Stony Brook University

OFFICIAL COPY

The official electronic file of this thesis or dissertation is maintained by the University Libraries on behalf of The Graduate School at Stony Brook University.

© All Rights Reserved by Author.

Design and Implementation of Reconfigurable Hardware for Real-Time Particle Filtering.

A Dissertation Presented

by

Akshay Athalye

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Electrical Engineering

Stony Brook University

August 2007

Stony Brook University

The Graduate School

Akshay Athalye

We, the dissertation committee for the above candidate for the
Doctor of Philosophy degree,
hereby recommend acceptance of this dissertation.

Petar M. Djurić, Advisor of Dissertation
Professor, Department of Electrical and Computer Engineering

John Murray, Chairperson of Defense
Professor, Department of Electrical and Computer Engineering

Sangjin Hong, Assistant Professor,
Department of Electrical and Computer Engineering

Jacob Sharony, Adjunct Professor,
Department of Electrical and Computer Engineering

Hui Zhang, Associate Professor,
Department of Mechanical Engineering

This dissertation is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

Design and Implementation of Reconfigurable Hardware for Real-Time Particle Filtering.

by

Akshay Athalye

Doctor of Philosophy

in

Electrical Engineering

Stony Brook University

2007

Particle Filtering is a Monte Carlo sampling based signal processing technique that is applied to systems described using dynamic state space models. For models that are non-linear and non-Gaussian, traditional filtering techniques fail in terms of filter performance. Particle filters can handle nonlinear and non-Gaussian systems much more efficiently than such methods. As a result, these filters have gained immense popularity in recent years. However, their high computational intensity, which is widely recognized in literature, makes them unsuitable for implementation on sequential platforms like DSPs. This fact, along with the absence of dedicated hardware for particle filtering has prevented their use in real time systems despite their suitability in terms of filter performance. The goal of this dissertation is to address this gap and develop hardware suitable to real time particle filtering. This research has progressed through the steps of algorithmic optimization, architecture development and physical implementation, and has produced the first FPGA prototype for a particle filter.

Often, real world systems require multiple models for accurate and complete description. A class of particle filters known as Multiple Model Particle Filters are applied to such systems. Starting from the hardware developed for the basic particle filter, we propose a parallel, distributed architecture for implementation of a novel multiple model particle filtering algorithm. The distributed processing units of the architecture interact using a data ex-

change protocol with low interconnect requirement and no communication bottleneck. This high speed architecture with its immense scalability is well suited to practical problems that require a intensive particle filtering, incorporate a large number of models and have real time processing requriements. The proposed architecture implemented on an FPGA platform and applied to a practical problem, results in a speedup of upto 100 times over a DSP implementation.

Flexibility of particle filters is another of their widely recognized assets. Within a general framework, the particle filter can be applied to a wide range of problems by simply modifiying certain filtering parameters. We exploit the concept of hardware reconfiguration to develop reconfigurable architectures, whereby the same particle filtering device can be used for different problems by simply specifying a set of parameters. We use a novel buffer controller based design methodology to develop a reconfigurable particle filtering hardware that can be easily tuned to the problem at hand. Run time reconfiguration for implementation of multiple model particle filters with dynamically changing model sets is also explored. For each of the hardware architecture proposed, an FPGA based evaluation of speed and resource requirement is performed and the overall improvements over a sequential DSP based implementation of the corresponding algorithm are analyzed.

With these contributions, this dissertation takes a significant step in enabling the application of particle filters to practical systems requiring real time processing.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my advisor Prof. Petar M. Djurić for his training and guidance throughout my PhD program. I would also like to thank my co-advisor Prof. Sangjin Hong for setting an excellent example through his own hard work and dedication. All my family and friends both at Stony Brook and elsewhere deserve special gratitude for making my time here pleasant and enjoyable.

Finally, I would like to thank my committee members Prof. John Murray and Dr. Jacob Sharony for taking the time to review this dissertation and be on my committee.

Chapter 1

Introduction

1.1 Motivation

Particle Filters are a class of Sequential Monte Carlo methods that have gained immense popularity in solving complex signal processing problems over the past decade. These filters are applied to systems that are described by dynamic state space models. The well known Kalman Filter is the optimal solution in cases where the model is linear and Gaussian. However, all traditional filtering methods are severely hampered by nonlinearity, non-Gaussianity in the model. It is for such systems that particle filters greatly outperform traditional approaches such as Extended Kalman Filtering, Unscented Kalman filtering and Grid based methods. Since nonlinear and non-Gaussian systems are frequently encountered in practice, particle filters have attracted a great deal of attention from researchers and practitioners in the recent past. Moreover, due to their recursive nature, particle filters can be used for on-line sequential processing wherein filter estimate is sequentially updates as data becomes available.

However, particle filters are computationally highly intensive. Hence when implemented on sequential processors like DSPs, they cannot be applied to real time systems where speed is of essence. Most of the research on particle filtering to date has focused on their theory and on development of better variants of the basic particle filter. There has been vary little research into the development of dedicated hardware for particle filters. This dissertation intends to bridge this gap. When this effort was started, to the best of our knowledge, it was

the first of its kind in addressing this problem. Since then a few other efforts have started with the goal of developing hardware for particle filters.

Several practical problems, especially that of maneuvering target tracking require multiple models to cover system dynamics. The multiple model particle filter is an attractive solution for such cases since it makes interaction between the models easier and is not affected by nonlinearity and non-Gaussianity. With increasing number of models, the computational intensity of the filter increases even further and the execution becomes even slower on a sequential platform. On a DSP, for instance, the execution time scales exponentially with increasing number of models and the filter becomes infeasible for practical applications which frequently require several models in addition to real-time execution speed. Hardware that is capable of executing multiple model particle filtering in real time can be applied to a host of target tracking problems.

Another asset of particle filters is their flexibility. Within a consistent general framework, they can be applied to a wide range of different models irrespective of constraints such as linearity and Gaussianity. This property can be exploited at the hardware level by using the concept of reconfigurability. This allows a single particle filtering device to be used for a variety of problems in addition to allowing adaptation of filtering parameters during processing.

1.2 Contributions

The traditional particle filtering algorithm is not suited to real time hardware implementation. The first contribution of this dissertation is an in-depth analysis of the basic particle filtering algorithm from the point of view of hardware implementation. Modifications of the basic algorithm are then proposed from the point of view making the filter more suitable to concurrent spatial hardware. Architectures are developed for the resulting algorithm which allow for fully pipelined execution and minimize memory requirement using elegant memory access schemes. The architecture has led to the *first* hardware prototype of the particle filter on an FPGA platform. This prototype, applied to the bearings only tracking problem, shows a 50 times speedup as compared to the corresponding implementation on a sequential processor.

The modified algorithm and the architecture developed for the particle filter was then extended and adapted to multiple model particle filters. The architecture consists of distributed processing elements and a central unit controlling the communication and interaction between the various models used in the filtering. An overall communication scheme was developed based on a modified distributed resampling algorithm that allows all inter-block data exchange to be implemented with a single bus without introducing a communication bottleneck. This makes the architecture highly scalable with respect to the number of models and number of particles processed per model. An FPGA evaluation of the architecture for a practical tracking application is presented.

During the course of this work, another class of particle filters viz. Gaussian Particle Filters (GPFs) was explored from an implementation viewpoint. The GPF algorithm was modified such that it can be implemented in hardware without storing particles in memories between iterations. This makes the GPFs very attractive in applications requiring a large number of particles where the available memory size restricts the use of standard particle filters. A reconfigurable/parameterizable particle filtering architecture was developed which incorporates the ability to perform both standard particle filtering as well as Gaussian particle filtering on the same hardware by simply changing a small set of parameters. This architecture also allows for changing parameters of each filter in between iterations.

Finally, a design methodology was proposed for incorporating dynamically changing model sets in the multiple model particle filters implemented on a Xilinx FPGA platform. This represents a hardware realization of the Variable Structure Multiple Model particle filters. The methodology was evaluated for a simple particle filter realization, but will be enhanced and extended to a practical application in the future.

1.3 Organization of the Dissertation

The reset of this dissertation is organized as follows.

Chapter 2 lays down the foundation with a brief description of the theory of particle filters. The basic algorithms for the Sampling Importance Resampling Filter (SIRF) and the Gaussian Particle Filter (GPF) are explained along with their application to a the bearings

only tracking problem.

Chapter 3 describes the algorithmic analysis and modifications of the particle filters from an implementation perspective. The advantages of the modified algorithms in terms of resource requirement and speed are analyzed. The resulting algorithms are evaluated on a DSP platform that serves as a performance benchmark for the eventual hardware implementation.

Chapter 4 presents the development of a hardware architecture and various memory schemes for the SIRF. The implementation of Gaussian random number generators and other model specific mathematical computation blocks is also explained. This chapter includes fixed point analysis of the particle filter applied to the bearings only tracking problem. This analysis forms the basis of the fixed point scheme used in the implementation.

A design that can be reconfigured for two different types of particle filters viz. the SIRF and the GPF is presented in Chapter 5. This design also allows for parameterizing individual filter realization by specifying a small set of parameters.

A distributed architecture and communication scheme for multiple model particle filters is presented in Chpater 6. The architecture presented in chapter 4 forms the basic building block of the multiple model particle filter. Further, we explore a methodology that allows for incorporation of dynamically changing model sets in the multiple model particle filter implemented in a Xilinx FPGA platform.

Chapter 7 concludes the dissertation with a summary of our contributions and enlists future directions arising from this research effort.

The published works resulting from this effort can be found in [8, 6, 5, 46, 19, 9, 7, 10, 69, 18]

Chapter 2

Theory of Particle Filters

2.1 Dynamic State Space Models

Particle Filters are applied to systems that are formulated as dynamic state space models [42]. Many real world systems related to signal processing, image processing, communications and biology can be described by these models [40]. DSS models include a state equation that describes the evolution of the state of interest $\{\mathbf{x}_n; n \in \mathbb{N}\}$, $\mathbf{x} \in \mathbb{R}^{n_x}$ with time. This equation along with the initial distribution of the state $p(\mathbf{x}_0)$ describes the prior knowledge of the hidden Markov state process which is incorporated in the distribution $p(\mathbf{x}_n|\mathbf{x}_{n-1})$. The other equation in the DSS model, called the observation or measurement equation, describes the observations $\{\mathbf{y}_n,\ n \in \mathbb{N}\}, \mathbf{y} \in \mathbb{R}^{n_y}$ as a function of the state. The likelihood of the observations, $p(\mathbf{y}_n|\mathbf{x}_n)$ can be deduced from this equation. Mathematically, the DSS model is written as follows:

$$\mathbf{x}_n = f_n(\mathbf{x}_{n-1}, \mathbf{u}_n) \tag{2.1}$$

$$\mathbf{y}_n = h_n(\mathbf{x}_n, \mathbf{v}_n) \tag{2.2}$$

where f_n and h_n are possibly nonlinear state and observation equations respectively, and $\mathbf{u}_n$ and $\mathbf{v}_n$ are random noise vectors. The sequence of states and observations up to time instant n are written as $\mathbf{x}_{0:n}$ and $\mathbf{y}_{1:n}$ respectively. In the Bayesian context, the complete state information is incorporated in its posterior distribution $p(\mathbf{x}_{0:n}|\mathbf{y}_{1:n})$. In most applications,

the data (observations) arrive sequentially and hence it is necessary to update the posterior as the observations become available. Thus the goal in most real time applications is to estimate *recursively* in time the state $\mathbf{x}_n \forall n$ and possibly some functions of the state. These estimates are based on the posterior density and are given by expectations of the form [83]

$$I(g(\mathbf{x}_{0:n})) = \int g(\mathbf{x}_{0:n}) p(\mathbf{x}_{0:n}|\mathbf{y}_{1:n}) d\mathbf{x}_{0:n} \tag{2.3}$$

2.2 Analytical Solution to DSS Models

In sequential signal processing, the goal is to recursively update the state posterior as observations become available. Using Bayes theorem along with the Markovian assumption of the state evolution and the conditional independence of the observations given the state, we can decompose the posterior as

$$\begin{aligned} p(\mathbf{x}_{0:n}|\mathbf{y}_{1:n}) &= \frac{p(\mathbf{y}_{1:n}|\mathbf{x}_{0:n})p(\mathbf{x}_{0:n})}{p(\mathbf{y}_{1:n})} \\ &= \frac{p(\mathbf{y}_n|\mathbf{x}_{0:n}, \mathbf{y}_{1:n-1})p(\mathbf{x}_{0:n}|\mathbf{y}_{1:n-1})}{p(\mathbf{y}_n|\mathbf{y}_{1:n-1})} \\ &= \frac{p(\mathbf{y}_n|\mathbf{x}_n)p(\mathbf{x}_n|\mathbf{x}_{n-1})p(\mathbf{x}_{0:n-1}|\mathbf{y}_{1:n-1})}{p(\mathbf{y}_n|\mathbf{y}_{1:n-1})} \end{aligned} \tag{2.4}$$

The recursion can be clearly seen in (2.4), where $p(\mathbf{x}_{0:n}|\mathbf{y}_{1:n})$ is determined from $p(\mathbf{x}_{0:n-1}|\mathbf{y}_{1:n-1})$. The normalizing constant in (2.4) can be written using the Chapman-Kolmogorov equation as

$$p(\mathbf{y}_n|\mathbf{y}_{1:n-1}) = \int p(\mathbf{y}_n|\mathbf{x}_n)p(\mathbf{x}_n|\mathbf{y}_{1:n-1})d\mathbf{x}_n \tag{2.5}$$

The predictive density in the RHS of (2.5) can be obtained from the prior pdf of the state as follows

$$p(\mathbf{x}_n|\mathbf{y}_{1:n-1}) = \int p(\mathbf{x}_n|\mathbf{x}_{n-1})p(\mathbf{x}_{n-1}|\mathbf{y}_{1:n-1})d\mathbf{x}_{n-1}. \tag{2.6}$$

Thus, solving (2.4) requires evaluation of the following expression

$$p(\mathbf{y}_n|\mathbf{y}_{1:n-1}) = \int\int p(\mathbf{y}_n|\mathbf{x}_n)p(\mathbf{x}_n|\mathbf{y}_{1:n-1})p(\mathbf{x}_n|\mathbf{x}_{n-1})p(\mathbf{x}_{n-1}|\mathbf{y}_{1:n-1})d\mathbf{x}_n d\mathbf{x}_{n-1} \tag{2.7}$$

Frequently, many problems may also require prediction or smoothing apart from filtering. The prediction density can be expressed using the following recursion

$$p(\mathbf{x}_{n+l} \mid \mathbf{y}_{1:n}) = \int p(\mathbf{x}_{n+l} \mid \mathbf{x}_{n+l-1})p(\mathbf{x}_{n+l-1}|\mathbf{y}_n)d\mathbf{x}_{n+l-1}, \tag{2.8}$$

where $l > 1$.

Similarly, smoothing can be done recursively backward in time using the formula

$$p(\mathbf{x}_n \mid \mathbf{y}_{1:N}) = \int p(\mathbf{x}_{n+1}|\mathbf{y}_{1:N})\frac{p(\mathbf{x}_n|\mathbf{y}_{1:n})p(\mathbf{x}_{n+1} \mid \mathbf{x}_n)}{p(\mathbf{x}_{n+1}|\mathbf{y}_{1:n})}d\mathbf{x}_{n+1}, \tag{2.9}$$

where $n > N$.

As can be seen from (2.7),(2.8) and (2.9), the analytical solutions to the desired densities involve high dimensional integrals. Closed form solutions to these integrals are possible only in very specific cases. The simplest of these is the linear model with Gaussian noise, where the analytical solution of the filtering equation results in the Kalman Filter [54, 3, 43, 44] which is the optimal solution in this case. Many suboptimal methods have been proposed as a result of large amount of research for models that are nonlinear and non-Gaussian. Most popular among these methods is the Extended Kalman Filter [3, 51, 43, 57] which uses local linearization of the model around the previous estimate. This filter suffers from lack of robustness and poor performance in terms of accuracy of estimate[38, 99]. Other approaches include Approximate Grid Based Methods [76, 88], [89] Gibbs sampling and the Metropolis Hastings scheme.

2.3 Concept of Particle Filtering

Over the past few years a set of simulation based methods known as Sequential Monte Carlo (SMC) methods have become immensely popular in solving problems involving severe non-linearities and non-Gaussian noise. Although these methods were being researched since the late 1960's,([41, 1]) lack of computational power prevented their growth. However, with increase in computational power over the past decade, these methods have re-emerged and have grown to become among the most popular methods for nonlinear signal processing. Particle Filters fall under this broad category of SMC methods. They are simulation based methods and are known variously as bootstrap filter [38], condensation algorithm [73], interacting particle approximations [80] and survival of the fittest [55].

Particle filters represent (approximate) the desired densities (in our case the posterior) using a random measure composed of a set of samples or particle drawn from the space of the unknown state along with associated weights. This random measure approximating the posterior $p(\mathbf{x}_{0:n}|\mathbf{y}_{1:n})$, is written as $\left\{\mathbf{x}_{0:n}^{(m)}, w_n^{(m)}\right\}_{m=1}^{M}$, where $\mathbf{x}_{0:n}^{(m)}$ represents the m^{th} particle (trajectory), $w_n^{(m)}$ is its associated weight, and M is the total number of particles used The weights are normalized such that $\Sigma_{m=1}^{M} w_n^{(m)} = 1$. This random measure is used to approximate the posterior as

$$p(\mathbf{x}_{0:n}|\mathbf{y}_{1:n}) \approx \sum_{m=1}^{M} \delta(\mathbf{x}_{0:n} - \mathbf{x}_{0:n}^{(m)}) w_n^{(m)} \tag{2.10}$$

.

Using this approximation, any function g of the state defined by (2.3) can be estimated as follows:

$$\widehat{I}(g(\mathbf{x}_{0:n})) = \sum_{m=1}^{M} g(\mathbf{x}_{0:n}^{(m)}) w_n^{(m)} \tag{2.11}$$

2.3.1 Importance Sampling

The density from which the samples or particles are drawn greatly influences the performance of the particle filter. It can be clearly seen that the best performance will be obtained if

these particles were drawn from the posterior $p(\mathbf{x}_{0:n}|\mathbf{y}_{1:n})$ itself. In this case the weight of each of the M particles will be $w_n^{(m)} = \frac{1}{M} \forall m$. However, in nontrivial problems, it is not possible to directly draw samples from the posterior. Hence, one resorts to the technique of Importance Sampling (IS) which is fundamental to most SMC methods. Importance sampling is introduced in [35], and further described in the context of SMC methods in [30, 29, 4]. According to this technique, samples are drawn from another density (since drawing from the posterior is impossible), known as the Importance Function or Proposal Density. This importance function is represented as $\pi(\mathbf{x})$, and has the same support as the posterior. The unnormalized importance weight of each particle drawn from the importance function can be written as

$$w^{*(m)} = \frac{p(\mathbf{x}^{(m)})}{\pi(\mathbf{x}^{(m)})} \tag{2.12}$$

For sequential processing, it is important to be able to recursively update the importance weights as the observations become available. If the chosen proposal density can be factorized as

$$\pi(\mathbf{x}_{0:n}|\mathbf{y}_{1:n}) = \pi(\mathbf{x}_n|\mathbf{x}_{0:n-1}, \mathbf{y}_{1:n})\pi(\mathbf{x}_{0:n-1}|\mathbf{y}_{1:n-1}), \tag{2.13}$$

then using (2.4) and (2.13), a recursive weight update equation can be easily obtained as

$$\begin{aligned} w_n^{*(m)} &= \frac{p(\mathbf{x}_{0:n}|\mathbf{y}_{1:n})p(\mathbf{y}_{1:n})}{\pi(\mathbf{x}_n|\mathbf{x}_{0:n-1}, \mathbf{y}_{1:n})\pi(\mathbf{x}_{0:n-1}|\mathbf{y}_{1:n-1})} \\ &= w_{n-1}^{(m)} \frac{p(\mathbf{y}_n|\mathbf{x}_n)p(\mathbf{x}_n|\mathbf{x}_{n-1})}{\pi(\mathbf{x}_n^{(m)}|\mathbf{x}_{0:n-1}^{(m)}, \mathbf{y}_{1:n})} \end{aligned} \tag{2.14}$$

The importance weights are normalized as follows

$$w_n^{(m)} = \frac{w_n^{*(m)}}{\sum_{m=1}^{M} w_n^{*(m)}} \tag{2.15}$$

This sequential importance sampling procedure represents the basic particle filter.

2.3.2 Choice of Importance Function

The importance function $\pi(\cdot)$ plays a pivotal role in the performance of the filter. A poor choice of the proposal density will result in a large number of particles having insignificant weights. This not only results in poor filter performance, but also wastes computational power and resources since trajectories with insignificant contributions to the representation of the posterior are recursively updated. Strategies for choosing proposal densities have been widely proposed in the literature [37, 27, 73, 48, 49]. The density $p(\mathbf{x}_n^{(m)}|\mathbf{x}_{n-1}^{(m)}, \mathbf{y}_n)$ has been shown to be the optimal importance function in terms of minimizing the variance of importance weights [30]. Evaluation of this density is not straightforward, and hence the optimal importance function cannot be always used [4]. Sub-optimal approaches approximate the optimal importance function using local linearziations [30] or Gaussian approximations using the unscented transform [52]. The prior $p(\mathbf{x}_n^{(m)}|\mathbf{x}_{n-1}^{(m)})$ is widely used as the importance function. Although this density is not optimal the context of the above explanation, it is easy to sample from. Also using this density the expression for importance weights is greatly simplified. Due to these properties, the prior density is extremely attractive from hardware implementation point of view. Substituting the prior density as the IF in (2.14), the expression for the importance weights is given by

$$w_n^{*(m)} = w_{n-1}^{(m)} p(\mathbf{y}_n|\mathbf{x}_n) \tag{2.16}$$

2.3.3 Resampling

A common problem in particle filters is that the variance of the importance weights increases over time. This is commonly referred to as the '*weight degeneracy*' problem. As these particles are propagated in time, gradually the degeneracy becomes severe and eventually only a single particle with have normalized importance weight of 1 and all others will be zero. To prevent this degeneracy, an important procedure called Resampling is introduced into the particle filtering framework. It was first introduced in [92] and adapted for SIS in [71],[63]. Resampling is a routine applied to the particle filter in order to remove particles (trajectories) with small weights and replicate those with large weights. Theoretically, the goal of resampling is to generate a new set of particles by sampling with replacement from

the existing set of particles such that the resulting particles are i.i.d samples (with weight $1/M$) from the approximate posterior represented by the original set of particles. In other words, resampling is a procedure that replicates particles with large weights and discards those with small weights in a controlled manner. Various techniques for resampling have been proposed in the literature. These include systematic resampling [38],[29],[21], stratified resampling [14], random resampling and residual resampling [71, 72]. Although resampling is vital to the successful operation of the particle filter, it can on occasions cause an attrition of particles which leads to loss of diversity and biased estimates. Hence an effective measure should be used at each time instant to decide whether or not to resample. Frequently, the variance of the importance weights is used to indicate need for resampling. Following is a brief description of the above mentioned resamping methods.

• **Random Resampling:**

This is the simplest and most direct method for resampling. It was first introduced in [70]. Its operates using the following steps.

1. Let $\widetilde{\mathbf{x}}_n^{(i^{(m)})}$ be drawn from $\left\{\mathbf{x}_n^{(m)}, w_n^{(m)}\right\}_{m=1}^{M}$ with probability proportional to $a_n^{(m)}$. New weights associated with these particles are $\widetilde{w}_n^{(i^{(m)})} = \frac{w_n^{(m)}}{a_n^{i^{(m)}}}$.

2. Return the new random measure $\left\{\widetilde{\mathbf{x}}_n^{(i^{(m)})}, \widetilde{w}_n^{(i^{(m)})}\right\}_{i^{(m)}=1}^{M}$

This procedure has a complexity of $O(M \log M)$. Here $i^{(m)}$ represents the indexes of the particles after resampling. The above description serves as a general formulation of the resampling procedure. Other algorithms are special cases of this scheme where the probabilities $a^{(m)}$ are chosen differently for each algorithm.

• **Residual Resampling:**

Residual is performed via the following steps [71]

1. For $m = 1$ to M retain $N_m = \lceil M w_n^{(m)} \rceil$ copies of $\{\mathbf{x}_{0:n}^{(m)}\}$.

2. Calculate the residuum N_r = M - $\sum_{m=1}^{M} N_m$

3. Obtain N_r particles using random resampling and $\widetilde{w}_n^{(m)} = 1/M \; \forall \; M$.

4. Return the new random measure $\left\{\widetilde{\mathbf{x}}_n^{(m)}, \widetilde{w}_n^{(m)}\right\}_{m=1}^{M}$

Residual resampling has a complexity of $O(M)$.

• **Systematic Resampling:**

This algorithm is a simple modification of the Stratified Resampling approach proposed by [61]. This resampling is optimal in terms of variance of importance weights. These steps involved in systematic resampling are as follows.

1. For $m = 1$ to M, sample and index $j(m)$ according to the original weights such that $Pr\left(j(m) = m\right) = w_n^{*(m)}$

2. Assign resampled particle $\widetilde{\mathbf{x}}_n^{(m)} = \mathbf{x}_n^{j(m)}$ and weight $\widetilde{w}_n^{(m)} = 1/M$

3. Return new random measure $\left\{\widetilde{\mathbf{x}}_n^{(m)}, \widetilde{w}_n^{(m)}\right\}_{m=1}^{M}$

A theoretical comparison of the various resampling schemes can be found in [28]. The steps of sampling from the importance function, computation of importance weights and resampling form the most prevalent particle filter known as the Sampling Importance Resampling Filter (SIRF). We will focus on this filter at various points throughout the dissertation. The SIRF algorithm is summarized in Table 2.1.

2.4 Gaussian Particle Filters

Recently, a new class of particle filters, known as Gaussian Particle Filters (GPFs) has been introduced. We have seen that the SIRFs represent desired densities with a random measure composed of particles and weights. The GPFs on the other had are a class of Gaussian filters which approximate desired densities as Gaussian. The Gaussian approximation is described by its first two moments. The GPFs use a Monte Carlo (particle based) approach to calculate the first two moments of the Gaussian approximation. The GPF algorithm describes a technique of recursively updating the mean and variance of the Gaussian approximation and

Input:	The observation $\mathbf{y}_n$ and the random measure from the previous time instant $\{\widetilde{\mathbf{x}}_{n-1}^{(m)}, \widetilde{w}_{n-1}^{(m)}\}_{m=1}^{M}$	
Method:		
1	**Sample Step:**	Draw samples from $\pi(\mathbf{x}_n)$ to obtain $\{\mathbf{x}_n^{(m)}\}_{m=1}^{M}$.
2	**Importance Step:**	Assign weights to the particles to obtain the random measure $\{\mathbf{x}_n^{(m)}, w_n^{*(m)}\}_{m=1}^{M}$.
3	**Normalization:**	Normalize the weights by $w_n^{(m)} = w_n^{*(m)} / \sum_{m=1}^{M} w_n^{*(m)}$.
4	**Resample Step:**	Resample to obtain new random measure $\{\widetilde{\mathbf{x}}_n^{(m)}, \widetilde{w}_n^{(m)}\}_{m=1}^{M}$

Table 2.1: Sample Importance Resample Filter (SIRF) algorithm

propagating them in time. Compared to other Gaussian filters like the EKF, the GPF has been shown to be asymptotically optimal in terms of the number of particles. Hence it gives much improved performance particularly in case of nontrivial nonlinearities. The Gaussian approximation in the GPF restricts its application to cases where the desired density can be approximated as a unimodal Gaussian.

The recursive update of Gaussian approximation to the posterior involves two key steps.

1. Time Update: In this step, a Gaussian approximation to the predictive distribution $p(\mathbf{x}_n|\mathbf{y}_{0:n-1})$ is obtained. With the posterior at the previous instant $p(\mathbf{x}_{n-1}|\mathbf{y}_{0:n-1)}$ approximated as a Gaussian $\mathcal{N}(\mathbf{x}_{n-1}; \boldsymbol{\mu}_{n|n}, \boldsymbol{\Sigma}_{n|n})$, the predictive density is given by

$$p(\mathbf{x}_n|\mathbf{y}_{0:n-1}) = \int p(\mathbf{x}_n|\mathbf{x}_{n-1}) p(\mathbf{x}_{n-1}|\mathbf{y}_{0:n-1}) \tag{2.17}$$

A Monte Carlo approximation to this density can be written as

$$p(\mathbf{x}_n|\mathbf{y}_{0:n-1}) \approx \frac{1}{M} \sum_{m=1}^{M} p(\mathbf{x}_n|\mathbf{x}_{n-1}^{(m)}) \tag{2.18}$$

where $\mathbf{x}_{n-1}^{(m)}$ are particles drawn from $\mathcal{N}(\mathbf{x}_{n-1}; \boldsymbol{\mu}_{n-1|n-1}, \boldsymbol{\Sigma}_{n-1|n-1})$. The mean and covariance of the Gaussian approximation to the predictive density $p(\mathbf{x}_n|\mathbf{y}_{0:n-1})$ is obtained from these samples. This Gaussian approximation is written as $\mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_{n|n-1}, \boldsymbol{\Sigma}_{n|n-1})$.

2. Measurement Update: After receiving the n^{th} observation $\mathbf{y}_n$, the filtering distribution and its Gaussian approximation is given by

$$\begin{aligned} p(\mathbf{x}_n|\mathbf{y}_{0:n}) &= C_n p(\mathbf{y}_n|\mathbf{x}_n) p(\mathbf{x}_n|\mathbf{y}_{0:n-1}) \\ &\approx p(\mathbf{y}_n|\mathbf{x}_n)\mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_{n|n-1}, \boldsymbol{\Sigma}_{n|n-1}). \end{aligned} \tag{2.19}$$

An Importance sampling based approach is used to represent the filtering distribution in (2.19). Accordingly, an importance function $\pi(\cdot)$ is selected and samples are drawn from this density using the approximate predictive density. Weights are assigned to these samples according to

$$w_n^{(m)} = \frac{p(\mathbf{y}_n|\mathbf{x}_n^{(m)})\mathcal{N}(\mathbf{x}_n^{(m)}; \boldsymbol{\mu}_{n|n-1}, \boldsymbol{\Sigma}_{n|n-1})}{\pi(\mathbf{x}_n^{(m)})} \tag{2.20}$$

The particles and weights are used to calculate the parameters of the Gaussian approximation to this density $\mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_{n|n}, \boldsymbol{\Sigma}_{n|n})$. The first two moments of this approximation are propagated in time.

Unlike the SIRF, the entire set of particles is not propagated in the GPF. As a result, particle degeneration does not occur and Resampling is not needed in the GPF. This is an attractive feature from a hardware implementation perspective, since resampling is an inherently sequential process which acts as a bottleneck in real time processing.

As noted in [47] pp. 65, the implementation of the GPF can be simplified by using the prior density as the IF. This means that $\pi(\mathbf{x}_n^{(m)})$ is chosen to be equal to the $p(\mathbf{x}_n|\mathbf{y}_{0:n-1}) = \mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_{n|n-1}, \boldsymbol{\Sigma}_{n|n-1})$,, where $\boldsymbol{\mu}_{n|n-1}$ and $\boldsymbol{\Sigma}_{n|n-1}$ represent the sample mean and covariance of predictive distribution. This allows the samples generated in the time update step to be

used directly in measurement update. The GPF algorithm, using this importance function is summarized in Table 2.2.

Input: The observation $\mathbf{y}_n$ and first two moments of the Gaussian approximation of the posterior at previous instant $\boldsymbol{\mu}_{n-1}$ and $\boldsymbol{\Sigma}_{n-1}$

Method:

GPF - Time update algorithm.

1. Draw conditioning particles from $\mathcal{N}(\mathbf{x}_{n-1}; \boldsymbol{\mu}_{n-1}, \boldsymbol{\Sigma}_{n-1})$ to obtain $\{\mathbf{x}_{n-1}^{(m)}\}_{m=1}^{M}$.
2. Generate particles by drawing samples from $p(\mathbf{x}_n|\mathbf{x}_n = \mathbf{x}_{n-1}^{(m)})$ to obtain $\{\mathbf{x}_n^{(m)}\}_{m=1}^{M}$.

GPF - Measurement update algorithm

3. (a) Calculate weights by $\tilde{w}_n^{(m)} = p(\mathbf{y}_n|\mathbf{x}_n^{(m)})$.
   (b) Normalize the weights by $w_n^{(m)} = \tilde{w}_n^{(m)} / \sum_{m=1}^{M} \tilde{w}_n^{(m)}$.
4. Estimate the mean and covariance of the filtering distribution by
   (a) $\boldsymbol{\mu}_n = \sum_{m=1}^{M} w_n^{(m)} \mathbf{x}_n^{(m)}$
   (b) $\boldsymbol{\Sigma}_n = \sum_{m=1}^{M} w_n^{(m)} (\mathbf{x}_n^{(m)} - \boldsymbol{\mu}_n)(\mathbf{x}_n^{(m)} - \boldsymbol{\mu}_n)^T$

Table 2.2: Gaussian Particle Filter (GPF) algorithm.

2.5 Bearings Only Tracking

The complexity and performance of particle filters depends, to some extent, on the model to which the filters are applied. Specifically, the model nonlinearity, model dynamics and dimension of the state affect several performance aspects. These include estimation accuracy, required number of particles to achieve this, requirement of resampling and complexity of the importance sampling and weight computation. We consider various models throughout the dissertation, but one of the most frequently considered applications is Bearings-Only Tracking (BOT). This is frequently seen in SONAR and other angle-of-arrival based tracking systems [15, 81]. Particle Filtering for BOT was first introduced in [38]. The BOT problem is illustrated in Figure 2.1.

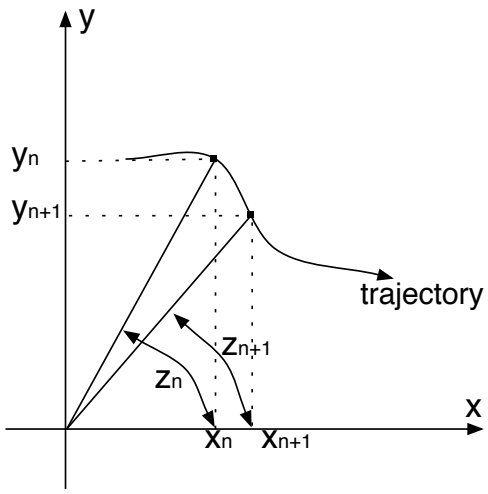


Figure 2.1: The Bearings-Only Tracking (BOT) problem.

BOT involves tracking of an object (typical examples include ships, planes and other moving vehicles) based upon its measured bearing (angle) with respect to a fixed sensor. The problem is typically difficult due to non availability of range information. The system state in BOT is 4 dimensional and includes the positions and velocities along the Cartesian x and y directions. The positions are expressed with reference to the fixed sensor location. The state equation in the BOT problem describes the system kinematics modelled with a Constant Velocity model. The sampling period is taken to be $T_S = 1$ unit. The measurement is the bearing or angle of the object with respect to the sensor axis (y axis in our case).

The DSS model for BOT is

$$\mathbf{x}_n = \mathbf{\Phi}\mathbf{x}_{n-1} + \mathbf{\Gamma}\mathbf{u}_n \quad n = 1, \ldots, N \tag{2.21}$$

$$z_n = \tan^{-1}(y_n/x_n) + v_n \tag{2.22}$$

where $\mathbf{x}_n = [x_n,\ vx_n,\ y_n,\ vy_n]^T, \mathbf{u}_n = [ux_n,\ uy_n]^T$,

$$\mathbf{\Phi} = \begin{bmatrix} 1 & T_s & 0 & 0 \\ 0 & T_s & 0 & 0 \\ 0 & 0 & 1 & T_s \\ 0 & 0 & 0 & T_s \end{bmatrix}, \mathbf{\Gamma} = \begin{bmatrix} T_s^2/2 & 0 \\ T_s & 0 \\ 0 & T_s^2/2 \\ 0 & T_s \end{bmatrix}. \mathbf{C_u} = \begin{bmatrix} 0.005 & 0 \\ 0 & 0.005 \end{bmatrix}$$

where $\mathbf{x}_n$ is the state vector which includes the positions and velocities x_n, vx_n, y_n, vy_n of the object in the x and y directions respectively. $\mathbf{u}_n$ is a zero mean Gaussian white noise process with covariance matrix $\mathbf{C_u}$. This vector can be thought of as representing the acceleration in the x and y directions. z_n represents the observed bearing of the object measured by the sensor at time n. v_n is a zero mean white noise process with variance $\sigma_v^2 = 0.001$. Before measurements are taken, the particle filter recursion is started with initial prior information in the form of a 4 dimensional Gaussian variable with known mean and covariance matrix.

As can be seen, this model is 4 dimensional and highly nonlinear due to a transcendental function in the observation equation. These are the scenarios in which traditional filters function poorly. The particle filters handle these situations efficiently. It has been shown through intense simulations in [38], [64], that particle filters are much more efficient for this problem than the traditional EKF.

2.5.1 SIRF for BOT

The SIRF applied to the BOT includes the usual steps of Sampling, Importance computation and Resampling. The output of the filter is an minimum mean square error (MMSE) estimate of the state vector. We chose the prior density as the IF. Although not optimal in terms of variance of importance weights, this choice is most efficient from a hardware implementation viewpoint. To prevent degeneracy, particularly due to the use of the prior density as the IF, we chose to perform resampling at each step. SIRF for the BOT problem is summarized Table 2.3.

Input: The observation z_n and previous estimates

1	Sample	Draw particles from $p(\mathbf{x}_n \vert \mathbf{x}_{n-1} = \mathbf{x}_{n-1}^{(m)})$ to obtain $\{\mathbf{x}_n^{(m)}\}_{m=1}^{M}$. $x_n^{(m)} = x_{n-1}^{(m)} + vx_{n-1}^{(m)} + ux_n^{(m)}$ $vx_n^{(m)} = vx_{n-1}^{(m)} + ux_n^{(m)}$ $y_n^{(m)} = y_{n-1}^{(m)} + vy_{n-1}^{(m)} + uy_n^{(m)}$ $vy_n^{(m)} = vy_{n-1}^{(m)} + uy_n^{(m)}$
2	Importance	Calculate weights by $w_n^{*(m)} = p(z_n \vert \mathbf{x}_n^{(m)})$. $w_n^{*(m)} = \widetilde{w}_{n-1}^{(m)} e^{-(2\pi\sigma_v^2)^{-1}\left(z_n - atan\frac{y_n^{(m)}}{x_n^{(m)}}\right)^2}$
3	Normalization	Normalize the weights by $w_n^{*(m)} = w_n^{*(m)} / \sum_{m=1}^{M} w_n^{*(m)}$.
4	Resampling	Resample particles to obtain new particles with weights $\frac{1}{M}$
5	Output	Calculate the outputs $\hat{x}_n = \sum_{m=1}^{M} w_n^{(m)} x_n^{(m)}$ $\hat{v}x_n = \sum_{m=1}^{M} w_n^{(m)} vx_n^{(m)}$ $\hat{y}_n = \sum_{m=1}^{M} w_n^{(m)} y_n^{(m)}$ $\hat{v}y_n = \sum_{m=1}^{M} w_n^{(m)} vy_n^{(m)}$

Table 2.3: SIRF applied to Bearings-Only Tracking

2.5.2 GPF for BOT

For efficient implementation, we use the prior density $p(\mathbf{x}_n|\mathbf{x}_{n-1})$ as the importance function as explained earlier. Conditioning particles are drawn from the Gaussian approximated posterior of the previous time instant $\mathcal{N}(\mathbf{x}_{n-1}; \boldsymbol{\mu}_{n-1}, \boldsymbol{\Sigma}_{n-1})$. This requires decomposition of the covariance matrix $\boldsymbol{\Sigma}_{n-1} = C_n \cdot C_n^T$. This is done using Cholesky decomposition. It is well known that $C_n \cdot \mathbf{q}$, where $\mathbf{q}$ is a (4 x 1) vector $[q_1\ q_2\ q_3\ q_4]^T$ of white Gaussian noise samples, results in samples with covariance matrix $\boldsymbol{\Sigma}_n$. The GPF algorithm for BOT is shown in Table 2.4

Input: The observation z_n and moments of previous Gaussian approximation $\boldsymbol{\mu}_{n-1}$ and $\mathbf{C}_{n-1}$ where C is Cholesky decomposed $\boldsymbol{\Sigma}$

1. Draw conditioning particles to obtain $\{\mathbf{x}_{n-1}^{(m)}\}_{m=1}^{M}$.
   $x_{n-1}^{(m)} = \mu_1 + C_{11}q_1$
   $vx_{n-1}^{(m)} = \mu_2 + C_{21}q_1 + C_{22}q_2$
   $y_{n-1}^{(m)} = \mu_3 + C_{31}q_1 + C_{32}q_2 + C_{33}q_3$
   $vy_{n-1}^{(m)} = \mu_4 + C_{41}q_1 + C_{42}q_2 + C_{43}q_3 + C_{44}q_4$
2. Draw particles from $p(\mathbf{x_n}|\mathbf{x_{n-1}} = \mathbf{x_{n-1}^{(m)}})$ to obtain $\{\mathbf{x_n^{(m)}}\}_{\mathbf{m=1}}^{\mathbf{M}}$.
   $x_n^{(m)} = x_{n-1}^{(m)} + vx_{n-1}^{(m)} + ux_n^{(m)}$
   $vx_n^{(m)} = vx_{n-1}^{(m)} + ux_n^{(m)}$
   $y_n^{(m)} = y_{n-1}^{(m)} + vy_{n-1}^{(m)} + uy_n^{(m)}$
   $vy_n^{(m)} = vy_{n-1}^{(m)} + uy_n^{(m)}$
3. Assign Weights
   $w_n^{*(m)} = w_{n-1}^{(m)} e^{-(2\pi\sigma_v^2)^{-1}\left(z_n - atan\frac{y_n^{(m)}}{x_n^{(m)}}\right)^2}$
4. Normalize the weights by $w_n^{(m)} = w{*}_n^{(m)} / \sum_{m=1}^{M} \bar{w}_n^{(m)}$.
5. Estimate mean and covariance by
   $\boldsymbol{\mu}_n = \sum_{m=1}^{M} w_n^{(m)} \mathbf{x}_n^{(m)}$
   $\boldsymbol{\Sigma}_n = \sum_{m=1}^{M} w_n^{(m)} (\mathbf{x}_n^{(m)} - \boldsymbol{\mu}_n)(\mathbf{x}_n^{(m)} - \boldsymbol{\mu}_n)^T$.

Table 2.4: **Gaussian particle filter for BOT**

2.5.3 Tracking Performance

An in-depth analysis of the tracking performance of the two filters for the BOT problem is investigated in various works including [38, 64, 29]. We performed some simulations to establish a performance benchmark to compare the performance of eventual hardware implementations. The two filters were applied to a generated true path using $M = 2000$ particles. Obviously, with increasing particles, the performance in terms of MSE improves. Hence, we compare the performance in terms of MSE to the Posterior Cramér Rao Lower Bound (PCRLB). The PCRLB for the BOT is calculated using a methodology presented in [103]. The MSE is obtained independently for each element of the state in the BOT problem.

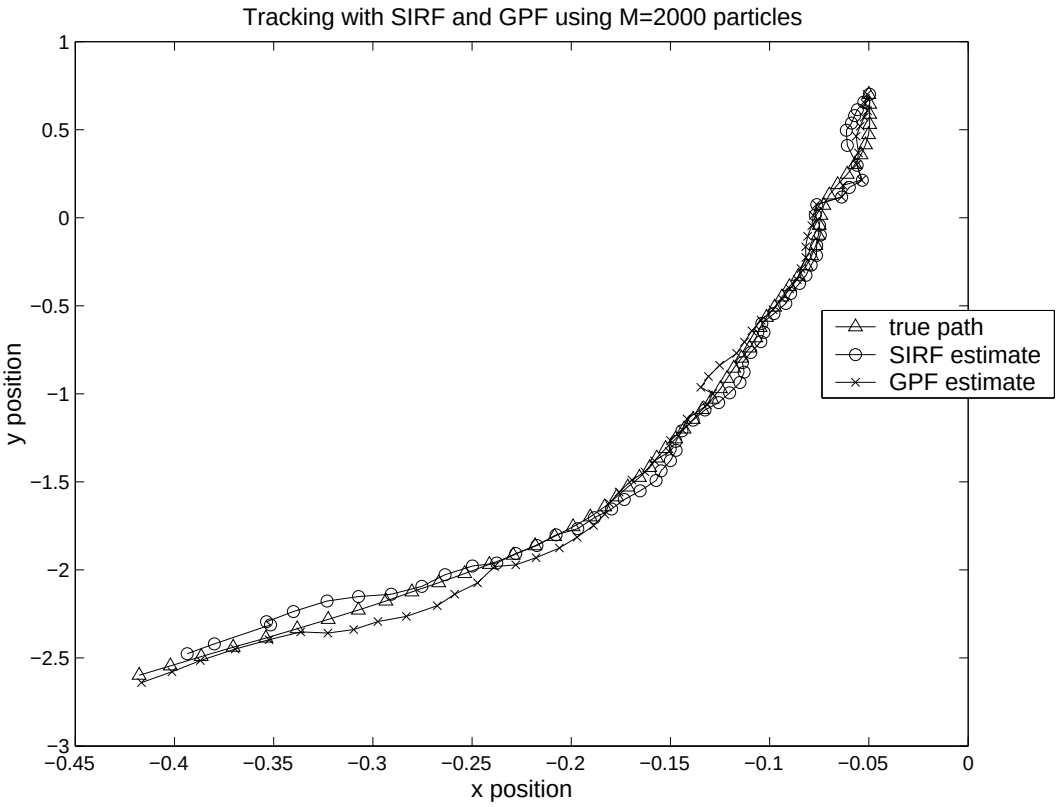


Figure 2.2: One realization of GPF and SIRF tracking.

The MSE for the x position can be expressed as

$$MSE_x(n) = \frac{1}{R}\sum_{r=1}^{R}(\hat{x}_n - x_n)^2 \tag{2.23}$$

where R is the total number of Monte Carlo realizations over which the MSE is averaged. Each of these realizations used observations from the same generated true state. For our experiment, we chose $R = 100$.

Figure 2.2 shows the tracking performance of one of the realizations. Figure 2.3 compares the performance of the SIRF and GPF each with $M = 2000$ particles for tracking the x and y positions with the PCRLB.

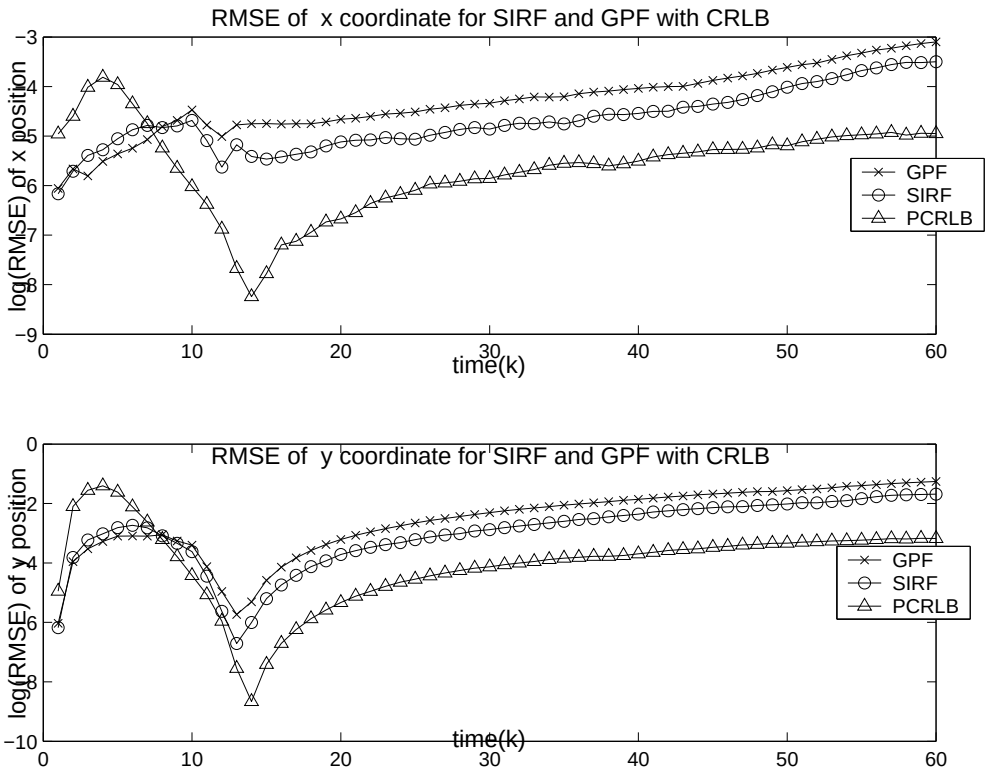


Figure 2.3: Comparison of MSE of x and y positions for GPF and SIRF with the PCRLB.

Chapter 3

Algorithmic Analysis and Modification for Implementation

3.1 Introduction

Until very recently, almost all the research on Particle Filtering was related to its theory. New algorithms were developed with the aim of improving tracking performance and being able to handle complex nonlinear and non-Gaussian problems. PFs, due to the nature of SMC techniques are inherently computationally intensive. This can be clearly seen from the algorithms presented in the previous chapter. Typically, PFs need to work with a large number of particles when handling complex real world scenarios. The processing of so many samples through mathematically complex steps, naturally makes the filtering intensive and slow on sequential machines. The traditional PF algorithms presented in the previous chapter are sequential in nature and as such not well suited to implementation. In this chapter, we analyze some of the algorithmic parameters of PFs and present several modifications that allow for efficient implementation, both in terms of efficient resource usage and increased speed due to the parallelism in the modified algorithms.

It is well understood in the field of VLSI signal processing, that the most impactful design modifications are those made at the algorithmic level. For example, the results published in [66, 90] show that the most dramatic power reduction in the final hardware results from optimizations at the algorithm level rather than silicon (gate/circuit) level. In this chapter,

we will first look briefly at the philophy of joint algorithm-architecture design, and then see how this can be applied to particle filters and the implementation related advantages arising therefrom.

3.2 Joint Algorithm-Architecture Design

While developing algorithms, its performance on a given problem is often treated as the sole design criterion. However, decisions made at this level affect several implementation aspects like speed, power and resources. Joint algorithm-architecture design implies exploration of the algorithm and architecture design spaces jointly i.e. using algorithm parameters to direct architectural choice, and using architecture details in turn to optimize the algorithm. This design paradigm works with coarse grain, blocks based algorithm descriptions to drive the design process. This reveals the inherent nature of the algorithm such as parallelism, data dependencies, and function complexity. These parameters are extracted and used in architecture exploration. The architecture can range from from a sequential processor to dedicated spatial hardware. For a particular chosen architecture, the implementation cost is estimated, and this is in turn used as feedback to the algorithm development for possible optimization.

High level design decisions include choice of algorithm for a given application, fine tuning the algorithm parameters, selection of the architecture and the determination of the implementation parameters.

3.2.1 Algorithm Parameters

A better understanding of the correspondence between algorithm properties and architectural implications, is a key to getting the most out of high level optimizations. For this purpose, it is necessary to charecterize the algorithm using a set of relevant, measurable metrics that can be used as guidelines in subsequent architecture exploration. Some of these parameters are [114, 39]

1. **Numerical Properties:** This property relates to the format used and precision required for representing various quantities (variables) in hardware. This affects the

architecture in terms of size of computation units and also execution time. There is a tradeoff between algorithm performance and architecture and implementation cost. Choice of appropriate word lengths and format is a good example of joint algorithm-architecture design explained above. In the sequel, we have carefully examined this issue in the context of particle filter implementation.

2. **Complexity and amount of Computation:** This characteristic can be easily extracted from the coarse block based functional description of the algorithm. A frequent way of quantifying this characteristic is by the number of basic operations (addition, multiplication, memory access) required by each functional block. This is used as the initial lower bound on the amount of resources needed in the target architecture platform to drive exploration of the architecture space.

3. **Regularity:** This property points to the ability to realize the algorithm with repetitive use of simpler basic operations. This allows for structured, cell based hardware design. Systolic processors attempt to exploit regularity in the algorithm.

4. **Locality:** The concept of locality is frequently used in the broad field of computer architecture. Locality is described as temporal or spatial. Temporal locality refers to the property of performing a function repeatedly in time. Spatial locality is more of an architectural metric which refers to the property of accessing spatially close resources frequently. In the VLSI domain temporal locality measures a variables persistence before being re-evaluated and spatial locality refers to the communication pattern between various processing blocks.

5. **Data dependency and Concurrency:** This property determines the critical path and also the scalability of the algorithm in terms of added hardware. Concurrency refers to the different tasks in the algorithm that are independent of each other and can be performed simultaneously as long as enough hardware resources are present. Data dependencies often determine the critical execution path and the achievable speed of the hardware.

3.2.2 Architectural Parameters

The architectural design space spans various levels of parallelism and programmability. It extends from sequential machines like DSPs to dedicated ASIC hardware. The architectural parameters influence the absolute implementation metrics i.e. speed, power and cost. The role of architecture charecterization is to provide measurable metrics as feedback to the algorithm development for suitable optimization and also to provide rough estimates of the cost of implementation. Some common architecture characterization metrics are

1. **Granularity of Processing blocks:** This usually governs the amount of flexibility in the architecture. Higher the granularity, more the flexibility.

2. **Time multiplexing:** This metric spans the design space from sequential to parallel machines. In sequential machines, the degree of time multiplexing is high. They consist of minimal resources which are reused in time. Parallel machines allocate hardware with the goal of increasing the throughput.

We will say more about exploration of the architectural design space when describing architectures developed for particle filters. In the following sections, we will describe some algorithmic modifications of particle filters for better implementation.

3.3 Algorithmic Modification of the SIRF

A coarse-grain, block based functional representation of the SIRF is shown in Figure 3.1. The input to the filter are observations which are sampled and used in the importance computation step. The output of the filter is the estimate of the hidden state. The traditional algorithm is loop intensive, in the sense that operations are repeated several times during a single recursion. For making a pipelined implementation possible, the concept of loop fusion or chaining is used. When there are several loops in an algorithm running for the same number of iterations, then all the operations on the same index can be combined in a single loop. This loop fusion can be used to combine the SIRF loops that process different operations for the same index. This allows overlapping of operations at algorithm level and pipelining at the architectural level. The pipelined SIRF algorithm for the BOT problem is shown in Table 3.1. The resampling step however is still completely sequential.

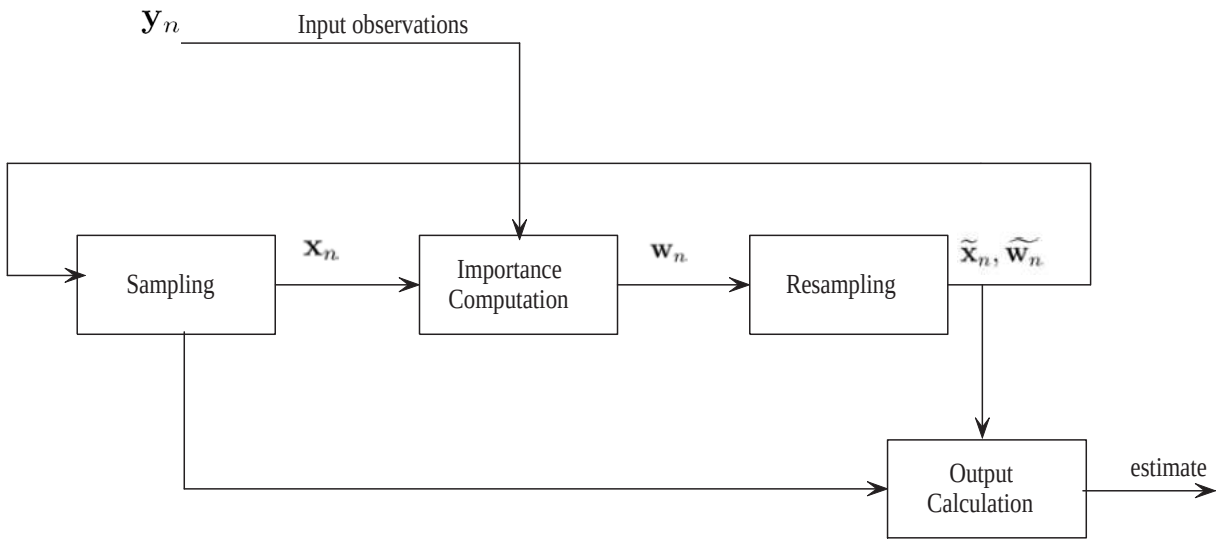


Figure 3.1: Functional block diagram of the SIRF

Traditional SIRF algorithm uses systematic resampling (SR) with normalized weights. Normalization is an intensive operation as it requires M divisions which are expensive in hardware. This operation may also act as a bottleneck since the resampling operation cannot start until the normalized set of weights is available. The first modification we present eliminates the need for the weight normalization step by using the unnormalized weights along with the sum the weights,W_n, directly in the resampling process. A modified SR algorithm which works with non-normalized weights is shown in Table 3.2. Using this algorithm M divisions during normalization are replaced with one division as shown in step 2 of the pseudocode. This is a generic pseudocode where, N_i is the input number of particles, N_o is the number of particles generated after resampling, and $\{w_n^{*(m)}\}_{m=1}^{M}$ is the input array of non-normalized weights from the importance step. In standard implementations $N_i = N_o = M$. The output i_n is an array of indexes, which shows the addresses of the particles in the original set that constitute the resampled set. The algorithm works by drawing the uniform random number U from the support $[0, \frac{W_n}{N_o}]$ and then updating it using step 10. At the same time, the sum of the first k particle weights is calculated (C) and compared with U. When $C < U$ the last particle is discarded and the weight of the particle $k+1$ is added to S. If $C > U$ the particle k will be replicated and the number of replications is proportional to $E(\widetilde{w}_n^{(k)} N_o / W_n)$. It should be noted that this modification does not change the effect of

Input: The observation z_n and the particles from the previous instant $\{\mathbf{x}_{n-1}^{(m)}\}_{m=1}^{M}$

Method:

for $m = 1, \ldots, M$,

1. Sample Step

$$x_n^{(m)} = x_{n-1}^{(m)} + vx_{n-1}^{(m)} + u_x^{(m)}$$
$$vx_n^{(m)} = vx_{n-1}^{(m)} + u_x^{(m)}$$
$$y_n^{(m)} = y_{n-1}^{(m)} + vy_{n-1}^{(m)} + u_y^{(m)}$$
$$vy_n^{(m)} = vy_{n-1}^{(m)} + u_y^{(m)}$$

2. Importance step

$$w_n^{*(m)} = e^{-(2\pi\sigma_v^2)^{-1}\left(z_n - atan\frac{y_n^{(m)}}{x_n^{(m)}}\right)^2}$$
$$W_n = W_n + w_n^{*(m)}$$

3. Calculate partial output.

$$\widehat{x}_n = \widehat{x}_n + x_n^{(m)} \cdot w_n^{*(m)}$$
$$\widehat{v}x_n = \widehat{v}x_n + x_n^{(m)} \cdot w_n^{*(m)}$$
$$\widehat{y}_n = \widehat{y}_n + x_n^{(m)} \cdot w_n^{*(m)}$$
$$\widehat{v}y_n = \widehat{v}y_n + x_n^{(m)} \cdot w_n^{*(m)}$$

end

4. Resample particles using sum of weights.
5. Scale the estimates

$$\widehat{x}_n = \widehat{x}_n / W_n$$
$$\widehat{v}x_n = \widehat{v}x_n / W_n$$
$$\widehat{y}_n = \widehat{y}_n / W_n$$
$$\widehat{v}y_n = \widehat{v}y_n / W_n$$

Table 3.1: SIRF for BOT with pipelining and loop fusion

resampling. In other words, traditional SR using normalized weights and the algorithm in Table 3.2 produce exactly the same result. We shall look more close at the resampling step

and its cycle-by-cycle operation when we consider its implementation.

Input: Non normalized set of weights $w_n^{(m)}$
Method:

1. (i_n)=SR(N_i,N_o,W_n, w_n^*)
2. $A_d = \frac{W_n}{N_o}$
3. Generate random number $U \sim \mathcal{U}[0, A_d]$
4. C=0, k=0
5. **for** $m = 1 : N_i$
6. **while** $(C < U)$
7. $k = k + 1$
8. $C = C + w_n^{*(k)}$
9. **end**
10. $U = U + A_d$
11. $i_n^{(m)} = k$
12. **end**

Table 3.2: Systematic Resamling with non normalized weights

The SIRFs have a high degree of spatial concurrency due to their regular nature and the fact that operations on different trajectories at the same sampling instant are independent of each other. This allows for parallelizing the SIRF using a distributed architecture. Operations of sampling (propagation) and weight calculation for different particles are independent and each require M iterations for one particle filter recursion. This allows for exploiting the loop-level parallelism using the software concept of loop unrolling [45]. Accordingly, a parallel architecture framework for the particle filter is suggested in Figure 3.2. Here each processing element (PE) processes a fraction of the total particles performing the steps of sampling and importance computation. Resampling is a sequential operation with data dependencies between each iteration. Hence it cannot be parallelized in the same way as the other steps. In the framework shown in Figure 3.2, the particles from each PE are sent to the central unit (CU) for resampling.

We have modified the resampling procedure in a way that allows for partial parallelization. According to this scheme, resampling is split into a two stage process. In the first stage, each PE sends *only its sum of weights* to the CU. Based on this the CU determines how

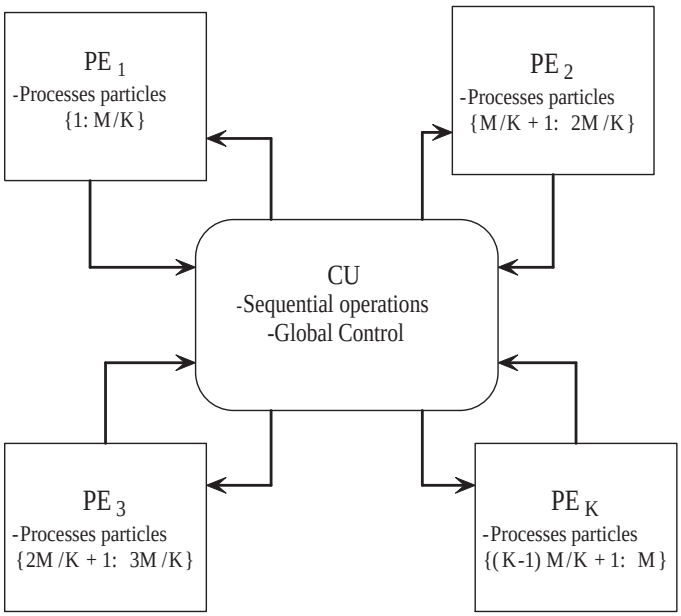


Figure 3.2: Parallel Architecture Framework.

many particles each PE should produce after resampling. Then concurrently in each PE, resampling takes place according to the algorithm in Table 3.2 to produce the appropriate number of particles. These particles are then propagated. The whole resampling process now consists of three operations:

1. CU resampling which is sequential operation in which the CU first calculates the number of particles $N^{(k)}$ that each PE should produce after resampling based on its sum of weights $W_n^{(k)}$ for $k = 1, ..., K$ where K is the number of PEs.

2. After the PEs get number $N^{(k)}$, resampling is executed in PEs in parallel. If resampling is performed based on Pseudocode 1, the input number of particles is equal for all the PEs $N_i = M/K$ and the output number of particles varies $N_o = N^{(k)}$.

3. Data exchange in which particles among PEs are exchanged in a way that PEs with the surplus send the particles to the PEs with the lack of particles. This step is necessary in order to assure that all the PEs have the same number of particles before the next sampling period.

Using this modification, the time for resampling in parallel implementation is reduced

K times in comparison with the implementation in which resampling is performed only by the CU. The average time for data exchange is reduced as well. In the parallel framework of Figure 3.2, sampling importance computation and parallel resampling (item 2. above) are mapped into PEs, while the sequential operations such as CU resampling and data exchange are mapped to the CU.

This scheme does add the overhead of the interconnect and data transfer. The interconnect pattern and protocols for data exchange between PEs and the CU is a complicated issue. They decide the efficiency of the parallel implementation and in terms of cost and scalability. Several strategies to handle these issues have been proposed in [17]. We will look more closely at these issues and propose solutions when we use the concept of a parallel architecture along with distributed resampling in the context of multiple model particle filters in Chapter 6

3.4 Algorithmic Modification of the GPF

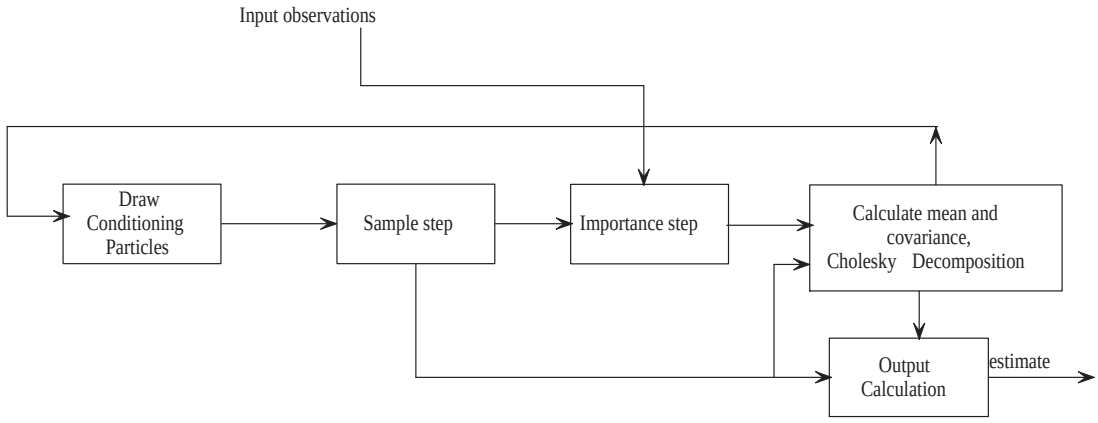


Figure 3.3: Functional block diagram of the GPF.

A coarse grain functional block diagram of the GPF is shown in Figure 3.3. Temporal concurrency in the GPF exists as a concurrency of operations that can be executed simultaneously [65]. Degree of temporal concurrency in the algorithm implies storage requirement for executing the algorithm in hardware. It is observed that the GPF contains four loops of M iterations, where each loop is used for calculation of one step in Table 2.2. Since the results from step one are used in the following steps, all M values of the states and weights

must be saved in the memory for further processing. However, all four steps have the same number of iterations and there are no inter-iteration data dependencies. As such these loops are suitable for loop fusion [96] like some operations in the SIRF as described previously. Steps 1, 2 and 3(a) in Table 2.2 can be easily fused. Weight normalization requires that all the weights are known in order to form the sum of the weights. These normalized weights are then used to calculate the mean and covariance of Gaussian approximation to the posterior. These operations in this form cannot be fused with the main loop. However, we present a modification to the algorithm that avoids normalization and allows for computing unscaled mean and covariance as part of the main execution loop. This is done by using multiply - and-accumulate (MAC) operation with the particles and non-normalized weights to partially calculate the mean. Calculation of the variance still poses a problem as it requires the mean to be calculated first. However, this step can be rewritten as:

$$\mathbf{\Sigma}_n = \frac{1}{W_n} \sum_{m=1}^{M} \tilde{w}_n^{(m)} \mathbf{x}_n^{(m)} \mathbf{x}_n^{T(m)} - \mu_n \mu_n^T. \tag{3.1}$$

The RHS of the subtraction is a constant and it can be calculated outside of the loop. Calculation of the value on the LHS of the subtraction can be fused with the main loop as MAC operations. This fused loop also exhibits loop-level parallelism which can be exploited as in case of the SIRF using loop unrolling and the parallel architecture framework of Figure 3.2. Here again each PE performs the independent operations on a fraction of the total number of particles in parallel with other PEs. The part of the algorithm mapped to each PE is presented in Table 3.3.

The weights w_n and the states $\mathbf{x}_n$ in Pseudocode 2 are represented without the superscript (m). This implies that these quantities are not saved in memories. This result is very important since it shows that there is no need for storing particles in hardware. Intuitively this makes sense, since the GPF does not propagate entire a complete random measure consisting of the particles and the weights like in the SIRF. Only the first two moments of a Gaussian approximation are propagated, and hence only these should require storing between two time instants. However, as we saw, algorithm modifications are necessary to exploit this property and translate it to reduced memory usage in hardware.

Input: The observation $\mathbf{y}_n$ and previous estimates μ_{n-1} and the matrix $\mathbf{C}_{n-1}$ s.t. $\mathbf{\Sigma}_{n-1} = \mathbf{C}_{n-1}\mathbf{C}_{n-1}^T$

Method:

for $m = 1$ **to** M

1. Draw a conditioning particle from $\mathcal{N}(\mathbf{x}_{n-1}; \mu_{n-1}, \mathbf{\Sigma}_{n-1})$ to obtain $\mathbf{x}_{n-1}$.
2. Draw a sample from $p(\mathbf{x}_n|\mathbf{x}_{n-1})$ to obtain $\mathbf{x}_n$.
3. (a) Calculate a weight by $\tilde{w}_n = p(\mathbf{y}_n|\mathbf{x}_n)$.
   (b) Update the current sum of weights by $W_n^k = W_n^k + \tilde{w}_n$.
4. Update the current mean and covariance by
   (a) $\mu_n^k = \mu_n^k + \tilde{w}_n\mathbf{x}_n$
   (b) $\mathbf{\Sigma}_n^k = \mathbf{\Sigma}_n^k + \tilde{w}_n\mathbf{x}_n(\mathbf{x}_n)^T$.

end

Table 3.3: Part of the GPF algorithm that runs in parallel on PEs after loop fusion is applied.

The GPF does not require resampling as pointed out in Chapter 2. Nevertheless, the CU needs to perform the additional processing required outside the main loop. This involves: final normalization (scaling) of the mean and covariance coefficients, calculation of the final covariance coefficients using Eq. 3.1 and Cholesky decomposition. Cholesky decomposition is necessary since step 1 of the GPF algorithm requires drawing conditioning particles from the Gaussian with the specified covariance matrix. This is done by first generating a vector of white Gaussian noise using standard techniques and then premultiplying this vector with the decomposed matrix C_n.

Thus the algorithm modifications in the GPF allow for exploiting the inherent temporal concurrency which results in a pipelined architecture and highly reduced memory usage. Spatial concurrency is exploited by modifying certain algorithm steps such that a parallel implementation is possible with distributed PEs and a CU.

For completeness, the complete modified GPF algorithm as applied to the BOT problem is presented in Table. 3.5

Input: W_n^k, μ_n^k and $\mathbf{\Sigma}_n^k$ for $k = 1, ..., K$.
Method:
1. Collect and update central sum of weights, mean and covariance

for $k = 1$ **to** K
(a) $W_n = W_n + W_n^k$.
(b) $\mu_n = \mu_n + \mu_n^k$
(c) $\mathbf{\Sigma}_n = \mathbf{\Sigma}_n + \mathbf{\Sigma}_n^k$

end
2. (a) Scale mean and covariance
(a) $\mu_n = \mu_n / W_n$
(b) $\mathbf{\Sigma}_n = \mathbf{\Sigma}_n / W_n$
3. Find the covariance estimate $\mathbf{\Sigma}_n = \mathbf{\Sigma}_n - \mu_n(\mu_n)^T$
4. The Cholesky decomposition of the matrix $\mathbf{\Sigma}_n$ in order to obtain $\mathbf{C_n}$.

Table 3.4: Part of the GPF algorithm that runs sequentially on the CU.

3.5 Data Flow Analysis

The particle filter is a highly data driven program. The execution of the filter or some operation within the filter is triggered by the availability of new data (token). Hence they are well suited to description by data flow graphs [105]. In data flow graphs, each instruction is viewed as the node of a graph. The output of the node is connected to all the other nodes that consume the output. Each node executes when its triggering input becomes available. The flow graph representation is the starting point of most architectural analysis and synthesis [39]. The branches in the graph represent the data dependences between the various operations. Branches representing control signals are included only when they express additional triggering conditions for a node. Other than this case, in a data flow graph representation the control flow is completely separated. This is very useful to analyze applications like the particle filter which is highly data dominated and control at the high level consists only of a counter that counts up to the number of particles. Thus the particle filter is a highly data dominated algorithm, with a regular structure and parallelism which is exploited by the modified algorithms presented in the earlier section. This fits well into the Single Instruction Multiple Data (SIMD) architecture environment [34].

Input: The observation z_n and the parameters μ_{n-1} and $\mathbf{\Sigma}_{n-1}$ from the previous time instant

Method:

for $m = 1, \ldots, M$,

1. Draw conditioning particles from $\mathcal{N}(\mathbf{x}_{n-1}; \mu_{n-1}, \mathbf{\Sigma}_{n-1})$

$x_{n-1} = \mu_1 + C_{11} \cdot q_1$
$vx_{n-1} = \mu_2 + C_{21} \cdot q_1 + C_{22} \cdot q_2$
$y_{n-1} = \mu_3 + C_{31} \cdot q_1 + C_{32} \cdot q_2 + C_{33} \cdot q_3$
$vy_{n-1} = \mu_4 + C_{41} \cdot q_1 + C_{42} \cdot q_2 + C_{43} \cdot q_3 + C_{44} \cdot q_4$

2. Generate particles $p(\mathbf{x}_n|\mathbf{x}_{n-1})$

$x_n = x_{n-1} + vx_{n-1} + u_x$
$vx_n = vx_{n-1} + u_x$
$y_n = y_{n-1} + vy_{n-1} + u_y$
$vy_n = vy_{n-1} + u_y$

2. Importance step

$w_n^* = e^{-(2\pi\sigma_v^2)^{-1}\left(z_n - atan\frac{y_n}{x_n}\right)^2}$
$W_n = W_n + w_n^*$

3. Calculate partial mean and covariance.

$\widehat{x}_n = \widehat{x}_n + x_n \cdot w_n^*$
$\widehat{v}x_n = \widehat{v}x_n + vx_n \cdot w_n^*$
$\widehat{y}_n = \widehat{y}_n + y_n \cdot w_n^*$
$\widehat{v}y_n = \widehat{v}y_n + vy_n \cdot w_n^*$
$\Sigma_n = \Sigma_n + w_n^* \cdot \mathbf{x}_n \cdot (\mathbf{x}_n)^T$

end

4. Calculate final mean and covariance (output estimate)

$\widehat{x}_n = \widehat{x}_n / W_n$
$\widehat{v}x_n = \widehat{v}x_n / W_n$
$\widehat{y}_n = \widehat{y}_n / W_n$
$\widehat{v}y_n = \widehat{v}y_n / W_n$
$\mathbf{\Sigma}_n = \mathbf{\Sigma}_n / W_n$ - $\mu_n^T \cdot \mu_n$

5. Cholesky decomposition of $\mathbf{\Sigma}_N = \mathbf{C}_n^T \cdot \mathbf{C}_n$

Table 3.5: Modified GPF for the BOT problem

3.5.1 Data flow and timing for SIRF

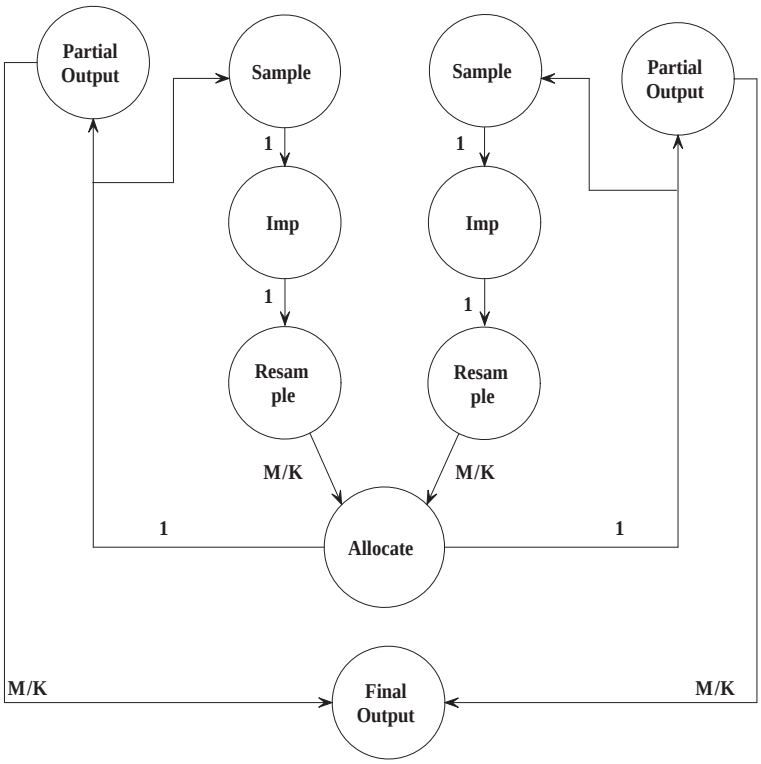


Figure 3.4: Dataflow graph for the SIRF.

Figure 3.4(a) gives a high level picture of the SIRF (with two PEs) data flow for a single recursion. It shows how a particle passes through the steps of sample, importance computation, calculation of sum of weights, output calculation and resampling. The modified SIRF algorithm uses loop fusion to perform sampling, importance computation, accumulation of sum of weights, and output calculation in the same loop. Hence these operations can be easily pipelined. The output generation rate of each of these blocks is one per cycle. Hence appropriate buffers can be placed between them to bring about pipelined execution at the block level. Each of these blocks in turn includes complex computational units and are internally pipelined. This is the concept of two-level pipelining which we shall analyze in a later chapter.

Once the weights of all the particles have been calculated, the resampling can start. Resampling is represented using two nodes, resample and allocate. In the Resample node, systematic resampling is performed and indices of particles after resampling are returned.

The Allocate step reads appropriate particles from the memories using these indices and propagates the resampled set of particles. The availability of this data triggers the output calculation step and the sample step of the next time instant. With loop fusion and pipelining, the effective latency of sample, importance and cumulative sum of weights calculation is 1.

Each of the K processing element processes M/K particles. In a parallel resampling scheme, the central unit collects the partial sum of weights of each PE and determines, via a step of CU resampling, the number of particles each PE should produce after resampling. The weights of all the particles are combined to form the cumulative sum of the entire set of the particles. This information is returned to the PEs, where the resample their own particle set in parallel. The resampled particles are then propagated over the interconnection network which is explained in later chapters.

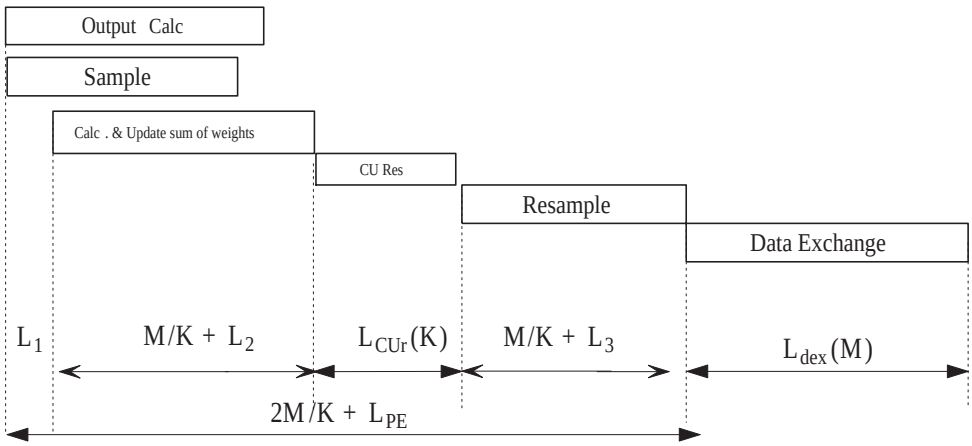


Figure 3.5: Block level timing of the SIRF.

Each PE processes M/K particles. Thus, processing of particles in the PE requires M/K cycles, excluding start up latency. CU resampling takes K cycles and resampling in the PEs takes M/K or $2M/K$ cycles depending upon the algorithm used for resampling. Thus the total time required for completing one recursion of the SIRF is given by $(2 \cdot M/K + L_{SIRF})$ cycles, where L_{SIRF} is the sum of the start up latencies of the various processing units. Figure 3.5 shows the block level timing diagram of the SIRF.

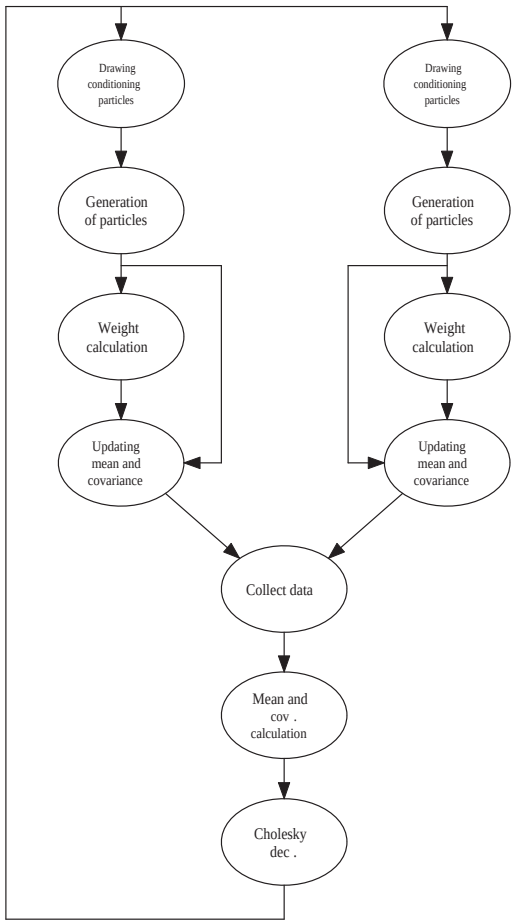


Figure 3.6: Dataflow graph for the GPF.

3.5.2 Dataflow for GPF

The dataflow graph of a GPF is shown in Figure 3.6. The branches indicate the data dependencies and the data generation rate of the various blocks. The Cholesky Decomposition can start only after processing of all blocks is complete. Calculating the final mean and covariance estimate of requires the full sum of weights. Hence this step cannot begin until the weights of all particles have been calculated.Cholesky decomposition is performed in the central unit after combining the partial results of the individual PEs.

The block level timing diagram for the GPF is shown in figure 3.7. It can be seen that the sample period of the GPF is $(M/K + L_{GPF} + L_{CU})$ cycles, where L_{GPF} is the sum of the start up latencies of the processing units and L_{CU} is the latency of the central unit. This unit adds the partial estimates of the mean and covariance and scales these by the sum of

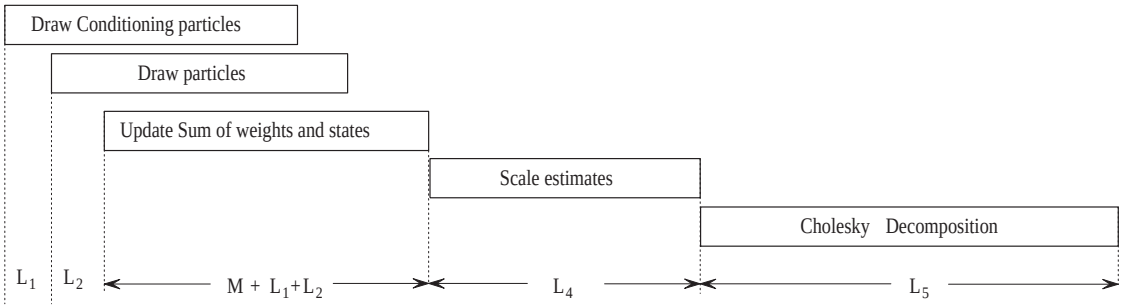


Figure 3.7: Block level timing of GPF.

weights. This seemingly simple step is mathematically complex in practical problems like the 4 dimensional BOT.The final estimate of the covariance matrix is passed onto the Cholesky Decomposition block. This gives a lower diagonal 4×4 decomposed matrix which is used in the sample step of the next recursion. Thus in a parallel implementation of the GPF, the only communication between the PEs and the central unit is exchange of partial estimates of mean and covariance and Cholesky decomposed matrix. Not only is the communication in GPF much less than the SIRF, but it is also completely *deterministic*. This makes for easy and fast design of interconnection networks. It ensures that communication is not the bottleneck in the operation of the parallel GPF. This greatly increases the scalability of the filter since adding more PEs will surely speed up the execution. The drawback of the GPF architecture is the complexity of the individual nodes in the dataflow.

3.6 Design Space Exploration

Design space exploration is the process of mapping a dataflow description to various points in the architectural with the aim of estimating implementation costs and tradeoffs. This enables the designer to easily choose an architecture to meet desired specifications. The range of the design space for the particle filter dataflow can be explored using a methodology based on combination of various works like [13, 11, 12, 39, 36]. Considering each node of the dataflow as a block of computation, the design space can be explored by varying different architectural parameters. Some of these are:

1. Type of Architecture : Sequential, parallel, multiplexed, programmable.

2. Component selection : This decides nature of computational blocks i.e. are they serial, parallel or pipelined, their word length and so on.

3. Pipelining : The depth of the pipeline at the block level. The depth of the pipeline of the units inside the computational blocks will be decided by the component selection.

The cost and performance characteristics are then estimated and compared against desired specifications. Some of the common specifications that are placed on designs are speed requirement, cost requirement(resources or area) and power requirement. For example parallel architectures though faster are often more expensive. For the PF, the degree of parallelism can be increased by adding more resources. Thus if we consider unlimited availability of resources, we can exploit maximum parallelism (i.e have M PEs, each processing a single particle). We initially target an FPGA platform for the implementation. Hence we assume that within the limits of a chosen device, we can use any amount of resources without any additional cost. We use this criterion and then design our architecture for *maximum throughput*. Since this is a data driven operation, the speed will depend on the number of data samples that can be processed in unit time or in other words, the rate of generation of output estimates. Among the parameters mentioned above, our level of pipelining is optimized for maximum throughput. Our block size is fixed to one algorithmic operation per block since this exploits maximum amount of regularity and concurrency. We chose our components to be maximally pipelined, since each of the processing blocks is on the critical path.

We explore the design space for this spatial implementation by varying the number of PEs or in more general terms, varying the degree of parallelism. The speed and cost of the resulting architecture is estimated using the specifications of the basic building blocks from the standard IP core repositories like the Xilinx Core Generator [109]. A 20% overhead was added to the speed and area estimates. Figure 3.8 shows the results of this exploration.

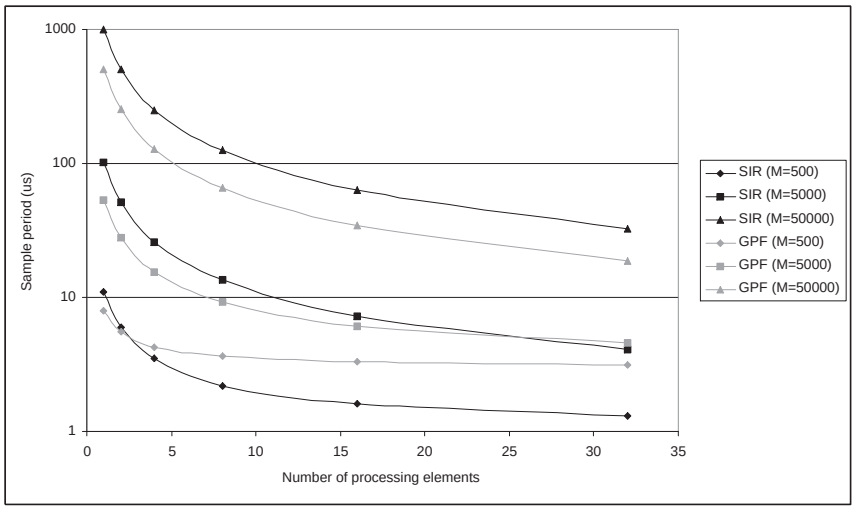


Figure 3.8: Results of DSE for SIRF and GPF obtained by varying the degree of parallelism for various number of particles.

Chapter 4

Architecture and FPGA Implementation of the SIRF

4.1 Introduction

The SIRF is the most frequently used PF in practice. We have seen that the SIRF works by representing the posterior using a random measure composed of samples or particles drawn from an Importance Function and associated weights. In this chapter, we deal with the hardware implementation of the SIRF. We continue from the data flow description presented in the previous chapter. We propose and evaluate architectures and memory schemes for SIRFs. In addition, we also develop the structure of each processing block of the SIRF and investigate the important issue of fixed point representation for various variables for the 4 dimensional BOT problem.

In developing the architecture, a distinction is made between generic operations and model dependent operations of the SIRF. The goal is provide a generic architecture which can be used for any problem with minimal effort in designing the model dependent steps. The proposed architecture can be used as a template to realize an SIRF applied to any model. All the block level control is incorporated into the proposed architecture. The model dependent operations involve mathematical computations on and can be designed independently and incorporated into the overall architecture. Two important algorithmic parameters that significantly affect the size and scalability of the architecture are the the

nubmer of particles used and the dimension of the state space. From here on, we shall refer to the number of particles used as M and the dimension of the state as N_s.

The main drawback of SIRFs is their computational complexity. From the dataflow model, we see that for each observation received, all the M particles need to be processed through the steps of sampling, importance computation and resampling. The sampling and importance computation steps typically involve transcendental exponential and nonlinear operations. Once all the M particles have been processed through the above mentioned steps, the estimate of the state at the sampling instant is calculated and the next input can be processed. These operations present significant computational load even on a state of the art DSP. The performance of the SIRF for the BOT problem described in Chapter 2 with $N_s = 4$ was evaluated on a TI TMS320C54x generation DSP [101]. With $M = 1000$ particles, the inputs to the filter could be processed at the rate of only 1kHz. Clearly this speed would prevent the use of PFs for online signal processing in real time applications where higher sampling rates and/or higher number of particles are needed for processing. Thus, design of dedicated hardware for the SIRF is needed if real time applications are to become feasible.

SIRF is a recursive algorithm. The sampling step uses the resasmpled particles of the previous instant to compute the particles of the current instant. This requires the particles to be stored in memories. We shall see later, that a straightforward implementation of the traditional SIRF algorithm would have a memory requirement of $2N_s \times M$ since the sampled and resampled particles need to be stored in different memories. Most practical applications involve nonlinear models and high dimensional states (large N_s) which implies a large number of particles M for SIRFs applied to these problems [26]. This would make the total memory requirement of $2N_s \times M$ very large. The architectures proposed in this paper, reduce this memory requirement to N_s memories of depth M (i.e. $N_s \times M$). This not only reduces the hardware resource requirement of the SIRF but also makes it more energy efficient due to reduced memory accesses [86].

The specifics of an SIRF implementation depend upon the properties of the model to which the SIRF is applied. However, from a hardware viewpoint, the high level data flow and control structure remains the same for every model. This chapter first describes the developement of efficient architectures for these generic operations of the SIRF. They include

the resampling step and memory related operations of the sample step. The other model dependent operations are data driven and involve mathematical computations. They can be easily incorporated into the proposed architectures for any model. Later in the chapter, we describe the implementation of the model dependent operations for the BOT problem along with the important fixed point analysis and representation scheme.

We develop two architectures, one using the traditional systematic resampling (SR) algorithm and the other using the new residual systematic resampling (RSR) algorithm. These architectures are referred to as Scheme 1 and Scheme 2 respectively. We have seen that the resampling operation in the SIRFs presents a bottleneck since it is inherantly sequential and also cannot be executed concurrently (pipelined) with other operations. Scheme 1 has a low complexity and simple control structure, but is generically slow since SR involves a *while* loop inside an outer *for* loop. As opposed to this, the RSR algorithm has a single *for* loop and hence scheme 2 is faster than scheme 1. We also propose modifications of these schemes which bring about partial parallelization of resampling and reduce the effect of the resampling bottleneck on the execution throghput.

4.2 Resampling Operation in SIRFs

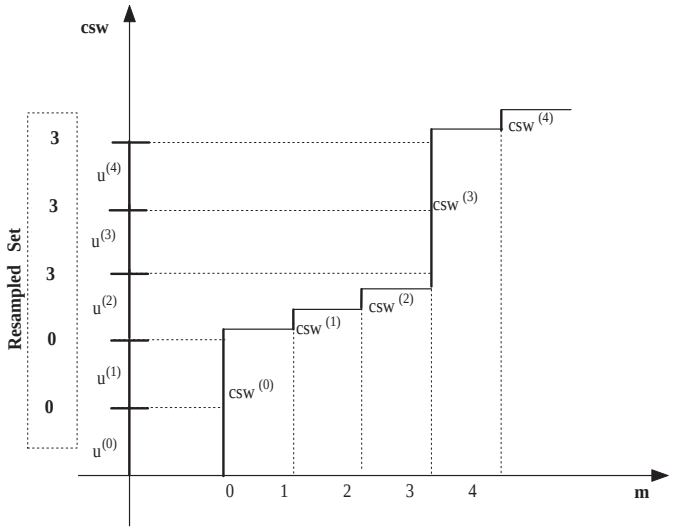
4.2.1 Systematic Resampling

The first architecture proposed in this paper uses the systematic resampling algorithm. This is the most commonly used resampling algorithm for PFs [30]. This algorithm functions by resampling with replacement from the original set of particles $\{\mathbf{x}_n^{(m)}\}_{m=0}^{M-1}$ to obtain a new set $\{\widetilde{\mathbf{x}}_n^{(m)}\}_{m=0}^{M-1}$, where resampling is carried out according to

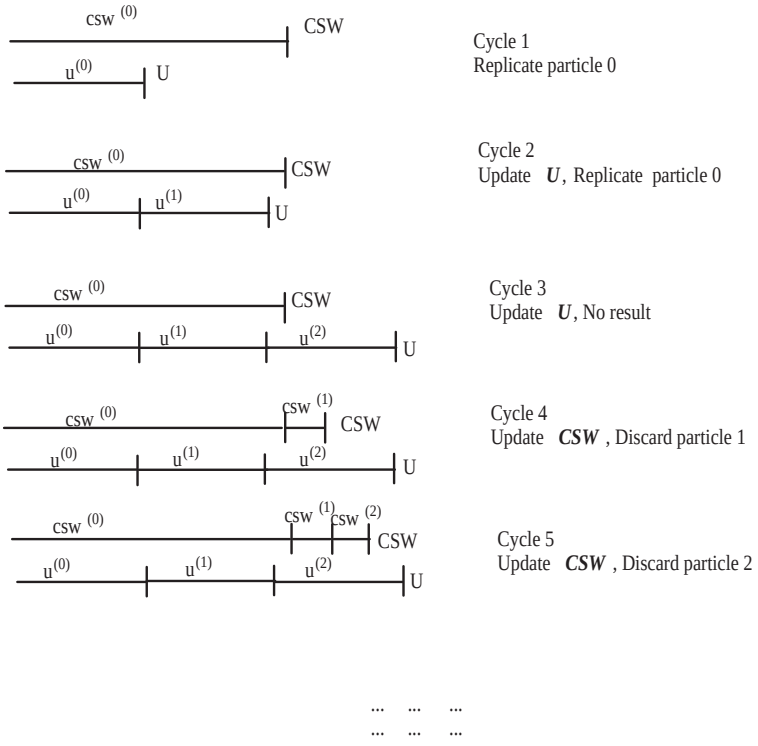
$$Pr(\widetilde{\mathbf{x}}_n^{(i)} = \mathbf{x}_n^{(j)}) = w_n^{(j)}$$

In other words, the particles drawn in the sample step and their weights form a distribution. The Resampled particles are drawn proportional to this distribution to replace the original set. The normalised weights of all resampled particles are set to $1/M$.

The SR concept for a PF that used 5 particles is shown in Fig. 4.1(a). First the cumulative sum of weights (CSW) of sampled particles is computed. This is presented in the figure for



(a) Resampling using cdfs



(b) Resampling done sequentially

Figure 4.1: The concept of systematic resampling.

the case of 5 particles ($M = 5$) with weights $w^{(0)}...w^{(4)}$. Then, as shown on the y axis of the graph, a function $u^{(m)}$ called the resampling function is systematically updated [30] and compared with the CSW of the particles. The corresponding particles are replicated to form the resampled set which for this case is $\{\mathbf{x}^{(0)}, \mathbf{x}^{(0)}, \mathbf{x}^{(3)}, \mathbf{x}^{(3)}, \mathbf{x}^{(3)}\}$. In the traditional SR algorithm, it is essential for the weights to be *normalized* such that their sum is one. However we use a modified resampling algorithm that avoids weight normalization by incorporating the sum of weights into the resampling operation as explained in Chapter 3. This avoids M divisions per SIRF recursion which is very advantageous for hardware implementation.

The determination of the resampled set of particles is done sequentially as is shown in Fig. 4.1(b). In each cycle, depending on the results of comparison between the two numbers U and CSW, which represent the current value of the resampling function and the CSW respectively, the relevant particle is replicated or discarded and the value of either the resampling function U or the cumulative sum of weights CSW is updated. As shown in the figure, in the first cycle, $u^{(0)}$ and $csw^{(0)}$ are compared. Since $CSW > U$, particle 0 is replicated and the *resampling function* is updated, while in cycle 4, since $CSW < U$, particle 1 is discarded and the CSW is updated. This process is repeated till the replicated set of particles is obtained.

As we shall see later, the SR algorithm needs $2M - 1$ cycles for execution in hardware.

4.2.2 Residual-Systematic Resampling Algorithm

Inspite of the low hardware complexity, the low speed of the SR algorithm may not be tolerable in case of high speed applications. For these cases, the residual systematic resampling (RSR) algorithm proposed in [17] can be used. This algorithm has a single *for* loop of M iterations and hence is twice faster than SR in terms of number of cycles. This algorithm is based on the traditional residual resampling algorithm [71]. In residual resampling (RR) the number of replications of a specific particle $\mathbf{x}^{(m)}$ is determined by truncating the product of the number of particles M and the particle weight $w^{(m)}$. The result is known as a replication factor. The sum of the replication factors of all particles, except for some special cases, is less than M. These remaining particles are obtained from the residues of the truncated products using some other mechanism like systematic resampling or random resampling. RR thus

requires two loops of M iterations: one for processing the truncated products and the other for processing residues. RSR calculates the replication factor of each particle similar to RR but it avoids the second loop of RR by including the processing of the residues by systematic resampling in the same loop. This is done by combining the resampling function U with the truncated product. As a result, this algorithm has only one loop and the processing time is independent of the distribution of the weights at the input. The RSR has an execution time of $M + L_{RSR}$ cycles, where the latency of the RSR datapath L_{RSR} is typically low ($L_{RSR} = 2$ for our implementation). The RSR algorithm for M particles is summarized in Table 4.1.

$(r) = RSR(M, w)$
1. Generate a random number $\Delta U^{(0)} \sim \mathcal{U}[0, \frac{1}{M}]$
2. **for** $m = 0$ **to** $M - 1$
3. $\quad r^{(m)} = \lfloor (w_n^{(m)} - \Delta U^{(m-1)}) \cdot M \rfloor + 1$
4. $\quad \Delta U^{(m)} = \Delta U^{(m-1)} + \frac{r^{(m)}}{M} - w_n^{(m)}$
5. **end**

Table 4.1: Residual Systematic Resampling (RSR) for the SIRF.

Fig. 4.2 graphically illustrates the RSR methods for the case of $M = 5$ particles. The RSR algorithm draws the uniform random number $U^{(0)} = \Delta U^{(0)}$ and updates it by $\Delta U^{(m)} = \Delta U^{(m-1)} + \frac{r^{(m)}}{M} - w_n^{(m)}$. The difference $\Delta U^{(m)}$ between the updated uniform number and the current weight is propagated. Fig. 4.2 shows that $r^{(0)} = 2$, ,i.e., particle 0 is replicated twice, $r^{(3)} = 3$ i.e particle 3 is replicated 3 times and all other particles are discarded. SR and RSR produce identical resampling result.

4.3 Architectures and Memory Schemes

We now elaborate on the development of architectures for the SIRF employing each of the two resampling mechanisms discussed in the previous section.

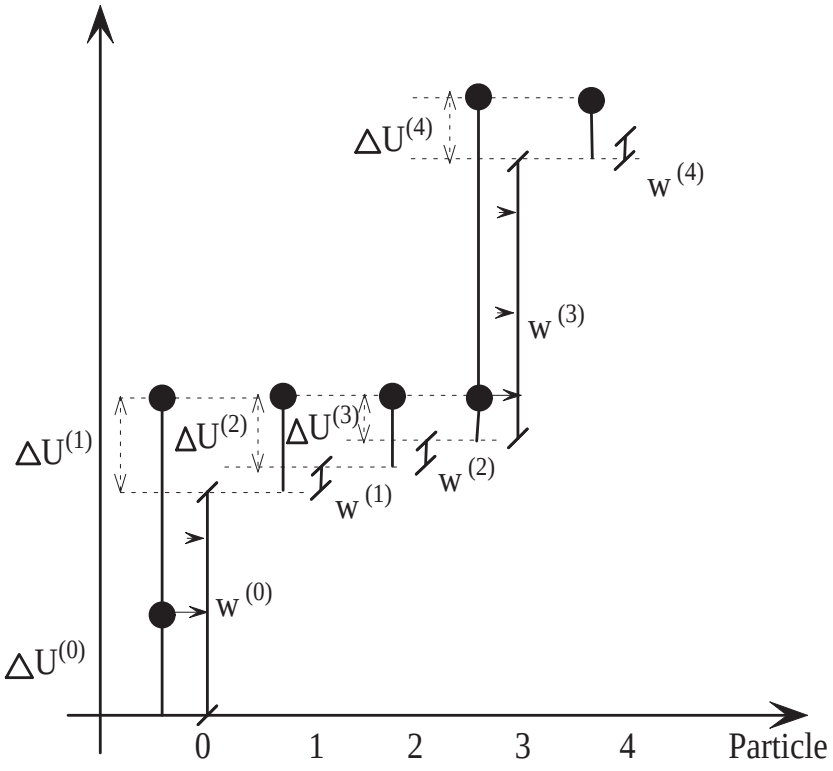


Figure 4.2: Residual-systematic resampling for an example with $M = 5$ particles.

4.3.1 Reduction of memory requirement

In the SIRF algorithm, the sampled particles $\{\mathbf{x}_n^{(m)}\}_{m=0}^{M-1}$ are generated by propagating the previous resampled particles $\{\widetilde{\mathbf{x}}_{n-1}^{(m)}\}_{m=0}^{M-1}$. This is done in the following manner using the DSS model:

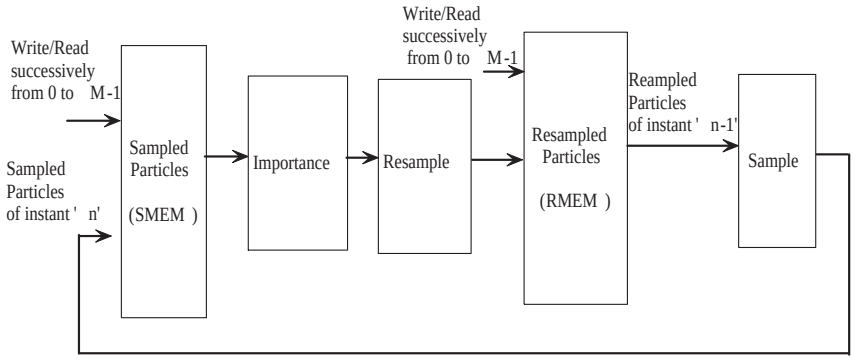
$$\mathbf{x}_n^{(m)} \sim p(\mathbf{x}_n|\widetilde{\mathbf{x}}_{n-1}^{(m)}), m = 0, 1, ...M-1 \tag{4.1}$$

A straightforward implementation of the SIRF would require $2 \times N_s$ memories of depth M, N_s for storing the sampled particles $\{\mathbf{x}_n^{(m)}\}_{m=0}^{M-1}$ and N_s for storing the resampled particles $\{\widetilde{\mathbf{x}}_n^{(m)}\}_{m=0}^{M-1}$. This implementation is shown in Fig. 4.3(a). At time instant n, the sampled particles $\{\mathbf{x}_n^{(m)}\}_{m=0}^{M-1}$ will be stored in the memory labelled $SMEM$. Their weights will be calculated in the importance computation step. Once all the weights have been determined, the resampling unit determines the resampled set of particles $\{\widetilde{\mathbf{x}}_n\}_{m=0}^{M-1}$, which are written to the memory labelled $RMEM$. The sample unit then reads particles from $RMEM$ for propagation. These memories are shown in Figure 4.3(a) for $N_s = 1$.

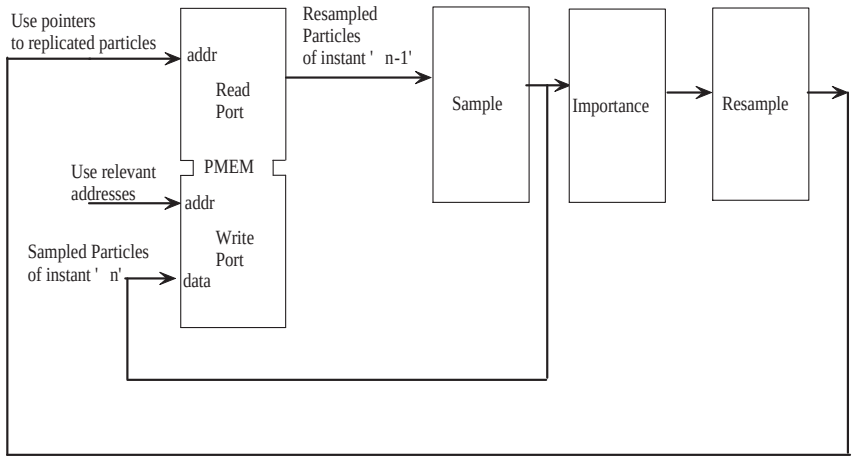
The memory schemes proposed here reduce this requirement to N_s memories of depth M. In our implementation, the resampling unit returns the set of indexes (pointers) of the replicated particles instead of the particles themselves. Then indirect addressing [45] can be used to read the set $\{\widetilde{\mathbf{x}}_n\}_{m=0}^{M-1}$ from the sample memory $SMEM$ itself for propagation. This means that the particles are propagated in the following manner:

$$\mathbf{x}_n^{(m)} \sim p(\mathbf{x}_n|\mathbf{x}_{n-1}^{ind(m)}) \tag{4.2}$$

where $ind(m)$ represents the array of indexes or pointers to the resampled particles. Here we make use of the fact that the resampled particles are in fact a *subset* of the particles in the sampled particles memory. Hence instead of replicating them and storing them in a different memory, they can be read from the same memory by using appropriate pointers. The sampling process involves reading of M resampled particles and writing of M sampled particles to the memory. If a single port memory is used the reads and writes cannot be done simultaneously. This would require that a resampled particle be read, propagated and written to the memory before the next resampled particle can be read. The execution of the sample step would then take $2(M+L_S)$ cycles where L_S is the latency of sample computation.



(a) Resampling requiring two memories



(b) Modified architecture needing only one dual port memory

Figure 4.3: Memories for storing particles. In the traditional implementation two memories would be needed. These are replaced by a single dual port memory.

This execution can be speeded up by using *dual port* memories [23] which are readily available on an FPGA platform[1]. This enables reading of $\{\widetilde{\mathbf{x}}_{n-1}\}_{m=0}^{M-1}$ and writing of $\{\mathbf{x}_n^{(m)}\}_{m=0}^{M-1}$ to be executed concurrently. Hence, the sample step for M particles can be done in $M + L_S$ cycles. The memory scheme is shown in Fig. 4.3(b) where the single dual port memory labelled $PMEM$ replaces the memories $SMEM$ and $RMEM$ of Fig. 4.3(a). Thus, use of index addressing reduces the memory requirement of the SIRF and use of dual port memories reduces the execution cycle time.

[1]We would like to point out here that on an ASIC platform, use of dual port memories incurs a 2x area penalty

However, using index addressing alone does not ensure that the scheme with the single memory will work correctly. We illustrate the reason for this with a simple example.

Consider the following one-dimensional random walk model:

$$x_n = x_{n-1} + q_n \tag{4.3}$$

$$y_n = x_n + v_n \tag{4.4}$$

Here x_n represents the one-dimensional state of the system and y_n is a noisy measurement. The symbols q_n and v_n are the process and the measurement noises respectively. Consider 5 sampled particles at instant $n-1$ (i.e. $\{x_{n-1}^{(m)}\}_{m=0}^{4}$). In the implementation of Fig. 4.3(a), these particles will be stored in the memory $SMEM$ at locations $SMEM[0], ..., SMEM[4]$. Suppose that after resampling, particle $x_{n-1}^{(0)}$ is replicated twice, $x_{n-1}^{(3)}$ three times, and that particles $x_{n-1}^{(1)}$, $x_{n-1}^{(2)}$ and $x_{n-1}^{(4)}$ are discarded. In the implementation with two memories, the resampled particles will be written to memory $RMEM$. The operations performed in the sample step at instant n for this case are shown in Fig. 4.4(a).

$$\begin{aligned}
SMEM[0] &= x_n^{(0)} = RMEM[0] + q_n^{(0)} \\
SMEM[1] &= x_n^{(1)} = RMEM[1] + q_n^{(1)} \\
SMEM[2] &= x_n^{(2)} = RMEM[2] + q_n^{(2)} \\
SMEM[3] &= x_n^{(3)} = RMEM[3] + q_n^{(3)} \\
SMEM[4] &= x_n^{(4)} = RMEM[4] + q_n^{(4)}
\end{aligned}$$

(a) For implementation with two memories

$$\begin{aligned}
x_n^{(0)} &= PMEM[0] + q_n^{(0)} \\
x_n^{(1)} &= PMEM[0] + q_n^{(1)} \\
x_n^{(2)} &= PMEM[3] + q_n^{(2)} \\
x_n^{(3)} &= PMEM[3] + q_n^{(3)} \\
x_n^{(4)} &= PMEM[3] + q_n^{(4)}
\end{aligned}$$

(b) For implementation with one memory

Figure 4.4: Memory operations in sample step

As seen from the figure, the sampled particles are written to the memory $SMEM$. In the reduced memory implementation of Fig. 4.3(b), the replicated particles are read out of the same single memory ($PMEM$ in this case) using resampled indexes. For this example,

the set of indexes of replicated particles is $\{0, 0, 3, 3, 3\}$. Thus the operations of the sample step will be as shown in Fig. 4.4(b). However, if sampled particles are written to successive locations of $PMEM$ as in the previous case, the particle $x_n^{(m)}$ will overwrite the resampled particle $\widetilde{x}_{n-1}^{(m)}$ causing an error if this particle has been replicated multiple times. In the above example, if the particle $x_n^{(0)}$ is written to $PMEM[0]$, then for the next particle, we will get

$$x_n^{(1)} = x_n^{(0)} + q_n^{(1)}$$

which is incorrect. Thus different strategies for writing sampled particles to the memory need to be devised for the reduced memory design to function correctly. In the following sections, we will explain the architectures developed using SR and RSR and how they handle reading and writing of the particle memory.

4.4 SIRF using Systematic Resampling (SR) : Scheme 1

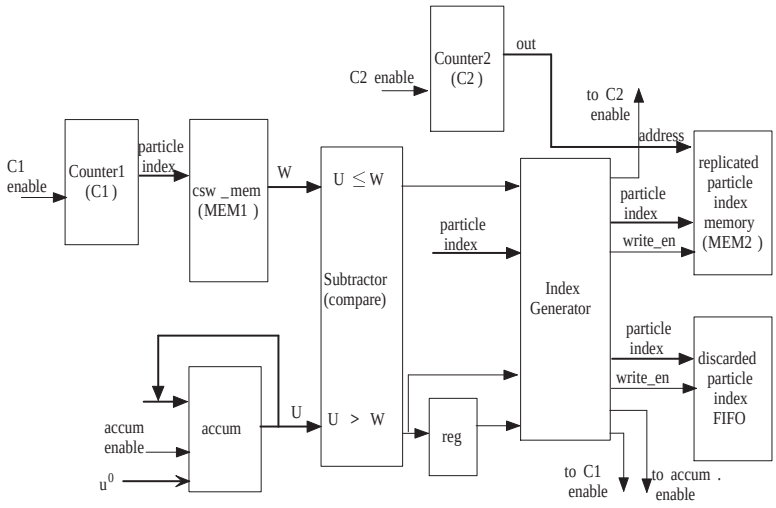


Figure 4.5: Architecture of Resampling Unit implementing SR

Fig. 4.5 shows the architecture for the resampling unit implementing the SR mechanism. The CSW is stored in the memory labelled $MEM1$ at locations corresponding to the ordinal number of the particle in the sampled set. The resampling function $u^{(m)}$ is generated using

an accumulator as shown. Both the memory and the accumulator are controlled by enable signals. The outputs of the accumulator and the memory (U and W) are compared for conditions $U \leq W$ and $U > W$ by using a subtractor. The values of the CSW are read from the memory using one counter (C1). The results of the comparison are passed to the *index generator* unit which determines whether to replicate or discard the particle (i.e., the index of the particle). The indexes of the replicated particles are stored in the memory $MEM2$.

This scheme also records the indexes of the discarded particles. These indexes are used while writing the sampled (propagated) particles back to the memory. A particle which has been generated by a replication, is written to the location of a discarded particle in the memory. The number of particles before and after resampling is the same. This means that for every replicated particle there will be a discarded particle. Hence this scheme can be used effectively for writing particles to the memory. Since the number of particles that will be discarded is non deterministic, we use a FIFO buffer of depth M to store the discarded particle indexes. The output of counter C1 at an instant is the index of the particle that is currently being processed. The comparator and the index generator unit bring about the resampling as in Fig. 4.1. If the particle is replicated, its index is written to $MEM2$ whose locations are addressed by counter $C2$, and the accumulator is enabled so as to update the value of the resampling function. If the particle is discarded or when all its replications are found, counter $C1$ is enabled CSW of the next particle is read from the memory. When an index is to be discarded, the write enable of the FIFO buffer is asserted and the index is written to it.

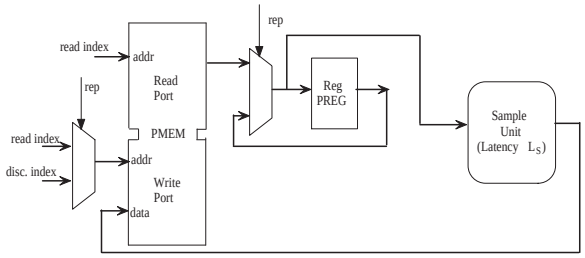
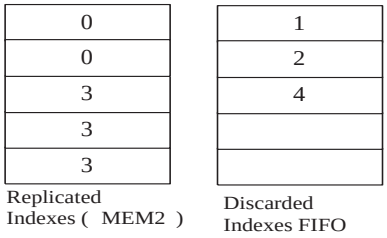


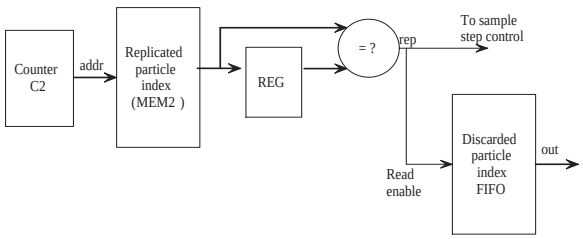
Figure 4.6: Architecture of Sample Unit

Fig. 4.6 shows the architecture for the sample step under this scheme. Once resampling is done, the memory $MEM2$ represents the array $ind(m)$ containing M replicated indexes. This memory is read sequentially and the indexes are used as addresses to the read port of

the dual port memory (PMEM) storing the particles. The output of this memory is the set $\{\widetilde{\mathbf{x}}_n\}_{m=0}^{M-1}$. Due to the nature of the SR algorithm, all the replications of a particular index will be written to successive locations in $MEM2$. Thus, since this memory is read sequentially, a replication can be detected by comparing the current read index with the previous one. When an index is read from $MEM2$ for the first time, the corresponding particle is read from the memory and stored in the temporary register $PREG$. After propagation this particle is written to its original location in the memory. When the same index is read from $MEM2$ in the following cycle, replication is detected, and the particle is read from the temporary register $PREG$ rather than from the memory (since the location in the memory will be overwritten by the propagated particle). Also, the read enable of the FIFO is asserted high and a discarded index is obtained which is used as address to the write port of $PMEM$ to write the replicated particle after propagation 4.7(b). We now further illustrate this scheme with our previous example.



(a) Contents of memory and FIFO after resampling



(b) Reading of replicated and discarded indexes

Figure 4.7: Addressing read and write ports of particle memory using stored indexes

Following the same case of the example, the contents of the replicated index memory

$MEM2$ and discarded index FIFO in Fig. 4.5 will be as shown in Fig. 4.7(a). The sample step starts by reading of the content of $MEM2$. The operations in various cycles are listed in TABLE 4.2. Fig. 4.7(b) shows how the FIFO is read. A replication is detected by comparing the current read index with the previous one. From the index memory contents, we see that in this case a replication will be indicated when $MEM2[1]$, $MEM2[3]$ and $MEM2[4]$ are read. The result of this comparison is used as read enable to the FIFO.

Cycle	Read address	Replication	Read from	Write to
1	0	No	$PMEM[0]$	$PMEM[0]$
2	0	Yes	$PREG$	$PMEM[1]$
3	3	No	$PMEM[3]$	$PMEM[3]$
4	3	Yes	$PREG$	$PMEM[2]$
5	3	Yes	$PREG$	$PMEM[4]$

Table 4.2: Function of sample step. Note that writing of particles is done L_S cycles after they are read where L_S is the latency of the sample unit

SR involves comparison of M values of the CSW with M values of the resampling function. As seen in Fig. 4.1(b), the two functions cannot be updated simultaneously, except when obtaining their initial values at the start of resampling. The result of the comparison in each cycle indicates which function is to be updated. Thus execution of SR requires $2M - 1$ cycles.

4.4.1 Modification of Scheme 1 for reduced execution time

Some properties of the SR algorithm can be used to partially parallelize resampling at the cost of added hardware.

Due to the systematic nature of the resampling function update, the final value of the resampling function is fixed. This value is $u^{(0)} + (M-1)/M$ for traditional SR and $u^{(0)} + S(M-1)/M$ for our implementation of resampling using non normalized weights [18]. Also the final value of the CSW, S, is also known to us. We can use this property of SR to split the resampling shown in Fig. 4.1(a) into two concurrent loops of $M/2$ iterations each. One loop determines the first $M/2$ resampled indexes by comparing $csw^{(0)}$ to $csw^{(M/2-1)}$ with $u^{(0)}$ to $u^{(M/2-1)}$ and the other loop determines the next $M/2$ resampled indexes by

comparing $csw^{(M/2)}$ to $csw^{(M-1)}$ with $u^{(M/2)}$ to $u^{(M-1)}$ From a hardware viewpoint, this would require reading of two values of the CSW simultaneously from the memory which can be accomplished by storing the CSW values ($MEM1$ in Fig. 4.5) in a dual port memory. Also the replicated particle index memory $MEM2$ would need to be dual port and the discarded index FIFO would be replaced by two FIFOs of half the size. All other logic blocks in Fig. 4.5 would be replicated. This would reduce the loop bound [84] of the SIRF recursion and increase its throughput. With this scheme, SR is split up into two parallel loops of $M/2$ iterations each. Execution time of SR is reduced to $2 \times M/2 - 1 = M - 1$ cycles at the cost of added hardware. As an extension of this concept, resampling can be split up into more than 2 loops of simultaneous comparisons due to the systematic update of the resampling function. However this would need more memory blocks and additional hardware. The tradeoff between added hardware and obtained speed is considered in Seciton 6.6.

4.5 SIRF with Residual Systematic Resampling (RSR) : Scheme 2

The second architecture introduced in this paper uses the RSR mechanism for resampling. The RSR has only one loop of M iterations and is faster than the SR. In this scheme too, the replicated particles are written to the locations of the discarded particles in the same dual port particle memory. Unlike scheme 1, after resampling in this scheme the indexes of *all* the particles are stored in one index memory. Another memory is used to store the *corresponding replication factors*. If an index has been discarded, a factor of 0 is recorded at the corresponding location in the replication factors memory. In this scheme, the indexes are arranged in such a way that all replicated indexes are written to the memory starting from location 0 up, while all discarded indexes are written to locations from $M - 1$ down. This method of storing indexes and replication factors is called *particle allocation with arranged indexes*.

Memory usage for the example described in Section 4.3.1 is shown in Fig. 4.8, where the indexes are arranged in the memory using the above mentioned method and the correspond-

ing replication factors are stored in a separate memory.

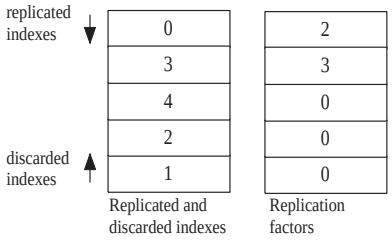


Figure 4.8: Contents of memories after the RSR method with particle allocation with arranged indexes.

From an implementation viewpoint, the RSR algorithm is benificial since it has a single *for* loop. To make the RSR algorithm suitable for implementation, we make some changes in Pseudocode 2 in Section 4.2.2. The changes incorporate particle allocation with arranged indexes in the algorithm and also allow for resampling using *non-normalized* weights. As in the case of SR this saves M divisions at each instant.

The modified RSR algorithm is shown in Table 4.3.

$(i, r) = RSR(M, S, w)$
1. Generate a random number $U \sim \mathcal{U}[0, 1]$
2. $K = M/S$
3. $ind_r = 0$, $ind_d = M - 1$
4. *for* $m = 1$ *to* M
5. $temp = w_n^{(m)} \cdot K - U$ // Temporary variable
6. $fact = \lceil temp \rceil$
7. $U = fact - temp$
8. *if* $fact > 0$ // Particle allocation
9. $i(ind_r) = m$, $r(ind_r) = fact$, $ind_r = ind_r + 1$
10. *else*
11. $i(ind_d) = m$, $r(ind_d) = 0$, $ind_d = ind_d - 1$
12. *end*
13. *end*

Table 4.3: Modified Residual systematic resampling (RSR) algorithm.

In the pseudocode in Table 4.3 there is one multiplication inside the loop and one division before the loop. The incorporation of the number K and generation of U from $\mathcal{U}[0, 1]$ is done to allow non normalized weights in the resampling algorithm. These changes do not affect

the correctness of the algorithm and the resampling results produced are same as Pseudocode 2.

Lines 9 and 11 bring about particle allocation with arranged indexes by writing replicated and discarded indexes to the top and bottom parts of the index memory respectively.

The memory related operations in the sample step are shown in Table 4.4. First, the replicated indexes are read sequentially from the memory of arranged indexes as shown in the first *for* loop (line 2). The corresponding replication factors of the indexes are also read at the same time. If the particle has been replicated, then it is propagated repeatedly. This is shown by the second *for* loop (line 5) whose iterations equal the replication factor for that particular index. Then, $r^{(ind_r)} - 1$ sampled particles are written to the addresses of the discarded particles (line 6) by reading the arranged index memory from the bottom. The first sampled particle rewrites the original replicated particle (line 3). Hence the replicated particle has to be stored in a variable *Reg*.

```
    (X) = Sampling(i, r, X)
1.      ind_r = 0, ind_d = M − 1
2.      for ind_r = 1 to length(ind_r)
3.          Reg = X(i(ind_r))
4.          X(i(ind_r)) = Sample(Reg), ind_r = ind_r + 1
5.          for k = r(ind_r) − 1 down to 1
6.              X(i(ind_d)) = Sample(Reg), ind_d = ind_d − 1
7.          end
8.      end
```

Table 4.4: Memory related operations of the sample step.

4.5.1 Architecture for Scheme 2

In this section, the architectures for the algorithms presented in Tables 4.3 4.4 are shown in Figure 4.9 and Figure 4.10. In Fig. 4.9, weights are stored in the memory MEM_w and addressed by the address counter that counts from 0 to $M-1$ and corresponds to the variable m. The index generator is the block in which the arithmetics from the lines 5,6 and 7 in Table 4.3 are implemented. The other part of the figure represents the implementation of the particle allocation step (lines 8 - 12 of Table 4.3). MEM_i stores the arranged indexes

and MEM_r stores the corresponding replication factors. Depending on whether a particle is replicated or not, its index is written to MEM_i at address pointed to by either the counter counting up ($counter_r$)or down ($counter_d$). The appropriate replication factor is written to the corresponding location in MEM_r.

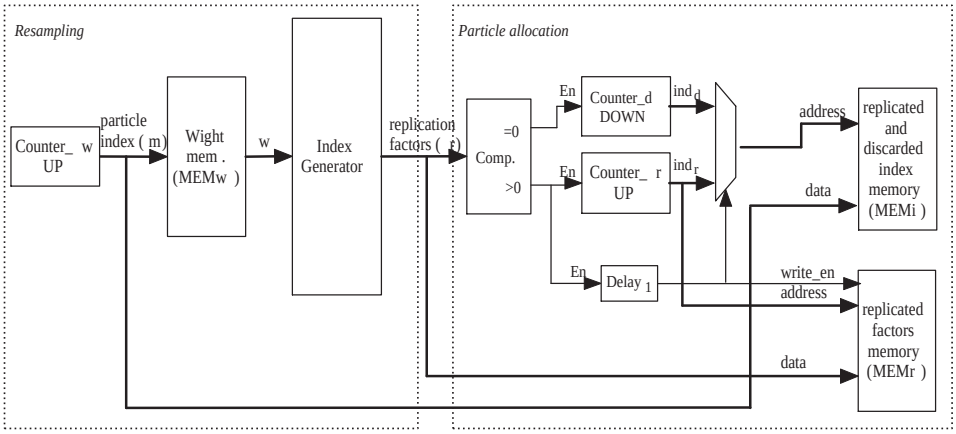


Figure 4.9: The architecture for the RSR algorithm combined with the particle propagation.

There are three main blocks in Fig. 4.10: address generation, address control, and particle generation and storing. One dual port memory $PMEM$ is used for storing particles. The arithmetics of the sampling step is implemented in the Sampling Unit. The delay between read and write operations for the memory $PMEM$ is determined by the pipeline latency of the Sample Unit (L_S). It is presented as $Delay_1$ in the figure. $Counter_f$ represents the variable k in Table 4.4. The replication from the memory MEM_r is used as the initial value to the down counter $Counter_f$. The other logic blocks are for generation of controls to bring about sampling as described in Table 4.4.

4.5.2 Modification of scheme 2 for reduced execution time

The RSR algorithm needs $M + L_{RSR}$ cycles for execution where the latency due to pipelining of the RSR datapath is L_{RSR}(2 in our case). Similar to the SR, the RSR algorithm can also be parallelized with addition of more hardware for reduced execution time. The RSR algorithm used for scheme 2 has only one loop in which the replication factor of a particle is determined and the value of the resampling function is systematically updated. This algorithm can also be modified for parallel execution by splitting the resampling process

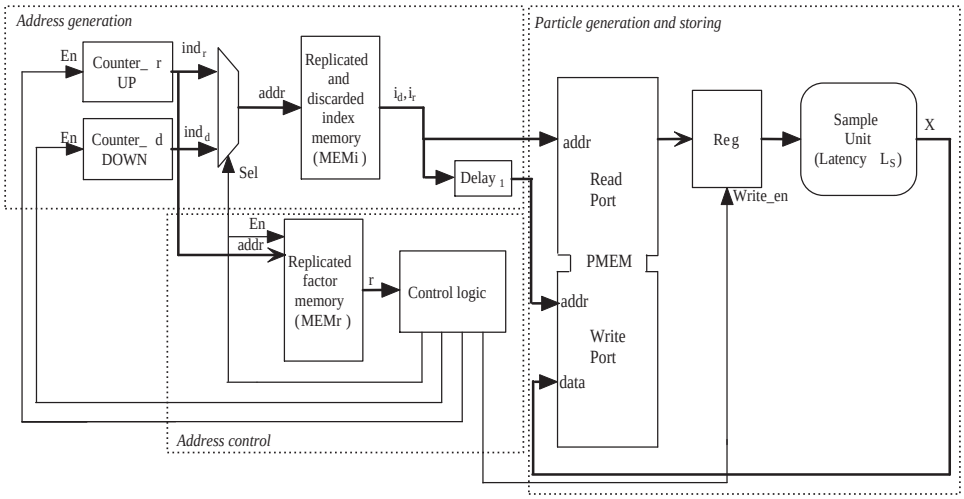


Figure 4.10: The architecture for memory related operations of the sampling step.

into multiple concurrent loops. The modified algorithm for 2 concurrent loops is shown in Pseudocode 5. The first loop does the usual RSR of Pseudocode 3 for the first $M/2$ particles from index 0 to $M/2-1$. The second loop does the same simultaneously for the remaining particles from index $M/2$ to M. This algorithm needs the cumulative sum of weights of the first $M/2$ particles. This is denoted as $S^{M/2}$. The initial value of the resampling function for the second loop is denoted by U^2 in the Pseudocode. Once again the factor K is included so as to allow resampling using non-normalized weights. This mechanism can also be directly extended to include more than two loops at the cost of adding more memory and hardware.

The execution time is thus reduced to $(M/2)+2+L_1$ cycles where the additional latency L_1 is introduced by the computation of $r^{M/2-1}$ and U^2 before the second loop can start.

4.6 Implementation of Gussian Noise Generator

The architecture and memory schemes described so far are generic and can be used for the implementation of the SIRF applied to any model. The computation operations of the sample step and the importance step are specific to the model under consideration. We now describe the design of these blocks for the SIRF applied to the BOT problem. This involves Gaussian noise generation which is also fairly generic since many problems involve Gaussian states.

Method: (i,r) $= RSR(M, S, S^{M/2}, w)$

1. Generate a random number $U^1 \sim \mathcal{U}[0,1]$
2. $K = M/S$
3. $r^{M/2-1} = \lceil (S^{M/2} - U^1)K \rceil$
4. $U^2 = U^1 + r^{M/2} - S^{M/2} \cdot K$

Loop 1

Initialize $ind_r^1 = 0$,
$ind_d^1 = M/2 - 1$
for $m = 0$ **to** $M/2 - 1$
  Perform steps 5 - 12
  in Table 4.3
end

Loop 2

Initialize $ind_r^2 = M/2$,
$ind_r^2 = M - 1$
for $m = M/2$ **to** $M - 1$
  Perform steps 5 - 12
  in Table 4.3
end

Table 4.5: Split-Loop implementation of the RSR algorithm for parallel implementation.

Most theoretical particle filter formulations, particularly for models involving Gaussian noise are based on the assumption of ideal (iid) white Gaussian noise (WGN) samples. However in the VLSI implementation, ideal WGN generation is not trivial. A large number of methods for generation of Gaussian noise have been proposed in the literature. Most of these start by using the uniform random numbers generated by a Linear Feedback Shift Register (LFSR) [85],[104],[74],[2]. These uniform numbers are then transformed into Gaussian variates by using transformations like the Box Muller method [62] or IIR filtering (Central Limit Theorem) [91]. Better quality random numbers can be generated using cellular automata instead of the LFSR [108],[94]. A detailed description on testing of noise generators can be found in [75]. We have presented guidelines for the choice of suitable noise generators for particle filtering in [8].

We use an implementation based on the Quantized look up table approach outlined in [25, 20]. In this method, the quantized values of the two Box Muller variables are stored in ROM look up tables. These values are accessed using an LFSR. Figure 4.11(a) shows the basic implementation of the noise generator from [25]. A non uniform quantization on the first half Box Muller variable, $\sqrt{-ln(x_1)}$ where x_1 is a uniform random number, is performed. This is done due to the steep characteristic of the $ln(\cdot)$ function around zero. Accordingly,

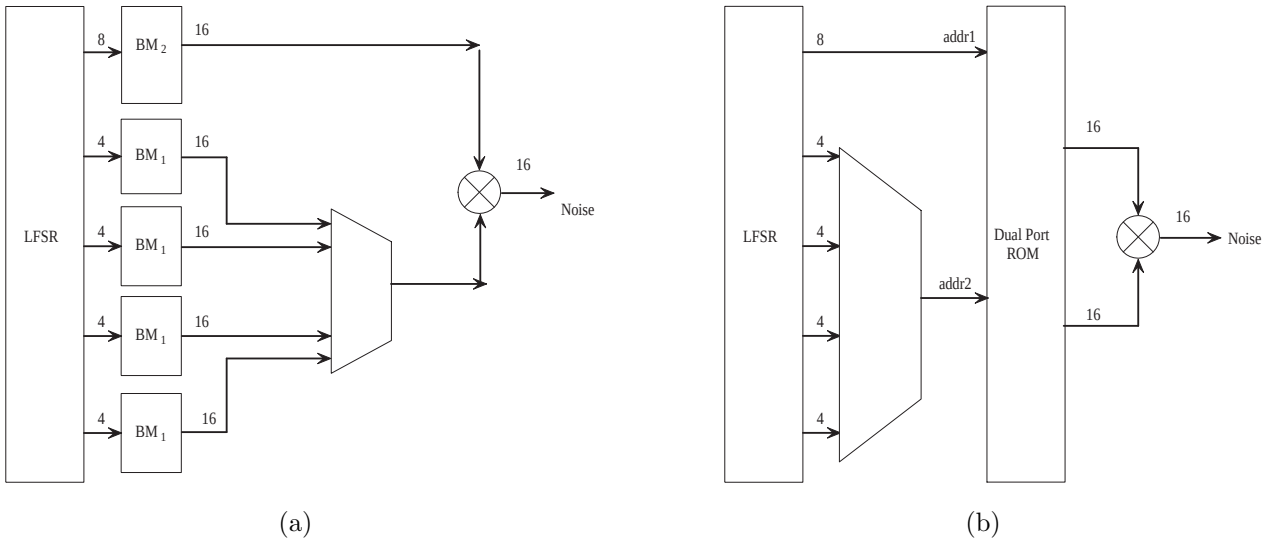


Figure 4.11: (a)Implementation of noise generator presented in [25]. (b) Architecture modified for efficient FPGA utilization.

smaller values are quantized using a smaller step. As the value of x_1 becomes smaller, the quantization step becomes smaller. Thus, based on the output of the LFSR, one of the ROMS storing quantized values of $\sqrt{-ln(x_1)}$ is addressed. The other ROM stores the other variable $\sqrt{2}cos(2\pi x_2)$. The two quantized half Box Muller variables are multiplied to give the $\mathcal{N}(0,1)$ distributed variable.

We modify the basic architecture of this noise generator to optimally use the FPGA resources. Firstly we combine all the 5 ROMS in the above 4.11(a) into a *single* dual port memory block. In figure 4.11(a) we see that the values at the outputs of the memories are fed to the multiplexer. In our design, we choose one of 4 possible *addresses* rather than data. This not only enables the use of a single memory block, but also eliminates the combinational logic between the multiplier and the memory which allows mapping to *adjacent memory and multiplier blocks* which in turn results in significantly reducing interconnect area (considering the number of noise generators that are needed in the particle filter design). Figure 4.11(b) shows the diagram of the modified noise generator implementation. Bits of appropriate order are masked by making them high or low, when accessing the memories. The output of the LFSR is split into two numbers as seen in the figure. Each of these with appropriate bit masking is used to read from one of the ports of the memory. The numbers are stored in the

ROM in fixed point format.

The noise generator is described using Verilog HDL. The look up tables are generated using the Xilinx Memory Editor. The resulting design is simulated using modelsim and then it is synthesized and put onto the device. Using the generated timing model, we back annotate the delays into the simulation and carry out a post place and route simulation. This helps us to know if the functionality of the model will be affected due to the delays. Table 4.6 shows the utilization of various resources on the FPGA by the noise generator module. The generated noise samples are analyzed using MATLAB. Figure 4.6 shows some of the statistics of the generated noise samples. These samples also pass the Kolmogorov-Smirnov test for Gaussianity.

Net maximum delay through module = 4.194 ns

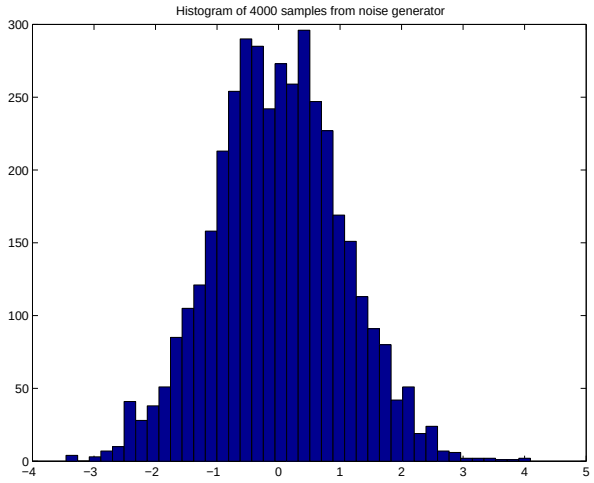
Resource	Number	Utilization %
External IOBs	43	4%
Mult Blocks	1	1%
RAM Blocks	1	1 %
Slices	56	1%
BUFMUXGs	1	16%

Table 4.6: Resource Utilization of noise generator for the device Xilinx Virtex II Pro 125 (XCV2P125), package ff1704, speed grade -6

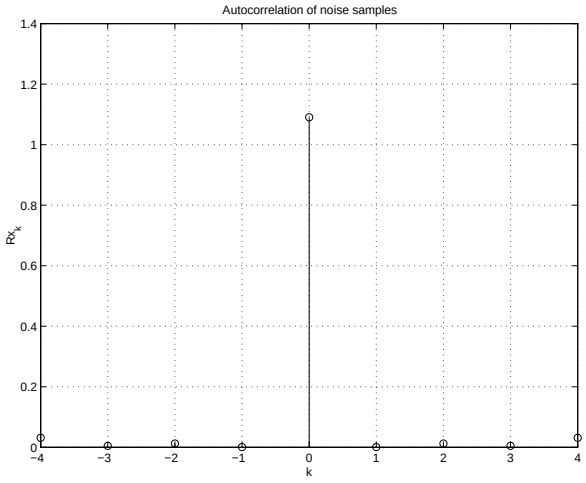
4.7 Implementation of the Sample and Importance computation steps

The generic architecture proposed for the SIRF includes a single memory for storing particles and a control structure for correctly reading and writing particles. The implementation of the Gaussian noise generator has been described in Section 4.6. Using these, the sample step for the BOT problem is implemented simply with a set of pipelined adders.

The block diagram of the whole Importance step is shown in Fig. 4.13. The first instance of the unrolled CORDIC core computes $\tan^{-1}(y/x)$. This produces an output in the range $[-\pi, \pi]$ [113]. The $\tan^{-1}()$ function has unique values in the range $[-\pi/2, \pi/2]$. Hence additional logic is needed to convert the output of the CORDIC core to this range. We observe



(a)



(b)

Figure 4.12: (a)Histogram for noise samples from the implemented noise generator (b)Autocorrelation upto lag of 4 for the noise samples.

that the value of the weight w reduces to 0 if the variable A shown in Fig. 4.13 is greater than 255. We use this fact to reduce the area and resources required by the importance step. Accordingly, only 8 LSBs of A are propagated through the weight computation logic. This leads to a loss in accuracy of at most 10%. The 8 MSBs are compared to a constant, and if A is greater than 255, the value 0 is selected as the output weight via a multiplexer. The delay units are added for synchronization due to the fine grain pipelining of the CORDIC units. The weights are summed using an accumulator as shown to produce the required sum of weights.

The implementation of the $exp()$ function is shown in Fig. 4.14. The input range of the CORDIC core used for the $exp()$ function is restricted to $[-\pi/4, \pi/4]$ [113]. Hence we split the input exponent into an integer and a fractional part based on the fixed point format used. The $exp()$ of the integer part is precalculated and stored in a ROM look up table. The $exp()$ of the fractional part is calculated using the CORDIC unit as shown and the two numbers are then multiplied for the final result.

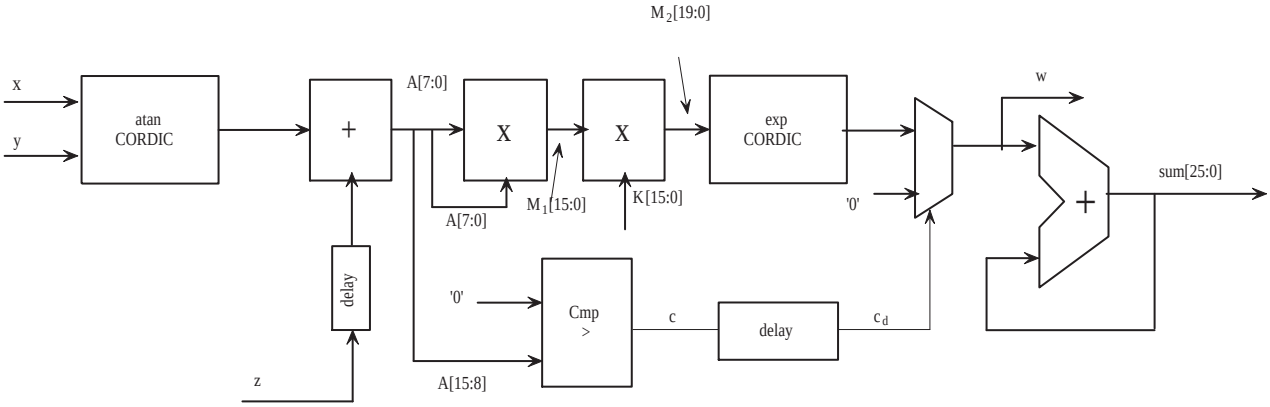


Figure 4.13: Block Diagram of Importance Step

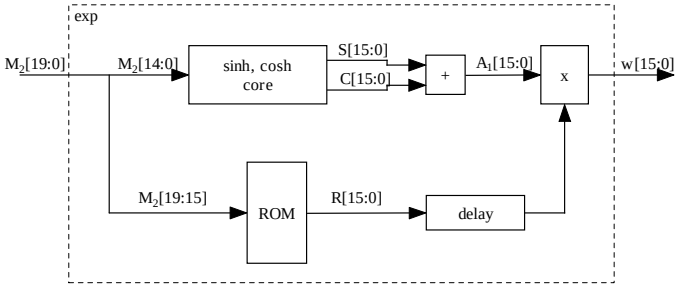


Figure 4.14: Block Diagram of implementation of *exp*() function.

4.8 Fixed point analysis for the SIRF

Fixed point analysis is an important issue in the implementation of data intensive algorithms like the SIRF. The numbers that the SIRF deals with are largely dependent upon the model and hence model parameters are used to drive the fixed point analysis. The SIRF involves complex transcendental and exponential operations in its execution. In software, all the numbers are represented and calculations are done using double precision floating point types. For efficiency of hardware implementation, the floating point types are realized as fixed point types. Fixed point numbers exist in registers with an imaginary (binary) point at a suitable location. The part to the left of the binary point is interpreted as the integer part of the number and the part to the right is interpreted as the fractional part. Implementing fixed point types involves a detailed analysis of all the variables involved in the algorithm and the effect that non ideal operations like truncation and rounding have on the filter performance.

For algorithms like the SIRF, where variables are random rather than deterministic, a statistical analysis over many realizations is needed to appropriately define the format needed for representing the variable. Various approaches have been suggested for this fixed point analysis [22, 58, 33, 82]. Most of them use some kind of statistical analysis based on a simulated dynamic range of each floating point variable. Often, floating to fixed point conversion is formulated as an optimization problem where the hardware cost is to be minimized while maintaining a specified system performance. A good overview of such techniques can be found in [95].

4.8.1 Properties of Fixed Point Numbers

We adopt a generalized format for the representation of fixed point numbers [59], [98]. In this format a fixed point type consists of a sign bit, integer part to the left of the imaginary binary point and a fractional part to the right of the point. Truncation occurs when the number of bits in the fractional part of the fixed point number is less then that required to represent the result. Overflow occurs when the output of an operation generates more bits on the MSB side than are available for representation. To model the effects of this, we can use several quantization and overflow modes. This allows us to easily simulate the behavior of the fixed point types in various modes of quantization and overflow. The characteristics associated with a fixed point declaration are

- Word length : Total length including sign bit
- Integer Word Length : Number of bits to left of binary point
- Quantization Mode : Truncation, rounding, etc
- Overflow Mode : Saturation, Wrap around, etc

In hardware operations on fixed point numbers are carried out like integer operations. The binary point in the number is purely imaginary and it is up to the architecture to perform appropriate scaling for intermediate operations and interpret the final result.

4.8.2 Method of Fixed Point Analysis

We use a statistical analysis method suggested in [58] utilizing the C++ concept of operator overloading [97], [93] along with SystemC fixed point types to analyze and evaluate the fixed point requirements of various variables. The advantage of using SystemC fixed point types is that different characteristics can be easily associated with the fixed point type declaration. Accordingly we define a class for monitoring the word length and integer word length of each variable. The maximum and minimum values, mean and variance of the word length and integer word length of each variable are recorded for each run of the SIRF. At each instant, statistics are collected over all the processed particles. To do this analysis, we start initially with the regular particle filter operating in floating point precision. When the value of a particular variable is calculated in floating point precision, it is cast into the systemC fixed point type with some specified format. The number of bits in this fixed point type are modified until the difference between the floating point value and its fixed point representation is within some predefined threshold. Then this information about the word format is assigned to an object of the monitor class for which the = operator is overloaded to update the statistics of the word length for that variable. Figure 4.15 shows the member functions and variables for the class used.

```
class fix_form
{
public:
        int wl, iwl, wl_sum, iwl_sum, wl_max, wl_min, iwl_max, iwl_min;
        double wl_mean, wl_var, iwl_mean, iwl_var;

        fix_form();
        fix_form(int N_fx, int Ni_fx);
        ~fix_form();

        fix_form& operator=(fix_form);
};
```

Figure 4.15: Members of class used for fixed point analysis

It is important to note the variation in the required word lengths over time. For the BOT problem, for example, as time progresses the target in general tends to move away from the origin and the largest range with respect to the origin that can be represented depends upon the integer word length chosen. We use this information to come up with a range specification for our filter for a particular fixed point format. Another specification

is the resolution of the filter. Due to the limited number of bits on the right of the binary point, there is a limitation the least value that the filter can resolve. This plays an important role in the SIRF or any SMC technique. In theory, any SMC technique should benefit from using a larger number of samples (particles). However if the fractional bit width used in the representation is small as compared to the variance of the sample space, then effectively, the number of unique samples reduces thus affecting the filter performance.

4.8.3 Critical Variables

The methodology described above, often results in formats that are not feasible for implementation due to resource limitations. Here, a distinction is made between variables that affect filter performance and those that affect filter stability. The so called *critical variables* are those whose accuracy cannot be compromised. More specifically, loss of accuracy in these variables will lead to the filter not functioning at all. Examples includes variables in denominators in division operations. On the other hand there are some variables for whom compromise on the bit width only affects the accuracy of the filter and not the functionality. For instance consider the value in the exponent of the importance computation step in the SIRF. If this value is very large(of the order of 10^5) , the resulting weight will be close to zero. A particle with such a small weight will almost surely be eliminated in resampling. Hence there is no need to have a very wide register for the weights which in turn means that we do not need a very wide register for the exponent. The approach we follow for these non critical variables is a type of back annotation of values. We decide the smallest(or largest) width used for representing a particular variable. Then we traverse the data flow in the reverse direction and back annotate the value though inverse functions and decide the width of all variables accordingly. A similar methodology has been used in [106].

4.8.4 Analysis of the SIRF for the BOT problem

The aforementioned analysis has been applied to the SRIF for the BOT problem. Figure 4.16 shows the variation in time of the statistics of the variables averaged over 5 realizations. This graph is important for deciding the range of tracking. The analysis has been done for all the variables but only some of them are shown on the graph.

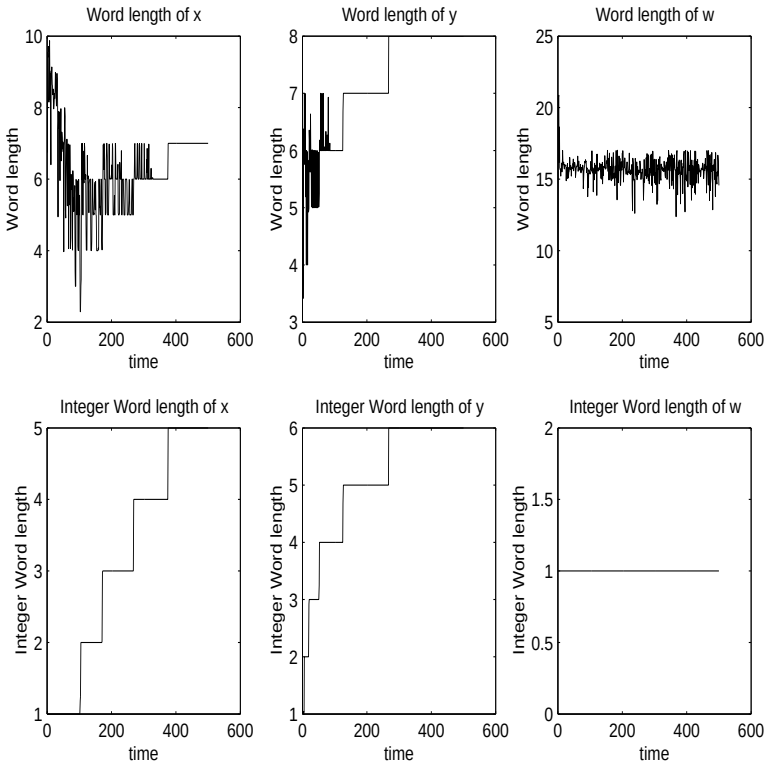


Figure 4.16: Graphs showing the variation in the fixed point format of the 3 variables with time averaged over 5 realizations

We choose to track in the range [-32,-32] to [32,32]. This gives 6 bits in the integer part of the representation. It can be seen that the weights which are the most critical variables in the SIR filter on an average need slightly more than 15 bits and this remains somewhat constant with time, Hence, for the weights we use 16 bits. The following table shows the general format that we use for all the variables in the SIR filter.

Finally, Figure 4.17 shows the shifting and scaling involved in the computation data flow. We start off with the two half Box Muller variables in look up tables for noise generation. Noise samples for the x and vx particle generation need to be multiplied by small coefficiants. Hence we use the shift-and-scale operation to avoid excessively large fractional bit width. A similar flow is used for fixed point scaling and computations in the other dimension to calculate y and vy.

Variable name	Max Word Length	Mean Word Length	Max Int Word Length	Mean Int Word Length	Used Format $< wl, iwl >$
x	12	6.45	5	- mon.inc-	$< 16, 6 >$
y	17	7.97	6	-mon. inc-	$< 16, 6 >$
vx	24	13.19	1	1	$< 16, 2 >$
vy	11	9.89	1	1	$< 20, 2 >$
imp1	6	2	6	2	$< 20, 4 >$
imp2	29	15.06	3	1.01	$< 18, 3 >$
imp3	40	26.33	4	1.01	$< 18, 2 >$
imp4	29	14.67	19	3.44	$< 18, 7 >$
weights	40	7.56	1	1	$< 16, 2 >$
sum of weights	40	10.65	13	7.66	$< 27, 13 >$

Table 4.7: Table representing the sizes of some of the variables in the SIR algorithm.

4.9 Evaluation

In this section, we present the results of the implementation and a comparison of the two proposed architectures. Both architectures were captured using Verilog HDL and synthesised on a Xilinx Virtex 2 pro FPGA platform. The design was verified using Modelsim from Mentor Graphics. After verification, the Verilog description was used as input to the Xilinx Development System which synthesized, mapped and placed and routed the designs on a Xilinx Virtex 2 pro device (XC2VP50-ff1152). The implemented design was verified through a post place and route simulation using Modelsim.

4.9.1 Execution Time

Fig. 6.12 shows the timing of operations for one recursion of the SIRF. In the figure, L_S and L_I represent the start up latencies of the sample and importance unit respectively. T_{res} is the number of cycles required for resampling. The total cycle time of the SIRF is then $T_{SIRF} = (M + L_S + L_I + T_{res})T_{clk}$, where T_{clk} is the system clock period.

As can be seen from the timing diagram, the resampling step is a bottleneck in the SIRF execution as it cannot be pipelined with other operations. Thus, T_{res} significantly affects the cycle time T_{SIRF}. Hence, development of faster and more efficient resampling algorithms is vital to the implementation of real time particle filters in high speed applications. The architectures and their modifications that have been presented in this paper, help to bring

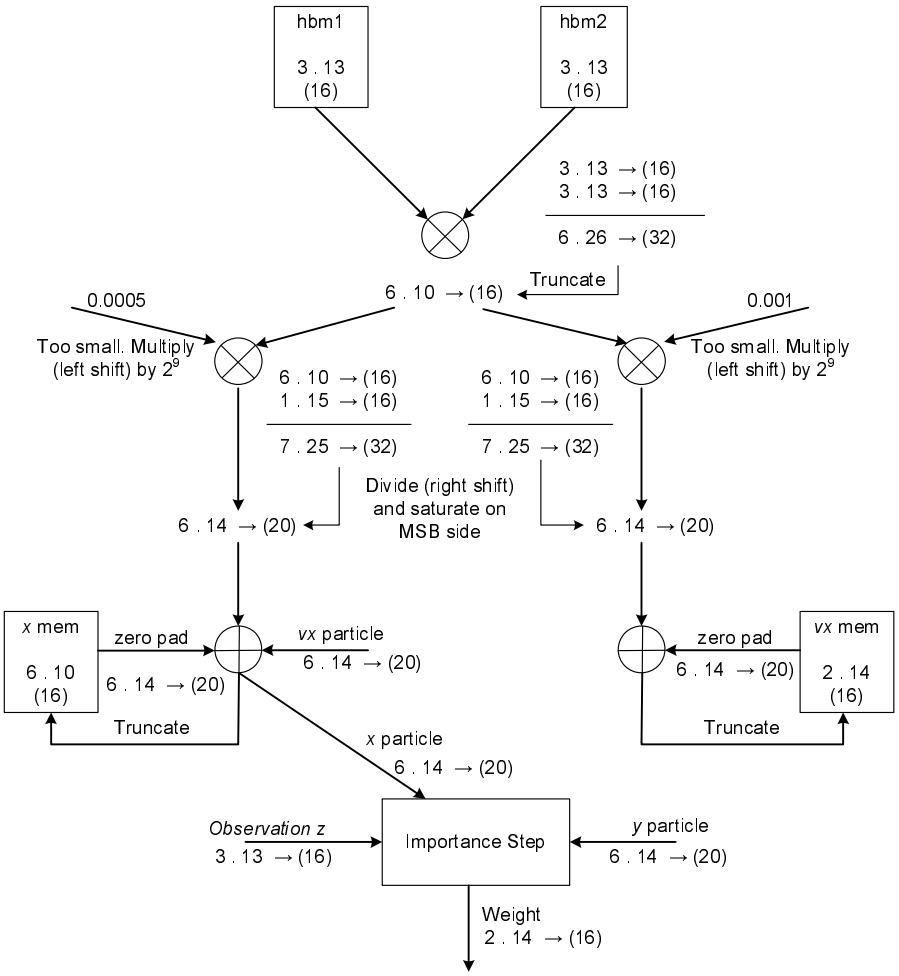


Figure 4.17: Fixed point scheme for SIRF

down T_{res} in different ways and hence reduce the effect of the resampling bottleneck. A resampling scheme should be chosen such that its time T_{res} satisfies the required T_{SIRF} for the application at hand. The modified versions of the two resampling schemes partially parallelize resampling and reduce execution time at the cost of added hardware. The time T_{res} and T_{SIRF} needed for resampling and one SIRF recursion respectively in terms of cycles is summarized in TABLE 4.8. k is the number of loops into which resampling is split as described in Section 4.4.1 and Section 4.5.2 for modified schemes.

In the table, L accounts for the start up latencies of all the units in the respective schemes.

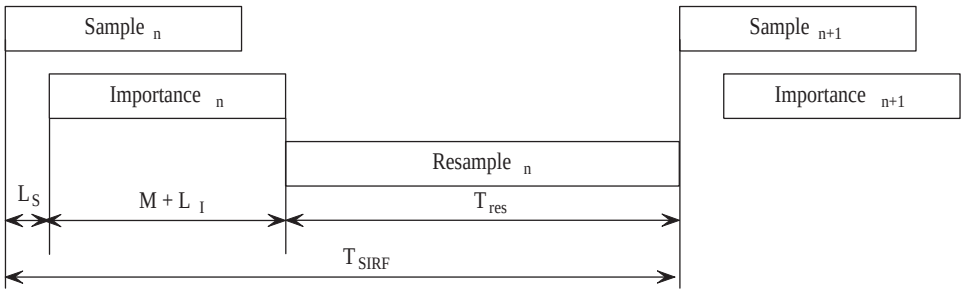


Figure 4.18: Timing of operations in SIRF

Time (cycles)	Scheme 1	Scheme 2	Modified Scheme 1 with k loops	Modified Scheme 2 with k loops
T_{res}	$2M-1$	$M+L_{rsr}$	$2(\frac{M}{k})-1$	$\frac{M}{2}+L_{RSR}$
T_{SIRF}	$3M+L$	$2M+L$	$(\frac{k+2}{k})M+L$	$(\frac{k+1}{k})M+L$

Table 4.8: Timing of SIRF using the different proposed architectures.

4.9.2 Resource Utilization

The architectures presented in the paper include all the memory related operations of the generic SIRF. We use a Virtex 2 pro FPGA platform for evaluation [111]. All memory modules are mapped to the 18Kb block RAMs available on the chip. The memory required by SIRF for processing a certain number of particles depends upon the dimension of the state and the number of bits used in the fixed point representation of the state, weights (or CSW) and indexes. The number of 18Kb blocks needed on the Virtex 2 pro device for storing M particles(also weights, CSWs or indexes), B_M, is given by

$$B_M = N_s\lceil\frac{M \cdot b}{18 \times 1024}\rceil \tag{4.5}$$

where b is the number of bits used for representation of the word in the memory.

TABLE 4.9 summarizes the total utilization of the proposed architectures. The model we have chosen for our evaluation is the $N_s = 4$ dimensional bearings-only tracking (BOT) problem. The total memory requirement of the two schemes *for the mentioned bit widths* shown in TABLE 4.9. Scheme 1 requires more memory than scheme 2 since it needs to store the CSW which has a wider fixed point representation. The amount of block RAMs available on a particular FPGA will determine the number of particles that an SIRF realization on

that device can process. Thus, (4.5) can be used as a guideline for selecting a device for a particular implementation. The mathematical units in Scheme 2 like the adders and multiplier were chosen to be 18 bits wide in input and output. The table also gives an estimate of the resources needed for the modified implementations of the two schemes with resampling being split into k parallel loops.

Resource	Scheme1 (implemented)	Scheme 2 (implemented)	Modified Scheme 1 (estimated)	Modified Scheme 2 (estimated)
Slices	199	294	$k \cdot 199$	$k \cdot 294$
Slice Registers	130	224	$k \cdot 130$	$k \cdot 224$
4 Input LUTs	232	348	$k \cdot 232$	$k \cdot 348$
Block RAMS	15	14	$11 + \lceil \frac{k}{2} \rceil$	$10 + \lceil \frac{k}{2} \rceil$
Block Multipliers	0	1	0	k

Table 4.9: Resource utilization for the two schemes on the XC2VP50-ff1152 device

Finally, TABLE 4.10 shows the comparison of cycle time and memory requirement (in terms of words) for the proposed schemes with a straightforward implementation starting from the traditional algorithm. This approach requires the sampled and resampled particles to be stored in different memories. Also if the index addressing schemes presented here are not used, another M cycles are added to the SIRF execution time since resampled particles first need to be read from one memory and written to another before the sample step can begin.

Parameter	Straightforward Implementation	Scheme 1	Scheme 2
Memory (words)	$2 \cdot N_s \cdot M$	$N_s \cdot M + 2$	$N_s \cdot M + 2$
T_{SIRF} (cycles)	$4 \cdot M + L$(SR) $3 \cdot M + L$(RSR)	$3 \cdot M + L$	$2 \cdot M + L$

Table 4.10: Comparison of memory requirement and cycle time with a straightforward implementation

4.9.3 Tracking Performance

The entire SIRF along with the computational units of sampling and importance for the above mentioned bearings only tracking problem was implemented on a Xilinx Virtex II pro device (XC2VP50FF1152). This FPGA prototype used the architecture described in scheme 1 with a resampling time $T_{res} = 2M - 1$ cycles as explained earlier. Each input sample is processed by the SIRF to produce an estimate of the unknown state at that sampling instant.

The sample and importance computation units have a latency $L_S = 8$ cycles and $L_I = 53$ cycles. $M = 2048$ particles are used for processing. Hence the SIRF cycle time is

$$T_{SIRF} = [(2048 + 8 + 53) + ((2 \times 2048) - 1)]\, T_{clk}$$

Thus one recursion of the SIRF needs 6024 cycles. The designed hardware can support clock frequencies of upto 118 MHz. Using a clock frequency of 100 MHz, we get the speed at which new samples can be processed $1/T_{SIRF} = 16KHz$. TABLE 4.11 shows the resource utilization of the entire SIRF for the BOT problem. The outputs are 32 bits wide and are in fixed point format. Their values are interpreted using SystemC fixed point data types and plotted using MATLAB. The figure also shows the tracking results obtained by the SIRF run in MATLAB using floating point representation for all variables. These results are also compared with tracking results obtained with the traditional EKF which for this model has execution speed of 10KHz on a DSP platform (TMS320C54x).

This performance figure of 16 KHz is for 2048 particles. In practice a much larger number of particles is needed for tracking in noisy environments. This makes the SIRF computationally very intensive and real time processing using any software platform or DSP is not possible even for low sample rates. The FPGA hardware SIRF on the other hand can process input samples at rates of upto 3.5KHz even with 10000 particles using the basic Scheme 1. Large number of particles will lead to increased memory requirement. But a large FPGA like the one chosen for our evaluation will support a high number of particles. Thus, the hardware realization of the SIRF not only allows for increased sample rates but also enables real time processing even with a very large number of particles.

By using the other schemes introduced in the paper, the SIRF can be made even faster.

Resource	Random no. generation	Sample	Importance computation	Resample	Top level logic	Total
Slices	300	411	1,535	199	623	3068 (13%)
Slice Registers	364	568	2,578	130	752	4392 (10%)
4 input LUTs	196	404	2,674	232	342	3848 (8%)
Block RAMs	2	8	1	7	0	18 (8%)
Block Multipliers	6	0	3	0	4	13 (6%)

Table 4.11: Resource utilization for entire SIRF for the bearings only tracking problem using Scheme 1.

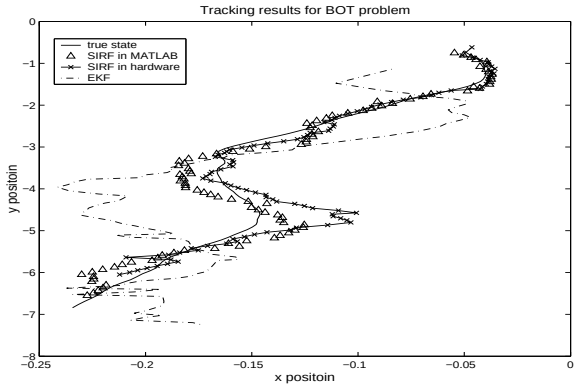


Figure 4.19: Results of tracking for the BOT problem

4.10 Conclusion

In this chapter, we have presented a generic architectural framework for the hardware realization of SIRFs applied to any model. The architectures reduce the memory requirement of the filter in hardware and make efficient use of the dual port memories available on an FPGA platform. Two architectural schemes, Scheme 1 and Scheme 2 were proposed based on the SR and RSR algorithms respectively. The resampling process cannot be pipelined with other operations and is a bottleneck in the filter execution. Hence for high speed applications, the high latency of SR in scheme 1 is unacceptable. Scheme 2 uses the faster but more complicated RSR algorithm which allows for lower SIRF cycle times. We also introduced modifications of the two schemes involving parallelization of the resampling process

by splitting it up into multiple concurrent loops. This allows for reducing the resampling latency and thus the SIRF cycle time at the cost of added hardware. The tradeoff between speed and hardware cost will dictate the choice of architecture for an SIRF realization.

We further described the design of the computational units of sample and importance step for the BOT problem. The sample step involved Gaussian noise generation which was investigated. A detailed fixed point analysis was performed on the filter variables to decide the representation scheme used.

Using all these results, an FPGA prototype of the SIRF for the BOT problem was implemented. To the best of our knowledge this was *first FPGA prototype* for a particle filter. For 2048 particles, this prototype can process input observations at 16 KHz which is about 32 times faster than speed achieved for the same problem with the same number of particles on a state of the art DSP.

Chapter 5

Design of Hardware for Reconfigurable Particle Filter Realizations

5.1 Introduction

In this chapter, we present a design methodology for hardware providing parameterizable, reconfigurable particle filter realizations. Specifically, we describe a design which can be configured for different types of particle filtering by modifying a minimal set of control parameters. Flexibility and generality of the particle filter are among its most attractive properties. Exploiting this in hardware requires a reconfigurable platform. The proposed methodology provides the capability of selecting a single particle filter from multiple possible realizations with maximum resource sharing. An autonomous buffer controller mechanism is proposed for the architecture which ensures correct data flow and synchronization of operations. Parameter adaptation and algorithm reconfiguration can be accomplished with negligible reconfiguration overhead through buffer controllers and a set of switches for transforming dataflow structures such that any desired particle filter can be implemented. Two target particle filters, SIRF and GPF, are realized on an FPGA platform based on the proposed methodology. However, the architecture can be extended for a wide range of particle filters with different sets of dynamics. The FPGA platform is used for rapid prototyping

and evaluation. The methodology and the benefits thereof are equally applicable to an ASIC platform. Through this work, we successfully demonstrate that implementation of a domain specific processor for particle filters is feasible with performance that is much higher than that of commercially available DSPs.

As we have seen in Chapter 2 and Chapter 3, execution of PF algorithms on sequential hardware affects real-time performance due to their computational intensity. Hence, it is highly desirable to develop a programmable particle filtering hardware that can replace the use of DSP. As we have done so far, we use the BOT problem as the target application. The main contribution of this work is a design methodology for a high speed, domain specific, parameterizable particle filtering hardware with reconfigurability. In practice, a PF algorithm for tracking must change to cope with the type of objectives and the dynamics of the target. A reconfigurable PF can be used in a wide range of applications depending on the statistical parameters of input observation and dynamic models. Reconfigurability in the proposed design includes the type of PF used, dimension of state spaces and the number of particles that may change dynamically to adapt to changing environments. For many real-time applications, dynamically varying degree of parallelism is desirable where processing elements duplicate for higher throughput processing. In addition, multiple instances of the same PF are useful for tracking more than one object at a time. It is also desirable to change the execution speed for power reduction if real-time requirement is relaxed. Thus, we can envision a reconfigurable processor that can support the above situation dynamically during run-time without redesigning particle filters whenever specifications change.

The use of reconfigurable DSP is not new [100], but most this effort has been focused at traditional fine grain operations such as multipliers and adders. To the best of our knowledge no previous works focus on reconfigurable architectures for the PF. Commercial DSPs provide the ultimate flexibility for PF design. Filtering parameters as well as type of filters can be easily adapted through change in software routines for a specific filtering task. Even though current DSPs are highly pipelined and support some level of concurrency, they are not suitable for high-speed real-time particle filtering. On the other hand, in the FPGA implementation, operational concurrency can be achieved by utilizing multiple processing elements or parallelizing the overall filtering algorithms. However, a key problem with the FPGA implementation using commercial design environment is that programming/reconfiguration is

done statically during the design stage with long time to program, and without special support at the implementation level. Recently, commercial FPGAs have started to support partial dynamic reconfiguration, but this comes at the cost of a complicated design flow, and larger overheads which make it impractical for large designs [112].

The reconfigurable PF architecture presented here is based on maximizing reuse of processing blocks common to different PF algorithms. The interconnection and interface between the processing blocks is done using distributed buffers, controllers, and multiplexers. The rest of the chapter is organized as follows. Section 5.2 briefly describes relevant characteristics of the PFs and the reconfigurable design proposed. Section 5.3 discusses the processing blocks involved in the design. The processing blocks of the SIRF have been described in Chapter 4. The design of some of the unique GPF processing blocks is explained briefly. Section 5.4 details the structure and operation of the distributed buffer controller which is a key element in the design. The combined architecture with processing blocks and buffer controllers is presented in Section 5.5. The designs are mapped to FPGA and are evaluated in Section 5.6. Finally, our contribution is summarized in Section 5.7.

5.2 Basis of the proposed design

5.2.1 Particle Filter Characteristics

For this design, we focus on two types of PFs viz. the SIRF and the GPF. The selection of an algorithm from these two types of filters depends upon the characteristics of the dynamic state space of interest. PFs, as many real-time signal processing algorithms, work on blocks of data as frames. They can be represented as coarse data-flow graphs such that nodes (or blocks) can be executed concurrently [87]. While the complexity of each node (or block) differs in granularity, the data-flow graph can be clearly represented as a function of data dependency. Each node in the data-flow graph executes on a set of data every iteration cycle. Depending on applications, the size of data set can be large requiring significant amount of buffers. Considering the dataflow within these applications, a two-level hierarchy is often obvious, where data frames are processed as a unit in a sequence of logic blocks at a global level, and elements within a frame are processed in a loop fashion within each block at a local

level. For example in the function $Y = h[f(X) + g(X)]$, the data frame X, $(x_1, x_2, \cdots, x_M)$ of dimension M is processed by the functions $f(\cdot)$ and $g(\cdot)$ in a loop, and, at the global level, the sums of elements from these two functions are processed by the function $h(\cdot)$ generating the data frame Y.

5.2.2 Block Level Pipelining

Block level pipelining provides a hardware realization of the dataflow model previously described. The block (frame) based processing is incorporated in the architecture by introducing two level pipelining, i.e. fine grained (register based) and block-level (buffer based) pipelining.

A buffer along with associated controller is inserted between successive processing elements at the block level. This buffer-controller serves as a pipeline element, and also allows for incorporation of local architectural parameters such as latency and rate difference between a pair of processing blocks. Through block level pipelining, we can achieve three key objectives:

1. It is possible to maintain concurrency of each processing block while providing correct synchronization between them for proper execution.

2. Since the control signals, data and clock become local, hardware implementation is much easier in terms of maintaining performance by minimizing clock skews and data routing. Any change in logic will only affect its buffer size and controller configuration thus simplifying reconfigurability and core reuse.

3. Since the entire design is centered around the buffers, performance mismatches between memory and logics are minimized.

Fig. 5.1 illustrates a block-level pipelining structure. Data are transferred by the read and write access of the buffers *concurrently but at different address locations.* The offset or difference between the write and read address can be determined from the rate differences of processing blocks as well as data dependency between them. Each buffer can have different offset as required by the logic operation writing to or reading from it. The overall operation

is logically viewed as just buffer to buffer operations separated by latency of the processing blocks introduced by the processing block logic implementation. The data path may be recursive.

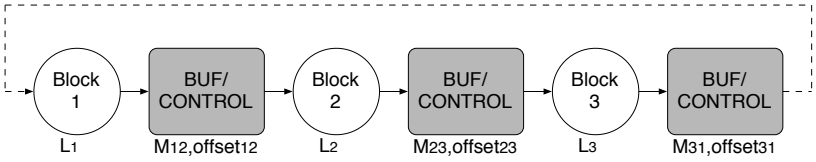


Figure 5.1: Illustration of the block-level pipelining structure of data flow.

The above structure is characterized by the parameters L_i, M_{ij}, and $offset_{ij}$, as indicated in Fig. 5.1. The values L_i represents the latency of processing block i and is obtained from the implementation, M_{ij} is the size of the data frame transferred between processing blocks i and j. The buffer write-read offset, $offset_{ij}$, is determined from the operational data dependency of the pair of processing blocks. While a pair of processing blocks in Fig. 5.1 produces and consumes the same number of data, their data rates are not necessarily identical.

5.2.3 Orthogonal Controller Design

When we consider dynamic run-time reconfiguration of PFs, a method for controlling the desired operation becomes an important issue. Two approaches for controller design are possible, namely centralized and distributed. In the centralized approach, a single large controller is used to generate the different control signals for all the units. While [53] takes a centralized controller approach arguing on the grounds of area overhead, [67, 24] take a distributed approach where the control is localized at the units and they operate by transferring information between them. We adopt a distributed approach in this paper. The main difference between our controller design methodology and other distributed approaches is that our design assumes that the execution characteristics of the processing block are known. This is a valid assumption since we can always characterize the processing blocks based on their implementation. This has the benefit that the processing blocks can be a logic core designed by another party. Moreover, we provide the flexibility in changing the controller locally

with minimum information and overhead. Such benefit is not possible with the centralized approach where a small change in logics translates to overall redesign of the controller. In the reconfigurable design, the centralized approach will lead to longer time to reconfigure the overall design because of the tightly integrated control signals. Our methodology is specifically targeted for block based processing for buffer centric applications. The methodology isolates local controllers from each other such that it is possible to provide operation predictability in the overall system design.

The two algorithms considered here share several common processing blocks but differ in their dataflow structures. Reconfiguration is achieved by configuring distributed buffer controllers for local processing blocks and structural controller for algorithm selection. We can view this reconfiguration scheme as one based on execution context.

5.3 Design of High Level Processing Units

5.3.1 Coarse Grain Data Flow Models

To fully utilize locality of the buffer controller, we follow three key strategies in designing the processing blocks:

1. Each processing block is designed to eliminate control signal dependency between processing blocks. If there exists such control dependency, we combine them for single processing block. If such integration is unavoidable, we treat the control signal as data between processing blocks under consideration.

2. We design each processing block so that the data consuming and producing rates are deterministic. Since all the blocks, including buffers, use a single global clock, relative rates can be defined. Moreover, the numbers of data produced and consumed are also made to be deterministic. It is also possible that the processing block is complicated enough to require its own local controller. However, we consider such highly localized controller to be a part of processing block. Thus, as long as we maintain deterministic input and output data flow, we can treat it as regular logic block.

3. We assume that there is only one global clock and all other clock signals feeding to processing blocks are derived from the global clock.

The architecture of the SIRF for bearings-only tracking problem is constructed as shown in Fig. 5.2. The figure shows both processing blocks and buffers (i.e., the buffer and its controller are shown with shaded boxes). Each dashed box is translated to a buffer controller. The same buffer controller is used for a grouped dashed box. A value inside the shaded box represents the number of data going into the buffer in each iteration. The Importance step has been split in to three blocks $IMP1$, $IMP2$ and $IMP3$ for maximizing resource sharing.

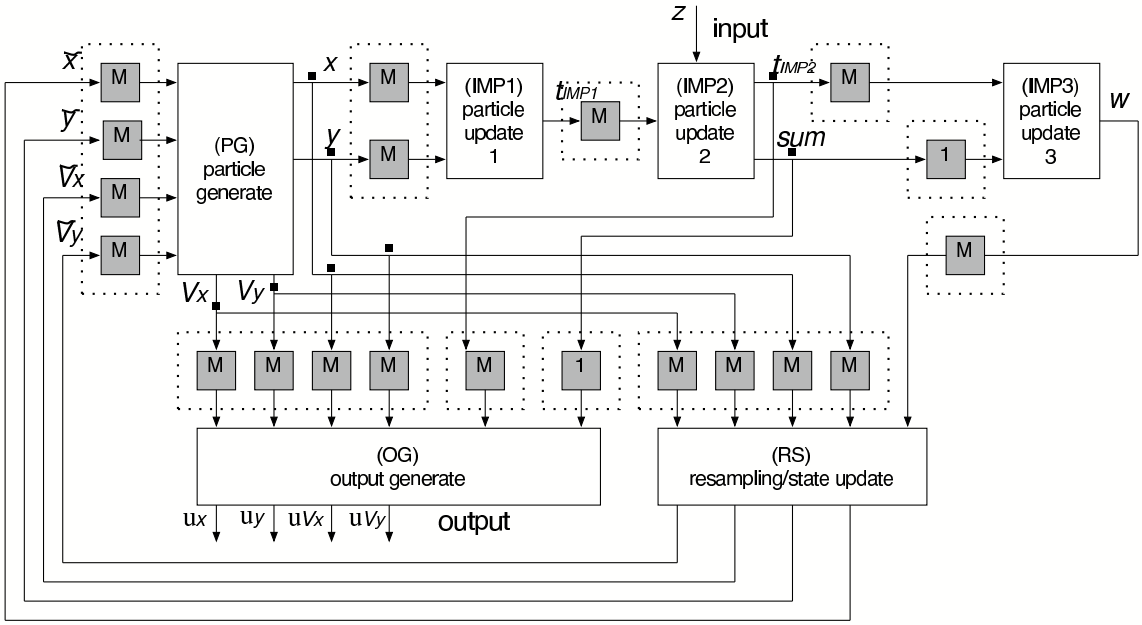


Figure 5.2: Dataflow graph (with buffers and processing blocks) of the SIRF particle filter.

Similarly, the architecture of the GPF for bearings-only tracking is constructed as shown in Fig. 5.3. The figure shows the processing blocks and the buffers. In these two architectures, each processing block doesn't change but the buffer controllers are reconfigured depending on the target algorithms.

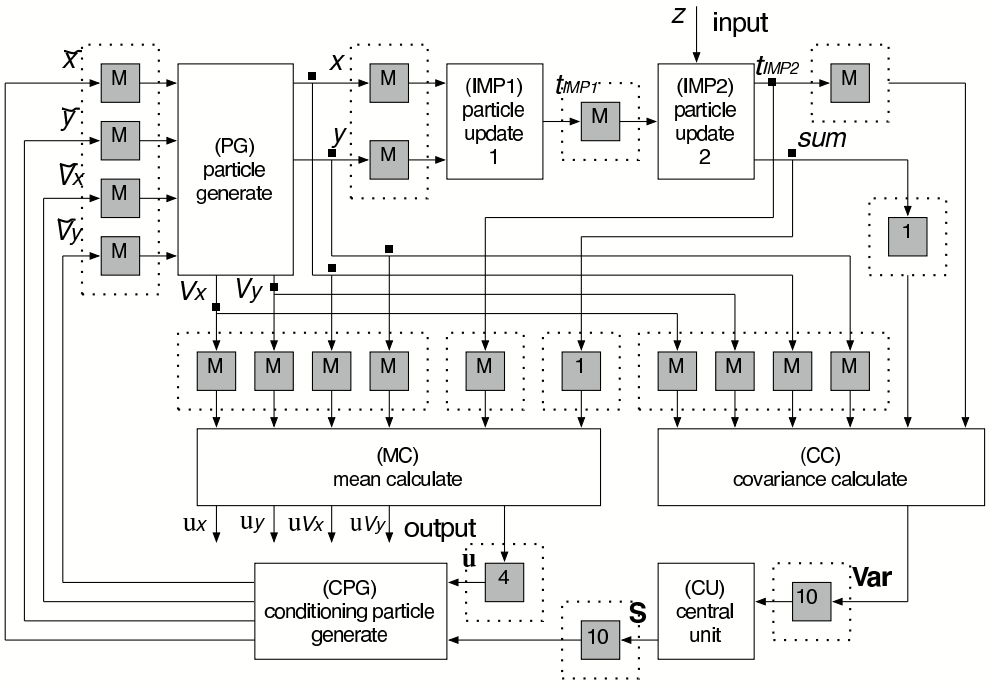


Figure 5.3: Dataflow graph (with buffers and processing blocks) of the GPF particle filter.

5.3.2 Shared Processing Blocks in Design

From the initial observation, we can classify the operations that are common to both filters. In this section, we describe the shared processing blocks that are common to the SIRF and GPF, and the processing blocks that are unique to each filter. In order to maximize the resource sharing, some of the processing blocks are divided into several smaller processing blocks.

1. Sample Step (Particle Generate (PG)): The design of this unit was described in 4.7.

2. Weight computation (Importance) step: The design of this unit was described in 4.7

5.3.3 Processing blocks unique to the SIRF

The Resampling block is unique only to the SIRF. The design of this unit was handled in detail in Chapter 4.

5.3.4 Processing blocks unique to the GPF

The GPF is designed based on the modified GPF algorithm in Table 3.5 in Chapter 3. Here we present the design of the processing blocks that are unique to the GPF.

1. Condition Particle Generation (CPG):

   In the CPG step, the decomposed covariance matrix $\mathbf{S}$ and the mean $\boldsymbol{\mu}$ obtained from the CU processing block are used for calculation of conditioning particles. The matrix $\mathbf{S}$ is a 4×4 upper triangular matrix, so the number of data that are transferred from the CU processing block is 10 (not 16). All the multipliers are pipelined and they operate concurrently producing M conditioning particles. Since, the outputs $(\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y)$ are computed using different number of operators, we have to introduce additional delay which is different for each state in order to get all the conditioning particles at the same time instant at the output. The CPG requires 4 random number generators.

   In the CPG processing block, there are 2 input buffers for $(\boldsymbol{\mu}, \mathbf{S})$ from the CU processing block and 4 output buffers for $(\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y)$ to the PG processing block. The data size of mean $\boldsymbol{\mu}$ is four and decomposed covariance $\mathbf{S}$ is 10. These data are generated sequentially to save interconnect buses. Internally, these data are used in parallel. The output data size is M. Initially, the mean and the decomposed covariance elements are obtained externally and not from the CU.

2. Covariance Calculate (CC):

   In the CC processing block, the partial covariance 4×4 matrix **Var** is calculated. This block generates the normalized partial covariance. In the CC processing block, there are 6 input buffers for (x, V_x, y, V_y) from the PG processing block and (t_{IMP2}, sum) from the IMP2 processing block. There is one output buffer **Var** to the CU processing blocks. These outputs are serialized.

3. Cholesky Decomposition Unit (CU) The central unit carries out the operation of scaling mean and covariance estimates and decomposing the covariance matrix using Cholesky decomposition. The pseudocode for this is presented in Table 5.1.

$$
\begin{aligned}
(\mathbf{Var}) &= \mathbf{GPF_{CH}}(\mathbf{x}, \mathbf{V_x}, \mathbf{y}, \mathbf{V_y}, \mathbf{w}) \\
S_{11} &= (Var_{11})^{1/2} \\
S_{12} &= Var_{12}/S_{11} \\
S_{13} &= Var_{13}/S_{11} \\
S_{14} &= Var_{14}/S_{11} \\
S_{22} &= (Var_{22} - S_{12} \cdot S_{12})^{1/2} \\
S_{23} &= (Var_{23} - S_{12} \cdot S_{13})/S_{22} \\
S_{24} &= (Var_{24} - S_{12} * S_{14})/S_{22} \\
S_{33} &= (Var_{33} - S_{13} \cdot S_{13} - S_{23} * S_{23})^{1/2} \\
S_{34} &= (Var_{34} - S_{13} \cdot S_{14} - S_{23} \cdot S_{24})/S_{33} \\
S_{44} &= (Var_{44} - S_{14} \cdot S_{14} - S_{24} \cdot S_{24} - S_{34} \cdot S_{34})^{1/2}
\end{aligned}
$$

Table 5.1: Pseudocode for Cholesky decomposition.

The architecture of the GPF CU for implementing Cholesky decomposition is shown in Fig. 5.4. The inputs and the outputs of the CU are produced once during the sampling period. As seen from this pseudocode, this operation involves several expensive operations like divisions and square root. There also exist data dependencies between the various coefficients of the Cholesky decomposed matrix. Due to this use of dedicated hardware units for each of these operations proves to be expensive due to the nature of the operators and does not speed up processing greatly due to the inherent data dependency. Observing concurrency of operations from the data flow for Cholesky decomposition, we see that at any time, the maximum number of concurrent square root operations that can be performed is one, and additions, multiplications and divisions are three. Hence we implement the Cholesky decomposition with only these resources. The square root operation is implemented using the CORDIC core [113] and the division is implemented using the pipelined divider core [110]. The intermediate results are stored in internal registers and sent back to the computation unit to use in the calculation of the next coefficients. All the hardware units are time multiplexed (reused by subsequent operations). Like any other processing block in the design, this unit has a local controller that generates the controls for the multiplexers and read/write signals for the input and output buffers. Due to its sequential nature, the Cholesky decomposition can also be implemented on a sequential coprocessor, like the IBM PowerPC core embedded in the Virtex II pro FPGA.

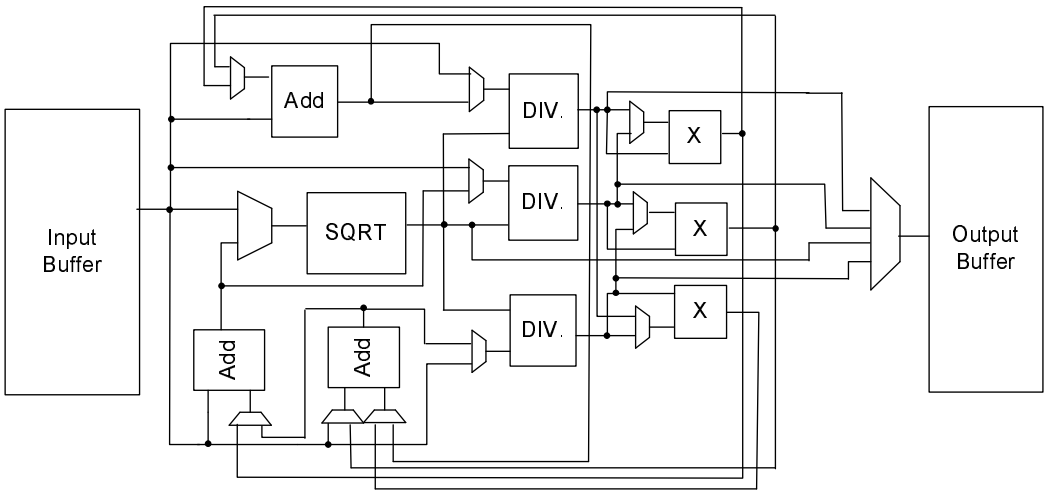


Figure 5.4: Architecture of the Unit Implementing Cholesky Decomposition

The bit widths and fixed point representation formats used in all the processing blocks were determined, for the BOT problem, using a methodology outlined in [58].

5.4 Distributed Buffer Controller

The operations in the block based processing are viewed as buffer to buffer operations with coarse grained processing blocks operating in between them. A block diagram of the buffer controller, which is a key element in the proposed design, is shown in Fig. 5.5. The buffer controller consists of concurrent controller and a dual-ported memory. When handling multidimensional data, the buffer controller has multiple dual port memory units controlled by the same controller (grouped dashed boxes in Fig. 5.2 and Fig 5.3. The number of memory units is equal to the dimension of the data. The concurrent controller has two logic sections: read and write. The write logic section is configured by the parameters L_i and nw_{ij}, and the read logic section is configured by D_{ij} and nr_{ij} where i and j denote the producing and consuming processing blocks respectively. Note that these parameters are derived from the dataflow structure and the processing block implementation details.

When this buffer controller is activated, both the write and read logic sections are concurrently executed. Initiation of the write section indicates that data have arrived at the processing block that is connected to this buffer as a producer. The actual data computed

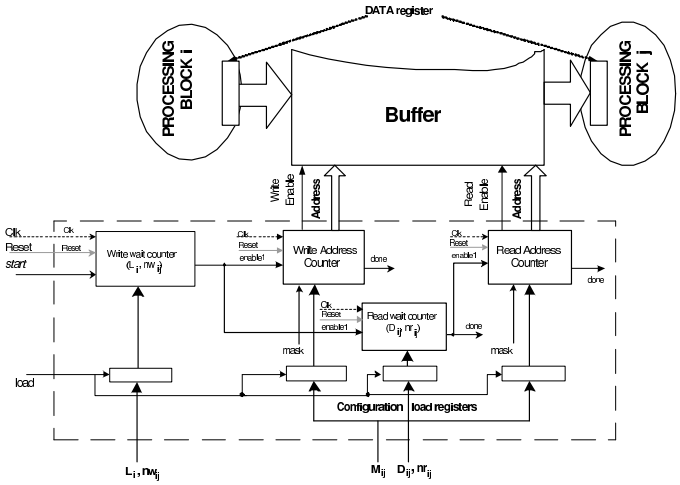


Figure 5.5: Block diagram of a buffer/controller consisting of a read and write processes. The controller is programmed with the buffer controller parameters.

by the producing processing block is valid at the buffer controller after waiting for L_i cycles. The write logic section will not write these L_i invalid data from the producer. This will guarantee correctly receiving the valid data stream if the producer is purely pipelined hardware. However, it is also possible that the processing block needs finite amount of computation time regardless of the pipeline depth (i.e., delayed data generation by the processing block). To support this type of processing block, we use one more parameter nw_{ij}. After this wait period ($L_i + nw_{ij}$), the data are written to the buffer. Once correct data samples are being written to the buffer, the read process is started by the read logic section. The parameter nr_{ij} represents the offset between writing and reading the data from the buffer. This parameter is to support data dependency. Even if there are no data dependency, it is also possible that the producer data generation rate is different from the consumer data consuming rate. To support such rate mismatch between two processing blocks connected by the buffer controller, we use another parameter D_{ij}. After this wait period ($\max[nr_{ij}, D_{ij}]$), the data are read from the buffer. Thus, the write logic section is configured by (L_i, nw_{ij}) and the read logic section is configured by (nr_{ij}, D_{ij}). The same buffer controller is used to support different data transfer characteristics by modifying these parameters.

There activation of the buffer controller is governed by three key synchronization signals: $start\_time_{ij}$, $write\_begin_{ij}$, and $read\_begin_{ij}$, where the index ij represents a buffer controller placed between the processing blocks i and j. The start of the write waiting process is

synchronized with the start read process of the previous buffer controller, indexed as ki. And the start of the read process is synchronized with the start of the write waiting process of the same buffer controller. They have the following relationships:

$$start\_time_{ij} = read\_start_{ki}, \tag{5.1}$$

$$write\_start_{ij} = start\_time_{ij} + L_i + nw_{ij}, \tag{5.2}$$

$$read\_start_{ij} = write\_start_{ij} + \max[nr_{ij}, D_{ij}]. \tag{5.3}$$

Each buffer controller is controlled by periodic signals, indicated by *start*, generated from the global controller with a counter driven by a global clock. This is illustrated in Fig. 5.6. For each buffer controller, the *start* timing signal indicates the start of one iteration process. Successive buffer controllers are separated by *start* signals. The global controller is also responsible for the loading of timing information into the local controllers.

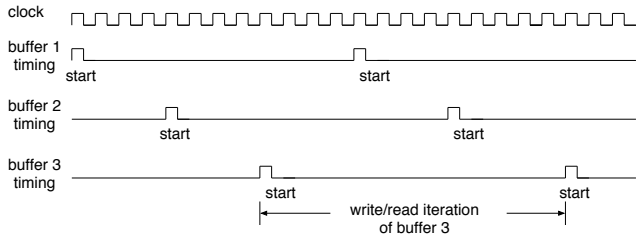


Figure 5.6: Relative timing of the buffer controllers. All buffer controllers are synchronized with a global clock. Each start signal is periodic and represents the iteration of the algorithm.

One of the main advantages of this scheme is that the buffer controller knows exactly when the data are being transferred. This is determined from the parameters. Thus, when the producing block or consuming blocks are not active (i.e., no data are written to or read from the buffer), the corresponding processing block can be disabled for energy saving. In the buffer controller, it may seem that the memory for buffer is used unnecessarily and large. However, the size of the buffer controller is well defined from the its parameters, and the actual size of the memory can be identical to the number of registers that may be needed in traditional pipelining with hand shake. The predictability of the processing block execution is especially beneficial for low power design. A simple hand shake mechanism does not have

a clear information of the processing block activity.

5.4.1 Extension for Rate Mismatch

Consider a buffer controller connected between two processing blocks i and j such that i is the producer and j is consumer for this buffer controller. Data generation rate of the producer block is P_i, which is the write rate of the buffer controller . Similarly, data consuming rate of consuming block is C_j which is equal to the data read rate of the buffer controller. We assume these clocks are derived from the global clock F_{clock}. We will consider two possible cases: $P_i > C_j$ which leads to buffer overflow and $P_i < C_j$ which causes buffer underflow. Note that the number of data, M_{ij}, transferred through the buffer controller should be a constant and known.

Parameter D_{ij} is used when there is a data rate mismatch between a pair of processing blocks. When $P_i \geq C_j$, producer writes to buffer at faster rate than that the consumer can read from buffer. Under this condition, the read process does not have to wait once there is at least one valid data in the buffer. The minimum value of D_{ij} with respect to the global clock is given by Eq (5.4).

$$D_{ij} = \frac{F_{global}}{P_i} \tag{5.4}$$

When $C_j > P_i$, the read process has to wait for the producer to write enough data to the buffer to prevent underflow. Eq (5.5) gives the minimum wait period for the read process.

$$D_{ij} = \left(\frac{M_{ij}}{P_i} - \frac{M_{ij} - 1}{C_j} \right) \times F_{global} \tag{5.5}$$

5.5 Combined Architecture

The structure of the reconfigurable hardware implementing SIRF and GPF for bearings-only tracking problem is shown in Figure 5.7. The figure shows both processing blocks and buffers. In addition, there are switches associated with all the shared processing blocks and buffer controllers for selecting the appropriate structure. These switches are configured

dynamically, along with buffer controller parameters, before the beginning of any iteration.

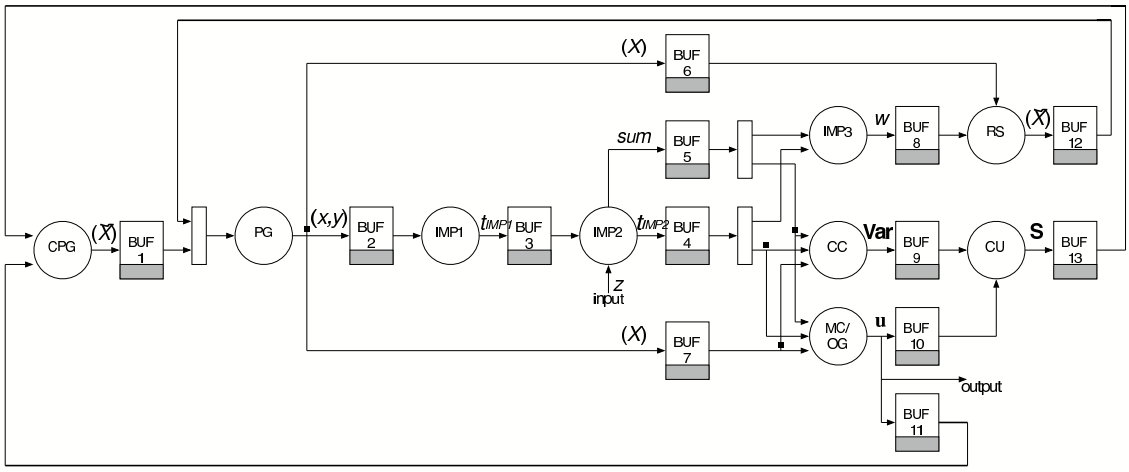


Figure 5.7: A dataflow graph structure of the reconfigurable particle filter. The structure contains both SIRF and GPF. Some buffer controllers are shared in the realization.

5.5.1 Buffer Controller Parameter Configuration

The parameters (M_{ij}, nr_{ij}, nw_{ij}) are derived from the functional (algorithmic) description of the particle filters. The parameters (L_i, D_{ij}) are determined from the processing block implementation. The synchronization parameters of buffer controllers and global controller are also determined from this information.

In the SIRF realization, key data dependencies that must be resolved are:

1. *sum* through BUF5 and the last value of t_{IMP2} through $BUF4$ must arrive at the same time. This requires $nw_{BUF5} = M + 1$, since $IMP2$ will take M cycles from its initialization to calculate the sum of weights.

2. the RS processing must wait for the sum of weights before it starts to execute. This dependency results in $nr_{BUF6} = M + 61$, where 61 is the sum of latencies of all processing blocks in the path from the PG block to the RS block.

3. The PG processing block must wait for the first data generated by the RS processing block (M cycles).

4. The particle arriving at the OG block via $BUF7$ and the corresponding weight arriving via $BUF5$ must be synchronized.

Similarly, the GPF has the following important data dependencies:

1. The particle and weight arriving at MC through $BUF7$ and $BUF4$ respectively must be synchronized. Hence $BUF7$ reading must delayed till the first weight is calculated.
2. Data arriving at the CPG block from $BUF11$ and $BUF13$ must be synchronized.
3. The CC and MC blocks need M cycles after reading their first data to generate result. Hence $nw_{BUF9} = nw_{BUF10} = M$.

Thus, using the buffer controller parameters nw and nr, the data dependencies in the algorithms can be quantified and incorporated into the design. In both realizations, the values of D are all one since there is no rate mismatch. The parameters of the other buffer controllers in the design are determined using a similar reasoning. The depth of each buffer is bounded by $\min(nr_{ij}, M_{ij})$, where nr_{ij} is the read offset and M_{ij} is the number of data words passing through that buffer per iteration. In reality, some buffer controllers in the design need to have additional dual port memory units to account for multi dimensional states. Accordingly, if N_s is the dimension of the state involved in the filtering, some buffer controllers in Fig. 5.7 will need to incorporate N_s dual port memory units of appropriate depth. The read/write operations of all memory units within a buffer controller will be done using the same control signals. The Edge Information Table (EIT) (TABLE 5.2) shows the values of the various buffer controller parameter for realization of SIRF and GPF using the architecture of Fig. 5.7 for the $N_s = 4$ dimensional bearings-only tracking problem. The sizes and format of memory in each buffer controller for a general N_s dimensional state model is also sown in TABLE 5.2. Note that M_{ij} stands for the number of data words passing through the buffer controller between block i and j, while M stands for the number of particles used for filtering.

For the BOT model with $N_s = 4$							General Case	
Buffers	Signal	L_i	nw_{ij}	nr_{ij}	D_{ij}	M_{ij}	No. of Mem. units	Depth of each unit
BUF1	$(\tilde{\mathbf{X}})$	11	[-, 1]	[-, 1]	[-, 1]	[-, M]	N_s	1
BUF2	(x, y)	8	[1, 1]	[1, 1]	[1, 1]	[M, M]	N_s	1
BUF3	(t_{IMP1})	23	[1, 1]	[1, 1]	[1, 1]	[M, M]	1	1
BUF4	(t_{IMP2})	20	[2, 2]	[1, 1]	[1, 1]	[M, M]	1	1
BUF5	(sum)	20	[$M+1$, $M+1$]	[1, 1]	[1, 1]	[M, M]	1	1
BUF6	$(\mathbf{X})$	8	[1, -]	[$M+61$, -]	[1, -]	[M, -]	N_s	M
BUF7	$(\mathbf{X})$	8	[1, 1]	[49, 47]	[1, 1]	[M, M]	N_s	49
BUF8	(w)	10	[1, -]	[1, -]	[1, -]	[M, -]	1	1
BUF9	(Var)	8	[-, M]	[-, 1]	[-, 1]	[-, 10]	1	1
BUF10	(μ)	8	[-, M]	[-, 1]	[-, 1]	[-, 4]	1	1
BUF11	(μ)	8	[-, M]	[-, 78]	[-, 1]	[-, 4]	1	N_s
BUF12	$(\tilde{\mathbf{X}})$	19	[M, -]	[1, -]	[1, -]	[M, -]	N_s	1
BUF13	(S)	1	[-, 75]	[-, 1]	[-, 1]	[-, 10]	1	1

Table 5.2: Edge Information Table (EIT) for reconfigurable realization. Each edge requires a buffer. Each entry, denoted by [a, b] represents parameters for SIRF and GPF (i.e., a is for the SIRF and b is for the GPF. The symbol - means that the buffer controller is not used for the corresponding realization). $\mathbf{X} = (x, V_x, y, V_y)$ and $\tilde{\mathbf{X}} = (\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y)$.

The total amount of buffer used for the synchronization for SIRF and GPF respectively is $4M + K_{SIRF}$ and K_{GPF} where K_{SIRF} and K_{GPF} are constants that depends on the implementations of the processing blocks. Thus, the GPF has a much lower buffer usage than the SIRF.

5.5.2 Synchronization Signals

The overall buffer synchronization parameters are generated according to (5.1), (5.2), and (5.3). Table 5.3 summarizes the parameters for all the buffer controllers for the SIRF and the GPF, respectively. The parameters for the start instant are computed with respect to f_{clock}. The executions of the PG, IMP, OG, and RS processing blocks are overlapped in time. The minimum iteration period of the SIRF is $T_{SIRF} = (2M + L_{SIRF}) \cdot T_{clk}$, where $L_{SIRF} = L_{PG} + L_{IMP} + L_{RS}$. Note that L_{OG} is not included since the output generation can be done within this cycle without creating a dependency. Thus, the iteration period, $2M+L_{SIRF} = 2M+89$, is indicated by the reset time instant. For the GPF, the executions of

CPG, PG, IMP, MC, CC, and CU processing blocks are overlapped. The minimum iteration period of the GPF is $T_{GPF} = (M + L_{GPF}) \cdot T_{clk}$, where $L_{GPF} = L_{CPG} + L_{PG} + L_{IMP} + L_{MC,CC} + L_{CU}$ and where L_{CU} includes the latency due to hardware and delayed output generation resulting from time-multiplexing within the CU processing block. The overall latency L_{GPF} is longer than the constant pipelining latency of the SIRF, L_{SIRF}. Thus, the iteration period, $M + L_{GPF} = M + 154$, is indicated by the reset time instant. For $M \gg L_{GPF}$, the GPF is almost twice faster than the SIRF.

Buffers	*start_time*	*write_begin*	*read_begin*
BUF1	[-, 0]	[-, 12]	[-, 13]
BUF2	[0, 13]	[9, 22]	[10, 23]
BUF3	[10, 23]	[34, 44]	[35, 45]
BUF4	[35, 45]	[57, 67]	[58, 68]
BUF5	[35, 45]	[$M + 56$, $M + 66$]	[$M + 57$, $M + 67$]
BUF6	[0, -]	[9, -]	[$M + 69$, -]
BUF7	[0, 13]	[9, 22]	[58, 69]
BUF8	[$M + 57$, -]	[$M + 68$, -]	[$M + 69$, -]
BUF9	[-, 68]	[-, $M + 76$]	[-, $M + 77$]
BUF10	[-, 68]	[-, $M + 76$]	[-, $M + 77$]
BUF11	[-, 68]	[-, $M + 76$]	[-, $M + 154$]
BUF12	[$M + 69$, -]	[$2M + 88$, -]	[$2M + 89$, -]
BUF13	[-, $M + 77$]	[-, $M + 153$]	[-, $M + 154$]
reset	[$2M + 89$, $M + 154$]	-	-

Table 5.3: Synchronization parameters for buffer controllers for SIRF and GPF. The synchronization points are a function of M.

When the value of the counter in the global controller coincides with the binary representation of the *start_time*, the corresponding buffer controller becomes activated. Thus, the global controller is simply an array of AND gates where the output of each gate controls the buffer controller. Assuming that M is 1024, we need a 12-bit counter for the global controller.

5.5.3 Reconfigurability and Parameterizability

As seen from the previous sections, execution of the hardware can be alternated between SIRF and GPF by changing buffer controller parameters and interconnect switch states. The advantage of this architecture is that the processing blocks in the design are slaves

to the buffer controllers which are simply configured by a small set of parameters. The processing blocks themselves do not implement any global controls. The buffer controllers with their appropriate parameters and global synchronization signals maintain the overall execution flow. As a result, changing parameters of the buffer controller allows for modifying individual filter characteristics dynamically between iterations within limitation of provided resources. The maximum number of particles that can be used is bounded, in case of the SIRF, by the depth of $BUF6$. The maximum dimension of the state is bounded by the number of dual port memory units in $BUF1$, $BUF2$, $BUF6$, $BUF7$ and $BUF12$. Within these bounds, the number of particles and the dimension of the state can be varied between iterations by changing the parameters and control signal timings of various buffer controllers in accordance with TABLE 5.2 and TABLE 5.3.

In practical scenarios changing the number of particles brings about a tradeoff between accuracy of the filter and the iteration period. Dynamically changing the dimension of the state is needed in multiple target tracking problems with unknown number of targets. In such problems, the sate vector represents positions and/or velocities of the targets being tracked. Depending upon the number of targets the dimension of the state changes. The proposed architecture allows for implementation of such algorithms since the dimension of the state can be changed dynamically between iterations.

The design methodology also allows for exploitation of inherent parallelizability in the PFs. The the blocks CPG, PG, $Imp1$, $Imp2$, $Imp3$ and OG perform *data parallel* computations each iteration, i.e. each block processes a large data set where the individual computations are independent of each other. Hence, these computations can be parallelized if multiple instances of the processing blocks are available such that each instance processes a fraction of the total M particles. The proposed methodology allows for easily incorporating additional processing blocks, if they are available, into the design. Using standard design methodologies, parallelizing the filters would need a major redesign. Moreover, the methodology is extremely scalable in terms of design complexity as more and more blocks are added to increase the degree of parallelizability. The resampling step is inherently sequential. However, we have developed several resampling algorithms and that allow for parallel and distributed resampling [17]. Processing blocks implementing such algorithms can also be incorporated into the design if parallelization of resampling is needed. Fig. 5.8

shows how multiple processing blocks can be included in the design using the buffer controller parameters for the SIRF. The RS block implements traditional resampling and hence cannot be parallelized. The execution period for the SIRF using parallelization reduces to $M/K+M+L_{SIRF}$ where K is the degree of parallelism. For the GPF, similar replication can be done which leads to an execution period of $(M/k + L_{GPF}) \cdot T_{clk}$. The timing parameters presented in TABLE 5.3 will scale accordingly for each buffer controller.

Parallelization provides higher speeds at the cost of higher resource and power consumption. The buffer controllers along with the interconnection switches allow for dynamically changing this degree of parallelism for power saving depending upon the speed requirement.

5.6 Physical Realization and Evaluation

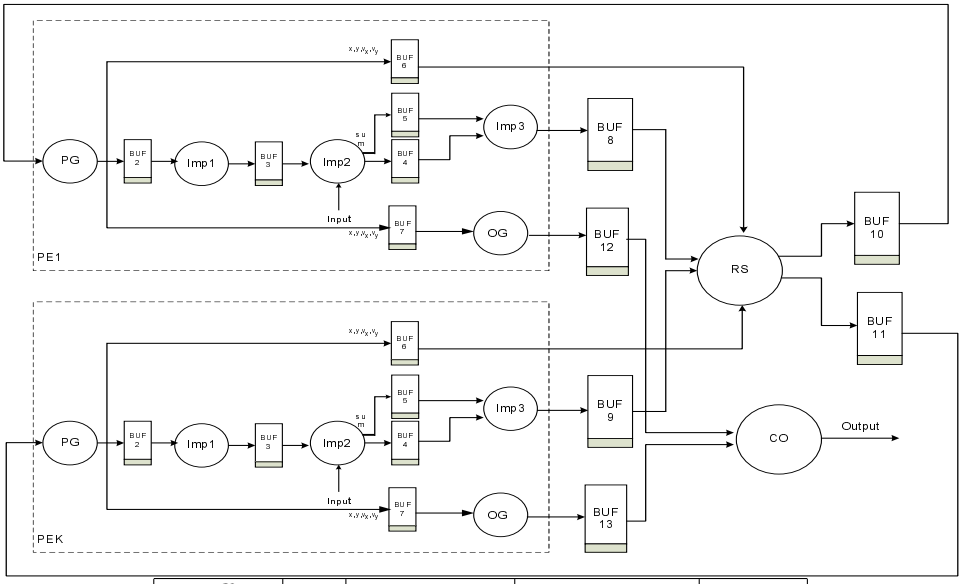
5.6.1 Processing Block and Buffer Controller Synthesis

Fig. 5.9 illustrates the percentage of power and area of the processing blocks synthesized on a Xilinx Virtex II pro device (XC2VP50). The actual achievable speed varies among the processing blocks. The global clock, f_{clock} is set to 100MHz for all the blocks for simplification of the controller design. It has been observed that overall speed is limited by the speed of the CORDIC. The power consumption of the RS block will increase with increasing M.

The buffer controller can be synthesized with either embedded BRAMs or with distributed memory on the FPGA. Table 5.4 illustrates the results for the buffer controller synthesis. As the number of word increases from 4 to 64, the number of slices increases due to larger counters and other peripheral logics. These delays are fast enough to be non bottleneck in the system integration.

Data Size	Slices	Read Delay (ns)	Write Delay (ns)
8×4	24 (44)	2.867 (2.927)	3.429 (2.038)
8×16	26 (46)	2.867 (3.291)	3.429 (2.958)
8×32	28 (52)	2.867 (3.383)	3.429 (3.680)
8×64	29 (77)	2.867 (3.711)	3.429 (3.826)

Table 5.4: Illustration of FPGA mapping result of BRAM based buffer controller for different word size. Data in parentheses are for the result of distributed memory based buffer controller.



Buffer	L_i	$nwij$	nr_{ij}	M_{ij}
BUF2	8	1	1	M/K
BUF3	23	1	1	M/K
BUF4	20	2	1	M/K
BUF5	20	$M/K + 1$	1	M/K
BUF6	8	1	$M/K + 60$	M/K
BUF7	8	1	49	M/K
BUF8	10	1	1	M/K
BUF9	10	1	1	M/K
BUF10	19	M	1	$M/2$
BUF11	19	M	1	$M/2$
BUF12	8	$M/2$	1	1
BUF13	8	$M/2$	1	1

Figure 5.8: Parallelizing SIRF execution by duplicating processing blocks.

In general, the buffer controller based on BRAM will use less logic resources than the one based on distributed memory. While the buffer controllers will not be bottleneck for the system performance, the interconnection between the them and processing blocks will be an issue. Since use of BRAM, reduces location flexibility, buffer controllers based on distributed memory reduce interconnect overhead in many cases.

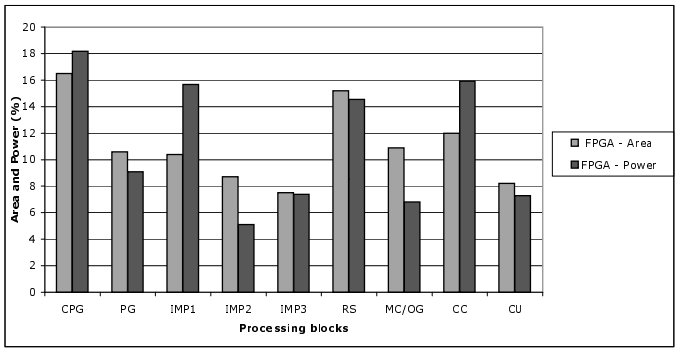


Figure 5.9: Percentage of FPGA resources of the processing blocks in terms of area and power consumption. The power is estimated at 100MHz. In the implementation, $M = 512$.

5.6.2 Execution Performance

The execution diagrams for the SIRF and the GPF of the reconfigurable architecture are shown in Fig. 5.10 and Fig. 5.11, respectively. The simulation shows the data transfer activities for two iterations of all the active buffer controllers. In the simulation, the value of M is chosen to be 256, which can be arbitrarily selected depending on the applications. In both filter realizations, the external input is synchronized with the start of the IMP2 processing block. As shown in the figures, the activities of the buffer controllers are overlapped, which indicates that the processing blocks are concurrently executed. The vertical lines in both figures represent the beginning of each iteration. It is clear that for the same M, the second iteration of the GPF starts quicker than that of the SIRF.

The performance of the reconfigurable PFs are faster because of operational concurrency in the implementation. When the concurrency is fully exploited, the GPF is much faster for large M. On the other hand, the SIRF is faster when the algorithm is executed on DSP because when these two algorithms are executed sequentially, there are more computations for the GPF. Fig. 5.12 shows two curves that correspond to the execution times for processing M particles using the SIRF and GPF algorithms. The curves represent the sampling period as a function of number of particles obtained from the implementation on Texas Instruments (TMS320C67x) processors. In sequential implementations, the sampling period increases almost linearly with the number of particles for $M = \{500, 1000, 5000\}$. We can observe that most of the speed up is from the functional concurrency exploitation and deep pipelining. While DSPs provide some degree of parallelism, functional concurrency cannot be fully

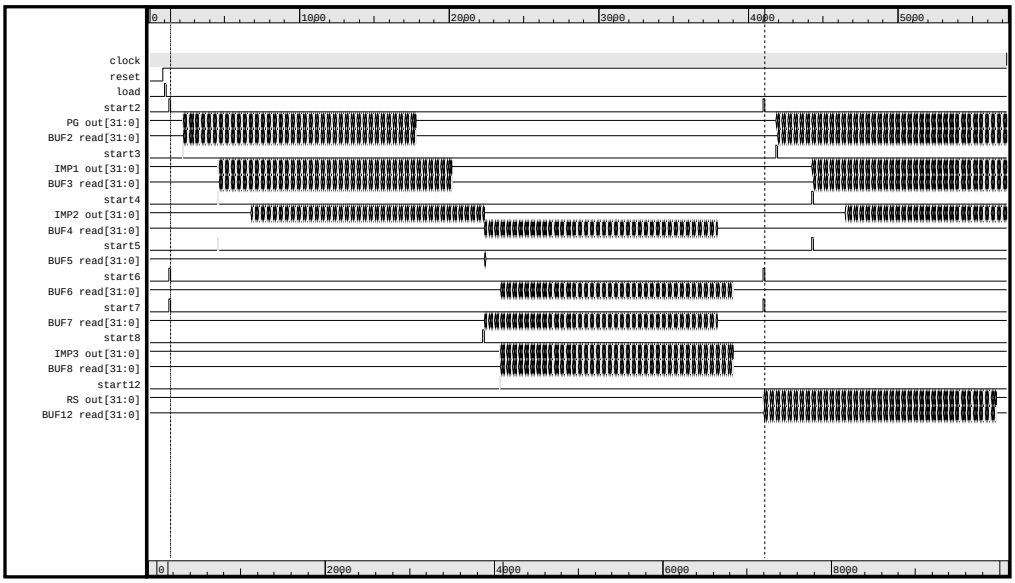


Figure 5.10: Timing diagram of SIRF. The vertical lines indicate the start of the iterations. The diagram illustrates buffer activity.

exploited.

Fig. 5.13 illustrates comparison of energy consumption. The plot is normalized so that the value indicates the energy required to process one particle. The energy for the DSP is estimated with Code Composer for instruction profiling and Power Estimation Spreadsheet. The FPGA resource power is estimated using Xilinx XPower. As shown in the figure, the energy per particle is much lower for the DSP than that of the FPGA. This is due to highly pipelined implementation for the FPGA implementation. However, as we have discussed, the potential throughput is much lower for the DSP implementation.

5.6.3 Discussion of Reconfiguration Overhead

The proposed architecture maximizes the buffer controller usage and minimizes the dynamic reconfiguration efforts. The design does not suffer from the additional memory used by buffers since we can view these buffers as pipeline registers (i.e., the registers are needed in any design with standard design flow whether it is distributed or centralized). In comparison to the processing with DSP, the energy consumption for FPGA will be more as it is using more resources and performing operations concurrently. These additional resources are necessary when the is of utmost concern.

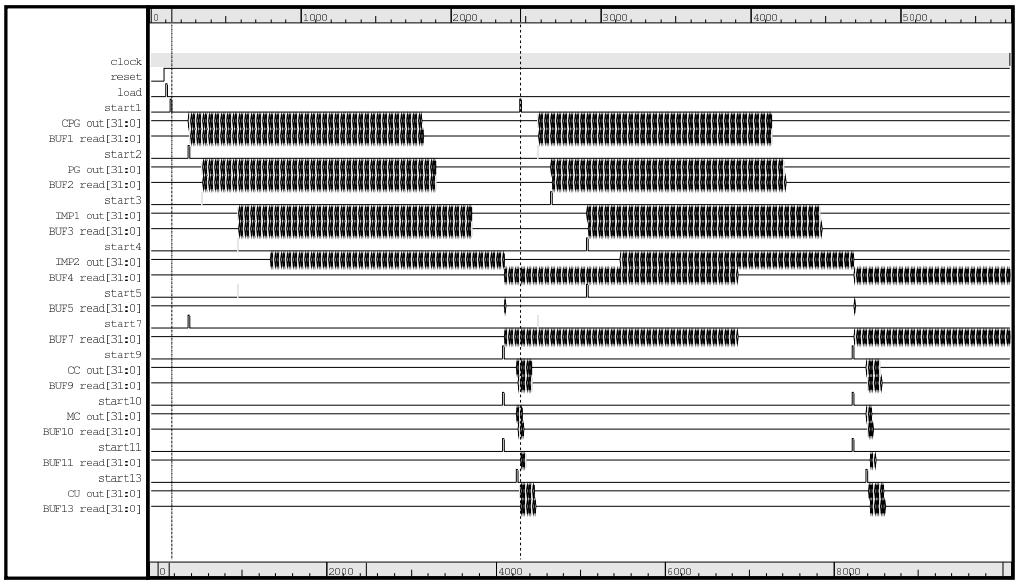


Figure 5.11: Timing diagram of GPF. The vertical lines indicate the start of iterations. The diagram illustrates buffer activity.

When we consider FPGA as the target platform, two separate implementation will take up much more area than the proposed design. In terms of power consumption and speed (i.e., when we compare SIRF of the proposed design vs. fixed SIRF), power dissipation due to the processing elements are identical and the speeds are also same because overall throughput is limited by the processing elements not buffer controller.

In the proposed design, 13% of resources are used for buffer controllers (with design targeted for $M = 512$). Usually, this percentage will go up for finer processing elements but will go down for coarser processing elements. Fixed SIRF design would use 71% of the resources used in the proposed design. Similarly, fixed GPF design would use 82%. Thus, if these two are implemented separately, it would use 50% more resources than that of the proposed combined design. This will go up if we implement more than two types of particle filters since we can still share many processing elements and buffer controllers.

In terms of speed, there is no gain for the fixed design since the speed bottleneck is due to processing elements not the buffer controllers. Moreover, power dissipation would not change when we compare the combined design to that of fixed designs. This is because only the processing elements for a specific particle filter are active.

In the proposed architecture, because of the very small amount of information associated

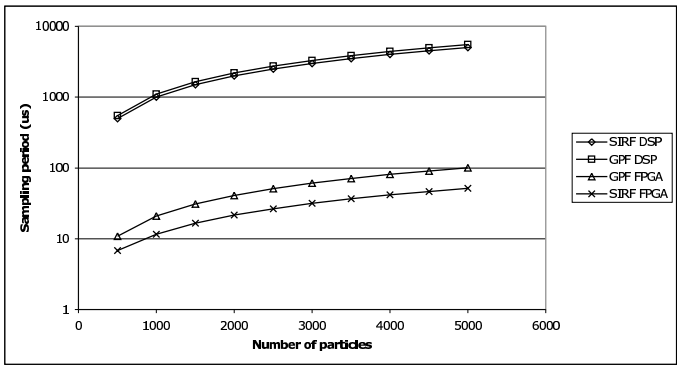


Figure 5.12: Sampling period of the reconfigurable PFs (GPF and SIRF) versus number of particles. The DSP version of the filters are implemented on TI TMS320C67 Series processor.

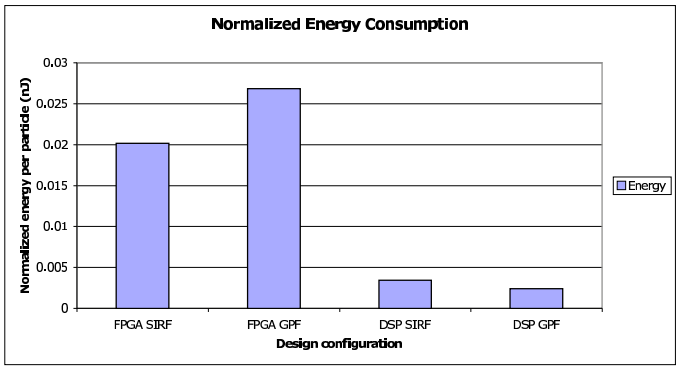


Figure 5.13: Normalized energy consumption of different design configuration. The energy is normalized for one particle. $M = 512$ for FPGA implementation.

with each structure (i.e., one set of data for each buffer controller and global controller), the reconfiguration time is almost non-existent (i.e., all the parameters for the controller and structural switch can be loaded with a few clock cycle but as low as one cycle simultaneously). This is illustrated in the simulation diagram showing that parameter loading takes a few cycles before the processing can begin. This is an attractive attribute since the execution flow of the algorithm can change without redesigning the overall controllers.

5.7 Conclusions

In this chapter, we introduced a buffer controller based design methodology for overall synchronization in reconfigurable PF realizations. The controller configuration is simple and

systematic such that the reconfiguration is significantly simplified. We illustrated the effectiveness of the reconfiguration by considering two different types of PFs. We have demonstrated that by completely controlling the data transfer behavior, the processing blocks become mere slaves of the buffer controllers in the overall execution. Moreover, the design strategy of the processing blocks is well defined to satisfy the design methodology. The design can be extended to support many different PFs. The reconfigurable processor outperforms conventional DSPs.

Chapter 6

Extension of Architecture to Multiple Model Particle Filtering

Multiple model (MM) filtering methods are used to estimate the states of dynamic systems that are inadequately described by a single model. In this Chapter, we present an algorithm and a hardware architecture for the traditional particle filter, i.e., the Sampling Importance Resampling Filter (SIRF), applied to systems with multiple interacting models. Due to superior abilities of the SIRF to handle nonlinear non-Gaussian models, such filters outperform traditional MM filters in several practical scenarios. We propose a parallel architecture for the hardware implementation of this algorithm consisting of distributed processing elements (PEs) performing model based operations and a central unit (CU) performing centralized process of resampling as required by the algorithm. This centralized processing is inherently sequential and requires a large amount of data to be exchanged between the PEs and the CU. We alleviate this problem by using a novel distributed resampling method that parallelizes the resampling process by distributing it to the PEs hence speeding up the resampling process and reducing communication requirement. Thereafter, a data exchange method is proposed that reduces the required interconnect to a single bus without causing communication bottleneck. The proposed architecture is evaluated on a Xilinx FPGA platform for a multiple model target tracking application.

It has been established in the literature, that multiple model filters used to track targets in practice, require a very large number of models to "cover" all possible target maneu-

vers. Moreover, the models that best describe the target dynamics change with time. This causes a heavy computational burden due to the large number of models, and also degrades performance since a large number of diverse models "compete" with each other during the estimation. In order to overcome this problem, the concept of adaptive or variable structure multiple models filters (VSMMF) has been introduced [60]. Under this framework, filters incorporate an *active model set* that changes over time. This adaptability can be incorporated into the proposed MM particle filtering architecture by exploiting the concept of reconfigurability. However, due to the sequential, on-line estimation nature of the particle filter, any reconfiguration of the hardware will need to be done without disturbing the execution flow of the particle filter. Recently, Xilinx FPGA devices have incorporated the feature of Partial Run Time Reconfiguration (PRTR) whereby, part of the FPGA can be reconfigured while the rest of the FPGA is still functioning as normal. This feature can be used to extend the proposed MM particle filtering architecture to a VSMM particle filter. However, mapping of the architecture to the Xilinx FPGA in order to utilize the PRTR feature is not straightforward. We have developed a systematic procedure for this mapping in the context of the particle filter.

6.1 Introduction

In model based filtering methods the unknown state of a system is estimated in time based on the received observations. These models have a state equation that describes the evolution of the sate (e.g. position of a moving target) and an observation equation that describes the measurements as a function of the state corrupted by noise. Frequently, several systems require a multiple model formulation of the state dynamics. In practice these models may be nonlinear and non-Gaussian. The interacting multiple model (IMM) methods have been extensively applied for state estimation in such scenarios [77]. They are suboptimal methods which approximate the posterior density of the state as a Gaussian. The traditional IMM filters consist of a bank of simultaneously operating Kalman filters (KFs) or extended Kalman filters (EKFs), each tuned to a different model. In addition to the state and measurement equations, these filters also require knowledge of the model transition probabilities which are used for combining individual filter estimates and initializing future filter recursions. In case

of nonlinear models, the traditional IMM filters perform poorly due to the shortcomings of the EKF. Alternative approaches for nonlinear and non-Gaussian IMMs include the unscented Kalman filter (UKF), [58], and the regularized particle filter (RPF) [16]. Some practical applications of the MM filtering approach are: tracking of maneuvering mobile station in CDMA environment [68] and tracking of maneuvering vehicles for adaptive cruise control [58].

Here, we use an SIRF based algorithm for dynamic systems described using multiple interacting models. A similar approach has been applied in [78]. Compared to this method, our algorithm does not require knowledge of the model probabilities and keeps a constant number of particles per model at all times. Our focus, is on development of a parallel architecture for the efficient hardware implementation of this algorithm which will enable its use in practical systems requiring real time processing. The architecture consists of distributed processing elements (PEs) and a central unit (CU). We use a novel distributed resampling approach which drastically reduces the data exchange requirement between the PEs and the CU and also parallelizes the inherently sequential resampling process. This significantly reduces the resampling time and increases the speed of the filter. The communication bottleneck and required interconnect density is reduced by using a mechanism based on distributed particle storage and index addressing to bring about the data exchange. The modular and distributed nature of this architecture with specific modules that are model dependent and others that are generic makes it amenable to partial reconfiguration on FPGA platforms so as to incorporate different models at different times with minimal redesign.

6.2 The MM SIRF algorithm

The multiple model system is described using multiple DSS models as

$$\mathbf{x}_n = f_n^k(\mathbf{x}_{n-1}, \mathbf{q}_{n-1}) \tag{6.1}$$

$$\mathbf{z}_n = h_n^k(\mathbf{x}_n, \mathbf{v}_n) \tag{6.2}$$

where $\mathbf{x}_n$ describes the dynamically evolving state of the system at time n, f_n^k is the possibly nonlinear model dependent system transition function, $k = 1, 2...K$ represents the model in-

dex, where K is the total number of models used. The symbol $\mathbf{z}_n$ represents the observations of the system, h_n^k is the possibly nonlinear model dependent observation function, and $\mathbf{q}_n$ and $\mathbf{v}_n$ are state and observation noise processes, respectively, that may be non-Gaussian. Unlike other multiple model filtering approaches, we *do not* consider a model for the transition probabilities. The prior model probabilities and transition probabilities are respectively assumed constant while the posterior model probabilities are accounted for implicitly in the algorithm as shall be seen. The input to the filter at every sampling instant is the observation $\mathbf{z}_n$. The goal of the filter is to estimate the state using this observation and the multiple model state description.

In our algorithm, the traditional SIRF is extended to multiple models. In the sample step a set of particles is drawn from each model. This is represented as $\left\{\mathbf{x}_n^{(i),(k)}\right\}_{i=1}^{N_s}$ where N_s is the number of particles used in *each* model and k is the model index number. The importance step in each model assigns a weight to each of its particles based on the received observations. Thus after these steps, we have a weighted set of particles from each model k, represented as $\left\{\left(\mathbf{x}_n^{(i),(k)}, w_n^{(i),(k)}\right)\right\}_{i=1}^{N_s}$. This set represents the individual model conditioned approximation to the state posterior.

We use the resampling step not only to avoid the generic weight degeneracy, but also to combine the weighted sets of particles from all the models. This resampled set of particles represents the combined posterior of the state over all models and can be used to compute the desired estimate. The resampling operation selects N_s of the total $K \times N_s$ sampled particles. This resampled set is denoted by $\left\{\widetilde{\mathbf{x}}_n^{(i)}\right\}_{i=1}^{N_s}$, where the particles are chosen such that

$$Pr(\widetilde{\mathbf{x}}_n^{(i)} = \mathbf{x}_n^{(j),(k)}) = w_n^{(j),(k)} \tag{6.3}$$

where $i, j = 1$ to N_s and $k = 1$ to K. The weight of each resampled particle is $1/N_s$ [29]. The posterior probability of each model is accounted for implicitly in this resampling step by the fact that the model with the largest weight will contribute the highest number of particles to the combined resampled set and hence to the final estimate. This *combined* set of N_s particles is then propagated to the sample step of each model to determine the sampled particles of the next time instant using (1). This brings about model interaction and makes

the algorithm robust as the space of each model is explored at every instant.

The above algorithm was applied to the problem of tracking a moving vehicle with two models described as in [58]. A constant velocity model describes straight line motion and an almost-constant speed turn model covers vehicle maneuvers like U turns and rotaries. The results of the tracking, with $N_s = 1000$ particles per model, averaged over 100 Monte Carlo runs are presented in Fig. 6.1. The above example with $N_s = 1000$ and $K = 2$, when evaluated on a DSP platform (TI-TMS320C67x), gave a processing speed of about 200Hz. Hence design of efficient hardware is essential to meet real time processing requirements, particularly in situations where a larger K and larger N_s are needed.

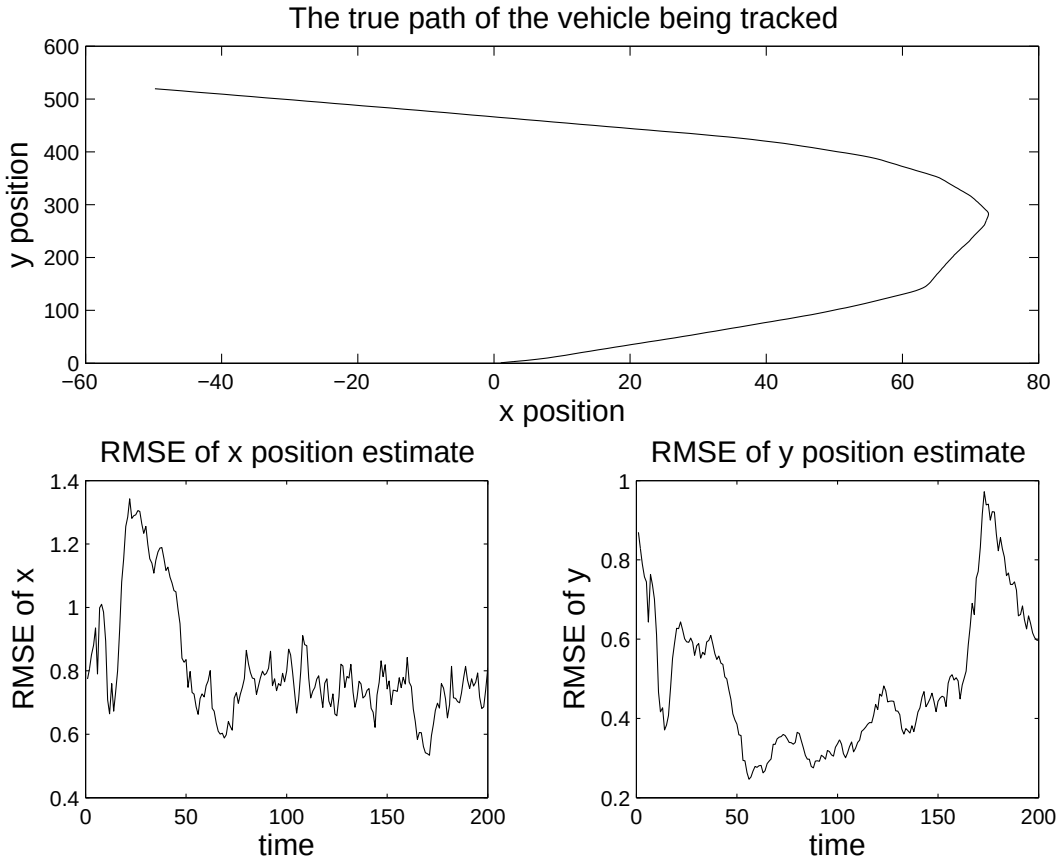


Figure 6.1: Performance of the algorithm on a tracking example.

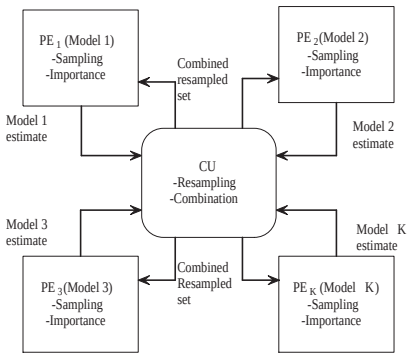


Figure 6.2: Parallel architecture model for the algorithm.

6.3 Parallel Architecture Framework

Fig. 6.9 shows a basic parallel framework for the implementation of the multiple model SIRF algorithm. The various operations mapped to the PEs and the CU are shown in the figure. Each PE is tuned to one particular model and performs the model dependent steps, i.e., the sample and importance steps for that model simultaneously with other PEs. The CU performs resampling and brings about the combination of the individual model estimates.

Resampling is a centralized operation which operates on particles and weights from *all* the models (PEs) to produce a combined resampled set. All traditional resampling algorithms are inherently sequential and cannot be parallelized to operate simultaneously on all PEs. Accordingly, if the traditional systematic resampling [29] is used, it would require all the $K \times N_s$ particles along with their weights to be sent from the PEs to the CU where they would be stored. Resampling would then need to systematically choose N_s out of the total $K \times N_s$ particles. From results presented in [6], this resampling would take $((K+1) \cdot N_s)$ cycles. TABLE 6.1 summarizes the amount of communication needed between PEs and CU if centralized resampling is used. The symbols b_w and b_p are the bit widths used for representing the particles and weights in fixed point format.

Data Transferred	Direction of Transfer	Amount
Weights	From *each PE* to CU	$K \times N_s \times b_w$
Sampled Particles	From *each PE* to CU	$K \times N_s \times b_p$
Resampled Particles	From CU to *each PE*	$N_s \times b_p$

Table 6.1: Data exchange requirement between PEs and CU if centralized resampling is used.

It is reasonable to assume b_p and b_w to be of the order of 16 or more. The number of used particles N_s is typically of the order of 1000 or more. Thus the traditional centralized resampling approach has a very high data communication requirement, introduces a high resampling latency and also poses a significant memory requirement in the CU. This affects the scalability of the architecture since increasing the number of models or PEs K, increases the data exchange requirement, resampling time and CU memory requirement by a factor of N_s. This scheme also poses a serious bottleneck since resampling cannot start until all particles and weights are sent to the CU and the sample step of the next instant cannot start till the resampling step of the previous instant, which takes $(K+1) \cdot N_s$ cycles is complete.

6.4 Distributed Resampling

To alleviate the problems of centralized resampling, we use the method of *distributed resampling*. This method is explained for the traditional (single model) SIRF in [17]. The amount that each model (PE) contributes to the set of N_s resampled particles depends upon the weight of the PE, i.e., sum of weights of all particles in that PE. Distributed resampling makes use of this fact to split up resampling into a two stage hierarchical process. Initially, only the *sum of weights of all particles* of each PE, is sent to the CU. This is denoted by W^k for PE_k. The CU then performs the first stage of resampling which determines the *number of particles* that each PE will contribute to the final resampled set based on its sum of weights. We call this value as the PE replication factor and denote it as R^k for PE_k. It is important to note that $\sum_{k=1}^{K} R^k = N_s$ and $0 \leq R^k \leq N_s$. This operation is done using the method of Residual Systematic Resampling (RSR), proposed in [17]. The hardware implementation of this algorithm along with a modification that avoids the need for weight normalization is presented in [6]. The following steps are performed during RSR in the CU to determine R^k.

$(R^k) = RSR(N_s, \{W^k\}_{k=1}^K)$

1. Generate a random number $U \sim \mathcal{U}[0,1]$
2. Calculate $S = \sum_{k=1}^{K} W^k$
3. $C = N_s/S$
4. *for* $k = 1$ *to* K
5. $\quad temp = W^{(k)} \cdot C - U$
6. $\quad R^k = \lceil temp \rceil$
7. $\quad U = R^k - temp$
8. *end*

Residual systematic resampling (RSR) algorithm to determine R^k.

The R^k values are then sent to the respective PEs. Each PE_k performs resampling on its N_s sampled particles to produce R^k particles simultaneously with the other PEs. This resampling is still sequential, but has reduced execution time since only N_s particles are processed simultaneously in each PE as opposed to $K \times N_s$ in the CU if centralized resampling is performed. We would like to stress here that distributed resampling is *not an approximation* and the results of centralized and distributed resampling are the same. In distributed resampling, the communication between PEs and the CU consists only of K values of sum of weights W^k and K replication factors R^k. This is a great reduction in comparison to the values presented in TABLE 6.1.

After distributed resampling, each PE contains a portion of the N_s resampled particles (R^k in PE_k). The sample step in each PE requires all the N_s resampled particles of the previous time instant. Hence each PE_k needs to obtain $N_s - R^k$ particles from the remaining PEs. We shall see in the next sections how this exchange is carried out efficiently without causing a communication bottleneck.

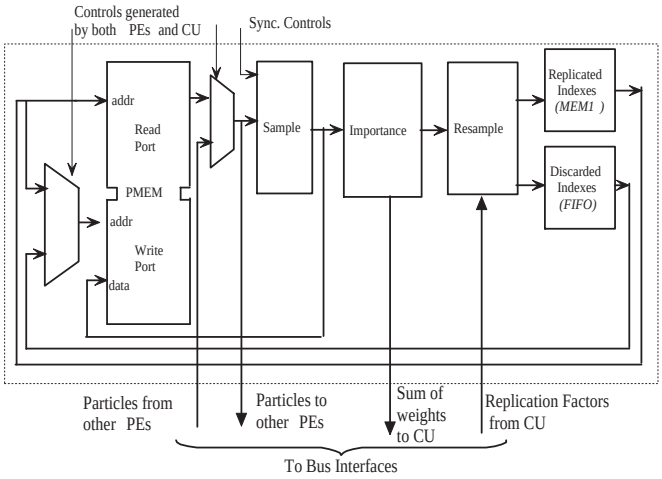


Figure 6.3: Architecture of a single PE

6.5 Architecture Description

6.5.1 Structure of PE

Fig. 6.3 shows the basic architecture of each PE. This is based on the memory schemes proposed in [8]. First N_s sampled particles are drawn in *each* PE using the *combined* set of N_s resampled particles from the previous instant. Weights of the particles are calculated by the importance step and are conveyed to the local resampling unit. After receiving the number R^k from the CU, resampling is done to determine which R^k of the N_s particles of the PE_k will be present in the final resampled set. The process returns the addresses or indexes of the R^k resampled particles in $PMEM$. They are written to the replicated index memory $MEM1$ while the other indexes are written to the discarded index FIFO. Thus after distributed resampling, the replicated index memory in each PE holds the R^k indexes of those particles in *its memory* $PMEM_k$ that are a part of the combined resampled set. The discarded particles indexes are used to write the sampled particles to $PMEM$ without inappropriately overwriting resampled particles [8], [6].

6.5.2 Inter-PE Communication

Fig. 6.4 shows the communication between the PEs in the form of abstract channels. Due to the nature of the algorithm, it is likely that one of the models is dominant at some

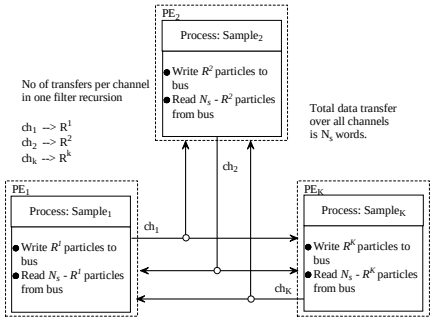


Figure 6.4: Communication channels between PEs.

instants and has a higher R^k than others. In such cases, there will be a large number of particles communicated from this PE to all others while communication between other PEs will be negligible. Due to these factors, having a dedicated bus from each PE to every other (complete connection) results in most of the interconnect being unused. The utilization is maximized when a single bus is used for the particle exchange. Particle distribution is then sequential and takes $N_s \times t$ cycles where t is the number of cycles needed to transfer a word (particle) over the bus. This depends upon the bus latency and the width of the bus B_w with respect to the b_p, the bit width used for representation of particles. Using the architecture of Fig. 6.3, this particle distribution is pipelined with the sample and importance steps. Hence communication bottleneck can be completely avoided if $t = 1$ cycle and $B_w = b_p$, i.e., one particle is available at the input of the sample step of each PE per cycle. If $t > 1$ cycle, then a larger bus width will need to be chosen to prevent bottleneck. We target an FPGA implementation where the PEs and the bus are located on a single chip and $t = 1$. Hence we choose $B_w = b_p$ for our implementation. For a general case in which PEs may be different processors with varying characteristics, the required bus specifications can be determined using the methodology presented in [13]. This work provides a methodology for calculating bus specifications to realize a communication of the form of Fig. 6.4 using a single bus.

6.5.3 Communication between PEs and CU

Fig. 6.5 shows the communication between PEs and CU in the form of abstract channels. The required communication between PEs and CU in each recursion is only $2 \cdot K$ words; K values of W^k from PEs to CU and K values of R^k from CU to PEs. Typical values of K

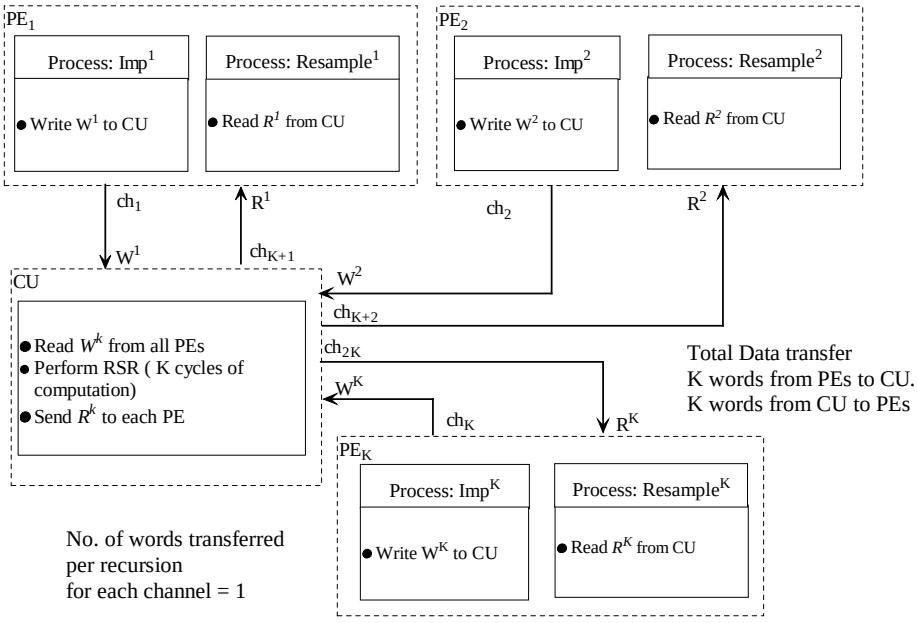


Figure 6.5: Communication channels between PEs and CU.

are between $2-6$. The RSR process in the CU has a computation time of K cycles. The PE-CU data exchange does not overlap with inter-PE communication. Hence we use the same bus of width b_p, that is used for communicating particles between PEs for the PE-CU communication of $2K$ words. Though fixed point width for various quantities depend upon the application, we have arrived at the following general relations by applying the statistical analysis method of [58]

$$b_p \approx b_w \tag{6.4}$$

$$b_{W^k} \approx 2 \times b_w = 2 \times b_p \tag{6.5}$$

$$b_{R^k} = log_2 N_s << b_{W^k} \tag{6.6}$$

where b_{W^k}, b_w and b_{R^k} are the widths used in the fixed point representation of the sum of weights, individual weights and the replication factors, respectively.

Applying the analysis of [13] to the communication of Fig. 6.5, gives us a required bus width of $K \times max(b_{W^k}, b_{R^k}) = K \times b_{W^k}$ to prevent a communication bottleneck. Using a bus of width b_p causes a communication latency of $K \times \lceil \frac{b_{W^k}}{b_p} \rceil$ in sending the sum of weights

to the CU and a latency of K cycles in obtaining the replication factors. Due to the small value of K, this latency is of the order of $10 - 20$ cycles which is negligible compared to the overall execution time of the filter.

6.5.4 Arbitration: The role of the CU

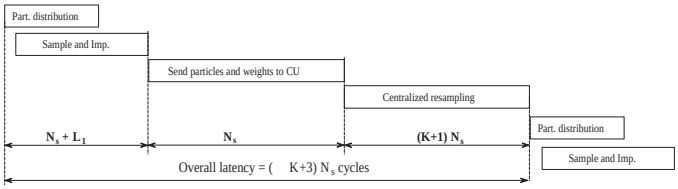
Each PE contains R^k resampled particles which need to be sent to other PEs over the bus. The values of R^k are determined in the CU and these values decide how many particles each PE will write to the bus. The CU incorporates a controller which consists of a counter and comparators which compare the output of the counter with the values R^k. Accordingly, control signals are generated which grant access of the bus to each PE_k for R^k cycles. During this time the sample step of this PE reads particles from the local memory while the sample step of the other PEs get their input particles over the bus. Due to the pipelining of the particle distribution with sample and importance steps, the time from the start of a recursion till all the weights of particles within a PE are calculated is $N_s + L_1$ cycles where L_1 is the start up latency of the PE datapath. The resampling in the PE takes $2 \times N_s$ cycles if systematic resampling is used and RSR operation in the CU takes $K + 2$ cycles (latency of RSR datapath is 2 cycles). This information is incorporated into the central controller to generate control signals for PE to CU transfer of sum of weights and CU to PE transfer of replication factors.

6.6 Evaluation

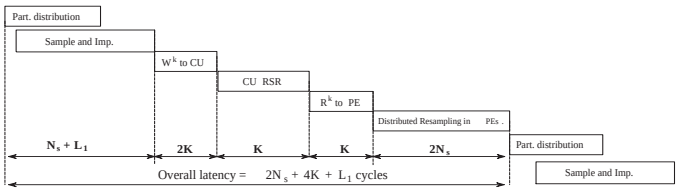
The proposed architecture drastically reduces the communication bottleneck and resampling time thus greatly speeding up the filter execution. Fig. 6.12 compares the timings of the filters using a centralized resampling approach and the proposed architecture with distributed resampling. The overall execution time of the filter for the two cases is given by

$$T_{cent} = (K + 3) \cdot N_s + L_1 \ (cycles) \tag{6.7}$$

$$T_{dist} = 3 \cdot N_s + 4 \cdot K + L_1 \ (cycles) \tag{6.8}$$



(a) Using centralized resampling.



(b) Using distributed resampling.

Figure 6.6: Timing of multiple model SIRF using (a) centralized and (b) distributed resampling.

where L_1 is the latency of the PE datapath. Since value of L_1 will be different for different PEs, the maximum L_1 over all the PEs is used in characterization of the filter. Additional buffering is used at other PEs for synchronization. The value of L_1 for our FPGA implementation with the bit widths mentioned in the previous sections was 75 cycles. This architecture is highly scalable in that, increase in execution time by inclusion of more PEs and more particles per PE, is extremely small as compared to a centralized resampling approach. Fig. 6.7 shows the variation of the filter execution time with varying number of PEs and particles per PE for the two cases.

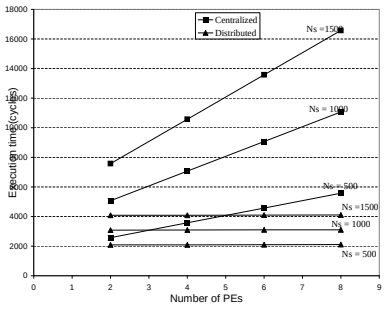


Figure 6.7: Scalability of proposed architecture.

Fig. 6.8 shows the overall architecture of the filter with two PEs along with timing diagrams showing the status of the bus and associated controls during the inter-PE and

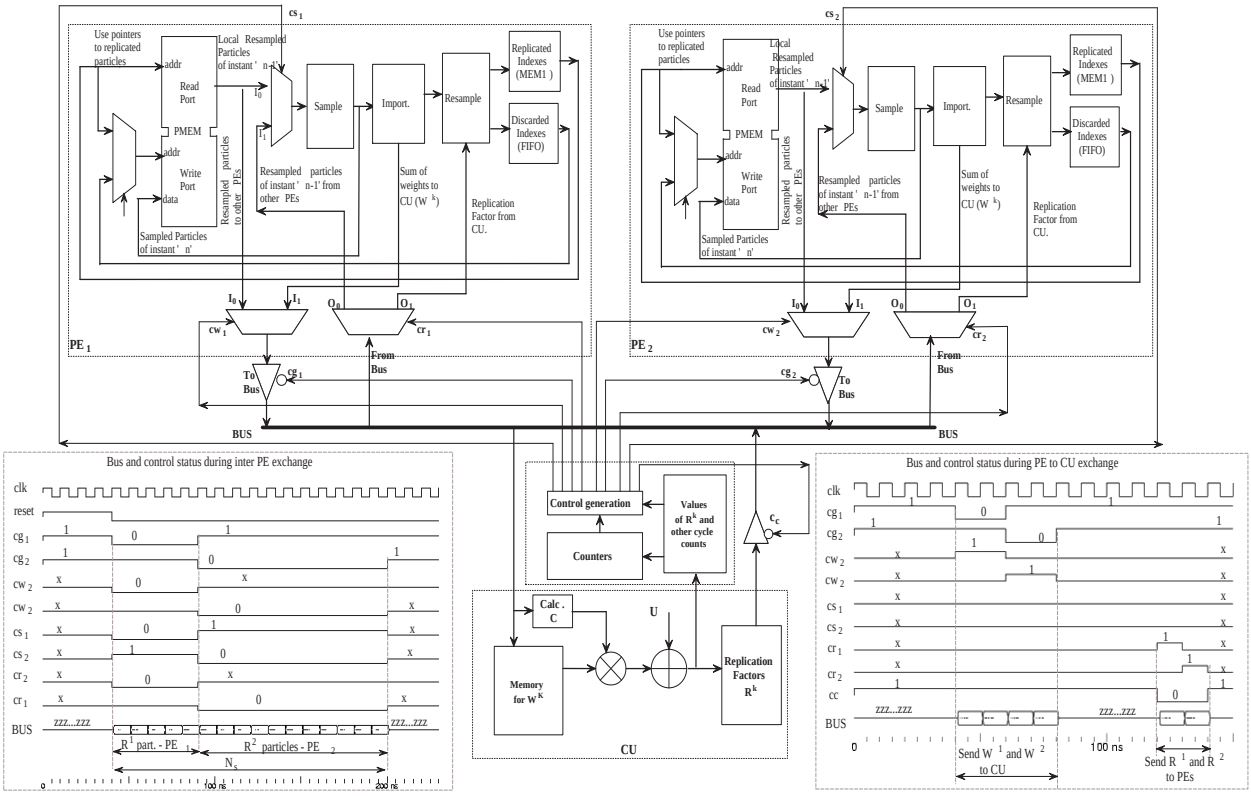


Figure 6.8: Full Architecture with two PEs.

CU-PE communication for a hypothetical case of $N_s = 16$. The same architecture can be extended for incorporating more PEs within the resource limitations of the target platform. Any bus architecture that supports a single master and multiple slaves can be used to realize the required interconnection. We target an FPGA platform for this implementation and realize the bus using macros which utilize long routing lines and tri-state buffers available on the FPGA platform. Using the above architecture, the MM SIRF was implemented on a Xilinx Virtex II device for the tracking application [58] discussed in Section 6.2 with $K = 2$ and $N_s = 1000$. All the memory required in the PEs is realized using block RAMs. All trigonometric and exponential functions are implemented using CORDIC units.

Unit	Slice	Slice.Reg	LUT	B. RAM	Mult	TBUF
PE_1	2,807	4,133	3,873	17	7	52
PE_2	4,597	6,872	5,423	17	10	52
CU	520	550	610	0	1	10

Table 6.2: Resource Utilization on XC2V6000 device

In Fig. 6.8, only the *"Sample"* and *"Import."* blocks are model dependent while the rest of the architecture is generic. TABLE 6.2 summarizes the resource utilization of various units on the target platform. Post place and route timing analysis determines that the maximum clock rate for this design is 60MHz. This means that for this example, the filter can process input observations at 20KHz.

6.7 Variable Structure Multiple Model Particle Filters

In most practical scenarios, using a fixed set of models for a tracking problem requires a very large number of models. This increases the computational burden and also results in degrading performance due to conflicting estimates of diverse models. To overcome this, Variable Structure Multiple Model (VSMM) filters are often used in practice. These filters have a base set of several models, but only a few of these, called the *active model set* are using for filtering at any given time. To make the proposed hardware for MM particle filters feasible in practice, model set adaptability must be incorporated in the hardware. This can be achieved by using the conept of reconfiguration. However, the sequential nature of the particle filter demands that this reconfiguration be done without disturbing the execution flow. For this purpose, we have utilized the PRTR feature recently introduced in Xilinx FPGAs.Using this feature, it is possible to modify a part of the FPGA while the rest of the FPGA is still functioning. However, access to the Xilinx FPGA for RTR is not completely flexible and needs a systematic partitioning of the FPGA fabric into static and reconfigurable partitions within certain pretty tight constraints. We have developed a methodology for mapping the particle filter dataflow to this FPGA so as to enable PRTR in the MM particle filtering architecture. We develop a cost function that guides the mapping of the dataflow nodes to these partitions by attempting to minimize the communication cost while ensuring that the required RTR is completed within certain time constraints.First, we describe the methodology for any real time signal processing application in general and then use it to map a simple particle filter dataflow to the FPGA with RTR.

6.7.1 Partial Run Time Reconfiguration with Xilinx FPGAs

Partial run time reconfiguration is defined as the ability to modify a part of the circuit "on the fly" while other parts are in operation. This ability is of great interest in adaptive real time signal processing applications. These applications can be specified as data flow graphs. We consider data flow graphs at module level granularity with individual nodes representing the processing modules and the edges representing the data dependency and communication requirement between them. Several adaptive signal processing applications, require the operationality of some of the modules to be changed from iteration to iteration while the structure of the data flow remains largely fixed. Partial run time reconfiguration (RTR) brings about an efficient hardware realization of such adaptive data flows.

Recently, commercially available Xilinx FPGAs have started to support partial RTR enabling certain parts of the FPGA to be reconfigured while the other parts are functioning. Realization of a data flow on such FPGAs can allow RTR of adaptive modules without having to suspend processing. However, in reconfigurable Xilinx FPGAs, any random region of the fabric cannot be configured at run time. Partial RTR requires virtual partitioning of the FPGA into a static portion and *columns* which can be reconfigured at run time. The time required for reconfiguration of a particular column is proportional to size of the bitstream configuring that column, which in turn depends upon the size (resource requirement) of the modules mapped to that column. Communication between modules located in different columns must happen over specialized bus macros. As we shall see, the use of these bus macros affects the design efficiency and resource usage in several ways. This communication cost can be reduced by moving nodes with high amount of data exchange requirement to the same partition. This increases the reconfiguration time of that partition while reducing the communication cost. We present a hueristic (cost function) based technique for mapping adaptive data flows to the Xilinx FPGAs by optimizing the communication cost within the allowable RTR time constraints.

There have been some efforts at enabling and optimizing RTR for hardware in the recent past. RTR has been introduced for Xilinx FPGAs in [112]. A design procedure for customized long-line based bus macros has been outlined in [102]. In [50] a method for implementing LUT based macros is presented. Design flows and methodologies for realizing dynamically reconfigurable applications on Xilinx FPGAs are presented in [31],[32]. Realization of partial

RTR has led to the concept of self reconfiguring systems based on FPGAs. In such systems a microcontroller which is embedded on a poriton of the FPGA fabric, is responsible for dynamically reconfiguring the remaing portion of the FPGA as per the demands of the appiclication. One such self reconfiguring system is described in [107]. The problem of spatial partitioning of data flows to heterogeneous architectures has been handled in various works like [56], [79]. In all such works, a given data flow is mapped to physically separated units like multiple processors and FPGAs. We consider the problem of mapping a data flow to a single Xilinx FPGA, virtually partitioned for reconfiguration purposes, to efficiently achieve RTR. Examples of signal processing applications which can be realized on Xilinx FPGAs using this method can be found in [100].

6.8 Problem Formulation

6.8.1 Target Architecture

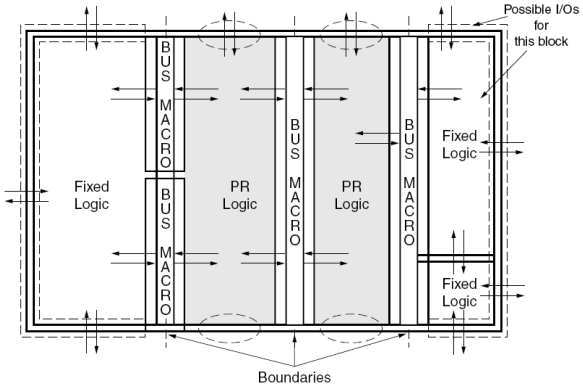


Figure 6.9: Architecture for using Xilinx FPGA for partial RTR[111].

Figure 6.9 shows the architecture of the Xilinx FPGA for achieving RTR [111]. On the Xilinx platform, only a column-wise access to the FPGA fabric for RTR is permitted. Restrictions require these columns to occupy the full height of the FPGA and be at least 4 CLBs wide ("RTR resolution"). Those modules of the data flow that require RTR need to be mapped to such columns and other modules can be palced in the static portion of the FPGA. An important consideration of this architecture is that the communication between any modules in different reconfigurable columns, or between a module in a reconfigurable

column and a module in the static portion of the FPGA must happen through the so called bus macros. These macros use the long line routing and connect to the modules through tri state buffers. The physical locations of these macros and of the reconfigurable column boundaries *must remain fixed*. As a result, modules in different columns although directly communicating, cannot use the local routing lines. This causes the routing between the modules to be spread out on the FPGA thus increasing the routing delay and reducing the utilization efficiency. If there is a large amount of data exchange between modules in different columns, the number of long lines on the FPGA may override limit on the available resources. This means that there may be cases when majority of the logic resources will be unused but the design size is limited due to the non availability of long lines. Recently, LUT based macros have been introduced for this communication between different virtual partitions [50]. However, these macros require design of arbitration units which increase resource utilization and deisgn complexity. This cost of inter partition communication based on bus macros can be reduced by

1. Reducing the number of reconfiguration partitions by mapping modules requiring RTR to same partition when possible.

2. Placing static modules having large communication requirement with a reconfigurable module in the same reconfigurable column and configuring them with the same context each time the RTR of the column is done.

RTR requires that reconfiguration of a column be done while the other parts are operating. Both the above options reduce the communication cost but increase the amount of reconfiguration information in a single column. This will eventually lead to a violation of the RTR time constraint. Hence there is a tradeoff between communication cost and reconfiguration time.

6.8.2 Data-flow Mapping

Figure 6.10 shows a generic modular granularity data flow graph for an adaptive signal processing algorithm which is to be realized on the Xilinx FPGA with RTR ability. The shaded modules of the data flow indicate those modules that may need to be reconfigured

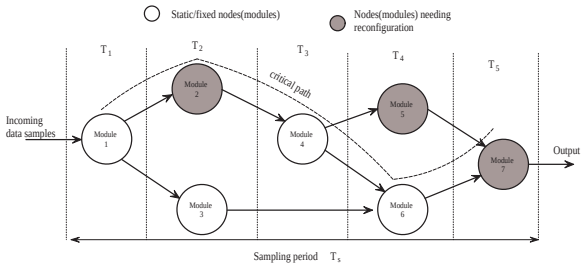


Figure 6.10: Generic real time processing data flow with nodes that need dynamic reconfiguration.

from iteration to iteration under the control of an external/internal processor. The edges denote the data dependencies between the various operations and indicate the communication requirement. The data dependencies of the graph govern the scheduling of operations into time slots.The processing times of the nodes on the critical path determing the overall latency/sampling period T_s.

The objective is to reduce the bus macro communication requirement while satisfying the RTR time constraint, which requires that reconfiguration of a particular column be completed while other modules are in operation so as to prevent suspension of processing during reconfiguration. The solution to the problem is a partitioned graph indicating the required number of reconfigurable columns on the FPGA along with the mapping of various nodes to these columns. Like the traditional partitioning problems, this problem is $\mathcal{NP}$-complete. Hence we need to use heuristic based methods to obtain a solution.

6.9 Cost Function

We use the Simulated Annealing algorithm to find a solution to the problem formulated in the previous sections. The cost function we choose accounts for the cost of communication using bus macros. The cost function is defined as follows:

$$Cost(S) = \sum_{i=1}^{N} \sum_{\substack{j=1 \\ j \neq i}}^{N} (1 + w \cdot \gamma_{ij}) \alpha_{ij} \cdot \lambda_{ij} \cdot \frac{c_{ij}}{C_A} \tag{6.9}$$

where $Cost(S)$ denotes the cost of a particular solution (S) which is a partitioned data flow graph. N is the total number of nodes (modules) in the data flow. λ_{ij} is a binary indicator variable which is 1 if modules i and j communicate and 0 otherwise. The cost of the communication needs to be considered only if the two modules lie in different partitions (columns) of the FPGA. This is accounted for by the indicator variable α_{ij}. If two modules lying on the critical path are put into different partitions, then the increased communication delay between them may imply an additional cost of increasing the overall latency. This is accounted for by the user defined factor w. If a high value of w is chosen, greater effort is made to map critical path modules to the same partition. If the overall latency is not a critical design issue, a small value of w can be chosen. A uniform effort is then made to minimize communication cost irrespective of whether the communication is along the critical path or not. The indicator variable γ_{ij} is 1 if the modules i and j are a part of the critical path. c_{ij} represents number of lines (bits) needed for communication between i and j and C_A is the total number of long lines in available on the chosen FPGA.

This cost needs to be optimized under the constraint that the reconfiguration time is within the required limits to prevent suspension of the processing during the reconfiguration. The time required for reconfiguration of a particular module depends upon the number of resources (CLBs) used by that module. The allowable reconfiguration time for a column is different for different solutions (S).

6.9.1 Time constraints for RTR

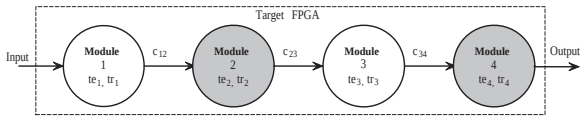
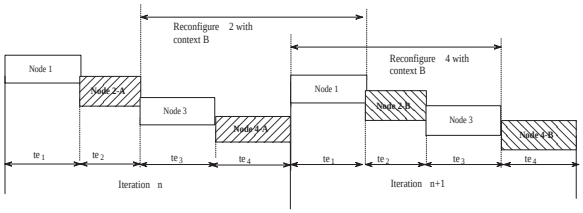


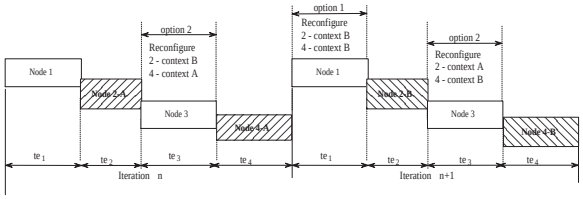
Figure 6.11: Simple 4 module data flow with two static modules and two modules requiring RTR.

Consider the simple data flow of Figure 6.11 having four modules. The operationality of shaded modules can change between iterations and hence they require RTR. Let $2A, 2B$ and $4A, 4B$ denote the two contexts of the reconfigurable modules 2 and 4. The execution time and required configuration time of module i is represented by te_i and tr_i respectively. For a reconfigurable module, te and tr are chosen to be maximum respective values over

its different contexts. The dynamic reconfiguration requirement states that if at instant n the incoming data is processed by $\{1,\ 2A,\ 3,\ 4A\}$ in that order, then the data at instant $n+1$ should be processed by $\{1,\ 2B,\ 3,\ 4B\}$ without having to suspend processing for reconfiguration. We illustrate the computation of the RTR time constraint based on two simple cases.



(a) Schedule for Case 1.



(b) Schedule for Case 2.

Figure 6.12: Timing diagram with possible reconfiguration schedules for the two partition cases.

• **Case 1:** In this case, both the reconfigurable modules are mapped to separate reconfigurable partitions (columns) on the FPGA. Figure 6.12(a) shows the timing diagram of successive iterations of this data flow. Since the two modules are mapped to independent reconfigurable partitions, they can be reconfigured at different times. To allow continuous operation without suspension, reconfiguration of a module must be done within the time that the other modules are functioning. These time intervals are shown on the timing diagram. From this diagram, the constraints that must be satisfied for RTR are

$$tr_2 < te_1 + te_3 + te_4 \tag{6.10}$$

$$tr_4 < te_1 + te_2 + te_3. \tag{6.11}$$

• **Case 2:** In the next case, both the reconfigurable modules are mapped to the same

partition (column) on the FPGA. Hence *both modules* are reconfigured together in one access. The time required for this reconfiguration is $tr_2 + tr_4$. The timing diagram for two iterations of the data flow in this case is shown in Figure 6.12(b). In this case there are two ways to schedule the required reconfiguration. The straightforward way is to reconfigure the partition with contexts $2B,\ 4B$ during execution of module 1 in iteration $n+1$. However, if $tr_2 + tr_4 > te_1$, this schedule is not feasible. Another way to schedule the reconfiguration is to access the partition once during execution of module 3 in iteration n to load contexts $2B,\ 4A$; and in iteration $n+1$ to load contexts $4B,\ 2A$ (it is assumed that processing from iteration $n+2$ proceeds with original contexts). Thus for this mapping there are two possible reconfiguration schedules. RTR can be done if any one of these schedules satisfies the reconfiguration time constraint. For this case, the constraint is specified as

$$tr_2 + tr_4 < max(te_1, te_3). \tag{6.12}$$

The same analysis is used to calculate the allowable reconfiguration times for different solutions of various data flows.

6.9.2 Partitioning Method

We use the simulated annealing approach with the cost function and constraints explained above. Any other heuristic based approach can also be used. The algorithm starts with an initial solution that places every module requiring RTR in to a *different* reconfigurable partition(column) on the FPGA. This minimizes the reconfiguration time per column. If this mapping does not satisfy the RTR time constraints, then no solution exists and the algorithm is terminated. Otherwise the cost of this mapping is calculated using (6.9) and is taken to be the current cost. At every iteration, a new solution is obtained by randomly selecting a module and moving it to some partition. If the chosen module requires reconfiguration, then only a move to one of the reconfigurable partitions is permitted. If the module is static, it can be moved to any of the reconfigurable partitions or remain in the static portion. Next, the reconfiguration and execution times are evaluated based on the data flow schedule to see if the solution satisfies the RTR constraint as explained. If this constraint is satisfied, the cost is calculated and the annealing proceeds in the traditional way. A relatively high

initial temperature is chosen to allow for exploration of the entire design space. The process continues till the temperature reduces to a user specified value. At the end of the process, we get the cost optimized partitioning solution. The solution gives us the required number of reconfigurable columns and the mapping of the modules to these columns and other static portions of the FPGA. If a static module is mapped to a reconfigurable column, it is loaded with the same context each time RTR is done.

6.10 Application to the Particle Filter

In this section, we demonstrate the application of the proposed technique to the realization of a simple particle filter used for 2-D tracking of maneuvering targets with RTR on a Xilinx FPGA. As seen in previous chapters, in the PF, each incoming data sample is processed by the so called steps of sampling, weight computation, resampling and output computation. Each of these steps requires M iterations, except resampling which requires $2M$ iterations, where M is the number of particles used. At the modular granularity, the PF can be split up into modules that are generic and those that are model dependent. The data flow at this granularity is shown in Figure 6.13. We use $M = 2000$ and a clock speed of $50MHz$ for the design. Processing of each incoming data sample thus takes $160\mu sec$ (critical path latency). The necessary characteristics of the processing modules are shown in Table 6.3. The bit widths of the communication lines between various modules are shown in Figure 6.13. For this design we have chosen the Xilinx Virtex II FPGA which has a configuration

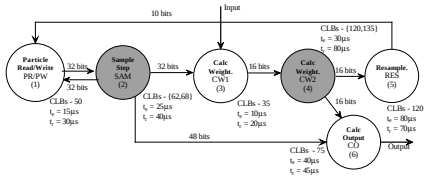


Figure 6.13: Data flow for a particle filter.

time per CLB of $0.6\mu sec$. The time for reconfiguration of each module is estimated based on its resource usage by implementing it in a seperate partition on the FPGA usign the partial reconfiguration flow outlined in [111]. For a reconfigurable module, the values tr and te are chosen to be the maximum values over all its contexts.

Module	Contexts	Critical Path	CLBs $\forall$ context	te μsec	tr μsec
PR/PW	1	yes	50	15	30
SAM	2	yes	$\{62, 68\}$	25	40
WC1	1	yes	35	10	20
WC2	2	yes	$\{120, 135\}$	30	80
RES	1	yes	120	80	70
OC	1	no	75	40	45

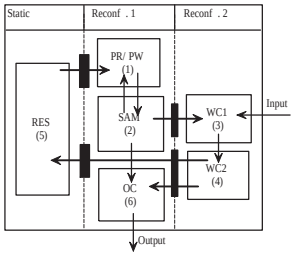
Table 6.3: Module Information Table

Figure 6.14 shows the partitioning result obtained by applying the method to the particle filter data flow. In the first case when $w = 0$ the algorithm attempts to minimize the overall communication cost. Communication between Modules 2 and 6 is wider than between Modules 2 and 3. Hence to minimize overall cost, Modules 2 and 6 are placed in the same partition. In the second case, the overall latency is a critical factor. Hence a high value of w is chosen. Priority is then given to minimizing communication on the cirtical path. This causes Modules 2 and 3 to be placed in the same partition and the RTR time constraint is satisfied by moving Module 6, despite its wider communication requirement, to a different partition since it does not lie on the critical path. The figure also shows the variation of the cost function during the annealing process for the two cases.

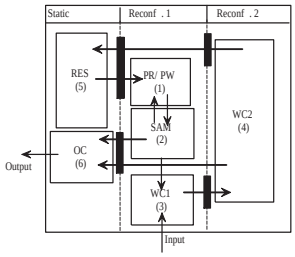
6.11 Conclusion

In this chapter, we described the development of a parallel architecture for efficient hardware implementation of a MM SIRF algorithm. Compared to traditional approaches, this algorithm does not require knowledge of transition probabilities and handles nonlinear and non-Gaussian models more efficiently. This architecture is based on a distributed resampling mechanism which greatly speeds up resampling and drastically reduces the data communication requirement. An efficient communication scheme was proposed which minimizes communication bottleneck and interconnect requirement. The architecture was used to implement and evaluate the MM SIRF for a practical tracking example on the Xilinx Virtex II platform.

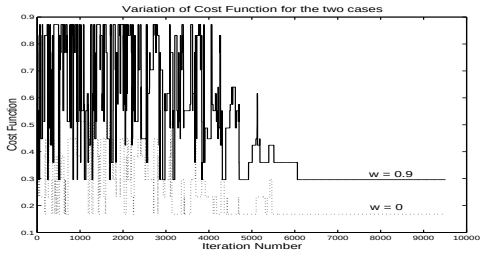
Many practical problems require VSMM type filtering involving multiple models that



(a) Result for $w = 0$.



(b) Result for $w = 0.9$.



(c) Variation of Cost Function

Figure 6.14: Partitioning results for particle filter for the two cases.

change with time. In order to extend the proposed MM SIRF architecture to this scenario, the hardware must support run-time reconfiguration. Recent commercial Xilinx FPGAs have started to support partial RTR. We developed a methodology for mapping the partical filter dataflow to a Xilinx FPGA such that the different models can be configured dynamically at run time. The methodology was tested using an elementary single-model realization of the SIRF. The same methodology can be used for mapping a VSMM particle filter to the Xilinx FPGA. This direction will be pursued as part of future work.

Chapter 7

Conclusion

Advances in VLSI technology have led to fast and efficient implementations of various signal processing algorithms enabling them to be applied to real time systems. However there are some methods whose complexity makes implementation on hardware platforms nontrivial. Moreover, although such algorithms show tremendous performance gains in theory, they are simple not formulated in a way suitable to hardware implementation. Hence reducing such algorithms to a feasible and efficient hardware implementation, is a challenging task. The Particle Filter is one such signal processing algorithm. Based on the Bayesian paradigm, it operates on Dynamic State Space models wherein the state of interest is hidden behind noisy observations. The particle filter is a Monte Carlo sampling method which operates by representing the posterior density by a set of weighted samples. Recursive propagation of this posterior representation makes these filters suitable for on-line sequential processing. Particle filters handle nonlinearity and non-Gaussianity in the state space model much more efficiently than traditional methods and hence they show significantly improved performance over such methods

As these filters have matured over the years, several important practical problems have been identified where the superiority of the particle filters makes them a desirable solution. However, due to lack of hardware capable of real time processing, they have not experienced widespread adoption. This dissertation has taken a significant step in enabling this by developing efficient hardware for real-time particle filtering. Along with implementation of the standard particle filter, reconfigurable architectures for these filters were explored with a

view to exploit their inherent flexibility. Reconfigurability includes the multiple variants of the particle filter being executed on the same hardware and also adapting of particle filtering parameters dynamically during execution.

7.1 Summary of Contributions

The main goal of this dissertation was to develop hardware capable of real time particle filtering. The specific contributions towards this effort can be summarized as follows.

- Algorithmic Analysis and Modification

  Particle filters have evolved over the years with most of the research focused on their theory. As a result, the traditional particle filtering algorithm is not suited to hardware implementation. We have performed an in-depth analysis of this algorithm and quantified its major bottlenecks and resource hungry operations. We have introduced several modifications to this algorithm with an analysis of how these modifications impact the hardware implementation. These modified algorithms were examined from a theoretical perspective and it was shown that the modifications do not affect the integrity of the particle filter.

- Architectures and Memory Schemes for SIRFs

  The Sampling Importance Resampling Filter (SIRF) is the most commonly used version of the particle filtering technique. One of the most important aspects of the implementation of SIRFs is that of particle storage and access before and after the sequential resampling process. We have introduced architectures that reduce the memory requirement and access time for this operation. The proposed schemes are highly scalable such that the execution time can be further reduced by addition of hardware resources. With these architectures, we pave the way for a generic particle filtering hardware framework that can be used to implement the SIRF for any problem with minimal redesign.

- Distributed Architecture for Multiple Model Particle Filters

Multiple model filters are used when a single model does not completely describe the probabilistic behaviour of the state. For instance, in several tracking problems, the target is maneuvering such that a single model does not cover all the target dynamics. Particle Filters are an ideal solution for cases when the models are nonlinear and non-Gaussian. Other than the fact that they are not severely affected by these properties, the particle filters provide an excellent means of model interaction and information exchange due to the point-wise representation of the posterior and the process of re-sampling. When implemented on hardware, a distributed architecture based on that for the standard particle filter can be used. However, this gives rise to a whole new set of problems regarding filter scalability and data exchange protocols for minimizing interconnect and communication latency.We have addressed these problems and introduced a pipelined particle distribution protocol that minimizes bus requirement and latency.

- Reconfigurable hardware for processing two different particle filters

  One limitation of the SIRF hardware is that the number of particles that can be used is limited by the memory available on the device. In some situations it may be required to increase the number of particles beyond this limit to ensure performance. For such cases, we have explored the implementation of another particle filter known as the Gaussian Particle Filter. The GPF is inferior in performance to the standard SIRF and is also less general. However, we have introduced an algorithmic modification for this filter that allows its execution without storing of particles in memories between recursions. This allows the GPF hardware to use any number of particles irrespective of the memory on the device. With these factors in mind, we have developed a reconfigurable architecture that incorporates the GPF and the SIRF. It is intended for applications where the SIRF is normally used, except when the particle requirement exceeds the available memory. In such cases, one can dynamically switch to GPF execution by specifying a small set of parameters.

- Dynamic reconfiguration of particle filter model

  The architecture introduced for the particle filter consists of some generic blocks and some model dependent blocks. In some cases, like the variable structure multiple model

particle filter, while the block level dataflow remains the same, the model used for the filtering may change dynamically. We have utilized the recently introduced partial run-time reconfiguration technique for commercial FPGAs to develop a design methodology that allows such a reconfigurable particle filter to be mapped on to the FPGA. The methodology minimizes the reconfiguration overhead while maintaining the execution flow of the particle filter undisturbed. The methodology has been tested with an elementary particle filter example, but can be extended to any practical scenario.

7.2 Future Direction

This effort lays the foundation for several research directions in the future. Some of these are enlisted here.

- Particle Filtering Cores for FPGAs

  As we have seen, the focus of the initial part of the dissertation was on developing *generic* architectures for particle filters. Particularly, we saw that using the proposed architecture, an SIRF can be realized for any model. The only design effort needed is for building the model dependent data driven computation blocks. An interesting research direction is looking for automated design methodologies that would read model parameters, synthesize the model dependent computation steps from standard libraries, integrate the synthesized blocks into the generic architecture framework, and present a combined netlist for implementation on an FPGA. This would help in widespread adoption of the particle filter by making the design process very convenient for engineers.

- General Location and Tracking Engine using PFs

  Particle Filters are gaining increasing popularity in indoor Real Time Location systems (RTLS) and robot navigation. When the particle filters applied to a variety of problems in these fields are examined, they differ in some minor aspects, but a general consistent framework is often obvious. The reconfigurable particle filter architectures presented in this dissertation, can exploit this fact to develop a generic location and tracking engine that can be configured on a case-by-case basis.

- Adaptive and Variable Structure Particle Filters

  In this dissertation, we have laid the foundation for adaptive particle filtering where the model parameters are changed dynamically between iterations. We also explored variable structure multiple model filters where model sets change dynamically. This adaptability makes particle filters very powerful and enable them to be applied to even the most complex of systems. We have demonstrated a methodology to realize such filters on commercial FPGAs with partial run time reconfiguration capabilities. This methodology was only evaluated on a *proof-of-concept* basis for an elementary particle filtering model. Extension of this to practical problems and development of algorithms to trigger the adaptation or change in structure is a promising direction for the future.

Bibliography

[1] H. Akashi and H. Kumamoto, "Construction of discrete time non linear filter by Monte Carlo methods with variance reduction techniques," *Systems and Control*, vol. 19, pp. 211–221, 1975.

[2] J. Alspector, J. Gannett, M. Parker, M. Parker, and R. Chu, "A VLSI efficient technique for generating multiple uncorrelated noise sources and its application to stochastic neural networks," *IEEE Transactions on Circuits and Systems*, vol. 38, pp. 109–123, Jan 1991.

[3] B. D. O. Anderson and J. B. Moore, *Optimal Filtering*. Prentice Hall, 1979.

[4] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filtering for online nonlinear/non-Gaussian Bayesian tracking," *IEEE Transactions on Signal Processing*, vol. 50, no. 2, pp. 174–188, February 2002.

[5] A. Athalye, M. Bolić, S. Hong, and P. M. Djuric, "Architectures and memory schemes for sampling and resampling in particle filters," in *IEEE DSP Workshop*, New Mexico, USA, Aug 2004.

[6] A. Athalye, M. Bolić, S. Hong, and P. M. Djurić, "Generic hardware architectures for sampling and resampling in particle filters," *Eurasip Journal on Applied Signal Processing*, no. 17, pp. 2888–2902, Sept. 2005.

[7] A. Athalye and S. Hong, "Technique for mapping partial reconfigurable dataflows to xilinx FPGAs," in *IEEE SOCC*, Washingtion, DC, USA, 2005.

[8] A. Athalye, S. Hong, and P. M. Djurić, "Effect of VLSI noise generators on performance and complexity of real time particle filters," in *Conferernce on Information Science and Systems*, Maryland USA, 2003.

[9] ——, "Distributed architecture and interconnection scheme for multiple model particle filters," in *ICASSP*, Tolouse, France, March 2006.

[10] A. Athalye, X. Liang, and S. Hong, "Dynamic coarse grain dataflow reconfiguration technique for real time system design," in *ISCAS*, Kobe, Japan, 2005.

[11] S. Bakshi and D. D. Gajski, "A component selection algorithm for high performance pipelines," University of California Irvine, Tech. Rep. 94-01b, 1994.

[12] ——, "A memory selection for high performance pipelines," University of California Irvine, Tech. Rep. 95-03, 1994.

[13] S. Bakshi, H. P. Juan, and D. D. Gajski, "Architectural exploration," Dept of Computer Science University of California Irvine, Tech. Rep. 93-10, March 1993.

[14] E. R. Beadle and P. M. Djurić, "A fast weighted Bayesian bootstrap filter for nonlinear model state estimation," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 33, pp. 338–343, 1997.

[15] S. Blackman and R. Popoli, *Design and Analysis of Modern Tracking System.* Norwood, MA: Artech House, 1999.

[16] Y. Boers, "On the number of samples to be drawn in particle filtering," *IEE Colloquium on Target Tracking: Algorithms and Applications*, pp. 5/1–5/6, Nov 1999.

[17] M. Bolić, "Architectures for the efficient implementation of particle filters," Ph.D. dissertation, Stony Brook University, 2004.

[18] M. Bolić, A. Athalye, P. Djurić, and S. Hong, "Algorithmic modification of particle filters for hardware implementation," in *EUSIPCO*, 2004.

[19] M. Bolić, A. Athalye, S. Hong, and P. M. Djurić, "Gaussian particle filters: Hardware implementation perspective," *VLSI Signal Processing*, 2007.

[20] E. Boutillon, J. L. Danger, and A. Ghazel, "Design of high speed AWGN communication channel emulator," *Analog Integrated Circuits and Signal Processing*, vol. 34, no. 2, pp. 133–142, Feb 2003.

[21] J. Carpenter, P. Clifford, and P. Fernhead, "Improved particle filter for non linear problems," *IEEE proceedings on radar and sonar navigation*, vol. 146, no. 1, pp. 2–7, 1999.

[22] R. Cmar, L. Rijnders, P. Schaumount, and I. Bolsens, "A methodology and design environment for DSP ASIC fixed point refinement," in *Confernce on Design Automation and Testing in Europe*, Munich, Germany, 1999, pp. 271–277.

[23] *Understanding Synchronous and Asynchronous Dual Port RAMs*, Cypress Semiconductor Corporation, July 2001, Application Note, Available from "www.cypress.com".

[24] J. Dalcolmo, R. Lauwereins, and M. Ade, "Code generation of data dominated dsp applications for fpga targets," in *Proceedings of the IEEE IWRSP*, 1998, pp. 162–167.

[25] J. L. Danger, A. Ghazel, E. Butillon, and H. Laamari, "Efficient FPGA implementation of gaussian noise generator for communication channel emulation," in *IEEE ICECS*, Laslik, Lebanon, 2000, pp. 572–578.

[26] F. Daum and J. Huang, "Curse of dimensionality and particle filters," in *Fifth ONR/GTRI Workshop on Target Tracking and Sensor Fusion*, Newport, RI, June 2002.

[27] P. M. Djurić, J. Kotecha, J. Zhang, Y. Huanga, T. Ghirmai, M. Bugallo, and J. Míguez, "Particle Filtering," *IEEE Signal Processing Magazine*, vol. 20, no. 5, pp. 19–38, 2003.

[28] R. Douc, O. Cappe, and E.Moulines, "Comparison of resampling schemes for particle filtering," in *Proceedings of the 4th ISPA*, 2005.

[29] A. Doucet, N. de Freitas, and N. Gordon, Eds., *Sequential Monte Carlo Methods in Practice*. Springer, 2001.

[30] A. Doucet, S. J. Godsill, and C. Andrieu, "On sequential Monte Carlo methods for Bayesian filtering," *Statics and Computing*, pp. 197–208, 2000.

[31] M. Dyer, C. Plessl, and M. Platzner, "Partially reconfigurable cores for Xilinx Virtex," in *Intl. Conf. on Field Programmable Logic (FPL)*, Sept. 2002.

[32] C. B. et al., "Designing partial and dynamically reconfigurable applications on Xilinx Virtex-II FPGAs using Handel-C," University of Erlangen-Nuremberg, Tech. Rep., 2004.

[33] C. Fang and R. A. Rutembar, "Toward efficient static analyis of finite-precision effects in dsp applications via affine arithmetic modeling," in *40th Design Automation Conference*, 2003, pp. 496–501.

[34] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, 1972.

[35] J. Geweke, "Bayesian inference in econometric models using Monte Carlo integration," *Econometrica*, vol. 57, no. 6, pp. 1317–1339, 1989.

[36] N. S. Ghazal, "Evaluation and guidance in processor architecture selection for dsp," Ph.D. dissertation, University of California at Berkeley, 2000.

[37] S. Godsill and T. Clapp, *Sequential Monte Carlo methods in practice*. Springer, 2001, ch. Improvement Strategies for Monte Carlo particle filters, pp. 139–158.

[38] N. J. Gordon, D. Salmond, and A. F. M. Smith, "Novel approach to non linear/non Gaussian Bayesian state estimation," *IEEE Proceedings-F*, vol. 140, no. 2, pp. 107–113, April 1993.

[39] L. M. Guerra, "Behavioral level guidance using property based design charecterization," Ph.D. dissertation, University of Calfornia, Berkeley, 1996.

[40] W. Guo, "Dynamic state space models," *Journal of Time Series Analysis*, vol. 24, no. 2, pp. 14–158, 2003.

[41] J. E. Handschin and D. Q. Mayne, "Monte carlo techniques to estimate conditional expectation in multi statge non linear filtering," *International Journal of Control*, vol. 9, pp. 547–559, 1969.

[42] P. J. Harrison and C. F. Stevens, "Bayesian forecasting," *Journal of Statistical Society*, vol. B, no. 38, pp. 205–247, 1976.

[43] A. C. Harvey, *Forecasting, Structural Time Series Models and Kalman Filter*. Cambridge University Press, 1989.

[44] M. Hayes, *Statistical Digital Signal Processing and Modeling*. Wiley, 1996.

[45] J. Henessy and D. Patterson, *Computer Architecture A Quantative Approach*, 3rd ed. Sab Francisco USA: Morgan Kaufman, 2003.

[46] S. Hong, A. Athalye, and P. M. Djurić, "Design methodology for reconfigurable particle filter realizations," *IEEE Transactions on Circuits and Systems*, 2007.

[47] J. H. Hotecha, "Sequential monte carlo methods for dss models with applications to communications," Ph.D. dissertation, Stony Brook University, New York, 2001.

[48] Y. Huang and P. M. Djurić, "A new importance function for particle filtering and itsapplication to blind detection in flat fading channels," in *International Conference Acoustics, Speech and Signal Processing (ICASSP)*, vol. 2, 2002, pp. 1617–1620.

[49] ——, "A hybrid importance function for particle filtering," *IEEE Signal Processing Letters*, vol. 11, no. 3, pp. 404–406, March 2004.

[50] M. Hübner, T. Becker, and J. Becker, "Real time LUT based network topologies for dynamic and partial self reconfiguration." in *SBCCI*, Brazil, Sept. 2004.

[51] A. H. Jazwinski, *Stochastic Processes and Filtering Theory*. New York: Academic Press, 1970.

[52] S. Julier and J. Uhlmann, "A new extension of the Kalman filter to nonlinear systems," in *Int. Symp. Aerospace/Defense Sensing, Simul. and Controls*, 1997.

[53] H. Jung, K. Lee, and S. Ha, "Efficient hardware controller synthesis for synchronous dataflow graph in system level design," in *ISSS*, 2000, pp. 79–84.

[54] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Journal of Basic Engineering Transactions, ASME*, vol. Ser.D, no. 83, pp. 95–108, 1960.

[55] K. Kanazawa, D. Koller, and S. J. Russel, "Stochastic simulation algorithm for dynamic probabalistic networks," in *Eleventh Annual Conference on Uncertainity AI*, 1995, pp. 346–351.

[56] M. Kaul, V. Srinivasan, S. Govindrajan, I. Ouaiss, and R. Vemuri, "Partitioning and synthesis for run-time reconfigurable systems using the SPARCS system." University of Cincinnati., Tech. Rep., 1999.

[57] S. M. Kay, *Fundamentals of Statistical Signal Processing Estimation Theory.* Prentice Hall, 1993.

[58] S. Kim, K. Kum, and W. Sung, "Fixed point optimization utility for C and C++ based digital signal processing," *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing*, vol. 45, no. 11, pp. 1455–1464, Nov 1998.

[59] S. Kim and W. Sung, "A floating point to fixed point assembly translator for the TMS320C25," *IEE Transactions on Circuits and Systems - II*, vol. 41, pp. 730–739, Nov 1994.

[60] T. Kirubarajan and Y. Bar-Shalom, "Tracking evasive move-stop-move targets with an MTI radar using a VS-IMM estimator," *SPIE Signal and Data Processing of small targets*, 2000.

[61] G. Kitagawa, "Monte Carlo filter and smoother for non-gaussian nonlinear state space models," *Journal of Computational and Graphical Statistics*, vol. 5, pp. 1–25, 1996.

[62] D. E. Knuth, *The Art of Computer Programming.* Adisson Welsley, 1998.

[63] A. Kong, J. Liu, and W. Wong, "Sequential imputations and Bayesian missing data problems," *Journal of American Statistical Association*, vol. 89, no. 425, pp. 278–288, 1994.

[64] J. H. Kotecha and P. M. Djurić, "Gaussian particle filtering," in *Workshop on Statistical Signal Processing*, Singapore, August 2001.

[65] M. Kumar, "Measuring parallelism in computation-intensive scientific/engineering applications," *IEEE Transactions on Computers*, vol. 37, no. 9, pp. 1088–1098, 1988.

[66] P. Landman, R. Mehra, and J. Rabaey, "An integrated CAD environment for low power design," *IEEE Design and Test of Computers*, vol. 13, no. 2, pp. 72–82, 1996.

[67] R. Lauwereins, M. Engels, M. Ade, and J. A. Peperstraete, "Grape-II: A system level prototyping environment for dsp applications," *IEEE Computers*, pp. 35–43, 1995.

[68] J. Lee and H. Ko, "Effective tracking for maneuvering mobile station via IMM filter in CDMA environment," *IEICE Transactions on Communications*, vol. E86-B, no. 11, pp. 3336–3339, November 2003.

[69] X. Liang, A. Athalye, and S. Hong, "Equalizing datapath for processing speed determination in block level pipelining," in *ISCAS*, Kobe, Japan, 2005.

[70] J. S. Liu and R. Chen, "Blind deconvolution via sequential imputations," *Journal of the American Statistical Association*, vol. 90, pp. 567–576, 1995.

[71] ——, "Sequential monte carlo methods for dynamic systems," *Journal of American Statistical Association*, vol. 93, pp. 1032–1044, 1998.

[72] J. S. Liu, R. Chen, and T. Logvinenko, *Sequential Monte Carlo Methods in Practice*. Springer Verlag, 2001, ch. A Theoreticlal Framework for Sequential Importance Sampling with Resampling, pp. 225–242.

[73] J. MacCormick and A. Blake, "A probabalistic exclusion principle for tracking multiple objects," in *Intl. Conference on Computer Vision*, 1999, p. 572578.

[74] G. Marsaglia, *A Current View of Random Number Generators*. Elviser Press, 1984.

[75] ——, *Die Hard battery of tests*, available from http://stat.fsu.edu/ geo/diehard.html.

[76] F. Martinerie and P. Forster, "Data association and tracking using hidden markov models and dynamic programming," in *ICASSP*, 1992.

[77] E. Mazor, A. Averbuch, Y. Bar-Shalom, and J. Dyan, "Interacting multiple model methods in target tracking: A survey," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 34, no. 1, pp. 103–123, January 1998.

[78] S. McGinnity and G. Irwin, "Multiple model bootstrap filter for maneuvering target tracking," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 36, no. 3, pp. 1006–1011, July 2000.

[79] B. Miramond and J. Delosme, "Design space exploration for dynamically reconfigurable architectures," in *Design Automation and Test in Europe(DATE)*, 2005.

[80] P. D. Moral, "Non linear filtering: Interacting particle solutions," *Markov Processes and Related Fields*, vol. 2, no. 4, pp. 555–580, 1998.

[81] S. Nardone, A. Lindgren, and K. Gong, "Fundamental properties and performance of bearings-only target motion analysis," *IEEE Transactions on Automatic Control.*, vol. 29, no. 9, pp. 775–787, 1984.

[82] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee, "Precisoin and error analysis of MATLAB applications during automated hardware synthesis for FPGAs," in *Confernce on Design Automation and Testing*, Munich, Germany, 2001, pp. 722–728.

[83] A. Papoulis, *Probability, Random Variables and Stochastic Processes.* McGraw-Hill International Editions, 1991.

[84] K. K. Parhi, *VSLI digital signal processing systems: design and implementation.* John Wiley & Sons, 1999.

[85] J. Proakis and M. Salehi, *Contemporary Communication Systems.* PWS Publishing Co, 1999.

[86] J. Rabaey, *Digital Integrated Circuits: A Design Perspective.* Prentice Hall, 1996.

[87] J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, "Fast prototyping of data path intensive architectures," *IEEE Design and Test of Computers*, pp. 40–51, 1991.

[88] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech processing," *IEEE Proceedings*, vol. 77, no. 2, pp. 257–285, February 1989.

[89] L. R. Rabiner and B. H. Juang, "An introduction to hidden markov models," *IEEE Acoustic, Speech, Signal Processing Magazine*, pp. 4–16, January 1986.

[90] A. Raghunathan, N. Jha, and S. Dey, *High Level Power Analysis and Optimization*. Kluwer Academic Publications, 1998.

[91] C. P. Robert and G. Casella, *Monte Carlo Statistical Methods*. Springer, 1999.

[92] D. B. Rubin, "The SIR algorithm," *Journal of the American Statistical Association*, 1987.

[93] H. Schildt, *The Complete C++ Reference*, 4th ed. McGraw Hill, 2003.

[94] B. Shackleford, M. Tanaka, R. Carter, and G. Snider, "Fpga implementation of neighbourhood of four cellular automata random number generators," in *FPGA'02*, Monterey, CA, Feb 2002.

[95] C. Shi, "Floating-point to fixed-point conversion," Ph.D. dissertation, University of California, Berkeley, 2004.

[96] S. G. Shiva, *Pipelined and Parallel Computer Architectures*. Harper Collins College Publisher, 1996.

[97] B. Stroustrup, *The C++ Programming Language*, 3rd ed. Addison Welsley, 1998.

[98] *SystemC User Guide*, Synopsys Inc, version 2.0.1.

[99] H. Tanizaki, *Non Linear Filters: Estimation and Applications*. New York: Springer Verlag, 1993, vol. 400.

[100] R. Tessier and W. Burleson, "Reconfigurable computing for digital signal processing: A survey," *Journal of VLSI Signal Processing*, vol. 28, pp. 7–27, 2001.

[101] *TMS320C54x DSP Library Programmers Reference*, Texas Instruments, August 2002.

[102] J. Throvinger, "Dynamic partial reconfiguration of an FPGA for computational hardware support," Master's thesis, Lund University of Technology, 2004.

[103] P. Tichavský, C. Murvachik, and A. Nehorai, "Posterior Cramér Rao lower bound for descrete-time nonlinear filtering," *IEEE Transactions on Signal Processing*, vol. 46, no. 5, pp. 1386–1396, May 1998.

[104] S. J. Upadhyay, "Noise generators," *Encyclopedia of Electrical and Electronic Engineering*, Dec 1998.

[105] A. H. Ween, "Dataflow machine architecture," *ACM Computing Surveys*, vol. 18, no. 4, pp. 365–396, 1986.

[106] M. Willerns, V. Burgens, H. Kendig, T. Grotker, and H. Meyr, "System level fixed point design based on an interpolative approach," in *34th Design Automation Conference*, Anaheim, CA, 1997.

[107] J. Williams and N. Bergmann, "Embedded linux as a platform for dynamically self reconfiguring systems-on-chip," in *International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2004.

[108] S. Wolfram, *Theory and Applications of Cellular Automata*. World Scientific Publishing Co, 1986.

[109] *Xilinx Core Generator*, Xilinx Inc, 2000.

[110] *Xilinx Pipelined Divider Core Specification*, v2.0 ed., Xilinx Inc., June 2000.

[111] *Virtex II pro platform FPGAs: Functional Description*, Xilinx Inc., San Jose, CA, 2003.

[112] *Two Flows for Partial Reconfiguration of Xilinx FPGAs*, Xapp290 v1.2 ed., Xilinx Inc., Sept. 2004.

[113] *Xilinx Cordic Core Specification*, v3.0 ed., Xilinx Inc., April 2005.

[114] N. Zhang, "Algorithm/architecture co-design for wireless communications systems," Ph.D. dissertation, Unibersity of California, Berkeley, 2001.