Architectures for Efficient Implementation of Particle Filters

A Dissertation Presented

by

Miodrag Bolić

to

The Graduate School

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in

Electrical Engineering

Stony Brook University

August 2004

Stony Brook University

The Graduate School

Miodrag Bolić

We, the dissertation committee for the above candidate for the Ph. D. degree, hereby recommend acceptance of this dissertation

---

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

John Murray, Chairperson of Defense
Associate Professor, Department of Electrical & Computer Engineering

---

Sangjin Hong, Assistant Professor
Department of Electrical & Computer Engineering

Kenny Qian Ye, Assistant Professor
Department of Applied Mathematics & Statistics

This dissertation is accepted by the Graduate School

Graduate School

Abstract of the Dissertation

Architectures for Efficient Implementation of Particle Filters

by

Miodrag Bolić

Doctor of Philosophy

in

Electrical Engineering

Stony Brook University

2004

Particle filters are sequential Monte Carlo methods that are used in numerous problems where time-varying signals must be presented in real time and where the objective is to estimate various unknowns of the signal and/or detect events described by the signals. The standard solutions of such problems in many applications are based on the Kalman filters or extended Kalman filters. In situations when the problems are nonlinear or the noise that distorts the signals is non-Gaussian, the Kalman filters provide a solution that may be far from optimal. Particle filters are an intriguing alternative to the Kalman filters due to their excellent performance in very difficult problems including communications, signal processing, navigation, and computer vision. Hence, particle filters have been the focus of wide research recently and immense literature can be found on their theory. Most of these works recognize the complexity and computational intensity of these filters, but there has been no effort directed toward the implementation of these filters in hardware. The objective of this dissertation is to develop, design, and build efficient hardware for particle filters, and thereby bring them closer to practical applications. The fact that particle filters outperform most of the traditional filtering methods in many complex practical scenarios, coupled with the challenges related to decreasing their computational complexity and improving real-time performance, makes this work worthwhile.

The main goals of this dissertation are to develop and modify particle filter algorithms and to develop physically feasible hardware architectures that allow for improving the processing speed of particle filters. The issues tackled include reduction of computational complexity, improving scalability of parallel implementation and reducing memory requirements. This work has resulted in the development of the first hardware prototype of a particle filter. The speed improvement in comparison with the implementation on the state-of-the-art digital signal processors is fifty times.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor, Professor Petar M. Djurić, for supporting me over these years, and for giving me a lot of freedom as well as excellent guidance in exploring different aspects of the problem. I would like also to thank my co-advisor Professor Sangjin Hong for his motivation and guidance in issues related to VLSI design. I also thank Professors John Murray and Kenny Ye for taking the time to review my dissertation.

I would like to thank my friends and colleagues at Stony Brook University with whom I have had the pleasure of working over the years. These include Akshay D. Athalye, with whom I worked closely on the project, Monica Fernandez-Bugallo, Katrien De Cock, Yufei Huang, Tadesse Ghirmai, Jae-Chan Lim, Mahesh Vemula, and all other members of the Cosine lab.

I would like to thank Stephen Shellhammer for hiring me as an intern at Symbol Technologies Inc. in 2001 where I have been working for three years. I would also like to thank all the members of Bluetooth and RFID teams at Symbol Technologies Inc. including Gary Schneider, Raj Bridgelall, Mike Knox, Adam Levine, Frank Boccuzzi, and Sean Connolly.

Financial support for this research came from the NSF under Award CCR-0220011 and it is gratefully acknowledged.

Chapter 1

Introduction

1.1 Motivation

The field of digital signal processing has always been driven by the advances in the signal processing algorithms and in very-large-scale-integrated (VLSI) technologies. At any given time, DSP applications impose numerous challenges on the implementation of DSP systems. Particle filters represent one of existing and very important challenges for implementation. There are many applications nowadays where particle filters can make considerable improvements in performance, but often they have not been used because they cannot meet the stringent requirements of real-time processing. These applications include wireless communications, robotics, navigation and tracking systems, where sequential (adaptive) signal processing is needed. A common problem in all of these applications is the requirement that dynamic signal parameters or states are estimated and/or detected in real-time. For example, in bearings-only tracking, it is critical to estimate the position and the velocity of an object in a two-dimensional plane based on noisy measurements of angular positions of the object. Or, in wireless communications, as signals are received, a joint operation of estimation and detection is used to estimate the time varying channel and to detect the transmitted symbols. In these applications, particle filters outperform traditional methods. They are, however, very computationally intensive, and this is their main drawback. Almost all of the literature on particle filtering is on its theory. To the best of our knowledge, there have been no serious efforts for designing hardware for particle filters. With this dissertation, our intention is to fill in this gap.

The main design objective is to develop hardware architectures that support high speed particle filtering. Often, in practical applications, a large number of particles needs to be used for computing the estimates of desired states. As the number of particles increases, the speed of the particle filter is seriously affected. The algorithms presented here have been applied to the bearings-only tracking problem. Using 1000 particles on a single state-of-the-art digital signal processor (DSP), we have obtained speeds of up to $1kHz$. Clearly, this speed would seriously limit the range of applications for which the particle filter can be used for real time processing. Hence for meeting speed requirements of real-time applications, it is necessary to have high throughput designs with ability to process a large number of particles in a given time.

1.2 Challenges and contributions

Particle filters perform three basic operations: generation of new particles (sampling from the space of unobserved states), computation of particle weights (probability masses associated with the particles) and resampling (a process of removing particles with small weights and replacing them with particles with large weights). These three steps make the basis of the most commonly used type of particle filters called Sample-Importance-Resampling Filters (SIRF). Particle generation and weight computation are computationally the most intensive steps.

The particle filtering speed is increased through both algorithmic modifications and architecture development. The main challenges for speed increase on the algorithmic level include reducing the number of operations and exploiting operational concurrency between the particle generation and weight computation steps. Resampling is not computationally intensive, but to increase speed we need modifications of the algorithm to allow for overlapping its operations with particle generation.

High speed requirements impose one-to-one mapping of particle filtering operations to the hardware blocks so that these blocks can perform their operations concurrently. Hardware platforms that support concurrent processing are FPGA and ASIC. We have chosen the FPGA platform, which is a standard choice during the prototyping phase because of its flexibility. Challenges on the architectural levels include choice of implementation of complex and non-linear operators, dealing with possible multidimensional structures in hardware, and choosing the proper level of pipelining for each hardware block. As a result, the maximum achievable speed is proportional to $1/(2MT_{clk})$, where M is the number of particles and T_{clk} is the clock period.

To increase speed further, we target a parallel implementation with multiple processing elements (PEs), where each PE processes a fraction of the total number of particles, and a single central unit (CU), which controls the operations of the PEs as shown in Figure 1.1. Specifically, the generation of new particles and the computation of their weights are processed in parallel by four PEs. The CU performs resampling and coordinates other operations including synchronization and estimation of the required unknowns. An interesting feature of the particle filters is that, at a particular time instant, particle generation and weight computation operations for different particles are independent which allows for concurrent processing of the particles in different PEs. However, parallelizability of the filter is affected by the resampling step. Resampling has the following disadvantages from a parallel hardware implementation viewpoint: the sampling period and memory requirements are increased, the data exchange in implementations with multiple PEs becomes a major bottleneck of the parallel design, and the complexity of the CU is greatly increased. The effects of resampling on performance and complexity are reduced by using modified resampling algorithms and architectures that allow for local and deterministic data exchange patterns.

In parallel hardware implementations, the resampling becomes a bottleneck due the necessity for exchanging a large number of particles through the interconnection network. A type of particle filters called Gaussian Particle Filter (GPF) does not require resampling. We modified the GPF algorithm in a way that the processing speed is twice higher than the speed of the SIRF algorithm, which is the main advantage of the GPF. The data exchange in GPFs, though not negligible, is significantly lower than in SIRFs and is deterministic. This makes the

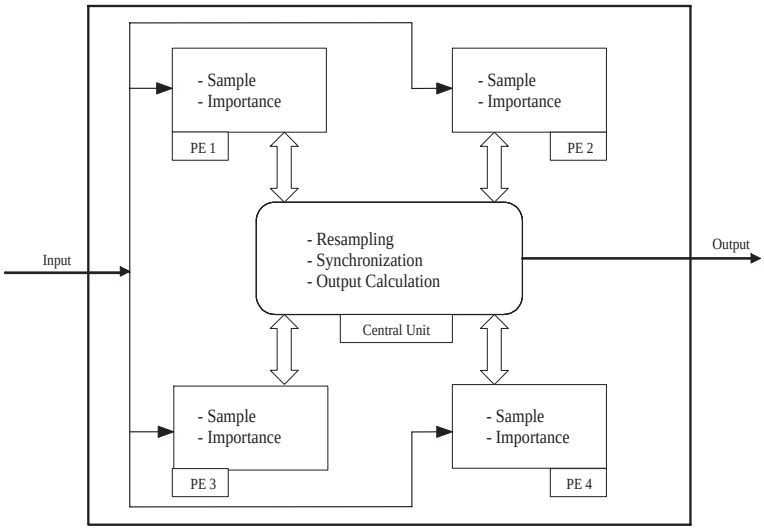


Figure 1.1: Block diagram of a parallel particle filter.

GPF a better candidate for parallel implementation. Another very attractive feature of the GPF is that, as opposed to SIRF, it can be implemented without storing particles between successive time instants. This eliminates the need for memories in hardware and provides freedom for using a large number of particles for processing without being constrained by the size of memory.

1.3 Organization of the dissertation

This document is organized in seven chapters. In this chapter, the motivation, contributions and results are described.

In Chapter 2, a short background on theory of particle filters is presented. Throughout the dissertation, performances of particle filters are estimated and the particle filter prototype is built for the bearings-only tracking applications. Bearings-only tracking model is introduced and pseudocode of the SIRF and GPF for bearings-only tracking is presented.

Basic terminology of VLSI signal processing architectures is presented in Chapter 3. The particle filter algorithm is analyzed from the standpoint of concurrency of operations and parallelism. Temporally concurrent architectures for both non-parallel and parallel particle filters are proposed.

New resampling algorithms that are more suitable for hardware implementation are proposed in Chapter 4. The new algorithms reduce the complexity of both hardware and DSP realization through addressing common issues such as decreasing the number of operations, number of memories and memory access. Moreover, the algorithms allow for higher processing speed by overlapping in time the resampling step with the other particle filtering steps. Since resampling is not dependent on any particular application, the analysis is appropriate for all types of particle filters that use resampling. One way of dealing with fixed point is-

sues in the residual resampling algorithm and the corresponding architecture are presented. Also, hardware architectures for SIRFs are presented together with the results of the FPGA implementation of the SIRF for the bearings-only tracking problem.

In Chapter 5, we propose resampling algorithms with architectures for parallel implementation of SIRFs. The proposed algorithms improve the scalability of the filter architectures affected by the resampling process. One of the main advantages of the new resampling algorithms is that communication through the interconnection network is reduced and made deterministic, which results in simpler network structure and increased sampling frequency. In the architectural part of the analysis, the area and speed of the SIRF implementation are estimated for different number of particles and different level of parallelism with FPGA implementation.

In Chapter 6, we analyze the algorithmic and architectural characteristics of GPFs. The GPF algorithm is modified in a way that there is no need for storing particles in memories between successive recursions. We have analyzed the GPF on the bearings-only tracking problem and the results are compared with the results of SIRF in terms of computational complexity, potential throughput, and hardware energy.

The current state of our work and research directions are summarized in Chapter 7.

1.4 Published work

This dissertation is based on the following previously published material:

- "Finite Precision Effect on Performance and Complexity of Particle Filters for Bearing-Only Tracking," M. Bolić, S. Hong, and P. M. Djurić, IEEE Asilomar Conference on Signals, Systems and Computers, 2002. Section 4.3.3
- "An Efficient Fixed-Point Implementation of Residual Systematic Resampling Scheme for High-Speed Particle Filters," S. Hong, M. Bolić, and P. M. Djurić, *IEEE Signal Processing Letters*, 2003. Section 4.2
- "New Resampling Algorithms for Particle Filters," M. Bolić, P. M. Djurić, S. Hong, ICASSP, 2003. Section 4.1.1
- "Resampling algorithms for particle filters suitable for parallel VLSI implementation," M. Bolić, P. M. Djurić, S. Hong, CISS, 2003. Section 5.2
- "Resampling Algorithms for Particle Filters: A Computational Complexity Perspective," M. Bolić, P. M. Djurić, and S. Hong, *EURASIP Journal of Applied Signal Processing*, 2004. Section 4.1
- "Algorithmic Modification of Particle Filters for Hardware Implementation," M. Bolić, A. Athalye, P. Djurić and S. Hong, EUSIPCO, 2004. Chapter 6.1
- "Design Complexity Comparison Method for Loop-Based Signal Processing Algorithms: Particle Filters," S. Hong, M. Bolić, P. M. Djurić, ISCAS, 2004. Section 7.2

- "Architectures and Memory Schemes for Sampling and Resampling in Particle Filters," A. Athalye, M. Bolić, S. Hong and P. Djurić, DSP/SPE Workshop, 2004. Section 4.3

The dissertation contains work presented in the following submitted papers:

- "A Design Study for Practical Physical Implementation of Gaussian Particle Filters," M. Bolić, A. Athalye, P. M. Djurić, S. Hong, submitted to *IEEE Transactions on Circuits and Systems I*, 2004. Chapter 6
- "Design and Implementation of Flexible Resampling Mechanism for High-Speed Parallel Particle Filters," S. Hong, S. S. Chin, M. Bolić, P. M. Djurić, submitted to *Journal of VLSI Signal Processings*, 2003. Section 5.4
- "Resampling Algorithms and Architectures for Distributed Particle Filters," M. Bolić, P. M. Djurić, S. Hong, submitted to *IEEE Transactions on Signal Processing*, 2003. Chapter 5

Chapter 2

Background on the theory of particle filtering

2.1 Background

In practice, the most commonly used devices for sequential signal processing apply algorithms based on the Kalman filter. When the addressed problems are linear in the unknowns and the noise is additive and Gaussian, the Kalman filter is the optimal solution [3, 52, 60, 80]. In cases of nonlinear signal models and/or non-Gaussian noise, one uses approximations of which the most popular choice is the extended Kalman filter [3, 52, 60, 80]. In many situations, this filter has poor performance with large biases [105], divergence [49], and lack of robustness [83].

Alternative approaches to sequential processing of nonlinear signals have been Gaussian sum filtering [2, 102], approximations of the first two moments of the densities [42, 82, 112], the unscented Kalman and related filters [59, 62], methods where relevant densities are evaluated over deterministic grids [16, 68, 73, 93, 103], and approaches which exploit Markov chain Monte Carlo (MCMC) sampling, including the Gibbs sampler or the Metropolis-Hastings scheme [7, 17, 19, 20].

Recently, much attention has been given to another group of methods known as sequential importance sampling procedures [38]. They are also known as particle filtering [18], bootstrap filtering [49], survivals of the fittest [65], condensation algorithms [81], and interacting particle approximations [28]. In this dissertation, we refer to them as particle filtering methods. In their derivation, the main principle is recursive generation of random measures that approximate the distributions of the unknowns. The random measures are composed of particles (samples) drawn from relevant distributions and of importance weights of the particles. These random measures allow for computation of all sorts of estimates of the unknowns, including minimum mean square error (MMSE) and maximum a posteriori (MAP) estimates. As new observations become available, the particles and the weights are propagated by exploiting Bayes theorem and the concept of sequential importance sampling [6, 36, 44, 45, 69].

2.2 Description of dynamic signals

Particle filters are used in non-linear problems where the interest is in tracking and/or detection of dynamic signals. The signals are described by a system of equations that model their evolution with time, and the equations usually have the form

$$\begin{aligned} \mathbf{x}_n &= \mathbf{f}_n(\mathbf{x}_{n-1}, \mathbf{u}_n) \\ \mathbf{z}_n &= \mathbf{g}_n(\mathbf{x}_n, \mathbf{v}_n) \end{aligned} \tag{2.1}$$

where $n \in \mathbb{N}$ is a discrete-time index, $\mathbf{x}_n \in \mathbb{R}^{d_x}$ is a signal vector of interest, and $\mathbf{z}_n \in \mathbb{R}^{d_z}$ is a vector of observations. The symbols $\mathbf{u}_n \in \mathbb{R}^{d_u}$ and $\mathbf{v}_n \in \mathbb{R}^{d_v}$ are noise vectors, $\mathbf{f}_n : \mathbb{R}^{d_x} \times \mathbb{R}^{d_u} \mapsto \mathbb{R}^{d_x}$ is a signal transition function, and $\mathbf{g}_n : \mathbb{R}^{d_x} \times \mathbb{R}^{d_v} \mapsto \mathbb{R}^{d_z}$ is a measurement function. The analytical forms of $\mathbf{f}_n(\cdot)$ and $\mathbf{g}_n(\cdot)$ are assumed known. The densities of $\mathbf{u}_n$ and $\mathbf{v}_n$ are parametric and known, but their parameters may be unknown, and $\mathbf{u}_n$ and $\mathbf{v}_n$ are independent from each other. The objectives are to estimate *recursively* the signal $\mathbf{x}_n$, $\forall n$, from the observations $\mathbf{z}_{1:n}$, where $\mathbf{z}_{1:n} = \{\mathbf{z}_1, \mathbf{z}_2, \cdots, \mathbf{z}_n\}$.

2.2.1 Filtering, predictive and smoothing densities

There are three densities that play a critical role in sequential signal processing, and they are known as the *filtering density*, $p(\mathbf{x}_n|\mathbf{z}_{1:n})$, the *predictive density*, $p(\mathbf{x}_{n+l}|\mathbf{z}_{1:n})$, $l \geq 1$, and the *smoothing density*, $p(\mathbf{x}_n|\mathbf{z}_{1:T})$, $T > n$. When in sequential signal processing we apply filtering, prediction or smoothing, they respectively carry all the information about the unknowns. Therefore, the recursive expressions of these densities provide the necessary operations when filtering, smoothing, or prediction are implemented. For filtering, we can write

$$p(\mathbf{x}_n|\mathbf{z}_{1:n}) = \frac{p(\mathbf{z}_n|\mathbf{x}_n)p(\mathbf{x}_n|\mathbf{z}_{1:n-1}) \int p(\mathbf{x}_n|\mathbf{x}_{n-1})p(\mathbf{x}_{n-1}|\mathbf{z}_{1:n-1})d\mathbf{x}_{n-1}}{\int\int p(\mathbf{z}_n|\mathbf{x}_n)p(\mathbf{x}_n|\mathbf{x}_{n-1})p(\mathbf{x}_{n-1}|\mathbf{z}_{1:n-1})d\mathbf{x}_{n-1}d\mathbf{x}_n} \tag{2.2}$$

and we see that high dimensional integrations are needed in the recursion of the filtering density at time n, $p(\mathbf{x}_n|\mathbf{z}_{1:n})$, from the filtering density at time $n-1$, $p(\mathbf{x}_{n-1}|\mathbf{z}_{1:n-1})$. The situation is similar for the predictive density, although the recursive relationship is more direct, i.e.,

$$p(\mathbf{x}_{n+l}|\mathbf{z}_{1:n}) = \int p(\mathbf{x}_{n+l}|\mathbf{x}_{n+l-1})p(\mathbf{x}_{n+l-1}|\mathbf{z}_{1:n})d\mathbf{x}_{n+l-1} \tag{2.3}$$

and for the smoothing density, where we have

$$p(\mathbf{x}_n|\mathbf{z}_{1:n}) = p(\mathbf{x}_n|\mathbf{z}_{1:n}) \int \frac{p(\mathbf{x}_{n+1}|\mathbf{z}_{1:n})\ p(\mathbf{x}_{n+1}|\mathbf{x}_n)}{p(\mathbf{x}_{n+1}|\mathbf{z}_{1:n})} d\mathbf{x}_{n+1}. \tag{2.4}$$

Obviously, the recursion for smoothing is carried out backwards, that is, $p(\mathbf{x}_n|\mathbf{z}_{1:n})$ is obtained from $p(\mathbf{x}_{n+1}|\mathbf{z}_{1:n})$. For convenience, in (2.2–2.4) we assumed that the fixed parameters of the model are known, and we proceed with this assumption in the rest of the dissertation.

In all of these equations, the integration is a key operation, and closed form solutions are possible in a very small number of situations. An important case is the linear model with additive Gaussian noise for which the solution is the well known Kalman filter [63, 64]. When the models are nonlinear or the noise is non-Gaussian, one resorts to approximated solutions, for example, obtained by the extended Kalman filter [3, 52, 60, 80] or Guassian sum filters [2, 102].

2.3 Particle filters

Particle filters base their operation on representing relevant densities by random measures composed of particles and weights and compute integrals by Monte Carlo methods. More specifically, at every time instant n a random measure $\{\mathbf{x}_{1:n}^{(m)}, w_n^{(m)}\}_{m=1}^M$ is defined, where $\mathbf{x}_n^{(m)}$ is the $m-$th particle of the signal at time n, $\mathbf{x}_{1:n}^{(m)}$ is the $m-$th trajectory of the signal, and $w_n^{(m)}$ is the weight of the $m-$th particle (or trajectory) at time instant n. If these particles are obtained from the observations $\mathbf{z}_{1:n}$ and the trajectories they form are drawn from the density $p(\mathbf{x}_{1:n}|\mathbf{z}_{1:n})$, then they approximate this density by

$$p(\mathbf{x}_{1:n}|\mathbf{z}_{1:n}) \simeq \sum_{m=1}^{M} w_n^{(m)} \delta(\mathbf{x}_{1:n} - \mathbf{x}_{1:n}^{(m)}).$$

If now, for example, an estimate of $E(h(\mathbf{x}_{1:n}))$ is needed, where $h(\cdot)$ is a function of $\mathbf{x}_{1:n}$, the estimate can be easily computed from

$$\hat{E}(h(\mathbf{x}_{1:n})) = \sum_{m=1}^{M} w_n^{(m)} h(\mathbf{x}_{1:n}^{(m)}). \tag{2.5}$$

In the implementation of particle filters, there are three important operations:

1. generation of particles (sample step),
2. computation of the particle weights (importance step), and
3. resampling.

The first two steps form the particle filter called Sequential Importance Sampling (SIS) filter. The filter that performs all three operations is called Sample Importance Resampling Filter (SIRF).

2.3.1 Generation of particles

The generation of particles $\mathbf{x}_n^{(m)}$ is performed by drawing them from an importance density function, $\pi(\mathbf{x}_n)$. If we choose an importance function whose form is

$$\pi(\mathbf{x}_{1:n}) = \pi(\mathbf{x}_1|\mathbf{z}_1) \prod_{k=1}^{n} \pi(\mathbf{x}_k|\mathbf{x}_{1:k-1}, \mathbf{z}_{1:k}) \tag{2.6}$$

we can compute the weights of the particles recursively. Namely, the particles $\mathbf{x}_n^{(m)}$ are drawn according to

$$\mathbf{x}_n^{(m)} \sim \pi\left(\mathbf{x}_n|\mathbf{x}_{n-1}^{(m)}, \mathbf{z}_{1:n}\right). \tag{2.7}$$

The importance density $\pi(\mathbf{x}_n|\mathbf{x}_{n-1}, \mathbf{z}_{1:n})$ plays a pivotal role in the design of particle filters because it generates particles that have to represent a desired density. If the drawn particles

are in regions of the signal space where the density has negligible values, the estimates obtained from the particles and their weights would be poor and subsequent tracking of the signal would very likely diverge. By contrast, if the particles are from regions where the probability mass is significant, the particle filter will have improved performance.

The role of the importance density is well understood, and in the literature various strategies have been proposed for its design [24, 36, 81, 88]. One might argue that $p(\mathbf{x}_n|\mathbf{x}_{1:n-1}^{(m)}, \mathbf{z}_n)$ is an optimal importance function [36, 116]. However, it has drawbacks in that it is difficult for sampling and the updating of the particle weights requires integrations. Suboptimal functions have been proposed using local linearizations [36] and Gaussian approximations of it using the unscented transform [108]. A different approach was introduced in [92], where the samples are obtained by a two step procedure and the proposed filter is known as auxiliary particle filter. Another method for drawing particles was presented in [86], where resampling is carried out from a continuous approximation of the posterior density $p(\mathbf{x}_n|\mathbf{z}_{1:n})$.

2.3.2 Importance step

The importance step (I) consists of two steps: computation of the weights and normalization. In the former step the weights are evaluated up to a proportionality constant and subsequently, in the latter they are normalized. If the importance function has the form (2.6), the weights are updated via

$$w_n^{*(m)} = w_{n-1}^{(m)} \frac{p(\mathbf{z}_n|\mathbf{x}_n^{(m)}) p(\mathbf{x}_n^{(m)}|\mathbf{x}_{n-1}^{(m)})}{\pi(\mathbf{x}_n^{(m)}|\mathbf{x}_{1:n-1}^{(m)}, \mathbf{z}_{1:n})}. \tag{2.8}$$

The normalization is carried out by

$$w_n^{(m)} = \frac{w_n^{*(m)}}{\sum_{j=1}^{M} w_n^{*(j)}}. \tag{2.9}$$

2.3.3 Resampling

One important problem of particle filters is that the weights of the particles degenerate. In other words, as time progresses, a few weights become very large and the remaining decrease in value to the point that they become negligible. The idea of resampling is to remove the trajectories that have small weights and to focus on trajectories that are dominating. Resampling was first introduced in [100] and proposed for SIS in various works [7],[70],[76]. A detailed theoretical discussion of resampling in the SIS context is given in [76]. Standard algorithms used for resampling are different variants of stratified sampling such as residual resampling (RR), branching corrections [28], systematic resampling (SR) [18, 38, 49] as well as resampling methods with rejection control [78, 79]. Formally, the basic random resampling algorithm is performed as follows [79]:

1. Let $\widetilde{\mathbf{x}}_n^{(i^{(m)})}$ be drawn from $\mathbf{x}_n^{(m)}$ independently with probability proportional to $a_n^{(m)}$ for $m = 1, ..., M$ and $i^{(m)} = 1, ..., M$. New weights associated with these samples are $\widetilde{w}_n^{(i^{(m)})} = {w_n}^{(m)}/a^{i^{(m)}}$.

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

Here, $i^{(m)}$ represents the indexes in the memory where particles are stored as a result of resampling. The above representation of the particle filter algorithm provides a certain level of generality. For example, the SIRF with a stratified resampling is implemented by choosing $a_n^{(m)} = w_n^{(m)}$ for $m = 1, ..., M$. Uniform resampling function $a_n^{(m)} = 1/M$ corresponds to the case without resampling. The auxiliary sample importance resampling (ASIR) filter can be implemented by setting $a_n^{(m)} = w_n^{(m)} p(\mathbf{z}_{n+1}|\boldsymbol{\mu}_{n+1}^{(m)})$ and $\pi(\mathbf{x}_n) = p(\mathbf{x}_n|\mathbf{x}_{n-1}^{(m)})$, where $\boldsymbol{\mu}_n^{(m)}$ is the mean, the mode or some other likely value associated with the density $p(\mathbf{x}_n|\mathbf{x}_{n-1}^{(m)})$.

Some of the resampling algorithms (for instance RR) deal with the array replication factors $r^{(m)}$ instead of the array indexes $i^{(m)}$ for $m = 1, ..., M$. Replication factors show how many times each particle is replicated as a result of resampling. Then, resampling is performed in a way that $r_n^{(m)}$ is sampled from $a_n^{(m)}$ whose support is defined by the particle $\mathbf{x}_n^{(m)}$ for $m = 1, ..., M$.

Resampling is very important in particle filtering because it prevents the particle filter from weight degeneracy. On the other hand, frequent resampling may lead to particle attrition, that is, degradation of the support of the particle filter. This loss of diversity occurs because the resulting random measure contains many copies of the same particle.

Resampling improves the estimation of future states by concentrating particles into domains of higher posterior probability. However, it reduces the accuracy of the current estimate by increasing the variance of the estimate after resampling [40]. So, resampling must be applied with caution. A common technique for assessing the need for resampling calculates an effective sample size defined by the variation of the weights [78]. Simple way to reduce the average number of resampling operations is to perform resampling for every n time instants. One commonly used metric for deciding when to resample is the effective sample size, which is the number of particles that represent the distribution if the associated weights were equal. Two estimators of the samples size were proposed in [18, 77]. The direct approach of resampling has a complexity of order $O(M \log M)$, where M is the number of particles. This complexity can be reduced to $O(M)$ if a set of M ordered uniform variates are obtained [99].

Review of the systematic resampling and residual resampling algorithms

In this section, resampling algorithms are reviewed. Note that here we do not consider rejection control algorithms because they are not suitable for high speed implementations. Their time for resampling cannot be determined beforehand because the execution time itself is a random variable. In other words, the algorithm requires random number generation, where the number of random draws cannot be predicted and is variable due to random rejections.

The SR algorithm performs resampling in the same way as the basic random resampling algorithm, with one exception. Instead of drawing each $U^{(m)}$ independently from $\mathcal{U}(0, 1)$ for $m = 1, ..., M$, where M is the number of resampled particles, it uses a uniform random number U according to $U \sim \mathcal{U}[0, \frac{1}{M}]$, and $U^{(m)} = U + (m - 1)/M$.

Figure 2.1 graphically illustrates the SR methods for the case of $M = 5$ particles with weights given in the table. SR calculates the cumulative sum of the weights $C^{(m)} = \Sigma_{i=1}^{m} w_n^{(i)}$,

and compares $C^{(m)}$ with the updated uniform number $U^{(m)}$ for $m = 1, ...N$. The uniform number $U^{(0)}$ is generated by drawing from the uniform distribution $\mathcal{U}[0, \frac{1}{M}]$ and updated by $U^{(m)} = U^{(m-1)} + 1/M$. The number of replications for particle m is determined as the number of times the updated uniform number is in the range $[C^{(m-1)}, C^{(m)})$. For particle one, $U^{(0)}$ and $U^{(1)}$ belong to the range $[0, C^{(1)})$, so that this particle is replicated twice, which is shown with two arrows that correspond to the first particle. Particles two and three are replicated once. Particle four is discarded ($r^{(4)} = 0$) because no $U^{(m)}$ for $m = 1, ..., N$ appears in the range $[C^{(3)}, C^{(4)})$.

m	$w^{(m)}$	$r^{(m)}$
1	7/20	2
2	6/20	1
3	2/20	1
4	2/20	0
5	3/20	1

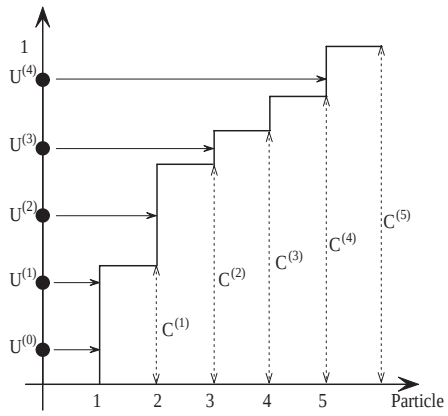


Figure 2.1: Systematic resampling for an example with $M = 5$ particles.

Purpose: Generation of an array of replication factors $\{r^{(m)}\}_{m=1}^{N}$ at time instant n, $n > 0$ using SR.
Input: An array of weights $\{w_n^{(m)}\}_{m=1}^{N}$, input and output number of particles, N and M, respectively.
Method:

```
(r)=SR(w, N, M)
    U ~ U(0, 1/M]        //Generate random number U
    s = 0                // Initializing cumulative sum of weights
    for m = 1 : N        // The main loop
        k = 0            // Replication factor counter
        s = s + w^(m)    // Updating the cumulative sum of weights
        while (s > U)    // Resampling loop
            k = k + 1
            U = U + 1/M
        end
        r^(m) = k        // Storing replication factors
    end
```

Pseudocode 1 Systematic resampling (SR) algorithm.

One possible algorithm for systematic resampling is shown in Pseudocode 1. Here, N is the input number of particles, M is the number of particles generated after resampling, and w is an array of normalized weights from the importance step. The output r is an array of replication factors, which shows how many times each particle is replicated. It is important to observe that SR is implemented using *two loops*.

The RR algorithm is shown in Pseudocode 2. N, M, w and r have the same meaning as in Pseudocode 1. RR is composed of two steps. In the first step, the number of replications of particles is calculated. Since this method does not guarantee that the number of resampled particles is M, the residual number of particles M_r is computed. Also, the residual weights $w_r^{*(m)}$ are computed. The second step requires resampling of residuals which produces M_r of the final M particles. In Pseudocode 2, this step is performed by SR with N input and M_r output particles. Before the SR algorithm can be applied, the residual weights $w_r^{*(m)}$ are normalized and the new weights $w_r^{(m)}$ are obtained for $m = 1, ..., M$. Finally, replication factors produced in these two steps are summed.

Purpose: Generation of an array of replication factors $\{r^{(m)}\}_{m=1}^{N}$ at time instant n, $n > 0$ using RR.
Input: An array of weights $\{w_n^{(m)}\}_{m=1}^{N}$, input and output number of particles, N and M, respectively.
Method:

```
(r)=RR(w, N, M)
    M_r = M                          // The number of particles left for resampling.
    for m = 1 : N                    // First step: the number of replications are computed.
        r^(m) = ⌊w^(m) · M⌋
        w_r^*(m) = w^(m) · M − r^(m) // The residues of the weights.
        M_r = M_r − r^(m)            // Updating the number of particles left for resampling.
    end
    if M_r > 0                       // Second step: processing residuals.
        for m = 1 : N                // Normalization of residual weights.
            w_r^(m) = w_r^*(m)/W_r   // W_r = Σ_{i=1}^{N} w_r^*(i)
        end
        (r_r)=SR(w_r, N, M_r)        // SR with the residual weights.
        for m = 1 : N
            r^(m) = r^(m) + r_r^(m)  // Updating replication factors.
        end
    end
```

Pseudocode 2 Residual resampling (RR) algorithm.

The best case in terms of speed of execution of the RR algorithm occurs when $M_r = 0$ (for example, when $w^{(m)}M$ is an integer for all $m = 1, 2, ..., M$). In that case there is only one step with m iterations. The worst case of RR arises when $M_r = N - 1$. One example is when one particle has a weight in the range $[1/N, 2/N)$ and the remaining $N - 1$ particles have weights less than $1/N$. Then step 2 requires resampling of $M_r = N - 1$ residual particles.

2.4 Recent progress

Recent advances in particle filtering have been in the improvements of various methods, and they include the design of importance functions, resampling strategies with decreased computational complexity, development of suboptimal algorithms with reduced computational

complexity, and results on convergence of particle filters. Many new contributions represent novelties in applications of particle filters. They include target tracking [47, 49], navigation [5], blind deconvolution of digital communication channels [23, 76], joint channel estimation and detection in Rayleigh fading channels [21, 22, 94], digital enhancement of speech and audio signals [48], time-varying spectrum estimation [37], computer vision [58], portfolio allocation [1], and sequential estimation of signals under model uncertainty [34].

2.5 Particle filters for the bearings-only tracking problem

In this dissertation particle filters are applied to the bearings-only tracking problem illustrated in Figure 2.2 where two positions of the object at time instants n and $n+1$ are shown. The measurements taken by the sensor to track the object are the bearings or angles (z_n) with respect to the sensor, at fixed intervals. The range of the object, that is the distance from the sensor is not measured. The unknown states that we are interested to estimate are the position and velocity of the tracked object in the Cartesian coordinate system ($\mathbf{x}_n = [x_n, V_{x_n}, y_n, V_{y_n}]^T$).

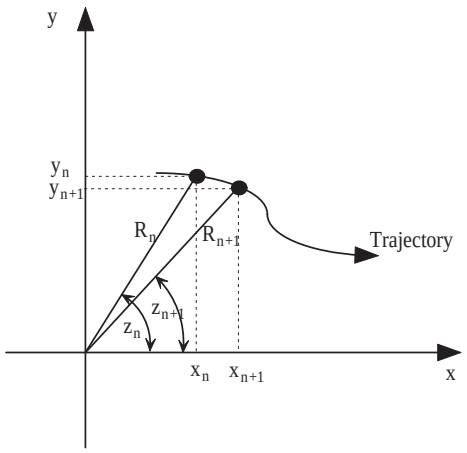


Figure 2.2: Illustration of the tracking problem.

The object moves in the $x-y$ plane according to the following state model [49]:

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

where $\mathbf{x}_n = [x_n,\ V_{x_n},\ y_n,\ V_{y_n}]^T, \mathbf{u}_n = [u_{x_n},\ u_{y_n}]^T$,

$$\mathbf{\Phi} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \mathbf{\Gamma} = \begin{bmatrix} 0.5 & 0 \\ 1 & 0 \\ 0 & 0.5 \\ 0 & 1 \end{bmatrix}.$$

Here x and y denote the cartesian coordinates of the target, V_x and V_y denote the velocities in the x and y directions, respectively. The system noise is a zero mean Gaussian white noise,

that is $\mathbf{u}_n \sim N(\mathbf{0}, q\mathbf{I}_2)$, where $\mathbf{I}_2$ is the 2×2 identity matrix. The initial state $\mathbf{x}_1$ describes the targets initial position and velocity. A prior for the initial state $p(\mathbf{x}_1)$, also needs to be specified for the model; we assume $\mathbf{x}_0 \sim N(\mu_0, \mathbf{P}_0)$.

The measurements consist of the true bearing of the target corrupted by a Gaussian error term. The measurement equation can be written as $z_n = \tan^{-1}(y_n/x_n) + v_n$ where the measurement noise is a zero mean Gaussian white noise, that is $v_n \sim N(0, r)$. Performance analysis of SIRFs applied to bearings-only tracking problem is given in [9].

The SIRF algorithm for bearings-only tracking utilizes the usual steps of sampling, importance, resampling and output calculation as shown in Pseudocode 3. We choose the prior importance function because it allows for the simplest implementation.

Purpose: Obtain the estimates of position and velocity $\hat{x}_n$,$\hat{V}_{x_n}$,$\hat{y}_n$,$\hat{V}_{y_n}$.
Input: The observation z_n and the particles $\{\mathbf{x}_{n-1}^{(m)}\}_{m=1}^{M}$.
Method:

1. *Sampling step (S)*
   Draw sample from $p(\mathbf{x}_n|\mathbf{x}_n = \mathbf{x}_{n-1}^{(m)})$ to obtain $\{\mathbf{x}_n^{(m)}\}_{m=1}^{M}$ for $m = 1, \ldots, M$.
   $x_n^{(m)} = x_{n-1}^{(m)} + V_{x_{n-1}}^{(m)} + u_{x_n}^{(m)}$
   $V_{x_n}^{(m)} = V_{x_{n-1}}^{(m)} + u_{x_n}^{(m)}$
   $y_n^{(m)} = y_{n-1}^{(m)} + V_{y_{n-1}}^{(m)} + u_{y_n}^{(m)}$
   $V_{y_n}^{(m)} = V_{y_{n-1}}^{(m)} + u_{y_n}^{(m)}$

2. *Importance step (I)*
   Calculate weights by $w_n^{*(m)} = p(z_n|\mathbf{x}_n^{(m)}) = w_{n-1}^{(m)} e^{-(2\pi\sigma_v^2)^{-1}(z_n - atan\frac{y_n^{(m)}}{x_n^{(m)}})^2}$, for $m = 1, \ldots, M$.

   Normalize the weights by $w_n^{(m)} = w_n^{*(m)} / \sum_{m=1}^{M} w_n^{*(m)}$, for $m = 1, \ldots, M$.

3. *Resample* particles to obtain new particles and weights $\{\widetilde{\mathbf{x}}_n^{(m)}, \widetilde{w}_n^{(m)} = \frac{1}{M}\}_{m=1}^{M}$.

4. *Calculate the outputs*
   $\hat{x}_n = \sum_{m=1}^{M} w_n^{(m)} x_n^{(m)}$
   $\hat{V}_{x_n} = \sum_{m=1}^{M} w_n^{(m)} V_{x_n}^{(m)}$
   $\hat{y}_n = \sum_{m=1}^{M} w_n^{(m)} y_n^{(m)}$
   $\hat{V}_{y_n} = \sum_{m=1}^{M} w_n^{(m)} V_{y_n}^{(m)}$

Pseudocode 3 SIRFs applied to the bearings-only tracking problem.

2.6 Gaussian particle filtering

It can be seen that SIRFs operate by propagating the filtering and the predictive densities recursively in time. GPFs on the other hand operate by approximating desired densities as Gaussians. Hence only the mean and the variance of the densities are propagated recursively in time. In brief, GPFs are a class of Gaussian filters in which Monte Carlo (particle filter based) methods are employed to obtain the estimates of the mean and covariance of the relevant densities and these estimates are recursively updated in time [71], [72]. Propagation of the first two moments only instead of the whole particle set significantly simplify the parallel implementation of the GPF. Even though this approximation of the filtering and predictive densities using unimodal Gaussian distributions restricts the application of GPFs, there is still a broad class of models for which this approximation is valid.

The GPF can be considerably simplified when the prior density is used as importance function. This means that $\pi(\mathbf{x}_n|\mathbf{x}_{n-1}, \mathbf{z}_{1:n})$ is given by $p(\mathbf{x}_n|\mathbf{x}_{n-1})$. The GPF operations are shown in Pseudocode 4. In the first step, the conditioning particles are drawn from the Gaussian density with mean vector and covariance matrix that are computed in the previous particle filter recursion. These conditioning particles are used in the sample step where particles $\{\mathbf{x}_n^{(m)}\}_{m=1}^M$ are drawn from $p(\mathbf{x}_n \mid \mathbf{x}_n = \mathbf{x}_{n-1}^{(m)})$. Weights are computed and normalized in the same way as in SIRFs. Finally, particles and their weights are used to compute the mean vector and covariance matrix.

Purpose: Obtain the estimates of the states $\boldsymbol{\mu}_n$.
Input: The observation $\mathbf{z}_n$ and previous estimates $\boldsymbol{\mu}_{n-1}$ and $\boldsymbol{\Sigma}_{n-1}$
Setup: Mean $\boldsymbol{\mu}_0$ and covariance $\boldsymbol{\Sigma}_0$ based on prior information.
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 $p(\mathbf{x}_n \mid \mathbf{x}_n = \mathbf{x}_{n-1}^{(m)})$ to obtain $\{\mathbf{x}_n^{(m)}\}_{m=1}^M$.

GPF - Measurement update algorithm

3. (a) Calculate weights by $w_n^{*(m)} = p(\mathbf{z}_n \mid \mathbf{x}_n^{(m)})$.
   (b) Normalize the weights by $w_n^{(m)} = w_n^{*(m)} / \sum_{m=1}^M w_n^{*(m)}$.
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) $\boldsymbol{\Sigma}_n = \sum_{m=1}^M w_n^{(m)} (\mathbf{x}_n^{(m)} - \boldsymbol{\mu}_n)(\mathbf{x}_n^{(m)} - \boldsymbol{\mu}_n)^\top$.

Pseudocode 4 GPF algorithm with the prior density as the importance function.

2.6.1 GPF for bearings-only tracking

Operations of GPFs for the bearings-only tracking problem are shown in Pseudocode 5. Conditioning particles are drawn form Gaussian distribution with parameters $(\boldsymbol{\mu}_n, \boldsymbol{\Sigma}_n)$ where $\boldsymbol{\mu}_n$ is 4×1 vector and $\boldsymbol{\Sigma}_n$ is 4×4 triangular matrix. This step requires generation of four Gaussian random numbers per particle filtering iteration. Independent Gaussian random numbers with zero mean and unit variance are labeled as $q_1, \ldots, q_4$. In order to compute

random numbers with proper variance it is necessary to find matrix $\mathbf{C}_n$ where $\mathbf{\Sigma}_n = \mathbf{C}_n \cdot \mathbf{C}_n^\top$. Particle generation and weight computation steps are the same as for SIRFs and they are based on the state and observation equations from the bearings-only tracking model. The output estimate of the GPF is the mean of position and velocity. Besides the mean, the coefficients of the covariance matrix are computed so that they can be used in the next particle filtering recursion.

Purpose: Obtain the estimates of position and velocity $\boldsymbol{\mu}_n = \{\mu_{x_n}, \mu_{v_{x_n}}, \mu_{y_n}, \mu_{v_{y_n}}\}$.
Input: The observation z_n and previous estimates $\boldsymbol{\mu}_{n-1}$ and $\mathbf{C}_{n-1}$ where $\mathbf{C}_n$ is Cholesky decomposed covariance matrix $\mathbf{\Sigma}_n$.
Method:

1. Draw conditioning particles (for $m = 1, \ldots, M$):
   $x_{n-1}^{(m)} = \mu_{x_{n-1}} + C_{11}q_1$
   $v_{x_{n-1}}^{(m)} = \mu_{v_{x_{n-1}}} + C_{21}q_1 + C_{22}q_2$
   $y_{n-1}^{(m)} = \mu_{y_{n-1}} + C_{31}q_1 + C_{32}q_2 + C_{33}q_3$
   $v_{y_{n-1}}^{(m)} = \mu_{v_{y_{n-1}}} + C_{41}q_1 + C_{42}q_2 + C_{43}q_3 + C_{44}q_4$

2. Generate particles (for $m = 1, \ldots, M$):
   $x_n^{(m)} = x_{n-1}^{(m)} + v_{x_{n-1}}^{(m)} + u_{x_n}^{(m)}$
   $v_{x_n}^{(m)} = v_{x_{n-1}}^{(m)} + u_{x_n}^{(m)}$
   $y_n^{(m)} = y_{n-1}^{(m)} + v_{y_{n-1}}^{(m)} + u_{y_n}^{(m)}$
   $v_{y_n}^{(m)} = v_{y_{n-1}}^{(m)} + u_{y_n}^{(m)}$

3. (a)Calculate weights by $w_n^{*(m)} = w_{n-1}^{(m)} e^{-(2\pi\sigma_v^2)^{-1}(z_n - atan\frac{y_n^{(m)}}{x_n^{(m)}})^2}$, for $m = 1, \ldots, M$.

   (b) Normalize the weights by $w_n^{(m)} = w_n^{*(m)} / \sum_{m=1}^{M} w_n^{*(m)}$, for $m = 1, \ldots, M$.

4. Estimate the mean and covariance by
   $\boldsymbol{\mu}_n = \sum_{m=1}^{M} w_n^{(m)} \mathbf{x}_n^{(m)}$
   $\mathbf{\Sigma}_n = \sum_{m=1}^{M} w_n^{(m)} (\mathbf{x}_n^{(m)} - \boldsymbol{\mu}_n)(\mathbf{x}_n^{(m)} - \boldsymbol{\mu}_n)^\top$.

Pseudocode 5 GPF applied to the bearings-only tracking problem.

Chapter 3

Characterization of the particle filtering algorithms and architectures

In this chapter, architectural and algorithmic properties of SIRFs are analyzed. The chapter starts with the brief introduction into the field of VLSI signal processing. Then, the modifications of the sequential SIRFs are presented so that they become suitable for hardware implementation. These modifications are based on exploiting operational concurrency. Parallel algorithms and architectures for SIRFs are shown as well. Finally, the savings in the size of memories and the number of memory accesses are considered for different addressing schemes.

3.1 VLSI signal processing

3.1.1 Joint algorithm and architecture development

Many new applications would not be possible nowadays without reducing the complexity of signal processing algorithms despite the remarkable improvements in VLSI technologies [96, 98]. As technology evolves, entirely new domains of application open. Technology improvements have allowed for using Monte Carlo sampling techniques in complex simulations and for applying these techniques in slower real-time systems. In this dissertation we show that high speed implementation of sequential importance sampling algorithms is feasible using current technology.

The results published in [66, 95] have made it apparent that the most dramatic power reduction and speed improvements stem from optimization at the highest level of design hierarchy. In particular, case studies indicate that high-level decisions regarding selection and optimization of algorithms and architectures can improve design performance metrics much more effectively than gate and circuit level optimizations. This suggests that the estimation of architectural parameters is needed to facilitate rapid high-level design exploration, especially when the goal is to achieve an area efficient, high-speed circuit implementation. One of the important aspects of the field of VLSI signal processing is focusing on joint study of both algorithms and architectures for custom implementation of digital signal processing systems [51]. If the main design goal is high speed, then it is extremely important that algorithms match

the architectures well. By matching algorithms and architectures, the number of operations is not decreased, but the overhead caused by memory access and data communication can be reduced. By reducing this overhead, together with exploiting concurrency that exists in the algorithm in hardware, a significant increase in speed can be achieved.

"If the new functionality takes advantage of mathematical approaches for which good fast algorithms are not yet known, the application can potentially benefit from both hardware performance improvements and from algorithm efficiency breakthroughs [1]". In this dissertation, the benefit is derived from the joint algorithm and architecture development of particle filters. This work is inline with the current trend toward hardware intensive signal processing implementations. Since particle filters are very complex algorithms, the main goal of this dissertation is to increase the speed of particle filters. By focusing on speed improvements we hope that particle filters will become appropriate for most real-time signal processing applications. The speed is increased through gradual modification of the algorithms and through developing architectures for these algorithms. Hence, in this dissertation we propose a set of modifications of the particle filter algorithm from the standpoint of hardware implementation and a set of corresponding architectures.

3.1.2 Types of signal processing architectures

The main architectures for signal processing are programmable DSP, application-domain specific processors (ADSP), and application specific processors (ASP) [89, 91, 109]. DSP processors and DSP cores are used when low speed and high flexibility are necessary. ADSPs are designed to execute only a fixed or limited set of algorithms with a restricted number of adjustable parameters. Higher speed can be achieved if the designed space is further narrowed. The ASPs are optimized to execute a single algorithm. The result is very efficient implementation at the expense of flexibility. For every change in specifications, a redesign is required.

From the system level point of view, we expect that the particle filter will be used as a subsystem in a multifunctional system such as a radar or communication system. Since the functionality of the algorithm does not change in a system and speed is the main design goal, we design a particle filter as a fixed-function subsystem. So, in this dissertation, only the ASP architectures with single-chip implementation of particle filters are considered.

3.1.3 Basic terminology

The *latency* of an algorithm is the time it takes to generate an output value from the corresponding input value [50]. *Throughput* is defined as the reciprocal of the time difference between successive outputs. We define the *execution time* of particle filters as the time necessary to process one observation by the particle filter. The execution time also corresponds to the sampling period. The *minimum execution time* is defined as the minimum sampling period that can be achieved with particle filters when there are unrestricted hardware resources. The minimum sampling period of an algorithm is defined only by its recursive parts. Nonrecursive

[1] cited from [98]

parts usually do not limit the sample rate.

Concurrency metrics capture the ability for resource to be accessed in parallel. Studies to determine the amount of available concurrency have been conducted, and results have influenced both architecture and compiler design [61, 74, 87, 107]. In this dissertation, concurrency of operations and spatial concurrency will be considered the most.

The concurrency of operations is a type of temporal concurrency which quantifies the expected number of operations that will be simultaneously executed. It is also referred to as *functional parallelism*. The main technique used here is *chaining* [27, 101]. Chaining makes it possible to pipeline arithmetic operations between the functional units. Then, the output of one functional unit can be the input of another functional unit. Chaining is done through *loop fusion* on the algorithm level. Loop fusion allows for combining two or more loops that are executed the same number of times using the same indices.

In order to obtain the best performances from a parallel system, the application should be partitioned into a set of tasks that can be executed concurrently - *spatial concurrency*. In evaluating the ability of making the algorithm parallel, the degree of parallelism and data dependency are considered. *The degree of parallelism* is a measure of the numbers of threads of computation that can be carried out simultaneously. Data dependencies are the result of precedence constraints between operations. Data dependencies are studied through temporal and spatial locality. The concept of locality has been heavily studied during the last three decades (e.g. [75, 90, 113]). *Temporal locality* is described as the tendency for a program to reuse the data or instructions which have recently been used. *Spatial locality* is the tendency for a program to use the data or instructions neighboring those which were recently used.

Topology metrics characterize the relative arrangement of operations, without regard to their functionality.

Speed-up and efficiency [29, 101] are metrics for performance evaluation of parallel systems. *Speed-up* is defined as the ratio of the execution time of the best possible serial algorithm (on a single processor) to the execution time of the chosen algorithm on a parallel system based on n-processors. *Efficiency* is defined as the ratio of speed-up to the number of processors.

Since the main requirement is to increase the speed of particle filters, *one-to-one mapping* from the algorithm to the architecture is performed in most cases. This means that each particle filtering operation has its own hardware block. In this way, the highest speed can be achieved at the expense of hardware resources. On the other hand, *functional multiplexing* is used when there is no significant increase in performance when one-to-one mapping is used. When functions are multiplexed, the same hardware block can be used to execute different operations at different time instants.

3.2 Algorithm modifications

3.2.1 SIRFs - functional view

The functional view of a SIRF is shown in Figure 3.1. For one input sample (observation $\mathbf{z}_n$), SIRFs sequentially perform particle generation, weight computation, normalization,

resampling, and output calculation. The first four operations are in the critical path, which means that optimization techniques for speed have to be applied to these steps. A particle filter is an iterative algorithm, which means that the new particle generation step cannot start until the random measure from the resampling step is computed. SIRFs belong to the class of block processing algorithms, where each of the SIRF operations works on a block of data. As such, SIRF operations have large latencies.

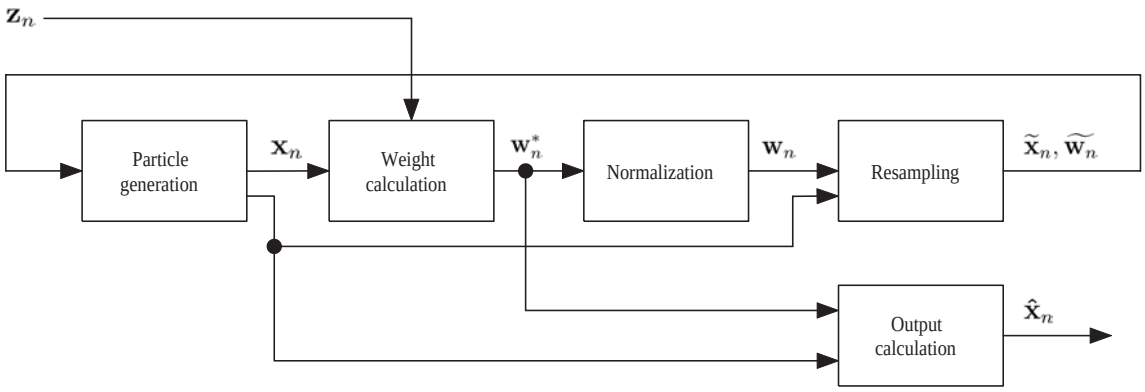


Figure 3.1: Functional view of SIRFs.

If we consider SIRF topology, it is clear from Pseudocode 3 that particle generation and weight computation are model dependent. Resampling does not depend on the accepted state space model which is an important property of SIRFs. It allows for analyzing algorithmic and architectural properties of SIRFs for the generic state space model.

3.2.2 Complexity

The complexity of SIRFs depends on the complexity and dimension of the underlying state-space model. The number of operations per sample in SIRFs is immense. For one input sample in the bearings-only tracking problem, M exponential, atan and other complex functions have to be calculated and $2M$ Gaussian random numbers have to be generated. In order to get acceptable performances, the number of particles M should be of the order of thousands.

The main difference between SIRFs and other DSP algorithms can be seen not only in the number of operations that SIRFs perform, but also in the type of operations. SIRFs are used to solve non-linear problems, so that they operate using non-linear functions. It can be noticed in Pseudocode 3 that all the non-linear computations are placed in the importance step. In general, non-linear functions can be found in a sample step as well in two places: the generation of random numbers and the process equation. The type of non-linear functions depends on the problem at hand. The most common non-linear function is the exponential function since the most frequent assumption for the observation noise is that it is Gaussian.

A first order complexity of an algorithm can be obtained by counting the number and the type of involved arithmetic operations. The performances of SIRFs for the bearings-only tracking problem are evaluated using the popular TI TMS320C54x DSP [106] (generation C54 of Texas Instruments processors is chosen because its DSP library contains the transcendental

functions and random number generators). The overall number of clock cycles is $81.5M + 781$. If we choose $M = 1000$, and the maximum clock rate for the processor of $160MHz$, and take into account some control overhead, then the maximum sampling rate is $1.8kHz$.

3.2.3 Concurrency of operations

Normalization

The normalization step requires the use of an additional loop of M iterations as well as M divisions per observation. It has been noted that normalization represents an unnecessary step which can be merged with the resampling and the computation of the importance weights. In that case, there would be no normalization block in Figure 3.1 and the input to the resampling block are non-normalized weights $\mathbf{w}_n^*$, together with the sum of weights W_n. Avoiding normalization requires additional changes which depend on whether resampling is carried out at each time instant. For particle filters which perform resampling at each time instant, the following arguments should be used when the resampling routine is called from Pseudocode 1: $(r_n) = SR(w_n^*, N, M, W_n)$ where W_n is added for non-normalized weights. Since the weights are not normalized, the uniform random number in the systematic resampling routine should be drawn from $[0, W_n/M)$ and updated with W_n/M.

When particle filters do not perform resampling at each time instant, modifications in the computation of the importance weights are necessary. In Pseudocode 3, we can see that the calculation of a new weight requires multiplication with the weight from the previous time instant. When the weights are not scaled, this multiplication could cause dynamic range problems, and therefore, the weight should be scaled at this point. However, in order to avoid M divisions with the sum of the weights W_n, $\ln(W_n)$ can be calculated once per particle filtering recursion and added to the exponent of the exponential function.

Using this approach, only one division (W_n/M) is performed in the resampling step, which significantly reduces the dynamic range problem for fixed precision arithmetics which usually appears with division. The computational burden is reduced as well since the normalization requires M divisions.

Concurrency in the sample, importance and resampling steps

The SIRF presented in Pseudocode 3 is implemented in such a way that there is a separate loop of M iterations for each of the particle filtering operations. In this way, temporal locality that exists in SIRFs is not utilized. However, temporal locality is preserved in the particle generation and weight computation steps since these steps use local, recently calculated data. After the particle is generated in the sampling step, it can be immediately used for weight computation. Since there are no data dependencies between the particles, weight computation and particle generation steps can be overlapped in time, which corresponds to the process of loop fusion on the algorithmic level. In addition, output calculation can be overlapped in time with the weight computation and particle generation steps, i.e. as soon as a particle and its weight are known, it is possible to execute one multiple and accumulate (MAC) operation of the output calculation step.

The resampling step cannot be overlapped in time with the weight computation step because it is necessary that the overall sum of weights is known before resampling starts. The overall sum of weights is known only after all the weights in the weight computation step are calculated. However, it is possible for resampling to overlap in time with the next particle generation step. Namely, as soon as one particle is resampled, it can be used as an input to the particle generation step. The SIRF in which the operations are overlapped in time is shown in Pseudocode 6. Loops from the sample, importance, and output calculation steps are fused. Normalization is not used. Since non-normalized weights are used for output calculation, it is necessary to divide the final result of output calculation with the sum of weights (output scaling). Resampling is fused with the particle generation step. Particles that have to be replicated are used in the next particle generation step directly. There are three counters in the loops: m is used to address the particles from the previous time instant, k is used for placing new sampled particles and l is used to count the number of times each particle is replicated.

3.3 Architectures

3.3.1 Temporarily concurrent architecture

In order to achieve minimum execution time, one-to-one mapping between the particle filtering operations and hardware resources is done which allows for utilizing operational concurrency. Hence, several operations can be executed at the same time and their blocks are pipelined in hardware implementation. Figure 3.2 shows the timing diagram with the latencies of different operations. The timing diagram is based on the algorithm described in Pseudocode 6 and the architecture presented in Figure 3.1. The output of each block is generated at clock speed. The only exception is the output calculation block which output is generated once per particle filter sampling period.

Particle generation, weight calculation, output calculation and a part of resampling are pipelined which is possible because these operations are performed in the same loop and all the variables are locally used (concept of temporal locality). The latency of these operations is $(M + L_S + L_I) \cdot T_{clk}$ where T_{clk} is the clock period, M is the number of particles and L_S and L_I are latencies due to fine-grained pipelining in the particle generation and weight computation steps respectively. Resampling is pipelined with the particle generation step. The problem with overlapping of these two steps is in the fact that duration of the existing resampling methods is not deterministic. For example, in the SR algorithm (Pseudocode 1) there is a *while* loop which number of recursion is dependent on the distribution of particle weights. The new algorithms suitable for pipelining the resampling with the next particle generation step are proposed and described in following chapters. In Figure 3.2, the systematic resampling algorithm is considered. The SR algorithm takes $2M - 1$ clock cycles for the worst case. It means that after the first M cycles at least one particle will be resampled. Since these particles are used in the particle generation step, the next SIRF recursion can start after M clock cycles from the beginning of the resampling step. Hence, the minimum execution time of non-distributed SIRF is $(2M + L)T_{clk}$ [13] where L is the sum of all the pipelining latencies of the SIRF blocks.

Purpose: Obtain the estimates of position and velocity $\hat{x}_n$,$\hat{V}_{x_n}$,$\hat{y}_n$,$\hat{V}_{y_n}$.
Input: The observation z_n and the particles $\{\mathbf{x}_{n-1}^{(m)}\}_{m=1}^{M}$.
Initial setup: $W_n = 0$, $\hat{x}_n = 0$, $\hat{V}_{x_n} = 0$, $\hat{y}_n = 0$, $\hat{V}_{y_n} = 0$, $k = 1$
Method:

for $m = 1 : M$
  for $l = 1 : r^{(m)}$

1.*Sampling step.*

$$x_n^{(k)} = x_{n-1}^{(m)} + V_{x_{n-1}}^{(m)} + u_{x_n}^{(m)}$$
$$V_{x_n}^{(k)} = V_{x_{n-1}}^{(m)} + u_{x_n}^{(m)}$$
$$y_n^{(k)} = y_{n-1}^{(m)} + V_{y_{n-1}}^{(m)} + u_{y_n}^{(m)}$$
$$V_{y_n}^{(k)} = V_{y_{n-1}}^{(m)} + u_{y_n}^{(m)}$$

2. *Importance step*

$$w_n^{*(k)} = w_{n-1}^{*(k)} e^{-(2\pi\sigma_v^2)^{-1}(z_n - atan\frac{y_n^{(k)}}{x_n^{(k)}})^2},$$
$$W_n = W_n + w_n^{*(k)}$$

3. *Output calculation*

$$\hat{x}_n = \hat{x}_n + w_n^{*(k)} x_n^{(k)}$$
$$\hat{V}_{x_n} = \hat{V}_{x_n} + w_n^{*(k)} V_{x_n}^{(k)}$$
$$\hat{y}_n = \hat{y}_n + w_n^{*(k)} y_n^{(k)}$$
$$\hat{V}_{y_n} = \hat{V}_{y_n} + w_n^{*(k)} V_{y_n}^{(k)}$$

$k = k + 1$

  end
end

4. *Resample* particles using, for instance, the SR algorithm: (r)=SR(w_n^*, M, M, W_n)

Output scaling
$\hat{x}_n = \hat{x}_n / W_n$
$\hat{V}_{x_n} = \hat{V}_{x_n} / W_n$
$\hat{y}_n = \hat{y}_n / W_n$
$\hat{V}_{y_n} = \hat{V}_{y_n} / W_n$

Pseudocode 6 SIRF for bearings-only tracking with operation overlapping.

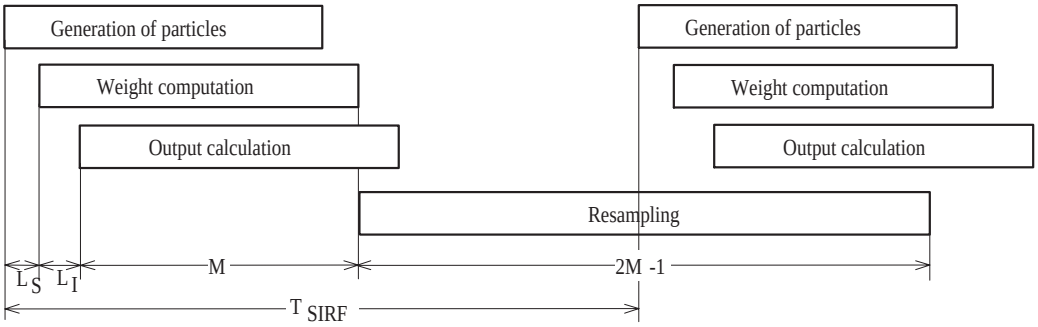


Figure 3.2: Timing diagram for SIRF.

3.3.2 Parallel architecture

The distributed architecture for SIRFs is shown in Figure 3.3. It consists of processing elements (PEs) and a central unit (CU). Since there are no data dependencies during particle generation and computation of the weights, these steps can be easily parallelized and pipelined. This segment of particle filtering is a data parallel single instruction multiple data (SIMD) algorithm [29, 39]. As such, particle generation and weight computation for the M particles can be partitioned in K PEs, where $1 \leq K \leq M$. Each PE performs the same operations in time on different particles and each PE is responsible for processing $N = M/K$ particles where both K and N are integers. The CU carries out partial or full resampling and particle routing as well as overall control. Full resampling means that the overall resampling procedure is performed by one logic unit. In the following chapters, we will show that resampling can be distributed to PEs and that the CU is then responsible only for a small portion of resampling.

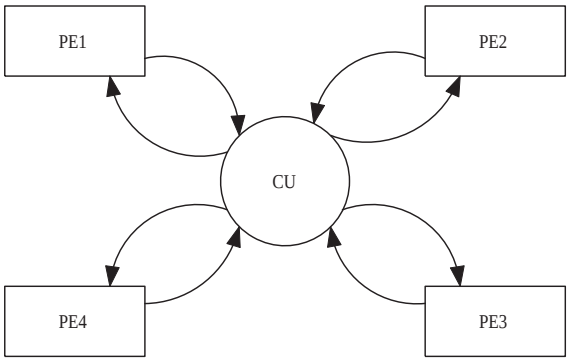


Figure 3.3: Architecture of the distributed particle filter with a CU and four PEs.

In parallel implementation, we distinguish three operations that carry out the resampling task:

1. Computation – involves the bare resampling procedure whose result is an array of indexes which show the replicated particles and their addresses.
2. Communication – represents exchanging of particles among the PEs based on the resampling results. We refer to it as particle routing. Particle routing defines the protocol and the network architecture for exchanging particles and it is the main focus of Chapter 5.

3. Scheduling – includes (a) determining of which particles in the PEs are routed and which are stored locally, (b) placing of particles in the destination PEs, and (c) addressing used for indexes. We refer to it also as particle allocation.

When K PEs are used, the minimum execution time is $(2M/K + L)T_{clk}$. Here, we consider that the pipelining latency L is the same as in the non-parallel implementation and that the processing time is reduced K times. The goal of parallel implementation is to develop algorithms and architectures that can reach the minimum execution time. Our strategy towards achieving the minimum execution time is to allow for deterministic communication during particle routing. Then, we can overlap the particle routing and the next sampling step to allow for pipelining in hardware of their operations, so that the particle routing will not increase the execution time of the SIRF.

In Figure 3.4, the speedup versus the number of PEs for different M is presented for the case of spatial implementation of the SIRF. The platform is FPGA, although it can be ASIC as well since it allows for spatial implementation. The curve saturates faster when M is small, because M/K becomes closer to the value of the constant latency L. So, there is no significant gain in increasing the level of parallelism when M/K becomes close to L. Of course, fully parallel implementation (when $K = M$) is not practical because it does not take advantage of pipelining, it causes complex communication protocols, and there is no gain in speed-up because $M/K = 1$ is much smaller than L. In the simulations, we assumed that T_{clk} remained the same as we increased the level of parallelism.

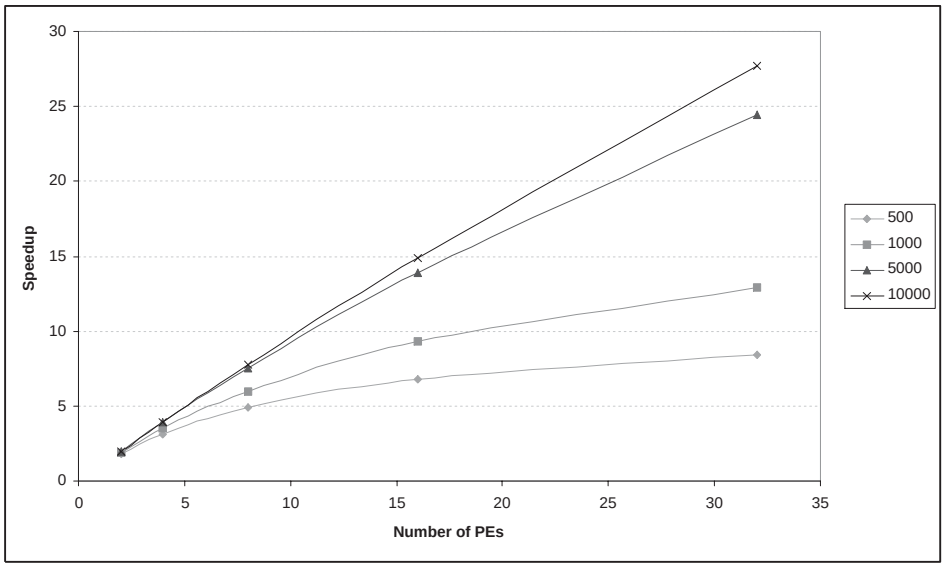


Figure 3.4: Speedup versus the number of PEs of the distributed SIRF for $M = 500$, 1000, 5000, and 10000. Spatial implementation of the SIRF is assumed.

Next, we show why the communication pattern is non-deterministic and the connections among the PEs are changed after each sampling period. Let the number of particles that PE_k produces after resampling be $N^{(k)}$ for $k = 1, ..., K$, $0 \leq N^{(k)} \leq M$ and $\Sigma_{k=1}^{K} N^{(k)} = M$. It is important to note that $N^{(k)}$ is a random number which depends on the overall distribution of the weights. The PEs with $N^{(k)} > N$ have surplus of particles and they need to exchange particles with the PEs with shortage of particles for which $N^{(k)} < N$. The number $N^{(k)}$

changes after each sampling period so that it is necessary to connect different PEs in order to perform particle routing. The number of particles that have to be exchanged among the PEs is $N_M = \Sigma_{k:N^{(k)}>N}^{K}(N^{(k)} - N) = \Sigma_{k:N^{(k)}<N}^{K}(N - N^{(k)})$.

The mean and maximum number of particles that are exchanged through the network for the SIRF applied to the bearings-only tracking problem is shown in Figure 3.5. In the figure, the curve "1024-mean" shows the average number of particles and the curve "1024-max" the maximum number of particles that are exchanged through the interconnection network for all the PEs when $M = 1024$. In the case when PEs are connected to the CU using a single bus, the execution time corresponds to the worst case of the number of particles exchanged through the network. This means that in most cases, the execution time would be over-specified.

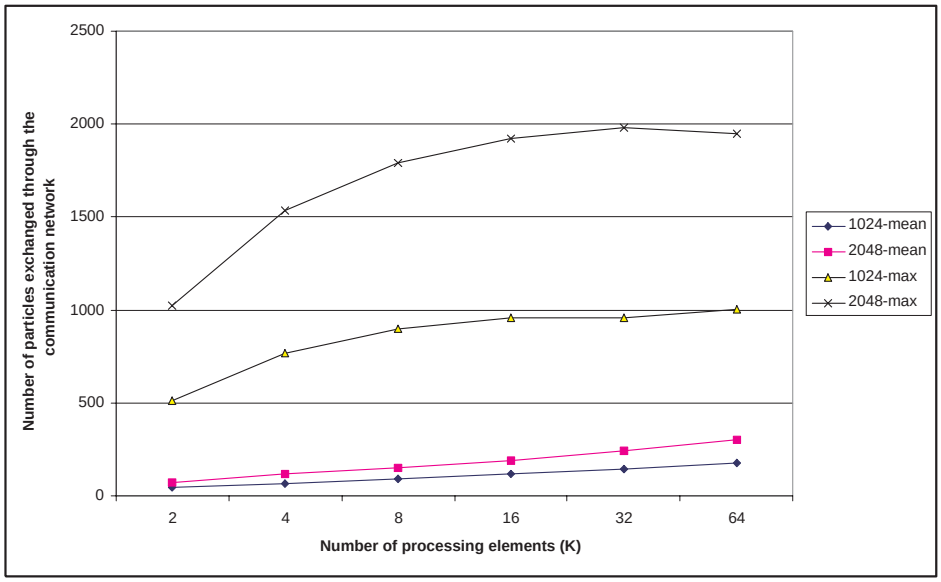


Figure 3.5: Average and maximum number of particles exchanged over the network for $M = \{1024, 2048\}$ particles.

3.3.3 Memory schemes for resampling

Proper particle allocation can reduce the number of memory accesses and the size of the memory for storing particles. The allocation is performed through index addressing, and its execution can be overlapped in time with the particle generation step. In Figure 3.6, three different outputs of resampling for the particles with weights from the example shown in Figure 2.1 are considered. In Figure 3.6(a), the indexes represent positions of the replicated particles. For example, $i^{(2)} = 1$ means that particle 1 replaces particle 2. Particle allocation is easily overlapped with particle generation using $\widetilde{x}^{(m)} = x^{(i^{(m)})}$ for $m = 1, ...M$, where $\{\widetilde{x}^{(m)}\}_{m=1}^{M}$, is the set of resampled particles. The randomness of the resampling output makes it difficult to realize in place storage so that additional temporary memory for storing resampled particles $\widetilde{x}^{(m)}$ is necessary. In Figure 3.6(a), particle 1 is replicated twice and occupies the locations of particles 1 and 2. Particle 2 is replicated once and must be stored in the memory of $\widetilde{x}^{(m)}$ or it would be rewritten. We refer to this method as *particle allocation with index addressing*.

In Figure 3.6(b), the indexes represent the number of times each particle is replicated. For example, $r^{(1)} = 2$ means that the first particle is replicated twice. We refer to this method as *particle allocation with replication factors*. This method still requires additional memory for particles and memory for storing indexes.

The additional memory for storing the particles $\widetilde{x}^{(m)}$ is not necessary if the particles are replicated to the positions of the discarded particles. We call this method *particle allocation with arranged indexes* of positions and replication factors (Figure 3.6(c)). Here, the addresses of both replicated particles and discarded particles as well as the number of times they are replicated (replication factor) are stored. The indexes are arranged in a way that the replicated particles are placed in the upper and the discarded particles in the lower part of the index memory. In Figure 3.6(c), the replicated particles take the addresses $1 - 4$ and the discarded particle is on the address 5. When one knows in advance the addresses of the discarded particles, there is no need for additional memory for storing the resampled particles $\widetilde{x}^{(m)}$, because the new particles are placed on the addresses occupied by the particles that are discarded. It is useful for particle filters applied to multi-dimensional models since it avoids need for excessive memory for storing temporary particles.

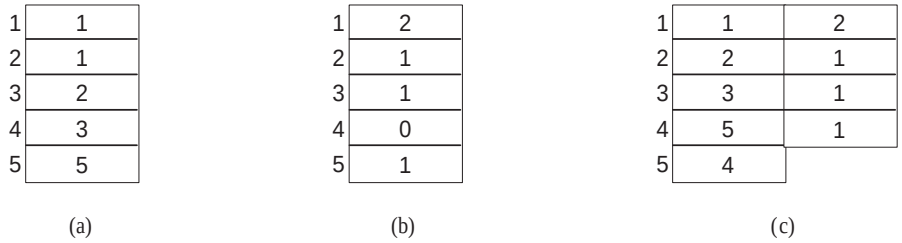


Figure 3.6: Types of memory usages: (a) indexes are positions of the replicated particles, (b) indexes are replication factors, (c) indexes are arranged positions and replication factors.

In SR, particles can be allocated during the resampling procedure so that there is no need for index array. However, the additional particle memory $\widetilde{x}^{(m)}$ is used and the number of memory access is doubled because particles have to be stored to and then read from that memory during the next sampling step.

Memory access is very often bottleneck in the signal processing applications. Chaining of functional units and their concurrent operations notably reduce memory access. Chaining allows for accessing intermediate results through the registers instead of through the memory. The following savings are achieved for the loop fusion shown in Pseudocode 6:

- Weigh computation: $2M$ readings from the memory are avoided because particles $x^{(m)}$ and $y^{(m)}$ are used from the registers.
- Since normalization is not used, M reading from and M writing to the weight memory are avoided.
- Particle allocation step: Since particles are not allocated during resampling step but particle allocation is pipelined with the particle generation step, there is no need for storing and then reading resampled particles from the memory. This saves $4M$ writing and $4M$ reading operations.

The final result of this analysis is that by exploiting concurrency of operations memory access can be significantly reduced ($7M$ readings and $5M$ writings are saved). It allows for increasing the sampling rate and reducing power consumption.

Chapter 4

Algorithms and architectures for particle filters

In this chapter new resampling algorithms that allow for faster execution and/or for reducing memory requirements are described. Then, the architectures that support some of these algorithms are presented. Changes in the resampling algorithm necessary for fixed point-implementation are explained as well. Finally, results of the implementation of the SIRF on the state-of-the-art FPGA platform are given.

4.1 New resampling algorithms

The main goals of this section are development of resampling methods that allow for increased speeds of SIRFs, that require less memory, that achieve fixed timings regardless of the statistics of the particles, and that are computationally less intensive. Development of such algorithms is critical for practical implementations. The performance of the algorithms is analyzed when they are executed on a DSP and specially designed hardware. We investigate sequential resampling algorithms and analyze their computational complexity metrics including the number of operations as well as the class and type of operation by performing behavioral profiling [43].

The main feature of the random resampling algorithm, referred to as residual-systematic resampling (RSR) and described in Section 4.1.1, is to perform resampling in fixed time that does not depend on the number of particles at the output of the resampling procedure. The deterministic algorithms, discussed in Section 4.1.2, are threshold based algorithms, where particles with moderate weights are not resampled. Thereby significant savings can be achieved in computations and in the number of times the memories are accessed. We show two characteristic types of deterministic algorithms: a low complexity algorithm and an algorithm that allows for overlapping of the resampling operation with the particle generation and weght computation. Both the random and deterministic algorithms reduce the number of operations and the number of memory accesses. Hence, they represent an alternative to the existing algorithms that are used in simulations. The performance and complexity analysis are presented in Section 4.1.3.

4.1.1 Residual-systematic resampling algorithm

We propose a new resampling algorithm which is based on stratified resampling, and we refer to it as residual systematic resampling (RSR) [10]. Similar to RR, RSR calculates the number of times each particle is replicated except that it avoids the second iteration of RR when residual particles need to be resampled. Recall that in RR the number of replications of a specific particle is determined in the first loop by truncating the product of the number of particles and the particle weight (Pseudocode 2). In RSR instead, the updated uniform random number is formed in a different fashion, which allows for only one iteration loop and processing time that is independent of the distribution of the weights at the input. The RSR algorithm for N input and M output (resampled) particles is summarized in Pseudocode 7.

Purpose: Generation of an array of replication factors $\{r^{(m)}\}_{m=1}^{N}$ at time instant n, $n > 0$.
Input: An array of weights $\{w_n^{(m)}\}_{m=1}^{N}$, input and output number of particles, N and M, respectively
Method:

$(r) = RSR(N, M, w)$
$\Delta U^{(0)} \sim \mathcal{U}[0, \frac{1}{M}]$ //Generate random number $\Delta U^{(0)}$
for $m = 1$ **to** N // The main loop
  $r^{(m)} = \lfloor (w_n^{(m)} - \Delta U^{(m-1)}) \cdot M \rfloor + 1$ // Computation of the replication factors
  $\Delta U^{(m)} = \Delta U^{(m-1)} + \frac{r^{(m)}}{M} - w_n^{(m)}$ // updating the random number
end

Pseudocode 7 Residual systematic resampling (RSR) algorithm.

In Pseudocode 7, N is the input and M is the output number of particles of the resampling procedure. Even though it is common to have $M = N$, they can be different in several cases including

1. The SR or RSR are used as second steps of the residual resampling algorithm. Then, $N > M$.

2. If there are large deviations from previous estimates it might be beneficial to change the number of particles. Then M can be both larger and smaller than N.

3. In the case of parallel resampling described in [11] where after resampling each parallel element can have a surplus $N < M$ or shortage $N > M$ of particles.

Figure 4.1 graphically illustrates the RSR method for the case of $N = M = 5$ particles with weights given in the table. The RSR algorithm draws the uniform random number $U^{(0)} = \Delta U^{(0)}$ in the same way as in SR but updates it by $\Delta U^{(m)} = \Delta U^{(m-1)} + \frac{r^{(m)}}{M} - w_n^{(m)}$. In the figure, we display both $U^{(m)} = \Delta U^{(m-1)} + \frac{r^{(m)}}{M}$ and $\Delta U^{(m)} = U^{(m)} - w_n^{(m)}$. Here, the uniform number is updated with reference to the *origin of the currently considered weight*, while in SR it is propagated with reference to the *origin of the coordinate system*. The difference $\Delta U^{(m)}$ between the updated uniform number and the current weight is propagated. Figure

4.1 shows that $r^{(1)} = 2$ and that $\Delta U^{(1)}$ is calculated and then used as the initial uniform random number for particle two. Particle four is discarded because $\Delta U^{(3)} = U^{(4)} > w^{(4)}$, so that $\lfloor (w_n^{(4)} - \Delta U^{(3)}) \cdot M \rfloor = -1$ and $r^{(4)} = 0$. It is important to note that SR and RSR produce identical resampling result.

m	$w^{(m)}$	$r^{(m)}$
1	7/20	2
2	6/20	1
3	2/20	1
4	2/20	0
5	3/20	1

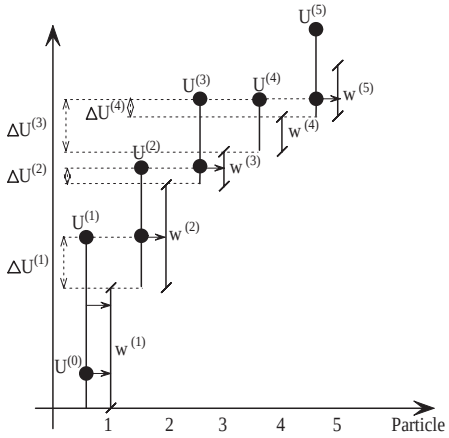


Figure 4.1: Residual-systematic resampling for an example with $M = 5$ particles.

For the RSR method, it is natural to use particle allocation with replication factor and arranged indexes because the RSR produces replication factors. In the particle generation step, the *for* loop with the number of iterations that corresponds to the replication factors is used for each replicated particle. The difference between the SR and the RSR methods is in the way the inner loop in the resampling step for SR and particle generation step for RSR are performed. Since the number of replicated particles is random, the *while* loop in SR has an unspecified number of operations. To allow for an unspecified number of iterations, complicated control structures in hardware are needed [26]. The main advantage of our approach is that the *while* loop of SR is replaced with a *for* loop with known number of iterations.

4.1.2 Deterministic resampling

Overview

In the literature, threshold based resampling algorithms are based on the combination of residual resampling and rejection control and they result in non-deterministic timing and increased complexity [78, 79]. Here, we develop threshold based algorithms whose purpose is to reduce complexity and processing time. We refer to these methods as partial resampling (PR) because only a part of the particles are resampled.

In partial resampling, the particles are grouped in two separate classes: one composed of particles with moderate weights and another, with dominating and negligible weights. The particles with moderate weights are not resampled, whereas the negligible and dominating particles are resampled. It is clear that on average, resampling would be performed much faster because the particles with moderate weights are not resampled. We propose several PR algorithms which differ in the resampling function.

Partial resampling: sub-optimal algorithms

Partial resampling could be seen as a way of a partial correction of the variance of the weights at each time instant. PR methods consist of two steps: one in which the particles are classified as moderate, negligible or dominating and the other in which one determines the number of times each particle is replicated. In the first step of PR, the weight of each particle is compared with a high and a low thresholds, T_h and T_l, respectively where $T_h > 1/M$ and $0 < T_l < T_h$. Let the number of particles with weights greater than T_h and less than T_l be denoted by N_h and N_l, respectively. A sum of the weights of resampled particles is computed as a sum of dominating $W_h = \sum_{m=1}^{N_h} {w_n}^{(m)}$ for $w_n^{(m)} > T_h$ and negligible weights $W_l = \sum_{m=1}^{N_l} {w_n}^{(m)}$ for $w_n^{(m)} < T_l$. We define three different types of resampling with distinct resampling functions $a_n^{(m)}$.

The resampling function of the first partial resampling algorithm (PR1) is shown in Figure 4.2(a) and it corresponds to the stratified resampling case. The number of particles at the input and at the output of the resampling procedure is the same and equal to $N_h + N_l$. The resampling function is given by:

$$ {a_n}^{(m)} = \begin{cases} {w_n}^{(m)}, & \text{for } w_n^{(m)} > T_h \text{ or } w_n^{(m)} < T_l, \\ (1 - W_h - W_l)/(M - N_h - N_l), & \text{otherwise} \end{cases} $$

The second step can be performed using any resampling algorithm. For example, the RSR algorithm can be called using: $(r) = RSR(N_h + N_l, N_h + N_l, w_n^{(m)}/(W_h + W_l))$, where the RSR is performed on the $N_h + N_l$ particles with negligible and dominating weights. The weights have to be normalized before they are processed by the RSR method.

The second partial resampling algorithm (PR2) is shown in Figure 4.2(b). The assumption that is made here is that most of the negligible particles will be discarded after resampling, and consequently, particles with negligible weights are not used in the resampling procedure. Particles with dominating weights replace those with negligible weights with certainty. The resampling function is given as:

$$ {a_n}^{(m)} = \begin{cases} {w_n}^{(m)} + W_l/N_h, & \text{for } w_n^{(m)} > T_h \\ (1 - W_h - W_l)/(M - N_h - N_l), & \text{for } T_l < w_n^{(m)} < T_h \\ 0, & \text{otherwise} \end{cases} $$

The number of times each particle is replicated can be found using $(r) = RSR(N_h, N_h + N_l, (w_n^{(m)} + W_l/N_h)/(W_h + W_l))$ where the weights satisfy the condition $w_n^{(m)} > T_h$. There are only N_h input particles and $N_h + N_l$ particles are produced at the output.

The third partial resampling algorithm (PR3) is shown in Figure 4.2(c). The weights of all the particles above the threshold T_h are scaled with the same number. So, PR3 is a deterministic algorithm whose resampling function is given as

$$ {a_n}^{(m)} = \begin{cases} (N_h + N_l)/(M), & \text{for } w_n^{(m)} > T_h \\ 1/M, & \text{for } T_l < w_n^{(m)} < T_h \\ 0, & \text{otherwise} \end{cases} $$

The number of replications of each dominating particle may be less by one particle than necessary because of the rounding operation. One way of resolving this problem is to assign

that the first $N_t = N_l - \lfloor \frac{N_l}{N_h} \rfloor N_h$ dominating particles are replicated $r = \lfloor \frac{N_l}{N_h} \rfloor + 2$ times, while the rest of $N_h - N_t$ dominating particles are replicated $r = \lfloor \frac{N_l}{N_h} \rfloor + 1$ times. The weights are calculated as $w^{*(m)} = w^{(m)}$ where m represents positions of particles with moderate weights, and as $w^{*(l)} = w^{(m)}/r + W_l/(N_h + N_l)$ where m are positions of particles with dominating weights and l of particles with both dominating and negligible weights.

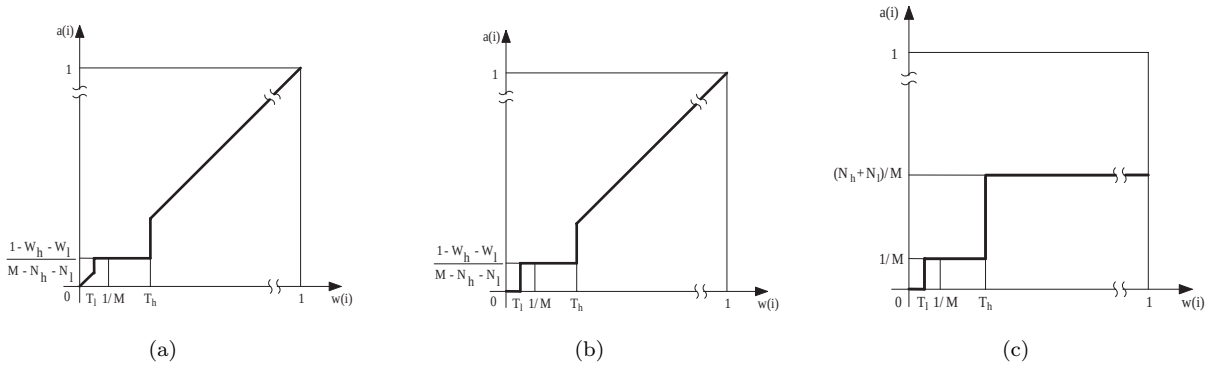
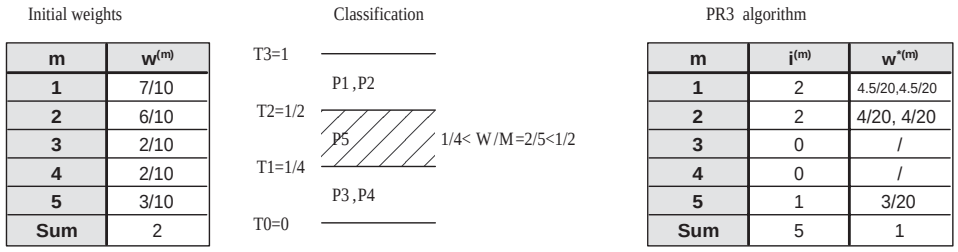


Figure 4.2: Resampling functions for the partial resampling algorithms (a) PR1, (b) PR2 and (c) PR3.

Another way of performing partial resampling is to use a set of thresholds. The idea is to perform initial classification of the particles while the weights are computed and then to carry out the actual resampling together with the particle generation step. So, the resampling consists of two steps as in the PR2 algorithm where classification of the particles is overlapped with the weight computation. We refer to this method as Overlapped Partial Resampling (OPR).

A problem with the classification of the particles is the necessity of knowing the overall sum of non-normalized weights in advance. The problem can be resolved as follows. The particles are partitioned according to their weights. The thresholds for group i are defined as T_{i-1}, T_i for $i = 1, ..., K$ where K is the number of groups, $T_{i-1} < T_i$ and $T_0 = 0$. The selection of thresholds is problem dependent. The thresholds that define the moderate group of particles satisfy $T_{k-1} < W/M < T_k$. The particles that have weights greater than T_k are dominant particles, and the ones with weights less than T_{k-1}, negligible particles.

In Figure 4.3 we provide a simple example of how this works. There are four thresholds (T_0 to T_3) and non-normalized particles are compared with the thresholds and properly grouped. After obtaining the sum of weights W, the second group for which $T_1 < W/M < T_2$, is the of group of particles with moderate weights. The first group contains particles with negligible weights, and the third group is composed of particles with dominating weights. An additional loop is necessary to determine the number of times each of the dominating particles is replicated. However, the complexity of this loop is of order $O(K)$, which is several orders of magnitude lower than the complexity of the second step in the PR1 algorithm ($O(M)$). Because the weights are classified, it is possible to apply similar logic for the second resampling step as in the PR2 and PR3 algorithms. In the figure, the particles P1 and P2 are replicated twice and their weights are calculated using the formulae for weights for the PR3 method.



m	$w^{(m)}$
1	7/10
2	6/10
3	2/10
4	2/10
5	3/10
Sum	2

m	$i^{(m)}$	$w^{*(m)}$
1	2	4.5/20,4.5/20
2	2	4/20, 4/20
3	0	/
4	0	/
5	1	3/20
Sum	5	1

Figure 4.3: OPR method combined with the PR3 method used for final computation of weights and replication factors.

Discussion

In the PR1, PR2 and PR3 algorithms, the first step requires a loop of M iterations for the worst case (of number of computations) with two comparisons per each iteration (classification in three groups). Resampling in the PR1 algorithm is performed on $N_l + N_h$ particles. The worst case for the PR1 algorithm occurs when $N_l + N_h = M$, which means that all the particles must be resampled, thereby implying that there cannot be improvements from an implementation standpoint. The main purpose of the PR2 algorithm is to improve the worst case timing of the PR1 algorithm. Here, only N_h dominating particles are resampled. So, the input number of particles in the resampling procedure is N_h, while the output number of particles is $N_h + N_l$. If the RSR algorithm is used for resampling, then the complexity of the second step is $O(N_h)$.

PR1 and PR2 contain two loops and their timings depend on the weight statistics. As such, they do not have advantages for real-time implementation in comparison with RSR, which has only one loop of M iterations and whose processing time does not depend on the weight statistics. In the PR3 algorithm, there is no stratified resampling. The number of times each dominating particle is replicated is calculated after the first step and it depends on the current distribution of particle weights and of the thresholds. This number is calculated in $O(1)$ time, which means that there is no need for another loop in the second step. Thus, PR3 has simpler operations than the RSR algorithm.

The PR algorithms have the following advantages from the perspective of hardware implementation: (1) the resampling is performed faster on average because it is done on a much smaller number of particles, (2) there is a possibility of overlapping the resampling with the particle generation and weight computation, and (3) if the resampling is used in a parallel implementation [11], the number of exchanged particles among the processing elements is smaller because there are less particles to be replicated and replaced. There are also problems with the three algorithms. When $N_l = 0$ and $N_h = 0$, resampling is not necessary. However, when $N_l = 0$ or $N_h = 0$ but not at same time, the PR algorithms would not perform resampling even though it could be useful.

Application of the OPR algorithm requires a method for fast classification. For hardware and DSP implementation, it is suitable to define thresholds that are a power of two. So, we take that $T_i = 1/2^{K-i}$ for $i = 1, ..., K$ and $T_0 = 0$. The group is determined by the position of the most significant "one" in the fixed point representation of weights. Memory allocation for the groups could be static or dynamic. Static allocation requires K memory banks where the

size of each bank is equal to the number of particles because all the particles could be located in one of the groups. Dynamic allocation is more efficient and it could be implemented using ways similar to the linked lists where the element in a group contains two fields: the field with the address of the particle and the field that points out to the next element on the list. Thus, dynamic allocation requires memory with capacity of $2M$ words. As expected, overlapping increases the resources.

4.1.3 Particle filtering performance and complexity

Performance analysis

The proposed resampling algorithms are applied and their performance is evaluated for the joint detection and estimation problem in communication [22, 35] and for the bearings-only tracking problem [49].

Joint detection and estimation

The experiment considered a Rayleigh fading channel with additive Gaussian noise with a differentially encoded BPSK modulation scheme. The detector was implemented for a channel with normalized Doppler spreads given by $B_d = 0.01$, which corresponds to fast fading. An $AR(3)$ process was used to model the channel. The AR coefficients were obtained from the method suggested in [115]. The proposed detectors were compared with the *clairvoyant* detector, which performs matched filtering and detection assuming that the channel is known exactly by the receiver. The number of particles was $N = 1000$.

In Figure 4.4, the bit error rate (BER) versus signal-to-noise ratio (SNR) is depicted for the PR3 algorithm with different sets of thresholds, i.e., $T_h = \{2M, 5M, 10M\}$ and $T_l = \{1/(2M), 1/(5M), 1/(10M)\}$. In the figure, the PR3 algorithm with the thresholds $2M$ and $1/2M$ is denoted as PR3(2), the one with thresholds $5M$ and $1/5M$ as PR3(5) and so on. The BER for the matched filter (MF) and for the case when the systematic resampling is performed are shown as well. It is observed that the BER is similar for all types of resampling. However, the best results are obtained when the thresholds $2M$ and $1/2M$ were used. Here, the effective number of particles that is used is the largest in comparison with the PR3 algorithm with greater T_h and smaller T_l. This is a logical result, because according to PR3, all the particles are concentrated in the narrower area between the two thresholds producing in this way a larger effective sample size. PR3 with thresholds $2M$ and $1/2M$ slightly outperforms the systematic resampling algorithm which is a bit surprising. The reason for this could be that the particles with moderate weights are not unnecessarily resampled in the PR3 algorithm. The same result is obtained even with different values of Doppler spread.

In Figure 4.5, BER versus SNR is shown for different resampling algorithms: PR2, PR3, OPR, and SR. The thresholds that are used for the PR2 and PR3 are $2M$ and $1/2M$. The OPR uses $K = 24$ groups and thresholds which are power of two. Again, all the results are comparable. The OPR and PR2 algorithms slightly outperform the other algorithms.

Bearings-only tracking

We tested the performance of SIRFs by applying the resampling algorithms to bearings-only tracking [49] with different initial conditions. In the experiment, PR2 and PR3 are used

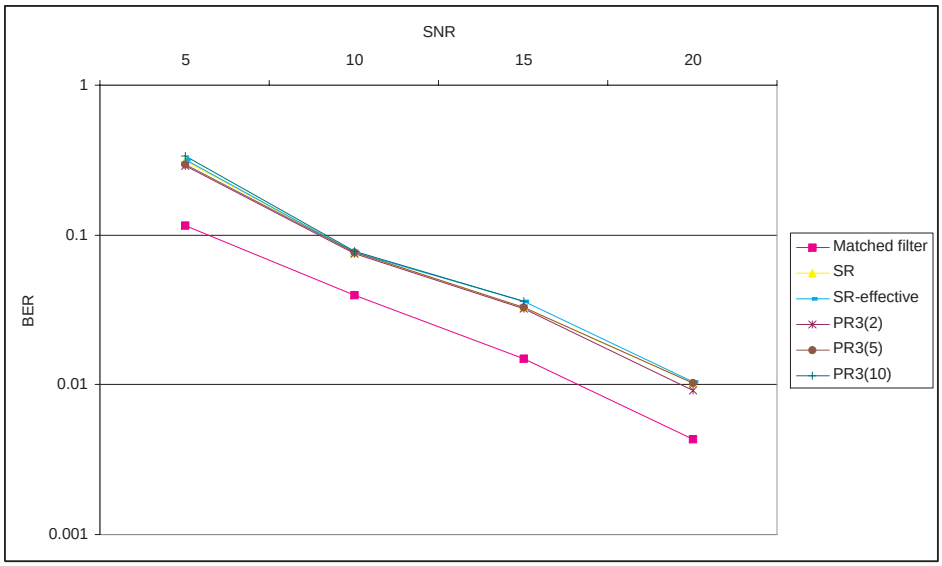


Figure 4.4: Performance of the PR3 algorithm for different threshold values applied to joint detection and estimation problem in wireless communications.

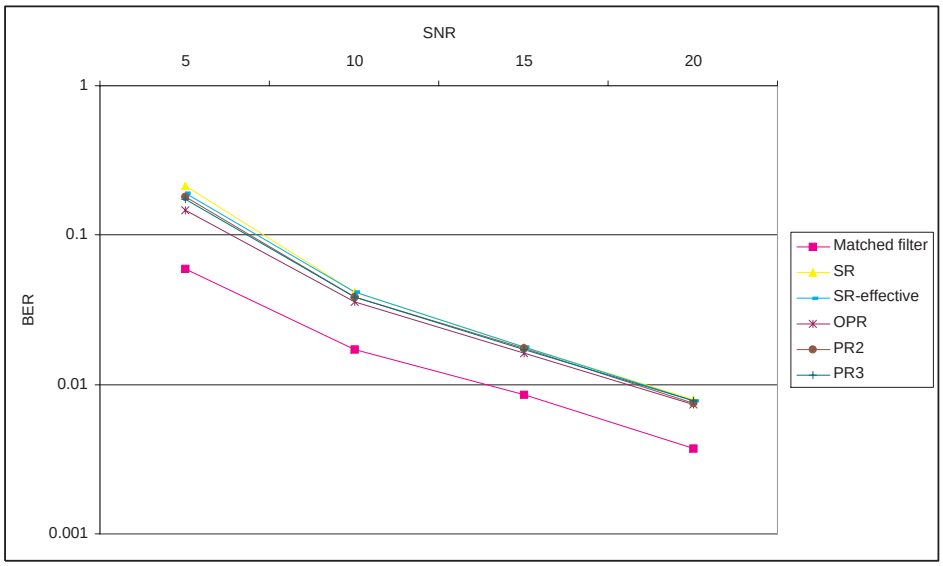


Figure 4.5: Comparison of the PR2, PR3 and OPR algorithms with systematic resampling applied to the joint detection and estimation problem in wireless communications.

with two sets of threshold values, i.e., $T_h = \{2M, 10M\}$ and $T_l = \{1/(2M), 1/(10M)\}$. In Figure 4.6, we show the number of times when the track is lost versus number of particles, for two different pairs of thresholds. We consider that the track is lost if all the particles have zero weights. In the figure, the PR3 algorithm with thresholds $2M$ and $1/2M$ is denoted as PR3(2) and the one with thresholds with thresholds $10M$ and $1/10M$ as PR3(10). The used algorithms are SR, SR performed after every 5-th observation, PR2 and PR3. The resampling algorithms show again similar performances. The best results for PR2 and PR3 are obtained when the thresholds $10M$ and $1/10M$ are used.

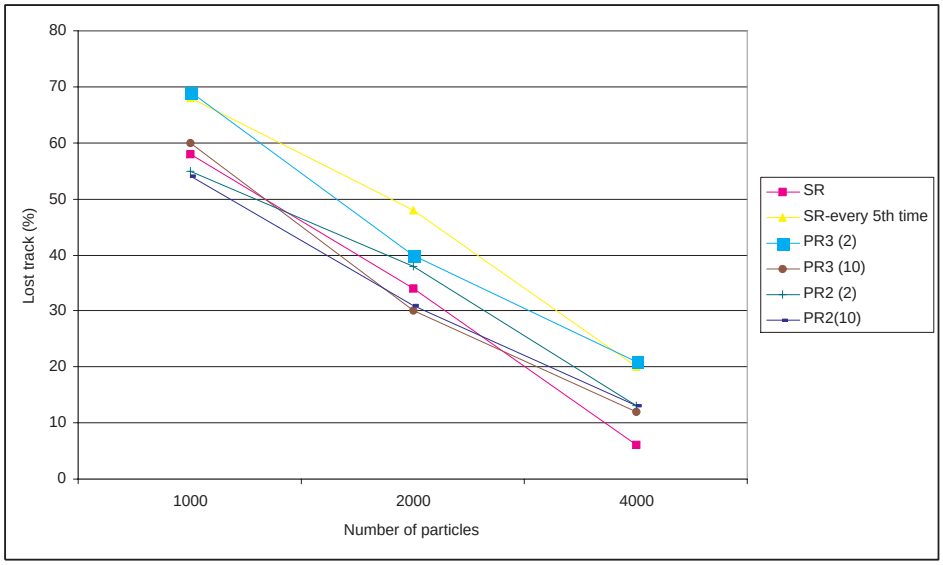


Figure 4.6: Number of times when track is lost for the PR2, PR3 and SR applied to the bearings-only tracking problem.

Complexity analysis

The complexity of the proposed resampling algorithms is evaluated. We consider both computation complexity as well as memory requirements. We also present benefits of the proposed algorithms when concurrency in hardware is exploited.

Computational complexity

In Table 4.1, we provide a comparison of the different resampling algorithms. The results for RR are obtained for the worst case scenario. The complexity of the RR, RSR, and PR algorithms is of $O(N)$, and the complexity of the SR algorithm is of $O(max(N, M))$ where N and M are the input and output numbers of particles of the resampling procedure.

	SR	RR	RSR	PR3
Multiplications	0	N	N	0
Additions	$2M + N$	$6N$	$3N$	$2N$
Comparisons	$N + M$	$3N$	0	$2N$

Table 4.1: Comparison of the number of operations for different resampling algorithms.

When the number of particles at the input of the resampling algorithm is equal to the number of particles at the output, the RR algorithm is by far the most complex. While the number of additions for the SR and RSR algorithms are the same, the RSR algorithm performs M multiplications. Since multiplication is more complex than addition, we can view that the SR is a less complex algorithm. However, when N is a power of two such that the multiplications by N is avoided, the RSR algorithm is the least complex.

The resampling algorithms SR, RSR and PR3 were implemented on the Texas Instruments (TI) floating-point digital signal processor (DSP) TMS320C67xx. Several steps of

profiling brought about five-fold speed-up when the number of resampled particles was 1000. The particle allocation step was not considered. The number of clock cycles per particle was around 18 for RSR and 4.1 for PR3. The SR algorithm does not have fixed timing. The mean duration was 24.125 cycles per particle with standard deviation of 5.17. On the processor TMS320C6711C whose cycle time is 5 ns, the processing of RSR with 1000 particles took $90\mu s$.

Memory requirements

In our analysis, we considered the memory requirement not only for resampling but for the complete SIRF. The memory size of the weights and the memory access during weight computation do not depend on the resampling algorithm. We consider particle allocation without indexes and with index addressing for the SR algorithm and with arranged indexing for RSR, PR2, PR3 and OPR (see Section 3.3.3). For both particle allocation methods, the SR algorithm has to use two memories for storing particles. In Table 4.2 we can see the size of memories for the RSR, PR2, PR3 algorithms. The difference among these methods is only in the size of the index memory. For the RSR algorithm which uses particle allocation with arranged indexes, the index memory has a size of $2M$, where M words are used for storing the addresses of the particles that are replicated or discarded. The other M words represent the replication factors.

The number of resampled particles for the worst case of the PR2 algorithm corresponds to the number of particles in the RSR algorithm. Therefore, their index memories are of the same size. From an implementation standpoint, the most promising algorithm is the PR3 algorithm. It is the simplest one and it requires the smallest size of memory. The replication factor of the dominating particles is the same and of the moderate particles is one. So, the size of the index memory of PR3 is M, and it requires only one additional bit to represent whether a particle is dominant or moderate.

The OPR algorithm needs the largest index memory. When all the SIRF steps are overlapped, it requires different access pattern than the other deterministic algorithms. Due to possible overwriting of indexes that are formed during the weight computation step with the ones that are read during particle generation, it is necessary to use two index memory banks (M_{i1} and M_{i2}). Furthermore, particle generation and weight computation should access these memories alternately. Writing to M_{i1} is performed in the resampling step in one time instance whereas in the next one, the same memory is used by particle generation for reading. The memory bank M_{i2} is used alternately. If we compare the memory requirements of the OPR algorithm with that of the PR3 algorithm, it is clear that OPR requires four times more memory for storing indexes for resampling.

In the SR algorithm, two memories for storing particles with capacities N_sM are necessary and we refer to them as M_{s1} and M_{s2} in Table 4.1.3. The disadvantage of the SR algorithm without indexes is an additional access of the state memories during the resampling step. In the SIRF which uses the SR algorithm with indexes, the particle allocation step can be pipelined with the sample step so that states from the state memories are read and written only once during the sample step. Then, the particles are read from memory M_{s1} and written to memory M_{s2} in one sample step and then read from M_{s2} and written to M_{s1} in the following sample step. This is an alternating process and switching of memories requires additional logic. The algorithms that require the alternating process are labeled with *alt.* in Table 4.1.3.

In Table 4.1.3, we can see that the particles are written and read from the state memory at the same time. The advantage of the RSR, PR and OPR algorithm over the SR algorithm is that they use only one state memory. This advantage is paid by using additional logic for handling an index array.

	SR without indexes	SR with indexes	RSR	PR2	PR3	OPR
States	$2N_sM$	$2N_sM$	N_sM	N_sM	N_sM	N_sM
Weights	M	M	M	M	M	M
Indexes	0	M	$2M$	$2M$	M	$4M$

Table 4.2: Memory capacity for different resampling algorithms.

		SR without indexes	SR with indexes	RSR, PR2, PR3	PPR
Sample	Read	States (M_{s1})	States (M_{s1}, M_{s2} alt.), Indexes	States (M_{s1}), Indexes	States (M_{s1}), Indexes (M_{i1}, M_{i2} alt.)
	Write	States (M_{s2})	States (M_{s2}, M_{s1} alt.)	States (M_{s1})	States (M_{s1})
Importance	Read	-	-	-	-
	Write	Weights	Weights	Weights	Weights
Resample	Read	Weights, States (M_{s2})	Weights	Weights	Weights
	Write	States (M_{s1})	Indexes	Indexes	Indexes(M_{i1}, M_{i2} alt.)

Table 4.3: Pattern of the memory access for different resampling algorithms.

SIRF speed improvements

The SIRF sampling frequency can be increased in hardware by exploiting temporal concurrency. Since there are no data dependencies among the particles in the particle generation and weight computation, the operations of these two steps can be overlapped. Furthermore, the number of memory accesses is reduced because during weight computation, the values of the states do not need to be read from the memory since they are already in the registers.

In order to achieve the higher speed, particle filter algorithms are modified in a way that they do not use implicit normalization step. Normalization in the RSR method can be approached in the same way as in Section 3.2.3. Normalization in the PR methods could be avoided by including information about the sum of weights W_n in the thresholds by using $T_{hn} = T_h W_n$ and $T_{ln} = T_l W_n$.

The timing operations for a hardware implementation where all the blocks are fine-grain pipelined are shown in Figure 4.7(a). Here, the particle generation and weight calculation operations are overlapped in time and normalization is avoided. The symbol L is the constant hardware latency defined by the depth of pipelining in the particle generation and weight computation, T_{clk} is the clock period, M is the number of particles, and T is the minimum processing time of the any of the basic SIRF operations. The SR is not suitable for hardware implementations where fixed and minimal timings are required, because its processing time depends on the weight distribution and it is longer than MT_{clk}. So, in order to have resampling operation performed in M clock cycles, RSR or PR3 algorithms with particle allocation with arranged indexes must be used. The minimum SIRF sampling period that can be achieved is $(2M + L)T_{clk}$.

OPR in combination with the PR3 algorithm allows for higher sampling frequencies. In the OPR, the classification of the particles is overlapped with the weight calculation as shown in

Figure 4.7(b). The symbol L_R is the constant latency of the part of the OPR algorithm that determines which group contains moderate, and which negligible and dominating particles. The latency L_R is proportional to the number of ORP groups. The speed of the SIRF can almost be increased twice if we consider pipelined hardware implementation. In Figure 4.7(b), it is obvious that the SIRF processing time is reduced to to $(M + L + L_R)T_{clk}$.

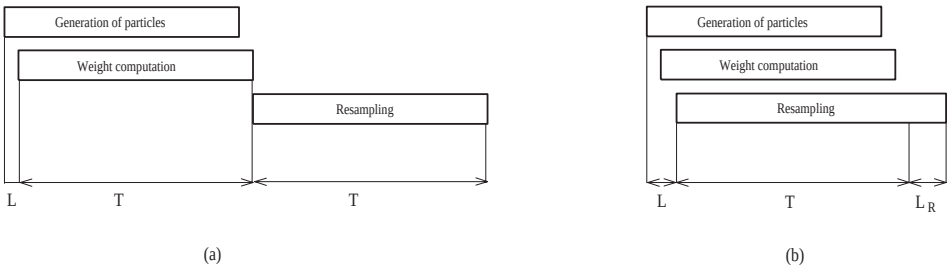


Figure 4.7: The timing of the SIRF with the (a) RSR or PR methods and (b) with the OPR method.

Additional speed improvements of the SIRF with RSR

Since the RSR step has the same duration as the sample-importance steps ($\sim M$) and they use different hardware resources, it is possible to run concurrently two SIRFs on the same hardware at the same time. In Figure 4.8, coarse timing diagram of SIRF operations is presented where operations of SIRF 2 are collored gray. The input observation at time instant n is denoted as z_n. Filters work in a way that resampling step of one SIRF is executed concurrently with the sample and importance steps of the other filter. For example, SIRF 1 performs particle generation and weight computation at time instants n, $n+2$ and so on, and SIRF 2 at time instants $n+1$, $n+3$ and so on. In Figure 4.8, resampling of SIRF 1 that corresponds to processing of the observation z_n is performed concurrently with the sample and importance steps of SIRF 2 which processes the observation z_{n+1} .

Executing two SIRFs which have the same model structure but use some different parameter is also possible. This feature is desirable in some applications in which the estimates of two SIRFs can be compared for different parameter setups. Running two SIRFs concurrently is achieved at the expense of doubling memory requirements and increasing controller complexity but without increasing logic area requirements of the data-path.

Concurrent execution of two SIRFs that use the SR algorithm is not possible with the same high speed requirements. As shown in Figure 3.2, duration of SR is $2M-1$ so that there are no empty time slots for another concurrent particle filter that runs at maximum speed.

Final remarks

We summarize the impact of the proposed resampling algorithms on the SIRF speed and memory requirements.

1. The RSR is an improved residual resampling algorithm with higher speed and fixed processing time. As such, besides for hardware implementations, it is a better algorithm for resampling that is executed on standard computers.

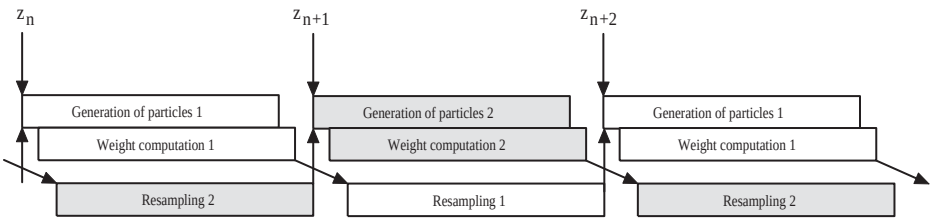


Figure 4.8: The timing of the two SIRFs which operations are overlapped and which share the same hardware resources.

2. Memory requirements are reduced. The number of memory access and the size of the memory are reduced when RSR or any of PR algorithms are used for multidimensional state space models. These methods can be appropriate for both hardware and DSP applications where the available memory is limited. When the state-space model is one-dimensional then there is no purpose of adding an index memory and introducing a more complex control. In this case, the SR algorithm is recommended.

3. In hardware implementation and with the use of temporal concurrency, the SIRF sampling frequency can be considerably improved. The best results are achieved for the OPR algorithm at the expense of hardware resources.

4. The average amount of operations is reduced. This is true for PR1, PR2 and PR3 since they perform resampling on a smaller number of particles. This is desirable in PC simulations and some DSP applications.

4.2 Residual resampling in fixed-point arithemetics

A low-complexity RR scheme for particle filters is presented in this section. The proposed scheme uses a simple "particle-tagging" method to compensate for a possible error that can be caused by finite precision quantization in the resampling step of particle filtering. The scheme guarantees that the number of particles after resampling is always equal to the number of particles before resampling. The resulting scheme is suitable for high-speed physical realization when the number of particles is a power of two.

4.2.1 Proposed resampling scheme

For correct functioning of resampling, it is necessary that the sum of all weights after normalization is equal to one. However, this condition is not satisfied in VLSI implementation due to the finite precision effect. In RR, the number of replicated particles is calculated first by truncation or rounding of the product $w^{(m)}M$. In the case of truncation, the number of particles produced after this step is in general less than M. Then, it is necessary to process the residues in order to compensate for the number of particles (see Pseudocode 2). Here, a RR algorithm suitable for fixed-point implementation is considered. Residues are processed using a memory-addressing scheme and a tagging method, and this procedure guarantees the correct number of particles after resampling.

In hardware implementation with fixed-point number representation, the weights are quantized with K bits (excluding the sign bit), where $K = \log_2(M)$. A naive approach would quantize the value of the weight by simple truncation. Then the integer representation of the K bits corresponds to the number of times the particle should be replicated. The simple truncation may result in a total number of replicated particles less than M. This is illustrated in Table 4.4, where $M = 4$ and $K = 2$. The second column represents the decimal values of the particle weights, and the third column their binary representation with two bits. The fourth column provides the replication factor, which is a decimal equivalent value of the K bits indicated in bold. According to the table, particle $x^{(1)}$ will be replicated twice, particle $x^{(2)}$ will be replicated once, and particles $x^{(3)}$ and $x^{(4)}$ will be eliminated. Because of the quantization, the sum of the resampled particles is not equal to four. In general, $R = \sum r^{(m)}$ may not be equal to M.

Particles	Weights	Quantized Weights	Replication Factor ($r^{(m)}$)
$x^{(1)}$	0.748	0.**10**	2
$x^{(2)}$	0.250	0.**01**	1
$x^{(3)}$	0.001	0.**00**	0
$x^{(4)}$	0.001	0.**00**	0

Table 4.4: Resampling with quantization by simple truncation.

For solving this problem, one might consider using a conventional rounding method. When the rounding is employed, however, it is possible that the sum of all replicated particles

Three Bits	Rounding Scheme	Result	Tag Status
000	Truncate	**0**	none
001	Truncate	**0**	Tag3
010	Truncate	**0**	Tag2
011	Round	**1**	none
100	Truncate	**1**	none
101	Truncate	**1**	Tag3
110	Truncate	**1**	Tag2
111	Truncate	**1**	Tag1

Table 4.5: Rounding/truncation scheme and tags.

be larger than M. It should be noted that both simple truncation and rounding methods complicate the hardware. For example, when the sum of all replicated particles is less than M, the hardware must decide which particles to additionally replicate so that the total number of particles is M. On the other hand, if the sum of all replicated particles is larger than M, the hardware must somehow choose some of the already replicated particles for removal. These two scenarios require additional iteration (i.e., scanning of all the weights for reevaluation) and selection of particles for additional replication or removal. Therefore, a modification is necessary for efficient hardware implementation.

In order to resolve the problem of not having M resampled particles, we propose that all the weights are quantized with two additional bits such that $K = \log_2(M) + 2$, excluding the sign bit. The two additional bits are used to create tags. Then a final quantization is performed, which consists of rounding and truncation, as shown in Table 4.5. The entries of the first column are the last three bits of the binary representation of the weight, the second column the applied rounding scheme, the third column the resulting least significant bit, and the last column the Tag status. Notice that the bit pattern **111** is not rounded, but tagged, since an adder is needed to incorporate carry propagation to the most significant bit. However, the bit pattern **011** is rounded where simple bit reversal is sufficient. The difference between Tag1 and Tag2 is to indicate that Tag1 has higher priority for replication. For simplicity in the implementation, however, these do not have to be distinguished especially when the value of M is large (i.e., more than 64). The particles with Tag3 are used only when the total number of replicated particles is less than M.

When $R = \sum r^{(m)} < M$, the tagged particles may be replicated once more. Note that, $R + T_1 + T_2 + T_3 > M$ where T_1, T_2, and T_3, are the sums of particles with tags Tag1, Tag2, and Tag3, respectively. R has priority over T_1, T_2, or T_3, and therefore the tagged particles are over-written in case $R = M$. A reason that tagging is used instead of rounding for all cases is to avoid a situation where R exceeds M such that it may be possible to exclude particles that are very important but located physically toward the end of the memory location (this will be described in the next section). The proposed scheme is illustrated in Table 4.6 on the same example as in Table 4.4.

The modified quantization scheme ensures that the sum of all replication factors is closer to M than with the resampling based on quantization by simple truncation. In addition, only a single iteration (i.e., scanning of all the weights) is necessary, which saves computation time

Particles	Weights	Quantized Weights	Round/ Truncate	Replication Factor ($r^{(m)}$)
$x^{(1)}$	0.748	0.1011	0.**11**	3
$x^{(2)}$	0.250	0.0011	0.**01**	1
$x^{(3)}$	0.001	0.0000	0.**00**	0
$x^{(4)}$	0.001	0.0000	0.**00**	0

Table 4.6: Resampling with the proposed scheme.

(i.e., cuts processing time by half) and minimizes hardware complexity.

There are three special cases, which must be considered carefully. First, there is a situation where one particle has a weight equal to 1.0 and the rest are all zero. Without any special modification, the scheme will get all the weights to zero since it only considers the K least significant bits. To avoid this problem, the weight of 1.0 in decimal representation is represented as $1 - 2^{-(K+1)}$. Then the tagging method will guarantee that the total number of replicated particles is M. Second, it is also possible that all the weights are zero. This may happen when the estimate of the state being estimated diverges and it is not possible to accurately compute the weights with finite precision. This situation can be detected in the weight calculation stage prior to resampling, and thus no resampling is performed. The third special case occurs when the number of particles in the resampling is greater than M due to rounding. The algorithm then produces the correct number of particles but the number of some particles is not proportional to the respective weights.

In order to illustrate the performance of the proposed resampling scheme, M weights are randomly generated and their weight distributions after resampling are compared with that of full precision resampling. Figure 4.9 illustrates the replication factors both with and without tagging for $M = 128$. The results are obtained by simulating the logic structure of the scheme, where a random set of 128 normalized weights is used to implement the resampling. As shown in the figure, the scheme with tagging is very close to that of full precision resampling, which is plotted with a solid line. The sum of the replicated particles illustrated by the resampling without tagging is less than M. Additional simulation results are published in [55].

4.2.2 A logic structure

A logic structure of the proposed scheme is shown in Figure 4.10. The particles and their weights are stored in a memory and they are provided prior to resampling. The same address is used to access the particles and weights. Each weight is read and decoded. An integer value of K bits representing r is loaded to the down-counter for particle replication. While a particle is being replicated in the other memory starting from the lowest address, the same particle, if it is tagged (Tag1 or Tag2), is also written to the same memory but starting from the highest address. This is to ensure that when we have enough replicated particles, the tagged particles can be over written and discarded. Thus, the total number of replicated particles is always M. However, it is still possible that $R + T_1 + T_2 < M$. This problem is resolved by having a small memory for storing tagged particles (Tag3) and these particles are copied to the resampled

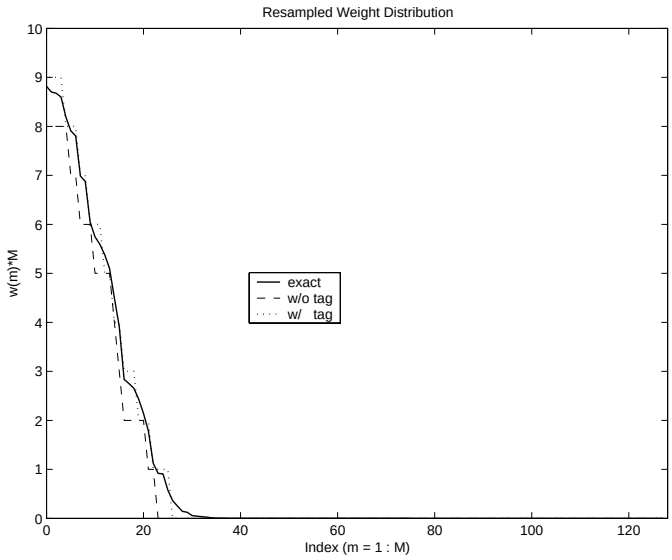


Figure 4.9: Comparison of exact resampling and resampling with and without tagging. The particles are ordered according to their weights, where the first particle has the largest weight.

memory. This condition is checked by adding addresses of memory locating the last insertion of replicated particles and tagged particles. If the sum of these two addresses is less than M, the tagged particles (Tag3) are inserted from the starting address of particles right after R. Although we have assumed that the number of particles is a power of two, the scheme can be extended to handle arbitrary number of particles. Moreover, parallelization is possible for high-throughput applications.

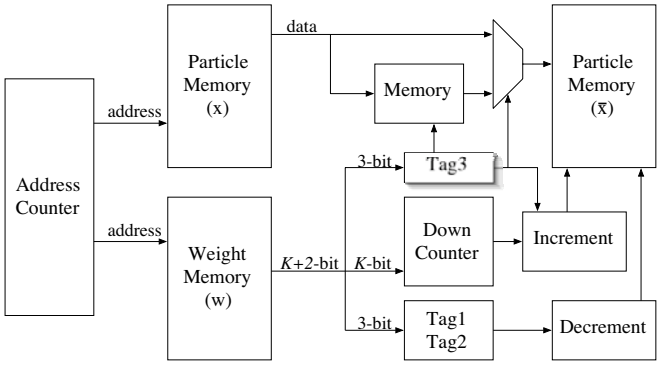


Figure 4.10: A logic diagram that illustrates the structure of the proposed resampling scheme. It is assumed that the particles $x^{(m)}$ and their weights $w^{(m)}$ are provided prior to resampling. The resampled particles $\tilde{x}^{(m)}$ are stored in a separate memory.

4.3 Architectures for SIRFs based on the RSR algorithm

In this section architectures for SIRFs are presented. The RSR algorithm is modified so that it is suitable for hardware implementation. Then, the architecture for RSR and memory related operations of the sample step are analyzed. These architectures remain unchanged irrespective of the model to which the SIRF is applied. Next, architectures for the sample (state-space model based operations) and importance steps are presented. Based on these architectures, the SIRF for the bearings-only tracking model is implemented in FPGA on the Xilinx Virtex II Pro platform. The resource utilization and latency of the design are presented.

4.3.1 Adjusting the RSR algorithm for hardware implementation

In this section the following changes to the algorithm shown in Pseudocode 7 are presented:

1. Resampling is performed using non-normalized weights,
2. The number of operations is reduced by recognizing the operations that repeat in Pseudocode 7, and
3. Particle allocation with arranged indexes is incorporated.

The modified RSR algoritm is shown in Pseudocode 8.

Non-normalized weights can be incorporated by replacing M in the lines 1, 3 and 4 in Pseudocode 7 with M/W_n , where W_n is the sum of all the weights at time instant n. We see that in the line 3 and line 4 of Pseudocode 7, there is one multiplication and one division of the weights and the factor M/W_n. In Pseudocode 8, the code is modified in a way that there is only one multiplication inside the loop. It is done by multiplying the expression in the line 4 of Pseudocode 7 by M/W_n so that there is the same factor $w_n^{*(m)} \cdot M/W_n$ in both expressions in lines 3 and 4 of Pseudocode 7. In order to avoid multiplying ΔU with the factor M/W_n, we modified the way in which the random number U is generated. It is generated in the range of $(0, 1]$ instead in the range of $(0, W_n/M]$ (line 1 of Pseudocode 8,). Thus, there is one multiplication inside the loop and one division before the loop in Pseudocode 8. When the weights are normalized, there is no need for division.

In the second part of Pseudocode 8, indexes of the replicated and discarded particles are generated for the particle allocation method with the arranged indexes (see Section 3.3.3). At the beginning, the index of the replicated particles is set to zero and of the discarded to $M - 1$ as shown in line 3. When the replication factor is greater than zero, the address of the replicated particles are stored $i_r^{(ind_r)} = m$ and the index is incremented so the it points to the address where the next replicated particle will be stored (line 9). Similarly, the address of the discarded particles and their index are updated when the replication factor is equal zero as shown in line 11.

Purpose: Generation of arrays of replicated and discarded particle indexes i_r and i_d, and the array of the replication factors r at time instant n, $n > 0$.

Input: An array of weights $\{w_n^{*(m)}\}_{m=1}^{M}$, the number of particles M and the sum of weights W_n.

Method:

$(i_r, i_d, r) = RSR(M, W_n, w_n)$

1. $U \sim \mathcal{U}[0,1]$ // Generating a random number U.
2. $K = M/W_n$ // Calculating the constant.
3. $ind_r = 0$, $ind_d = M - 1$ // Set the initial values for indexes.
4. **for** $m = 1$ **to** M // Main resampling loop.
5. $temp = w_n^{*(m)} \cdot K - U$ // Temporary variable.
6. $r^{(ind_r)} = \lceil temp \rceil$ // Storing the replication factor.
7. $U = temp - r^{(ind_r)}$ // Updating the uniform random number.
8. **if** $r^{(ind_r)} > 0$ // Particle allocation
9. $i_r^{(ind_r)} = m$, $ind_r = ind_r + 1$ // Storing the address of the replicated particle.
10. **else**
11. $i_d^{(ind_d)} = m$, $ind_d = ind_d - 1$ // Storing the address of the discarded particle.
12. **end**
13. **end**

Pseudocode 8 Modified Residual systematic resampling (RSR) algorithm.

The way in which memory related operations in the sample step are performed is shown in Pseudocode 9. The first *for* loop (line 2) is used to address the array of replicated indexes and the number of iterations in the loop is determined by the number of replicated particles. There is an internal *for* loop (line 6) which number of iterations is equal to the replication factors. The overall number of iterations of both loops is M. Operations of the sample step are performed on particles that have to be replicated $X(i_r^{(ind_r)})$. Then, $r^{(ind_r)} - 1$ sampled particles are written to the addresses of the discarded particles (lines 6, 7). The first sampled particles rewrites the original replicated particle (line 3) so that the replicated particle has to be stored in variable reg.

4.3.2 A logic structure

In this scheme, the RSR algorithm is combined with particle allocation method with arranged indexes. The addresses of both the replicated and discarded particles are stored in one memory (Figure 4.11) and they are arranged in a way that the replicated particles are placed in the upper and the discarded particles in the lower part of the index memory. Indexes of the replicated particles are incremented, while indexes of the discarded particles are decremented. The replicated factors are stored in the separate memory. The control block used controlling the operations of writing to and reading from the particle memory PMEM is derived from the replicated factors and it will be elaborated further.

Next, the architectures for the algorithms presented in Pseudocodes 8 and 9 are shown in Figures 4.12 and 4.13 respectively [4]. In Figure 4.12, weights are stored in the memory MEM_w and addressed by the address counter that corresponds to the variable m in Pseudocode 8.

Purpose: Generation of an array of particles $\{X_{n-1}^{(m)}\}_{m=1}^{M}$ at time instant n, $n > 0$.
Input: Arrays of replicated and discarded particle indexes i_r and i_d, the array of the replication factors r, and the array of particles $\{X_{n-1}^{(m)}\}_{m=1}^{M}$ at time instant $n-1$, $n > 0$.
Method:

$(X) = Sample(i_r, i_d, r, X)$

1. $ind_r = 0$, $ind_d = M - 1$ // Set the initial values for indexes.
2. **for** $ind_r = 1$ **to** $length(ind_r)$ // The main sampling loop.
3. $reg = X(i_r^{(ind_r)})$ // The replicated particle is stored in variable *reg*.
4. $X(i_r^{(ind_r)}) = Sample(reg)$ // Particle generation is performed
5. $ind_r = ind_r + 1$ // and the replicated particle is overwritten.
6. **for** $k = r^{(ind_r)} - 1$ **down to** 1
7. $X(i_d^{(ind_d)}) = Sample(reg)$ // Particle generation is performed
8. $ind_d = ind_d - 1$ // and the replicated particle is written to // the address of the discarded particle.
9. **end**
10. **end**

Pseudocode 9 Memory related operations of the sample step.

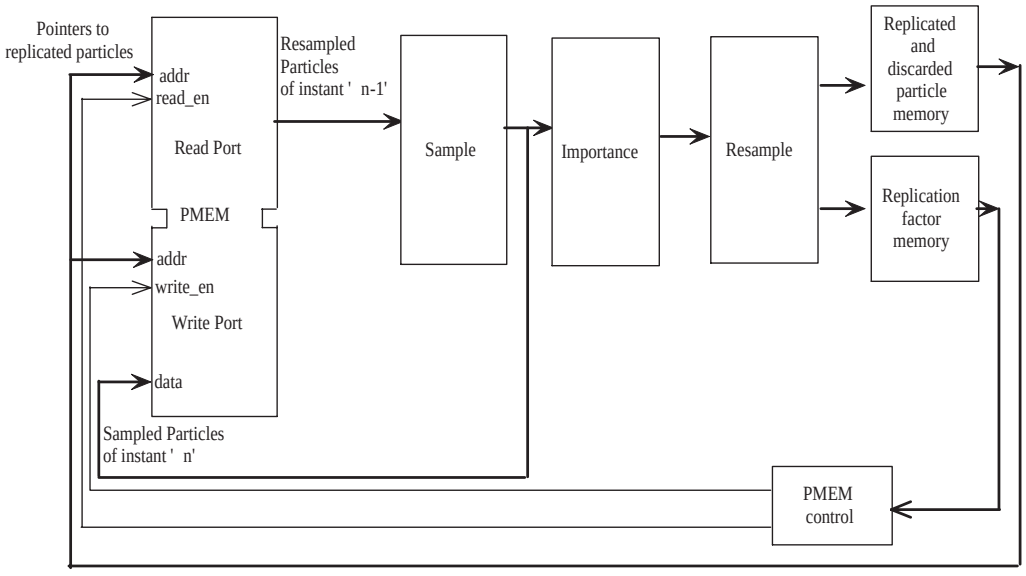


Figure 4.11: The architecture of the SIRFs with the RSR algorithm

Index generator is the block in which the arithmetics from the lines 5,6 and 7 from Pseudocode 8 is implemented. The index generator is simple and it contains two adders and one multiplier. The other part of the figure represents the architecture of the particle allocation step. Indexes of both the replicated and discarded particles are stored in the index memory MEM_i, while the replication factors are stored in the memory MEM_r. Since both indexes are stored in the same memory, the conflict is resolved using the multiplexor. When the replication factor is greater than zero, $Counter_r$ is enabled and its content is incremented. After the delay of one cycle ($delay_1$), the multiplexor allows for writing the particle index m into the memory MEM_i and at the same time the replication factor is written to the memory MEM_r. When the replication factor is zero, $Counter_d$ is enabled, the index of the discarded particle is written to the memory MEM_i while nothing is written to the memory MEM_r.

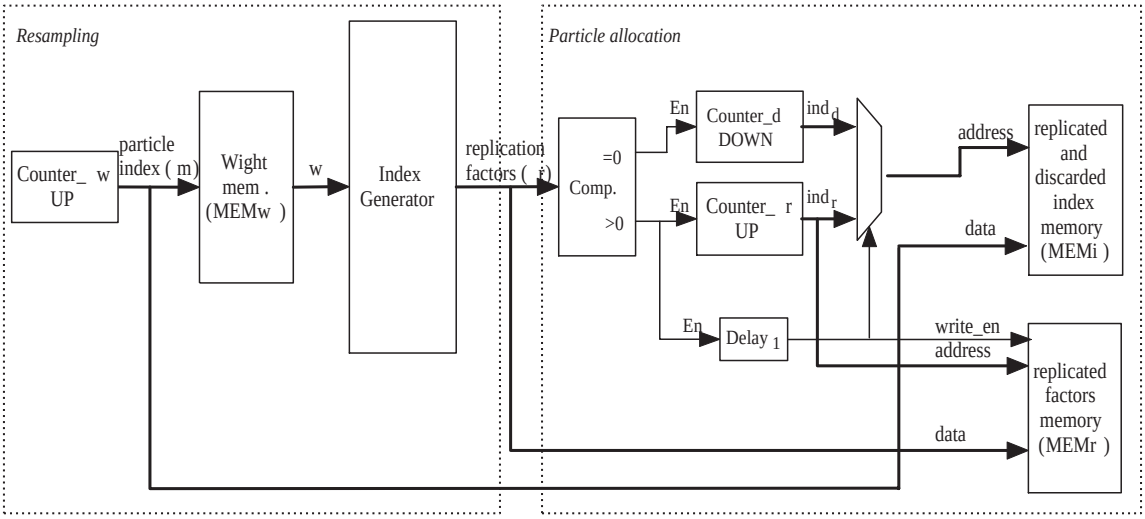


Figure 4.12: The architecture for the RSR algorithm combined with the particle allocation.

There are three main blocks in Figure 4.13: address generation, address control, and particle generation and storing. One memory $PMEM$ is used for storing particles, so that the particles are read from and written to it during the sampling step. Since reading and writing operations are performed concurrently, PMEM is a dual port memory. 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. It is presented as $delay_1$ in the figure. The memory of indexes MEM_i is used to address the particles in the memory PMEM in a way shown in Pseudocode 9. $Counter_f$ represents the variable k. The value of the replication factor is read from the memory MEM_r and written to the $Counter_f$. At this time instant, the output of the comparator 1 is set and the memory MEM_i is addressed by the $Counter_r$. Then, the replicated particle is read from the memory $PMEM$ and written to the register Reg in the next clock cycle. The write enable signal for the register is derived also from the Comparator 1. After one clock cycle, the output of the flip-flop (ff) is enabled which initiates decrementing the $Counter_d$. When $k < 2$, the flip-flop is reset and $Counter_r$ is incremented.

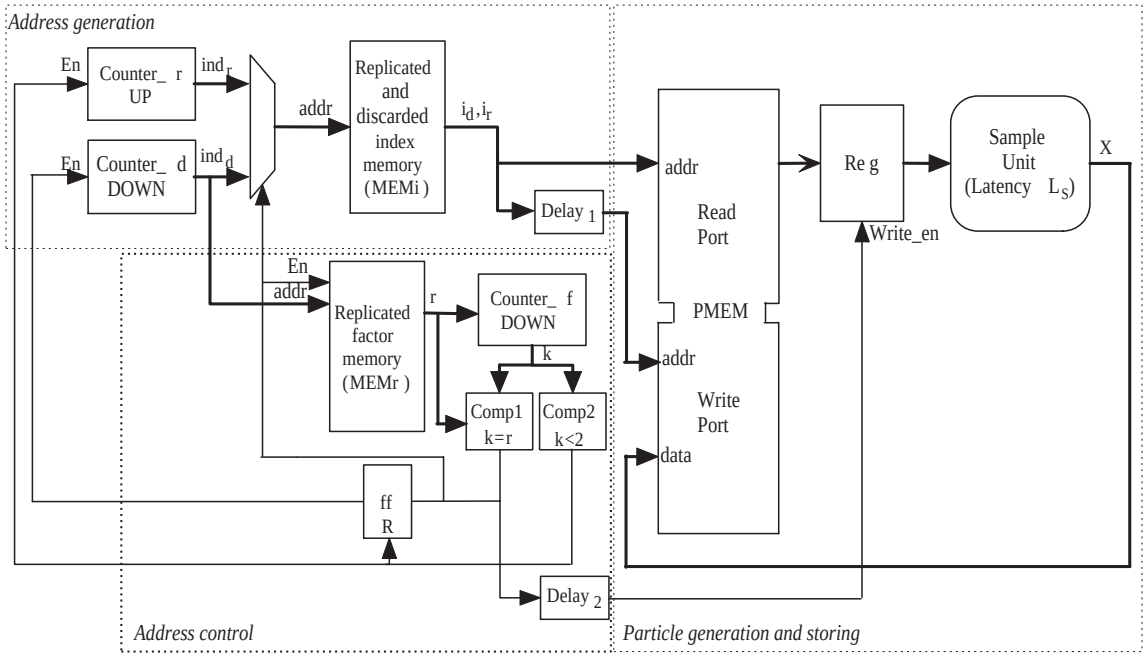


Figure 4.13: The architecture for memory related operations of the sampling step.

4.3.3 Architectures for SIRFs for bearings-only tracking

In section 4.3.2, the architectures for resampling step and memory related operations of the particle generation step for a generic particle filter are presented. In this section, we present the particle filtering operations that are model-dependent: particle generation and weight computation. The implementation is temporally concurrent in order to allow for achieving maximum speed.

Figure 4.14 shows a block diagram of the sampling unit which is a part of the architecture presented in Figure 4.13. The arithmetic operations of the particle generation step are described in Pseudocode 3. The sampling unit contains two noise generators which are implemented using Box-Muller approach [46]. The noise generator is a combination of lookup tables and arithmetic logic. Using the lookup tables eliminates a large latency generated by the arithmetic logic unit. Buffers associated with input vectors $(x_{n-1}, V_{x_{n-1}}, y_{n-1}, V_{y_{n-1}})$ and output vectors $(x_n, V_{x_n}, y_n, V_{y_n})$ are not shown in Figure 4.14.

Arithmetic operations of the weight computation step are illustrated in Pseudocode 3. The main operations are multiplications, trigonometric function $arctan()$, and exponent function $exp()$. Unrolled CORDIC [110, 111] is used as the operator for $atan()$ and $exp()$ because of its regular implementation structure. Additional logic is used for the correction of the angle-fault problem in the algorithm for *atan*. The purpose of this correction is to resolve ambiguity from $(-x, y)$ and $(x, -y)$ (i.e., their *atan* values are identical even though their physical locations are different). Next, the input z_n is subtracted from the output of the *atan* block. Since *atan* implementation is fine-grained pipelined, it is necessary to delay the input signal. The latency of the delay element is the same as the pipelining depth of the *atan* block. The subtracted values are then squared and multiplied by the constant $K = (2\pi\sigma_v^2)^{-1}$. Finally, the exponential operation is performed.

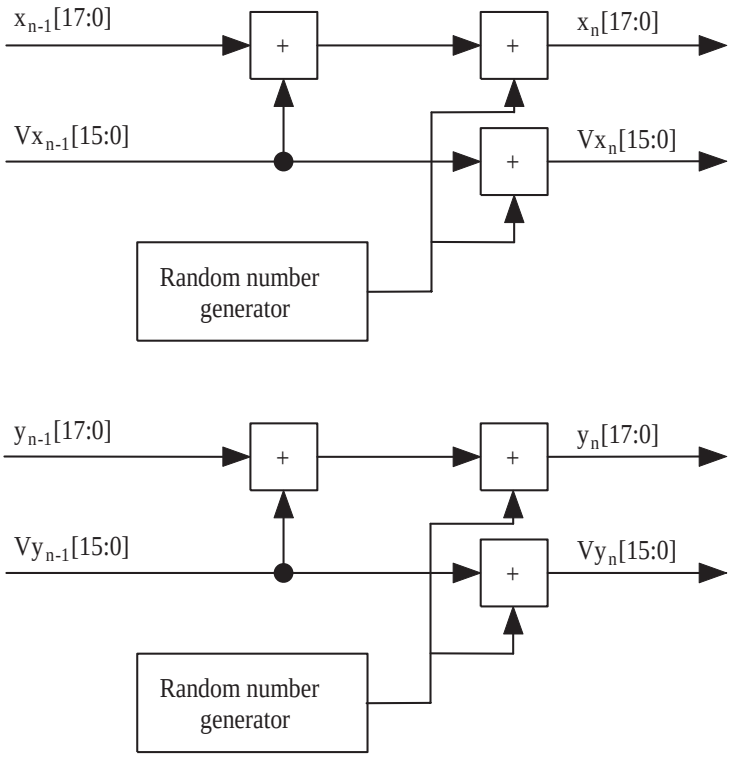


Figure 4.14: The architecture for the sampling unit for the bearings-only tracking problem.

All the variables in this design are converted to the fixed-point representation. The position coordinates are represented by 18 bits and the input z_n is represented by 16 bits [8]. It is observed that the final results (weights w_1) are equal 0 in finite precision arithmetic if variable A is greater than 255. In order to reduce area requirements, we used the multipliers with the lowest possible number of bits for which the loss in performance is 10% at the most. So, the first multiplier in the figure multiples only 8 bit numbers. The output of the comparator is used as a select signal for the output multiplexor. When the output c is logic one ($A > 255$) then $w_n = 0$. The latency of the delay element ($delay_2$) is equal to the pipelining depth of the exponential block.

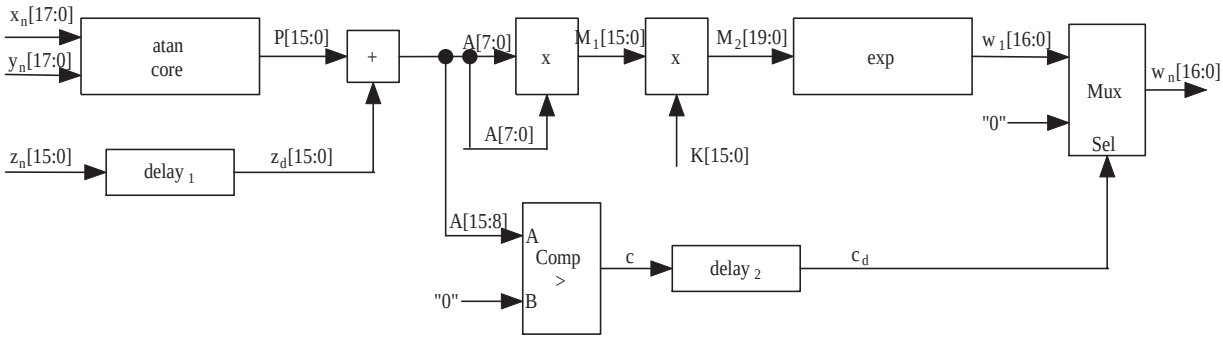


Figure 4.15: The architecture for the weight computation step for the bearings-only tracking problem.

The architecture for the exponential function is shown in Figure 4.16. It uses CORDIC

core from the Xilinx Virtex II Pro library. The outputs of the core are *sinh* and *cosh* and the input is restricted to the range of $[-\pi/4, \pi/4]$. This requires that the exponent of the exponential function is expressed as the sum of the integer and the fractional part. The exponential function of the integer part of the exponent is pre-calculated and stored in the look-up table represented by the ROM with 32 16-bit words. The delay unit is used after the ROM because the outputs of both the ROM and the CORDIC core must be synchronized. The latency of the delay is equal to the pipelining depth of the used CORDIC core. Finally, the signal A_1 and the delayed signal R are multiplied.

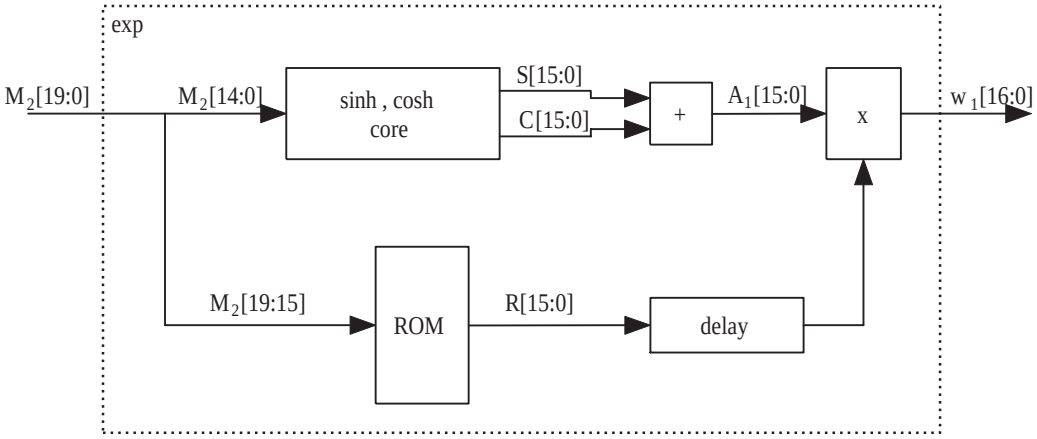


Figure 4.16: Implementation of the exponential function in Xilinx Virtex II Pro FPGA.

4.3.4 FPGA implementation results

In this section, we present the results of the implementation of the architectures described in Sections 4.3.2 and 4.3.3 on a Xilinx Virtex II pro FPGA platform. The architectures were captured using Verilog HDL and 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, placed and routed the design on a Xilinx Virtex II Pro device (XC2VP125). We present the resources utilization results as a combination of the number of logic slices, multiplier blocks, and memory bits.

The resource requirements for the SIRF are shown in Table 4.7. We see that the amount of resources used is small (less than 4% of the overall resources) so that there is a space for parallel design if higher speed is required. The implementation of the sample, importance, and resampling blocks is based on the architectures presented in the previous two sections.

The finite precision approximation of variables is performed in the SystemC language [104] using similar approach as in [67]. The particles are represented using 18 bits with one sign bit, one bit to the left and 16 bits to the right of the decimal point, while the weights are represented with 16 bits, one bit in the decimal and 15 bits in the fractional part. The final memory requirements are: four $18 \times M$ memories for storing particles, one $16 \times M$ memory for storing weights, and two $16 \times M$ memory for storing replication factors and indexes. The 16-bit memory for indexes specifies the maximum number of particles that can be processed by the particle filter to $M = 65536$. So, the overall storage space is $120 \times M$ bits. In this

design, M is restricted to 1024 by the size of the Xilinx Virtex II Pro block RAMs. In the particle generation step, 2 memories are used for the random number generation and 4 for storing the particles. One memory is used in the importance step for storing weights and two in the resampling step for storing indexes and replication factors.

The speed of the implementation is limited by the speed of the slowest block, which is the CORDIC block. In the best case, the clock frequency of the fine-grained pipelined 24-bit CORDIC is 175MHz. To be on the safe side, the clock frequency of this design is chosen to be 100MHz.

Resource	Sample	Importance	Resampling	Total	% of the available resources
Slices	341	1535	177	2053	3.7%
Block RAMs	6	1	2	9	1.6%
Block multipliers	0	3	1	4	1.7%

Table 4.7: Resource utilization for the SIRF implementation of the bearings-only tracking problem on the Xilinx XC2VP125 device.

Chapter 5

Algorithms and architectures for distributed particle filters

In this section, we propose novel resampling algorithms with architectures for efficient distributed implementation of particle filters. The proposed algorithms improve the scalability of the filter architectures affected by the resampling process. The main design goal in this section is to minimize the execution time of the SIRF. This is done through exploiting data parallelism and pipelining of operations. In order to decrease SIRF execution time, an algorithm that allows for distributed resampling and reduced communication in the interconnection network is proposed. This algorithm is presented in Section 5.2 and is named distributed Resampling with Proportional Allocation (RPA). It yields the same resampling result as the sequential resampling method (for example systematic resampling). Further improvement of the execution time is achieved through making the communication through the interconnection network deterministic and local. These algorithms use non-proportional sampling (Resampling with Non-proportional Allocation - RNA), and they are presented in Section 5.3. Different architectures suitable for distributed RPA and RNA algorithms are discussed in Section 5.4. The objective in these architectures is to pipeline the communication through interconnection network (particle routing) with the subsequent sampling step as described in Section 3.3.2. We also evaluate architecture parameters on an FPGA platform.

5.1 Centralized resampling

Centralized resampling is a straightforward approach to implementing the SIRFs based on the architecture presented in Figure 3.3. Particle generation and weight computation are executed in parallel by the PEs. The CU carries out resampling and particle routing as well as overall control. The sequence of operations and directions of communication are shown in Figure 5.1(a). Here, the CU is responsible for full resampling, which is performed sequentially and therefore the resampling time is not scaled by K. The communication requirements of this implementation are immense. The CU collects the M weights in order to perform resampling and returns M indexes and replication factors to the PEs. Here, we assume that particle allocation with arranged indexes is applied. While the communication of weights and indexes is deterministic, the particles are routed in a non-deterministic fashion. The number

of particles transferred between PE_k and the CU is $|N^{(k)} - N|$ for $k = 1, ..., K$. The direction of communication is from the PE to the CU for the PE with particle surplus after resampling ($N^{(k)} - N > 0$) and from the CU to the PE for the PE with particle shortage ($N^{(k)} - N < 0$). The overall amount of particles that has to be transferred through the network is $M/2$ for the worst case. Even in the fully connected network, the scalability of the implementation is significantly affected by the sequential resampling and particle routing.

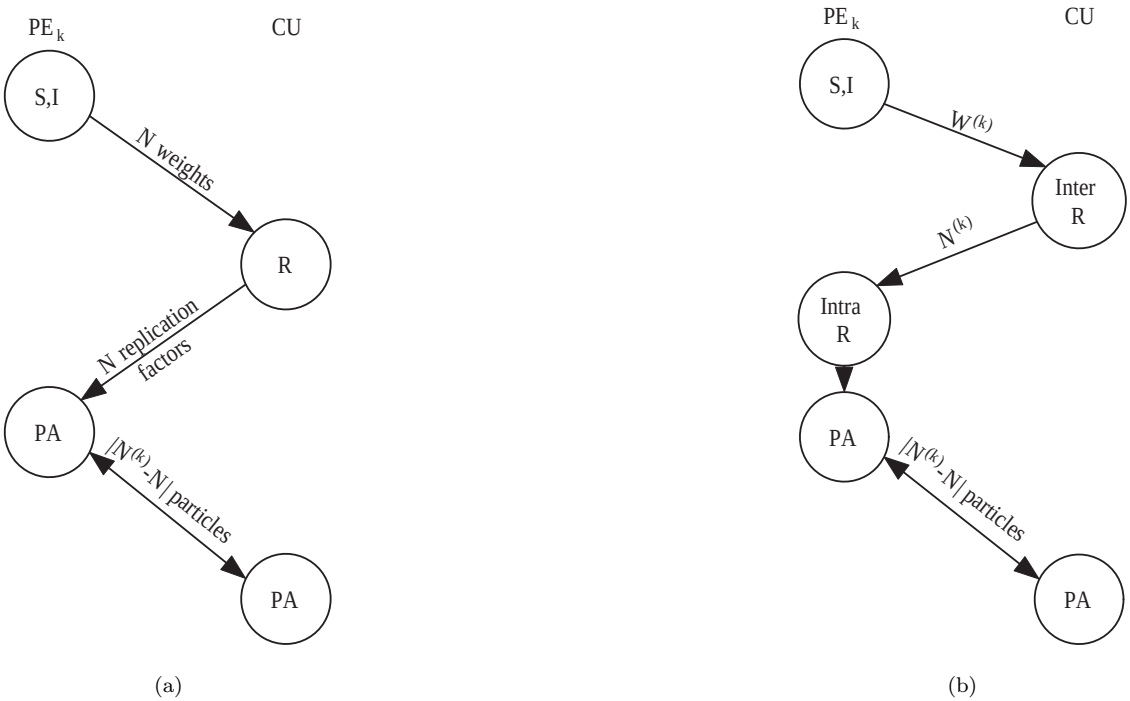


Figure 5.1: Sequence of operations performed by the k-th PE and the CU for (a) centralized resampling and (b) RPA. The direction of communication as well as data that are sent are presented. The abbreviations are: S-sample, I-importance, R-resampling, PA-particle allocation.

5.2 Distributed RPA

In this section, a method based on stratified sampling with proportional allocation is described. The sample space is divided into K disjoint areas or strata, where each stratum corresponds to a PE. The density of particle weights can then be written as a mixture of K densities restricted to the corresponding strata. Proportional allocation among strata is used, which means that more samples are drawn from the strata with larger weights. After the weights of the strata are known, the number of particles that each stratum replicates is calculated using RSR, and this process is denoted as inter-resampling since it treats the PEs as single particles. Finally, resampling is performed inside the strata which is referred to as intra-resampling. So, the resampling algorithm is accelerated by using loop transformation or specifically loop distribution [114], which allows for having an inner loop that can run in parallel on the PEs (intra-resampling) with small sequential centralized pre-processing (inter-rasampling). The weight of the PE is calculated as a sum of the weights of the particles inside

the PE, i.e. $W^{(k)} = \sum_{i=1}^{N} w^{(i,k)}$ for $k = 1, ..., K$. A diagram and the sequence of operations performed by the PE and the CU are shown in Figure 5.1(b).

The algorithm for RPA is shown by Pseudocode 10. The inputs of the algorithm are the PE weights and the output is the number of particles $N^{(k)}$ that each PE will produce after resampling, where $E(N^{(k)}) = MW^{(k)}$ for $k = 1, ..., K$. The RSR algorithm is applied to get $N^{(k)}$, for $k = 1, 2, ..., K$ by propagating the uniform random number in a similar fashion as in the systematic resampling algorithm. In the algorithm, $N^{(k)}$ is obtained by truncating $(W^{(k)} - U^{(k)}) \cdot M$. The minimum value of the truncated product is -1 so that the minimum value of $N^{(k)}$ is zero. Resampling is performed in each PE in parallel during the intra-resampling step. The input of the intra-resampling algorithm is the number of particles that should be generated in the resampling procedure. We have to stress that there is no difference in results between RPA and sequential resampling.

Purpose: Calculation of the number of particles $N^{(k)}$ for the intra-resampling algorithm.
Input: Array of PE weights $W^{(k)}$ for $k = 1, ..., K$.
Method:

$U^{(1)} \sim U[0, \frac{1}{M}]$ //Generating random number $U^{(1)}$.
for $k = 1$ **to** K //Inter-resampling loop.
  $N^{(k)} = \lfloor (W^{(k)} - U^{(k)}) \cdot M \rfloor + 1$ // Calculating replication factors.
  Send $N^{(k)}$ to PE_k
  $U^{(k+1)} = U^{(k)} + \frac{N^{(k)}}{M} - W^{(k)}$ // Updating the uniform random number.
end
do in parallel
  Intra-resampling for all PEs
end

Pseudocode 10 A distributed RPA algorithm that utilizes the RSR approach.

The RSR algorithm is very attractive for hardware implementation since it has only one loop (there are two loops in systematic resampling), it can be easily pipelined so that it can calculate a replication factor per clock cycle, and it easily deals with different number of particles at the input and at the output. In systematic resampling *while* loop has unknown number of iterations which makes it difficult to apply pipelining. Of course, the same resampling result would be obtained if residual or systematic resampling are applied as the inter-resampling algorithms.

An example of particle exchange for the RPA algorithm is shown in Figure 5.2. The SIRF architecture with four PEs is considered, where each PE processes $N = 100$ particles. The distribution of the normalized PE weights before resampling is presented in the table. After inter-resampling, the number of particles that each PE will produce is determined and it is $200, 50, 105$ and 45 respectively. So, PEs 1 and 3 have surpluses of particles. In this example, PE_1 sends 50 particles to both PE_2 and PE_4, and PE_3 sends 5 particles to PE_4.

The main difference between the centralized and distributed RPA lies in reducing the amount of deterministic communication and the move of the resampling from the CU to the PEs (Figure 5.1). The time for the resampling procedure in distributed RPA is reduced $M/(M/K + K)$ times, where M/K corresponds to the intra-resampling time and K is a

Weights of the PEs before resampling

	PE weights before resampling
$W^{(1)}$	0.5
$W^{(2)}$	0.125
$W^{(3)}$	0.2625
$W^{(4)}$	0.1125
Sum	1

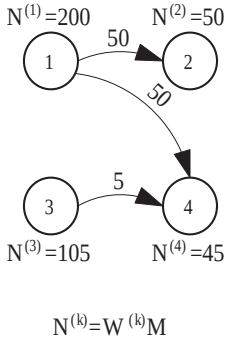


Figure 5.2: An example of particle exchange for the RPA algorithm.

time for inter-resampling. It is important to note here that inter-resampling requires global communication among the PEs, while intra-resampling is completely local within the PEs. The $2M$ words representing weights and indexes that are exchanged in the centralized resampling are reduced to $2K$ words ($W^{(k)}$ and $N^{(k)}$) in RPA. However, scalability of the implementation is still affected by the particle routing step, which is unchanged. If we assume equal clock period for resampling and the other particle filters steps, then $T_{ex} = (2M/K + L_p + K + M_r)T_{clk}$, where K represents the delay of inter-resampling and M_r is the delay of particle routing. When the PEs and the CU are connected with a single bus, then the delay M_r becomes dominant. Scalability of the design is affected so much by the bus structure, that there is almost no gain in pursuing parallel implementation. The efficient architecture that uses the K buses and supports pipelining the particle routing with the sample step is proposed in Section 5.4.1.

5.3 Distributed RNA

Even though distributed RPA allows for distributed and parallel implementation of resampling, it requires a complicated scheme for particle routing which implies a complex CU design and area increase. Besides, there is a need for an additional global pre-processing step (inter-resampling) which introduces an extra delay. These problems can be solved by using an RNA algorithm. Here, we use the term *group* instead of PE, where a group is formed from one or more PEs.

In every application of sampling, two key decisions have to be made: how to choose strata and how many samples $N^{(k)}$ to generate in each stratum. In both RPA and RNA, strata are induced based on neighborhood criteria, so that particles inside one group form a stratum. In RPA, the number of particles drawn is proportional to the weight of the stratum. On the other hand, in RNA the number of particles within a group after resampling is fixed and equal to the number of particles per group, $N^{(k)} = N$. So, full independent resampling is performed by each group. In addition, routing of particles among the groups after resampling is necessary due to the possibility of having very unequally distributed weights. The main advantage of RNA is that routing of particles is deterministic and is planned in advance by a designer. This is a very important difference in comparison with distributed RPA where the particle routing step represents the biggest problem for implementation due to its unpredictability. Another

characteristics of RNA is that the weights after resampling are not equal to $1/M$, but they are equal inside the groups $\widetilde{w}_n^{(k,i)} = W^{(k)}$, for $k = 1, 2, \ldots K$ and $i = 1, 2, \ldots N$.

The general particle filter algorithm with RNA is outlined by Pseudocode 11.

1. Exchange particles among groups deterministically.
2. Generate particles in each group in parallel by sampling $\mathbf{x}_n^{(k,i)} \sim \pi(\mathbf{x}_n)$ for $k = 1, \ldots, K$ and $i = 1, \ldots, N$.
3. Perform the importance step in each group in parallel. The weights are calculated by $w_n^{*(k,i)} = w_{n-1}^{(k,i)} \frac{p(\mathbf{z}_n|\mathbf{x}_n^{(k,i)}) p(\mathbf{x}_n^{(k,i)}|\mathbf{x}_{n-1}^{(k,i)})}{\pi(\mathbf{x}_n^{(k,i)})}$ for $k = 1, \ldots, K$ and $i = 1, \ldots, N$.
4. Normalize the weights of the particles with the sum of the weights in the group: $w_n^{(k,i)} = \frac{w_n^{*(k,i)}}{W^{(k)}}$ where $W^{*(k)} = \sum_{j=1}^{N} w_n^{*(k,j)}$ and $W^{(k)} = W^{*(k)} / (\sum_{j=1}^{K} W^{*(k)})$ for $k = 1, \ldots, K$.
5. Perform resampling inside the groups and obtain new random measures $\{\widetilde{\mathbf{x}}_{1:n}^{(k,i)}, \widetilde{w}_n^{(k,i)} = W^{(k)}\}$ for $k = 1, \ldots, K$ and $i = 1, \ldots, N$.
6. Go to step 1.

Pseudocode 11 Particle filter steps when distributed RNA is used.

There are several differences in comparison with the original SIR filter. Here, normalization is performed with the local sum $W^{(k)}$. Resampling is performed locally per each group and the weights are equal inside the group. In our further analysis, we consider particle filters where the normalization step is avoided by a simple modification of the resampling algorithm shown in Section 3.2.3.

A comparison of the resampling and the particle routing steps of RPA and RNA are shown in Table 5.1.

	RPA	RNA
Inter-resampling	Calculate $U^{(k)}$ and $N^{(k)}$	None
Intra-resampling	N input, $N^{(k)}$ output particles	N input and output particles
Particle routing	Depends on the particle distribution	Deterministic

Table 5.1: Comparison of the RPA and RNA steps.

We distinguish between three methods of particle exchange after resampling: regrouping, adaptive regrouping and local exchange. These methods are presented in Figure 5.3 which is based on the same example described in Figure 5.2. The description of these methods is provided in the sequel.

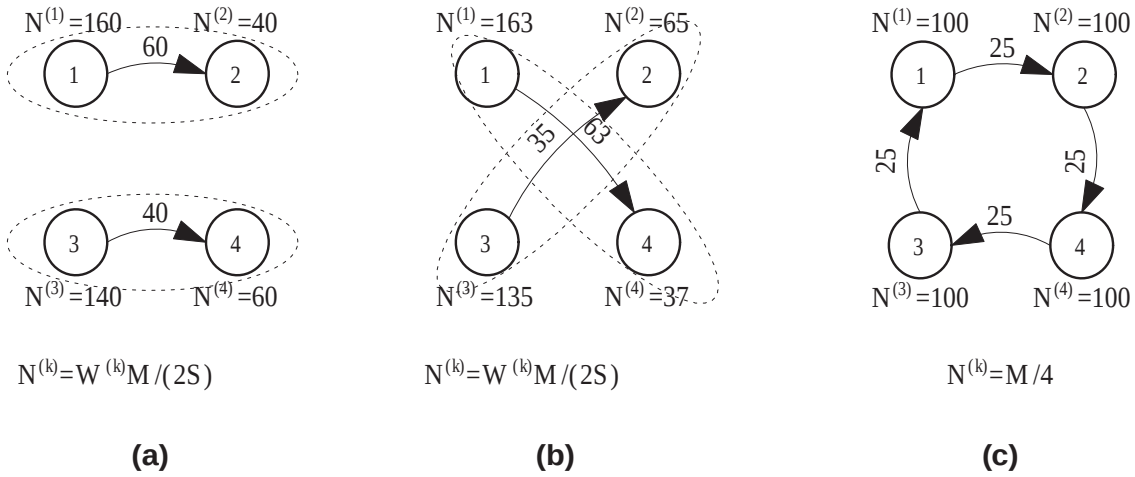


Figure 5.3: An example of particle exchange for RNA algorithms with (a) regrouping, (b) adaptive regrouping and (c) with local exchange. Here, S is the sum of weights in the group.

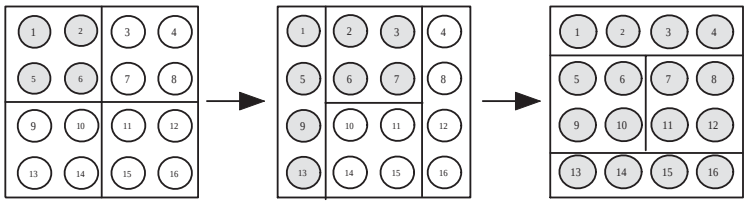
Distributed RNA with regrouping

In RNA with regrouping, resampling and particle routing are performed inside the groups using the RPA method. For example, in Figure 5.3(a) PE_1 and PE_2 form one and PE_3 and PE_4 another group. The RPA algorithm is applied for both groups. As a result, PE_1 and PE_2 produce 160 and 40 particles after resampling, so that 60 particles from PE_1 are transferred to PE_2. At the next sampling instant, the PEs are rearranged so that they form different groups. For example, the new groups can be composed of PE_1 and PE_3, and PE_2 and PE_4. After each time instant, regrouping is performed so that particles are exchanged among PEs and the variance is reduced.

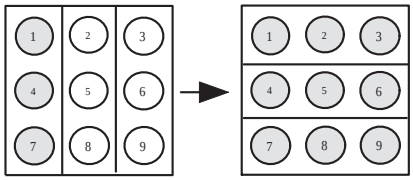
In RNA with regrouping, resampling and particle routing are done in parallel on several groups where each group consists of R PEs. For example, in Figure 5.4(a), $K = 16$ PEs are divided into four groups where each group has $R = 4$ PEs. The particle filter steps (sample, importance and resampling) are performed for each group concurrently and independently (steps 2-5 of the RNA algorithm). In RNA with regrouping, particles are implicitly exchanged among PEs when PEs are regrouped. The period of regrouping is denoted as distribution factor D.

It is interesting to analyze the meaning of the distribution factor D. When all the particles with non-zero weights are in one PE, we are interested in the number of cycles needed that these particles propagate to all other PEs. This is exactly determined by the distribution factor and it is shown in Figure 5.4. We assume that only PE_1 has non-zero weights. In the first sampling period, three more PEs ($2, 5$ and 6) that belong to the same group as PE_1 will receive non-zero weights from PE_1 after resampling (Figure 5.4(a)). Propagation of the weights during particle routing is presented with a dark color. In the subsequent time instant, groups are formed from different PEs so that the non-zero particles are further propagated (PEs $3, 7, 9$ and 13). Thus, full resampling is done in D sampling periods.

The criteria for choosing the parameters R and D for given K are the overall minimal



(a)



(b)

Figure 5.4: Routing in RNA with regrouping for (a) $K = 16$, $R = 4$ and $D = 3$, and (b) $K = 9$, $R = 3$ and $D = 2$.

distribution factor and simplicity of communication network and distributed controllers. The optimal parameters R and D can be chosen as the minimum of the curve RD for the feasible values of R and D. In the case of square grid mesh structure as in Figure 5.4, the optimal choice for parameters are $R = \sqrt{K}$ and $D = 2$. Non optimal and optimal regrouping are shown in Figures 5.4(a) and (b).

Since the simplicity of the controllers is one of the main design goals, we restrict the number of PEs per groups to be 2 or 3. If the number of PEs in the group is four or larger, a very complicated controller is necessary in order to perform fast particle routing as described in Section 5.4.1. When $R = 2$, the local controllers are simple because there is only one PE with surplus and one with shortage of particles. Choosing so small value for R could cause high distribution factor and large number of periods until full propagation of particles is achieved. If $R = 2$ and $K = 16$, the minimal distribution factor is $D = 6$.

RNA with adaptive regrouping

RNA with regrouping uses the predefined fixed rules to form the groups and does not take advantage of knowing the distribution of the group weights. By utilizing this knowledge, it is possible to achieve faster weight balancing (smaller D) than with regrouping with fixed rules. RNA with adaptive regrouping forms groups from the PEs with the largest and the smallest group weights. For example, in Figure 5.3(b) PE_1 and PE_3 have the largest and the smallest PE weights so that they form one group. The other group is formed from the remaining PEs. Inside the groups, the RPA algorithm is applied. Weights after resampling are calculated based on step 5 of Pseudocode 2. This method utilizes the Randez-Vouz load

balancing algorithm [32], which is a simple greedy algorithm that associates the heavily and the lightly loaded groups. The main disadvantages of RNA with adaptive regrouping are that groups contain only two PEs ($R = 2$) and the connections among the PEs are not local in general.

Distributed RNA with local exchange

In RNA with regrouping, the RPA algorithm is still performed inside groups so that the particle routing process is still random, even though it is done on a smaller set of particles. Randomness during particle routing makes it difficult for pipelining between the particle routing and sampling steps.

The example of the RNA algorithm with local exchange is shown in Figure 5.3(c). Resampling is done inside the PE and then particles are exchanged in a deterministic way only among the neighboring PEs. Routing is done through local communication. The amount of particles sent between PEs is fixed and defined in advance. In the example, it is $N/4 = 25$. This is a very important difference in comparison with the RNA with regrouping where particles are routed among the PEs in the group non-deterministically (except when $R = 2$). Since groups are formed from one PE, the weights after resampling are set to $W^{(k)}/N$. Local communication can give rise to a large number of periods until full resampling is achieved, which restricts the level of parallelism.

As the communication between the PEs is only local, the choice of architectural parameters is similar as in the case of RNA with regrouping for $R = 2$. Local communication can give rise to a large number of periods until full resampling is achieved, which restricts the level of parallelism.

5.3.1 Effects of resampling on obtained estimates

In SIRFs, the output estimate before resampling can be calculated as: $\overline{g} = \sum_{m=1}^{M} w^{(m)} g(x^{(m)})$, where $x^{(m)}$ are the states of the particles, $g(\cdot)$ is an arbitrary function, and $w^{(m)}$ represents a normalized importance weight [40, 41]. For parallel implementation, the estimate can be written in the form: $\overline{g} = \sum_{k=1}^{K} W^{(k)} \sum_{i=1}^{N} w^{(k,i)} g(x^{(k,i)})/W^{(k)} = \sum_{k=1}^{K} W^{(k)} \overline{g}^{(k)}$, where $\overline{g}^{(k)}$ represents the expected value of $g(x)$ from a distribution $w^{(k,i)}$ in the $k-$th PE. The estimate after applying distributed RPA is of the form: $\widetilde{g} = 1/M \sum_{k=1}^{K} \sum_{i=1}^{N} N^{(k,i)} g(x^{(k,i)})$ where $N^{(k,i)}$ represents the number of times the particle k, i is replicated after resampling and $E(N^{(k,i)}) = w^{(k,i)} M$. The estimate after applying distributed RNA is of the form: $\widehat{g} = 1/N \sum_{k=1}^{K} W^{(k)} \sum_{i=1}^{N} N^{(k,i)} g(x^{(k,i)})$, where the number of replications of each particle is calculated as $E(N^{(k,i)}) = w^{(k,i)} N/W^{(k)}$. It is easy to show that $E(\widetilde{g}) = E(\widehat{g}) = \sum_{k=1}^{K} W^{(k)} \overline{g}^{(k)}$, which is equal to $\overline{g}$. This means that both estimates of $\overline{g}$ are unbiased. The result is expected for both types of sampling due to Theorem 5.1 from Cochran [25], which claims that if in every stratum the sample estimate is unbiased, then the overall estimate too, is an unbiased estimate of the population mean.

SIRFs with full and without resampling can be considered as special cases of the RNA algorithm. In the first case, $K = 1$ and the whole resampling is performed inside one PE. In

the second case, $K = M$ so that resampling is performed on a single particle. Since the input and output of the resampling is only one particle, there is actually no resampling.

It is not easy to compare $Var(\widetilde{g})$ and $Var(\widehat{g})$ in general. It was observed by simulations that there was almost no difference in the variances if the weights are equally distributed among the PEs. However, the variance of the RNA algorithm was much greater when there was only one PE with non-zero weights. This problem can be resolved by exchanging the particles between PEs after resampling deterministically (step 1 of the RNA algorithm).

5.3.2 Performance analysis

In this section, the performances of the sequential particle filter and the particle filter with distributed RNA with local exchange with different number of PEs are compared. Particle filters are applied to the bearings only tracking problem with the model and initial conditions as in [49]. The mean square error (MSE) and the percentage of diverged tracks are chosen as a performance metrics. We consider that the track diverges if all importance weights have negligible values or the mean square error is outside the pre-defined limits.

The architectural model that is chosen for the particle filter with distributed RNA is the 2-cube torus type network [101]. We consider 2-ary, 4-ary and 8-ary torus networks. In the model it is assumed that each PE has a single input and output port and that the communication protocol is full duplex. Deterministic particle routing is done in a way that particles are sent to the PE above and to the PE that is left. In this way, particles are routed with a statically scheduled communication pattern.

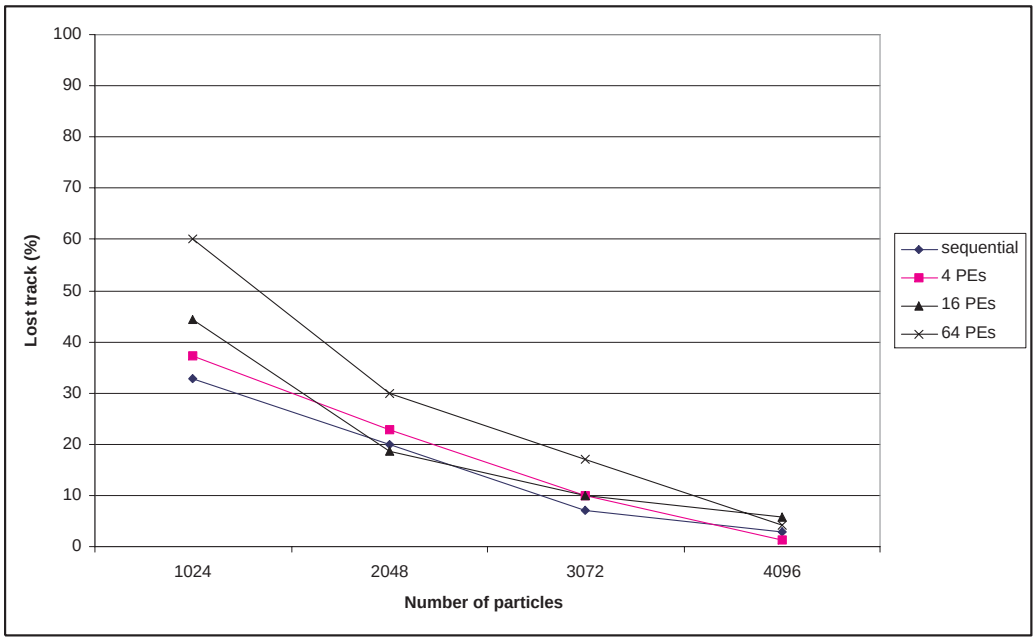
The simulation results are shown in Figure 5.5. We can see that all the MSEs are comparable. This is an important result, because for initial conditions and model from [49], there is a time instant in which only a small amount of the particles survives. The deterministic routing solves the problem of having only one PE of particles with large weights and the rest with negligible weights.

5.4 Particle filter architectures with distributed resampling

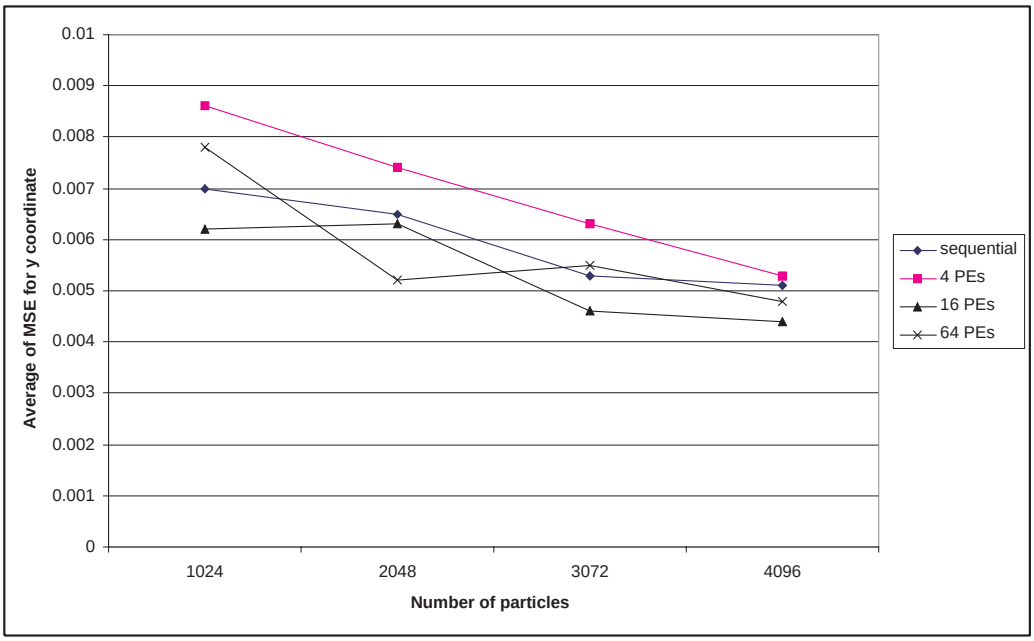
5.4.1 Distributed RPA architectures

One possible architecture for distributed RPA with four PEs that allows for pipelining the particle routing step with the next sampling step is shown in Figure 5.6. The main idea is to store the particles that will be routed among the PEs into dedicated memories in the CU and to have very fast interface capable of reading particles from the CU and routing them to the PEs in one clock cycle.

The particles that are replicated as a result of the resampling for PE_k are stored into local memories Mem_{kk} for $N^{(k)} < N$. When there is a surplus of particles, these particles are stored in CU memories Mem_{ki} for $i = 1, ...K$ and $k \neq i$. For example, the memory Mem_{12}



(a)



(b)

Figure 5.5: (a) Percentage of divergent tracks and (b) MSE versus the number of particles for different levels of parallelism. In the case of 4, 16 and 64 PEs, RNA with local exchange is applied.

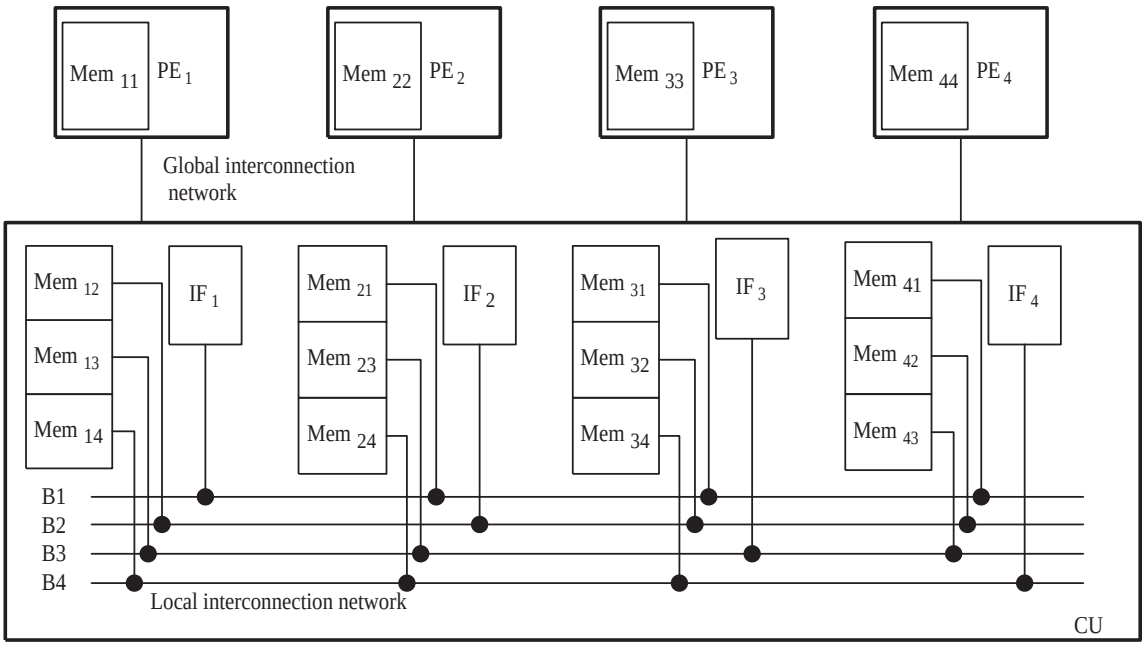


Figure 5.6: Architecture of the SIRF with distributed RPA with four PEs. The CU is implemented to support pipelining between the particle routing and sampling steps.

is used to store the surplus of particles from PE_1 that should be routed to PE_2. If there is a shortage of particles in PE_k, then PE_k reads particles from the interface IF_k which is connected to the memories Mem_{ik}.

Routing is performed through three steps. First, particles from the PEs with the particle surplus are sent to the CU through the global interconnection network. Then, routing is performed through the IF block inside the CU using the buses B_i for $i = 1, ..., 4$. Each IF is connected to the corresponding memories with a bus and it acts as a master on the bus. Finally, particles are transferred to the destination PEs through the global interconnection network. The size of the memories is determined for the worst case (when one PE acquires all the N particles from another PE) and it is N words, where each word consists of the particles and their replication factors. So, the overall memory requirements are $16N = 4M$ words which is 4 times more than in the sequential case.

The timing diagram for the PE with particle shortage together with its communication with CU is presented in Figure 5.7. Resampling is performed using the following steps:

1. CU performs inter-resampling and sends the output number of particles $N^{(k)}$ to PE_k for $k = 1, ..., K$. The CU also calculates the amount of data that should be transferred among the PEs.
2. The PEs perform intra-resampling so that the first $N^{(k)} \leq N$ particles are stored into the local memory Mem_{kk} and when $N^{(k)} > N$, the surplus is sent to the CU.
3. The particles are allocated to the corresponding memories Mem_{ki}. The PEs have no information how the particles are further routed in the CU.
4. During the sampling step, the PE reads the particles first from the local memories. The

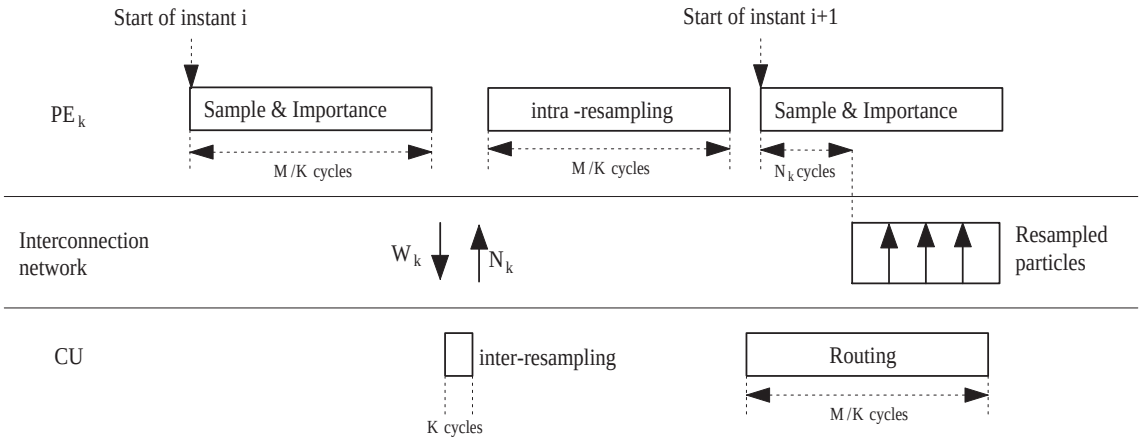


Figure 5.7: Timing diagrams for the SIRF with distributed RPA. Communication through the interconnection network is shown for the PE_k with shortage of particles.

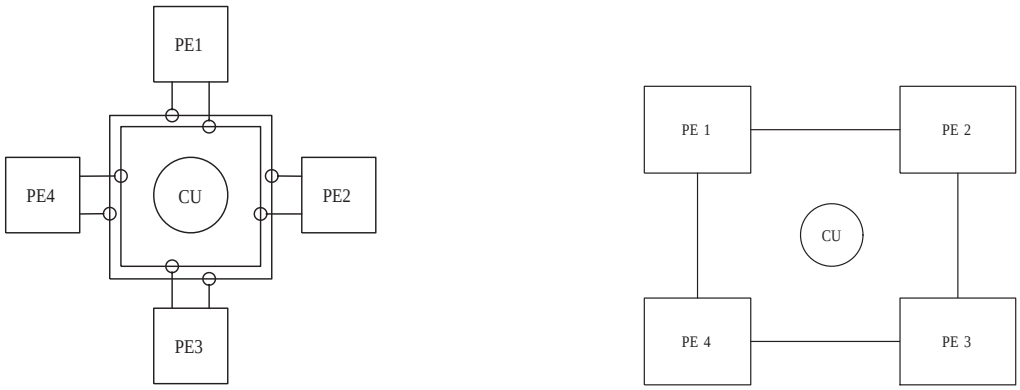
PEs with the shortage of particles, acquire the rest of particles from the IF as shown in Figure 5.7.

This architecture has an execution time very close to the minimum execution time at the expense of increased resources. There are four parallel buses from the PEs to the CU and four parallel buses inside the CU. The area is also increased because particles are additionally stored inside the CU. The clock speed is limited by the memory access and by the complexity of the CU. The design methodology and implementation results for the distributed RPA in ASIC are given in [56].

5.4.2 Distributed RNA architectures

For $K = 4$, we can distinguish among several architectures for SIRFs that utilize the RNA slgorithms. In Figure 5.8(a), a particle filter architecture that can be used for all RNA algorithms with four PEs is presented. Since the connections are not local, it is especially suitable for RNA with adaptive regrouping. Two lines in the figure represent buses used for particle routing. The algorithm running on the CU configures switches so that only two PEs access one bus at any given time [54, 84]. In the case of RNA with fixed regrouping, the switches are configured in fixed order. For example, if $D = 2$, the switches can be configured so that the following sequence is repeated: 12 and 34, 13 and 24. In RNA with adaptive regrouping, the switches are configured so that they connect the PEs with largest and smallest weights. The RNA with local exchange can also be run on the same architecture.

A simpler architecture is shown in Figure 5.8 (b). The network topology that is chosen is a 2×2 mesh. The network is static and based only on local interconnections. The CU is simple and its functions are collecting partial sums of weights and outputs, returning the final sum of weights to the PEs and the overall control. The CU is connected to the PEs through a single bus. However, the RNA with adaptive regrouping cannot be applied because not all the PEs are physically connected.



(a) (b)

Figure 5.8: Architectures for particle filters with $K = 4$ PEs that support (a) all RNA algorithms and (b) does not support RNA with adaptive regrouping.

The architectures become more complex for a higher level of parallelism. A scalable architecture that can support both methods of RNA with regrouping (adaptive and fixed) for $K \leq 4$ and their ASIC implementation is presented in [56].

5.4.3 Space exploration for distributed particle filter with RNA with local exchange

The area and speed of the distributed particle filter with RNA with local exchange and particle allocation with replication factor are estimated for the bearings-only tracking problem. The same parameters and model are used as in [49]. The range of interest is restricted to the region $[-32, 32] \times [-32, 32]$. As a benchmark, the chosen hardware platform is Xilinx Virtex-II Pro [117]. The resources are analyzed as a combination of the number of logic slices, multiplier blocks and memory bits.

The finite precision approximation of variables is performed in the SystemC language [104]. The particles are represented using 24 bits with one sign bit, five bits to the left and 18 bits to the right of the decimal point, while the weights are represented with 16 bits with one bit in decimal and 15 bits in the fractional part. The complex mathematical functions are implemented using CORDIC, and the Gaussian random number generator is implemented using the Box-Muller method. The implementation is spatially concurrent in order to achieve maximum speed.

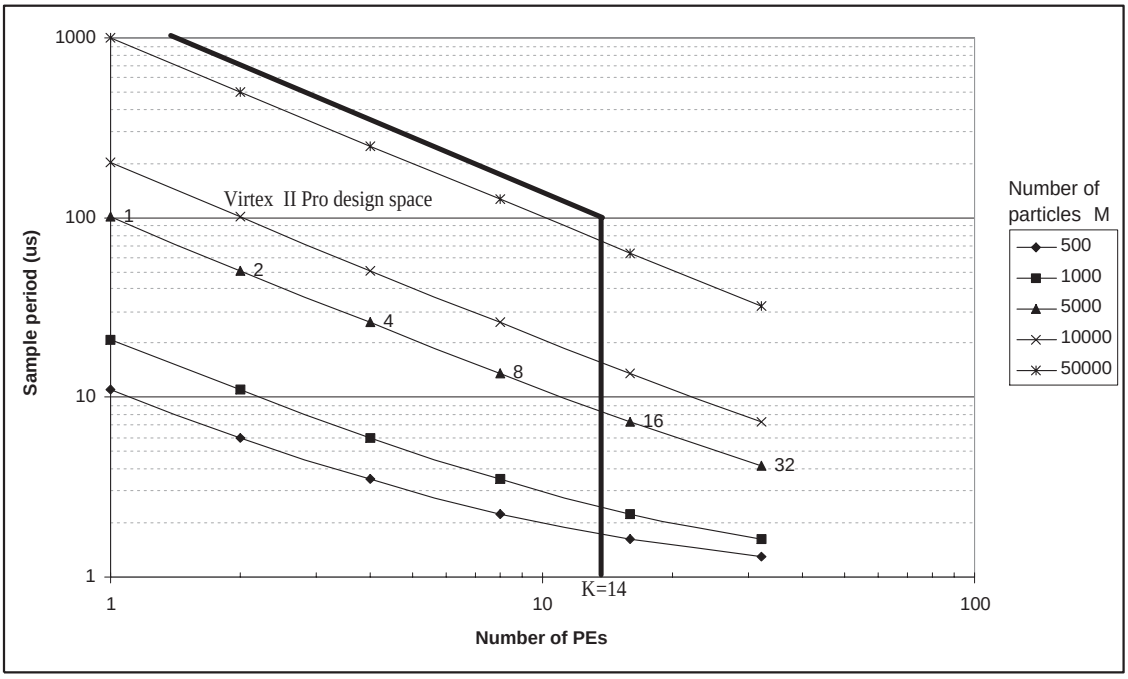
In Figure 5.9(a) we present the execution time as functions of K. The latency and the clock period that are used are $L_p = 100$ and $T_{clk} = 10ns$. The number of particles that the particle filter uses determines the execution time of the filter. The area of the graph bounded by the bold line represents the design space area for the Virtex II Pro family. For smaller M, the design space is determined by the logic blocks which increases with the level of parallelism, and for large M by the memory size.

Area/level of parallelism	1	2	4	8	16	32
Memory bits for M=10000 (Mbits)	1.53*	1.62*	1.94	2.3	2.59	5.1
Multiplier blocks	10	20	40	80	160	320
Number of slices (Kslices)	4	8	16*	32*	64*	128*
Xilinx chip that fits the design	XC2VP20	XC2VP30	XC2VP40	XC2VP70	-	-

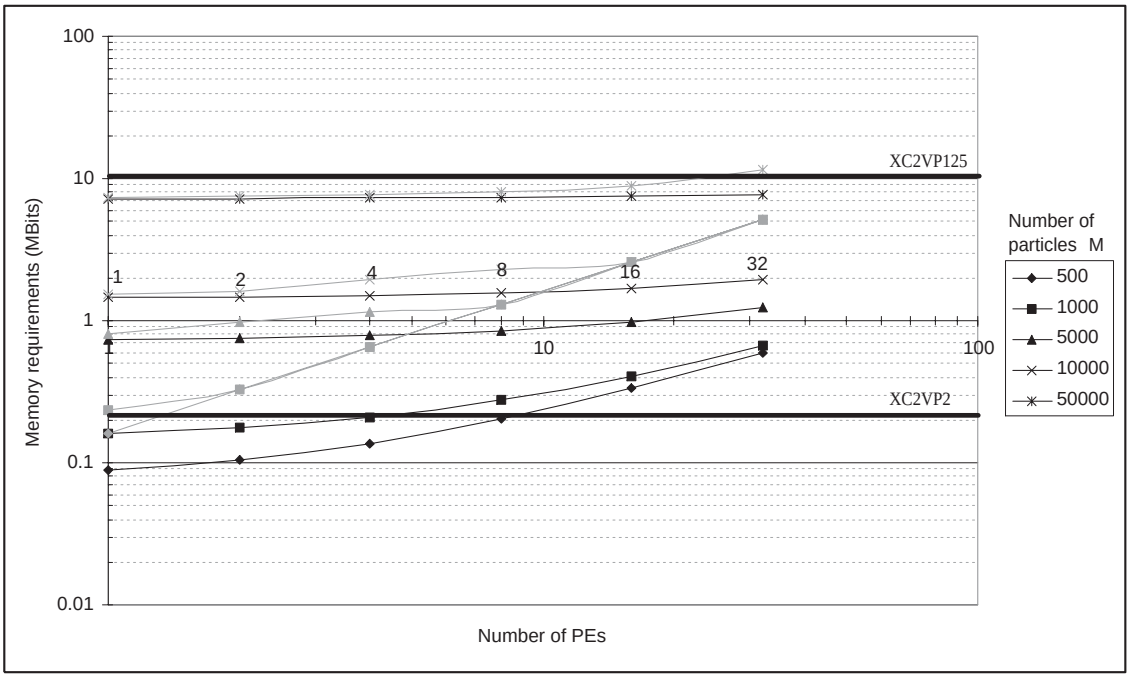
Table 5.2: The number of memory bits, slices and block multipliers for the distributed particle filter implementation with RNA with local exchange. The Virtex II Pro chips that can be fitted by the particle filter parameters are listed. The star shows which parameter determined in choosing the chip.

In Figure 5.9(a), M represents the number of particles used when the particle filter is running, and in Figure 5.9(b) M is the overall size of the memory in the design. So, $M = 1000$ means that the implementation supports only particle filters that use 1000 or less particles. In the distributed implementation, M particles are implemented using M/K memory banks. In the Figure, memory requirements are analyzed for two different topologies. With the black line, memory requirements are represented for the case when we can choose memories of the exact size. With the grey line, the memory requirements for Xirtex II Pro are shown. The design space for Virtex II Pro devices is below the bold lines, which represent available block RAM in the smallest (XC2VP2) and in the largest (XC2VP125) Virtex II Pro devices. For example, the design space of XC2VP2 is below the bottom bold line. In Virtex II Pro FPGA, the memories are implemented using coarse 18 Kbits block RAMs. For example, although 1000 of 16 bit weights can fit into one block RAM memory, when $K > 1$ they occupy K block RAM modules. In this case, utilization of the block RAM modules is low because only a portion of them is occupied. For example, for $K = 8$, the same amount of block RAMs is necessary for all $M < 5000$, so that the curves for different M are overlapped. The approximate number of block RAM modules is calculated as $\lceil BM/(KS) \rceil KS$, where B is the number of bits and S is the size of block RAM memory which is 18Kbit. In the figure, we also consider that the Box-Muller generator is implemented using one block RAM module, and we use two random number generators in parallel per PE. Additional memory (due to the usage of memory for random number generators) especially contributes to the overall number of bits when M is low ($M < 1000$) and K is large ($K \geq 16$).

It is interesting to compare the number of memory slices, number of multiplers and the number of bits with the corresponding values from the Virtex II Pro family, which is shown in Table 5.2. In the table, the number of particles is $M = 10000$. The number of slices for components in the dataflow is calculated and is multiplied by the factor of 1.5 in order to take into account the controllers and unused slices. The '*' represents which parameter of the memory, number of slices or multipliers blocks determines the choice of the Xilinx chip. In the same table, the corresponding Xilinx chip is shown as well. For a lower level of parallelism ($K \leq 4$), the design is memory dominated, while for a higher level of parallelism ($K > 4$), it is logic dominated. The design with $K > 14$ PEs cannot fit into the commercial Virtex II Pro FPGAs.



(a)



(b)

Figure 5.9: (a) Execution time and (b) memory requirements versus the number of PEs for RNA with local exchange for the SIRF with M =500, 1000, 5000, 10000 and 50000 particles.

5.5 Final remarks

In this section, two methods for distributing the resampling step suitable for distributed real-time FPGA implementation are proposed. The practical guidelines for choosing the resampling method depend primarily on the desired performance, communication pattern and complexity of the CU.

SIRF performance of centralized resampling and the RPA algorithm are the same as the sequentially implemented SIRF. However, there are no advantages in using centralized resampling since the RPA algorithm is faster and has a simpler CU. On the other hand, the RNA algorithm trades SIRF performance for speed improvement. So, RPA algorithm is a good choice when it is necessary to preserve performance, but with significant increase in complexity.

Communication pattern in the RPA algorithm is non-deterministic. As such, it requires point-to-point network to achieve the minimum execution time. The RNA algorithm can also achieve minimum execution time, but its architecture consists only of local connections. The communication pattern of the RNA algorithm with regrouping is somewhere in between the RNA algorithm with local exchange and the RPA algorithm. If the size of the group is larger than two, the RNA algorithm with regrouping also suffers from a non-deterministic communication pattern. However, the amount of particles that have to be exchanged inside groups is smaller than for the RPA algorithm.

The complexity of the CU of the RPA algorithm is very high since it has to implement a complex routing protocol through point-to-point network. The CU of the RNA algorithm with local exchange is simple and is not responsible for particle routing after resampling. The RNA algorithm with regrouping has to have control units in every group when groups contain more than two PEs. So, when speed is important and when it is required that design time is low (low complexity of the CU and of the scheduling and protocol in interconnection networks) the RNA algorithm with local exchange is the preferred solution.

Chapter 6

Architectures for Gaussian particle filters

In this chapter, we analyze algorithmic and architectural characteristics of Gaussian Particle Filters (GPFs) [15]. In comparison with the traditional SIRFs, GPFs show a great potential for high-speed implementation through exploiting operational concurrency. Moreover, the GPFs are inherently suitable for parallel implementation with multiple PEs and a single CU. We use these characteristics to propose a modified GPF algorithm that is suitable for parallel hardware realization. With the modified algorithm, there is no need for storing particles in memories between successive recursions. This property is of particular interest in developing flexible architectures for particle filtering, because the memory size is not a constraint for the number of used particles. Although the computational complexity of GPFs is higher than that of SIRFs, GPFs can be implemented without the resampling procedure. This entails that it has higher concurrency that can be exploited for greater throughput.

This chapter is organized as follows. Section 6.1 presents the proposed modifications of GPFs and an analysis of the algorithmic complexity of GPFs in terms of temporal and spatial concurrency. Section 6.2 discusses implementation issues for sequential and concurrent implementations of GPFs. A data flow analysis and high level architecture characterization is given in this section. A lower level comparison between SIRFs and GPFs in terms of resource utilization and speed is presented in Section 6.3.

6.1 Algorithmic modifications and complexity characterization

6.1.1 Temporal concurrency

GPF algorithms are presented in Pseudocode 4 and Pseudocode 5. It is observed that the GPF contains four loops of M iterations, where each loop is used for calculation of one step in Pseudocode 4. 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 [14]. However,

all four steps have the same number of iterations and as such they are suitable for loop fusion [101].

Steps 1, 2 and 3(a) in Pseudocode 4 can be easily fused. Weight normalization (step 3(b)) requires that all the weights are known in order to form the sum of the weights and as such is not appropriate for loop fusion. However, we can modify the algorithm so that normalization is not necessary by using non-normalized weights for calculating the mean and covariance coefficients. Of course, the mean and covariance coefficients must be scaled in the end with the sum of weights W_n. Step 4(b) cannot be fused in its original form since the mean is not known during the computation of the covariance coefficients until all of the M particles are processed. However, this step can be rewritten as follows:

$$\mathbf{\Sigma}_n = \frac{1}{W_n} \sum_{m=1}^{M} w_n^{(m)} \mathbf{x}_n^{(m)} \mathbf{x}_n^{\top(m)} - \boldsymbol{\mu}_n \boldsymbol{\mu}_n^{\top}. \tag{6.1}$$

The second term on the right is constant, and it can be calculated outside the loop. The first term of the modified step 4(b) can be fused with the previous three steps. The fused steps are presented in Pseudocode 12. Note that the particles $\mathbf{x}_n^{(m)}$ do not need to be saved, which is advantageous for hardware implementation. There is additional processing outside the main loop that is still necessary (Pseudocode 13) and it involves: final computation of the mean and covariance coeficients (step 2), calculation of the final covariance coefficients using (6.1) (step 3) and Cholesky decomposition (step 4). Cholesky decomposition is needed because Gaussian random number generation in step 1 of Pseudocode 12 utilizes $\mathbf{C}_n$ which is the square root of the covariance matrix. In non-parallel implementations (i.e., with a single PE), step 1 in Pseudocode 13 is not necessary.

6.1.2 Spatial concurrency

Spatial concurrency can be exploited for developing parallel architectures by mapping independent operations to multiple hardware units that operate in parallel. Spatial concurrency is exploited for multiple PEs execution.

It can be seen from the GPF algorithm that the sample and weight calculation steps for each of the M particles are independent of the other particles. The operations of each iteration of the loop in Pseudocode 12 are independent. Moreover the loop does not have loop-carried dependence [53]. Thus the loop level parallelism can be exploited for increasing throughput. This is done by having multiple independent PEs each processing a fraction of the total number of particles. The number of PEs, K, defines the degree of parallelism. The maximum degree of parallelism is obtained when each PE consists of only one particle ($K = M$).

Dependence exists among the iterations during the mean and covariance estimation steps. To parallelize this step, partial sums of the mean and covariance ($\boldsymbol{\mu}_n^k$ and $\mathbf{\Sigma}_n^k$) are calculated by PEs and then added in the end in the CU (step 1 of Pseudocode 13). In the pseudocode, quantities with subscript k represent the result of operations in the $k-$th PE. During the final addition of partial sums, the PEs communicate with the CU and the amount of data transferred corresponds to the dimension of the state space model.

In summary, with the modified algorithm, we perform in the PEs concurrent operations

Input: The observation $\mathbf{z}_n$ and previous estimates $\boldsymbol{\mu}_{n-1}$ and matrix $\mathbf{C}_{n-1}$ s.t. $\boldsymbol{\Sigma}_{n-1} = \mathbf{C}_{n-1}\mathbf{C}_{n-1}^{\top}$
For $n = 1$, mean $\boldsymbol{\mu}_0$ and covariance $\boldsymbol{\Sigma}_0$ are based on prior information.
Setup: Sum of weight, mean and covariance elements in the $k-$th PE: $W_n^k = 0$, $\boldsymbol{\mu}_n^k = 0$, $\boldsymbol{\Sigma}_n^k = 0$.
Method:

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

1. Draw a conditioning particle from $\mathcal{N}(\mathbf{x}_{n-1}; \boldsymbol{\mu}_{n-1}, \boldsymbol{\Sigma}_{n-1})$ to obtain $\mathbf{x}_{n-1}^{(m)}$.
2. Draw a sample from $p(\mathbf{x}_n \mid \mathbf{x}_{n-1}^{(m)})$ to obtain $\mathbf{x}_n^{(m)}$.
3. (a) Calculate a weight by $w_n^{*(m)} = p(\mathbf{z}_n|\mathbf{x}_n^{(m)})$.
   (b) Update the current sum of weights by $W_n^k = W_n^k + w_n^{*(m)}$.
4. Update $\boldsymbol{\mu}_n^k$ and $\boldsymbol{\Sigma}_n^k$ by
   (a) $\boldsymbol{\mu}_n^k = \boldsymbol{\mu}_n^k + w_n^{*(m)}\mathbf{x}_n$
   (b) $\boldsymbol{\Sigma}_n^k = \boldsymbol{\Sigma}_n^k + w_n^{*(m)}\mathbf{x}_n^{(m)}\mathbf{x}_n^{(m)\top}$.

end

Pseudocode 12 Part of the GPF algorithm that runs in parallel on PEs after loop fusion is applied. The symbol k denotes the $k-$th PE and K denotes the total number of PEs.

Input: W_n^k, $\boldsymbol{\mu}_n^k$ and $\boldsymbol{\Sigma}_n^k$ for $k = 1, ..., K$.
Setup: Sum of the weights $W_n = 0$, initial mean $\boldsymbol{\mu}_n = 0$ and covariance $\boldsymbol{\Sigma}_n = 0$.
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) $\boldsymbol{\mu}_n = \boldsymbol{\mu}_n + \boldsymbol{\mu}_n^k$
   (c) $\boldsymbol{\Sigma}_n = \boldsymbol{\Sigma}_n + \boldsymbol{\Sigma}_n^k$
   end
2. Scale mean and covariance
   (a) $\boldsymbol{\mu}_n = \boldsymbol{\mu}_n / W_n$
   (b) $\boldsymbol{\Sigma}_n = \boldsymbol{\Sigma}_n / W_n$
3. Compute the covariance estimate $\boldsymbol{\Sigma}_n = \boldsymbol{\Sigma}_n - \boldsymbol{\mu}_n\boldsymbol{\mu}_n^{\top}$
4. Compute the Cholesky decomposition of the matrix $\boldsymbol{\Sigma}_n$ in order to obtain $\mathbf{C}_n$.

Pseudocode 13 Part of the GPF algorithm that runs sequentially on the CU and exchanges data with K PEs.

on the particles whereas in the CU we carry out sequential post processing.

6.1.3 Computational complexity characteristic

A comparison of number of operations and memory requirements for one recursion of SIRFs and GPFs is shown in Table 6.1. The number of operations for particle generation and weight calculation is not presented because it is the same for both filters. Additional complexity for GPFs is added due to generation of conditioning particles and computation of the mean vector and covariance matrix of the states. We would like to stress here that the complexity of these steps is related to the dimensionality of the model N_s as $O(N_s^2)$. For the GPF, the number of multiplication operations for the computations of the mean vector and covariance matrix is equal to $N_s(N_s+1)$, whereas for the SIRF, the number of multiplications for calculation of the mean is N_s. On the other hand, the memory requirements of SIRFs are dominant. They increase with the increase of the dimension of the model as shown in [30]. In the table, (N_s+2) data per particle are used for storing N_s states, a weight and an index that is a result of the resampling process.

Algorithms	Gaussian random number generator	Multiplication operations		Memory requirements
		Drawing conditioning particles	Computation of estimate	
GPF	$6N_sM$	$N_s(N_s+1)M/2$	$N_s(N_s+1)M$	0
SIRF	$2N_sM$	0	N_sM	$(N_s+2)M$

Table 6.1: Comparison of the number of operations and memory requirements of SIRFs and GPFs in a PE. One sampling period of particle filter is analyzed. The operations that are the same in the particle generation and weight calculation steps are not considered.

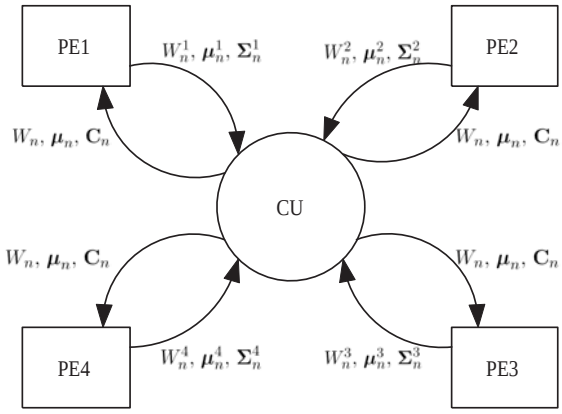


Figure 6.1: Parallel GPF model with $K = 4$ PEs.

To present the amount of data exchange required for SIRFs and GPFs, we consider SIRFs with the RPA alogtihm described in Section 5.2 and in [12]. In both filters, operations without data dependencies are mapped to the PEs in order to allow for parallel processing. In GPFs, these are the four operations described by Pseudocode 12. In SIRFs with parallel resampling, the particle generation, weight calculation and partial resampling are mapped to the PEs. The parallel architecture for GPFs that consists of four PEs and the CU is shown in Figure 6.1.

For an N_s dimensional model, the amount of data acquired by the CU from each of the K PEs is: $N_s(N_s+1)/2 + N_s + 1 = (N_s^2 + 3N_s)/2 + 1$, where $N_s(N_s+1)/2$ is the number of data in the covariance matrix $\mathbf{\Sigma}_n^{(k)}$ (which is symmetric), N_s is the number of data in the mean vector $\boldsymbol{\mu}_n^k$, and 1 corresponds to the sum of weights W_n^k. The CU sends back to the PEs the calculated mean $\boldsymbol{\mu}_n$ and the Cholesky factorization $\mathbf{C}_n$ of the final covariance matrix. This is an additional $N_s(N_s+1)/2 + N_s$ data. Since, the same data are sent to each PE, it is beneficial to use mechanisms that allow for simultaneous transfer of data from the CU to all the PEs. For the bearings-only tracking problem $N_s = 4$, so the number of data sent from each PE to the CU is 15 and the number of data sent from the CU back to the PEs is 14. This data exchange is depicted in Figure 6.1. Overall, however, the number of data that are transferred through the interconnection network is much smaller than in the case of SIRFs, and it is fixed over time. This feature greatly increases the scalability [53] of the filter and also ensures that data exchange is never the dominant operation of GPFs. In contrast, in the worst case of SIRFs $M/2$ particles are sent over the interconnection network [12].

Parameter	**Effects on Algorithmic Parameters**		**Effects on Architectural Parameters**	
	SIRF	GPF	Sequential implementation	Spatial implementation
Number of operations	Linear increase	Quadratic increase	Sample period	Area
Number of particles	Increase with model dimension as in [30]		Sampling period and memory requirements	Sampling period and area
Data exchange	Proportional to M	Proportional to N_s^2	Sampling period	Sampling period
Finite precision word length	Increase with the number of operations		Sampling period	Sampling period and area

Table 6.2: The effect of model increase on algorithmic and architectural parameters.

The complexity characteristics of SIRFs and GPFs are summarized in Table 6.2. In the table, the effects of the increase in model dimension on different particle filter parameters are shown together with their resources requirements. The summary of the effects is as follows:

1. For GPFs, the number of operations per particle increases quadratically with model dimension. This significantly affects the complexity of the units for drawing conditioning particles and covariance estimation as well as the complexity of the CU. For SIRFs, the number of operations increases linearly with model dimension.
2. The number of particles needed to achieve a required accuracy increases with model dimension and that affects the sampling periods of both SIRFs and GPFs. However, SIRFs are more affected since there is an additional time for accessing memories. There is a significant area increase in the spatial implementation of SIRFs, because physical memories are necessary to store particles and weights.
3. For GPFs, data exchange requirements increase quadratically with model dimension, but the amount of data that is transferred between the PEs and the CU is several orders of magnitude lower than that for SIRFs. Besides, the data exchange pattern of GPFs is deterministic.
4. For GPFs, the complexity of mathematical operations increases resulting in a very large word length for finite precision processing. In such cases, floating point implementation is the more feasible option which implies requirements for increased area and/or increased sampling period. The finite precision processing of SIRFs is less affected by increase in model dimension.

6.2 Implementation issues

6.2.1 Sequential processing

In sequential processing that is applied on DSPs, concurrency is only exploited by the compiler, and it depends on the number of arithmetic units within the processors. The computation complexity is mainly dominated by the number of mathematical computations and memory access. A sequential implementation of SIRFs is faster because their mathematical operations are fewer and simpler than the operations of GPFs. To illustrate this, we consider the bearings-only tracking problem where the model dimension is four and only one DSP is used. Figure 6.2 shows two curves that correspond to the execution times for processing M particles using the SIR and GPF algorithms. The curves represent the sampling period as a function of number of particles (for the Analog Devices floating point DSP TigerSharc ADC-101S). Similar results are obtained from the implementation on Texas Instruments processors. In sequential implementations, the sampling period increases almost linearly with the number of particles for $M = \{500, 1000, 5000\}$.

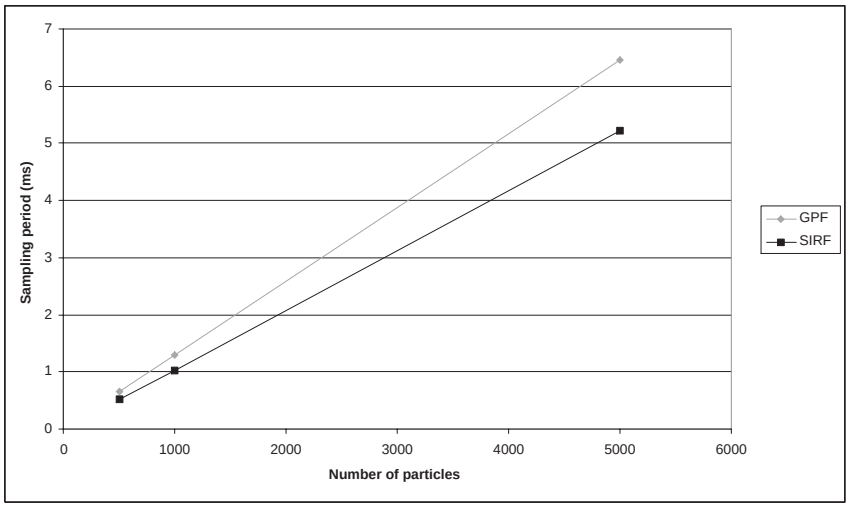


Figure 6.2: Sampling period of GPFs and SIRFs versus number of particles. The filters are implemented on Analog DSP TigerSharc ADC-101S.

From the figure, for moderate number of particles, we deduce that SIRFs can achieve faster sampling periods than GPFs. When the number of particles is large, the DSP cannot satisfy the high throughput constraint and multiple processors must be utilized. It is realistic to assume that in multiprocessor configurations, the transfer of one particle through the interconnection network takes more time than the clock period. This time would increase if we use shared memory. Thus data exchange through the interconnection network would present a bottleneck for SIRFs, which makes the GPFs better candidates for multiprocessor applications. GPFs have higher scalability than SIRFs and are also flexible in terms of the maximum number of used particles.

6.2.2 Concurrent processing

Temporal and spatial concurrency can be exploited with ASIC or FPGA implementations. On such platforms, particle filters can be executed with dedicated operators and through duplication of hardware. Dataflow for parallel GPF implementation is presented in Figure 6.3. It is important to note that each processing logic element of the GPF is on the critical path. For the SIRF, on the other hand, the output calculation is not on the critical path and can be done in parallel with the sample step.

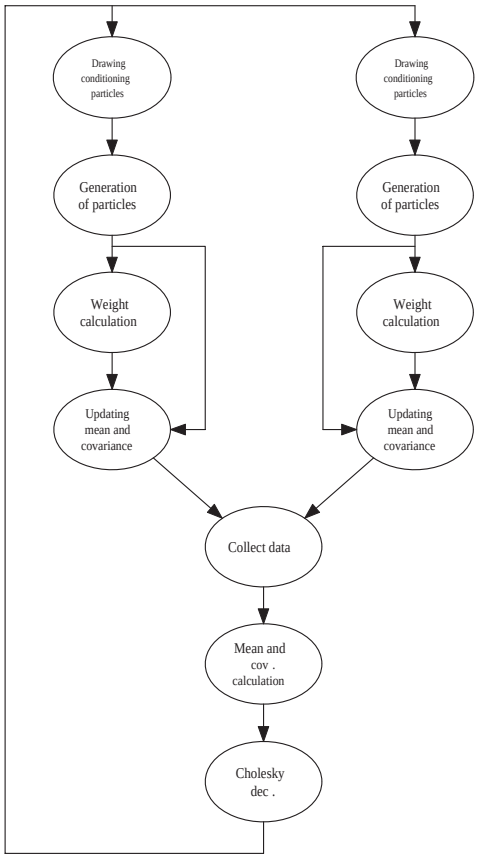


Figure 6.3: Dataflow of parallel GPF.

Figure 6.4 shows the timing diagram with the latencies of various operations of GPFs including the data transfer. The outputs of the first three logic blocks of the data flow are generated at clock speed, while the output of the updating mean and covariance block (step 4 in Pseudocode 12) and the output of the CU are generated at the particle filter sampling speed. The minimum sampling period that can be achieved with parallel GPFs for different number of particles and processing elements is presented in Figure 6.5. For SIRFs whose resampling is distributed to the PEs, the minimum sampling period that can be achieved is $(\frac{2M}{K} + L_{SIR} + L_{dex}(M)) \cdot T_{clk}$, where $\frac{2M}{K} + L_{SIR}$ is the latency of processing in the PEs and the $L_{dex}(M)$ is the latency of the communication after particle allocation.

In the case of GPFs, the minimum sampling period is $(\frac{M}{K} + L_{GPF} + C) \cdot T_{clk}$, where, L_{GPF} is the net latency of the critical path. The latency $L_{GPF} = \sum_{i=1}^{3} L_i$ accounts for the start up

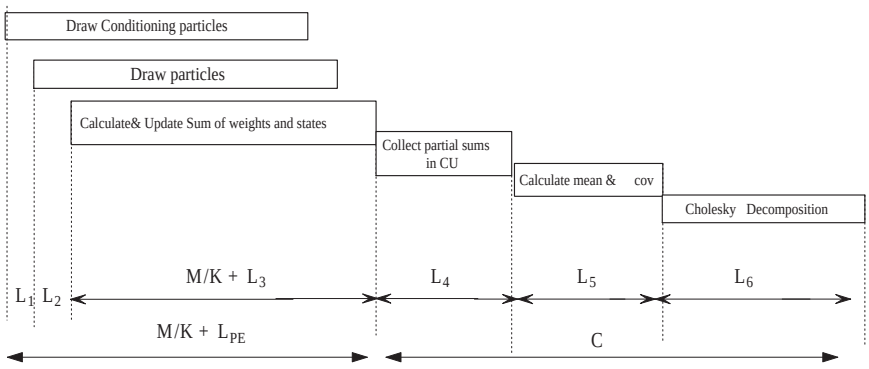


Figure 6.4: Timing diagram for GPF.

latencies of the various blocks inside the PEs. The constant term $C = \sum_{i=4}^{6} L_i$ accounts for the latency of both the CU and the data exchange between the PEs and the CU. Thus, we see that though GPFs have a larger constant latency of the CU, for large number of particles the latency of the GPFs will be smaller than that of SIRFs. This is primarily because for SIRFs resampling is required which is not only sequential and dependent on the result of processing *all* the particles in the PE, but also requires particle redistribution that has an execution time proportional to M. In our simulations, we assume that the clock period is equal for both filters $T_{clk} = 10$ns and that $L_{GPF} = 3 \cdot L_{SIR} = 300$ and $L_{dex}(M) = 0$ for SIRF. In Figure 6.5, the minimum execution time for SIRFs is presented by black line and for GPFs by gray line. The large total latency affects the scalability of GPFs when M is small. On the other hand, when M is large, the sampling period of GPFs is almost twice smaller than the one of SIRFs. Hence GPFs are appropriate for high speed applications that require large number of particles on platforms that have enough resources for spatial parallel implementation.

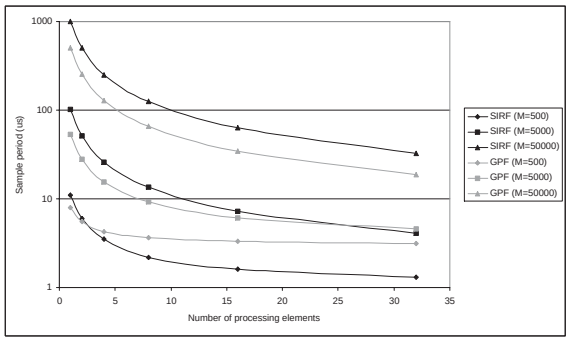


Figure 6.5: Minimum sampling period versus number of PEs of parallel GPFs and SIRFs for $M = \{500, 5000, 50000\}$. Spatial implementation of particle filters is assumed.

6.2.3 Architectures and resource requirements

We consider spatial implementations of GPFs applied to bearings-only tracking [49], with a one-to-one mapping between each operation and the hardware resource. A 16-bit representation of the particles is chosen. The mean square error (MSE) of tracking is used as

a performance evaluation criterion for fixed precision analysis, where the error due to finite precision arithmetic is kept within the limits of $\pm 10\%$ of the floating point value. Our simulations indicate that the steps $1-3$ in Pseudocode 12 are less sensitive to finite precision effects. The MSE within defined limits is achieved using 16-bit representation for the operations of these steps. However, step 4 in Pseudocode 12 and all the steps of Pseudocode 13 are very sensitive to finite-precision effects. One of the operations is Cholesky decomposition which requires that the input matrix is a positive definite, a condition that may not be satisfied for representations which use a small number of bits. In order to alleviate this problem, 40 bits of precision are necessary for operations in step four of Pseudocode 12. Similarly, all the operations that are executed in the CU (Pseudocode 12) require more than 50 bits.

The GPF has 5 major computational units: generation of conditioning particles, particle generation, weight computation, mean and covariance calculation, and central unit.

Overall operation:

The block diagram of the GPF is shown in Figure 6.6. The GPF works in a way that the operation of the CU is dependent on the result of the operation of the PEs and vice versa. Hence the PEs and CU cannot operate simultaneously. Once all M particles are processed, the PEs send the computed mean, the covariance matrix, and the sum of weights ($\boldsymbol{\mu}_n, \boldsymbol{\Sigma}_n$, W_n) to the CU. The CU starts its operations after receiving all the data from PEs, and PEs remain idle during this time. Once the operation of the CU is complete, the results are sent to the PEs and the next recursion is started. The particle generation step comprises taking the conditioning particles and producing final particles, which are the inputs for the steps of weight computation and updating of the mean and the covariance matrix. The weight computation step takes also input observations and computes the 16-bit weights. In this step, the calculation of the exponential and arctangent functions is attained by using Coordinate Rotation Digital Computer (CORDIC) algorithms [110]. The output and the internal operations of the updating of the mean and covariance matrix are represented using 40 bits. Since there are 14 40-bit outputs which are generated once in a recursion, we assume a single 40-bit bus that connects the PEs and the CU.

The three blocks, generation of conditioning particles, particle generation, and weight computation, consume and produce M particles. Block updating the mean and covariance (MCC) consumes M particles while producing $\boldsymbol{\mu}_n$ and $\boldsymbol{\Sigma}_n$ only once during the sampling period. Buffering is necessary because the MCC block can start computing the mean and covariance coefficients only after the corresponding weights are computed in the weight computation block. Hence, the latency of the buffer is equal to the latency of the weight computation block.

Generation of conditioning particles (GCP): In this step, the decomposed covariance matrix $\mathbf{C}_n$ and the mean $\boldsymbol{\mu}_n$ obtained from the CU are used for generation of conditioning particles. The matrix $\mathbf{C}_n$ is a triangular $4x4$ matrix, so that the number of multiplications in this step is 10. All the multipliers are pipelined and they operate concurrently producing M conditioning particles in $M+L_1$ clock cycles. Here, L_1 is the latency of one multiplier and four adders. Since the outputs $\{x^{(m)}, V_x{}^{(m)}, y^{(m)}, V_y{}^{(m)}\}_{m=1}^{M}$ are computed using a different number of operators, we have to introduce additional delay which is different for each state in order to obtain all the conditioning particles at the same time instant at the output.

This step requires 4 random number generators (RNGs). These RNGs produce Gaussian

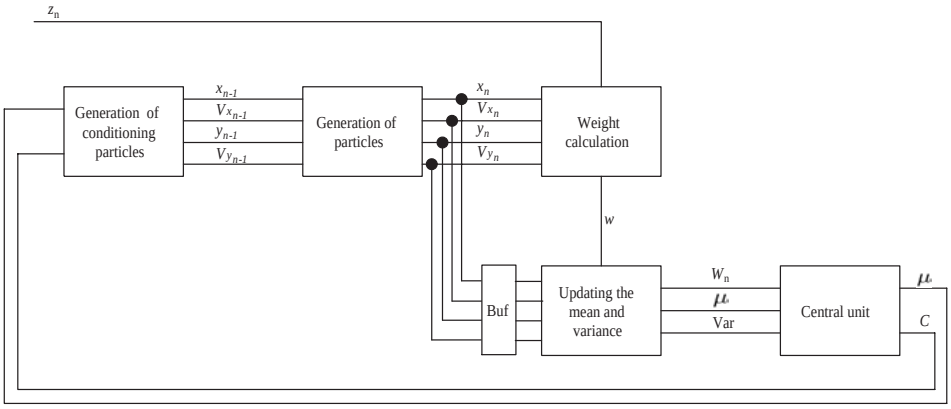


Figure 6.6: Block diagram of GPF.

random numbers with properties required by the model. The RNGs are based on a look-up table approach detailed in [46]. The block diagram of the GCP step is shown in Figure 6.7.

Initially, the mean and the decomposed covariance elements are obtained from the input pins and not from the CU. This requires additional control and a multiplexor at the input of this block.

Particle generation, weight computation: Typical operations processed by the *particle generation* and *weight computation* steps are identical to those of the SIRF except that particles do not need to be stored in memories. The architectures for these steps are presented in Section 4.3.3.

Mean and Covariance Calculate (MCC): In the MCC step, the partial covariance 4x4 matrix $\mathbf{\Sigma}_n$ and 4x1 mean vector $\boldsymbol{\mu}$ are calculated (Figure 6.8). The number of multiplication operations is equal to $N_s + N_s(N_s+1)/2 = (N_s^2 + 3N_s)/2$ where N_s represents the dimension of the model. For the bearings-only tracking problem with $N_s = 4$, 14 multiplications are required. All 14 outputs are accumulated, so that 14 accumulators are necessary. All the blocks operate concurrently on M particles in $M + L_3$ clock cycles. The latency of the critical path L_3 consists of the latencies of two multipliers and an accumulator.

Central Unit (CU): The inputs and the outputs of the CU are produced once during the sampling period. The output $\boldsymbol{\mu}_n$ is the output of the overall particle filter, while $\boldsymbol{\mu}_n$ and $\mathbf{C}_n$ are used as inputs for the GCP block in the next particle filtering recursion. The sum of the particles, W_n, is used in the CU because the mean and covariance elements in the MCC are computed using non-normalized weights and these elements have to be properly scaled in the CU. Since all the operations in the CU are done only once in a sampling period, time multiplexing is performed in order to preserve hardware resources (Figure 6.9). So, only one divider is used for scaling the mean and covariance elements (step 2 of Pseudocode 13) and one multiplier for adjusting the covariance elements (step 3 of Pseudocode 13). The number of divisions is $(N_s^2 + 3N_s)/2 = 14$ because both the mean and the covariance matrices need scaling with the sum of weights. Adjusting the covariance elements requires 10 addition and 10 multiplication operations. Cholesky decomposition is presented as a block with 10 inputs and outputs. It performs complex operations such as square roots and divisions. Since processing

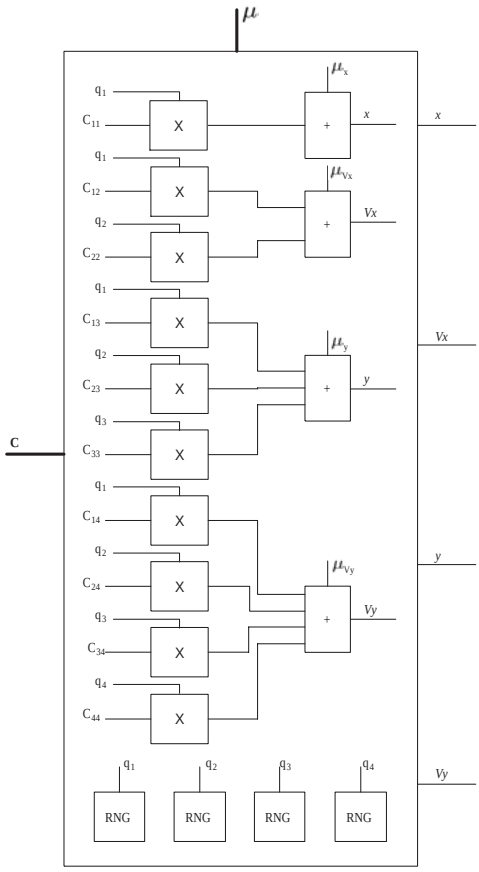


Figure 6.7: Block diagram of generation of conditioning particles.

of each operation depends on the result of the previous one, Cholesky decomposition is also suitable for time-multiplexing.

6.3 Comparisons and tradeoffs between SIRFs and GPFs

6.3.1 Energy with speed constraints

The modeling of high-level energy is performed on a module level by estimating the power functions of the elements associated with each module such as adders, multipliers, registers or memories [85, 97]. The power for each module is initially estimated using the Xilinx Spreadsheet Power Tools and verified using Xilinx XPower.

The energy of a single PE implementation of SIRFs and GPFs calculated for a particle filter sampling period is shown in Figure 6.10 (a) and (b), respectively. The filters are applied to the bearings-only tracking problem. The energy is estimated for various number of particles ($M = \{500, 1000, 2000, 5000, 10000\}$) and for various maximum sampling frequencies ($f_s = \{1, 5, 10, 50\}$kHz). The minimum depth of pipelining of the functional blocks that satisfies a given sampling frequencies is calculated and applied in order to reduce energy. The number of bits used is 16 for all SIRF variables and steps 1-3 of Pesudocode 2 for the GPF. Step 4 of

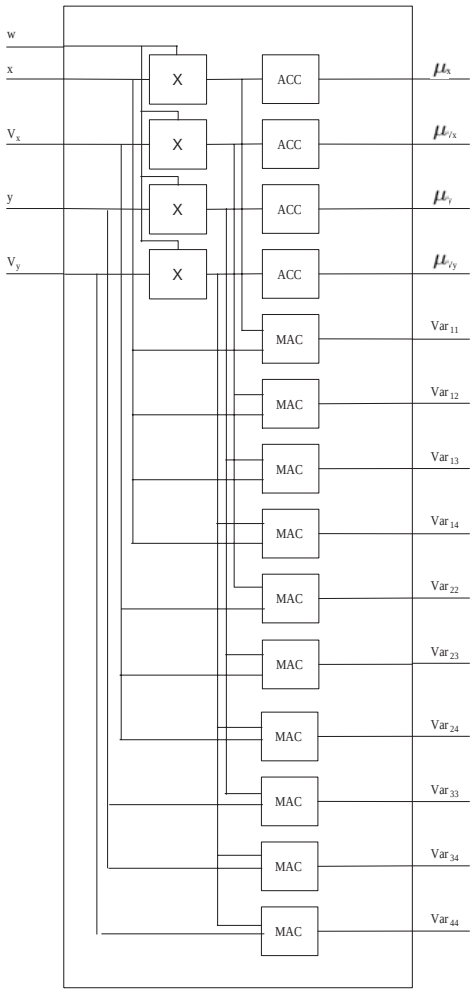


Figure 6.8: Block diagram of the mean and covariance computation step.

Pseudocode 2 and all the steps in Pseudocode 3 are represented in fixed point with 40 bits. The operating clock frequency is defined by the speed of the CORDIC which is the slowest individual unit in the data path. For our analysis, the clock frequency is set to 100 MHz. From Figure 6.10, it is clear that for smaller number of particles and lower frequencies, SIRFs dissipate less energy than GPFs. It is important to note that with the increase of number of particles, the energy of SIRFs becomes comparable and even higher than the energy of GPFs (for more that 14,000 particles). There are two reasons for faster increase of the energy in SIRFs. Since SIRFs are memory dominant, with the increase of number of particles, the size of memory increase results in higher energy. Secondly, the number of operations of the CU of SIRFs is a function of M, while for GPFs the number of operations of the CU is a constant, so that the energy of the CU of SIRFs increases linearly and the energy of GPFs stays constant.

SIRF implementations with a single PE cannot achieve higher requirements such as processing of $10,000$ particles at 5kHz. The energy for the parallel implementation of SIRFs and GPFs which utilize multiple PEs is presented in Figure 6.11. The energy of SIRFs is calculated for the worst case data exchange among PEs and the CU which corresponds to transferring $[M - M/K]$ particles. The data exchange is modeled using a shared memory. Again, we can see that the energy of SIRFs is lower than the energy of GPFs when the number of particles

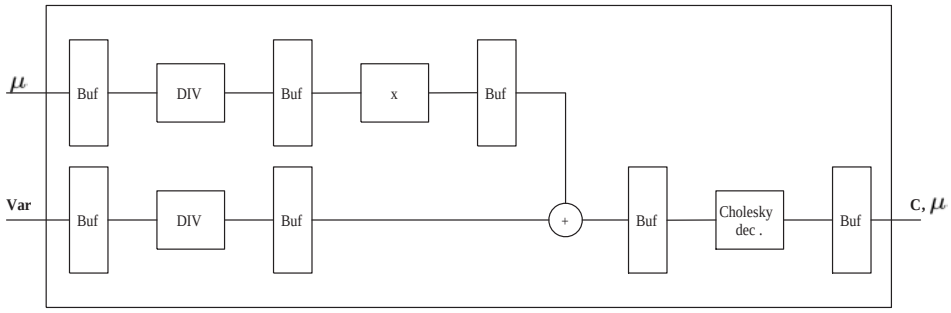
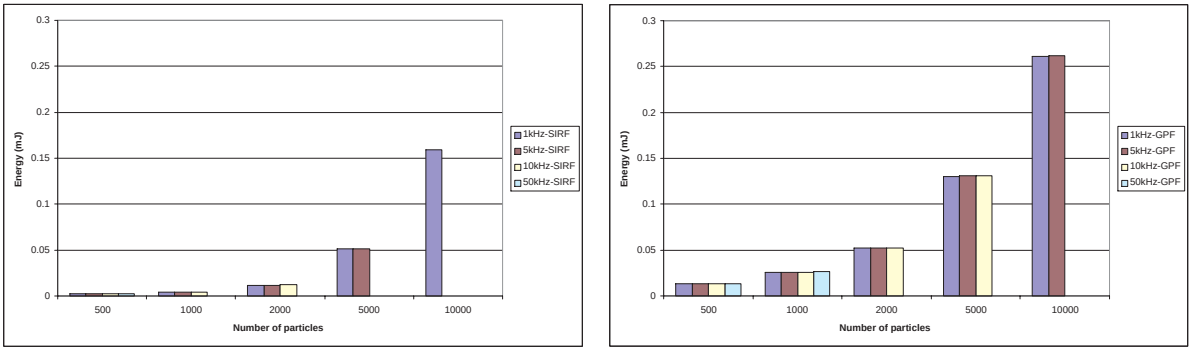


Figure 6.9: Block diagram of the time-multiplexed central unit for GPFs.



(a) SIRF energy

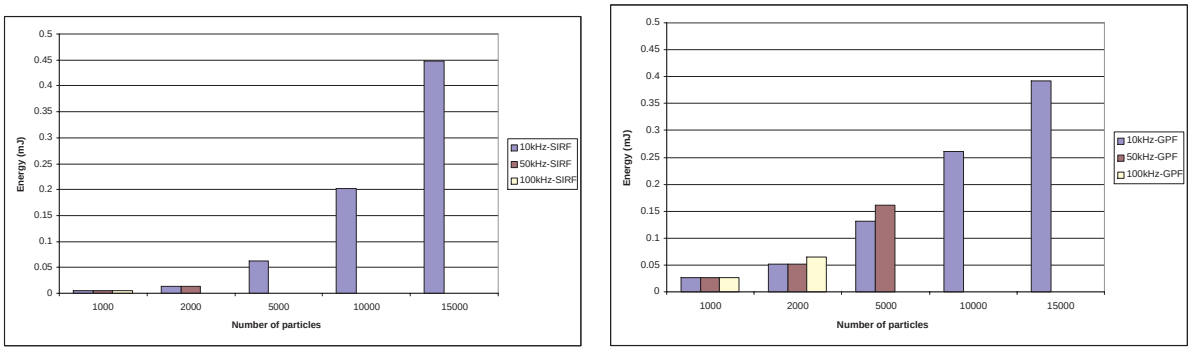
(b) GPF energy

Figure 6.10: Energy versus number of particles for SIRFs and GPFs with a single PE for different sampling rates.

is low. However, for $15,000$ particles and 1kHz sampling speed, the energy of SIRFs is higher than the energy of GPFs.

6.3.2 Area requirements in FPGA

The percentage of required resources for the hardware blocks of PEs for the GPF is shown in Figure 6.12. We used the Xilinx Virtex II pro FPGA platform [117] to estimate the resource and area requirements. The PEs calculate the covariance and mean coefficients in one clock cycle, and so a fully spatial design is used for them. The considered resources are the number of occupied slices and the number of used block multipliers. The only reason for domination of the step in which covariance matrix and mean are estimated, is the large number of bits used for fixed point representation. To be able to calculate four mean and 10 covariance coefficients per clock cycle, 14 multipliers are required. With a 40-bit representation, each multiplier occupies 9 multiplier blocks each with 18×18 bits on the Virtex II Pro chip. However, for lower dimensional models, the ratio of resources in hardware blocks will look different. For example, for two dimensional tracking models only three covariance coefficients and two mean coefficients are necessary for estimation, while the importance step is almost the same. In this case, the area of the importance step would dominate.



(a) SIRF energy

(b) GPF energy

Figure 6.11: Energy versus the number of particles for SIRFs and GPFs implemented with four PEs for different maximum sampling rates.

The operations that take place in the CU are sequential due to data dependency and have computationally intensive tasks such as square rooting, division, multiplication and addition. Thus, the CU is a good candidate for time-multiplexed implementation where hardware resources are shared in time by various operations. This implementation minimizes hardware without degrading execution throughput. We estimate that the overall sampling frequency of the GPF is reduced about 4% for time-multiplexed implementation in comparison to spatial implementation. However, area saving in the CU is about 90%.

With a very tight performance criterion, the necessary precision of the GPF's CU is more than 50 bits. This is because the coefficients of the covariance matrix are very small and their truncation may entail violation of the positive definiteness of the covariance matrix. If there is a violoation, the matrix cannot be decomposed and hence the recursion cannot proceed. So, when the CU is realized with the floating-point library [31] of Xilinx II Pro for estimating the area and latency, the clock frequency of the slowest floating point block (divider) is twice less than the speed of the slowest synthesized fixed-point block using 50-bit precision. For a GPF implemented by a single PE with $M = 5000$ particles, the sampling frequency is decreased 1.0167 times, the latency is increased 1.3 times, and the logic area is increased 1.02 times in comparison with the area of the particle filter with a time multiplexed CU that uses 50-bit precision fixed-point arithmetics. Thus, the floating point implementation is an alternate solution when the maintenance of precision is the key issue. Then the throughput and area are not affected significantly.

We also evaluated and compared the area requirements of SIRFs and GPFs. Resource requirements are represented using the number of Virtex II Pro logic slices and block RAMs (Figure 6.13). Here, the area is evaluated for various number of particles and various sampling frequencies. The number of PEs is adaptively changed so that the particle filters meet the sampling frequency requirements. For example, for SIRFs with $M = 5000$ and $f_s = 1$ kHz we choose an implementation with 2 PEs because they are necessary to satisfy the requirements. The GPF algorithm does not contain memory for storing particles. However, it uses one block RAM per random number generator for implementing the Box-Muller method [33]. From Figure 6.13(a) and 6.13(b) it is clear that SIRFs are memory dominated and GPFs are logic

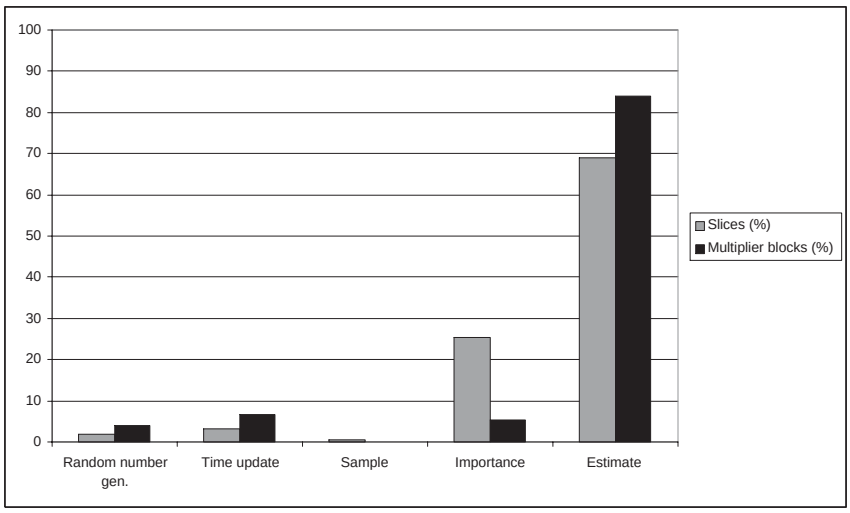
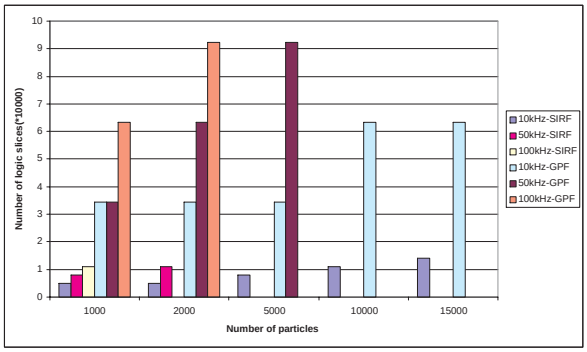
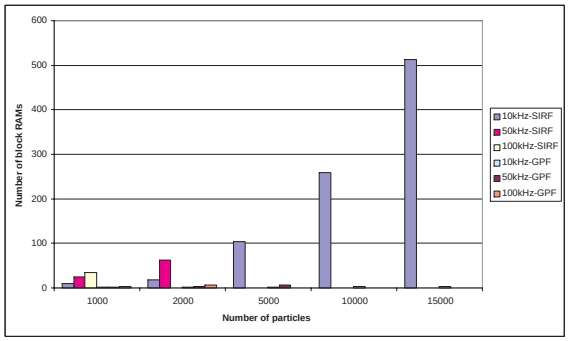


Figure 6.12: Percentage of number of slices and multipliers of each block in PEs estimated for Xilinx Virtex II Pro chips.

dominated.



(a) Area of SIRFs and GPFs



(b) Memory of SIRFs and GPFs

Figure 6.13: (a) Area in the number of slices and (b) number of block RAMs versus number of particles. The area is evaluated for different sampling frequencies and for necessary number of PEs so that the particle filters satisfy sampling frequency requirements.

Chapter 7

Conclusion

This chapter summarizes the principle ideas and contributions of this dissertation. As with most research efforts, the attempt to address one set of challenging problems has given rise to a whole new set of problems to be solved. This chapter starts with the summary of the dissertation followed by the directions for future research.

7.1 Summary

Despite amazing progress in the signal processing field, there are still many difficult scientific and engineering problems which cannot be resolved with traditional signal processing algorithms. Particle filters are among the latest innovations that attempt to bridge the existing gap between what was and was not doable in filtering theory until recently. They are applied in many fields including wireless communications, navigation systems, sonar, and robotics, and it has been shown that they outperform most of the traditional filters in complex practical scenarios. The particle filters are computationally very intensive which is their main drawback.

In this dissertation, physically feasible VLSI architecture for particle filters are developed with the emphasis on high speed design. Joint algorithmic and architectural design is adopted. There are two basic algorithmic challenges in the SIR algorithm: reducing the computational intensity of the algorithm and accelerating the process of resampling. We have proposed new resampling algorithms whose processing time is not random and that are more suitable for hardware implementation. The new resampling algorithms reduce the number of operations and memory access and/or allow for overlapping the resampling step with weight computation and particle generation. Even though the algorithms are developed with the aim of improving the hardware implementation, these algorithms could also be considered as resampling methods in simulations on standard computers because they reduce the execution time. Also, other particle filtering algorithms are considered (GPF) that do not require resampling, which makes them simpler for implementation.

Particle filtering algorithms are modified so that higher speed and lower memory requirements are achieved for the proposed architectures. Memory requirements and sampling period of the modified SIRFs and GPFs are shown in Table 7.1. Modified SIRFs contain two loops while the modified GPFs have only one. In spatial implementations where each operation

is mapped to its dedicated hardware block, the GPF is twice faster than the SIRF and has an execution time of MT_{clk} per recursion. GPF can also be implemented without the need for storing particles. Memory requirement and the number of memory access are reduced for the modified SIRF. This reduction is especially obvious for multi-dimensional models and it approaches two-fold reduction as the dimension increases. On the other hand, the computational complexity of the GPF is much higher. For example, it requires two times more random number generators and many more multipliers.

Parameter	Original SIRF	Modifies SIRF	Modified GPF
Sampling period	$5MT_{clk}$	$2MT_{clk}$	MT_{clk}
Particle and weights memory	$2N_sM$	$(N_s+2)M$	0
Memory access	$2(2N_s+1)M$	$2(N_s+3)M$	0
Computational complexity	Medium N_s RNG, 8 MUL	Meduim N_s RNG, 8 MUL	High $2N_s$ RNG, 30 MUL

Table 7.1: Comparison of the parameters of several particle filtering algorithms. Random number generators and mulptipliers are denoted as RNG and MUL respectively. The number of multipliers is calculated for the bearings-only tracking problem, while all other values are for the generic particle filter.

The SIRF is implemented in FPGA for the bearings-only tracking problem and the sampling frequency that is achieved for the implementation with 1000 particles is $50kHz$. This is about 50 times increase in comparison with the implementation on the state-of-the art DSP processors.

Additional speed improvements are achieved through parallel implementation. The main algorithmic and architectural challenges have been in reducing communication requirements in the resampling step and in reducing the complexity of the central unit. New parallel resampling algorithms that makes communication through the interconnection network deterministic are developed and corresponding architectures and communication protocols are proposed. Also, parallel algorithms and architectures for GPFs are developed. Communication requirements of GPFs are significantly lower. The number of data sent is proportional to N_s^2 while in the SIRF it is proportional to N_sM. However, even though communication requirements for the SIRF are high, with the modifications that allow for deterministic communication it is possible to overlap in time the particle exchange with the next particle generation step. In this way, the sampling period of SIRFs is not increased due to communication and it is proportional to $2MT_{clk}/K$. Complexity of the central unit in GPF is very high since it has to perform Cholesky decomposition while the complexity of the central unit for the new parallel SIRF algorithm is low since it is responsible only for communication.

7.2 Extensions and future work

This work can be extended in several directions including comparison of different types of particle filtering algorithms, developing automated procedure for floating to fixed point conversion of particle filters, developing application-domain specific and/or reconfigurable architectures for particle filters.

- There are several types of particle filters that are commonly used. For instance, the mixture Kalman filter is used in wireless communications for joint channel estimation and symbol detection. A possible research direction would involve the comparison of different types of particle filters from the software and hardware implementation point of view. Interestingly, each type of particle filters imposes different requirements for efficient hardware implementation. For example, the mixture Kalman filter uses much smaller number of particles than the SIRF (less than hundred). The main emphasis of the design, then, should be in exploiting functional and not data parallelism because there are no benefits in using parallel architectures described in Chapter 5.

- Important part of hardware implementation of particle filter was the conversion of floating to fixed point arithmetic. Since one-to-one mapping of operations to hardware block is performed, it is very important to use small number of bits in fixed point representation in order to reduce area requirements. Finite precision analysis is performed using an approach similar to the one described in [67]. This is an ad-hoc method that requires extensive simulations. If there is any change in the input parameters, the whole simulation has to be run again. Reconfigurable and domain specific particle filter implementations require much faster and more reliable finite precision analysis.

- Since the main goal of this dissertation is high speed implementation of particle filters, we used the application specific architectures. They are optimized only for a particular algorithm so that they provide the highest throughput. However, these architectures are not flexible. Very interesting research direction would be to implement an application-domain specific processor for particle filters. This processor would be based on combination of dedicated hardware blocks and programmable blocks so that it would achieve much higher throughput than that of the commercial DSPs and still provide a certain level of flexibility. The domains of applications that are considered are tracking and navigation. The same processor will be able to handle several different tracking and navigation models.

- Many real-time signal processing algorithms including particle filters work on blocks of data as frames. In such systems, a two-level hierarchy is often obvious, where data frames are processed by the logic blocks at global level, and elements within a frame are processed in a loop fashion at local level. Currently, we are in the process of developing reconfigurable architectures for particle filters that incorporate block level pipelining [57]. Through block level pipelining, we can achieve several objectives. First, it is possible to maintain concurrency of each processing block while providing correct synchronization between processing blocks for proper execution. Second, since control signals, data, and clock become local, hardware implementation is much easier in terms of maintaining performance by minimizing clock skews and data routing. In addition, any change in logic affects only its buffer configuration and controller such that reconfigurable design and/or core reuse is possible.

Bibliography

[1] O. Aguliar and M. West, "Bayesian dynamic factor models and portfolio allocation," *Journal of Business and Economic Statistics*, 2000.

[2] D. L. Alspach and H. W. Sorenson, "Nonlinear Bayesian estimation using Gaussian sum approximation," *IEEE Transactions on Automatic Control,* pp. 439-447, 1972.

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

[4] A. Athalye, M. Bolić, S. Hong and P. Djurić, "Architectures and Memory Schemes for Sampling and Resampling in Particle Filters," accepted for publication, 11th Digital Signal Processing and 3rd Signal Processing Education Workshops, 2004.

[5] N. Bergman, *Recursive Bayesian Estimation: Navigation and Tracking Applications.* Ph.D. Thesis, Department of Electrical Engineering, Linköping, Sweden, Dissertation No. 579, 1999.

[6] J. M. Bernardo and A. F. M. Smith, *Bayesian Theory.* New York: John Wiley & Sons, 1994.

[7] C. Berzuini, N. G. Best, W. R. Gilks, and C. Larizza, "Dynamic conditional independence models and Markov chain Monte Carlo methods," *Journal of the American Statistical Association*, vol. 92, pp. 1403-1412, 1997.

[8] M. Bolić, S. Hong, and P. M. Djurić,"Finite Precision Effect on Performance and Complexity of Particle Filters for Bearing-Only Tracking," Proceedings of the 36th IEEE Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, CA, November 2002.

[9] M. Bolić, S. Hong and P. M. Djurić, "Performance and Complexity Analysis of Adaptive Particle Filtering for Tracking Applications," Proceedings of the 36th IEEE Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, CA, November 2002.

[10] M. Bolić, P. M. Djurić, S. Hong, "New Resampling Algorithms for Particle Filters," *Proceedings of ICASSP*, Hong Kong, 2003.

[11] M. Bolić, P. M. Djurić, S. Hong, "Resampling Algorithms and Architectures for Distributed Particle Filters," submitted to *IEEE Transactions on Signal Processing*, 2003.

[12] M. Bolić, P. M. Djurić, S. Hong, "Resampling algorithms for particle filters suitable for parallel VLSI implementation," Proceedings of CISS, Baltimore, MD, 2003.

[13] M. Bolić, P. M. Djurić, and S. Hong,"Resampling Algorithms for Particle Filters: A Computational Complexity Perspective," accepted for publication, *EURASIP Journal of Applied Signal Processing*, 2004.

[14] M. Bolić, A. Athalye, P. Djurić and S. Hong, "Algorithmic Modification of Particle Filters for Hardware Implementation," accepted for publication, EUSIPCO, 2004.

[15] M. Bolić, A. Athalye, P. M. Djurić, S. Hong, "A Design Study for Practical Physical Implementation of Gaussian Particle Filters," submitted to *IEEE Transactions on Circuits and Systems I*, 2004.

[16] R. S. Bucy, "Bayes theorem and digital realization for nonlinear filters," *Journal of Astronautical Sciences*, vol. 80, pp. 73-97, 1969.

[17] B. P. Carlin, N. G. Polson, and D. S. Stoffer, "A Monte Carlo approach to nonnormal and nonlinear state-space modeling," *Journal of the American Statistical Association,* vol. 87, pp. 493-500, 1992.

[18] J. Carpenter, P. Clifford, and P. Fearnhead, "Improved particle filter for non-linear problems," *IEE Proceedings on Radar and Sonar Navigation*, vol. 146, no.1, pp. 2-7, 1999.

[19] C. K. Carter and R. Kohn, "On Gibbs sampling for state space models," *Biometrika,* vol. 81, pp. 541-553, 1994.

[20] C. K. Carter and R. Kohn, "Markov chain Monte Carlo in conditionally Gaussian state space models," *Biometrika*, vol. 83, no. 3, pp. 589-601, 1996.

[21] R. Chen, X. Wang, and J. S. Liu, "Monte Carlo filter for adaptive detection in fading channels," *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, Asilomar, CA 1999.

[22] R. Chen, C. Wang, and J. S. Liu, "Adaptive joint detection and decoding in flat-fading channels via mixture Kalman filtering," *IEEE Transactions on Information Theory,* vol. 46, no. 6, pp. 2079-2094, 2000.

[23] T. C. Clapp and S. J. Godsill, "Fixed lag smoothing using sequential importance sampling," in *Bayesian Statistics 6* Eds. J. M. Bernardo, J. O. Berger, A. P. Dawid, and A. F. M. Smith, Oxford: Oxford University Press, pp. 743-752, 1999.

[24] T. Clapp and S. Godsill, "Improvement strategies for Monte Carlo particle filters," in *Sequential Monte Carlo Methods in Practice*, A. Doucet, N. de Freitas, and N. Gordon, Eds., New York: Springer Verlag, 2001.

[25] W. G. Cohran, *Sampling Techniques,* London: Wiley, 3nd edition, 1963.

[26] K. Compton, S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software" (PDF), *ACM Computing Surveys*, Vol. 34, No. 2. pp. 171-210. June 2002.

[27] Cray Research Inc., "Cray X-MP and Cray Y-MP computer systems," Egan, MN, 1988.

[28] D. Crisan, P. Del Moral, and T. J. Lyons, "Non-linear filtering using branching and interacting particle systems," *Markov processes and Related Fields*, vol. 5, no. 3, pp. 293-319, 1999.

[29] D. E. Culler, J. P. Singh, A. Gupta , *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers, 1st edition,August 1998.

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

[31] Digital Core Design Inc. ,"Pipelined Floating Point Libraries," available at www.dcd.pl.

[32] J. L. Dekeyser, C. Fonlupt, and P. Marquet, "Analysis of synchronous dynamic load balancing algorithms," *Advances in Parallel Computing*, vol 11, pp. 455-462, 1995.

[33] L. Devroye, *Non-Uniform Random Variate Generation*, Springer-Verlag, 1996.

[34] P. M. Djurić, "Sequential estimation of signals under model uncertainty," in *Sequential Monte Carlo Methods in Practice*, A. Doucet, N. de Freitas, and N. Gordon, Eds., New York: Springer Verlag, 2001.

[35] P. M. Djurić, J. H. Kotecha, J. Zhang, Y. Huang, T. Ghirmai, M. F. Bugallo, and J. Míguez, "Particle filtering," *IEEE Signal Processing Magazine*, vol. 20, no. 5, pp. 19-38, 2003.

[36] A. Doucet, S. J. Godsill, and C. Andrieu, "On sequential Monte Carlo sampling methods for Bayesian filtering," *Statistics and Computing*, pp. 197-208, 2000.

[37] A. Doucet, S. Godsill, and M. West, "Monte Carlo filtering and smoothing with application to time-varying spectral estimation," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Istanbul, Turkey, 2000.

[38] A. Doucet, N. de Freitas, and N. Gordon, Eds., *Sequential Monte Carlo Methods in Practice*, New York: Springer Verlag, 2001.

[39] R. Duncan, "A survey of parallel computer architectures", *IEEE Computer*, 23(2), pp. 5-16, 1990.

[40] P. Fearnhead, *Sequential Monte Carlo methods in filter theory,* PhD Thesis, Merton College, University of Oxford, 1998.

[41] G. S. Fishman, *Monte Carlo: Concepts, Algorithms and Applications*, Springer series in operational research, Springer-Verlag, 1st edition, 1995.

[42] S. Frühwirth-Schnatter, "Applied state space modeling of non-Gaussian time series using integration-based Kalman filtering," *Statistics and Computing*, vol. 4, pp. 259-269, 1994.

[43] A. Gerstlauer, R. Doemer, J. Peng, and D. Gajski, *System Design: A Practical Guide with SpecC,* Kluwer Academic Publishers Inc. June, 2001

[44] J. Geweke, "Antithetic acceleration of Monte Carlo integration in Bayesian inference," *Journal of Econometrics*, vol. 38, pp. 73-90, 1988.

[45] J. Geweke, "Bayesian inference in econometric models using Monte Carlo integration," *Econometrica*, vol. 57, pp. 1317-1339, 1989.

[46] J.-L. Danger, A. Ghazel, E. Boutillon, H. Laamari. "Efficient FPGA implementation of Gaussian noise generator for communication channel emulation," Proceedings of IEEE ICECS Conference, Laslik, Lebanon, 2000.

[47] W. R.Gilks and C. Berzuini, "Following a moving target – Monte Carlo inference for dynamic Bayesian models," *Journal of the Royal Statistical Society, B*, vol. 63, p. 127-146, 2001.

[48] S. Godsill and P. J. W. Rayner, *Digital Audio Restoration - A Statistical Model Based Approach*. New York: Springer Verlag, 1998.

[49] N. J. Gordon, D. J. Salmond, and A. F. M. Smith, "A novel approach to nonlinear and non-Gaussian Bayesian stte estimation," *IEE Proceedings F*, vol. 140, pp. 107-113, 1993.

[50] L. M. Guerra, *Behavioral Level Guide Using Property-Based Design Characterization*, Ph.D. Thesis, University of California Berkeley, 1996.

[51] B. Haller, "Dedicated VLSI Architectures for Adaptive Interference Suppression in Wireless Communication Systems," *Circuits and Systems for Wireless Communications*, Kluwer Academic Publishers, 1999.

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

[53] J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach* , 3rd Edition, Morgan Kauffmann Publishers, May 2002.

[54] M. C. Herbordt, C. C. Weems, J. C. Corbett, "Message-Passing Algorithms for a SIMD Torus with Coteries," *ACM SIGArch Computer Architecture News* 19, 1 pp. 69-78, July, 1991.

[55] S. Hong, M. Bolić, and P. M. Djurić,"An Efficient Fixed-Point Implementation of Residual Systematic Resampling Scheme for High-Speed Particle Filters," accepted for publication in *IEEE Signal Processing Letters*, 2003.

[56] S. Hong, S. S. Chin, M. Bolić, P. M. Djurić, "Design and Implementation of Flexible Resampling Mechanism for High-Speed Parallel Particle Filters," submitted to *Journal of VLSI Signal Processings*, 2003.

[57] S. Hong, M. Bolić, P. M. Djurić, "Design Complexity Comparison Method for Loop-Based Signal Processing Algorithms: Particle Filters," accepted for publication, ISCAS, 2004.

[58] M. Isard and A. Blake, "Condensation – conditional density propagation for visual tracking," *International Journal of Computer Vision*, vol. 28, no. 1., pp. 5-28. 1998.

[59] K. Ito and K. Xiong, "Gaussian filters for nonlinear filtering problems," *IEEE Transactions on Automatic Control*, vol. 45, no. 5, pp. 910-927, 2000.

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

[61] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines, 3rd International Symposium on Architectural Support for Programming Languages and Operating Systems, pp. 272-282, 1989.

[62] S. J. Julier, J. K. Uhlmann, and H. F. Durrant-Whyte, "A new method for the nonlinear transformation of means and covariances in filters and estimators," *IEEE Transactions on Automatic Control*, vol. 45, no. 3, pp. 477-482, 2000.

[63] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Journal of Basic Engineering Transactions*, ASME, Ser. D 82, pp. 35-45, 1960.

[64] R. E. Kalman and R. S. Bucy, "New results in linear filtering and prediction theory," *Journal of Basic Engineering Transactions*, ASME, Ser. D 83, pp. 95-108, 1960.

[65] K. Kanazawa, D. Koller, and S. J. Russel, "Stochastic simulation algorithms for dynamic probabilistic networks," *Proceedings of the Eleventh Annual Conference on Uncertainty in AI*, UAI, pp. 346-351, 1995.

[66] A. Kienhuis, *Design Space Exploration of Stream-Based Dataflow Architectures,* PhD Thesis, Delft University of technology, 1999.

[67] S. Kim, Ki-Il Kum, and W Sung, "Fixed-Point Optimization Utility for C and C++ Based Digital Signal Processing Programs," *IEEE Transaction on Circuits and SystemsII: Analog and Digital Signal Processing*, pp. 1455-1464, vol. 45, no. 11, Nov 1998.

[68] G. Kitagawa, "Non-Gaussian state-space modeling of nonstationary time series," *Journal of the American Statistical Association,* vol. 82, pp. 1032-1063, 1987.

[69] T. Kloek and H. K. van Dijk, "Bayesian estimates of system equation parameters: An application of integration by Monte Carlo," *Econometrica*, vol. 46, pp. 1-19, 1978.

[70] A. Kong and J.S Liu and W.H. Wong, "Sequential Imputations and Bayesian Missing Data Problems," *Journal of American Statistical Association*, Vol. 89, no. 425, pp. 278-288, 1994.

[71] J. H. Kotecha and P.M. Djurić,"Gaussian Particle Filtering," Proceedings of SSP, Singapore, 2001.

[72] J. H. Kotecha, *Monte Carlo for dynamic state space models with applications to communications*, PhD. Thesis, Stony Brook University, Dec. 2001.

[73] S. C. Kramer and H. W. Sorenson, "Recursive Bayesian estimation using piece-wise constant approximations," *Automatica*, vol. 24, pp. 789-801, 1988.

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

[75] D. J. Kuck, *The Structure of Computers and Computations*, John Wiley, New York, NY, 1978.

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

[77] J. S. Liu and R. Chen, "Sequential Monte Carlo methods for dynamic systems," *Journal of the American Statistical Association,* vol. 93, pp. 1032-1044, 1998.

[78] J. S. Liu, R. Chen, and W. H. Wong, "Rejection control and sequential importance sampling," *Journal of the American Statistical Association,* vol. 93, no. 443, pp. 1022-1031, 1998.

[79] J. S. Liu, R. Chen, W.H. Wong, "A Theoretical Framework for Sequential Sampling and Resampling", *Sequential Monte Carlo Methods in Practice*, pp 225-247, New York: Springer Verlag, 2001.

[80] L. Ljung and T. Södereström, *Theory and Practice of Recursive Identification*. Cambridge, MA: The MIT Press, 1987.

[81] J. MacCormick and A. Blake, "A probabilistic exclusion principle for tracking multiple objects," *Proceedings of the International Conference on Computer Vision*, pp. 572-578, 1999.

[82] C. J. Masreliez, "Approximate non-Gaussian filtering with linear state and observation relations," *IEEE Transactions on Automatic Control,* vol. 20, pp. 107-110, 1975.

[83] R. J. Meinhold and N. D. Singpurwalla, "Robustification of Kalman filter models," *Journal of the American Statistical Association*, vol. 84, pp. 489-486, 1989.

[84] R. Miller, V. K. Prasanna Kumar, D. I. Reisis, and Q.F. Stout, "Meshes with Reconfigurable Buses," Proc. Fifth MIT Conf. Advanced Research in VLSI, pp. 163178, 1988.

[85] S. Mohanty, S. Choi, J. Jang, V. K. Prasanna, "A Model-based Methodology for Application Specific Energy Efficient Data Path Design using FPGAs," IEEE 13th International Conference on Application-specific Systems, Architectures and Processors (ASAP 2002), 2002.

[86] C. Musso, N. Oudjane, and F. Le Gland, "Improving regularized particle filters," in *Sequential Monte Carlo Methods in Practice*, A. Doucet, N. de Freitas, and N. Gordon, Eds., New York: Springer Verlag, 2001.

[87] A. Nicolau and J.A. Fisher, "Measuring the parallelism available for very long instruction word architectures," IEEE Transactions on Computers, Vol. 33, No. 11, pp. 968-976, 1984.

[88] N. Oudjane and C. Musso,"Progressive correction for regularized particle filters," *Proceedings of the 3rd International Conference on Information Fusion*, 2000.

[89] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*, John Wiley & Sons, 1998.

[90] D. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA, 1990.

[91] P. Pirsch, *Architectures for Digital Signal Processing,* John Wiley & Sons, NY, 1998.

[92] M. K. Pitt and N. Shepard, "Filtering via simulation: auxiliary particle filters," *Journal of the American Statistical Association*, vol. 94, no. 446, pp. 590-599, 1999. 1997.

[93] A. Pole and M. West, "Efficient numerical integration in dynamic models," Research Report, 136, Department of Statistics, University of Warwick, 1988.

[94] E. Punskaya, C. Andrieu, A. Doucet, and W. J. Fitzgerald, "Particle filtering for demodulation in fading channels with non-Gaussian additive noise," *IEEE Transactions on Communications*, vol. 49, no. 4, pp. 579-582, 2001.

[95] J. M. Rabaey, M. Pedram(eds), *Low Power Design Metodologies,* Kluwer Academic Publisher, 1996.

[96] J. M. Rabaey, W. Gass, R. Brodersen, T. Nishitani, "VLSI Design and Implementation Fuels the Signal-Processing Revolution," *IEEE Signal Processing Magazine*, pp.22-37, January 1998.

[97] A. Raghunathan, N. K. Jha, S. Dey, "High-level Power Analysis and Optimization," Kluwer Academic Publisher, 1998.

[98] M. A. Richards, G. A. Show, "Chips, Architectures and Algorithms: Reflections on the Exponential Growth of Digital Signal Processing Capability," submitted to the *IEEE Signal Processing Magazine*.

[99] B. D. Ripley, *Stochastic Simulation*, New York: Wiley, 1987.

[100] D. B. Rubin, J. M. Bernardo and M. H. De Groot and D. V. Lindley and A. F. M. Smith, "Bayesian Statistics 3," Oxford: University Press, pp 395-402, 1988.

[101] S. G. Shiva, *Pipelined and parallel computer architectures,* Harper Collins College Publisher, 1996.

[102] H. W. Sorenson and D. L. Alspach, "Recursive Bayesian estimation using Gaussian sums," *Automatica*, vol. 7, pp. 465-479, 1971.

[103] H. W. Sorenson, "Recursive estimation for nonlinear dynamic systems," in *Bayesian Analysis of Time Series and Dynamic Models*, J. C. Spall, Ed. New York: Dekker, 1988.

[104] *SystemC 2.0.1 Language Reference Manual*, Available from Open SystemC Initiative at http://systemc.org, 2003.

[105] H. Tanizaki, *Nonlinear Filters: Estimation and Applications*. Lecture Notes in Economics and mathematical Systems, vol. 400. New York: Springer Verlag, 1993.

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

[107] G. S. Tjaden and M. J. Flynn, "Detection and parallel execution of parallel instructions," *IEEE Transactions on Computers*, Vol. 19, No. 10, pp. 889-895, 1970.

[108] R. van der Merwe, A. Doucet, J. F. G. de Freitas, and E. Wan, "The unscanted particle filter," in *Advances in Neural Information Processing*, (NIPS 13), 2000.

[109] J. Van Meerbergen, *Embedded Multimedia Systems in Silicon,* http://www.ics.ele.tue.nl/ jef/education/5p520/.

[110] J. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Trans. Electronic Computing*, Vol EC-8, pp330-334, Sept 1959.

[111] S. Wang, V. Piuri, "A Unified View of CORDIC Processor Design," Application Specific Professors, Ch 5 EC-8, pp. 121-160, Kluwer Academic Press, 1996.

[112] M. West, P. J. Harrison, and H. S. Migon, "Dynamic generalized linear models and Bayesian forecasting, (with discussion)," *Journal of the American Statistical Association*, vol. 80, pp. 73-97, 1985.

[113] M. J. Wolfe, "Automatic vectorization, data dependence, and optimization for parallel computers," in *Parallel Processing for Supercomputing and Artificial Intelligence*, K. Hwang and De Groot (eds.), Ch. 11, McGraw-Hill, New York, NY, 1989.

[114] M. Wolfe, *High Performance Compilers for Parallel Computing,* Addison-Wesley Publishing Company, 1996.

[115] H. Zamiri-Jafrian and S. Pasupathy, "Adaptive MLSD receiver with identification of flat fading channels," *International Conference on Acoustics, Speech and Signal Processing*, 1997.

[116] V.S. Zaritskii, V. B. Svetnik, and L. I. Shimelevich, "Monte Carlo techniques in problems of optimal data processing," *Automatic Remote Control*, vol. 12, pp. 95-103, 1975.

[117] Xilinx Inc., "Virtex-II Pro Patforms FPGA: Functional Description," available from www.xilinx.com, June 2003.