Design Methodology for Domain Specific Parameterizable Particle Filter Realizations

Sangjin Hong, *Senior Member, IEEE*, Jinseok Lee, *Student Member, IEEE*, Akshay Athalye, *Member, IEEE*, Petar M. Djurić, *Fellow, IEEE*, and We-Duke Cho, *Member, IEEE*

***Abstract*—This paper presents a reconfigurable particle filter design methodology for a real-time bearings-only tracking application. The methodology provides the capability of selecting a single particle filter from multiple particle filter realizations with maximum resource sharing. The autonomous buffer controller mechanism for the architecture ensures correct operation of the particle filters. Parameter adaptation and algorithm reconfiguration can be accomplished with negligible reconfiguration overhead through buffer controllers and a set of switches for transforming dataflow structures such that any desired particle filter can be implemented. Two target particle filters, sample importance resample filter (SIRF) and Gaussian particle filter (GPF), are realized using field programmable gate array (FPGA) based on the proposed methodology. However, the architecture can be extended for a wide range of particle filters with different sets of dynamics. This paper successfully demonstrates that implementation of a domain specific processor for particle filters is feasible with performance that is much higher than that of commercially available digital signal processors (DSPs).**

***Index Terms*—Buffer controller, field programmable gate array (FPGA) design methodology, particle filter, reconfigurable design.**

I. Introduction

PARTICLE filtering is an emerging powerful methodology for sequential signal processing, especially for nonlinear and non-Gaussian problems [1]–[3]. The applications include wireless communications, navigation systems, sonar, and robotics, where sequential (adaptive) signal processing is needed [2], [4]. A common problem in all of these applications is the estimation and/or detection of dynamic signal parameters or states in real time. Executing different types of particle filtering algorithms on state-of-the-art digital signal processors (DSPs) suffers significantly due to lack of concurrency exploitation. Particle filters (PFs), which require a significant amount of computations, present an important challenge for hardware implementation. There are many applications where PFs can make considerable improvements in performance but often they have not been used because they cannot meet the stringent requirements of real-time processing. Hence, it is highly desirable to develop a programmable particle filtering hardware that can replace the use of DSP.

In this paper, we consider the design of such a programmable particle filtering hardware for tracking applications. In practice, a PF algorithm for tracking must change to cope with the type of objectives and the dynamics of the target. The main contribution of the paper is a design methodology for a high speed, domain specific, parameterizable particle filtering hardware with reconfigurability. A reconfigurable PF can be used in a wide range of applications depending on the statistical parameters of input observation and dynamic models. Reconfigurability includes the type of PF used, dimension of state spaces, and the number of particles that may change dynamically to adapt to changing environments. For many real-time applications, dynamically varying degree of parallelism is desirable where processing elements duplicate for higher throughput processing. In addition, a multiple instance of the same PF is useful for tracking more than one object at a time. It is also desirable to change the execution speed for power reduction if the real-time requirement is relaxed. Thus, we can envision a reconfigurable processor that can support the above situation dynamically during run-time without redesigning particle filters whenever specifications change.

The use of reconfigurable DSP is not new [5], but most of reconfigurability has been focused at traditional fine-grain operations such as multipliers and adders. To the best of our knowledge no previous works focus on the reconfigurable architecture for the PF. Commercial DSPs provide ultimate flexibility for PF design. Filtering parameters as well as type of filters can be easily adapted through change in software routines for a specific filtering task. Even though current DSPs are highly pipelined and support some level of concurrency, they are not suitable for high-speed real-time particle filtering. On the other hand, in the field programmable gate array (FPGA) implementation, operational concurrency can be achieved by utilizing multiple processing elements or parallelizing the overall filtering algorithms. However, a key problem with the FPGA implementation using commercial design environment is that programming/reconfiguration is done statically during the design stage with long time to program, and without special support at the implementation level. Recently, commercial FPGAs have started to support partial dynamic reconfiguration, but this comes at the cost of a complicated design flow and larger overheads which make it impractical for large designs [6].

The reconfigurable PF architecture presented here is based on maximizing reuse of processing blocks common to different PF algorithms. The interconnection and interface between the

Manuscript received March 26, 2004; revised June 23, 2004, January 29, 2005, June 2, 2005, December 13, 2005, and April 24, 2006. This work was supported by the Ubiquitous Computing and Network (UCN) Project, the Ministry of Information and Communication (MIC) 21st Century Frontier R&D Program in Korea. This paper was recommended by Associate Editor T. Arslan.

S. Hong, J. Lee, A. Athalye, and P. M. Djurić are with the Department of Electrical and Computer Engineering, Stony Brook University, Stony Brook, NY 11794-2350 USA (e-mail: snjhong@ece.sunysb.edu; jselee@ece.sunysb.edu; athalye@ece.sunysb.edu; djuric@ece.sunysb.edu).

W.-D. Cho is with the Department of Electronics Engineering, College of Information Technology, Ajou University, Suwon, Korea (e-mail: chowd@ajou.ac.kr).

Digital Object Identifier 10.1109/TCSI.2007.904690

processing blocks is done using distributed buffers, controllers, and multiplexers. In the design, we consider two different types of particle filtering algorithms, namely Gaussian particle filter (GPF) and sample importance resample filter (SIRF), for the bearings-only tracking problem [7], [8]. Using the proposed architecture, dynamic reconfiguration can be done by simply loading a set of parameters into the controllers. We demonstrate the methodology using commercial FPGA.

The rest of the paper is organized as follows. Section II briefly describes the background and motivation of the work. We discuss characteristics of PFs and requirements for reconfigurable architectures. In Section III, two types of algorithms, SIRF and GPF, for bearings-only tracking (BOT) applications are discussed. Section IV discusses the structure of the processing blocks used for performing operations in the PFs. The combined architecture with processing blocks and buffer controllers is presented in Section V. The designs are mapped to FPGA and are evaluated in Section VI. Finally, our contribution is summarized in Section VII.

II. Background and Motivation

A. PF Characteristics

PFs are used in nonlinear signal processing where the interest is in tracking and/or detection of random signals. PFs base their operations on representing the posterior density of the state of interest by discrete random measures composed of particles (samples) and their respective weights [1]–[3]. Accordingly, the posterior at every time instant t is represented by the random measure $\{x_{1:t}^{(m)}, w_t^m\}_{m=1}^M$ where $x_t^{(m)}$ is the mth particle of the signal at time t, $x_{1:t}^{(m)}$ is the mth trajectory of the signal, and $w_t^{(m)}$ is the weight of the mth particle (or trajectory) at time t. The particles are drawn from known densities (importance functions, IFs) and weights assigned to each particle using the principle of importance sampling [3]. This weighted set of particles can then be used to find various estimates of the state like the MMSE or MAP estimate. As new observations become available, the particles are propagated and the weights are calculated by exploiting Bayes theorem and sequential importance sampling [3]. There are many variation of PFs, but in this paper we consider two general types. One employs the sample-importance-resample (SIR) algorithm and we refer to these filters as sample-importance-resample filters (SIRFs). The other filters do not employ resampling but use Gaussian approximations to desired densities, and we call these filters Gaussian particle filters (GPFs) [7]. The selection of an algorithm from these two types of filters depends upon the characteristics of the dynamic state space of interest.

PFs, as many real-time signal processing algorithms, work on blocks of data as frames. They can be represented as coarse data-flow graphs such that nodes (or blocks) can be executed concurrently [9]. While the complexity of each node (or block) differs in granularity, the data-flow graph can be clearly represented as a function of data dependency. Each node in the data-flow graph executes on a set of data every iteration cycle. Depending on applications, the size of data set can be large requiring significant amount of buffers. Considering the dataflow within these applications, a two-level hierarchy is often obvious, where data frames are processed as a unit in a sequence of logic blocks at a global level, and elements within a frame are processed in a loop fashion within each block at a local level. For example, for the function $Y = h[f(X) + g(X)]$, the data frame $X, (x_1, x_2, \ldots, x_M)$ of dimension M is processed by the functions $f(\cdot)$ and $g(\cdot)$ in a loop, and, at the global level, the sums of elements from these two functions are processed by the function $h(\cdot)$ generating the data frame Y.

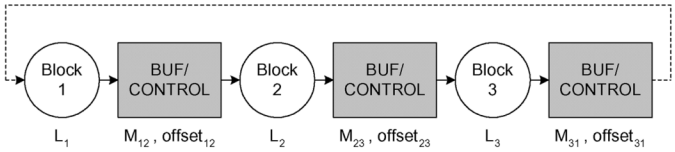

Fig. 1. Illustration of the block-level pipelining structure of data flow. A possible recursion is also illustrated with dotted connection between the processing blocks C and A.

B. Block-Level Pipelining

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

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

1) It is possible to maintain concurrency of each processing block while providing correct synchronization between them for proper execution.
2) Since the control signals, data and clock become local, hardware implementation is much easier in terms of maintaining performance by minimizing clock skews and data routing. Any change in logic will only affect its buffer size and controller configuration so that reconfigurable design and/or core reuse is possible.
3) Since the entire design is centered around the buffers, performance mismatches between memory and logics are minimized.

Fig. 1 illustrates a block-level pipelining structure. Data are transferred by the read and write access of the buffers concurrently but at different address locations. The offset or difference between the write and read address can be determined from the rate differences of processing blocks as well as data dependency between them. Each buffer can have different offset as required by the logic operation writing to or reading from it. The overall operation is logically viewed as just buffer to buffer operations separated by latency of the processing blocks introduced by the processing block logic implementation. The data path may be recursive.

The above structure is characterized by the parameters L_i, M_{ij}, and offset_{ij}, as indicated in Fig. 1. The values L_i represents the latency of processing block i and is obtained from the implementation, M_{ij} is the size of the data frame transferred between processing blocks i and j. The buffer write-read offset,

offset_{ij}, is determined from the operational data dependency of the pair of processing blocks. While a pair of processing blocks in Fig. 1 produces and consumes the same number of data, their data rates are not necessarily identical.

C. Orthogonal Controller Design

When we consider dynamic run-time reconfiguration of PFs, a method for controlling the desired operation becomes an important issue. Two approaches for controller design are possible, namely centralized and distributed. In the centralized approach, a single large controller is used to generate the different control signals for all the units. While [10] takes a centralized controller approach arguing on the grounds of area overhead, [11] and [12] take a distributed approach where the control is localized at the units and they operate by transferring information between them. We adopt a distributed approach in this paper. The main difference between our controller design methodology and other distributed approaches is that our design assumes that the execution characteristics of the processing block are known. This is a valid assumption since we can always characterize the processing blocks based on their implementation. This has the benefit that the processing blocks can be a logic core designed by another party. Moreover, we provide the flexibility in changing the controller locally with minimum information and overhead. Such benefit is not possible with the centralized approach where a small change in logics translates to overall redesign of the controller. In the reconfigurable design, the centralized approach will lead to longer time to reconfigure the overall design because of the tightly integrated control signals. Our methodology is specifically targeted for block based processing for buffer centric applications. The methodology isolates local controllers from each other such that it is possible to provide operation predictability in the overall system design.

The two algorithms considered in this paper share several common processing blocks but differ in their dataflow structures. Even though only two possible PFs are considered, the architecture can be easily extended for other PFs as long as the resources are available. Reconfiguration is achieved by configuring distributed buffer controllers for local processing blocks and structural controller for algorithm selection. We can view this reconfiguration scheme as one based on execution context.

III. Target Application: Bearings-Only Tracking

In this paper, we consider design of PFs applied to the BOT problem. Here two positions of the object of interest 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 goal is to estimate the position (x_n, y_n) and velocity (x'_n, y'_n) of the tracked object in the Cartesian coordinate system.

A. SIRF Particle Filters

The data flow of the SIRF for the BOT problem is shown in Fig. 2.

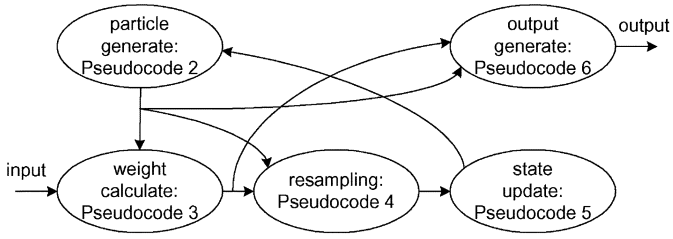

Fig. 2. Dataflow of SIRF.

In Pseudocode 1, the SIRF algorithm for processing one observation is presented. The input argument to the SIRF is the observation z, and the outputs are the estimates of the system states $(\hat{x}, \hat{V}_x, \hat{y}, \hat{V}_y)$ which represent estimates of the position and velocity, respectively. The symbols n_x, n_y, in the pseudocode denote two length M sequences of Gaussian random numbers used in the sample step. In processing the input observation, the SIRF sequentially performs the following three steps: particle generation or sampling (SIRF_S), computation of the importance weights (SIRF_I), and systematic resampling (SIRF_{SR}). There are two additional steps, one of which is used for updating the states after resampling (SIRF_U), and the other, used for calculating the estimates (SIRF_O). Each step processes M particles where M is typically of the order of a few hundred to a few thousand.

Pseudocode 1: Processing of one observation by the SIRF.

$$
\begin{aligned}
&(\hat{x}, \hat{V}_x, \hat{y}, \hat{V}_y) = SIRF(z, n_x, n_y)\\
&\quad (x, V_x, y, V_y) = SIRF_S(\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y, n_x, n_y)\\
&\quad (w, S_M) = SIRF_I(x, y, \tilde{w}, z)\\
&\quad (i) = SIRF_{SR}(w)\\
&\quad (\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y, \tilde{w}) = SIRF_U(x, V_x, y, V_y, i)\\
&\quad (\hat{x}, \hat{V}_x, \hat{y}, \hat{V}_y) = SIRF_O(x, V_x, y, V_y, w)
\end{aligned}
$$

The generation of particles is performed by drawing them from the importance density in the sample step. In Pseudocode 2, the sample step for the BOT is presented. The input argument is the set of resampled particles from the previous time instant $(\tilde{X} = \{\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y\})$. For the sake of simplicity of implementation, the prior density of the state is selected as importance density. The output of the sampling step is a new vector of states $X = \{x, V_x, y, V_y\}$.

Pseudocode 2: Algorithm for Sample step.

$$
\begin{aligned}
&(x, V_x, y, V_y) = SIRF_S(\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y, n_x, n_y)\\
&\quad \text{for m=1:M}\\
&\qquad x(m) = \tilde{x}(m) + \tilde{V}_x(m) + 0.5n_x(m)\\
&\qquad V_x(m) = \tilde{V}_x(m) + n_x(m)\\
&\qquad y(m) = \tilde{y}(m) + \tilde{V}_y(m) + 0.5n_y(m)\\
&\qquad V_y(m) = \tilde{V}_y(m) + n_y(m)\\
&\quad \text{end}
\end{aligned}
$$

The algorithm for computation of the importance weights (importance step) is shown in Pseudocode 3. First, the weights are evaluated up to a proportionality constant and subsequently, they are normalized. The input arguments are the current observation z and the sampled particles x and y.

Pseudocode 3: Algorithm for Importance step.

$(w, S_M) = SIRF_I(x, y, \tilde{w}, z)$
  $S_M = 0$
  for $m = 1 : M$
    $w^*(m) = e^{-(2\pi\sigma_v^2)^{-1}\cdot(z - atan\frac{y(m)}{x(m)})^2}$
    $S_M = S_M + w^*(m)$
  end
  for $m = 1 : M$
    $w(m) = \frac{w^*(m)}{S_M}$
  end

The process of resampling is performed using the traditional systematic resampling algorithm [3]. This algorithm is presented in Pseudocode 4. Here, w is an array of scaled weights from the importance step. When the weights are normalized, the sum of weights is 1. The output i is an array of indexes, which indicates the particles that form the resampled set.

Pseudocode 4: Algorithm for systematic resampling.

$(i) = SIRF_{SR}(\tilde{w})$
  Generate random number $U \sim U[0, \frac{1}{M}]$
  $CSW = 0, k = 0$
  for $m = 1 : M$
    while $(CSW < U)$
      $k = k + 1$
      $CSW = S + w(k)$
    end
    $U = U + \frac{1}{M}$
    $i(m) = k$
  end

The particles have to be updated (Pseudocode 5) in order defined by the index array i from the resampling step. The output of the updated states is the new random measure $(\tilde{X}, \tilde{W})$, where $\tilde{X} = \{\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y\}$ and $\tilde{w}(m) = 1/M$ for $m = 1, \ldots, M$. The states updating is performed simply by indirect addressing $x(i(m))$ using the index array$i(m)$.

Pseudocode 5: Algorithm for state update.

$(\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y, \tilde{w}) = SIRF_U(x, V_x, y, V_y, i)$
  for m=1:M
    $\tilde{x}(m) = x(i(m))$
    $\tilde{V}_x(m) = V_x(i(m))$
    $\tilde{y}(m) = y(i(m))$
    $\tilde{V}_y(m) = V_y(i(m))$
    $\tilde{w}(m) = 1/M$
  end

The computation of the output estimates is shown in Pseudocode 6.

Pseudocode 6: Algorithm for output generation (Computation of estimates).

$(\mu_x, \mu_{V_x}, \mu_y, \mu_{V_y}) = SIRF_O(x, V_x, y, V_y, w)$
  $\mu_x = \sum_{m=1}^{M} w(m)x(m)$
  $\mu_{V_x} = \sum_{m=1}^{M} w(m)V_x(m)$
  $\mu_y = \sum_{m=1}^{M} w(m)y(m)$
  $\mu_{V_y} = \sum_{m=1}^{M} w(m)V_y(m)$

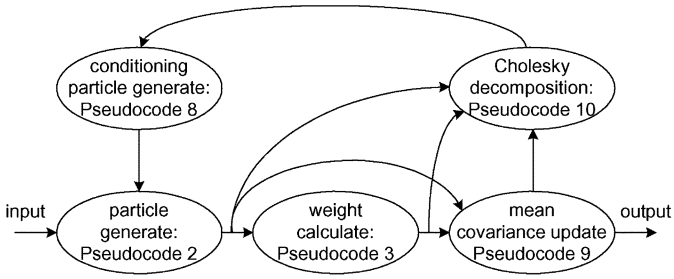

Fig. 3. Dataflow of GPF.

B. Gaussian Particle Filters (GPFs)

The details of the GPF algorithm are discussed in [7]. A dataflow for the GPF algorithm is shown in Fig. 3. Each element of GPF is on the critical path. In Pseudocode 7, the GPF algorithm for processing one observation is presented. The input and output argument to the GPF are the same as for the SIRF. Also, sample step, weight computation (importance) step, and computation of estimates are the same as in the SIRF algorithm. Generation of conditioning particles $(\mathrm{GPF}_{\mathrm{CPG}})$, covariance calculation $(\mathrm{GPF}_{\mathrm{CC}})$, and Cholesky decomposition operation $(\mathrm{GPF}_{\mathrm{CH}})$ are specific for the GPF algorithm. The resampling operation of SIRFs is absent in GPFs.

Pseudocode 7: Processing of one observation by a GPF.

$(\mu_x, \mu_{V_x}, \mu_y, \mu_{V_y}) = GPF(z, n_x, n_y)$
  $(\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y) = GPF_{CPG}(\mathbf{S}, \mu_{\mathbf{x}}, \mu_{\mathbf{V_x}}, \mu_{\mathbf{y}}, \mu_{\mathbf{V_y}}, \mathbf{n_1}, \mathbf{n_2}, \mathbf{n_3}, \mathbf{n_4})$
  $(x, V_x, y, V_y) = GPF_S(\tilde{x}, V_x, \tilde{y}, V_y, n_x, n_y)$
  $(w, S_M) = GPF_I(x, y, \tilde{w}, z)$
  $(\mu_x, \mu_{V_x}, \mu_y, \mu_{V_y}) = GPF_O(x, V_x, y, V_y, w)$
  $(\mathbf{Var}) = GPF_{CC}(x, V_x, y, V_y, w, \hat{x}, \hat{V}_x, \hat{y}, \hat{V}_y)$
  $(\mathbf{S}) = GPF_{CH}(x, V_x, y, V_y, w)$

Generation of conditioning particles $(\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y)$ is presented in Pseudocode 8. Conditioning particles are drawn from a Gaussian distribution with parameters $(\boldsymbol{\mu}, \mathbf{Var})$ where $\boldsymbol{\mu}$ is a 4×1 vector and $\mathbf{Var}$ is a 4×4 covariance matrix. In order to draw Gaussian random numbers, matrix $\mathbf{S}$ is necessary, where $\mathbf{Var} = \mathbf{S} \cdot \mathbf{S^T}$. This is an upper triangular matrix obtained by Cholesky decomposition of the covariance matrix.

Pseudocode 8: Algorithm for conditioning particle generation.

$(\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y) = GPF_{CPG}(\mathbf{S}, \mu_{\mathbf{x}}, \mu_{\mathbf{V_x}}, \mu_{\mathbf{y}}, \mu_{\mathbf{V_y}}, \mathbf{n_1}, \mathbf{n_2}, \mathbf{n_3}, \mathbf{n_4})$
  for m=1:M
    $\tilde{x}(m) = \mu_x + S_{11} \cdot n_1(m)$
    $\tilde{V}_x(m) = \mu_{V_x} + S_{12} \cdot n_1(m) + S_{22} \cdot n_2(m)$
    $\tilde{y}(m) = \mu_y + S_{13} \cdot n_1(m) + S_{23} \cdot n_2(m) + S_{33} \cdot n_3(m)$
    $\tilde{V}_y(m) = \mu_{V_y} + S_{14} \cdot n_1(m) + S_{24} \cdot n_2(m) + S_{34} \cdot n_3(m) + + S_{44} \cdot n_4(m)$
  end

In the GPF, the posterior is approximated as a Gaussian. The mean and covariance of this Gaussian are calculated from the weighted set of particles. The covariance calculation is shown in Pseudocode 9. The covariance coefficients are first updated in the loop and then the final calculation is performed. Cholesky decomposition is shown in Pseudocode 10. The purpose of this step is to calculate the square root of the covariance matrix so that $\mathbf{Var} = \mathbf{S} \cdot \mathbf{S^T}$. As opposed to the other steps, this step is not performed in the loop and its complexity does not depend on the number of particles.

Pseudocode 9: Algorithm for Covariance calculation.

$(\mathbf{Var}) = \mathbf{GPF_V}(\mathbf{x}, \mathbf{V_x}, \mathbf{y}, \mathbf{V_y}, \mathbf{w}, \mu_{\mathbf{x}}, \mu_{\mathbf{V_x}}, \mu_{\mathbf{y}}, \mu_{\mathbf{V_y}})$

for m=1:M

$Var_{11} = Var_{11} + x(m) \cdot x(m) \cdot w(m)$
$Var_{12} = Var_{12} + x(m) \cdot V_x(m) \cdot w(m)$
$Var_{13} = Var_{13} + x(m) \cdot y(m) \cdot w(m)$
$Var_{14} = Var_{14} + x(m) \cdot V_y(m) \cdot w(m)$
$Var_{22} = Var_{22} + V_x(m) \cdot V_x(m) \cdot w(m)$
$Var_{23} = Var_{23} + y(m) \cdot V_x(m) \cdot w(m)$
$Var_{24} = Var_{24} + V_x(m) \cdot V_y(m) \cdot w(m)$
$Var_{33} = Var_{33} + y(m) \cdot y(m) \cdot w(m)$
$Var_{34} = Var_{34} + y(m) \cdot V_y(m) \cdot w(m)$
$Var_{44} = Var_{44} + V_y(m) \cdot V_y(m) \cdot w(m)$

end

$Var_{11} = Var_{11} - \mu_x \cdot \mu_x$
$Var_{12} = Var_{12} - \mu_{V_x} \cdot \mu_x$
$Var_{13} = Var_{13} - \mu_y \cdot \mu_x$
$Var_{14} = Var_{14} - \mu_{V_y} \cdot \mu_x$
$Var_{22} = Var_{22} - \mu_{V_x} \cdot \mu_{V_x}$
$Var_{23} = Var_{23} - \mu_y \cdot \mu_{V_x}$
$Var_{24} = Var_{24} - \mu_{V_y} \cdot \mu_{V_x}$
$Var_{33} = Var_{33} - \mu_y \cdot \mu_y$
$Var_{34} = Var_{34} - \mu_{V_y} \cdot \mu_y$
$Var_{44} = Var_{44} - \mu_{V_x} \cdot \mu_{V_y}$

Pseudocode 10: Algorithm for Cholesky decomposition.

$(\mathbf{Var}) = \mathbf{GPF_{CH}}(\mathbf{x}, \mathbf{V_x}, \mathbf{y}, \mathbf{V_y}, \mathbf{w})$

$S_{11} = (Var_{11})^{1/2}$
$S_{12} = Var_{12}/S_{11}$
$S_{13} = Var_{13}/S_{11}$
$S_{14} = Var_{14}/S_{11}$
$S_{22} = (Var_{22} - S_{12} \cdot S_{12})^{1/2}$
$S_{23} = (Var_{23} - S_{12} \cdot S_{13})/S_{22}$
$S_{24} = (Var_{24} - S_{12} * S_{14})/S_{22}$
$S_{33} = (Var_{33} - S_{13} \cdot S_{13} - S_{23} * S_{23})^{1/2}$
$S_{34} = (Var_{34} - S_{13} \cdot S_{14} - S_{23} \cdot S_{24})/S_{33}$
$S_{44} = (Var_{44} - S_{14} \cdot S_{14} - S_{24} \cdot S_{24} - S_{34} \cdot S_{34})^{1/2}$

IV. Reconfigurable Particle Filter Design

A. Coarse-Grain Data Flow Models

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

1) Each processing block is designed to eliminate control signal dependency between processing blocks. If there exists such control dependency, we combine them for single processing block. If such integration is unavoidable, we treat the control signal as data between processing blocks under consideration.
2) We design each processing block so that the data consuming and producing rates are deterministic. Since all the blocks, including buffers, use a single global clock, relative rates can be defined. Moreover, the numbers of data produced and consumed are also made to be deterministic. It is also possible that the processing block is complicated enough to require its own local controller. However, we consider such highly localized controller to be a part of processing block. Thus, as long as we maintain deterministic input and output data flow, we can treat it as regular logic block.
3) We assume that there is only one global clock and all other clock signals feeding to processing blocks are derived from the global clock.

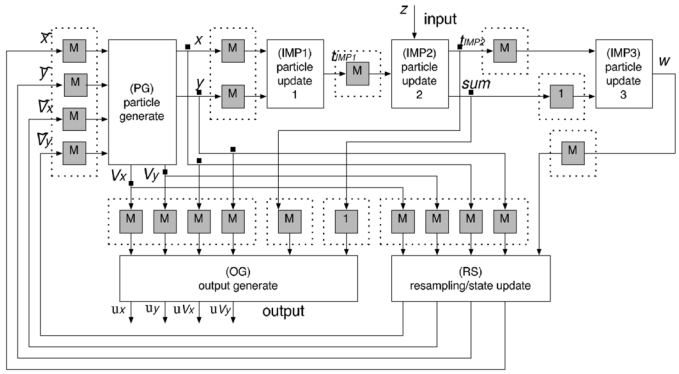

Fig. 4. Dataflow graph (with buffers and processing blocks) of the SIRF particle filter. Each dashed box is translated to a buffer controller. The same buffer controller is used for a grouped dashed box. A value inside the shaded box represents the number of data going into the buffer in each iteration.

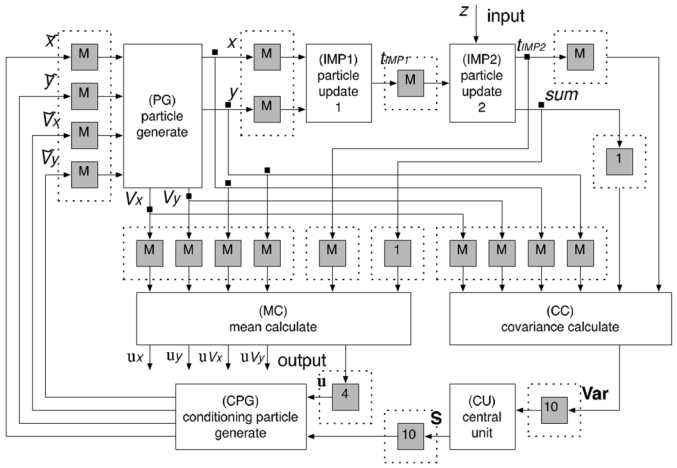

Fig. 5. Dataflow graph (with buffers and processing blocks) of the GPF particle filter. Each dashed box is translated to a buffer controller. The same buffer controller is used for a grouped dashed box. A value inside the shaded box represents the number of data going into the buffer in each iteration.

The architecture of the SIRF for the BOT problem is constructed as shown in Fig. 4. The figure shows both processing blocks and buffers (i.e., the buffer and its controller are shown with shaded boxes). Resampling and state update are combined into one block and output (estimate) computation step gets data directly from the sample step [13].

Similarly, the architecture of the GPF for BOT is constructed as shown in Fig. 5. The figure shows the processing blocks and the buffers. The diagram is a block level pipeline configuration of the dataflow shown in Fig. 3. In these two architectures, each processing block does not change but the buffer controllers are reconfigured depending on the target algorithms.

B. Shared Processing Blocks in Design

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

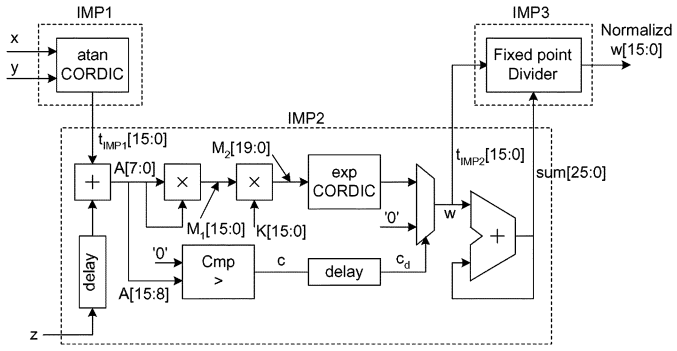

Fig. 6. Block diagram of Importance step.

1) Sample Step (Particle Generate): In the Particle Generate (PG) processing block, there are four buffers associated with input vectors $(\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y)$ and 4 buffers associated with output vectors (x, V_x, y, V_y). The input vectors stored in the input buffers are generated by the RS processing block. The outputs from the PG processing block are used again by the RS processing block in the following iteration. In addition, two more buffers are associated with (x, y) to be used by the IMP1 processing block. All inputs and outputs are of the size of M.

The arithmetic operations of the PG step are described in Pseudocode 2. The outputs are computed in parallel operations. In the PG processing block, there are two noise generators. For the generation of noise samples, we use the Box–Muller approach for efficient FPGA implementation [14]. The design is based on our previous study on effects of VLSI noise generators on performance and complexity of real-time PFs [15]. The noise generation is a combination of a lookup table and arithmetic logic.

2) Weight Computation (Importance) Step: The arithmetic operations of the weight computation step are illustrated by Pseudocode 3. The $\mathrm{atan}()$ and $\exp()$ operations are done using unrolled CORDIC. Because of the size of the arithmetic units, and to maximize resource sharing, the importance step is functionally split into three processing blocks: IMP1, IMP2, and IMP3. All of these processing blocks will be a part of the SIRF, but the GPF uses only IMP1 and IMP2. The normalization, which is done in the IMP3, is done by another unit in the GPF to improve the speed by more than 30%. This is illustrated in the next section.

The block diagram of the whole Importance step along with the mentioned sub-steps is shown in Fig. 6. The IMP1 processing block has two input buffers for (x, y) from the PG processing block and one output buffer to the IMP2 processing block. The IMP1 processing block computes $\mathrm{atan}(y/x)$ and generates temporary data sequence (t_{IMP1}) of size M. The CORDIC unit determines the angle by vector rotation. This produces an output in the range $[-\pi, \pi]$ [16]. The $\mathrm{atan}()$ function has unique values in the range $[-\pi/2, \pi/2]$. Hence, additional logic is used to convert the output of the CORDIC core to this range. All the variables are represented in fixed point format. The inputs to the IMP1 unit are 16 bits wide. After fixed point analysis, we observe that the value of the weight w reduces to 0 in fixed point if the variable A shown in Fig. 6 is greater than 255. We use this fact to reduce the area and resources required by the importance step. Accordingly, only 8 LSBs of A are propagated through the weight computation logic. This leads to a loss in accuracy of at most 10%. The 8 MSBs are compared with a constant, and if A is greater than 255, the value 0 is selected as the output weight via a multiplexer. The delay units are added for synchronization due to the fine-grain pipelining of the CORDIC units. The weights are summed using an accumulator as shown and the sum of weights is sent to the IMP3 block for normalization.

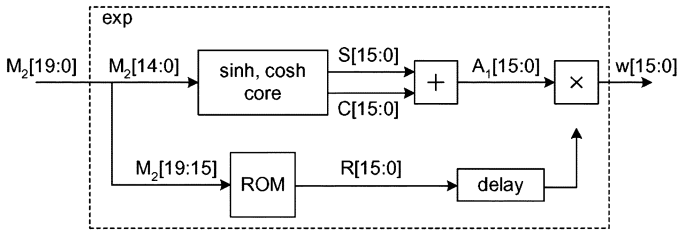

Fig. 7. Block diagram of implementation of $\exp()$ function.

Block IMP2 performs the other mathematical operations and the $\exp()$ function to find the value of the weight (when it is not zero). The implementation of the $\exp()$ function is shown in Fig. 7. The input range of the CORDIC core used for the $\exp()$ function is restricted to $[-\pi/4, \pi/4]$ [16]. Hence, we split the input exponent into an integer and a fractional part based on the fixed point format used. The $\exp()$ of the integer part is precalculated and stored in a ROM look-up table. The $\exp()$ of the fractional part is calculated using the CORDIC unit as shown and the two results are then multiplied for the final result. IMP2 has two output buffers; one for (t_{IMP2}) to the IMP3 processing block and the other for (sum) also to the IMP3 processing block.

The IMP3 processing block has two input buffers for (t_{IMP2}, sum) from the IMP2 processing block. The IMP3 normalizes each unnormalized weight t_{IMP2} with sum. The normalized weights $(\tilde{w})$ are then stored in the output buffer to be used by the RS processing block along with the particles generated at the PG processing block.

It is very important to note that, even though we are creating buffers at the output of these processing blocks in the figure, when the data are used by the successive processing block right away, we can replace these buffers by pipeline registers in the actual implementation. In this case, the value of latencies of the blocks will be added to the last buffer controller.

3) Mean Calculate/Output Generation (MC/OG): The MC processing block is used by both SIRF and GPF filters to generate the filter outputs. When only the SIRF is considered, the output can be efficiently generated by

$$\mu_x = 1/M \sum_{m=1}^{M} \tilde{x}(m) \tag{1}$$

where $(\tilde{x}, \tilde{y}, \tilde{V}_x, \tilde{V}_y)$ are from the RS processing block. Since it is normalized at the RS processing block, simple adders can be used. But in the GPF, the RS processing block is not used. Thus, normalized inputs are not available. To share the processing blocks and the buffers, we design the block to use the data generated from the PG processing block with weights and sum calculated at the IMP2 processing block. Then this block implements the Pseudocode 6 and the outputs are normalized with the value sum as

$$\mu_x = 1/sum \sum_{m=1}^{M} x(m) \cdot t_{\mathrm{Imp2}}(m). \tag{2}$$

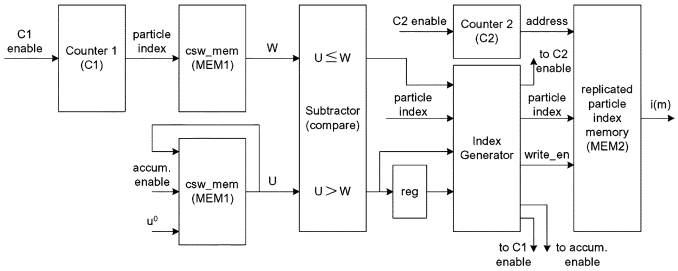

Fig. 8. Architecture of resampling unit implementing systematic resampling (SR).

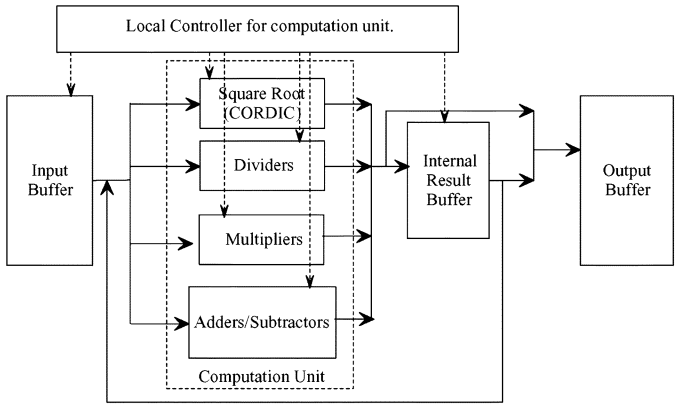

Fig. 9. Architecture of central unit implementing Cholesky decomposition.

C. Processing Blocks Unique To the SIRF

1) Resampling/State Update (RS): The RS processing block implements both the *resampling process* and the *state update* computations described in Pseudocode 4 and Pseudocode 5. The RS processing block has five input buffers with four buffers for (x, y, V_x, V_y) from the PG processing block and one buffer for normalized weights (w) from the IMP3 processing block. The resampled outputs $(\tilde{x}, \tilde{y}, \tilde{V}_x, \tilde{V}_y)$ are stored in four output buffers. All inputs and outputs are data streams of size M.

The architecture of the resampling unit performing systematic resampling (SR) is shown in Fig. 8. The input weights are cumulatively summed to form the cumulative sum of weights (CSW) and are stored in the memory. The comparator compares the corresponding values of the CSW and the resampling function U. Based on the result of this comparison, the index generator unit computes the resampled index and generates appropriate controls for updating the two functions. The whole resampling process takes $2M$ cycles. However, valid indexes will be available at the output after at most M cycles in the worst case. Thus, the PG processing block and the OG processing blocks must wait for M cycles before reading the valid data. A more detailed description of the implementation of the resampling step for SIRFs can be found in [17].

D. Processing Blocks Unique To the GPF

1) Condition Particle Generation (CPG): In the CPG step, the decomposed covariance matrix $\mathbf{S}$ and the mean $\boldsymbol{\mu}$ obtained from the CU processing block are used for calculation of conditioning particles. The matrix $\mathbf{S}$ is a 4×4 upper triangular matrix, so the number of data that are transferred from the CU processing block is 10 (not 16). All the multipliers are pipelined and they operate concurrently producing M conditioning particles. Since the outputs $(\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y)$ are computed using different number of operators, we have to introduce additional delay which is different for each state in order to get all the conditioning particles at the same time instant at the output. The CPG requires four random number generators [14].

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

2) Covariance Calculate (CC): In the CC processing block, the partial covariance 4×4 matrix $\mathbf{Var}$ is calculated. This processing block implements the first loop of the Pseudocode 9. This block generates the normalized partial covariance. In the CC processing block, there are six input buffers for (x, V_x, y, V_y) from the PG processing block and (t_{IMP2}, sum) from the IMP2 processing block. There is one output buffer $\mathbf{Var}$ to the CU processing blocks. These outputs are serialized.

3) Central Unit (CU): The architecture of the GPF CU for implementing Cholesky decomposition is shown in Fig. 9. The inputs and the outputs of the CU are produced once during the sampling period. The CU processing block executes Pseudocode 9. As seen from this pseudocode, this operation involves several expensive operations like divisions and square root. There also exist data dependencies between the various coefficients of the Cholesky decomposed matrix. Due to this use of dedicated hardware units for each of these operations proves to be expensive due to the nature of the operators and does not speed up processing greatly due to the inherent data dependency. Observing concurrency of operations from the data flow for Cholesky decomposition, we see that at any time, the maximum number of concurrent square root operations that can be performed is one, and additions, multiplications, and divisions are three. Hence, we implement the Cholesky decomposition with only these resources. The square root operation is implemented using the CORDIC core [16] and the division is implemented using the pipelined divider core [18]. The intermediate results are stored in internal registers and sent back to the computation unit to use in the calculation of the next coefficients. All the hardware units are time multiplexed (reused by subsequent operations). Like any other processing block in the design, this unit has a local controller that generates the controls for the multiplexers and read/write signals for the input and output buffers. Due to its sequential nature, the Cholesky decomposition can also be implemented on a sequential coprocessor, like the IBM PowerPC core embedded in the Virtex II pro FPGA.

The bit widths and fixed point representation formats used in all the processing blocks were determined, for the BOT problem, using a methodology outlined in [19].

E. Distributed Buffer Controller

The operations in the block based processing are viewed as buffer to buffer operations with coarse-grained processing blocks operating in between them. A block diagram of the buffer controller, which is a key element in the proposed design, consists of concurrent controller and a dual-ported memory [20]. When handling multidimensional data, the buffer controller has multiple dual-port memory units controlled by the same controller (grouped dashed boxes in Figs. 4 and 5). The number of memory units is equal to the dimension of the data. The concurrent controller has two logic sections: read and write. The write logic section is configured by the parameters L_i and nw_{ij}, and the read logic section is configured by D_{ij} and nr_{ij} where i and j denote the producing and consuming processing blocks, respectively. Note that these parameters are derived from the dataflow structure and the processing block implementation details.

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

There activation of the buffer controller is governed by three key synchronization signals: $start\_time_{ij}$, $write\_begin_{ij}$, and $read\_begin_{ij}$, where the index ij represents a buffer controller placed between the processing blocks i and j. The start of the write waiting process is synchronized with the start read process of the previous buffer controller, indexed as ki. The start of the read process is synchronized with the start of the write waiting process of the same buffer controller. They have the following relationships:

$$start\_time_{ij} = read\_start_{ki} \tag{3}$$

$$write\_start_{ij} = start\_time_{ij} + L_i + nw_{ij} \tag{4}$$

$$read\_start_{ij} = write\_start_{ij} + \max[nr_{ij}, D_{ij}]. \tag{5}$$

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

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

V. Combined Architecture

The structure of the reconfigurable hardware implementing SIRF and GPF for the BOT problem is shown in Fig. 10. The figure shows both processing blocks and buffers. In addition, there are switches associated with all the shared processing blocks and buffer controllers for selecting the appropriate structure. These switches are configured dynamically, along with buffer controller parameters, before the beginning of any iteration.

A. Buffer Controller Parameter Configuration

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

In the SIRF realization, key data dependencies that must be resolved are as follows.

1) sum through $\mathrm{BUF5}$ and the last value of t_{IMP2} through $\mathrm{BUF4}$ must arrive at the same time. This requires $nw_{\mathrm{BUF5}} = M + 1$, since IMP2 will take M cycles from its initialization to calculate the sum of weights.
2) The RS processing must wait for the sum of weights before it starts to execute. This dependency results in $nr_{\mathrm{BUF6}} = M + 61$, where 61 is the sum of latencies of all processing blocks in the path from the PG block to the RS block.
3) The PG processing block must wait for the first data generated by the RS processing block (M cycles).
4) The particle arriving at the OG block via $\mathrm{BUF7}$ and the corresponding weight arriving via $\mathrm{BUF5}$ must be synchronized.

Similarly, the GPF has the following important data dependencies.

1) The particle and weight arriving at MC through $\mathrm{BUF7}$ and $\mathrm{BUF4}$, respectively, must be synchronized. Hence, $\mathrm{BUF7}$ reading must delayed until the first weight is calculated.

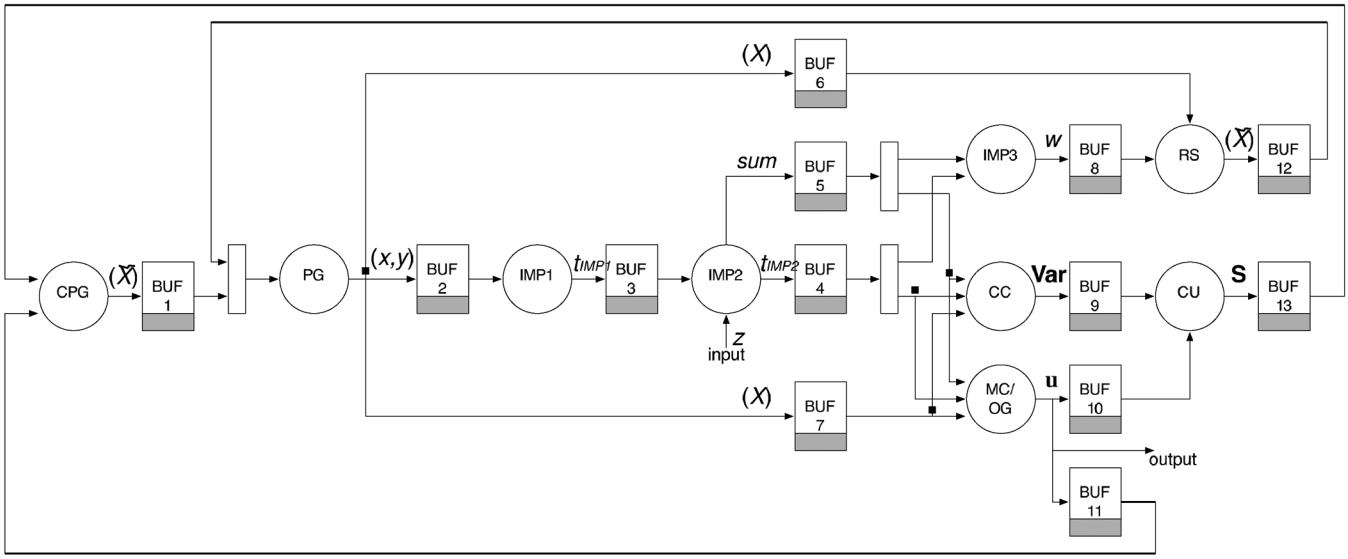

Fig. 10. Dataflow graph structure of the reconfigurable particle filter. The structure contains both SIRF and GPF. Some buffer controllers are shared in the realization.

TABLE I
EDGE INFORMATION TABLE (EIT) FOR RECONFIGURABLE REALIZATION. EACH EDGE REQUIRES A BUFFER. EACH ENTRY, DENOTED BY $[a, b]$ REPRESENTS PARAMETERS FOR SIRF AND GPF (I.E., a IS FOR THE SIRF AND b IS FOR THE GPF. THE SYMBOL — MEANS THAT THE BUFFER CONTROLLER IS NOT USED FOR THE CORRESPONDING REALIZATION). $\mathbf{X} = (x, V_x, y, V_y)$ AND $\bar{\mathbf{X}} = (\bar{x}, \bar{V}_x, \bar{y}, \bar{V}_y)$

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

2) Data arriving at the CPG block from $\mathrm{BUF11}$ and $\mathrm{BUF13}$ must be synchronized.
3) The CC and MC blocks need M cycles after reading their first data to generate result. Hence, $nw_{\mathrm{BUF9}} = nw_{\mathrm{BUF10}} = M$.

Thus, using the buffer controller parameters nw and nr, the data dependencies in the algorithms presented in Section III can be quantified and incorporated into the design. In both realizations, the values of D are all 1 since there is no rate mismatch. The parameters of the other buffer controllers in the design are determined using a similar reasoning. The depth of each buffer is bounded by $\min(nr_{ij}, M_{ij})$, where nr_{ij} is the read offset and M_{ij} is the number of data words passing through that buffer per iteration. In reality, some buffer controllers in the design need to have additional dual-port memory units to account for multi dimensional states. Accordingly, if N_s is the dimension of the state involved in the filtering, some buffer controllers in Fig. 10 will need to incorporate N_s dual-port memory units of appropriate depth. The read/write operations of all memory units within a buffer controller will be done using the same control signals. The Edge Information Table (EIT) (Table I) shows the values of the various buffer controller parameter for realization of SIRF and GPF using the architecture of Fig. 10 for the $N_s = 4$ dimensional BOT problem. The sizes and format of memory in each buffer controller for a general N_s dimensional state model is also shown in Table I. Note that M_{ij} stands for the number of data words passing through the buffer controller between block i and j, while M stands for the number of particles used for filtering.

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

B. Synchronization Signals

The overall buffer synchronization parameters are generated according to (3), (4), and (5). Table II summarizes the param-

TABLE II
SYNCHRONIZATION PARAMETERS FOR BUFFER CONTROLLERS FOR SIRF AND GPF. THE SYNCHRONIZATION POINTS ARE A FUNCTION OF M

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

eters for all the buffer controllers for the SIRF and the GPF, respectively. The parameters for the start instant are computed with respect to f_{clock}. The executions of the PG, IMP, OG, and RS processing blocks are overlapped in time. The minimum iteration period of the SIRF is $T_{\text{SIRF}} = (2M + L_{\text{SIRF}}) \cdot T_{\text{clk}}$, where $L_{\text{SIRF}} = L_{\text{PG}} + L_{\text{IMP}} + L_{\text{RS}}$. Note that L_{OG} is not included since the output generation can be done within this cycle without creating a dependency. Thus, the iteration period, $2M + L_{\text{SIRF}} = 2M + 89$, is indicated by the reset time instant. For the GPF, the executions of CPG, PG, IMP, MC, CC, and CU processing blocks are overlapped. The minimum iteration period of the GPF is $T_{\text{GPF}} = (M + L_{\text{GPF}}) \cdot T_{\text{clk}}$, where $L_{\text{GPF}} = L_{\text{CPG}} + L_{\text{PG}} + L_{\text{IMP}} + L_{\text{MC,CC}} + L_{\text{CU}}$ and where L_{CU} includes the latency due to hardware and delayed output generation resulting from time-multiplexing within the CU processing block. The overall latency L_{GPF} is longer than the constant pipelining latency of the SIRF, L_{SIRF}. Thus, the iteration period, $M + L_{\text{GPF}} = M + 154$, is indicated by the reset time instant. For $M \gg L_{\text{GPF}}$, the GPF is almost twice as fast as the SIRF.

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

C. Reconfigurability and Parameterizability

As seen from the previous sections, execution of the hardware can be alternated between SIRF and GPF by changing buffer controller parameters and interconnect switch states. The advantage of this architecture is that the processing blocks in the design are slaves to the buffer controllers which are simply configured by a small set of parameters. The processing blocks themselves do not implement any global controls. The buffer controllers with their appropriate parameters and global synchronization signals maintain the overall execution flow. As a result, changing parameters of the buffer controller allows for modifying individual filter characteristics dynamically between iterations within limitation of provided resources. The maximum number of particles that can be used is bounded, in case of the SIRF, by the depth of BUF6. The maximum dimension of the state is bounded by the number of dual-port memory units in BUF1, BUF2, BUF6, BUF7, and BUF12. Within these bounds, the number of particles and the dimension of the state can be varied between iterations by changing the parameters and control signal timings of various buffer controllers in accordance with Tables I and II.

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

The design methodology also allows for exploitation of inherent parallelizability in the PFs. The blocks CPG, PG, IMP1, IMP2, IMP3, and OG perform *data parallel* computations each iteration, i.e., each block processes a large data set where the individual computations are independent of each other. Hence, these computations can be parallelized if multiple instances of the processing blocks are available such that each instance processes a fraction of the total M particles. The proposed methodology allows for easily incorporating additional processing blocks, if they are available, into the design. Using standard design methodologies, parallelizing the filters would need a major redesign. Moreover, the methodology is extremely scalable in terms of design complexity as more and more blocks are added to increase the degree of parallelizability. The resampling step is inherently sequential. However, we have developed several resampling algorithms and that allow for parallel and distributed resampling [21]. Processing blocks implementing such algorithms can also be incorporated into the design if parallelization of resampling is needed. Fig. 11 shows how multiple processing blocks can be included in the design using the buffer controller parameters for the SIRF illustrated in Table III. The RS block implements traditional resampling and, hence, cannot be parallelized. The execution period for the SIRF using parallelization reduces to $M/K + M + L_{\text{SIRF}}$ where K is the degree of parallelism. For the GPF, similar replication can be done which leads to an execution period of $(M/k + L_{\text{GPF}}) \cdot T_{\text{clk}}$. The timing parameters presented in Table II will scale accordingly for each buffer controller.

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

VI. PHYSICAL REALIZATION AND EVALUATION

A. Processing Block and Buffer Controller Synthesis

Fig. 12 illustrates the percentage of power and area of the processing blocks synthesized on a Xilinx Virtex II pro device (XC2VP50). The actual achievable speed varies among the processing blocks. The global clock f_{clock} is set to 100 MHz for all the blocks for simplification of the controller design. It has been observed that overall speed is limited by the speed of the

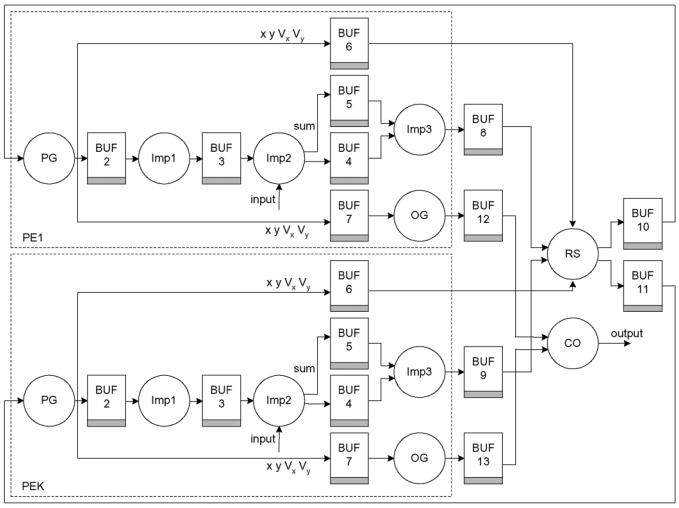

Fig. 11. Parallelizing SIRF execution by duplicating processing blocks.

TABLE III
PARAMETERS OF THE VARIOUS BUFFER CONTROLLERS

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

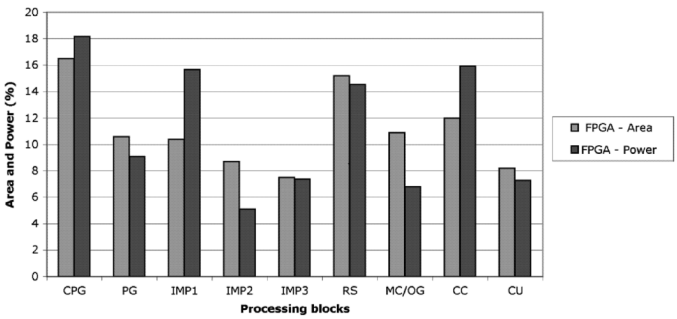

Fig. 12. Percentage of FPGA resources of the processing blocks in terms of area and power consumption. The power is estimated at 100 MHz. In the implementation, $M = 512$.

CORDIC. The power consumption of the RS block will increase with increasing M.

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

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

TABLE IV
ILLUSTRATION OF FPGA MAPPING RESULT OF BRAM BASED BUFFER CONTROLLER FOR DIFFERENT WORD SIZE. DATA IN PARENTHESES ARE FOR THE RESULT OF DISTRIBUTED MEMORY BASED BUFFER CONTROLLER

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

B. Execution Performance

The execution diagrams for the SIRF and the GPF of the reconfigurable architecture are shown in Figs. 13 and 14, respectively. The simulation shows the data transfer activities for two iterations of all the active buffer controllers. In the simulation, the value of M is chosen to be 256, which can be arbitrarily selected depending on the applications. In both filter realizations, the external input is synchronized with the start of the IMP2 processing block. As shown in the figures, the activities of the buffer controllers are overlapped, which indicates that the processing blocks are concurrently executed. The vertical lines in both figures represent the beginning of each iteration. It is clear that for the same M, the second iteration of the GPF starts more quickly than that of the SIRF.

The performances of the reconfigurable PFs are faster because of operational concurrency in the implementation. When the concurrency is fully exploited, the GPF is much faster for large M. On the other hand, the SIRF is faster when the algorithm is executed on DSP because when these two algorithms are executed sequentially, there are more computations for the GPF. Fig. 15 shows two curves that correspond to the execution times for processing M particles using the SIRF and GPF algorithms. The curves represent the sampling period as a function of number of particles obtained from the implementation on Texas Instruments (TMS320C67x) processors. In sequential implementations, the sampling period increases almost linearly with the number of particles for $M = \{500, 1000, 5000\}$. We can observe that most of the speed-up is from the functional concurrency exploitation and deep pipelining. While DSPs provide some degree of parallelism, functional concurrency cannot be fully exploited.

Fig. 16 illustrates comparison of energy consumption. The plot is normalized so that the value indicates the energy required to process one particle. The energy for the DSP is estimated with Code Composer for instruction profiling and Power Estimation Spreadsheet. The FPGA resource power is estimated using Xilinx XPower. As shown in the figure, the energy per particle is much lower for the DSP than that of the FPGA. This is due to the highly pipelined implementation for the FPGA implementation. However, as we have discussed, the potential throughput is much lower for the DSP implementation.

C. Discussion of Reconfiguration Overhead

The proposed architecture maximizes the buffer controller usage and minimizes the dynamic reconfiguration efforts. The

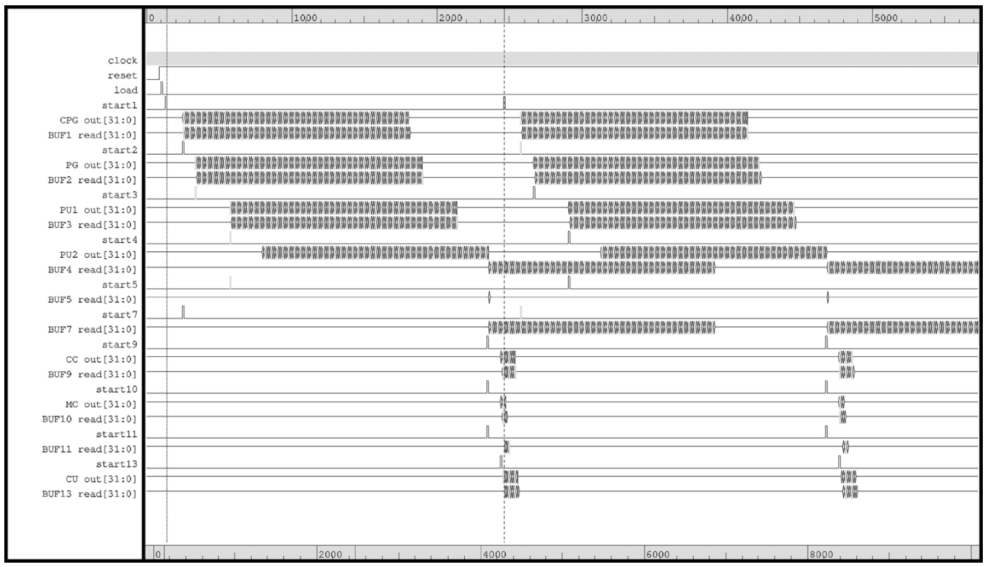

Fig. 13. Timing diagram of SIRF. The vertical lines indicate the start of the iterations. The diagram illustrates buffer activity. Overlap of the buffer activities indicate concurrent execution of the processing blocks.

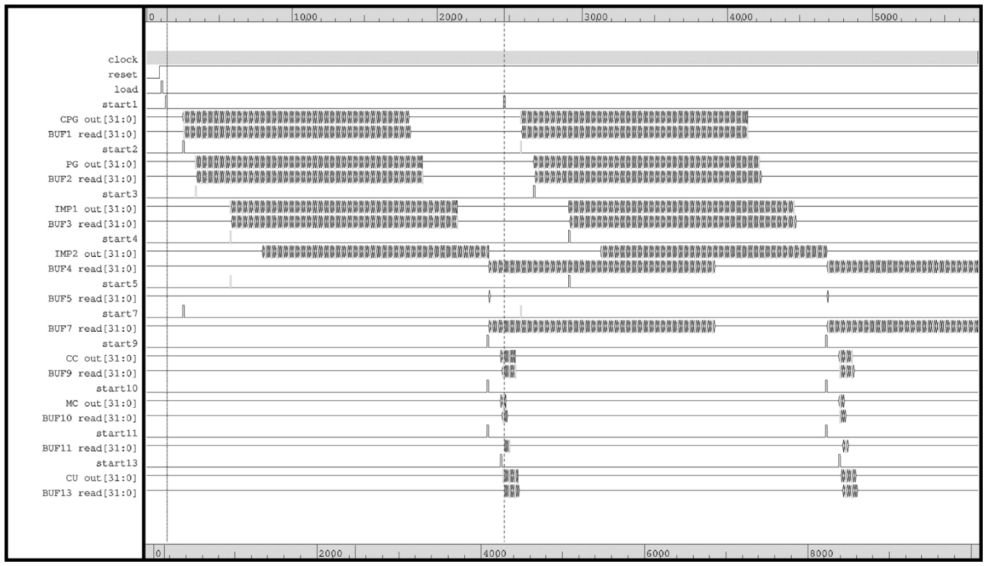

Fig. 14. Timing diagram of GPF. The vertical lines indicate the start of iterations. The diagram illustrates buffer activity. Overlap of buffer activities indicate concurrent execution of the processing blocks.

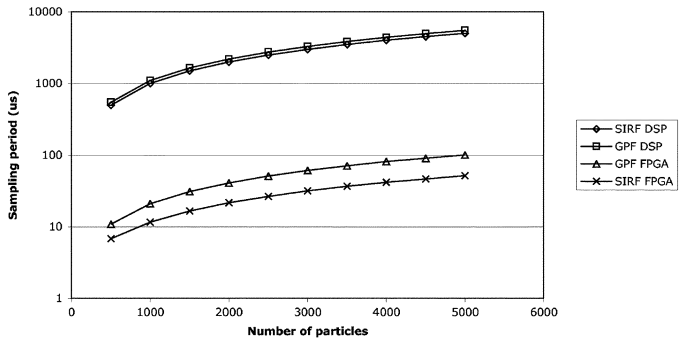

Fig. 15. Sampling period of the reconfigurable PFs (GPF and SIRF) versus number of particles. The DSP version of the filters are implemented on TI TMS320C67 Series processor.

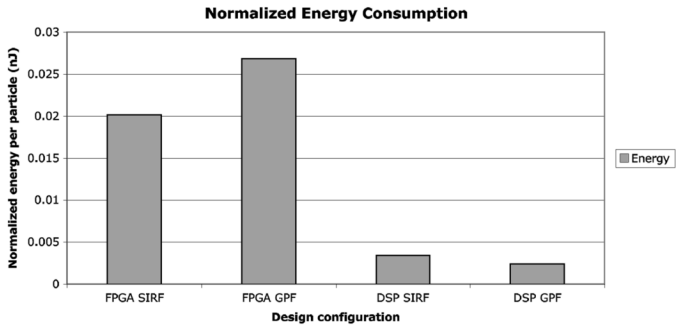

Fig. 16. Normalized energy consumption of different design configuration. The energy is normalized for one particle. $M = 512$ for FPGA implementation.

design does not suffer from the additional memory used by buffers since we can view these buffers as pipeline registers (i.e., the registers are needed in any design with standard design flow whether it is distributed or centralized). In comparison

to the processing with DSP, the energy consumption for FPGA will be more as it is using more resources and performing operations concurrently. These additional resources are necessary when the reconfigurability and high execution speed are of utmost concern.

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

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

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

VII. Conclusion

This paper introduced an effective design methodology for overall synchronization in reconfigurable PF realizations. The controller configuration is simple and systematic such that the reconfiguration is significantly simplified. The paper illustrates the effectiveness of the reconfiguration by considering two different types of PFs. We have demonstrated that by completely controlling the data transfer behavior, the processing blocks become mere slaves of the overall execution. Moreover, the design strategy of the processing blocks is well defined to satisfy the design methodology. The design can be extended to support many different PFs. The reconfigurable processor outperforms conventional DSPs.

References

[1] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for online nonlinear/non-Gaussian bayesian tracking," *IEEE Trans. Signal Process.*, vol. 50, no. 2, pp. 174–188, Feb. 2002.

[2] N. J. Gordon, D. J. Salmond, and A. F. M. Smith, "A novel approach to nonlinear and non-Gaussian Bayesian state estimation," *Proc. IEE–F: Radara, Sonar and Navigation*, vol. 140, pp. 107–113, 1993.

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

[4] P. M. Djurić, J. H. Kotecha, J. Zhang, Y. Huang, T. Ghirmai, M. F. Bugallo, and J. Miguez, "Particle filtering," *IEEE Signal Process. Mag.*, vol. 20, no. 5, pp. 19–38, Sep. 2003.

[5] R. Tessier and W. Burleson, "Reconfigurable computing and digital signal processing: A survey," *J. VLSI Signal Process.*, vol. 28, no. 1–2, pp. 7–27, May/Jun. 2001.

[6] Xilinx Inc., "Two flows for partial reconfiguration of Xilinx FPGAs," xapp290 v1.2 edition. Sep. 2004.

[7] J. H. Kotecha and P. M. Djurić, "Gaussian particle filtering," *IEEE Trans. Signal Process.*, vol. 51, no. 10, pp. 2592–2601, Oct. 2003.

[8] M. Bolić, S. Hong, and P. M. Djurić, "Finite precision effect on performance and complexity of particle filters for bearings only tracking," in *Proc. Asilomar Conf.*, 2002, pp. 838–842.

[9] J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, "Fast prototyping of datapath-intensive architectures," *IEEE Design Test*, vol. 8, no. 2, pp. 40–51, Jun. 1991.

[10] H. Jung, K. Lee, and S. Ha, "Efficient hardware controller synthesis for synchronous dataflow graph in system level design," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 10, no. 4, pp. 423–428, Aug. 2002.

[11] R. Lauwereins, M. Engels, M. Ade, and J. A. Peperstraete, "Grape-II: A system level prototyping environment for DSP applications," *IEEE Comput.*, vol. 28, no. 2, pp. 35–43, Feb. 1995.

[12] J. Dalcolmo, R. Lauwereins, and M. Ade, "Code generation of data dominated DSP applications for FPGA targets," in *Proc. 9th Int. Workshop Rapid Syst. Prototyping*, 1998, pp. 162–167.

[13] S. Hong, M. Bolić, and P. M. Djurić, "An efficient fixed-point implementation of residual systematic resampling scheme for high-speed particle filters," *IEEE Signal Process. Lett.*, vol. 11, no. 5, pp. 482–485, May 2003.

[14] J.-L. Danger, A. Ghazel, E. Boutillon, and H. Laamari, "Efficient FPGA implementation of Gaussian noise generator for communication channel emulation," in *Proc. 7th IEEE ICECS*, Laslik, Lebanon, Dec. 2000, pp. 366–369.

[15] A. Athalye, S. Hong, and P. M. Djurić, "Effects of VLSI noise generators on performance and complexity of real-time particle filters," in *Proc. Conf. Inf. Sci. Syst. (CISS)*, Johns Hopkins Univ., Baltimore, MD, Mar. 2003.

[16] Xilinx Inc., "Xilinx Cordic Core Specification DS249," v3.0 ed. Apr. 2005.

[17] A. Athalye, M. Bolić, S. Hong, and P. M. Djurić, "Generic hardware architectures for sampling and resampling in particle filters," *EURASIP J. Appl. Signal Process.*, vol. 2005, no. 17, pp. 2888–2903, 2005.

[18] Xilinx Inc., "Xilinx Pipelined Divider Core Specification," v2.0 ed. Jun. 2000.

[19] S. Kim, K. Kum, and W. Sung, "Fixed point optimization utility for C and C++ based digital signal processing," *IEEE Trans. Circuits Syst. II: Analog Digit. Signal Process.*, vol. 45, no. 11, pp. 1455–1464, Nov. 1998.

[20] M. Sadasivam and S. Hong, "Autonomous buffer controller design for concurrent execution of block level pipelined dataflows," in *Proc. IEEE Computer Society ISVLSI*, Feb. 2004, pp. 303–304.

[21] M. Bolic, P. M. Djurić, and S. Hong, "Resampling algorithms and architectures for distributed particle filters," *IEEE Trans. Signal Process.*, vol. 53, no. 7, pp. 2442–2450, Jul. 2005.

Sangjin Hong (S'98–M'99–SM'04) received the B.S. and M.S. degrees in electrical engineering and computer science from the University of California, Berkeley, and the Ph.D. degree in electrical engineering and computer science from the University of Michigan, Ann Arbor.

He is currently with the Department of Electrical and Computer Engineering at State University of New York, Stony Brook (SUNY). Before joining SUNY, he worked at Ford Aerospace Corp. Computer Systems Division as a Systems Engineer. He also worked at Samsung Electronics in Korea as a Technical Consultant. His current research interests are in the areas of low-power VLSI design of multimedia wireless communications and digital signal processing systems, reconfigurable systems-on-chip design and optimization, VLSI signal processing, and low-complexity digital circuits.

Prof. Hong has served on numerous Technical Program Committees for IEEE conferences. He is a member of Eta Kappa Nu and Tau Beta Pi.

Jinseok Lee (S'06) received dual B.S. degrees in electrical and computer engineering from State University of New York, Stony Brook (SUNY) and Ajou University, Korea, in 2005. He is currently pursuing the Ph.D. degree in the Department of Electrical and Computer Engineering at SUNY.

His research interests include multi-modal signal processing, sensor node architecture optimization, complex signal processing algorithm deisgn, and various estimation problems for wireless sensor networks.

Akshay Athalye (S'04–M'05) received the B.S. degree in electrical engineering from the University of Pune, India, in May 2001. Since then, he has been pursuing the Ph.D. degree in the Department of Electrical and Computer Engineering at the State University of New York, Stony Brook (SUNY).

His primary research interests lie in development of dedicated hardware for intensive signal processing applications. His work encompasses algorithmic modifications, architecture development, and implementation and use of reconfigurable SoC design for real-time signal processing applications. His secondary research interests lie in design and implementation of algorithms for efficient signal processing in RFID systems. He has served as an external reviewer for various journals and conferences affiliated to the IEEE and EURASIP.

Petar M. Djurić (S'86–M'90–SM'99–F'06) received the B.S. and M.S. degrees in electrical engineering from the University of Belgrade, Belgrade, Serbia, in 1981 and 1986, respectively, and the Ph.D. degree in electrical engineering from the University of Rhode Island, Kingston, RI, in 1990.

From 1981 to 1986, he was a Research Associate with the Institute of Nuclear Sciences, Vinĉa, Belgrade. Since 1990, he has been with the State University of New York, Stony Brook (SUNY), where he is Professor in the Department of Electrical and Computer Engineering. He works in the area of statistical signal processing, and his primary interests are in the theory of modeling, detection, estimation, and time series analysis and its application to a wide variety of disciplines including wireless communications and biomedicine.

Prof. Djurić has served on numerous technical committees for the IEEE and has been invited to lecture at universities in the United States and overseas. He is the current Vice President–Finance of the IEEE Signal Processing Society. He was the Area Editor for Special Issues of the *Signal Processing Magazine* and Associate Editor of the IEEE TRANSACTIONS ON SIGNAL PROCESSING. He has been on the Editorial Boards of the IEEE TRANSACTIONS ON SIGNAL PROCESSING, *IEEE Signal Processing Magazine*, *Digital Signal Processing*, *Journal of Applied Signal Processing*, *Signal Processing*, and *Journal on Wireless Communications and Networking*.

We-Duke Cho (S'83–M'89) received the B.S. degree in 1981 from Sogang University, Seoul, Korea, and the M.S. and Ph.D. degrees from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, in 1983 and 1987.

Currently, he is a Professor of Department of Electronics Engineering, College of Information Technology, Ajou University, Korea. His research interests include ubiquitous computing/network, sensor network, Post-PC (next generation PC Smart PDA), interactive DTV broadcasting technology, high-level home server and gateway, digital broadcasting and mobile convergence platform technology, and wireless networks.