecture with PolySAF 16

3.2 Software-to-hardware interface layers 18

3.3 Switchbox structure 20

3.4 PDR scaling example 21

3.5 Mapping of the EKF onto the Faddeev algorithm 22

3.6 2D Faddeev systolic array 23

3.7 Vertical projections of the 2D Faddeev SA 25

3.8 FSA internal architecture. 26

3.9 DSA internal structure. 28

3.10 Hybrid PDR partial bitstream flow 29

4.1 Design floorplan 31

4.2 FSA and DSA trade off 32

4.3 FPGA vs. PowerPC 34

4.4 Reconfiguration times 35

Acronyms

BM	Busmacro
BRAM	Block RAM
DMA	Direct Memory Access
DSA	DWT Systolic Array
DSP	Digital Signal Processing
DWT	Discrete Wavelet Transform
EAPR	Early Access Partial Reconfiguration
EDK	Embedded Development Kit
EKF	Extended Kalman Filter
ESA	European Space Agency
FPGA	Field Programmable Grid Array
FSA	Faddeev Systolic Array
HWICAP	Hardware Internal Configuration Access Port
ICAP	Internal Configuration Access Port
IDWT	Inverse Discrete Wavelet Transform
IP	Intellectual Property
ISE	Integrated Software Environment
KF	Kalman Filter
LUT	Look Up Table
NASA	National Aeronautics and Space Administration
OPB	Open Peripheral Bus
PB	Partial Bitstream
PDR	Partial Dynamic Reconfiguration
PolySAF	Polymorphic Systolic Array Framework
PR	Partial Reconfiguration
PRR	Partial Reconfiguration Region

RAM	Random Access Memory
SA	Systolic Array
SDK	Software Development Kit
SRAM	Static RAM
UAV	Unmanned Aerial Vehicle
VLSI	Very Large Scale Integration

Chapter 1

Introduction

Launch and operation of spacecraft systems is becoming an increasingly expensive and risky undertaking. Therefore space agencies like NASA (National Aeronautics and Space Administration) and the ESA (European Space Agency) are striving to maximize the science impact of future missions through emphasis on unprecedented levels of autonomy and on-board processing. This has a direct impact on the capabilities and responsiveness of on-board computing systems. While general-purpose microprocessors have been the workhorse of space-borne computers for many years, they unfortunately cannot continue to support the emerging demands for high performance embedded computing of future space missions [1]. Current space grade computer chips, like the BAE RAD750, are several generations behind current processors [1]. Over the past few years SRAM (static random access memory) based FPGAs (field programmable gate array) have made significant strides in device fabric features, such as support for partial dynamic reconfiguration, immersed IP (intellectual property) components (embedded digital signal processing (DSP) blocks and Block RAMs (BRAM)), have become viable options for the high demands of emerging aerospace applications. Pingree et al. list the benefits of using FPGAs for space-based applications as "higher level of reuse, reduced risk of obsolescence, simplified modification and update, and increased implementation options through modularization."

A sample of compute intensive applications that are useful for spacecraft include navigation, guidance, control, image and signal processing algorithms, among others. It will not be uncommon for future on-board computers to concurrently support and accelerate multiple compute intensive applications. In many situations these multiple classes of algorithms may have to work cooperatively to maximize the science benefits of a mission [2–4]. For example, a spacecraft entering another planet's atmosphere will likely encounter unknown

conditions. The extent of processing time allocated to an image-processing algorithm may then have to be traded off with the time allocated for a Kalman filter in a cooperative navigation system.

An interesting example that exhibits the need for dynamic responsiveness, with respect to computation capabilities needed between a navigation algorithm and an image processing algorithm, would be a close flyby of comets or asteroids. A flyby can unexpectedly affect the surrounding gravitation fields (thus putting a Kalman filter on overdrive) while the object is of high scientific interest visually (thus putting the image processor on overdrive). Such a system would have been useful in the deep impact mission. This mission consisted of two separate systems: the impactor and the flyby. The impactor navigated its way directly into the comet, while the flyby positioned itself near the path of the impactor to observe the outcome [5]. The flyby had separate subsystems for navigation and image processing. In a future mission these independent systems could be replaced by a single and more powerful system that dynamically scales between the two tasks as needed to optimize the trade-off. More navigation processing means the flyby may be able get closer to the collision, while more image processing means the flyby may be able to process more images.

Real time responsiveness on FPGAs translates to the ability to partially and dynamically reconfigure the device in real time. While this has been the holy grail of the FPGA community for several years and many approaches have been proposed, none of these approaches have been adapted into a practical system. Partial dynamic reconfiguration (PDR) is a method of reconfiguring only a portion of the FPGA while other portions remain active. This eliminates the requirement of an off-chip reconfiguration circuit, improves reconfiguration speeds, and allows uninterrupted processing on the FPGA. However, several options exist to improve reconfiguration speed, each with a unique cost (area and time). The costs are also dependent on the underlying macro and micro architectures, which in turn are dependent on the applications being accelerated on the FPGA.

In order to ground the exploration of PDR in reality, I have considered two algorithms of practical interest: the Extended Kalman Filter (EKF) and the Discrete Wavlet Trans-

form (DWT). To establish the underlying macro-architecture I have adopted the hardware-software partitioning approach that maps certain regions of each algorithm onto an embedded soft-core microprocessor, and the rest onto a scalable accelerator framework. This framework, also called the PolySAF (Polymorphic Systolic Array Framework), takes advantage of well-studied techniques of accelerating matrix-based operations on systolic arrays. It is within this framework that I have attempted to explore the strengths and weaknesses of several PDR techniques and studied their impact on performance (reconfiguration speed), area requirements, and constraints on floor planning.

The EKF algorithm was chosen because it forms the core of navigation systems and provides an accurate estimate of a system's state based on noisy inputs and models. The DWT algorithm was chosen because it is a widely used image compression algorithm based on a matrix based decomposition method. Both of these algorithms are compute intensive and can be accelerated by systolic array (SA) architectures. Systolic arrays are a group of repeated processing elements (PE) connected in a regular pattern. A polymorphic systolic array framework (PolySAF) was developed that allowed various systolic array accelerators to be dynamically configured within the FPGA by designing reprogrammable interconnections between PE and utilizing PDR to change the logic within each PE. It is shown in a case study how the DWT and EKF algorithms can be mapped to an on-board microprocessor and the PolySAF to enable dynamic scaling of the algorithms performance. This results in a Virtex-4 based system that outperforms the RAD750 for the EKF and DWT algorithms, but is still flexible and responsive.

By exploring and extending scalable systolic array accelerators for common compute intensive algorithms used in space and integrating methods for improving the performance of partial dynamic reconfiguration, a high performance rapidly adaptable space-based computer system can be realized that improves the flexibility and performance of future space missions. To the author's knowledge, these techniques have not yet been combined in a working design.

Next, Chapter 2 provides a more in-depth background on some of the important components of the design. Chapter 3 presents a detailed description of the system design. Chapter 4 presents the results and an analysis of the system. Finally, Chapter 5 presents conclusions and proposes future research directions.

Chapter 2

Background and Related Work

2.1 Spacecraft Navigation

2.1.1 The Kalman Filter

To navigate in space an autonomous spacecraft must accurately estimate its state from noisy measurements. The Kalman filter (KF), defined in algorithm 2.1, processes each of these measurements and returns the optimal estimate of the system state and error [6–8]. Since its discovery 47 years ago [9], the Kalman filter has become one of the most important filtering algorithms [10]. There are many variations on the Kalman filter and extensive mathematical background information would be required to explain it fully. What follows is a brief overview of the filter, with emphasis on aspects that influenced the design of the filter accelerator.

The Kalman filter is the recursive solution to estimating a state, and is optimal with respect to the least mean squared error [6]. It is composed of two phases: predict and update (fig. 2.1). During the predict phase the next state of the system is predicted based on the current estimate. This phase can be run repeatedly until new measurements are available, however the error of the estimate increases at each time step. When new measurements are available, the update phase is run. In this phase the state and error estimates are updated based on the new measurements, which decreases the error estimate. The filter is very flexible, for example it can be used to estimate a system's state from only a single sensor, estimate the bias in sensors, determine an unknown system model, or predict a future state.

2.1.2 Extended Kalman Filter

A limitation of the standard KF is that it is only applicable to linear systems. Most

Algorithm 2.1 Kalman Filter (KF)

Assumptions:

State,	$\mathbf{x}_k = \mathbf{A}\mathbf{x}_{k-1} + \mathbf{B}\mathbf{u}_k + \mathbf{w}_k$	(2.1)
Measurement,	$\mathbf{y}_k = \mathbf{C}\mathbf{x}_k + \mathbf{v}_k$	(2.2)
System Noise,	$\mathbf{w}_k \sim N(0, \mathbf{Q})$	
Measurement Noise,	$\mathbf{v}_k \sim N(0, \mathbf{R})$	
Independent Noise,	$E(\mathbf{w}_k, \mathbf{v}_k^T) = 0$	
Time Invariant,	$\{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{R}, \mathbf{Q}\}$	

Given:

System Model,	$\mathbf{A}$
Control Model,	$\mathbf{B}$
Measurement Model,	$\mathbf{C}$
Standard Deviation of w_k,	$\mathbf{Q}$
Standard Deviation of v_k,	$\mathbf{R}$

Initialize:

Estimated State,	$\hat{\mathbf{x}}_0$
Estimated Covariance,	$\mathbf{P}_0$

Input:

Control,	$\mathbf{u}_k$
Measurement,	$\mathbf{z}_k$

Output:

Optimal State Estimate,	$\hat{\mathbf{x}}_k$
Error Covariance Estimate,	$\mathbf{P}_k$

Predict:

State Prediction,	$\hat{\mathbf{x}}_{k\|k-1} = \mathbf{A}\hat{\mathbf{x}}_{k-1\|k-1} + \mathbf{B}\mathbf{u}_k$	(2.3)
Covariance Prediction,	$\mathbf{P}_{k\|k-1} = \mathbf{A}\mathbf{P}_{k-1\|k-1}\mathbf{A}^T + \mathbf{Q}$	(2.4)

Update:

Kalman Gain,	$\mathbf{K}_k = \mathbf{P}_{k\|k-1}\mathbf{C}^T[\mathbf{C}\mathbf{P}_{k\|k-1}\mathbf{C}^T + \mathbf{R}]^{-1}$	(2.5)
State Update,	$\hat{\mathbf{x}}_{k\|k} = \hat{\mathbf{x}}_{k\|k-1} + \mathbf{K}_k[\mathbf{z}_k - \mathbf{C}\hat{\mathbf{x}}_{k\|k-1}]$	(2.6)
Covariance Update,	$\mathbf{P}_{k\|k} = \mathbf{P}_{k\|k-1} - \mathbf{K}_k\mathbf{C}\mathbf{P}_{k\|k-1}$	(2.7)

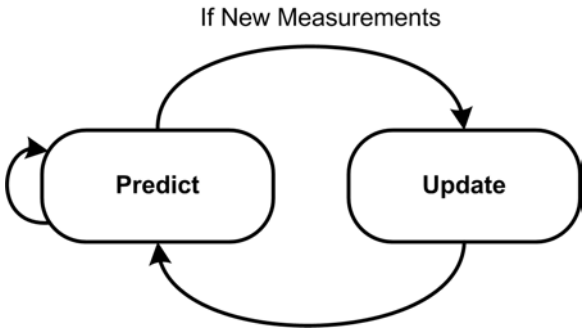

Fig. 2.1: Kalman filter process cycle.

real-world applications cannot be accurately modeled as only a linear system, notably real-world navigation is most accurately modeled as a nonlinear system [11]. The Extended Kalman Filter (EKF) dynamically linearizes the system equations to allow application of the KF equations (algorithm 2.2) [11–14]. The EKF is not optimal because the system must be linearized about the current state estimate. Its accuracy is dependent on how frequently it is run. This linearization is accomplished by using the first-order Taylor series expansion. It involves calculating a matrix of partial derivatives (Jacobian matrix) for functions $f()$ and $h()$ in equations (2.8, 2.9). The Jacobian matrices $\mathbf{F}_k$ (2.10) and $\mathbf{H}_k$ (2.11) must be processed at every predict and update time step.

2.1.3 KF and EKF Hardware Architectures

The computational complexity of even the linear KF makes it difficult to run the filter efficiently on traditional on-board microprocessors. KF acceleration approaches use both novel parallel architectures and algorithm enhancements to make the filter more computationally efficient [15–17]. Particularly, hardware implementations of KFs have been shown to dramatically improve performance [8, 18–20]. Just applying C to HDL methods have shown to be relatively inefficient [14, 19].

There have been some implementations of linear KFs on FPGAs in the literature [7, 14, 21], but these do not extensively address some of the limitations of specific features of the FPGA platform such as microprocessor or memory interfaces. Due to the inclusion of nonlinear functions and linearization, the EKF is more difficult to efficiently implement in hardware [13, 19, 22]. A flexible co-design allows the nonlinear functions to be processed on the microprocessor, so the filter can easily be adjusted for different problems.

The most computationally expensive operation in the Kalman filter is matrix inversion, which usually requires expensive division operations. Look-up tables (LUT) have been used to improve division latency [23], but LUT are not practical for floating-point division. Floating-point math has many advantages, like significantly increased precision and range. The trade-off is floating-point IP cores consume more area and introduce additional timing complexity, requiring more control logic.

Algorithm 2.2 Extended Kalman Filter (EKF)

Assumptions:

State,	$\mathbf{x}_k = f(\mathbf{x}_{k-1}, \mathbf{u}_k) + \mathbf{w}_k$	(2.8)
Measurement,	$\mathbf{y}_k = h(\mathbf{x}_k) + \mathbf{v}_k$	(2.9)
System Noise,	$\mathbf{w}_k \sim N(0, \mathbf{Q})$	
Measurement Noise,	$\mathbf{v}_k \sim N(0, \mathbf{R})$	
Independent Noise,	$E(\mathbf{w}_k, \mathbf{v}_k^T) = 0$	
Time Invariant,	$\{\mathbf{R}, \mathbf{Q}\}$	

Given:

Linearized State Model,	$\mathbf{F}_k = \frac{\partial f}{\partial x}\|_{\hat{\mathbf{x}}_{k-1\|k-1}, \mathbf{u}_k}$	(2.10)
Linearized Measurement Model,	$\mathbf{H}_k = \frac{\partial h}{\partial x}\|_{\hat{\mathbf{x}}_{k\|k-1}}$	(2.11)
Standard Deviation of v_k,	$\mathbf{R}$	
Standard Deviation of w_k,	$\mathbf{Q}$	

Initialize:

Estimated State,	$\hat{\mathbf{x}}_0$
Estimated Covariance,	$\mathbf{P}_0$

Input:

Control,	$\mathbf{u}_k$
Measurement,	$\mathbf{z}_k$

Output:

Optimal State Estimate,	$\hat{\mathbf{x}}_k$
Error Covariance Estimate,	$\mathbf{P}_k$

Predict:

State Prediction,	$\hat{\mathbf{x}}_{k\|k-1} = f(\hat{\mathbf{x}}_{k-1\|k-1}, \mathbf{u}_k)$	(2.12)
Covariance Prediction,	$\mathbf{P}_{k\|k-1} = \mathbf{F}_k \mathbf{P}_{k-1\|k-1} \mathbf{F}_k^T + \mathbf{Q}$	(2.13)

Update:

Expected Measurement,	$\hat{\mathbf{y}}_k = h(\mathbf{x}_{k\|k-1})$	(2.14)
Kalman Gain,	$\mathbf{K}_k = \mathbf{P}_{k\|k-1} \mathbf{H}_k^T [\mathbf{H}_k \mathbf{P}_{k\|k-1} \mathbf{H}_k^T + \mathbf{R}]^{-1}$	(2.15)
State Update,	$\hat{\mathbf{x}}_{k\|k} = \hat{\mathbf{x}}_{k\|k-1} + \mathbf{K}_k [\mathbf{z}_k - \hat{\mathbf{y}}_k]$	(2.16)
Covariance Update,	$\mathbf{P}_{k\|k} = \mathbf{P}_{k\|k-1} - \mathbf{K}_k \mathbf{H}_k \mathbf{P}_{k\|k-1}$	(2.17)

2.1.4 The Faddeev Algorithm

This Faddeev algorithm is a popular method for computing the Schur complement,

$$\mathbf{CA}^{-1}\mathbf{B} + \mathbf{D} \tag{2.18}$$

given the arrangement,

$$\mathbf{M}_{(N+P)x(N+M)} = \begin{bmatrix} \mathbf{A}_{NxN} & \mathbf{B}_{NxM} \\ -\mathbf{C}_{NxM} & \mathbf{D}_{PxM} \end{bmatrix} \tag{2.19}$$

by annulling matrix $\mathbf{C}$. Specifically, a $\mathbf{W}$ multiple of $\mathbf{A}$ is added to $-\mathbf{C}$ such that $-\mathbf{C} + \mathbf{WA} = \mathbf{0}$, meaning $\mathbf{W} = \mathbf{CA}^{-1}$. If $\mathbf{B}$ is also multiplied by $\mathbf{W}$ and added to $\mathbf{D}$, then the result in the lower right quadrant is $\mathbf{WB} + \mathbf{D}$, which expands to $\mathbf{CA}^{-1}\mathbf{B} + \mathbf{D}$. As shown in table 2.1, by properly arranging the four inputs ($\mathbf{A}$,$\mathbf{B}$,$\mathbf{C}$,$\mathbf{B}$) any of three basic matrix operations (matrix inverse, multiplication or addition), or a combination, can be performed.

KFs are composed of basic matrix operations: multiplication, addition, subtraction, and inversion. These operations can be efficiently implemented as systolic arrays (SA), particularly by using the Faddeev algorithm [7, 23–25], the benefits of which stem from its regularity, scalability, flexibility, and linearity.

Table 2.1: Examples of some matrix operation mapped to the Faddeev algorithm (not an exhaustive list).

Operation	Input(M=)	Output
Schur	$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ -\mathbf{C} & \mathbf{D} \end{bmatrix}$	$\mathbf{CA}^{-1}\mathbf{B} + \mathbf{D}$
Inversion	$\begin{bmatrix} \mathbf{A} & \mathbf{I} \\ -\mathbf{I} & \mathbf{0} \end{bmatrix}$	$\mathbf{IA}^{-1}\mathbf{I} + \mathbf{0} = \mathbf{A}^{-1}$
Addition	$\begin{bmatrix} \mathbf{I} & \mathbf{I} \\ -\mathbf{C} & \mathbf{D} \end{bmatrix}$	$\mathbf{CI}^{-1}\mathbf{I} + \mathbf{D} = \mathbf{C} + \mathbf{D}$
Multiplication	$\begin{bmatrix} \mathbf{I} & \mathbf{B} \\ -\mathbf{C} & \mathbf{0} \end{bmatrix}$	$\mathbf{CI}^{-1}\mathbf{B} + \mathbf{0} = \mathbf{CB}$
Multiplication & Addition	$\begin{bmatrix} \mathbf{I} & \mathbf{B} \\ -\mathbf{C} & \mathbf{D} \end{bmatrix}$	$\mathbf{CI}^{-1}\mathbf{B} + \mathbf{D} = \mathbf{CB} + \mathbf{D}$
Inverse & Addition	$\begin{bmatrix} \mathbf{A} & \mathbf{I} \\ -\mathbf{I} & \mathbf{D} \end{bmatrix}$	$\mathbf{IA}^{-1}\mathbf{I} + \mathbf{D} = \mathbf{A}^{-1} + \mathbf{D}$

2.2 Discrete Wavelet Transform

Many image/signal applications such as compression, target recognition, classification, etc., are composed of algorithms that can be accelerated by linear SAs. A subset of these algorithms and their SA implementations include: DWT [26], K-means clustering [27], Bayes classifier [28], eigenvalue calculation [29], etc. The DWT algorithm was selected for this application because it is a powerful filtering algorithm that has been used in aerospace applications [30, 31] for both on-board and off-line image compression. The 1D version of the algorithm involves multiple levels of decomposition, where each level includes a low-pass and high-pass filter followed by sub-sampling by 2. The 2D version of the algorithm first performs the low-pass and high-pass filtering across each row and then across each column of the result from the previous high-pass and low-pass filters, as shown in fig. 2.2(a). At each level of decomposition the image is sub-sampled by two in the x and y directions resulting in quarter sized subimages. This results in the recursively partitioned image structure in fig. 2.2(b). The inverse discrete wavelet transform (IDWT) can be used to completely recover the original image. No compression is accomplished directly by the DWT. Yet, a separate image compression algorithm can now be applied to the result of the DWT and achieve very efficient compression, due to the sparsity created by the DWT [31].

2.3 Systolic Arrays

A systolic array architecture is a network of simple processing elements (PE) which rhythmically process and pass data to nearby neighbors to process larger more complex tasks [8, 15]. Systolic arrays are most commonly 1D or 2D, since these structures can be efficiently mapped onto VLSI (Very Large Scale Integration) designs. A systolic array usually only consists of 1, 2, or 3 types of PEs, which are repeated throughout the design. This simplifies the design process, since the designers may focus on optimizing a single PE. Systolic arrays have a regular data flow and structure, this simplifies scheduling and reduces the number of control signals. Since each PE is only connected with its neighbors, the routing overhead and signal latency is reduced. Matrix operations can be efficiently implemented as systolic arrays by utilizing the parallelism, scalability, resource reuse, and

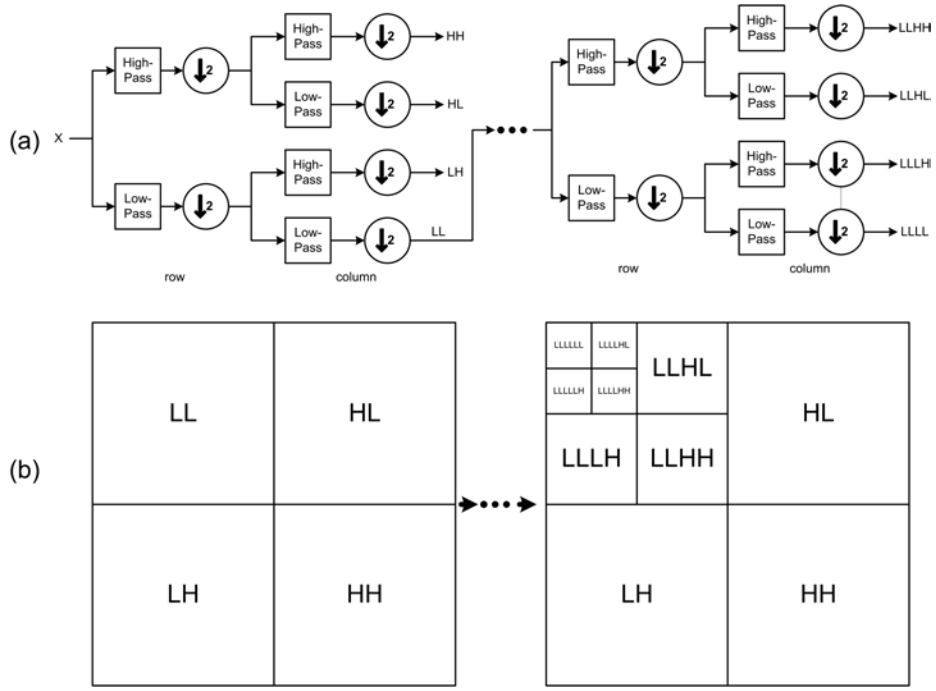

Fig. 2.2: (a) 2D DWT algorithm. (b) Image partitioning after three decomposition levels. L=Low-pass. H=High-pass.

minimal data communication provided by systolic arrays [8, 15].

2.4 Reconfigurable Architectures

During run-time the system model or requirements may change due to sensor/actuator failure, environment changes, or at scheduled times. Many authors have proposed reconfigurable systems to handle these situations [21, 32–35]. These approaches use soft-reconfiguration, which dynamically merges or switches between multiple filters. To the author's knowledge, no Kalman filter implementation actually dynamically reconfigures the hardware. This technique was suggested as a future research direction by Salcic and Lee [21].

There have been several other dynamically reconfigurable processors designed for other applications. Many of these approaches use custom coursed grained reconfigurable architectures [36]. These architectures are dependent on new novel hardware platforms being manufactured. Using currently available FPGAs and partial reconfiguration methods have the benefits of cost and faster time to market.

2.5 Partial Dynamic Reconfiguration

Partial Dynamic Reconfiguration (PDR) [37–39] is the process of reconfiguring only a portion of an FPGA at run-time, after initial configuration, while the other portions remain active. PDR is one of the most efficient ways of having different applications on single device, hence an efficient use of space on the device. This can reduce device count, power consumption, and overall cost. The time it takes to reconfigure a portion of the FPGA will be proportionally smaller than reconfiguring the entire device.

Currently the tool provided by Xilinx for doing dynamic partial reconfiguration is by following the Early Access Partial Reconfiguration (EAPR) method [40]. This process involves first implementing the part of the logic which will not change during run-time, called static logic [39]. The logic that will change during run-time is implemented as a partially reconfigurable region (PRR or PR region). Next a bitstream for the initial configuration of the entire chip, and one for each alternate configuration for each module implemented on PR region is generated. Busmacros (BM) are used to bridge communication between the static and PPRs. These busmacros must be placed manually for each PRR. The alternate configurations for a PR region can be configured through an off-chip interface, like JTAG, or the on-chip interface, the Internal Configuration Access Port (ICAP). On the Virtex-4 FPGAs, frame addresses span the height of 16 CLBs (one HCLK row) instead of the entire height of the chip as was seen in previous Virtex FPGAs [41] (fig. 2.3). With EAPR, two partial regions may not overlap vertically in the same clock region, so a clock region basically sets the granularity of the partial region sizes. Increasing the height of the PR region by one slice into the next clock region would cause it to occupy both clock regions, restricting any other partial region from occupying either clock region. This is because EAPR does not allow reconfiguration of only a portion of a frame. It is, however, possible to place multiple PR regions in separate columns of a clock region.

2.5.1 Partial Bitstream Compression and Decompression

As the complexity of FGPA architectures have increased, so has the bitstream size that is required to configure the device [39]. With PDR being extensively used in the designs for

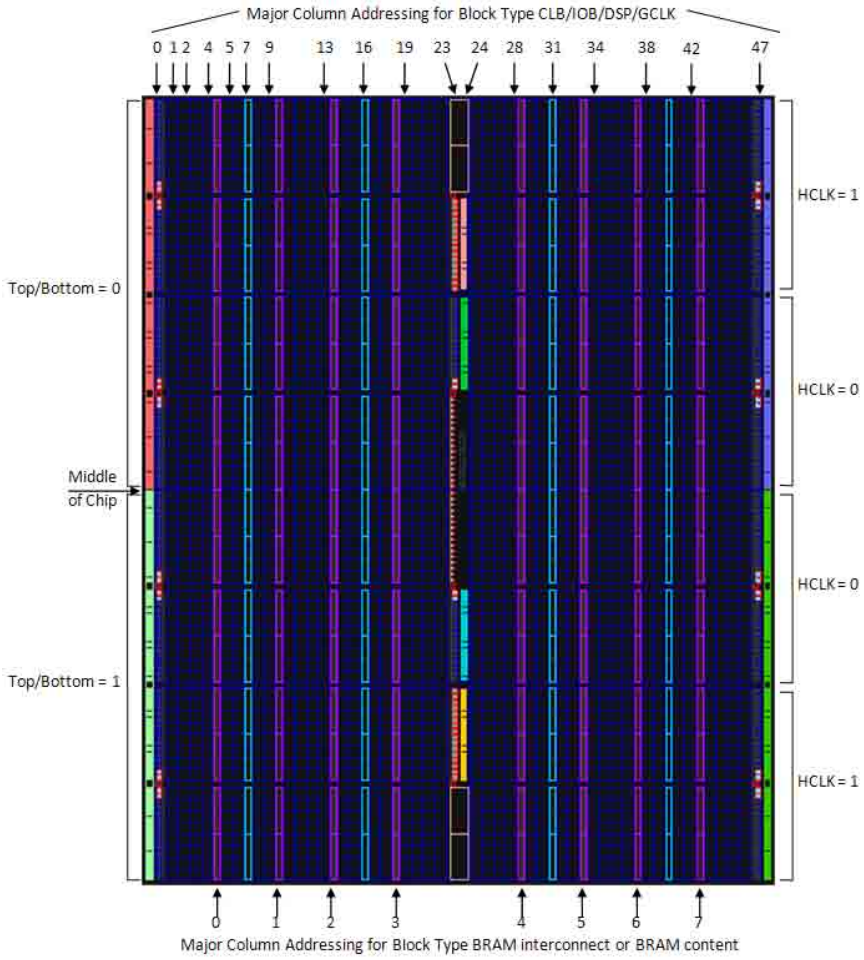

Fig. 2.3: Configuration layout of the Virtex 4 FPGA.

better efficiency, there is a need to reduce the overhead of reconfiguration, which increases as the bitstream size increases. Storing the bitstream on on-chip memory (BRAM) results in less reconfiguration time compared to off-chip storage [39]. Thus, compressing bitstreams and storing as many bitstreams on the chip as possible is advantageous [42]. Run-length encoding (RLE) for the compression of partial bitstreams (PB) is used, as it provides a consistently good performance for most of the test cases without the need for storing a dictionary on limited BRAM resources. Other compression methods that have been used for bitstream compression include LZW [43], intra-configuration compression [44], optimized RLE [42], and arithmetic coding [45].

2.5.2 Partial Bitstream Relocation

A powerful technique to augment PDR is the concept of bitstream relocation. It involves slightly modifying the contents of a PB to mold it into a form that can be loaded onto a similar but different PRRs on the device. This is done to reduce the number of partial bitstreams needed to store and compile for a design. For example, if there were five PR regions that the bitstream could be relocated to, it would save storage of four bitstreams and compilation time of those bitstreams. The trade-off is the additional overhead of software and hardware, and the strict restrictions on floorplaning required to enable relocation. These methods are strongly tied to a specific family of devices and system architecture on the FPGA.

The design presented here integrates the methods developed by Carver [39, 46] with a RLE decompression algorithm. A limitation of this solution is it does not allow static routing to pass through the PRRs. Other PDR methods have ability to relocate non-identical columns [47], find and correct faults in the bitstream [48], allow static routing through PRRs [49], and reduced relocation overhead [50, 51]. While each of these PDR enhancements is effective to a certain extent, a polymorphic systolic array can benefit considerably through a suitable application of a combination of relocation and compression. To the author's knowledge, no prior work has combined these approaches.

Chapter 3

Proposed Design

First, the system architecture containing the polymorphic systolic array (PolySAF) implemented on an FPGA is presented. Then the mapping of the EKF and DWT algorithms onto this architecture is presented. Finally is a section on how on-chip bitstream decompression/relocation methods were used to facilitate dynamic scaling of the architecture.

3.1 System Architecture

The system on the FPGA consists of a microprocessor and a co-processor (fig. 3.1). The co-processor contains the controller, a pseudo-cache and the polymorphic systolic array framework (PolySAF). The co-processor is sent instructions and data from the microprocessor over a 32-bit FSL (fast system link) bus. The controller transacts with the pseudo-cache to marshal data for the PolySAF.

3.1.1 Microprocessor

The microprocessor used in the case study is the Xilinx soft-core (configured in the FPGA logic) MicroBlaze [52] processor with an included internal floating-point unit. The microprocessor serves three purposes: (i) It is available for computing portions of an algorithm that are deemed better suited for execution in software. For instance, in the EKF algorithm, the nonlinear functions are better suited for software-based execution. (ii) It hosts software necessary to support partial dynamic reconfiguration, bitstream decompression, and relocation. (iii) It is responsible for marshaling data and scheduling operations on the co-processor.

3.1.2 Pseudo-Cache

The pseudo-cache is so named because while it does not have all the features of a

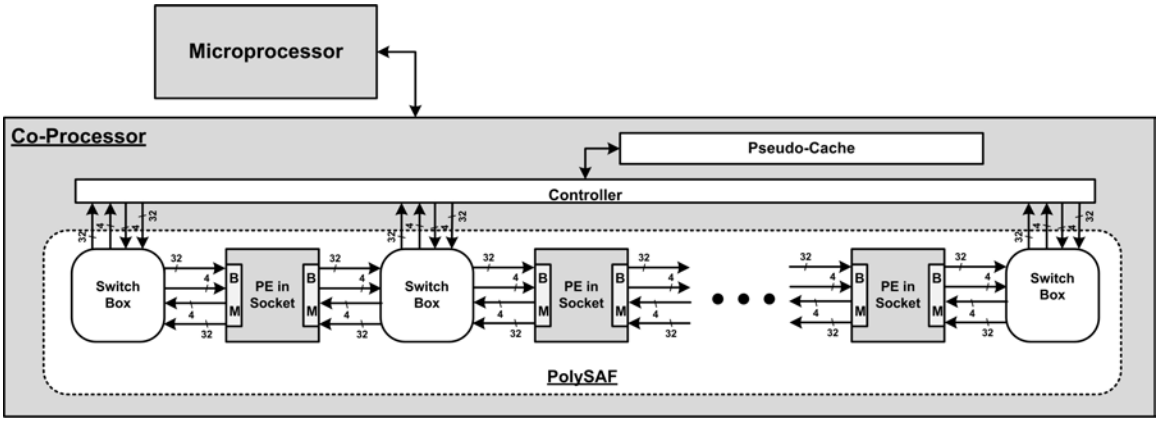

Fig. 3.1: This figure shows the system architecture with details on the PolySAF.

traditional cache, it serves as a partially refreshable buffer storing a sub-set of the microprocessor memory's contents, and provides low latency read/write access to the PolySAF. Soft-tables on the microprocessor keep dirty bits for the data stored in both memories. When data is made dirty by the microprocessor the corresponding pseudo-cache blocks are freed, and the data must be refreshed on the co-processor if it is used there again. If data is made dirty by the co-processor the cached version is sent back to the microprocessor if it is used there again. This ensures data is only synchronized between the microprocessor and co-processor when necessary. The pseudo-cache is partitioned into blocks. A matrix is stored in the pseudo-cache as one matrix row per block. This simplifies addressing, but more importantly allows access to the transposed version of a stored matrix by incrementing the address by blocks rather than words. The draw-back of this technique is a matrix must be stored in contiguous blocks, which can cause fragmentation in the pseudo-cache.

3.1.3 Software Interface

Instructions for reading and writing data to the co-processor's pseudo-cache from the microprocessor, reading and writing data from the co-processor's pseudo-cache to the PolySAF, programming the switchboxes, and reseting the co-processor are made available. Table 3.1 shows a generic co-processor instruction bitmap and table 3.2 describes the available opcodes.

Table 3.1: Bitmap for the co-processor instruction. R = Reserved.

31:28	27:23	22:19	18	17:13	12:8	7	6:0
opcode	data tag	swtichbox selection	R	offset	count	R	block

Table 3.2: List of co-processor instructions based on the instruction in table 3.1.

Instruction	Description
FSL Write	This instruction is followed by the given number of elements sent from the microprocessor, which are written to the pseudo-cache starting at the given offset within the given block.
FSL Read	This instruction is followed by the given number of elements sent from the pseudo-cache to the microprocessor, starting at the given offset within the given block.
Array Read	The controller reads the given number of elements from the selected swtichbox and writes them to the pseudo-cache starting and the given offset within the given block.
Array Write	The controller writes the given number of elements to the selected swtichbox with the given control tag. The elements are read from the pseudo-cache starting and the given offset within the given block.
Array Transposed Write	The controller writes the given number of elements to the selected swtichbox with the given control tag. The elements are read in from the pseudo-cache at the given offset within a block. Starting at the block given, the address is incremented by a block for each read.
Set Swtichbox	The given switchbox is reprogrammed with the given tag.
Reset	Resets the co-processor and PolySAF.

The software (written in C) provides multiple layers of abstraction for utilizing the co-processor (fig. 3.2). First a layer of wrapper functions for the instructions in table 3.2 and functions for maintaining consistency with the pseudo-cache (e.g. *dirty*() and *sync*()) are provided. On top of this layer is a layer of functions for each application specific array being implemented. For example, a matrix operation is accelerated by a Faddeev systolic array (FSA), configured on the PolySAF, is executed with the function call: *void schur(Matrix* E, Matrix* A, Matrix* B, Matrix* C, Matrix* D,...);*. The source code for the schur function is available in Appendix A.1. The abstract data type *Matrix* contains all of the matrix elements, the size of the matrix, and its memory location (microprocessor memory, co-processor pseudo-cache, or both). Finally, a top layer of functions pertaining

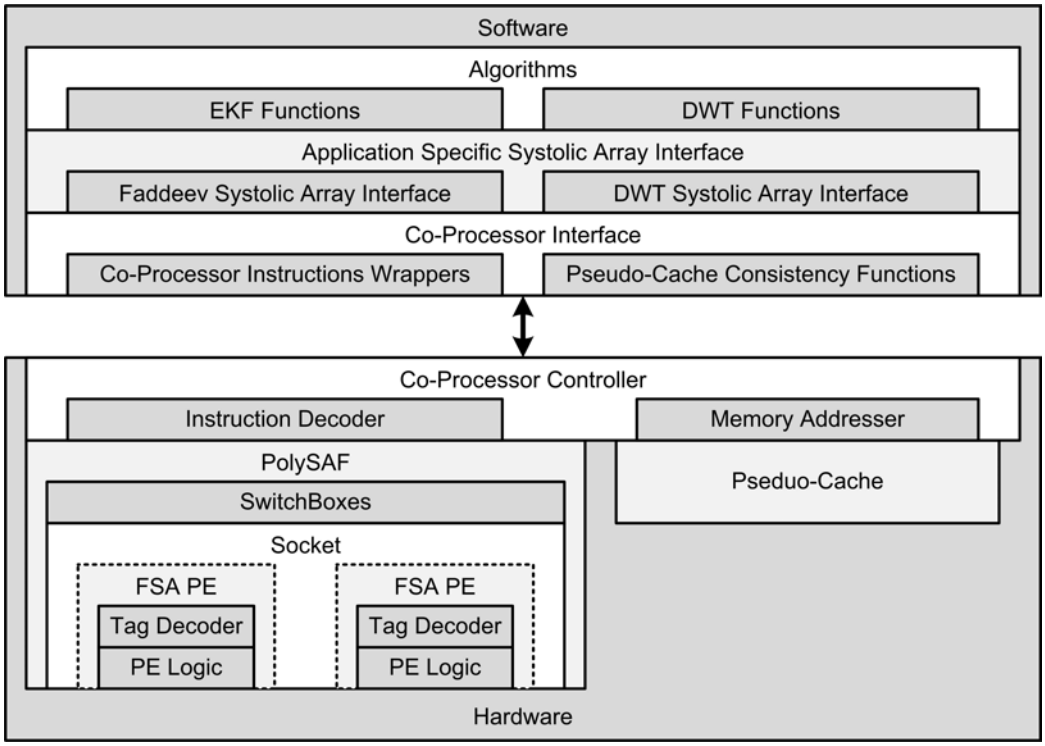

Fig. 3.2: This figure shows the software interface abstraction layers and the hardware interfaces within the co-processor for implementing the EKF and DWT algorithms.

to each algorithm is provided (e.g. *Predict()* for the EKF).

3.1.4 PolySAF

The PolySAF is composed of Sockets that are PR regions (fig. 3.1). Each socket has four 32-bit buses and four 4-bit (expandable to up to 8 bits) buses that link it to its two neighboring switchboxes. The 32-bit buses are intended to allow PEs residing in a socket to send and receive up to 32-bit data elements. The 4-bit buses are intended to carry control information, i.e. data tags. This tag value is specified in the instruction from the microprocessor for each set of data sent to the PolySAF. Within each socket, asynchronous busmacros (BM) are inserted to allow wires from the PR region (contained within the socket) to connect to wires from the static switchboxes. By coordinating the reconfiguration of the sockets and controlling the routing inside switchboxes, it is possible to dynamically scale the number of participating PEs in a systolic array.

3.1.5 Programmable Switchboxes

In order to enable the mapping of different systolic arrays to the PolySAF, it must be possible to change the routing between PEs for each specific systolic array. The switchboxes allow the data and control buses to be looped back to the source socket, routed to the next socket, routed to the controller, or disabled. The switchboxes must allow disabling of a sockets ports, because during partial reconfiguration a socket will have erroneous outputs. Rather than allow for all 64 possible routing configurations, a subset of 27 possible routing configurations, listed in table 3.3, is allowed. A switchbox is modeled after fig. 3.3 with five 36-bit 2-to-1 muxes, three 4-bit enable gates, and one 5-bit register. The register value sets the select bit for the MUXES and the enable bit for the AND gates. Its value can be changed by the controller. The switchboxes could also be implemented as PR regions. This method is not efficient, however, as it would take orders of magnitude longer to reconfigure a switchbox compared to reprogramming a switchbox (1 cycle). Since these switchboxes do not introduce extra clock cycles, the timing characteristics of a SA should remain the same after being mapped to the PolySAF.

3.1.6 Scaling

When two systolic arrays are concurrently sharing the PolySAF, it is possible to scale-up one systolic array by increasing the number of sockets to host its PEs, at the cost of scaling-down a proportional number of PEs belonging to the other systolic array. Specifically the following transfer of control protocol shown with an example in fig. 3.4 is used. The

Table 3.3: Routing configurations allowed by a switchbox. E=East, N=North, W=West and 0=Disabled.

Destination	Source													
E	0	E	W	N	0	0	0	0	0	0	E	E	E	0
N	0	0	0	0	E	W	0	0	0	E	W	0	0	W
W	0	0	0	0	0	0	E	W	N	N	0	W	N	N
E	W	W	W	N	N	N	N	E	W	N	N	0	0	
N	E	0	0	E	W	0	0	W	E	E	W	E	W	
W	0	E	N	0	0	E	W	N	N	W	E	W	E	

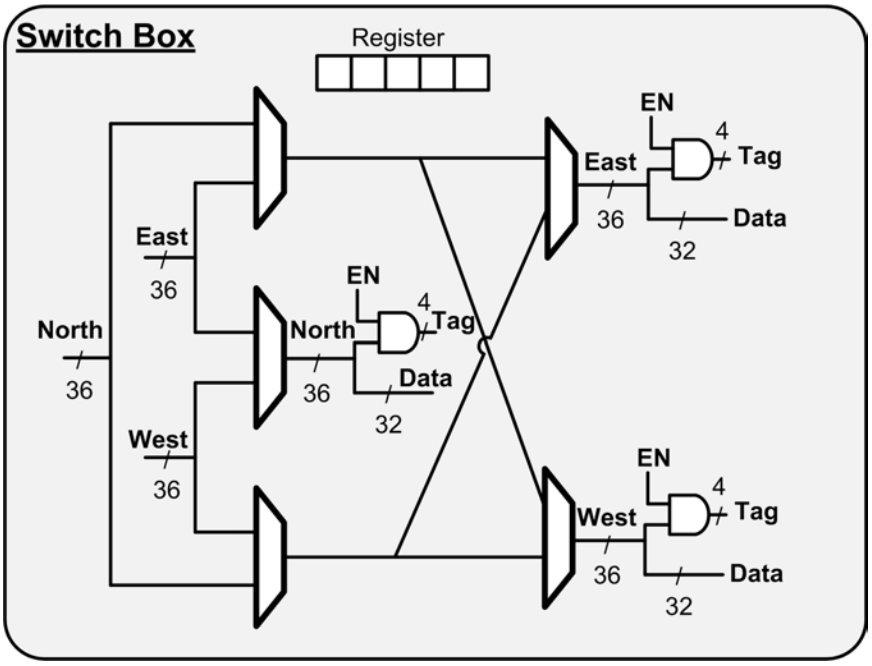

Fig. 3.3: This figure shows the internal structure of a switchbox.

first step involves disconnecting a socket (socket C in fig. 3.4(a)) from the SA it currently belongs to (App 1). This requires rerouting data and control signals inside neighboring switch boxes (fig. 3.4(b)). The second step (fig. 3.4(b)) involves reconfiguring the socket via the ICAP from the microprocessor. During this process both systolic arrays are still functional, albeit with one of them having a lesser number of PEs. The state of the registers within a newly reconfigured PR is unknown because PDR does not allow the registers to be initiated. So the third step involves resetting the PE in the newly configured socket. The fourth step involves rerouting signals in the appropriate switch boxes to augment the SA of App 2 with the newly created PE (fig. 3.4(c)).

3.2 Application of the EKF

3.2.1 Hardware-Software Partitioning

The EKF algorithm (algorithm 2.2) can be partitioned into nonlinear equations (2.12, 2.14, 2.10, 2.11) and linear algebra equations (2.13, 2.15, 2.16, 2.17). The nonlinear equa-

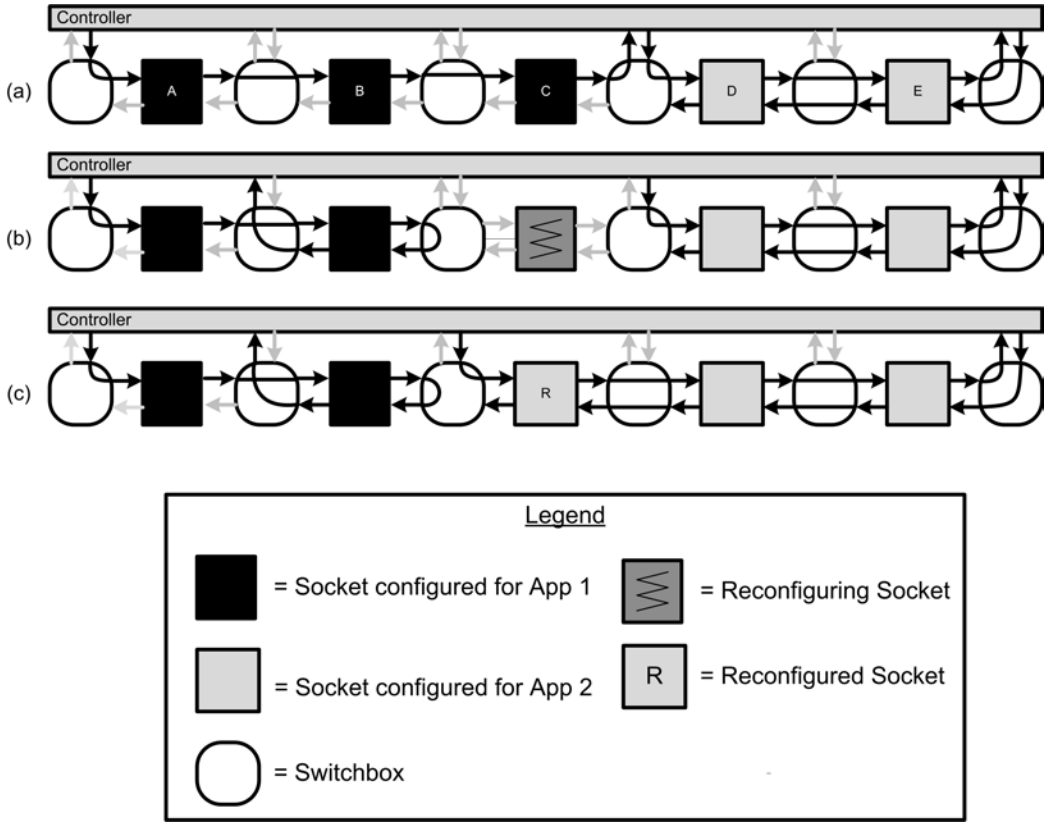

Fig. 3.4: This figures shows an example of scaling down one SA (App 1) and scaling up another SA (App 2).

tions are unique to each EKF instance, therefore it is impractical to create a hardware accelerator for this portion of the algorithm. Instead the nonlinear equations are implemented in software on the embedded microprocessor. Since the linear algebra equations in the predict and update phases are consistent across most EKF instances, varying only in scale, they can be efficiently mapped to a hardware accelerator. Specifically the EKF is mapped onto a SA that implements the Faddeev algorithm, as shown in fig. 3.5. Two runs of the Faddeev SA (FSA) are needed for computing the Predict phase and seven runs are needed for the Update phase.

3.2.2 Faddeev Systolic Array

The direct mapping of the Faddeev algorithm to a 2D systolic array array is shown in fig. 3.6 [7]. It is composed of boundary cells and internal cells. The staggered matrices

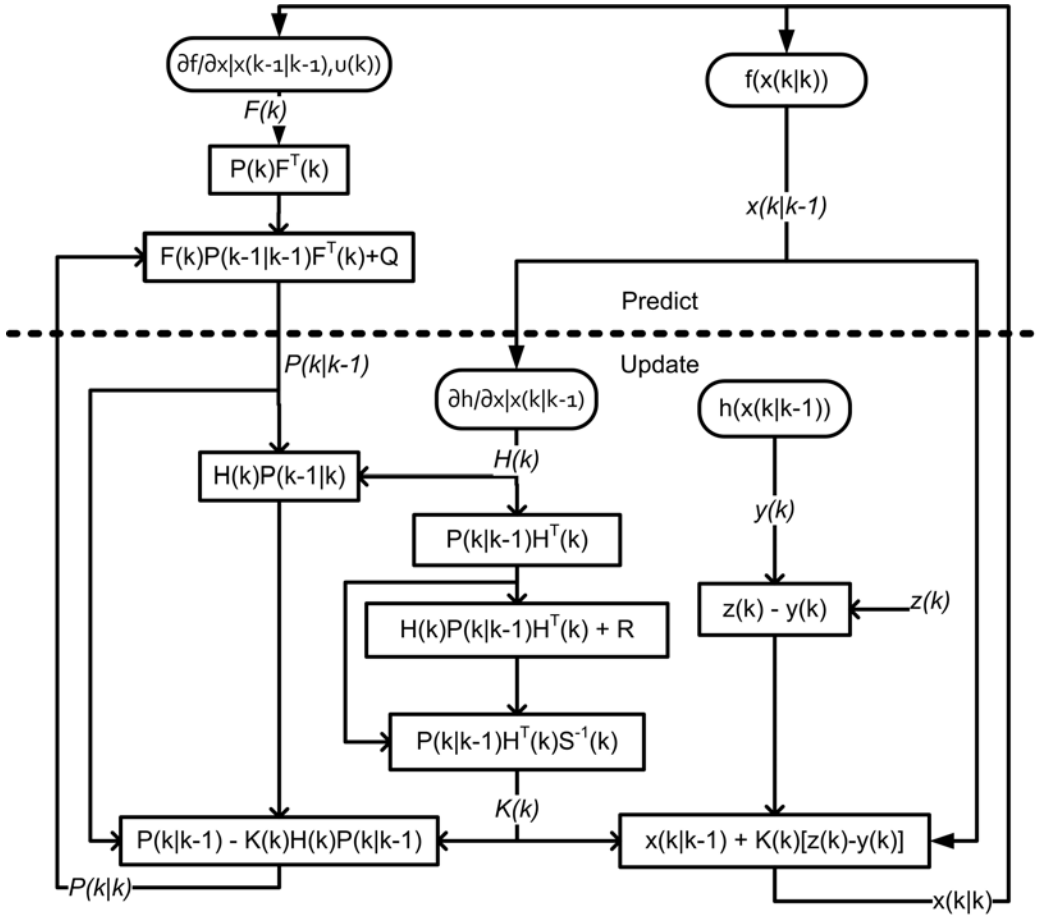

Fig. 3.5: This figures shows the mapping of the EKF onto the Faddeev algorithm. Rounded boxes represent operations performed in software on the microprocessor, and squared boxes represent operations performed by the a the hardware accelerator.

A,B,C and **D** from equation (2.19) are the input. The SA operates in two phases. First matrix [**A B**] is triangulated. During this phase nearest neighbor pivoting is performed to maintain numeric stability. Nearest neighbor pivoting is invoked (via the swap flag) when the incoming element is greater than the stored element in the boundary cell (table 3.4. The internal cells operate according to table 3.5. After the first phase the triangulated matrix [**A B**] will be stored in the array. Next matrix **-C** is annulled in the matrix [**-C D**]. The final result will be the staggered output from the last row of the SA (fig. 3.6).

In a scalable SA the number of processing elements (PEs) and the size of the input must be independent, since the number of PEs can change irrespective of the input. A 2D SA scales by $O(2N)$ PEs while a 1D linear SA scales by $O(1)$. A single-precision floating-

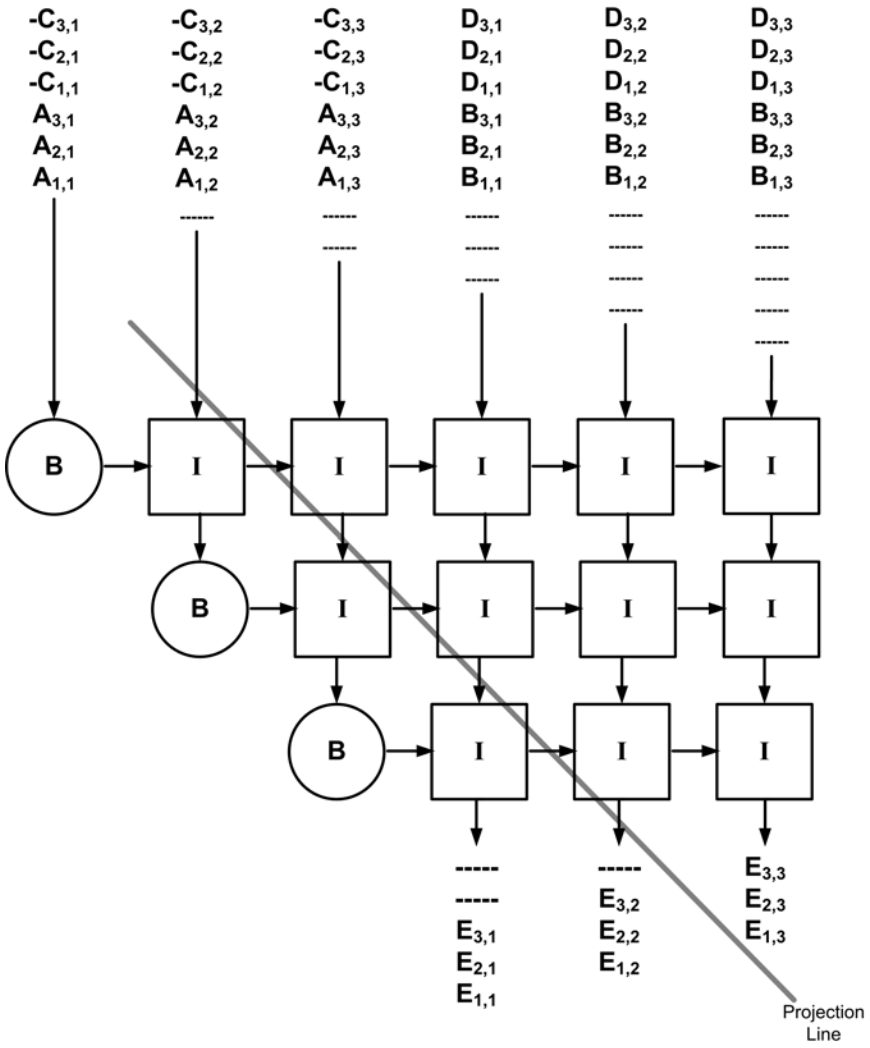

Fig. 3.6: 2D direct mapping of the Faddeev systolic array. B=Boundary Cell, I=Internal Cell.

Table 3.4: Boundary cell.

Matrix Row	$\|x\| > \|P\|$	Q	swap	New P
A/B	1	$-P/x$	1	x
	0	$-x/P$	0	P
C/D	Don't Care			

point based Faddeev PE can require 724 slices allowing approximately 7 to be concurrently placed on a Virtex-4 SX35 FPGA, if routing and control logic are included. In order to have reasonable granularity for scaling in this application, a linear SA is needed.

Table 3.5: Internal cell.

swap	$\|x\| > \|P\|$ Output	New P
1	$Qx + P$	x
0	$QP + x$	P

The Faddeev SA (FSA) used in this application is the result of by projecting the 2D SA onto a vertical array consisting of one boundary cell and one internal cell for each level/PE (fig. 3.7(a)). The input to the FSA as the row-major ordered stream of **M** from equation (2.19). Each element is tagged to identify its respective matrix (**A,B,C or D**), with exception of elements from the first and last row, which are tagged accordingly. The first element of a row entering a PE is sent to the boundary cell, which calculates the quotient (fig. 3.8 and Appendix A.2). The rest of the elements in a row (i.e., not the first element in the row) are buffered for the internal cell. When the quotient from the boundary cell is available, processing of these elements begins. The internal register P is implemented as a self-feedback FIFO. During processing the next element (i.e., the next column in fig. 3.6) is popped from the FIFO then sent to the back of the FIFO. This register cannot be implemented as a simple shift register because the length of the input is variable. Initially the P FIFO is initialized with the elements of the first row of an input matrix(**M**). On the last row the FIFO is emptied, placing the PE in a cleared state for the input matrix(**M**).

With this FSA (fig. 3.7(a)) an over-sized ($PE < N$) input is handled by recursively processing the data until the result is reached (symbolized as a dotted line in fig. 3.7(a/b)). As the number of PEs in the SA increases, the number of times the output needs to be recursively processed is proportionally reduced according to $\lceil \frac{N}{2R} \rceil$, where R is the number of PEs/sockets, and N is the height and width of the input matrix **A**. This feature allows the SA to be easily scaled, which is one of the primary reasons it is used in this system.

For an input of size $4NxN$ (where A, B, C, D are all of size NxN) the size of the output at each level is $(2N - i)x(2N - i)$ for $i = 1...N$, so the final output is NxN. Even when another stream is started immediately after the first, there will be a gap in the data stream where the PEs are not being utilized. So the data stream is looped back through the PEs

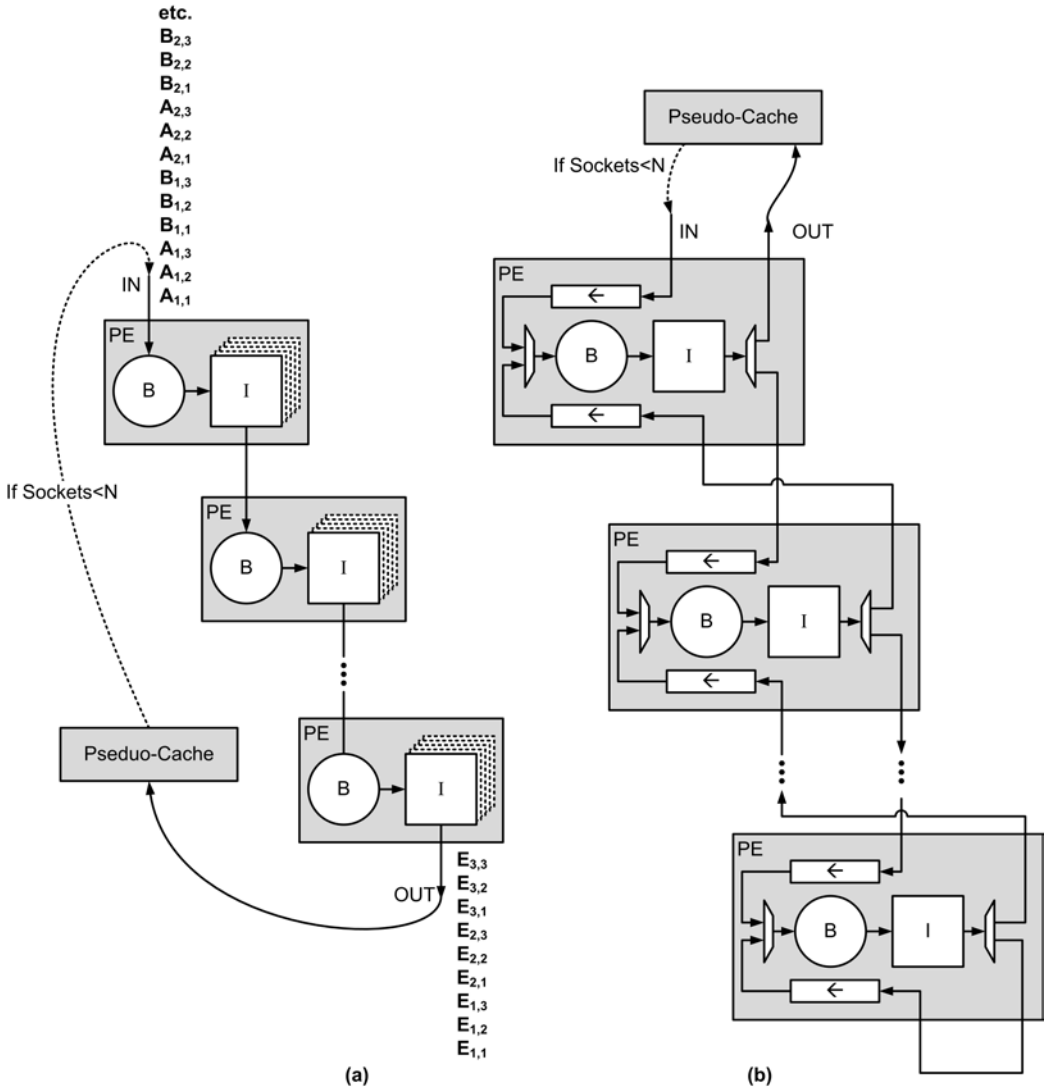

Fig. 3.7: (a) Vertical projection of the 2D Faddeev SA in fig. 3.6. (b) Looped adaptation of fig. 3.7(a). B=Boundary Cell, I=Internal Cell.

to increase utilization (fig. 3.7(b)). To allow this looping, each PE separately buffers the top and bottom inputs, and then separately schedules the streams onto the boundary and internal cells.

It can be observed that if each arithmetic operation took one clock cycle, this architecture would be inefficient. Yet, in the case of single precision floating-point arithmetic, each operation takes multiple cycles. Specifically, the single precision floating-point divider had to be reduced by 70% so that the FSA PE containing a divider would fit within a socket.

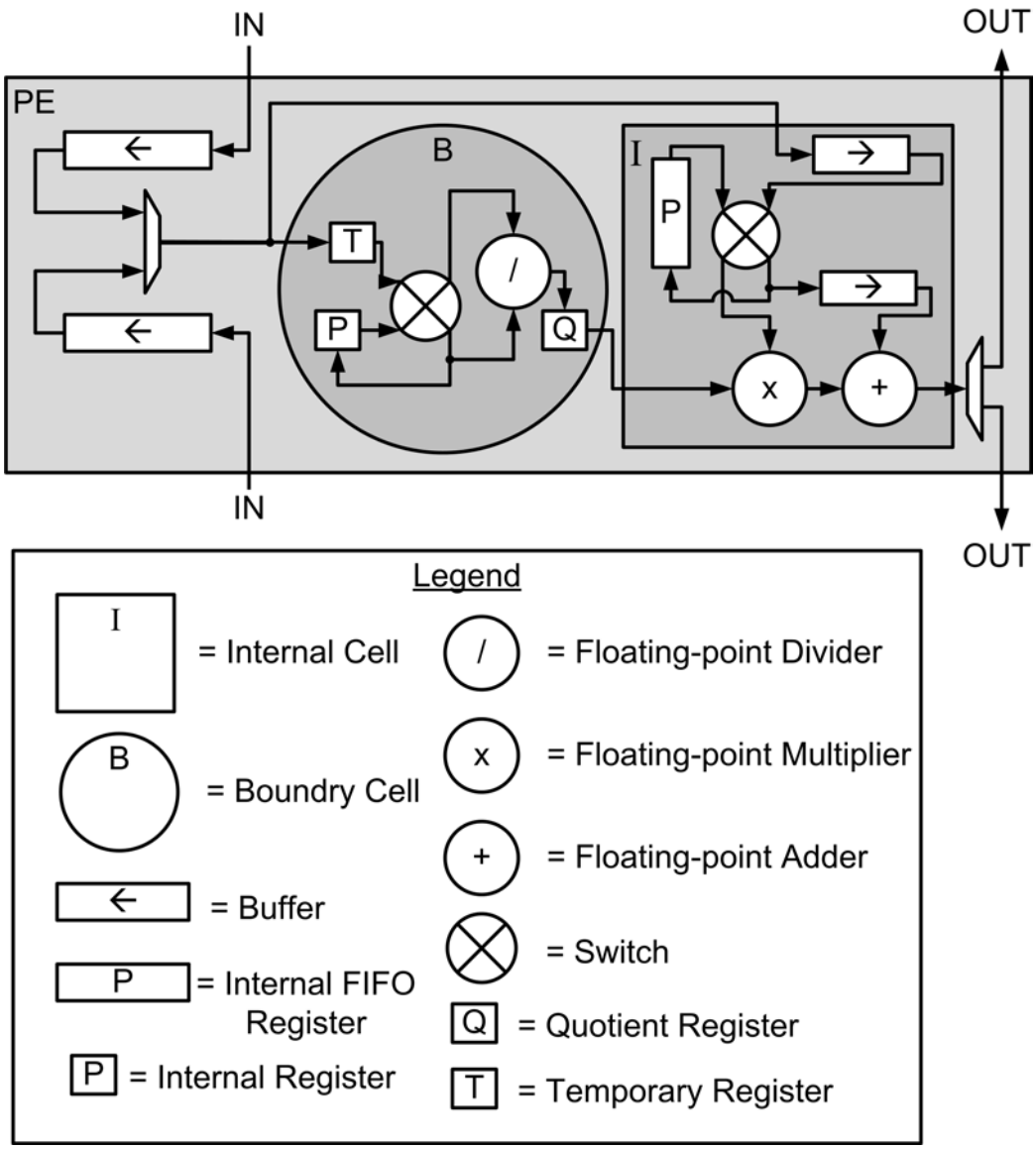

Fig. 3.8: FSA internal architecture.

This was accomplished by reducing its input rate from one element every clock cycle to one element every 14 clock cycles. So after the divider starts calculating one element, up to 14 elements of the previous row (stored in a buffer) can be procecessed by the internal cell. Another feature of this SA is it uses only a single input and output stream, which simplifies the control and memory interface to the array. This is important since the SA will be applied to the PolySAF that has limited flexbility in its interface.

3.3 Application of the DWT

The systolic array implementation of the DWT (DSA: DWT systolic array) shown in fig. 3.9 is a modification of other similar designs [26]. Buffers to allow usage of pipelined single precision floating-point cores have been added to the basic architectures. Also, the architecture was modified to only pass elements directly to neighboring PEs, rather than skipping a PE. The input is fed in at the left of the SA and directly passed to the next PE to the right. Partial sums are fed into the right of the array and are buffered until a corresponding input is received from the left. When an input is available it is multiplied by the low-pass and high-pass weights, added to corresponding partial sums, then passed to the next PE to the left. For the initial iteration the partial sums feed into the array will be zeros. In recursive iterations, the partial sum input will be the output of the previous iteration. Since the output is decimated by two in the DWT, only half of the operations are actually required. A bit in the control tag of each element is toggled so each PE will ignore every other element.

If the number of taps is greater than the number of available PEs each level must be run multiple times by passing the output of the SA to the partial sums input of the last PE (symbolized as a dotted line in fig. 3.9). Therefore, the DSA computes per run, a high-pass filter operation, a low-pass filter operation, and decimation by two. This results in the need for three such runs to compute one level of decomposition for the 2D DWT, resulting in the LL (lo-low), LH (low-high), HL (high-low), and HH (high-high) subimages.

3.4 Hybrid PDR

The Hybrid PDR process presented by Carver [39,46] involves augmenting EAPR with partial bitstream decompression (PBD) and partial bitstream relocation (PBR) software. This Hybrid PDR method has been integrated with the PolySAF to improve scaling performance. The Hybrid PDR processes run on the embedded microprocessor and reads partial bitstreams (PB) from embedded BRAMs. Figure 3.10 illustrates the basic flow of the partial bitstreams.

Run-length encoding (RLE) is used to compress the partial bitstreams offline. The

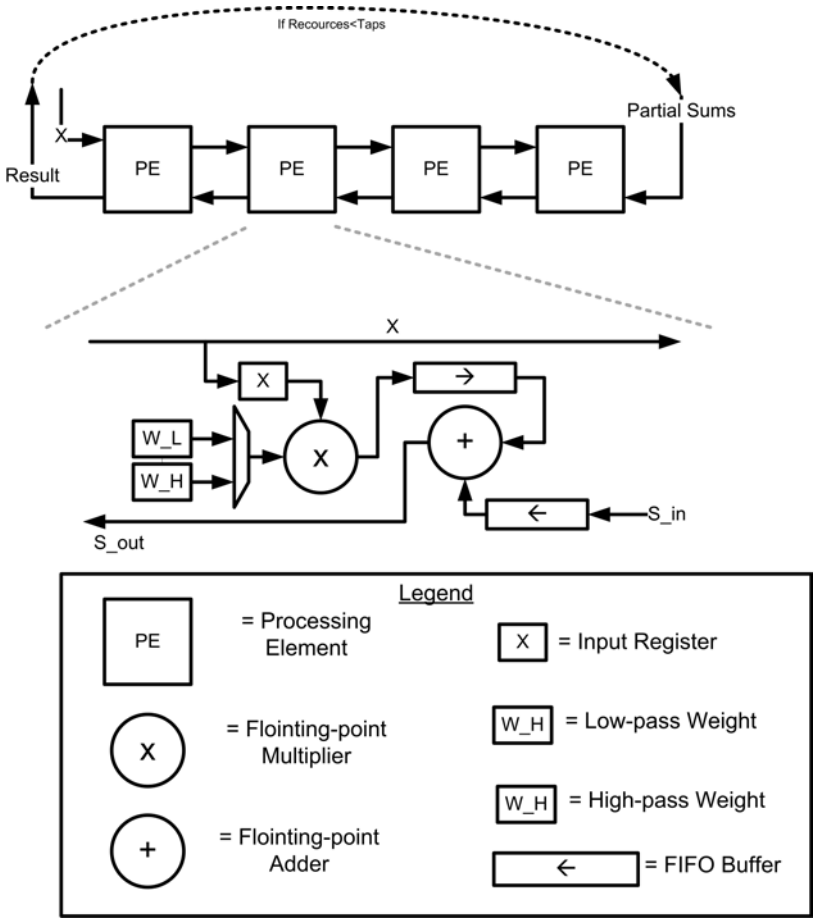

Fig. 3.9: DSA internal structure.

RLE algorithm implemented here compresses sequences of up to 127 bytes into two bytes. These compressed PBs are stored on a compact flash card that attaches to the ML402 board and communicates with the FPGA through the SystemACE. At startup of the software on the MicroBlaze transfers two compressed PBs (one for the FSA PE and one for the DSA PE) to the embedded BRAMs. The reason for caching the PBs on BRAMs is to avoid the long delay of reading the PBs from the flash card. During decompression, only 128 bytes of memory is required, which is a key advantage over other decompression algorithms that require a dictionary. The decompressed portions of the partial bitstreams are then sent to the relocation software. The relocation software offsets the frame addresses in a PB so it can be configured in multiple socket locations. If the source and destination locations are on opposite sides of the chip (top vs. bottom) the frame contents must be bit-reversed, since

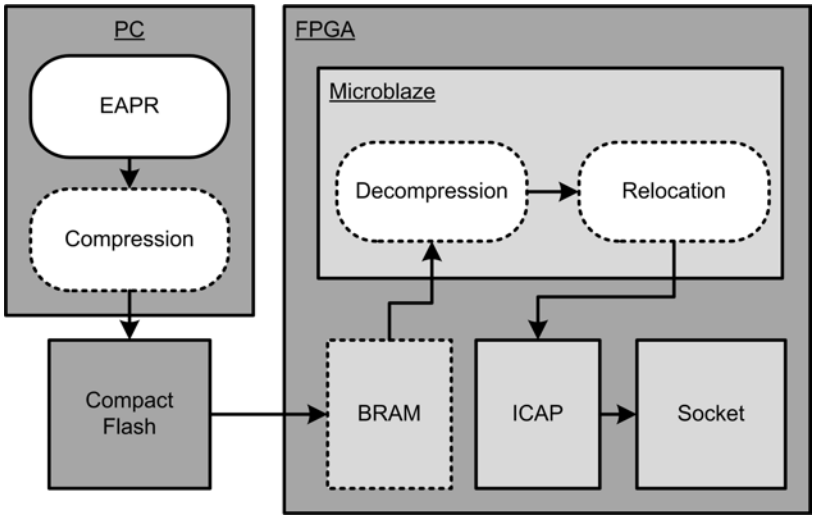

Fig. 3.10: This figure shows the data flow of a partial bitstream in the Hybrid PDR method. Dotted boxes represent optional stages.

the bottom half the chip is a mirror image of the top half of the chip. The relocated PB is then sent to the OPB (Open Peripheral Bus) HWICAP (Hardware ICAP) buffer (capacity of 2K bytes). The decompression and relocation software run iteratively until the ICAP buffer is filled. At that point, the microprocessor triggers the ICAP which configures the device with the buffered frames at the given addresses.

The reason for choosing a bitstream relocation method was based on a salient feature of systolic arrays: the PEs are architecturally identical. It is inefficient to store a PB on BRAM for every possible location of each PE. It is more efficient to store a single version of each type of possible PE and relocate to the desired PR region.

Chapter 4

Results and Analysis

4.1 FloorPlan

The layout of the floorplan for the system on the Virtex-4 SX35 FPGA is shown in fig. 4.1(a). It can be seen that the sockets (PR regions) of the PolySAF are distributed on the left side of the chip (highlighted in white) and the components of the static region (MicroBlaze, pseudo-cache, controller, switchboxes, etc.) are distributed on the right side of the chip, except for one clock region on the left side that is also allocated for the static region. This clock region had to be allocated as part of the static region because: (i) there is an I/O port that connects to the System ACE on the ML402 board, requiring a static route through this clock region, which prohibits placing a relocatable PR region of the dimensions in this design. (ii) Additional BRAMs were required for the Microprocessor memory and pseudo-cache that were in scarcity, hence making them unavailable for a sixth socket. The busmacros have been stacked on the right side of the sockets, to avoid necessitating any static routes passing through the PR regions. Figure 4.1(b) shows that no static signals cross the PR boundaries except for clock signals. The Virtex-4 SX35 is used for this case-study, but a larger FPGA can fit a proportionally greater number of PR regions. It is estimated that the Virtex 4 SX55 can hold up to 14 sockets.

As noted before, a frame (which is the same height as a clock region) is the smallest component of reconfiguration. The size of the PR regions were specifically chosen to be the height of one frame to maximize the trade-off between flexibility, resource utilization, and reconfiguration time. A larger PR region would require occupying another clock region, which would make reconfiguration slower (must reconfigure two frames per column occupied by the PR region instead of one), waste BRAMs and DSPs (if they are only partially covered by the PR region), and would limit the number of sockets that could be allocated on the

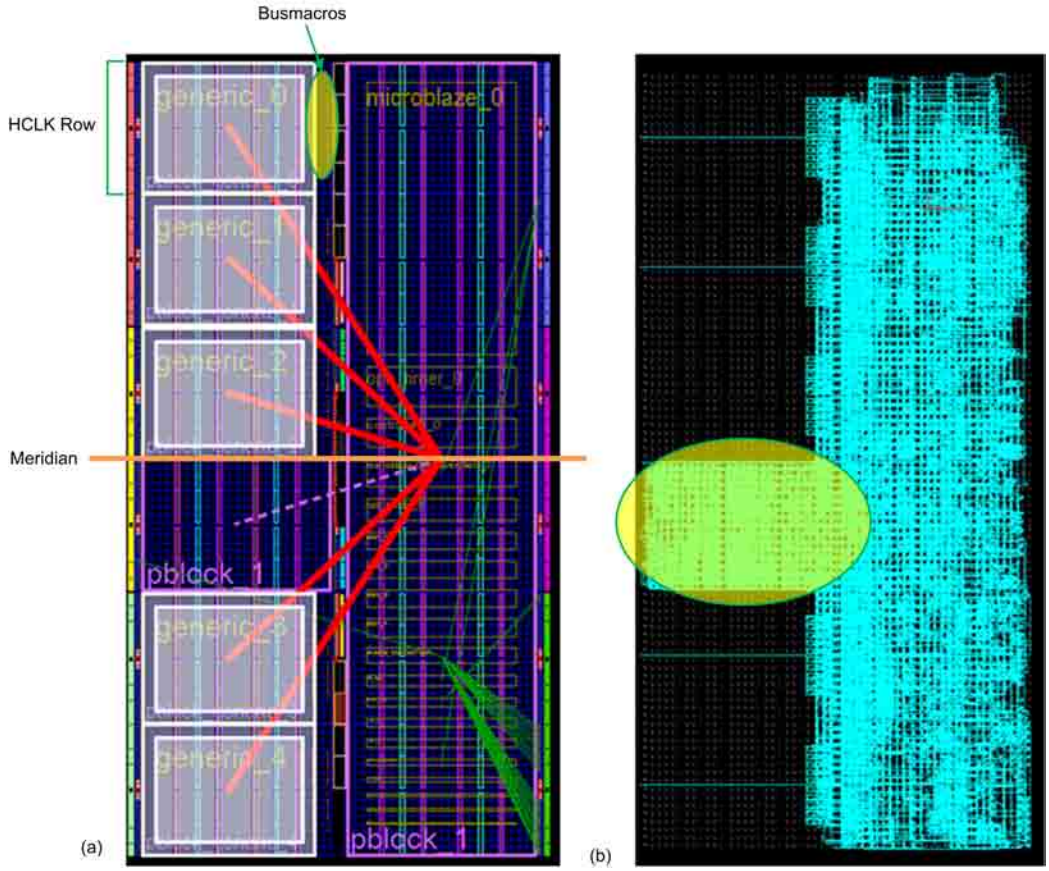

Fig. 4.1: (a) Shows the design floorplanned with the sockets highlighted in white, and the busmacros circled in yellow. (b) The the routed static design.

device (two PR regions may not occupy the same frame). A smaller PR region would not be large enough to hold the FSA PE, and it would waste BRAMs and DSPs (if they are partially covered by the PR region). Multiple smaller PEs, however, can be placed within a socket. This has the draw-back of decreasing the granularity of scaling and interconnections, since multiple PEs are reconfigured for each socket and there is only one connection to the controller for multiple PEs. In the test cases only one PE resides in a socket which resides in one clock region (fig. 4.1(a)). The floorplan is designed in Xilinx's PlanAhead software specifically configured for PDR (Appendix B).

4.2 Performance of the FSA and DSA on the PolySAF

All test cases where run on a Xilinx Virtex-4 SX35 based ML 402 board running at 100

MHz. Figure 4.2(a) shows the average number of cycles taken to complete one iteration of the Faddeev algorithm on the PolySAF for a varying number of sockets (1 to 5) configured as PEs for the FSA and when the input matrices **A, B, C, D** are of equal dimensions N=M=P. Figure 4.2(b) shows the number of cycles taken to complete one iteration of the DSA (low-pass and high-pass filtering of a vector with decimation) for a varying number of sockets (1 to 5) configured as PEs for the DSA and for an input vector (row or column of an image) of 16 elements and varying number of filter taps (1 to 16).

For the test case of the PolySAF in full FSA configuration, it was observed that from the perspective of the microprocessor, 45% of the time was spent controlling accelerated operations, 25% was spent doing non-linear operations, and 29% was spent transferring data to or from the co-processor. For the DSA mode of operation, 31% of the time was spent on data transfers and 69% of the time was spent on data computations on the accelerator, if the pre-loading of the image to the cache is counted.

4.3 Performance of the EKF and DWT (FPGA vs. PowerPC)

The results of test cases run on the FPGA are compared to software implementations

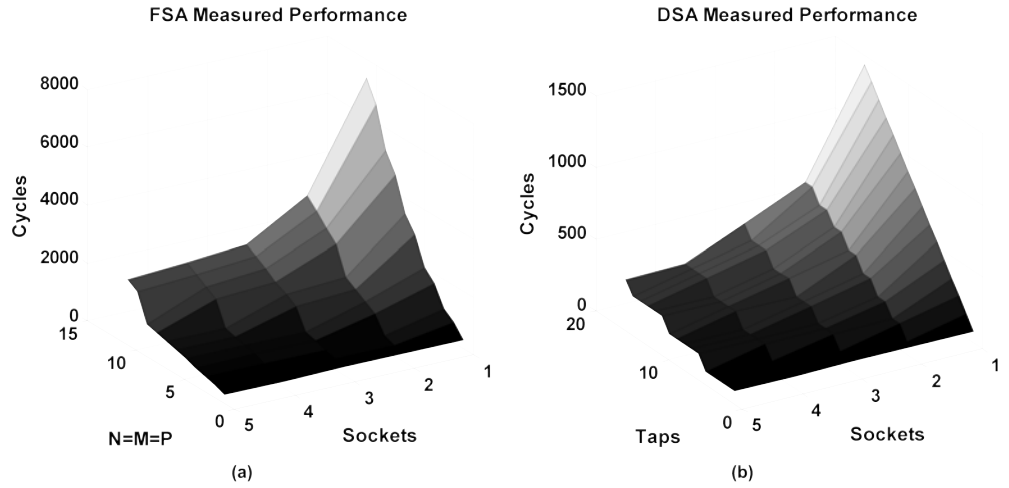

Fig. 4.2: (a) Trade-off between size of the input (1 to 16) number of sockets (1 to 5) for the FSA. (b) Trade-off between the number of filter taps (1 to 16) number of sockets (1 to 5) for in an input vector of 16 on the DSA.

of the same algorithms run on a Virtutech Simics PowerPC 750 simulator [53] running at 150 MHz (equivalent to the embedded RAD750 used in many space applications). The FPGA based design is 4.18x faster (fig. 4.3(a)) for the EKF algorithm, for an example of an autonomous UAV (unmanned air vehicle) described by Ronnback [11]. The parameters of the EKF test case are: number of states = 10, number of measurements = 9, number of control inputs = 6. The FPGA based design is 6.61x faster (fig. 4.3(b)) for the 2D DWT algorithm, where the matrix size = 64x64 and number of taps of the high-pass and low-pass filters is 4.

It is necessary to note that the simulator for the PowerPC 750 is overly optimistic because it does not model memory latencies or cache performance. Therefore, performance on an actual device is expected to be worse, giving the FPGA design an even better speedup than observed in fig. 4.3.

The performance of the pseudo-cache for the EKF test cases was 85% hit rate at the granularity of a word (32 bits) since a word is the smallest unit of data that can be replaced in the cache from the microprocessor's memory. However for the DWT, there were no pseudo-cache misses because the entire image was pre-loaded prior to access by the PolySAF and the filter operates on intermediate results stored in the pseduo-cache for the rest of the algorithm.

4.4 Analysis of PDR

There are several factors that affect the reconfiguration latency in this design, including: the size of the partial bitstream, time for relocation, location of the PR region, time for bitstream decompression, and external memory latency. The uncompressed partial bitstream (PB) for a socket configured as either a FSA-PE or a DSA-PE is 88KB, needing 88 BRAMs total. However, the RLE compressed PB for a FSA-PE is 59KB and DSA-PE is 39KB, needing only 49 BRAMs or 44% fewer BRAMs. Figure 4.4 provides a comparison of reconfiguration time for every different reconfiguration method provided by the Hybrid PDR method. Some of more important observations are summarized below, assuming the convention in table 4.1.

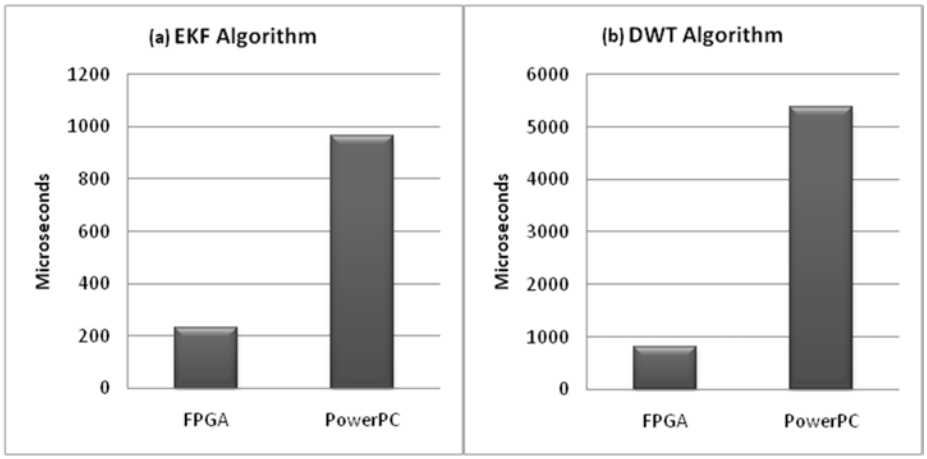

Fig. 4.3: (a)Comparison of the EKF run the FPGA and PowerPC for the test case defined by Ronnback [11]. (b) Comparison of the DWT run on the PowerPC for a test case of a 64x64 image and 4 taps.

- <B>reduces the reconfiguration latency by 86% compared to <F>. This is the fastest reconfiguration method, but requires significant use of valuable BRAMs (44 for every PE's PB).
- <B, C>reduces the number of BRAMs needed by a factor of 2 compared to <B>, but the decompression process (in software) increases the reconfiguration time by approximately 150%.
- There was negligible difference between <B>and <B, R>. However, <B, R>avoids the need for extra sets of BRAMs to store a separate PB for each socket.
- The performance of <B, C, R>was similar to <B, C>, but was significantly poorer for <B, C, M>. This is caused by the overhead spent bit-reversing each frame.
- Hence the primary Hybrid PDR case, <B, C, R>, performed about 2.7x better than the standard method of <F>.

As a caveat it takes 20 microseconds for one iteration of the FSA with 3 PEs and N=M=P=10, which means 2621 iterations of the FSA can pass in the time it takes to reconfigure one PE. This emphasizes the observation that PDR is not fast enough to be used in between operations of an algorithm, like between the update and predict stages of

Table 4.1: Symbols used for discussion of PDR.

Labels	Meaning
F →	PB stored on flash card.
C →	PB is compressed and decompression is performed by the MicroBlaze.
B →	PB is stored on BRAM
R →	PB is relocated with source and destination sockets on the same side of the meridian line (orange line in fig. 4.1).
M →	PB is relocated with source and destination sockets on different sides of the meridian line.

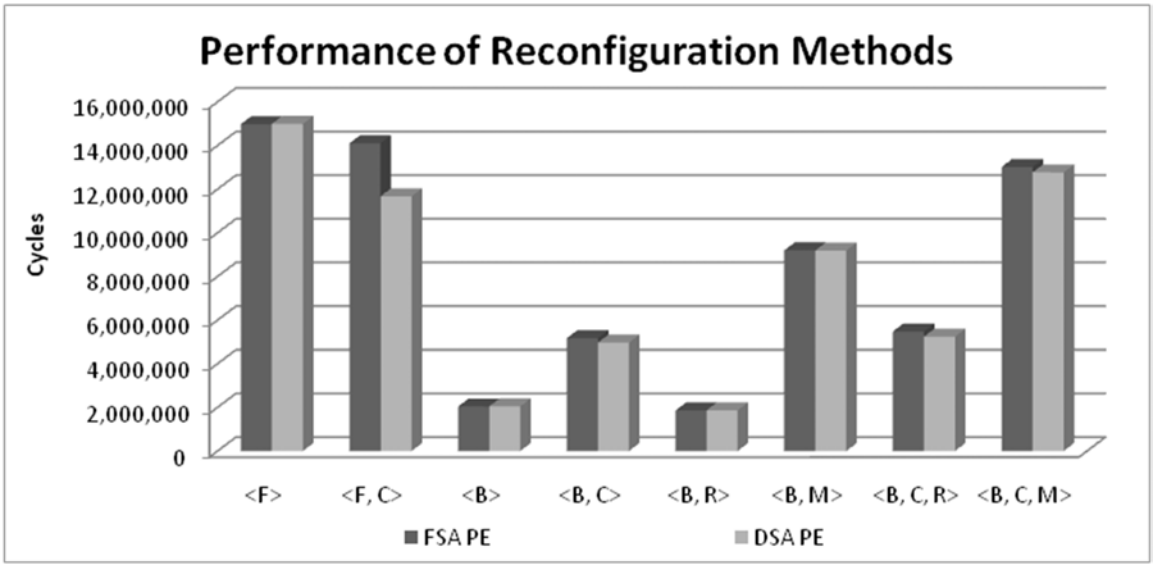

Fig. 4.4: Reconfiguration latencies using the notation in table 4.1.

the EKF. It can be useful, however, if used intermittently in a system, e.g. for infrequently scaling the hardware accelerator.

4.5 Area Analysis

Table 4.2 shows the device utilization of the DWT and Fadeev PEs. It can be observed that the Fadeev PE dominates the area and hence the size of the PR region is determined by this PE. There is significant area overhead incurred from partitioning the architecture into separate closed relocatable PRs on the FPGA. More PEs can be allocated to a static non-reconfigurable design, because the place and route algorithms are able to better utilize

the FPGA resources.

Table 4.3 indicates that when no partial reconfiguration was used, it is possible to fit 13 DWT PEs on the device or seven FSA PEs. Tests also reveled that it is possible to fit five static FSA PEs and five static DSA simultaneously on the same FPGA. This negates the advantages realized by Hybrid PDR in the two SA test case. However, it is possible to implement several more SAs on the PolySAF, beyond just the FSA and DSA. Hence, to fully realize the benefits gained from the flexibility of the PolySAF, several SAs should be implemented in a system.

Table 4.2: PE resource utilization.

PE	**Slices**	**DSPs**	**FIFOs**
DSA	344	8	3
FSA	1060	8	5

Table 4.3: Static PE device utilization.

Mode	**FSA PE**	**DSA PE**
Reconfigurable	0-5	0-5
Static FSA	7	0
Static DSA	0	13
Static FSA + DSA	5	5

Chapter 5

Conclusions and Future Work

5.1 Conclusions

A polymorphic systolic array framework (PolySAF) that works in conjunction with an embedded microprocessor on an FPGA, to allow for dynamic and complimentary scaling of acceleration levels of two simultaneous SA used by two different algorithms was presented. This design took advantage of recent advances in PDR technology as well as traditional design techniques such as SA and hardware-software co-design to obtain an efficient multi-application acceleration system. The flexible framework and modular software allowed this design to host a broad range of algorithms, including more complex applications in the area of aerospace embedded systems.

A case study showed that the FPGA implementation of the EKF and DWT algorithms outperforms an equivalent implementation on the RAD750, while still allowing the algorithms performance to scale dynamically. An area analysis, however, showed that the implementation of these two particular algorithms alone are not efficient compared to a static implementation on the FPGA. The advantage of this design is its flexibility to be applied to additional algorithms and on more sophisticated FPGAs. If three or more algorithms were implemented simultaneously, rather than just two, then this framework would provide an advantage over using static hardware accelerators. It was also shown in the case study that classic partial dynamic reconfiguration can be improved by caching bitstreams on-chip, which is made possible by integrating bitstream compression/decompression and bitstream relocation. Both of these methods (decompression and relocation) introduced additional overhead in the PDR process, which could be avoided if more on-chip memory were available.

Several special techniques had to be applied to make this design possible, and each technique has its trade-offs. Dynamic partial reconfiguration allowed regions of the hardware to be changed dynamically but also requires the boundaries of the partial regions to be defined at compile time, limiting FPGA resource utilization. Relocation of bitstreams requires only a single bitstream to be stored for each array but also increases the reconfiguration overhead and requires all static routing be excluded from partial regions. Compression allows for more partial regions to fit into the limited on-chip BRAMs but also creates overhead needed for decompression. Storing the bitstreams in BRAMs decreased the load time for reconfiguration but also consumed a large portion of the on-chip BRAM, limiting the number of PE that can be placed. Floating-point math increased the precision of the results but also vastly increased the area consumed by each PE. The systolic array accelerators were designed to allow scaling but this increased the logic overhead compared to a static design.

Since this design has been shown to outperform an existing state of the art spacecraft computer, it has the potential to improve the performance of future space missions by allowing the spacecraft to perform more tasks more quickly while still being able adapt to changing requirements.

5.2 Future Work

5.2.1 PolySAF

There are some notable limitations of the PolySAF. First the design only allows mapping of SAs that are linear, have a regular data flow, and at most 32 bits of data and 4 bits for control going in either direction. Each PE must be small enough to fit within a socket, yet it should utilize as much of the socket's PR region as possible. This framework is also most efficient when there is only one type of PE per SA mapped, since an additional bitstream is required for every type of PE. Since the number of PEs is assumed to change dynamically the SA should be scalable. It is possible to map most systolic arrays into a form that fits these constraints, but the resulting SA may not be efficient.

The pseudo-cache has only a single port, a dual-ported pseudo-cache would allow for

increased parallelism. DMA (Direct Memory Access) could also improve performance. Finally, the operation of the PolySAF is dependent on instructions and initial data being sent from the microprocessor. If the microprocessor can not provide instructions fast enough to keep the instruction buffer full, the performance of the PolySAF can be affected.

5.2.2 FSA

A major advantage of the FSA is that it does matrix inversion, addition, and multiplication on the same SA operating in the same way. This is also a disadvantage because simple operations, like addition, still require the same amount of time to complete as more complex applications, like matrix inversion. The simple approach of mapping all of the nonlinear equations in the EKF to software and all the linear equations to hardware may not be optimal. For example, small vector operations may run faster in software depending on the size. Additionally, some of the possibly nonlinear equations turn out to be linear in many cases, so they could potentially be accelerated. The software could be modified to automaticly recognize if a linear algebra operation would run faster in hardware or software and schedule it accordingly.

5.2.3 DSA

An major limitation of this design is related to scaling. The PE has only two registers to store the high-pass and low-pass weights. When the number of taps is greater than the number of PEs, the weights must be reset between each recursive run of the array. If these registers were replaced with a FIFO it would allow all of the weights needed by recursive runs to be pre-stored, so no refreshing of the weights would be required between runs.

5.2.4 Hybrid PDR

There many limitations and constraints with this Hybrid PDR method. Namely, PR region may not have static routes passing through it. Each PR region must partition a rigid portion of the device time. Each relocatable PR region must share the same types of components (slices, BRAMs, and DSPs) in the same relative locations. Some of the

limitations that could be improved are optimizing the compression algorithm for bitstreams, design a hardware module for faster decompression, design a faster ICAP wrapper, and improving frame bit-reversing.

References

[1] P. Pingree, J.-F. Blavier, G. Toon, and D. Bekker, "An fpga/soc approach to on-board data processing enabling new mars science with smart payloads," *IEEE Aerospace Conference*, pp. 1–12, Mar. 2007.

[2] N. Trawny, A. I. Mourikis, S. I. Roumeliotis, A. E. Johnson, J. F. Montgomery, A. Ansa, and L. H. Matthies, "Coupled vision and inertial navigation for pin-point landing," in *NASA Science and Technology Conference*, 2007.

[3] J. Nygards, P. Skoglar, M. Ulvklo, and T. Hgstrm, "Navigation aided image processing in uav surveillance: Preliminary results and design of an airborne experimental system," *Journal of Robotic Systems*, vol. 21, no. 2, pp. 63–72, 2004.

[4] A. Garzelli and F. Nencini, "Panchromatic sharpening of remote sensing images using a multiscale kalman filter," *Pattern Recognition*, vol. 40, no. 12, pp. 3568 – 3577, 2007 [Online]. Available: `http://www.sciencedirect.com/science/article/B6V14-4NS0KM1-1/2/1c7988a64bf69cb64fe99323494547e7`.

[5] "Deep impact fact sheet," National Aeronautics and Space Administration & Jet Propulsion Laboratory. Technical Report, 2004 [Online]. Available: `http://solarsystem.nasa.gov/deepimpact/mission/factsheet-color.pdf`.

[6] G. Welch and G. Bishop, *An Introduction to the Kalman Filter*. Chapel Hill, NC, USA: University of North Carolina at Chapel Hill, 1995.

[7] A. Bigdeli, M. Biglari-Abhari, Z. Salcic, and Y. T. Lai, "A new pipelined systolic array-based architecture for matrix inversion in fpgas with kalman filter case study," *EURASIP Journal on Advances in Signal Processing*, no. 1, pp. 75–75, 2006.

[8] F. Gaston and G. Irwin, "Systolic kalman filtering: an overview," *Control Theory and Applications, IEE Proceedings D*, vol. 137, no. 4, pp. 235–244, July 1990.

[9] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME Journal of Basic Engineering*, no. 82 (Series D), pp. 35–45, 1960 [Online]. Available: `http://www.cs.unc.edu/\~{}welch/kalman/media/pdf/Kalman1960.pdf`.

[10] C. Yukun, S. Xicai, and L. Zhigang, "Research on kalman-filter based multisensor data fusion," *Journal of Systems Engineering and Electronics*, vol. 18, no. 3, pp. 497 – 502, 2007 [Online]. Available: `http://www.sciencedirect.com/science/article/B82XK-4PYS2XV-B/2/f3388c3dbabec62c81be694197a61eb7`.

[11] S. Ronnback, "Development of an ins/gps navigation loop for uav," Master's thesis, Lulea University of Technology, 2000 [Online]. Available: `http://epubl.luth.se/1402-1617/2000/081/LTU-EX-00081-SE.pdf`.

[12] F. Orderud, "Comparison of kalman filter estimation approaches for state space models with nonlinear measurements," *SIMS 46th Conference on Simulation and Modeling*, 2005.

[13] 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.

[14] V. Bonato, E. Marques, and G. Constantinides, "A floating-point extended kalman filter implementation for autonomous mobile robots," *International Conference on Field Programmable Logic and Applications*, pp. 576–579, Aug. 2007.

[15] H. Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 1, pp. 37–46, Jan. 1982.

[16] M. Azimi-Sadjadi, T. Lu, and E. Nebot, "Parallel and sequential block kalman filtering and their implementations using systolic arrays," *IEEE Transactions on Signal Processing*, vol. 39, no. 1, pp. 137–147, Jan. 1991.

[17] W.-S. C. Henry Ker-Chang Chang, "Systolic array implementation of second-order kalman filter," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 28, no. 4, pp. 1128–1143, Oct. 1992.

[18] R. Barnes and A. Dasu, "Hardware/software co-designed extended kalman filter on an fpga," in *The International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2008.

[19] V. Bonato, R. Peron, D. Wolf, J. de Holanda, E. Marques, and J. Cardoso, "An fpga implementation for a kalman filter with application to mobile robotics," *SIES: International Symposium on Industrial Embedded Systems*, pp. 148–155, July 2007.

[20] S.-Y. Kung and J.-N. Hwang, "Systolic array designs for kalman filtering," *IEEE Transactions on Signal Processing*, vol. 39, no. 1, pp. 171–182, Jan. 1991.

[21] Z. Salcic and C.-R. Lee, "Fpga-based adaptive tracking estimation computer," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 37, no. 2, pp. 699–706, Apr. 2001.

[22] W.-T. Lin and D.-C. Chang, "The extended kalman filtering algorithm for carrier synchronization and the implementation," *ISCAS: Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 4034–4037, May 2006.

[23] S.-G. Chen, J.-C. Lee, and C.-C. Li, "Systolic implementation of kalman filter," *IEEE Asia-Pacific Conference on Circuits and Systems*, pp. 97–102, Dec. 1994.

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

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

[26] S. Syed and M. Bayoumi, "A scalable architecture for discrete wavelet transform," *CAMP: Proceedings of the Computer Architectures for Machine Perception*, pp. 44–50, Sept. 1995.

[27] Z. Li and S. Hauck, "Configuration compression for virtex fpgas," *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 147–159, 2001.

[28] N. Maria, A. Guerin-Dugue, and F. Blayo, "1d and 2d systolic implementations for radial basis function networks," *Proceedings of the Fourth International Conference on Microelectronics for Neural Networks and Fuzzy Systems*, pp. 34–45, Sept. 1994.

[29] A. Ahmedsaid, A. Amira, and A. Bouridane, "Efficient systolic array for singular value and eigenvalue decomposition," *MWSCAS: Proceedings of the 46th IEEE International Midwest Symposium on Circuits and Systems*, vol. 2, pp. 835–838, Dec. 2003.

[30] S. Hong, W. Moon, H.-Y. Paik, and G.-H. Choi, "Data fusion of multiple polarimetric sar images using discrete wavelet transform (dwt)," *IEEE International Geoscience and Remote Sensing Symposium*, vol. 6, pp. 3323–3325, 2002.

[31] D. Santa-Cruz and T. Ebrahimi, "An analytical study of jpeg 2000 functionalities," *Proceedings of the International Conference on Image Processing*, vol. 2, pp. 49–52, 2000.

[32] G. Zhang, Y.-L. Yang, and Z.-Q. Wang, "Adaptive fault tolerant control system design for nonlinear systems with actuator failures," *Proceedings of the International Conference on Machine Learning and Cybernetics*, vol. 1, pp. 499–505, Aug. 2005.

[33] Y. Zhang, X. Chen, M. Chen, and Y. Geng, "A novel approach for satellite attitude reconfigurable fault-tolerant control," *2nd IEEE Conference on Industrial Electronics and Applications*, pp. 2612–2616, May 2007.

[34] G. Ducard and H. Geering, "A reconfigurable flight control system based on the emmae method," *American Control Conference*, p. 6 pp., June 2006.

[35] E. Monmasson, B. Robyns, E. Mendes, and B. De Fornel, "Dynamic reconfiguration of control and estimation algorithms for induction motor drives," *ISIE: Proceedings of the IEEE International Symposium on Industrial Electronics*, vol. 3, pp. 828–833, 2002.

[36] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, pp. 642–649, 2001.

[37] C. Kao, "Benefits of partial reconfiguration," *Xcell*, pp. 65–67, 2005.

[38] M. J. Wirthlin and B. L. Hutchings, "Improving functional density using run-time circuit reconfiguration," *IEEE Trans. Very Large Scale Integrgrated Systems*, vol. 6, no. 2, pp. 247–256, 1998.

[39] R. Barnes, A. Dasu, J. Carver, and R. Kallam, "Dynamically reconfigurable systolic array accelerators: A case study with ekf and dwt algorithms," submited to the Institution of Engineering and Technology (IET) Computers & Digital Techniques. In Review.

[40] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Invited paper: Enhanced architectures, design methodologies and cad tools for dynamic reconfiguration of xilinx fpgas," *International Conference on Field Programmable Logic and Applications*, pp. 1–6, Aug. 2006.

[41] Xilinx, *Virtex-4 Configuration Guide*, Xilinx, Inc., 2007 [Online]. Available: `http://www.xilinx.com/support/documentation/user_guides/ug071.pdf`.

[42] S. Hauck and W. D. Wilson, "Runlength compression techniques for fpga configurations," in *FCCM: Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, p. 286. Washington, DC, USA: IEEE Computer Society, 1999.

[43] A. Dandalis and V. Prasanna, "Configuration compression for fpga-based embedded systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 12, pp. 1394–1398, Dec. 2005.

[44] J. H. Pan, T. Mitra, and W.-F. Wong, "Configuration bitstream compression for dynamically reconfigurable fpgas," *IEEE/ACM International Conference on Computer Aided Design*, pp. 766–773, Nov. 2004.

[45] M. Martina, G. Masera, A. Molino, F. Vacca, L. Sterpone, and M. Violante, "A new approach to compress the configuration information of programmable devices," *Proceedings of the Design, Automation and Test in Europe Conference*, vol. 2, p. 4 pp., Mar. 2006.

[46] J. Carver, R. N. Pittman, and A. Forin, "Relocation and automatic floor-planning of fpga partial configuration bit-streams," Microsoft Research Technical Report, WA, Aug. 2008.

[47] T. Becker, W. Luk, and P. Cheung, "Enhancing relocatability of partial bitstreams for run-time reconfiguration," *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 35–44, April 2007.

[48] D. P. Montminy, R. O. Baldwin, P. D. Williams, and B. E. Mullins, "Using relocatable bitstreams for fault tolerance," in *AHS: Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems*, pp. 701–708. Washington, DC, USA: IEEE Computer Society, 2007.

[49] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, "Modular dynamic reconfiguration in virtex fpgas," *Computers and Digital Techniques, IEE Proceedings*, vol. 153, no. 3, pp. 157–164, May 2006.

[50] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert, "Replica: A bitstream manipulation filter for module relocation in partial reconfigurable systems," *Proceedings of the 19th*

IEEE International Parallel and Distributed Processing Symposium, pp. 151b–151b, Apr. 2005.

[51] F. Ferrandi, M. Novati, M. Morandi, M. D. Santambrogio, and D. Sciuto, "Dynamic reconfiguration: Core relocation via partial bitstreams filtering with minimal overhead," *International Symposium on System-on-Chip*, pp. 1–4, Nov. 2006.

[52] Xilinx, *MicroBlaze Processor Reference Guide: Embedded Development Kit EDK 9.1i*, Xilinx, Inc., 2007 [Online]. Available: `URL:http://www.xilinx.com/support/documentation/sw_manuals/edk91i_mb_ref_guide.pdf`.

[53] Virtutech, *Simics/PM-PPC Target Guide*, Virtutech AB, Sweden, 2006 [Online]. Available: `http://www.cs.sfu.ca/~fedorova/Tech/simics-guides-3.0.26/simics-target-guide-pmppc.pdf`.

Appendices

Appendix A

Code Snippets

Included here are two small snippets of code from the project. First is the C-software code used to execute the schur complement. By selecting the correct inputs, this function serves as the root function for executing all accelerated operations on the FSA. Next is the Verilog RTL code which performs the primary calculations and logic in the boundary cell of the FSA.

A.1 Schur Complement

```
void schur(Matrix* E, Matrix* A, Matrix* B, Matrix* C, Matrix* D,
                                        int N,int M,int P,int R, int T)
{
        //If matrices are not already in cache, send them to the cache
        co_write(A);
        co_write(B);
        co_write(C);
        co_write(D);

        //Mark matrix E as dirty,
        //i.e. it must be sent from cache if used by microprocessor
        co_dirty(E);

        int i,j;
        int NM=N+M, NP=N+P, R2=R*2, MP=M+P;      //pre-compute some values
        int consumed = 0,out_box=0,in_box=0;

        while(N>0)
        {
                //SETUP BOXES
                in_box = 0;
                sbox(DDU,in_box);          //configures switch box
                if(N>R) //data width > resources
                {
                        if(N>=R2)
                        {
                                out_box = 0; //start and end at PE 0
                                sbox(DRU,out_box);
                        }
```

```
        else
        {
                out_box = R2-N;
                sbox(DRL, out_box);
        }
        sbox(LDD,R); //last switchbox loops back
        consumed = R2-out_box; //elements to be consumed
}
else //data width < resources
{
        out_box = N;
        consumed = out_box;
        sbox(DLD, out_box);
}
//LOAD ARRAY
NM=N+M; //pre-compute
for(i=0; i<N; i++)
{
        if(i==0) //write first row
        {
                array_write(SET_A, in_box, (N), A->map[i]);
                array_write(SET_B, in_box, (M), B->map[i]);
        }
        //write rows from A/B
        else
        {
                array_write(ND_A, in_box, (N), A->map[i]);
                array_write(ND_B, in_box, (M), B->map[i]);
        }
}
for(i=0; i<P; i++)
{
        //last row
        if(i==P-1)
        {
                if(T)
                        array_twrite(UNSET_C, in_box,
                        (N), i, C->map[P-1]);
                else
                        array_write(UNSET_C, in_box,
                        (N), C->map[i]);
        array_write(UNSET_D, in_box, (M), D->map[i]);
        }
        //write rows from C/D
        else
        {
                if(T)
                        array_twrite(ND_C, in_box,
                        (N), i, C->map[P-1]);
                else
                        array_write(ND_C, in_box,
                        (N), C->map[i]);
```

```
                                        array_write(ND_D,in_box,(M),D->map[i]);
                                }
                        }
                        N -= consumed;  //consume
                        NP=N+P; //pre-compute
                        NM=N+M;
                        for(i=0; i<(NP); i++)
                        {
                                array_read(out_box,(NM),E->map[i]);
                        }
                        sbox(RDL,out_box);
                }
}
```

A.2 Boundary Cell

```
module boundary_core2(clk,x,m,s,pin,pout,set,p_rdy,nd,rfd,y_rdy,y,rst);

//PORTS
        input [31:0] x, pin;
        input nd,m,clk,set,rst;

        output reg [31:0] y;
        output reg y_rdy=0, s=0, p_rdy;
        output rfd, pout;

//Registers
        reg rfd_reg=1;
        reg [31:0] x_reg;
        reg m_reg = 0, nd_reg = 0, set_reg=0;

//Wires
        wire s0;
        wire [31:0] pout;

        //*****MATH******
        wire gte_result;
        gte_i gte (
        .a({1'b0,x[30:0]}), // Bus [31 : 0]
        .b({1'b0,pin[30:0]}), // Bus [31 : 0]
        .clk(clk),
        .result(gte_result)// Bus [31 : 0]
        );

        reg [31:0] div_a, div_b;
        reg div_nd=0;
        wire [31:0] div_result;
        div_i div (
        .a(div_a), // Bus [31 : 0]
        .b(div_b), // Bus [31 : 0]
```

```
        .nd(div_nd),
        .rfd(div_rfd),
        .clk(clk),
        .result(div_result), // Bus [31 : 0]
        .rdy(div_rdy),
        .err(div_err),
        .rst(rst)
        );

//wire assignments
        assign s0 = (gte_result==1 && m_reg==0) || set_reg;
        assign pout = s0 ? x_reg : pin;

//synchronous processing
   always@(posedge clk) begin
//------------INPUT PROCESSING-------
                p_rdy <= nd;
                nd_reg <= nd;
                set_reg <= set;
                if(nd) begin
                        x_reg <= x;
                        m_reg <= m;
                end
                set_reg <= set;
                div_a <= s0 ? pin : x_reg;
                div_b <= pout==0 ? 32'h3f800000 : pout;
                div_nd <= nd_reg;


//------------OUTPUT PROCESSING-------
                y <= {~div_result[31],div_result[30:0]};
                if(nd_reg) s <= s0;
                y_rdy <= div_rdy;

        end
//--------RFD PROCESSNG-----------
        always @(posedge clk) begin
                if(nd) rfd_reg <= 0;
                else if(div_rdy) rfd_reg <= 1;
        end

        assign rfd = rfd_reg && !nd;

endmodule
```

Appendix B

Design Flow

B.1 EDK Flow

This design applied the EAPR design flow from Xilinx. Structuring the design flow correctly is an important part of allowing this design to work correctly. In Xilinx Embedded Development Kit (EDK) the components of the system are connected and laid out. Here the features of the Microblaze are defined such as including an internal floating-point unit, the size of the Microblaze memory, etc. System features like the clock frequency, I/O pins, etc., are also defined here. The Microblaze is connected to the controller through an FSL bus. A parameter is set that defines the number of connected sockets, and then the controller is connected directly to each swtichbox. Each switchbox is connected to the neighboring sockets. It does not matter what is contained within these sockets as long as they have the correct I/O ports, unless system is to be simulated. For simulation the busmacros are switched to behavioral model equivalents by changing a parameter. This allows the system to be simulated or even placed on the FPGA, but it does not allow PDR. In EDK a netlist can be generated for the design that can be imported into PlanAhead.

B.2 PlanAhead Flow

After the design is imported into PlanAhead each socket is placed as a partial region on the FPGA such that each region is the same dimension and covers the same resources as the other partial regions. Then each busmacro is placed on the edge of the appropriate partial region. The designer must be careful not to create situations that force routing across any partial regions. The designer can now import multiple modules for each partial regions. Since the modules are relocatable, only one module of each type needs to imported, thus

reducing the compile time. These modules are designed and simulated separately in ISE (Integrated Software Environment). A netlist of the module is generated in ISE with no I/O pins, global clocks disabled, and with a module name that matches the targeted partial region in PlanAhead. This can be accomplished with a simple wrapper. PlanAhead will then place and route the static and partial regions separately. The designer may be required to resolve area constraints at this point. PlanAhead will generate a system bitstream, a partial bitstream for each PR region module, and blank bitstreams for each PR region. To be compatible with the relocation software, the header of each bitstream is removed. Then each bitstream is compressed using the RLE compressor. The resulting bitstreams are stored a Compact Flash card and inserted into the FPGA board.

B.3 Flow Automation

This modified EAPR flow is automated somewhat by a custom script that offers the following features:

- Update PlanAhead netlist with netlist from EDK.
- Automatically remove headers, compress then copy partial bitstreams to the compact flash card.
- Compile the Microblaze application.
- Combine the static bitstream with the compiled application.
- Flash FPGA with combined bitstream.
- Copy combined bitstream in appropriate format to the compact flash card.

It is entirely possible to further automate the design flow by adding additional features to this script.