EURASIP Journal on Applied Signal Processing 2005:4, 575–587

Simplifying Physical Realization of Gaussian Particle Filters with Block-Level Pipeline Control

Sangjin Hong
Department of Electrical and Computer Engineering, Stony Brook University (SUNY), Stony Brook, NY 11794-2350, USA
Email: snjhong@ece.sunysb.edu

Petar M. Djurić
Department of Electrical and Computer Engineering, Stony Brook University (SUNY), Stony Brook, NY 11794-2350, USA
Email: djuric@ece.sunysb.edu

Miodrag Bolić
School of Information Technology and Engineering, University of Ottawa, Ottawa, ON, Canada K1N 6N5
Email: mbolic@site.uottawa.ca

Received 30 March 2004; Revised 25 September 2004; Recommended for Publication by Andy Wu

We present an efficient physical realization method of particle filters for real-time tracking applications. The methodology is based on block-level pipelining where data transfer between processing blocks is effectively controlled by autonomous distributed controllers. Block-level pipelining maintains inherent operational concurrency within the algorithm for high-throughput execution. The proposed use of controllers, via parameters reconfiguration, greatly simplifies the overall controller structure, and alleviates potential speed bottlenecks that may arise due to complexity of the controller. A Gaussian particle filter for bearings-only tracking problem is realized based on the presented methodology. For demonstration, individual coarse grain processing blocks comprising particle filters are synthesized using commercial FPGA. From the execution characteristics obtained from the implementation, the overall controller structure is derived according to the methodology and its temporal correctness verified using *Verilog* and *SystemC*.

Keywords and phrases: particle filters, Gaussian particle filter, distributed controller, block-level pipelining.

1. INTRODUCTION

Particle filter has been studied theoretically and its feasibility has been demonstrated in the literature [1, 2, 3, 4, 5, 6, 7]. They perform extremely well in estimating unknown variables in many statistical signal processing problems including channel estimation in wireless communication and tracking variables in bearings-only tracking applications [8, 9, 10, 11]. However, realizing particle filters in hardware is not trivial. The algorithms consist of many complex processing blocks that are executed in both parallel and sequential nature. Direct one-to-one mapping from the operations of algorithm to the processing blocks in hardware implementation may result in complicated overall controller structure. Moreover, an efficient controller generation may even become practically intractable, or tedious, mainly due to the shear complexity of the algorithms. Hence, the main design issue in the particle filters realization is the development of their overall controller to completely maintain the concurrency of the filtering operations. The objective of this paper is to present a design methodology to simplify the overall physical design process in the particle filter realization.

At algorithmic level, particle filters work on blocks of data as frames. Such systems possess two unique execution characteristics. First, they can be represented as coarse data-flow graphs such that nodes (or blocks) can be executed concurrently [12]. 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. Second, each node in the data-flow graph executes a set of data per every iteration cycle. Depending on the specific application, the size of data set can be large requiring significant amount of buffers. Considering the data flow 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 global level, and elements within a frame are processed in a loop fashion within each block at local level. We take advantage of these characteristics in the proposed design methodology.

Two approaches for controller design are possible, namely centralized and distributed. According to the centralized

approach, a single large controller is used to generate the different control signals for all the units. While [13] takes a centralized controller approach arguing on the grounds of area overhead, [14, 15] take a distributed approach where the control is localized to the units and they operate by transferring information between them. We follow a distributed approach in this paper. Our methodology differs from the other distributed approaches in which our design is targeted for coarse granularity of the processing blocks. This has a benefit that the processing block can be core-designed by another party. Moreover, we provide the flexibility in changing the controller locally with minimum information and overhead as long as the execution characteristic of the processing block is known. Such benefit is not possible with the centralized approach where a small change in logics translates to overall redesign of the controller. A key benefit of the methodology is that local controllers are reconfigurable with a set of basic parameters, and are completely independent from one another. By a systematic method, it is possible to provide operation predictability in the overall system designs.

In this paper, we demonstrate an efficient controller design methodology for temporal correctness, and we do not consider finite precision effects inherent in the processing block design. The finite precision issue can be easily incorporated into the design by allowing proper scaling and providing sufficient bits for the number representation within each processing block. We note that such incorporation in the design does not alter the presented controller design methodology. We consider Gaussian particle filter (GPF) for design study. The correctness of our designs is verified with Verilog and SystemC simulations. The method can be applied to other systems possessing similar execution characteristics.

The remainder of this paper has five sections. Section 2 discusses background of particle filtering and there the GPF algorithm is discussed in detail. Section 3 addresses the proposed design methodology and design issues including the two-level pipelining and controller synthesis. The design study for GPF is presented in Section 4. The design is evaluated in Section 5. Finally, our contributions are summarized in Section 6.

2. PARTICLE FILTER FOR TRACKING

2.1. Background

Particle filters are used in problems which are represented using dynamic state-space (DSS) models. These models involve a state equation which shows how the state evolves with time and an observation equation that relates the noisy observations to the state. These equations have the following form:

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

where $n \in \mathbb{N}$ is a discrete-time index, $\mathbf{x}_n$ is a signal vector of interest, and $\mathbf{y}_n$ is a vector of observations. The symbols $\mathbf{u}_n$ and $\mathbf{v}_n$ are noise vectors, and $\mathbf{f}_n$ and $\mathbf{g}_n$ are a signal transition function and a measurement function, respectively, which are assumed known. In a particle filtering framework, the objective is to estimate *recursively* in time the signal $\mathbf{x}_n$, for all n, from the observations $\mathbf{y}_{1:n}$, where $\mathbf{y}_{1:n} = \{\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_n\}$.

The particle filters base their operation on representing relevant densities by particles (samples) drawn independently from a normalized importance function (IF) $\pi(\mathbf{x}_n|\mathbf{x}_{n-1}, \mathbf{y}_{1:n})$ that has the same support as the posterior density. Each sample has a weight associated with it. Accordingly, if the samples $\{\mathbf{x}_{0:n}^{(m)};\ m = 1, 2, \ldots, M\}$[1] are drawn from the importance density $\pi(\mathbf{x}_{0:n}|\mathbf{y}_{1:n})$ and represent the mth stream of particles of the unobserved state of the system, and $\mathbf{y}_{1:n}$ is the sequence of observed data, then an estimate of the posterior expectation $I(\mathbf{f}_n)$ of the function $\mathbf{f}_n(\mathbf{x}_{0:n})$ defined by

$$I(\mathbf{f}_n) = \int \mathbf{f}_n(\mathbf{x}_{0:n})\, p(\mathbf{x}_{0:n}|\mathbf{y}_{1:n}) d\mathbf{x}_{0:n} \tag{2}$$

can be obtained by the following expression [16, 17]:

$$\begin{aligned} \widehat{I}_M(\mathbf{f}_n) &= \Sigma_{m=1}^{M} \mathbf{f}_n\left(\mathbf{x}_{0:n}^{(m)}\right) w_n^{(m)}, \\ w_n^{(m)} &= \frac{\tilde{w}_n^{(m)}}{\Sigma_{j=1}^{M} \tilde{w}_n^{(j)}}, \\ \tilde{w}_n^{(m)} &= \frac{p\left(\mathbf{y}_{1:n} \middle| \mathbf{x}_{0:n}^{(m)}\right) p\left(\mathbf{x}_{0:n}^{(m)}\right)}{\pi\left(\mathbf{x}_{0:n}^{(m)} \middle| \mathbf{y}_{1:n}\right)}, \end{aligned} \tag{3}$$

where $\tilde{w}^{(m)}$ is the nonnormalized weight of the mth particle.

Densities that play a critical role in sequential signal processing are the *filtering density*, $p(\mathbf{x}_n|\mathbf{y}_{1:n})$, and the *predictive density*, $p(\mathbf{x}_{n+l}|\mathbf{y}_{1:n})$, $l \geq 1$. While sample importance resampling filters (SIRFs) operate by propagating the desired densities recursively in time, GPFs 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 (PF-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 [18, 19]. Propagation of only the first two moments instead of the whole particle set significantly simplifies 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 significantly simplified when the prior density is used as IF. This means that $\pi(\mathbf{x}_n|\mathbf{x}_{n-1}, \mathbf{y}_{1:n})$ is given by $p(\mathbf{x}_n|\mathbf{x}_{n-1})$. In this case, the GPF performs the steps presented in Algorithm 1.

In this paper, particle filters are applied to the bearings-only tracking problem illustrated in Figure 1 where two positions of the object at time instants n and $n + 1$ are shown.

[1] The notation $\mathbf{x}_{0:n}^{(m)}$ represents the set $\{\mathbf{x}_0^{(m)}, \mathbf{x}_1^{(m)}, \ldots, \mathbf{x}_n^{(m)}\}$.

Input: the observation $\mathbf{y}_n$ 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 $p(\mathbf{x}_n|\mathbf{x}_n = \mathbf{x}_{n-1}^{(m)})$ to obtain $\{\mathbf{x}_n^{(m)}\}_{m=1}^{M}$.
GPF—*measurement update algorithm.*
(3) (a)Calculate weights by $\tilde{w}_n^{(m)} = p(\mathbf{y}_n|\mathbf{x}_n^{(m)})$.
(b)Normalize the weights by $w_n^{(m)} = \tilde{w}_n^{(m)} / \sum_{m=1}^{M} \tilde{w}_n^{(m)}$.
(4) Estimate the mean and covariance of the filtering distribution by
(a)$\boldsymbol{\mu}_n = \sum_{m=1}^{M} w_n^{(m)} \mathbf{x}_n^{(m)}$,
(b)$\boldsymbol{\Sigma}_n = \sum_{m=1}^{M} w_n^{(m)} (\mathbf{x}_n^{(m)} - \boldsymbol{\mu}_n)(\mathbf{x}_n^{(m)} - \boldsymbol{\mu}_n)^{\top}$.

Algorithm 1: GPF algorithm with the prior density as the IF.

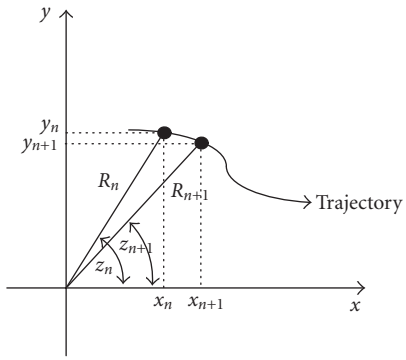

Figure 1: Illustration of the tracking problem.

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 of interest are the position and velocity of the tracked object in the Cartesian coordinate system ($\mathbf{x}_n = [x_n, x_n', y_n, y_n']^T$).

2.2. Algorithm partition strategy for physical design

Particle filtering involves many complex arithmetic computations and data access. In order to provide high-speed particle filter realization, the overall algorithm is partitioned into several modules. Each module is designed to maximize loop fusion such that data storage is minimized between modules. When DSPs are used, the loop fusion reduces memory requirement during the execution of the algorithms. From a concurrent physical realization point of view, such partitioning also minimizes the computational latency, which may be caused by pipelining with registers.

To fully utilize locality of the proposed design method, which is explained in Section 3, we follow two key strategies in defining the modules. First, each module is derived to eliminate control signal dependency between modules. Only data transfer between the modules is allowed in the algorithm. If there exists such control dependency, we combine them into a single module. Thus, each module will be designed as an independent processing block where control dependency is eliminated. Second, we design each module so that the numbers of data consuming and producing between the modules are deterministic. With deterministic number of data transfers between the modules, a set of simple distributed controllers can be derived. We describe a partitioned GPF particle filter algorithm for tracking in the following section and block-level pipelining in Section 3.

$$(\mu_x, \mu_{V_x}, \mu_y, \mu_{V_y}) = \mathrm{GPF}(z, n_x, n_y)$$

$$(\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y) = \mathrm{GPF}_{\mathrm{DC}}(\mathbf{S}, \mu_x, \mu_{V_x}, \mu_y, \mu_{V_y}, n_1, n_2, n_3, n_4)$$
$$(x, V_x, y, V_y) = \mathrm{BOT}_S(\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y, n_x, n_y)$$
$$(w, S_M) = \mathrm{BOT}_I(x, y, \tilde{w}, z)$$
$$(\hat{x}, \hat{V}_x, \hat{y}, \hat{V}_y) = \mathrm{BOT}_O(x, V_x, y, V_y, w)$$

$$(\mathbf{Var}) = \mathrm{GPF}_V(x, V_x, y, V_y, w, \mu_x, \mu_{V_x}, \mu_y, \mu_{V_y})$$
$$(\mathbf{Var}) = \mathrm{GPF}_{\mathrm{CH}}(x, V_x, y, V_y, w)$$

Algorithm 2: Processing one observation by a GPF.

$$(\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y) = \mathrm{GPF}_{\mathrm{DC}}(\mathbf{S}, \mu_x, \mu_{V_x}, \mu_y, \mu_{V_y}, n_1, n_2, n_3, n_4)$$
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

Algorithm 3: Algorithm for *drawing conditioning particles* for the bearings-only tracking example.

2.3. GPF particle filter algorithm for tracking

The details of the GPF algorithm are discussed in [18, 20]. In Algorithm 2, a GPF code for processing one observation is presented. Drawing conditioning particles, particle generation, particle update, covariance calculation, mean calculation, and Cholesky decomposition operations are specific for the GPF algorithm. Note that each element of the GPF is in the critical path.

Drawing conditioning particles $(\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y)$ is presented in Algorithm 3. Conditioning particles are drawn from a Gaussian distribution with parameters $(\boldsymbol{\mu}, \mathbf{Var})$, where $\boldsymbol{\mu}$ is 4×1 matrix and $\mathbf{Var}$ is a 4×4 triangular matrix. In order to draw Gaussian random numbers, a matrix $\mathbf{S}$ is necessary, where $\mathbf{Var} = \mathbf{S} \cdot \mathbf{S}^{\top}$.

The generation of particles is performed by drawing them from the importance density in the sampling step. In Algorithm 4, the sampling step for the bearings-only tracking is presented. The input arguments are the states of the particles obtained from the update state step from the previous time instant ($\tilde{\mathbf{X}} = \{\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y\}$). For sake of simplicity

$(x, V_x, y, V_y) = \mathrm{GPF}_{\mathrm{PG}}(\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y, n_x, n_y)$
for $m = 1$–M
$x(m) = \tilde{x}(m) + \tilde{V}_x(m) + 0.5n_x(m)$
$V_x(m) = \tilde{V}_x(m) + n_x(m)$
$y(m) = \tilde{y}(m) + \tilde{V}_y(m) + 0.5n_y(m)$
$V_y(m) = \tilde{V}_y(m) + n_y(m)$
end

Algorithm 4: Particle generation (sampling step).

$(w, S_M) = \mathrm{GPF}_{\mathrm{PU}}(x, y, \tilde{w}, z)$
(i) *calculation of weights and their sum*
$S_M = 0$
for $m = 1$–M

$w^*(m) = \tilde{w}(m)e^{-(2\pi\sigma_y^2)^{-1}\cdot(z-\text{ at an }\frac{y(m)}{x(m)})^2}$
$S_M = S_M + w^*(m)$
end
(ii) *normalization*
for $m = 1$–M
$w(m) = \frac{w^*(m)}{S_M}$
end

Algorithm 5: Particle update (importance step).

of implementation, the prior density of the state is selected as the importance density. The output of the particle generation (sampling step) is a new vector of states $\mathbf{X} = \{x, V_x, y, V_y\}$.

One possible realization of the particle update (importance step) which is used for weight calculations is shown in Algorithm 5. The particle update consists of two substeps: weight calculation and normalization. First, the weights are evaluated up to a proportionality constant and subsequently, they are normalized. The input arguments are the observation z, the arrays of states x and y, and updated weights from the previous time instant $\tilde{w}$.

The covariance calculation is shown in Algorithm 6. The covariance coefficients are first updated in the loop and then the final calculation is performed. Cholesky decomposition is shown in Algorithm 7. The purpose of this step is to calculate the square root of the covariance matrix so that $\mathbf{Var} = \mathbf{S} \cdot \mathbf{S}^{\top}$. 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.

The Computation of the output estimates is shown in Algorithm 8.

3. REALIZATION METHODOLOGY

3.1. Block-level pipelining of loop-based algorithms

The basis of the particle filter design is regular data flow with block-level pipelining. Block-level pipelining is a hardware realization of the data-flow model. Block-level pipelining, which will be elaborated subsequently, guarantees high throughput by maintaining operation concurrency among the processing blocks. The design incorporates block-like operations involved in particle filter by introducing two-level

$(\mathbf{Var}) = \mathrm{GPF}_V(x, V_x, y, V_y, w, \mu_x, \mu_{V_x}, \mu_y, \mu_{V_y})$
for $m = 1$–M
$\mathrm{Var}_{11} = \mathrm{Var}_{11} + x(m) \cdot x(m) \cdot w(m)$
$\mathrm{Var}_{12} = \mathrm{Var}_{12} + x(m) \cdot V_x(m) \cdot w(m)$
$\mathrm{Var}_{13} = \mathrm{Var}_{13} + x(m) \cdot y(m) \cdot w(m)$
$\mathrm{Var}_{14} = \mathrm{Var}_{14} + x(m) \cdot V_y(m) \cdot w(m)$
$\mathrm{Var}_{22} = \mathrm{Var}_{22} + V_x(m) \cdot V_x(m) \cdot w(m)$
$\mathrm{Var}_{23} = \mathrm{Var}_{23} + y(m) \cdot V_x(m) \cdot w(m)$
$\mathrm{Var}_{24} = \mathrm{Var}_{24} + V_x(m) \cdot V_y(m) \cdot w(m)$
$\mathrm{Var}_{33} = \mathrm{Var}_{33} + y(m) \cdot y(m) \cdot w(m)$
$\mathrm{Var}_{34} = \mathrm{Var}_{34} + y(m) \cdot V_y(m) \cdot w(m)$
$\mathrm{Var}_{44} = \mathrm{Var}_{44} + V_y(m) \cdot V_y(m) \cdot w(m)$
end
$\mathrm{Var}_{11} = \mathrm{Var}_{11} - \mu_x \cdot \mu_x$
$\mathrm{Var}_{12} = \mathrm{Var}_{12} - \mu_{V_x} \cdot \mu_x$
$\mathrm{Var}_{13} = \mathrm{Var}_{13} - \mu_y \cdot \mu_x$
$\mathrm{Var}_{14} = \mathrm{Var}_{14} - \mu_{V_y} \cdot \mu_x$
$\mathrm{Var}_{22} = \mathrm{Var}_{22} - \mu_{V_x} \cdot \mu_{V_x}$
$\mathrm{Var}_{23} = \mathrm{Var}_{23} - \mu_y \cdot \mu_{V_x}$
$\mathrm{Var}_{24} = \mathrm{Var}_{24} - \mu_{V_y} \cdot \mu_{V_x}$
$\mathrm{Var}_{33} = \mathrm{Var}_{33} - \mu_y \cdot \mu_y$
$\mathrm{Var}_{34} = \mathrm{Var}_{34} - \mu_{V_y} \cdot \mu_y$
$\mathrm{Var}_{44} = \mathrm{Var}_{44} - \mu_{V_x} \cdot \mu_{V_y}$

Algorithm 6: Algorithm for *covariance calculation* for the bearings-only tracking.

$(\mathbf{S}) = \mathrm{GPF}_{\mathrm{CH}}(x, V_x, y, V_y, w)$
$S_{11} = (\mathrm{Var}_{11})^{1/2}$
$S_{12} = \mathrm{Var}_{12}/S_{11}$
$S_{13} = \mathrm{Var}_{13}/S_{11}$
$S_{14} = \mathrm{Var}_{14}/S_{11}$
$S_{22} = (\mathrm{Var}_{22} - S_{12} \cdot S_{12})^{1/2}$
$S_{23} = (\mathrm{Var}_{23} - S_{12} \cdot S_{13})/S_{22}$
$S_{24} = (\mathrm{Var}_{24} - S_{12} \cdot S_{14})/S_{22}$
$S_{33} = (\mathrm{Var}_{33} - S_{13} \cdot S_{13} - S_{23} \cdot S_{23})^{1/2}$
$S_{34} = (\mathrm{Var}_{34} - S_{13} \cdot S_{14} - S_{23} \cdot S_{24})/S_{33}$
$S_{44} = (\mathrm{Var}_{44} - S_{14} \cdot S_{14} - S_{24} \cdot S_{24} - S_{34} \cdot S_{34})^{1/2}$

Algorithm 7: Algorithm for Cholesky decomposition.

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

Algorithm 8: Calculation of mean (computation of estimates).

pipelining, that is, fine-grained (register-based) and block-level (buffer-based) pipelining.

The main purpose of pipelining in block level is to insert a buffer with associated controller in order to encapsulate any architectural parameters such as latency and rate difference between a pair of processing blocks. Through block-level pipelining, we can achieve three key objectives. First, it is possible to maintain concurrency of each processing block while providing correct synchronization between processing blocks for proper execution. Second, since the control signals, data, and clock become local, hardware implementation is much easier in terms of maintaining performance

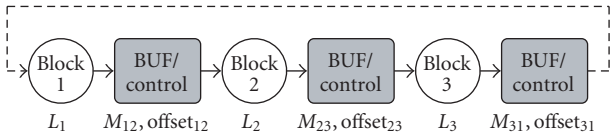

Figure 2: Illustration of the block-level pipelining structure of data flow. A possible recursion is also illustrated with dotted connection between the processing blocks 3 and 1.

by minimizing clock skews and data routing. Any change in logic will only affect its buffer configuration and controller so that reconfigurable design and/or core reuse is possible. Third, since entire design is centered around the buffers, performance mismatches between memory and logics are minimized. These objectives are basis of the buffer-centric design methodology that we are presenting in this paper.

Basically, data dependencies that may arise due to rate difference are handled by inserting a buffer between any pair of processing blocks as shown in Figure 2. To provide maximum flexibility of the controller, we simplify the design by assuming that all the components are controlled and synchronized by a single global clock. Buffers (BUF) are used as a basic block-level pipelining elements. Data are transferred by the read and write access of these buffers concurrently but at different address locations. The amount of difference in write and read address can be determined from the rate differences of processing blocks as well as data dependency of functions. Since the read and write operations are done using different addresses, there will be no conflict of address and concurrency among processing blocks. Each BUF$_i$ 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. One key benefit of the block-level pipelining is that each block can execute concurrently. Such systematic view of the buffers provides an opportunity for us to create memory centric platform.

Consider a set of parameters, denoted by L_i, M_{ij}, and offset$_{ij}$, indicated in Figure 2. In the figure, three processing blocks have different latencies and the same number of data is specified in any pair of processing blocks. The values of latency are obtained from the implementation of the processing blocks, the buffer write-read offset is determined from the operational data dependency of the pair of processing blocks, and the data size M_{ij} is obtained from the function specification of the processing blocks. Each processing block in Figure 2 produces and consumes data at the same rate. The buffer size must be larger than the value of the corresponding offset$_{ij}$.

We illustrate the operation of block-level pipelining. Consider the nonrecursion case where the input comes to the processing block 1 and the output is generated by the processing block 3. First, the data will be generated by the processing block 1 after latency L_1. Then, M_{12} data will be serially written to the first buffer. After offset$_{12}$, the processing block 2 will start processing and generating data for the second buffer after latency L_2. A similar process takes place at the other processing blocks and buffers. Note that for nonrecursive block-level pipelining, the last buffer is not needed. As long as data dependency is enforced, all the processing blocks are concurrently processing data without any stoppage.

3.2. Parameter extraction for controller design

The operations in block-based processing are viewed as buffer-to-buffer operations with coarse-grained processing blocks operating in between them. The primary parameters which decide the controller structure and overall physical realization are as follows.

(1) Logic latency (L_i). This is the latency of the processing block that needs to be handled so that data validity is preserved. Fine grain pipelining introduces this latency, where L_i is the pipeline depth of the data producing block.
(2) Write offset (nw_{ij}). This is the offset between read of the previous buffer and write of the current buffer after eliminating the logic latency. This is to support a producing processing block with delay data generation.
(3) Read offset (nr_{ij}). This is the offset between write into and read from buffer. This value actually represents the data dependency between the producer and the consumer. We consider a dual port buffer in which the value of nr_{ij} represents the difference between write and read of the buffer.
(4) Delay factor (D_{ij}). This is a delay factor of read of the current buffer and write of the current buffer in case of rate mismatch or irregular data stream.
(5) Block size (M_{ij}). This characterizes the size of data produced and consumed by the processing blocks. It, along with nr_{ij}, determines the maximum storage requirement in each buffer.
(6) Consuming rate (C_i). This is the rate of data consumption by the ith processing block.
(7) Producing rate (P_i). This is the rate of data produced by the ith processing block.
(8) Processing rate (F_i). This is the processing speed of the ith processing block.

The parameters (M_{ij}, nr_{ij}, nw_{ij}) are derived from the functional (algorithmic) data-flow description. The parameters (L_i, C_i, P_i, F_i, D_{ij}) are derived from the implementation of the processing blocks. From these two sets of parameters, we can create two tables: node information table (NIT) and edge information table (EIT). Figure 3 illustrates the usage of such tables in the design. A buffer controller will be created for each entry in the EIT. From these two tables, the buffer controller and global controller are designed. The generation of such tables will be discussed later.

3.3. Buffer controller design and synchronization

In the design methodology, the buffer controller is a key element. A block diagram of the buffer controller is shown in Figure 4. The buffer controller consists of concurrent controller and a dual-ported memory. The concurrent controller

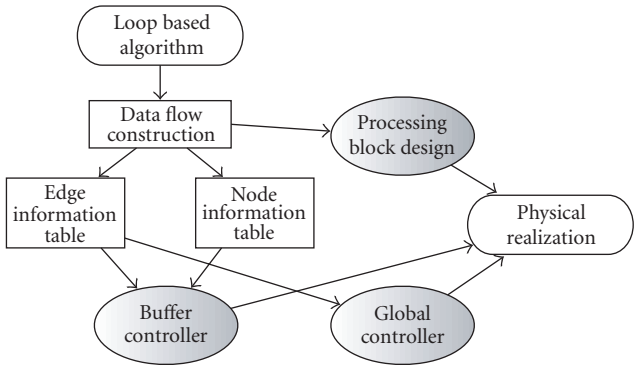

Figure 3: Design flow of the controller and overall physical realization.

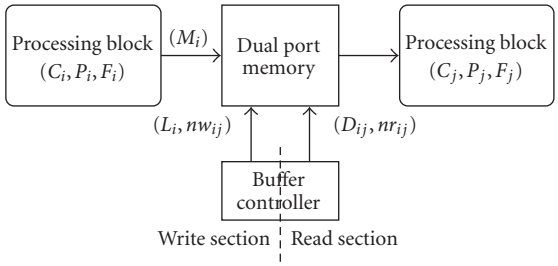

Figure 4: Block diagram of a buffer controller consisting of read and write processes.

has two logic sections: read and write controls [21]. These two controls may be driven by separate clocks and programmable so that the same buffer controller is used in the design by varying the parameters. Basically, the write logic section is configured by L_i and nw_{ij}, and the read logic section is configured by D_{ij} and nr_{ij}. Note that these parameters are derived from the data-flow structure and the processing blocks. Thus, interdependency between the buffer controllers is not necessary to operate correctly and they are autonomous and local.

Consider the buffer controller i. 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 are valid at the buffer controller after waiting for L_i. 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 start to be written to the buffer, the read process starts by the read logic section. The parameter nr_{ij} represents offset between writing

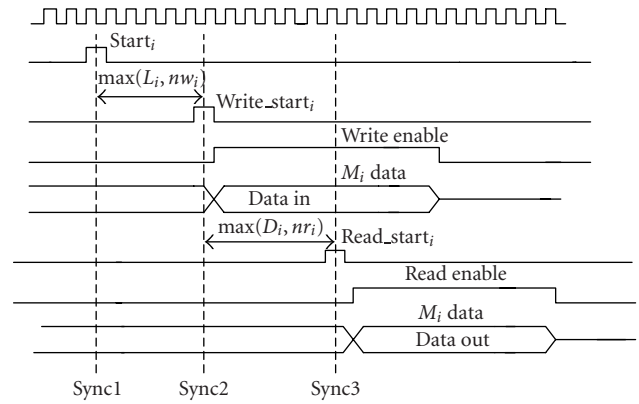

Figure 5: Illustration of buffer controller timing.

and reading the data from the buffer. This parameter supports data dependency. Even if there is no data dependency, it is also possible that the generation data rate of the producer is different from the consuming data rate of the consumer. To support 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 is read from the buffer. Once all the data are read out from the buffer, the buffer controller suspends itself until the buffer controller gets a signal from the outside. 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.

Figure 5 illustrates the timing relationship of a buffer controller where the index ij represents a buffer controller placed between the ith and jth processing blocks. Note that there are three key synchronization points, among the buffer controllers, indicated by sync1, sync2, sync3. These three synchronization points correspond to start_time_{ij}, write_start_{ij}, and read_start_{ij}. The start of the write waiting process is synchronized with the start read process of the previous buffer controller, indexed as ki. And the start of the read process is synchronized with the start of the write waiting process of the same buffer controller. These are the conditions that must be satisfied in order to derive the parameters of the NIT. They have the following relationships:

$$\begin{aligned} \text{start\_time}_{ij} &= \text{read\_start}_{ki}, \\ \text{write\_start}_{ij} &= \text{start\_time}_{ij} + L_i + nw_{ij}, \\ \text{read\_start}_{ij} &= \text{write\_start}_{ij} + \max\left[nr_{ij}, D_{ij}\right]. \end{aligned} \tag{4}$$

Thus, we can synchronize the processing blocks and vary the execution time using these parameters. Moreover, we can delay the entire operation by adjusting the appropriate value of nr_{ij}.

3.4. Relationship between buffer controllers

However, since it is necessary to synchronize each buffer controller, we illustrate the relationship between the global controller and the buffer controllers. An overview of the overall

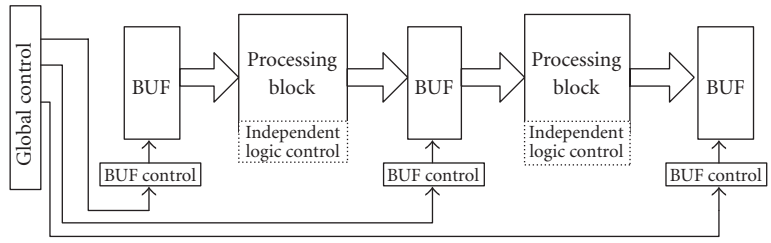

Figure 6: Illustration of interaction between global controller and local buffer controllers.

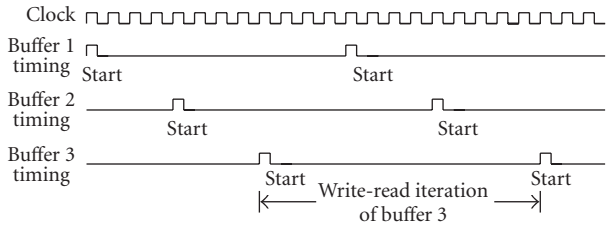

Figure 7: Relative timing of the buffer controllers. All buffer controllers are synchronized with a global clock.

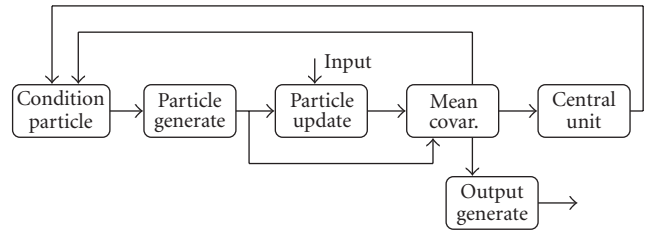

Figure 8: Data-flow graph of the GPF.

control scheme illustrating interaction with the global control and buffer control is shown in Figure 6. The global control generates a single clock to synchronize all the buffer controllers. In addition to the global clock, the global control generates basic timing information for each buffer controller.

Figure 7 illustrates that each buffer controller is controlled by periodic signals, indicated by start, generated from the global controller. The global controller is a counter that generates a set of start signals. This is illustrated in Figure 7. The figure shows three such timing signals for three buffers in Figure 6. For each buffer controller, the start timing signal indicates the start of one iteration process of the buffer controller. Successive buffer controllers are separated by start signals. The time durations between each start signal are controlled by the parameters described in the previous section.

4. GAUSSIAN PARTICLE FILTER DESIGN

4.1. Processing blocks in GPF design

In the design, we first define the operation of each processing block, and then the data-flow structure. Each processing block may have its own local controller for its operation. From the processing block design and the data-flow structure, we derive EIT and NIT. Finally, the buffer controllers and global controller are derived and designed. Although maintaining computation accuracy is very important, we do not consider the finite precision effects of the filters. Instead, we focus our study on the execution timing relationship between the processing blocks and controllers. The actual finite precision effects can be incorporated later into the design without affecting the overall controller.

The GPF filter has 5 major computational units: condition particle generation, particle generate, particle update, mean and covariance calculation, and central unit. Figure 8 shows the actual data-flow graph of the GPF under consideration. In [20], it is shown that the loops in the GPF can be fused. So, all the steps except Cholesky decomposition and variance calculation in the second part of Algorithm 8 can be executed inside one loop of M iterations. Cholesky decomposition and final variance calculation are sequential and their complexity is fixed and does not depend on the number of particles.

Condition particle generation (CPG)

In the CPG processing block, 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 the triangular 4×4 matrix, so that the number of data that is transferred from the CU processing block is 10 (not 16). All the multipliers are pipelined and they operate concurrently producing M conditioning particles. Since the outputs $(\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y)$ are computed using different number of operators, we have to introduce additional delay which is different for each state in order to get all the conditioning particles at the same time instant at the output. The CPG requires 4 random number generators [22].

In the CPG processing block, there are 2 input buffers for $(\boldsymbol{\mu}, \mathbf{S})$ from the CU processing block and 4 output buffers for $(\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y)$ to the PG processing block. The data size of mean $\boldsymbol{\mu}$ is 4 and of the 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.

Particle generate (PG)

In the PG processing block, there are 4 buffers associated with the input vectors $(\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y)$ and 4 buffers associated with

the output vectors (x, V_x, y, V_y). The input vectors stored in the input buffers are generated by the CPG processing block. In addition, 2 more buffers are associated with (x, y) to be used by the PU processing block. The data sizes of inputs and outputs are M. Because they are vectors, the buffer controller parameters will be identical.

The arithmetic operations of the particle generation are described in Algorithm 3. The outputs are computed in parallel operations. In the PG processing block, there are 4 noise generators. For the generation of noise samples, we use Box-Muller approach for efficient FPGA implementation [22]. The noise generation is a combination of a lookup table and arithmetic logic such that it eliminates large latency generated by the arithmetic logic unit.

Particle update (PU)

The arithmetic operations of the *particle update* are illustrated in Algorithm 5. The main arithmetic operations are multiplications, divisions, the trigonometric function $\arctan(\cdot)$, and the exponent function $\exp(\cdot)$. Unrolled CORDIC [23, 24] is used as the operator for $\arctan(\cdot)$ and $\exp(\cdot)$ for its regular implementation structure. For each $\arctan(\cdot)$ operation, a constant value $\pi/2$ and a multiplexer are used to correct the angle-fault problem in the algorithm for $\arctan(\cdot)$. The purpose of this correction is to resolve ambiguity from $(-x, y)$ and $(x, -y)$ (i.e., $\arctan(\cdot)$ value is identical even though their physical locations are different). Since there is no data dependency between the PG and PU processing blocks, the PU processing block will compute its outputs as soon as the data are available in the input buffers.

The PU processing block also takes $z(n)$ as an external observation input. During the iteration n, the value of $z(n)$ does not change. The PU processing block computes the rest of weight computation process as illustrated in Algorithm 4. These weights are not normalized, and they are accumulated to generate sum (S_M notation is used in the algorithm) at the end of weight calculation. The sum is used in the weight normalization step in the MCC 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 blocks will be added to the last buffer controller.

Mean and covariance calculate (MCC)

In the MCC processing block, the partial covariance 4×4 matrix **Var** and 4×1 mean vector $\boldsymbol{\mu}$ are calculated. The MCC computes only the first part (loop) of Algorithm 6 and the rest is computed in the CU processing block. The number of multiplication operations during mean and covariance operation is equal to $N_s + N_s(N_s + 1)/2 = (N_s^2 + 3N_s)/2$ for the GPF, where N_s represents the dimensionality of the model. For the bearings-only tracking problem with $N_s = 4$, it is 14 multiplications. All 14 outputs are accumulated, so that 14 accumulators are necessary. All the blocks operate concurrently on M particles.

In the MCC processing block, there are 6 input buffers for (x, V_x, y, V_y) from the PG processing block and (w, sum) from the PU processing block. The mean vector $\boldsymbol{\mu}$ is normalized by the sum at the end of operation. This requires only 4 divisions for the mean vector. There are 2 output buffers for $\boldsymbol{\mu}$ and **Var** to the CU processing blocks. These outputs are serialized. The μ from the MCC processing block is the output of the GPF. Thus, the output generation block for the GPF is to convert serially the received $\boldsymbol{\mu}$ to a parallel format.

Central unit (CU)

The inputs and the outputs of the CU are produced once during the sampling period. The CU processing block executes Algorithm 7. In addition, the CU processes the second part of Algorithm 6. Because of special functions such as division and square root operations, we design for time-multiplexing of operators. Since small number of data is to be computed, the delay incurred by this unit is not significant. The outputs are buffered within the processing block before being read out to the buffer controller for synchronization purposes. In the CU processing block, there are 2 input buffers for $(\boldsymbol{\mu}, \mathbf{Var})$ from the MCC processing block. There is 1 buffer for **S** to the CPG processing block. These outputs are serialized.

Output generation (OG)

The OG processing block is a simple serial-to-parallel transformation logic.

4.2. Concurrency and execution time of GPF

The timing diagram illustrating the major block operations is shown in Figure 9. A value n shown in the box indicates iteration index. As shown in the timing diagram, the executions of the CPG, PG, PU, MCC, and CU processing blocks are overlapped. When the CPG processing block starts its execution, the buffer will start to write after L_{CPG}. As soon as the first data is written, the buffer will start the read for the PG processing block. The PU processing block can start and the MCC processing block will follow. Since MCC has to normalize the mean, the CU waits for M to wait for the value sum. As soon as the CU generates the data, the next iteration of the CPG can proceed. The external input is synchronized with the start of PU. The output is generated from the MCC processing block. Thus, the minimum iteration period of the GPF is $T_{\text{GPF}} = (M + L_{\text{GPF}}) \cdot T_{\text{clk}}$, where $L_{\text{GPF}} = L_{\text{CPG}} + L_{\text{PG}} + L_{\text{PU}} + L_{\text{MCC}} + L_{\text{CU}}$, where L_{CU} includes latency due to hardware and delayed output generation resulting from time-multiplexing within the CU processing block.

5. PHYSICAL MAPPING

5.1. GPF realization

The data-flow graph of GPF for bearings-only tracking problem is constructed as shown in Figure 10. The figure shows both the processing blocks and the buffers.

Table 1 tabulates the primary parameters of each processing block for the GPF. Similarly, the GPF speed is limited by the speed of the CORDICs in the PU.

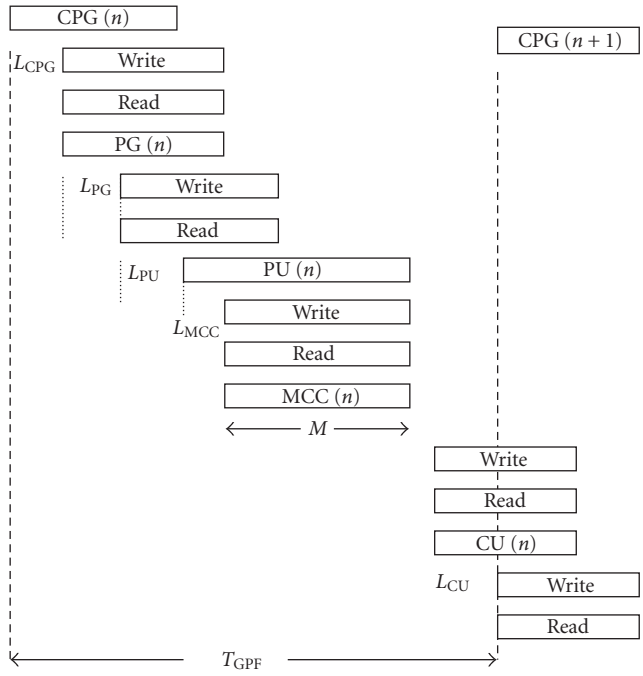

Figure 9: Timing diagram of the GPF.

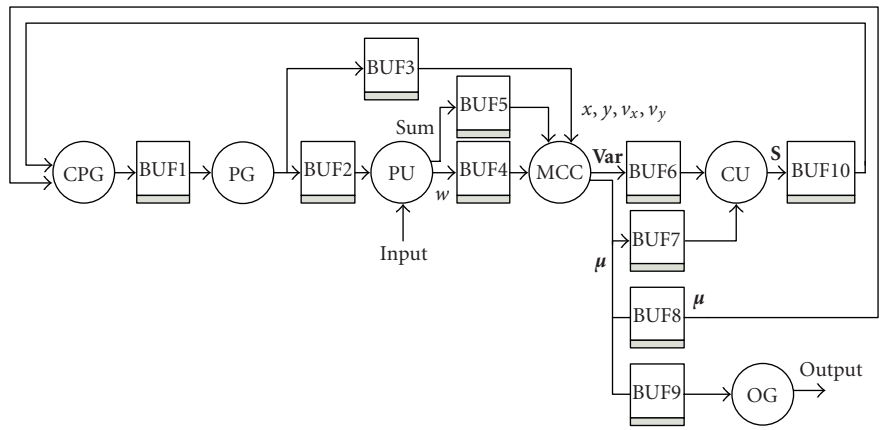

Figure 10: Data-flow graph (with buffers and processing blocks) of GPF.

Table 1: Node information table for GPF. 206 MHz is selected for the global clock f_{clock}.

Node	L	C (MHz)	P (MHz)	F (MHz)	FPGA (%)
N1 (CPG)	11	206	206	206	22.2
N2 (PG)	8	206	206	206	7.6
N3 (PU)	43	206	206	206	20.7
N4 (MCC)	8	206	206	206	31.7
N5 (CU)	1	206	206	206	17.8

Table 2 tabulates the data dependency between a pair of processing blocks. In the table, the multiple appearance of source and destination nodes indicates that there is more than one data connection with different characteristics. In the table, $nr_{E3} = 47$, which is the sum of $nr_{E2} + L_{PU} + nw_{E4} + nr_{E4}$. This is because the buffer has already written the data generated by the PG processing block but will delay the read operation for the MCC processing block for data

Table 2: Edge information table for GPF. Each edge requires a buffer.

Edge	Data	nr_{Ei}	nw_{Ei}	M_{Ei}	D_{Ei}
$E1$ (CPG–PG)	$(\tilde{x}, \tilde{V}_x, \tilde{y}, \tilde{V}_y)$	1	1	M	1
$E2$ (PG–PU)	(x, y)	1	1	M	1
$E3$ (PG–MCC)	(x, V_x, y, V_y)	47	1	M	1
$E4$ (PU–MCC)	(w)	1	2	M	1
$E5$ (PU–MCC)	(sum)	1	$M+1$	1	1
$E6$ (MCC–CU)	(**Var**)	1	M	10	1
$E7$ (MCC–CU)	$\boldsymbol{\mu}$	1	M	4	1
$E8$ (MCC–CPG)	$\boldsymbol{\mu}$	78	M	4	1
$E9$ (MCC–OG)	$\boldsymbol{\mu}$	1	M	4	1
$E10$ (CU–CPG)	(**S**)	1	75	10	1

Table 3: Parameters for buffer controllers for the GPF. Each value in parenthesis represents the actual counting parameter for the buffer controller.

Start	Time instant	Write start	Read start
Start1	0	start1 + $L_{\text{CPG}} + nw_{E1}$ (12)	start1 + $L_{\text{CPG}} + nw_{E1} + nr_{E1}$ (13)
Start2	start1 + $L_{\text{CPG}} + nw_{E1} + nr_{E1}$ (13)	start2 + $L_{\text{PG}} + nw_{E2}$ (22)	start2 + $L_{\text{PG}} + nw_{E2} + nr_{E2}$ (23)
Start3	start1 + $L_{\text{CPG}} + nw_{E1} + nr_{E1}$ (13)	start3 + $L_{\text{PG}} + nw_{E3}$ (22)	start3 + $L_{\text{PG}} + nw_{E3} + nr_{E3}$ (69)
Start4	start2 + $L_{\text{PG}} + nw_{E2} + nr_{E2}$ (23)	start4 + $L_{\text{PU}} + nw_{E4}$ (68)	start4 + $L_{\text{PU}} + nw_{E4} + nr_{E4}$ (69)
Start5	start2 + $L_{\text{PG}} + nw_{E2} + nr_{E2}$ (23)	start5 + $L_{\text{PU}} + nw_{E5}$ ($M+67$)	start5 + $L_{\text{PU}} + nw_{E5} + nr_{E5}$ ($M+68$)
Start6	start4 + $L_{\text{PU}} + nw_{E4} + nr_{E4}$ (69)	start6 + $L_{\text{MCC}} + nw_{E6}$ ($M+77$)	start6 + $L_{\text{MCC}} + nw_{E6} + nr_{E6}$ ($M+78$)
Start7	start4 + $L_{\text{PU}} + nw_{E4} + nr_{E4}$ (69)	start7 + $L_{\text{MCC}} + nw_{E7}$ ($M+77$)	start7 + $L_{\text{MCC}} + nw_{E7} + nr_{E7}$ ($M+78$)
Start8	start4 + $L_{\text{PU}} + nw_{E4} + nr_{E4}$ (69)	start8 + $L_{\text{MCC}} + nw_{E8}$ ($M+77$)	start8 + $L_{\text{MCC}} + nw_{E8} + nr_{E8}$ ($M+155$)
Start9	start4 + $L_{\text{PU}} + nw_{E4} + nr_{E4}$ (69)	start9 + $L_{\text{MCC}} + nw_{E9}$ ($M+77$)	start9 + $L_{\text{MCC}} + nw_{E9} + nr_{E9}$ ($M+78$)
Start10	start6 + $L_{\text{MCC}} + nw_{E6} + nr_{E6}$ ($M+78$)	start10 + $L_{\text{CU}} + nw_{E10}$ ($M+154$)	start10 + $L_{\text{CU}} + nw_{E10} + nr_{E10}$ ($M+155$)
Reset	$M+155$	—	—

synchronization. For $E4$ and $E5$, nw_{E4} and nw_{E5} are 2 and $M+1$, respectively. At $E8$, the read operation for the CPG will be delayed by nr_{E8} which is $nr_{E6} + L_{\text{CU}} + nw_{E10} + nr_{E10} = 78$. This will synchronize both $\boldsymbol{\mu}$ and **S** at the CPG processing block. At $E10$, $nw_{E10} = L_{\text{CU}_t} = 75$, which corresponds to the time taken by the CU processing block to generate the first data. For $E6$, $E7$, $E8$, and $E9$, $nw_{Ei} = M$. The values of D_{Ei} are all 1 since there is no rate mismatch.

Table 3 summarizes the parameters of all buffer controllers. These parameters are derived from the above two tables. For each buffer, the start time of each buffer controller and period of the buffer iteration are illustrated. The parameters for the start time are computed with respect to f_{clock}. The external input is synchronized with the start of the PU1.

Note that there are several key synchronization points. First, the buffer controllers for $E6$ and $E7$ have identical read beginning. Second, the buffer controllers for $E8$ and $E10$ also have identical beginning of the read. Third, the buffer controllers for $E7$, $E8$, and $E9$ have identical write beginning. Fourth, the buffer controllers for $E2$ and $E3$ have identical start time and write beginning. Fifth, the buffer controllers for $E3$ and $E4$ have the same read beginning. The iteration period is indicated by the reset time instant. From the previous discussion on timing diagram, the iteration period was $M + L_{\text{GPF}} = M + 155$ including the CU computation latency, and the values of nr_{Ei} and nw_{Ei} in the critical path.

Table 4 illustrates buffer controller usages and configuration parameters for the global controller. The buffer size for each buffer controller is also indicated. The factor of 4

Table 4: Synchronization parameters for the global controller. $M = 1024$ is assumed and the value of M changes the buffer controller 10 and the value of reset.

Buffer controller	Buffer size	Control parameter $(msb, \ldots, lsb)$
1	4 (+4)	000000000000
2	2 (+2)	000000001101
3	4 (+4)	000000001101
4	1 (+1)	000000010111
5	1 (+1)	000000010111
6	1 (+1)	000001000101
7	1 (+1)	000001000101
8	4 (+1)	000001000101
9	1 (+1)	000001000101
10	1 (+1)	010001001111
Reset	—	010010011001

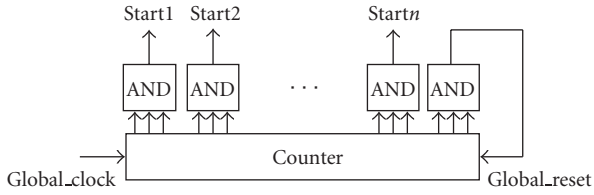

Figure 11: Illustration of the global controller structure.

implies that the vector of data (i.e., (x, V_x, y, V_y)) is controlled by one buffer controller. The maximum total amount of buffer used for the synchronization is bounded by about $4M$ where M is the number of particles used in the filtering. Note that the actual buffer size required by each buffer controller is bounded by $\min(nr_{Ei}, M_{Ei})$. Thus, $E3$ actually used a much smaller buffer than the maximum size indicated in the table. The global controller is a counter that generates start signals indicated by the time instants in Table 3. The entire process of generating the start signal repeats with periodicity indicated by the reset value. A block diagram of the global controller is illustrated in Figure 11. The input to each AND gate is a binary representation of the start time instant of each buffer controller. If $M = 1024$, the size of the counter is 11 bits. After each period, the global controller is reset and starts the operation over again. The control parameters are represented in binary number which will be used as inputs of the AND gate to generate the start signal at the correct time. Assuming that M=1024, we need a 12-bit counter for the global controller. The value of M changes the buffer controller 10 and the value of *reset*.

5.2. Design analysis

The execution performance of the particle filter, whose design is based on the proposed methodology, is kept at the maximum by fully exploiting operational concurrency. As discussed in the data-flow, most operations are fully fused to minimize possible latency and data dependency. A simple data transfer is necessary within each processing block.

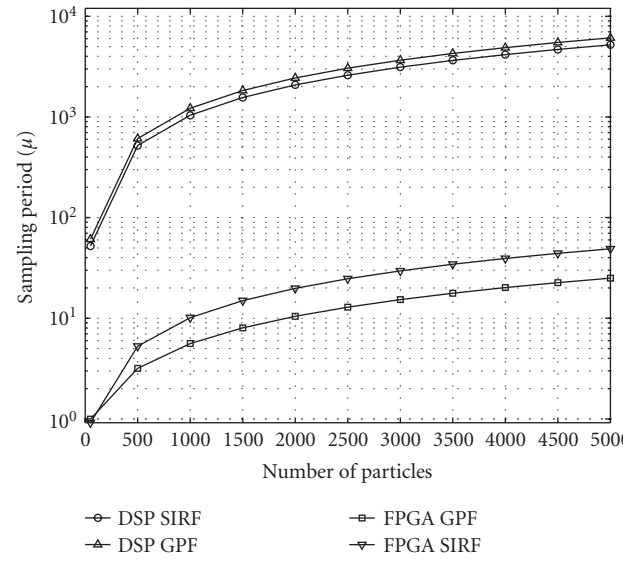

Figure 12: Performance comparison of different realizations. FPGA represents the design based on the proposed method and DSP represents the design based on DSP.

When concurrency is fully exploited, as illustrated in the timing diagram, the sampling period increases almost linearly with the number of particles for $M = \{500, 1000, 5000\}$. The sampling periods of the GPF are 3.14, 5.55, and 24.8 microseconds, respectively. The execution performance in terms of sampling period is evaluated for four different realizations. The comparison is plotted in Figure 12. The GPF and sample importance filter (SIRF) particle filtering algorithms are considered [25]. The two sets of curves show the sampling period versus the number of executed particles. When it is compared to DSP processors such as Analog Device TigerSharc DSP (similar performance is achieved with Texas Instruments DSP processors), the GPF realization is about 100 times faster mainly due to its operational concurrency. While the DSPs provide some degree of parallelism and

concurrency, both parallelism and pipelining cannot be fully exploited. When the performances of the GPF and the SIRF are compared, the GPF is approximately twice faster than that of SIRF when the number of particles is more than 1000. This is because the performance of the SIRF, when the proposed methodology is followed, is proportional to $2M$ where the GPF is proportional to M. However, the SIRF and the GPF are close in performance since latency of the GPF due to pipelining is much larger than that of SIRF. When both algorithms are executed on DSP, their performances are very close. While the SIRF takes more time when concurrency is exploited in the FPGA design, the GPF takes more cycles due to complicated arithmetic computations.

The proposed block-level pipelining design maximizes buffer controller usages and minimizes dynamic reconfiguration efforts. The design does not suffer from the amount of memory used for buffers since we can view those buffers as a collection of pipeline registers (i.e., these registers are needed in any design with standard design flow whether the design is based on distributed or centralized schemes). These pipeline registers cannot be eliminated if the maximum throughput is of utmost concern. Since we are eliminating the use of individual pipeline register, lower overall clock loading can be achieved.

Particle filter based on the proposed methodology supports parameter changes during run-time because the design methodology ensures flexibility. Moreover, the design can be extended to support wide ranges of particle filtering including the parallel operation. One biggest novelty of the architecture is that the buffer controller guarantees correct operation while maintaining maximum throughput. Moreover, because of very small 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 cycles but as low as one cycle simultaneously).

6. CONCLUSIONS

This paper introduces a simple design methodology of generating an overall controller for a particle filter realization. We have demonstrated that the proposed method is very effective when the data-flow structure is well defined by taking into account their execution characteristics. The entire design followed-block-level pipelining approach where controllers can easily be derived from the data flow and the parameters of processing blocks. We have considered GPF for validating our design methodology. The overall temporal correctness has been verified by Verilog and SystemC. The method is very important because the processing block execution characteristics can be changed with minimal change in the controller structure.

ACKNOWLEDGMENT

This work has been supported by the NSF under Award CCR-0220011.

REFERENCES

[1] N. J. Gordon, D. J. Salmond, and A. F. M. Smith, "Novel approach to nonlinear/non-Gaussian Bayesian state estimation," *IEE Proceedings-F: Radar, Sonar and Navigation*, vol. 140, no. 2, pp. 107–113, 1993.

[2] J. Carpenter, P. Clifford, and P. Fearnhead, "Improved particle filter for nonlinear problems," *IEE Proceedings-F: Radar, Sonar and Navigation*, vol. 146, no. 1, pp. 2–7, 1999.

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

[4] C. Musso, N. Oudjane, and F. L. Gland, "Improving regularized particle filters," in *Sequential Monte Carlo Methods in Practice*, A. Doucet, N. de Freitas, and N. Gordon, Eds., pp. 247–271, Springer-Verlag, New York, NY, USA, 2001.

[5] N. Oudjane and C. Musso, "Progressive correction for regularized particle filters," in *Proc. IEE Proceedings of the 3rd International Conference on Information Fusion*, vol. 2, Paris, France, July 2000.

[6] 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.

[7] R. van der Merwe, A. Doucet, N. de Freitas, and E. Wan, "The unscented particle filter," in *Advances in Neural Information Processing Systems 13 (NIPS)*, MIT Press, Cambridge, Mass, USA, December 2000.

[8] 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, no. 1, pp. 127–146, 2001.

[9] P. M. Djurić, J. H. Kotecha, J. Zhang, et al., "Particle filtering," *IEEE Signal Processing Mag.*, vol. 20, no. 5, pp. 19–38, 2003.

[10] R. Chen, X. Wang, and J. S. Liu, "Adaptive joint detection and decoding in flat-fading channels via mixture Kalman filtering," *IEEE Trans. Inform. Theory*, vol. 46, no. 6, pp. 2079–2094, 2000.

[11] T. Ghirmai, J. H. Kotecha, and P. M. Djurić, "Blind equalization for time-varying channels and multiple sample processing using particle filtering," *Digital Signal Processing*, vol. 14, pp. 312–331, 2004.

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

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

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

[15] J. Dalcolmo, R. Lauwereins, and M. Ade, "Code generation of data dominated DSP applications for FPGA targets," in *Proc. IEEE International Workshop on Rapid System Prototyping (RSP '98)*, pp. 162–167, Leuven, Belgium, June 1998.

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

[17] A. Doucet, S. J. Godsill, and C. Andrieu, "On sequential Monte Carlo sampling methods for Bayesian filtering," *Statistics and Computing*, vol. 10, no. 3, pp. 197–208, 2000.

[18] J. H. Kotecha and P. M. Djurić, "Gaussian particle filtering," in *Proc. IEEE Workshop on Statistical Signal Processing (SSP '01)*, Singapore, August 2001.

[19] J. H. Kotecha, *Monte Carlo for dynamic state space models with applications to communications*, Ph.D. thesis, Stony Brook University, Stony Brook, NY, USA, December 2001.

[20] M. Bolic, A. Athalye, P. M. Djurić, and S. Hong, "A study of Gaussian particle filters for practical physical implementation," submitted to *IEEE Transactions on Circuits and Systems* I.
[21] M. Sadasivam and S. Hong, "Autonomous buffer controller design for concurrent execution in block level pipelined dataflow," in *Proc. IEEE Computer Society Annual Symposium on VLSI Emerging Trends in VLSI Systems Design (ISVLSI '04)*, pp. 303–304, Lafayette, La, USA, February 2004.
[22] 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 International Conference on Electronics, Circuits and Systems (ICECS '00)*, vol. 1, pp. 366–369, Kaslik, Lebanon, December 2000.
[23] J. E. Volder, "The CORDIC trigonometric computing technique," *IRE Transactions Electronic Computers*, vol. 8, no. 3, pp. 330–334, 1959.
[24] S. Wang and V. Piuri, "A unified view of CORDIC processor design," in *Application Specific Processors*, pp. 121–160, Kluwer Academic, Boston, Mass, USA, November 1996.
[25] S. Hong, M. Bolić, and P. M. Djurić, "A design complexity comparison method for loop-based signal processing algorithms: particle filters," in *Proc. International Symposium on Circuits and Systems (ISCAS '04)*, Vancouver, Canada, May 2004.

Sangjin Hong received his B.S. and M.S. degrees in EECS from the University of California, Berkeley. He received his Ph.D. degree in EECS from the University of Michigan, Ann Arbor. He is currently with the Department of Electrical and Computer Engineering, Stony Brook University (SUNY), USA. Before joining SUNY, he worked at Ford Aerospace Corporation Computer Systems Division as a Systems Engineer. He also worked at Samsung Electronics, 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 SoC design and optimization, VLSI signal processing, and low-complexity digital circuits. Professor Hong served on numerous Technical Program Committees for IEEE conferences. Professor Hong is a Senior Member of IEEE.

Petar M. Djurić received his B.S. and M.S. degrees in electrical engineering from the University of Belgrade in 1981 and 1986, respectively, and received his Ph.D. degree in electrical engineering from the University of Rhode Island in 1990. From 1981 to 1986, he was Research Associate with the Institute of Nuclear Sciences, Vinča, Belgrade. Since 1990 he has been with Stony Brook University, where he is a 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. Professor Djurić is the Area Editor of special issues of the IEEE Signal Processing Magazine and Associate Editor of the IEEE Transactions on Signal Processing. He is also member of editorial boards of several professional journals.

Miodrag Bolić received his B.S. and M.S. degrees in electrical engineering from the University of Belgrade, Yugoslavia, in 1996 and 2001, respectively, and received his Ph.D. degree in electrical engineering from Stony Brook University (SUNY), USA. He is currently with the School of Information Technology and Engineering, the University of Ottawa, Canada. From 1996 to 2000, he was a Research Associate with the Institute of Nuclear Science, Vinča, Yugoslavia. From 2001 to 2004 he worked part-time at Symbol Technologies Inc., NY, USA. His research is related to VLSI architectures for digital signal processing and signal processing in wireless communications and tracking.