EURASIP Journal on Advances in Signal Processing
a SpringerOpen Journal

RESEARCH **Open Access**

Proposed hardware architectures of particle filter for object tracking

Howida A Abd El-Halym[1*], Imbaby Ismail Mahmoud[1] and SED Habib[2]

Abstract

In this article, efficient hardware architectures for particle filter (PF) are presented. We propose three different architectures for Sequential Importance Resampling Filter (SIRF) implementation. The first architecture is a two-step sequential PF machine, where particle sampling, weight, and output calculations are carried out in parallel during the first step followed by sequential resampling in the second step. For the weight computation step, a piecewise linear function is used instead of the classical exponential function. This decreases the complexity of the architecture without degrading the results. The second architecture speeds up the resampling step via a parallel, rather than a serial, architecture. This second architecture targets a balance between hardware resources and the speed of operation. The third architecture implements the SIRF as a distributed PF composed of several processing elements and central unit. All the proposed architectures are captured using VHDL synthesized using Xilinx environment, and verified using the ModelSim simulator. Synthesis results confirmed the resource reduction and speed up advantages of our architectures.

1. Introduction

The adoption of particle filters (PFs) in real-time systems is hampered by their computational complexity. The PF typically involves several complex mathematical operations, in addition to the large memory required to store and handle the particles. Parallel processing offers a possible solution to the real-time requirement. However, full parallelization is obstructed by the resampling step, which is serial in nature. Several efforts were expended to construct distributed resampling algorithms [1,2]. Implementations of PF applications on hardware or hardware/software co-design platforms are further challenging due to the resource constraints on such platforms. Design and implementation of a generic yet highly optimized architecture for all PF-based systems is not easy because of the wide range of applications to which particle filtering technique are applied [3-5].

In the following, we review the main directions to the hardware implementation of the PFs, especially for object tracking.

Boli'c [6] proposed architecture for distributed resampling with proportional allocation. The main idea is to store the particles to be routed among the processor elements into dedicated memories in the control unit, and to have very fast interface capable of reading particles from the central unit (CU) and routing them to PEs. The overall memory requirements for this architecture equal KM, where K is the number of PEs and M is the total number of particles.

Athalye et al. [7] presented generic architectures for the implementation of the Sequential Importance Resampling Filter (SIRF). The proposed architecture is based on using dual-port memory. The memory stores the addresses of the particles in its upper half, while the sampled particles are stored in the lower half of the memory. The idea is that the resampling unit returns the set of indexes (pointers) of the replicated particles instead of the particles themselves. Using index addressing alone does not ensure that the scheme with the single memory will work correctly. They used other memories to store the indexes of the replicated particles and the discarded particles. The size of overall used memory is $4M$: $2M$ depth dual-port memory to store the addresses and particles state vectors, M depth memory to store the replicated particles indexes and M depth FIFO to store the discarded particle indexes. They proposed two architectures to implement the SIRF using systematic resampling (SR) and using residual systematic resampling (RSR) algorithms.

* Correspondence: howidaaa@yahoo.com
[1]Nuclear Research Center, Atomic Energy Authority, Cairo, Egypt
Full list of author information is available at the end of the article

Hong et al. [2] proposed a parallel PF consisting of multiple processing elements (PEs) up to four PEs. The PEs are connected with a single CU responsible for resampling. The CU is designed to support both the distributed resampling algorithm with proportional allocation (RPA) and non-proportional allocation (RNA). The proposed resampling architecture reduces the overall resampling time by a factor equal to the number of PEs. The communication between the PEs is via a two-level interconnect. The first level is used for interactions between the PEs and the CU and the second level is exploited for interactions inside the CU. The CU contains buffers RB_i (i = 1:4) to store excess particles which will be transmitted to the different PEs. The size of these buffers is $2M$. Additional memory space is needed to store the tagged particles in tag buffers (TB). $M/5$ memory words are needed for each TB_i (i = 1:4).

Alarcon and Lopez [8] applied PFs for tracking the lines of a road in real time. For each image, the presence of the lane lines is detected and their center of mass is calculated. Three consecutive images frame^0, frame^1, and frame^2 are used for prediction and full tracking of the lane lines. The architecture is designed such that each particle is evaluated and appropriately updated independent of the rest.

Uk Cho et al. [9,10] proposed a PF algorithm specifically designed for object tracking. The architecture consists of five blocks: the particle initiator (particle sampler), coordinates comparator, particle normalizer, data output, and particle selector. The sample of the new position of the particle is done according to the resampled particles and Gaussian white noises from the particle selector and Gaussian distribution Lookup Table (LUT), respectively. They eliminated the division operation of the particle resampling by using a variable resampling range. The particle selector block performs the resampling step in the PF algorithm. It selects new particles among the current particles according to the uniform random distribution obtained from the random distribution LUT and the cumulative function from the cumulative LUT in the particle normalizer block.

Velmurugan [11] developed an FPGA implementation of the bearings-only tracker, similar to that in [7]. He used the Xilinx System Generator tool to speed up the development task. The use of this tool reduced the time spent in developing the system, and provided estimates of the FPGA resources consumed. The implementation lacks a higher-level module to recursively propagate the particles and update the state estimates over time, because this would involve developing a top-level VHDL module to control the memory reads and writes. The System Generator setup is not well suited to design this control operation, so it is not pursued. The implementation introduced performed the computations for M particles in a single iteration of the PF only.

Medeiros et al. [12] focused on the parallel computation of the particle weights of the color-based PF. The architecture is composed of a linear array of PEs, each consisting of an arithmetic logic unit and a small amount of memory, a digital input processor, and a digital output processor. A global control processor is responsible for controlling the operation of the PEs and is able to carry out global DSP operations.

Saha et al. [13] introduced a parameterized design framework for PFs. This general framework allows the system features (e.g., number of particles) to be defined as parameters according to the application considered. The memory banks are parameterized. The proposed architecture consists of an array of processor elements and a resampling unit with a set of parameterized interfaces. The processor element consists of a weight calculation unit, a noise generator, and a processor element core. This approach reduces the re-design effort.

Recently, Hendeby et al. [14] used General-Purpose Computing on Graphics Processing Unit techniques to make a parallel recursive Bayesian estimation implementation using PFs.

The main objective of most of the published studies on the hardware implementation of PFs is to parallelize the resampling step, or simplify it via the use of LUTs. It may be noted, however, that for a moderate number of particles, resampling itself is not computationally expensive. The PF hardware implementation has to consider all of the main four steps: the particle sampling, weighting, resampling as well as the output step. An efficient implementation should efficiently implement all these steps from the perspective of execution time, hardware resources, and robust performance.

Although, the distributed implementation proposed in [6] reduces the execution time to M/K instead of M clock cycles, this architecture uses MK memory locations instead of $2\ M$ in the straightforward implementation. Similarly, the parallel resampling architecture proposed in [2] reduces the overall resampling time by K but this architecture is limited to four PEs. In addition, this architecture uses $3\ M$ memory to store the particles: one memory to store M particles to be resampled, one to store the replicated M particles, and one to store the M particles which are used as inputs in the next sampling step. Uk Cho et al. [9,10] simplified the implementation of PFs by using LUTs; the penalty came from the increase of the implemented area as well as the execution time.

In this article, efficient novel hardware architectures of the SIRF are implemented. Three novel hardware architectures of the SIRF for object tracking are introduced. The first architecture is a two-step architecture with sequential resampling, where particle sample, weight, and output calculations are carried out in parallel during the first step, followed by sequential resampling in the second step. This first architecture serves as a core unit for the next two architectures. The second architecture speeds up the resampling step (second step) via a parallel, rather than a serial, architecture. The third architecture implements the SIRF as a distributed PF composed of several PEs and a CU. Each PE is in fact the two-step core of the first architecture. These architectures aim at enhancing the speed of operation while maintaining, at the same time, efficient utilization of logic, and memory resources. All the proposed architectures are implemented on a FPGA platform. A preliminary report covering the first two architectures was presented at a related conference [15].

2. Particle filters

The PF approximates recursively the sequence of posterior probability measures associated to a state-space dynamic model using a finite set of weighted samples. The key idea is to represent the required posterior density function by a set of random samples with associated weights and to compute estimates based on these samples and weights. The PF consists of two phases: prediction and update. The system is represented by state-space and observation equations. Consider an object that has a state X_t and observation Z_t at discrete time t. The previous state sequence at time t - 1, t - 2,...,2, 1 are denoted as X_{t-1}, X_{t-2},...,X_2 and X_1, respectively. $p(X_t|X_{t-1})$ describes the transition for state vector X_t (dynamic or motion model). Let all available observations be $Z_{1:t-1}$ = {Z_{t-1},..., Z_1}. The prediction uses the probabilistic system transition model to predict the posterior probability distribution at time t. So, we require to construct the probability density function (pdf) $p(x_t|z_t)$ assuming that $p(x_0|z_0) \equiv p(x_0)$ is the initial pdf of the state vector, which is known as the prior (z_0 is the set of no measurements). The posterior density $p(X_{1:t}|Z_{1:t})$ may be obtained recursively in the two stages: prediction and update.

Supposing that the required pdf $p(x_{t-1}|z_{t-1})$ at time t-1 is available, the prediction stage involves using the system model to obtain the prior pdf of the state at time t via the Chapman-Kolmogorov equation [16]

$$p(x_t \mid z_{t-1}) = \int p(x_t \mid x_{t-1})p(x_{t-1} \mid z_{t-1})dx_{t-1} \tag{1}$$

Equation 1 assumes that $p(x_t|x_{t-1}, Z_{1:t-1})$ = $p(x_t|x_{t-1}, Z_{t-1})$. This approximation is particularly useful in the common case when only a filtered estimate of $p(x_t|Z_{1:t})$ is required at each time step [17]. In such scenarios, only x_t need be stored, and so discard the path ($X_{1:t-1}$) and the history of the observations ($Z_{1:t-1}$). All practical software or hardware implementations of the PFs adopt this approximation to avoid intractable computation complexity. Throughout thisarticle, we also adopt this assumption.

At time step t, a measurement z_t becomes available, and this may be used to update the prior (update stage) via Bayes' rule

$$p(x_t \mid z_t) = \frac{p(z_t \mid x_t)p(x_t \mid z_{t-1})}{p(z_t \mid z_{t-1})} \tag{2}$$

where the normalization constant

$$p(z_t \mid z_{t-1}) = \int p(z_t \mid x_t)p(x_t \mid z_{t-1})dx_t \tag{3}$$

The likelihood function $p(z_t|x_t)$ is defined by the measurement model. In the update stage (Equation 2), the measurement z_t is used to modify the prior density to obtain the required posterior density of the current state. The recurrence relations (Equations 1 and 2) form the basis for the optimal Bayesian solution. This recursive propagation of the posterior density is only a conceptual solution in that, in general, it cannot be determined analytically. Extended Kalman filters (EKFs), Gaussian Sum Filters (GSFs) [18], and PFs approximate the optimal Bayesian solution.

At every time instant t, a random measure $\{x_t^m, w_t^m\}_{m=1}^M$ characterizes the posterior pdf $p(x_t|z_t)$, where $\{x_t^m, m = 1, ..., M\}$ is a set of support points (particles) with associated weights $\{w_t^m, m = 1, ..., M\}$. The weights are normalized such that $\sum_{m=1}^{M} w_t^m = 1$. Then, the posterior density function at t can be approximated as

$$p(x_t \mid z_t) \cong \sum_{m=1}^{M} w_t^m \delta(x_t - x_t^m) \tag{4}$$

where $\delta(.)$ is a Dirac delta function. The estimate E(h($X_{1:t}$)), where h(.) is a function of x_t, can be computed from

$$\widehat{E}(h(x_t)) = \sum_{m=1}^{M} w_t^m h(x_t^m) \tag{5}$$

Therefore, we have a discrete weighted approximation to the true posterior $p(x_t|z_t)$ by using the discrete random measures. In addition, the estimation is calculated as a weighted mean.

2.1. SIRFs overview

There are many variants of PEs, derived by an appropriate choice of the importance sampling density function or modification of the resampling step. SIRF is one of the most widely used PFs types. The SIRF has the advantage that the importance weights are easily evaluated and the importance density can be easily sampled. SIRF assumes that

- The state dynamics and measurement functions are known.

- The likelihood function is available.

SIRFs choose the importance density to be the transition prior and perform the resampling step at every time index [17].

For one input sample, SIRF performs the particle generation and the weight calculation. After M particles, the weight normalization, the resample, and finally the output are carried out as shown in Figure 1. To achieve minimum execution time it is required to devise a one-to-one mapping between the particle filtering operations and the hardware resources that allows for utilizing operational concurrency.

2.2. PF for tracking

PFs provide robust tracking for moving objects, especially in the case of nonlinear and non-Gaussian problems. PFs must be designed in a way to avoid the loss of tracking. For our image tracking application, the moving object is expected to remain within a region of interest (ROI) area of 32*32 pixels between two consecutive frames. So, the number of particles needed to represent the state space corresponding to this area is of moderate value.

Bolic et al. [19] provide a performance and complexity analysis of PF as applied to real-time object tracking. They address the effect of the number of particles and the sample rate. They found that the performance of the particles filters ceases to improve when the number of particles is greater than a certain number N. This number depends on the problem (object's trajectory; the dynamics of the object;...).

The previous related study, in the field of object tracking [5,20,21], used 64, 50, and 100 particles, respectively. In our previous study [22,23], we use 100 particles.

For hundreds particles, the ratio between the latency cycles and the number of particles affects the total execution time. So, it is important to consider all of the main four steps: the particle sampling, the weighting, resampling as well as the output step to efficiently build a hardware implementation of SIRF. An efficient implementation should efficiently implement all these steps from the perspective of execution time, hardware resources, and robust performance.

2.3. Architectures

This section is devoted to a full and detailed presentation of our first two SIRF architectures. The main goal of our implementation is to minimize the execution time and the used resources without affecting the performance.

2.3.1. The proposed two-step architecture with sequential resampling

The proposed architecture is shown in Figure 2[15]. This architecture is composed of a FIFO to store particles, a sample engine, a weight calculation, and accumulation engine, a resampling engine, and an output calculation engine. The FIFO is a $2M \times p$ dual-port FIFO to store the M particles. This FIFO has $2M$ locations to allow reading the current frame M particles while at the same time writing the replicated particles of the next frame to this same FIFO. The width of the FIFO word, p, is a five-component state vector, given by (x, y, v_x, v_y, I) to represent the X-position, Y-position,

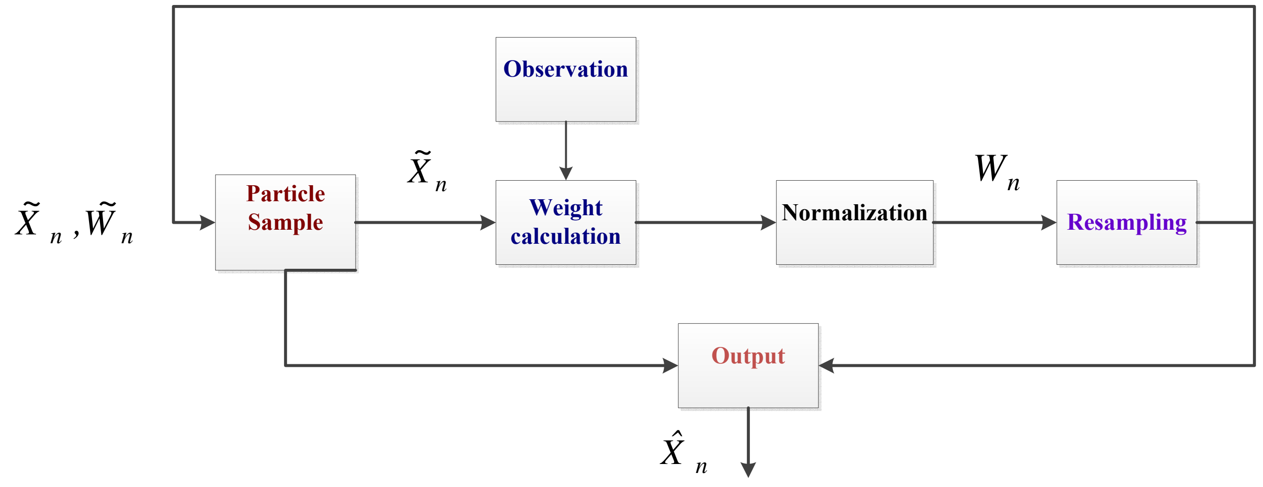


Figure 1 The block diagram of the SIRF [7].

the velocity in the X and Y directions and the gray level intensity, respectively. The resampling engine has two register files in addition to the resample logic. These two registers are the replication factor memory (RFM) which consists of $M \times n$ register file for storing the replication factor for each particle (where $M = 2^n$), and the weights memory (WM) which consists of $M \times w$ register file for storing the particle weights where w represents the length of weight bits. M is chosen to be 64. This value is suitable for a search area of 32*32 pixels for our object-tracking example.

In the following, we describe the function of the different sub blocks of Figure 2. The registers: *Particle R* and *NR* store the current particle vector and its replication factor, respectively. The contents of these registers are loaded into the *Particle S* and *NS* registers, respectively, if *NR* is non-zero. The WM register file contains the weights of the particles and the RFM register file contains the corresponding replication factors for the particles.

The operations of the PF are carried out as follow:

1. The particles are sampled (generated) and stored in the FIFO at the *Sample engine* using the *Random generator* [24] with the known initial state vector of the object. As the particle is sampled, it can be used for weight calculations.
2. During the process of weight calculation, the accumulation of the total sum of weights is carried out in the weight calculation engine.
3. As the sampling of the total number of particles finishes, the resampling engine starts the resampling process. The replication factor for each particle is calculated according to its weight as described in Section 2.3.1.2.
4. When the replication factors of the particles are calculated, the output engine starts the output calculation process as follows:

   a. Read one particle from the FIFO to *Particle R* register and the corresponding replication factor to *NR* register; continue reading until a non-zero value of *NR* is found. At that time, the particle is transferred to *Particle S* register and the content

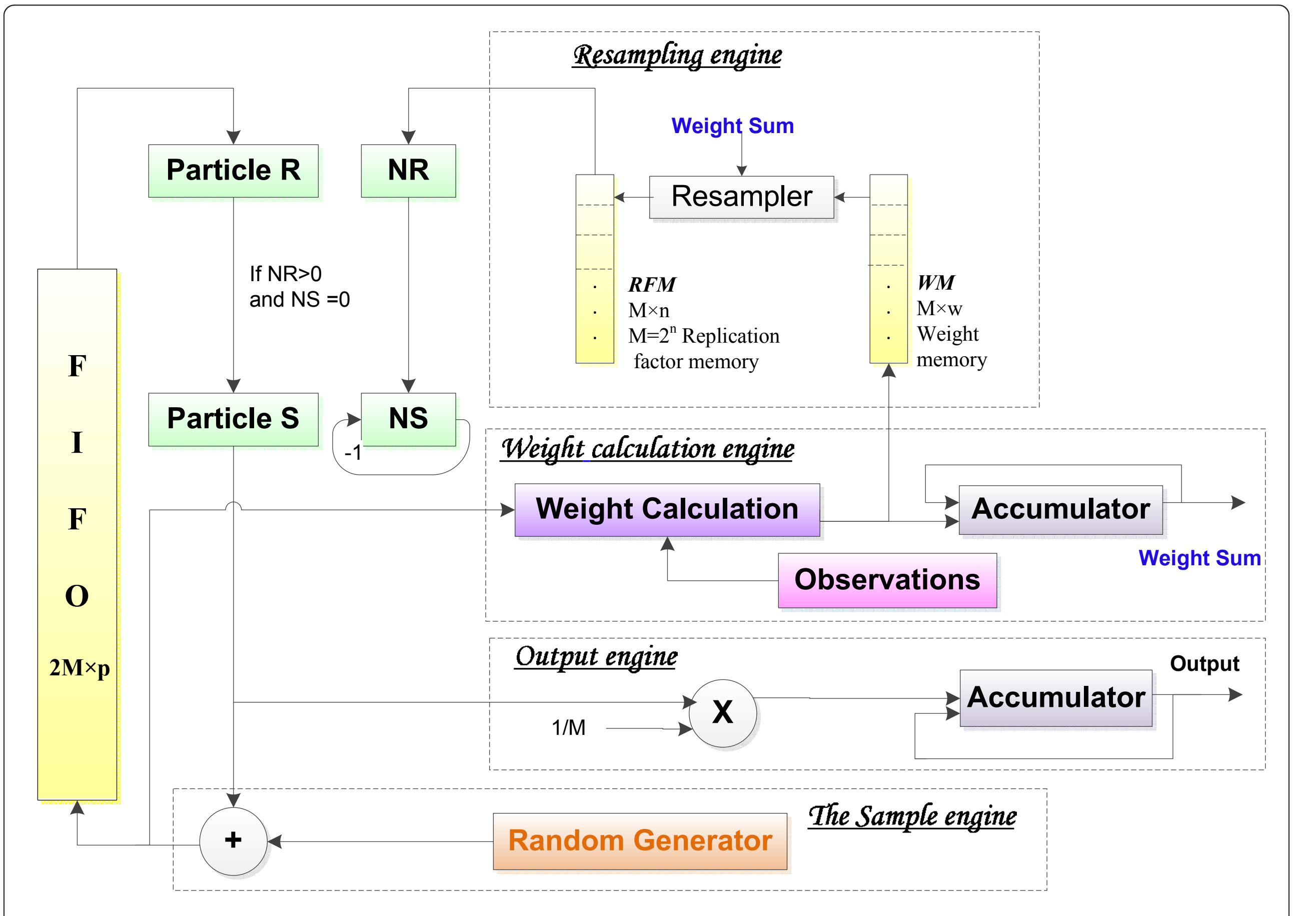


Figure 2 The two-step architecture with sequential resampling.

of register *NR* is transferred to register *NS*. *NS* is decremented at each clock cycle.

b. The contents of register *Particle S* are sampled by adding random values from the *Random Generator* and the resulting particle is written to the FIFO. At the same time, the generated particle is used to the weight calculation unit. In addition, the contents of register *Particle S* are used in the output calculation by multiplying the particle x and y values by $1/M$ and accumulated to get the mean output values (the X and Y positions of the tracked object).

c. During decrementing the contents of register *NS* to zero, the FIFO reading does not stop until a non-zero value of *NR* register is found. Figure 3 shows the state diagram of the read and write operations from and to the FIFO.

5. As the M particles are read and repeated according to their replication factors, the resampling engine calculates sequentially the replication factor for each particle for the next instant. The resampling engine calculates one replication factor every clock cycle.

6. Finally, again, the FIFO contains M particles and so the RFM contains M corresponding replication

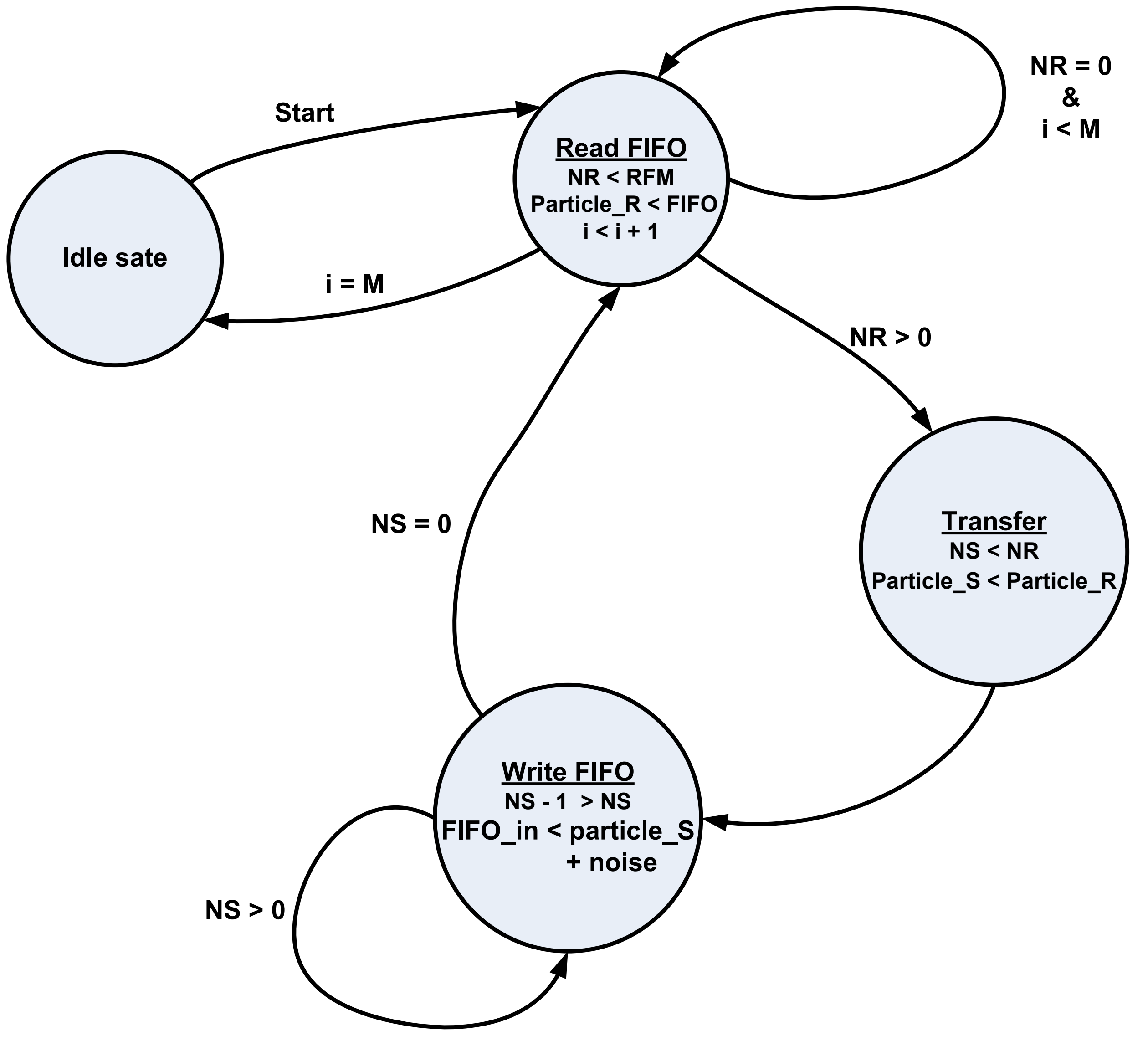


Figure 3 The read/write FIFO operations. Where FIFO_out represents the valid reading data from the FIFO and NR represents the corresponding replication factor read from the RFM. FIFO_in represents the data written to the FIFO.

factors. For each coming observation, repeat the steps 4 and 5.

Thus, based on the data independencies between the particles, our architecture carries out the three steps of the particle generation, the weight, and output calculations concurrently. The resampling step cannot be overlapped in time with the weight calculation step due to the necessity of knowing the overall sum of the weights. So, we select the modified RSR algorithm [25]. In the following, we describe the main function blocks of our proposed structure.

2.3.1.1. The weight calculation engine To obtain robust performance, we calculate the weight by using the multi likelihood functions proposed in [22]. For hardware simplification, we implement only the position and intensity likelihoods *LP* and the intensity likelihood *LI*, while the similarity likelihood is dropped. LP and LI are defined by:

$$LP \quad = \exp(-\frac{((i-x)^2 + (j-y)^2)}{2\sigma_p^2}) \tag{6}$$

$$LI \quad = \exp(-\frac{(I_t^{(i,j)} - z_t^{(i,j)})^2}{2\sigma_I{}^2}) \tag{7}$$

where (i, j) is the particle position and (x, y) is the previous object position. σ_P is the variance of the position. $I_t^{(i,j)}$ is the mean gray intensity level estimated at particle position (i, j) at time t, and $Z_t^{(i,j)}$ is the measured mean gray level intensity value at position (i, j) at time t. σ_I is the variance of the gray level intensity. The overall likelihood is taken as $L = LP * LI$.

Taking into consideration the complexity of the hardware implementation of the exponential function, we can simplify the *LP* and *LI* likelihood calculations by piecewise linear approximations. Figure 4a, b shows the Gaussian likelihood functions *LI* and *LP* and the corresponding linearized functions, respectively. The piecewise linear function for the position likelihood is chosen to have sharp vertex for more accurate tracking. On the other hand, the piecewise linear function for the intensity likelihood is chosen of a trapezoidal form to tolerate small intensity variations. To test whether this approximation affects the performance of the tracker significantly or not, we carried several MatLab experiments. Figure 4c, d documents one such MatLab experiment. Here, we consider the frame number 89 within the tracking sequence A of reference [26]. Starting with the same particle set, we calculate the normalized weight of these particles according to either the exponential functions or the linearized functions. Hence, we calculate the replication factors corresponding to each case. Figure 4c, d indicates that the tracker suffers only minor differences if the exponential weight functions are replaced with the numerically favorable linearized functions. All the variables are represented in the fixed point representation.

The weighting engine stores the index of the maximum weight values to be used in the resampling engine. In addition, the weighting engine accumulates the weight sum for use in the resampling unit.

2.3.1.2 The resampling engine The *Systematic Resampling* (SR) algorithm has two loops: one *for loop* of *M* iterations and a *while loop* with non-deterministic number of iterations, since this *while loop* depends on the normalized weight of each particle. We used the RSR algorithm. The flowchart is shown in Figure 5. The RSR algorithm is two times faster than the SR algorithm. The modified RSR algorithm [25] avoids the *M* divisions in hardware, and instead uses one division (*M* by the total sum of weights). The weight value of a particle is normalized, i.e., the range of weight of a particle is from 0 to 1. Therefore, the sum of weights for 64 particles ranges from 0 to 64. We use a LUT) of 64 entries to handle the inverse for the range of the sum of weights from 1 to 64 values.

Figure 6 shows a simplified structure of the resampling unit. First, the LUT is used to calculate the inverse value of the total weight W, then $D = M/W$ value is calculated using one multiplier. The replication factor $r^m = w^m * D$, $m = 1,..., M$ values are calculated and stored in the RFM.

For hardware simplification, we simply use $\Delta U(i) = 0$, $i = 1,...,M$ to eliminate the use of random generator. For the RR method [25], the sum of the replication factors of all the particles ($N = \sum_{m=1}^{M} r^m$) is less than M, except for special cases. The remaining particles are obtained using other mechanisms. We added the difference $(M - N)$ number to the replication factor of the particle having the largest weight. The index of the highest weight particle is stored during the calculation of the weight values in the weight calculation engine.

2.3.1.3 The execution time The timing diagram of the implemented SIRF is shown in Figure 7. The TL_g is the particle generation latency and equals two clock cycles. Apart from the first PF iteration, the total required time, in cycles, is $T_{SIRF} = (aM + TL_W + TL_{\text{res}})$, where TL_W is the start up latency of the weight calculation unit. TL_W equals one clock cycle for our simplified piecewise linear functions. Therefore, the total latency cycles are three. TL_{res} is the resampling time and equals $M + 1$ clock periods (one clock to add the $M - N$ to the highest weight replication factor). The value of a lies between 1

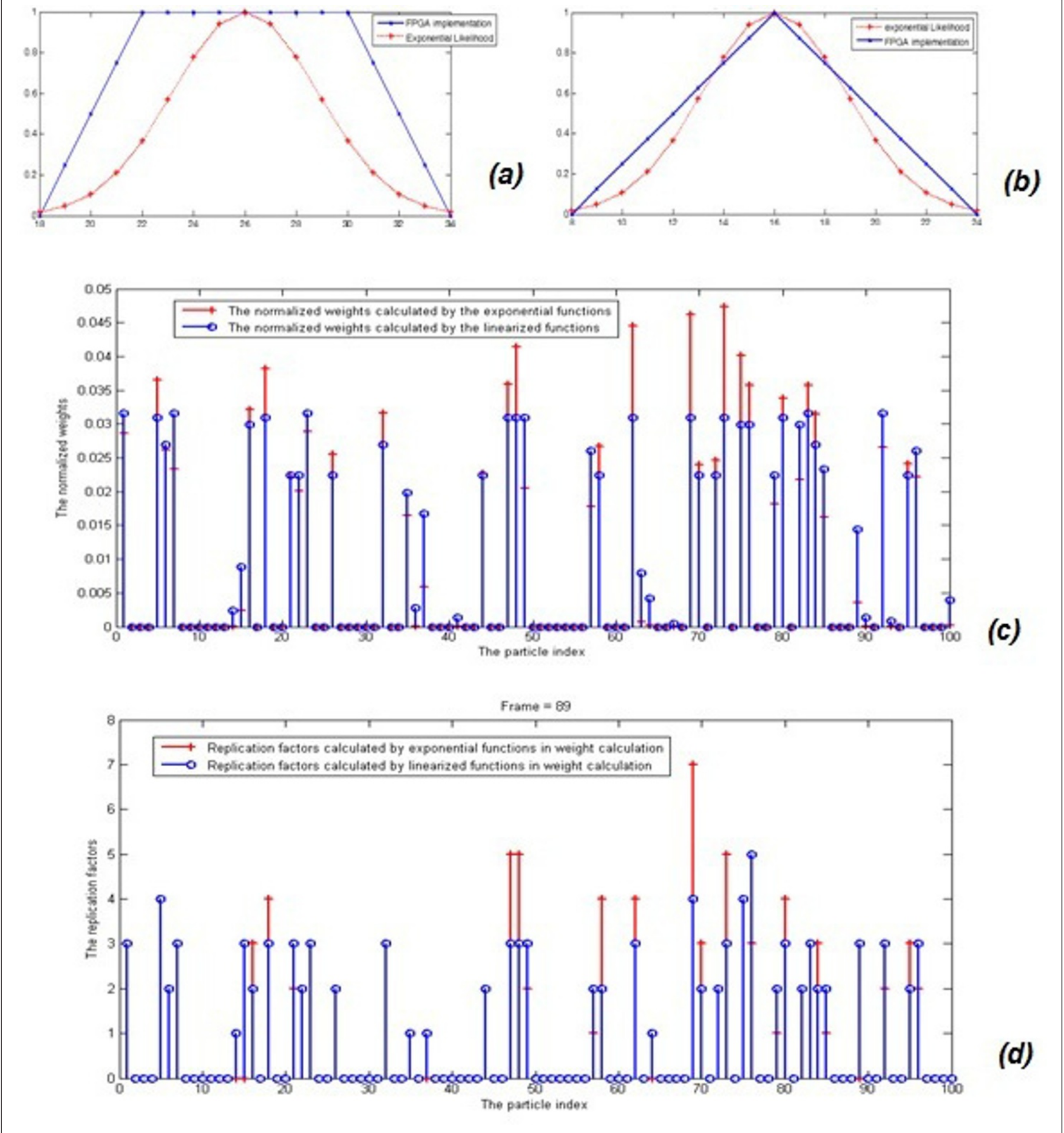


Figure 4 Comparison between the evaluated weights using the simplified piecewise linear functions and the exponential functions. **(a)** LI likelihood function. **(b)** LP likelihood function. **(c)** Normalized weights calculated by the exponential and the linearized likelihood functions. **(d)** Comparison between the replication factors calculated by the exponential and the linearized likelihood functions.

and 2, and depends on the replication factors of the particles. The worst case takes place if M - 1 particles have zero replication factors and only the last stored particle in the FIFO has a replication factor = M. In this case, the FIFO needs to read M - 1 particles first, until the FIFO reaches the particle number M. Then, the sampling, weighting, and the output calculations are started and repeated for this particle M times. For all other cases (when some of the particles have non-zero replication factors) the value of a is close to one.

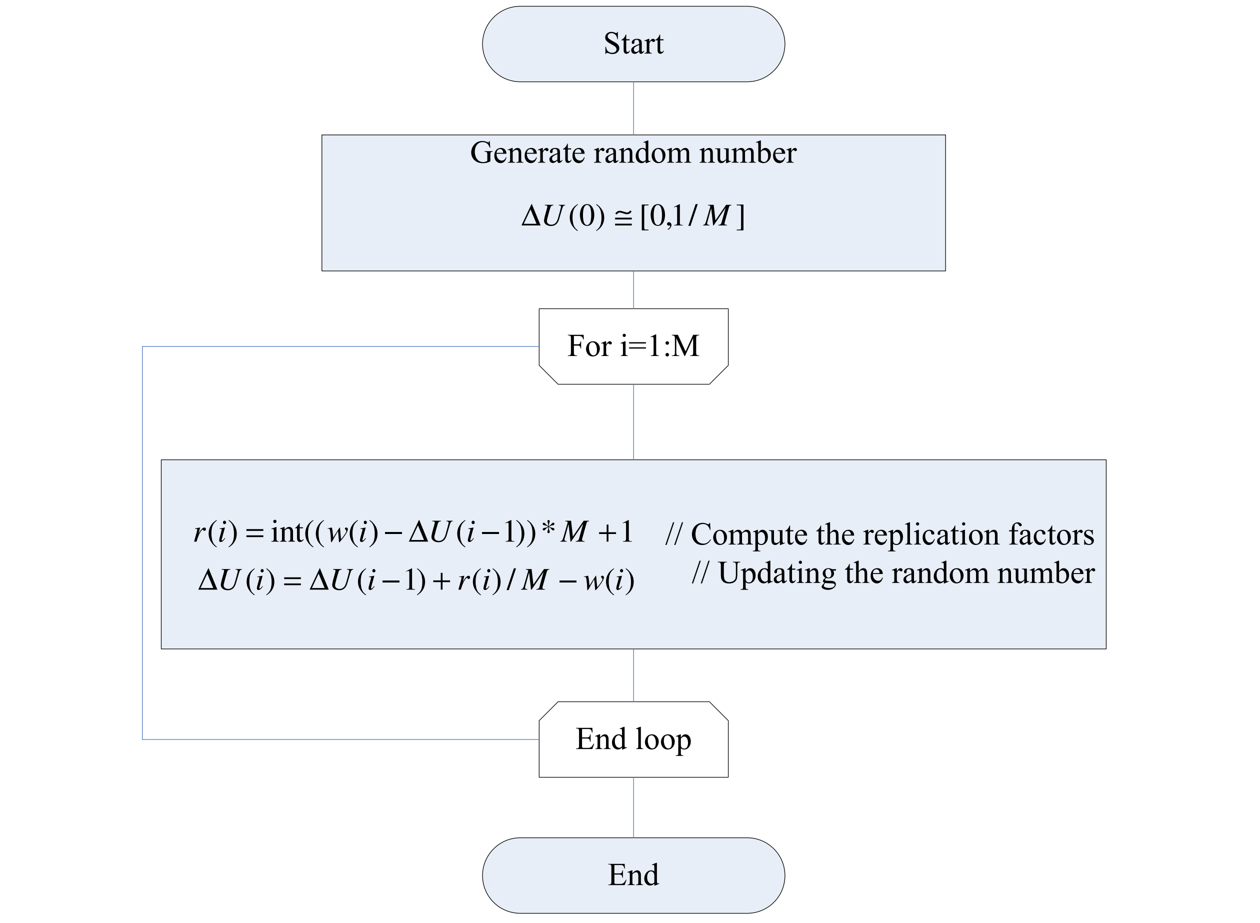

Start
Generate random number
$\Delta U(0) \cong [0,1/M]$
For i=1:M
$r(i) = \text{int}((w(i) - \Delta U(i-1)) * M + 1$ // Compute the replication factors
$\Delta U(i) = \Delta U(i-1) + r(i)/M - w(i)$ // Updating the random number
End loop
End

Figure 5 Flowchart of RSR algorithm.

2.3.1.4 Main features of the two-step architecture with sequential resampling The two-step architecture with sequential resampling, presented in Section 2.2.1 and Figure 2, has the following advantageous features relative to published architectures [2,6,7,9,10]:

i. FIFOs are used instead of memories whenever possible. This saves address decoding logic and enhances speed of operation.
ii. Linearized likelihood functions to speed up the weight calculations while preserving the localization features of the Gaussian likelihood functions

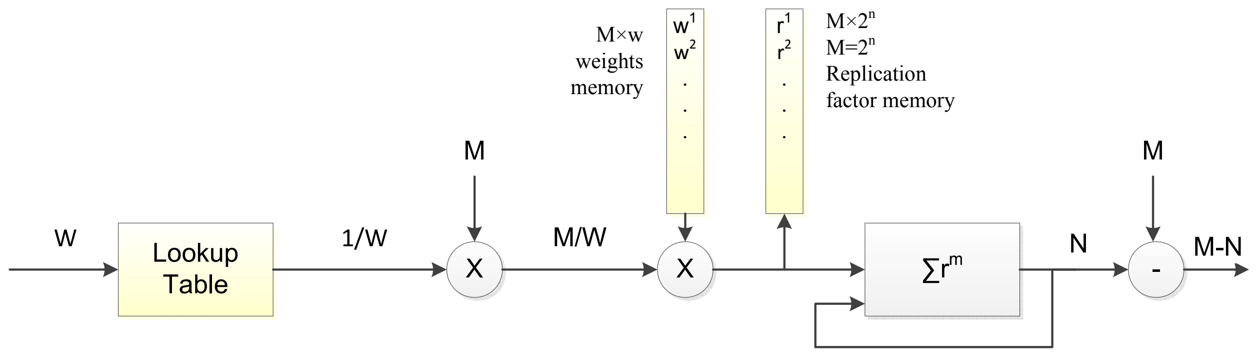


Figure 6 The resampling unit.

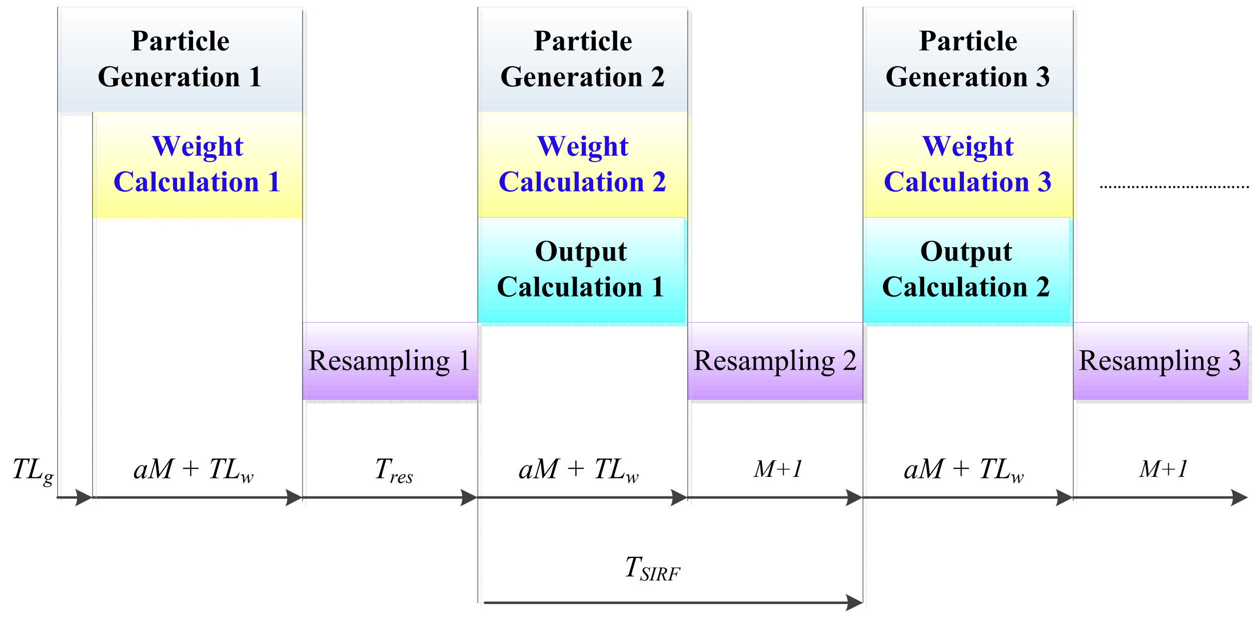


Figure 7 The timing of the two-step architecture with serial resampling.

2.3.2. The two-step architecture with parallel resampling
The two-step architecture with parallel resampling was proposed in [15]. Figure 8 shows the block diagram of this architecture, which is the same as the first architecture except that the resampling engine is a parallel engine.

2.3.2.1. The parallel resampling engine The RSR algorithm can also be modified for parallel execution purposes by splitting the resampling process into multiple concurrent loops. Each loop does the usual RR algorithm for M/L particles, where L is the number of loops (cf. Table 1). This mechanism reduces the execution time of the resampling to M/L cycles at the cost of adding more hardware. To speed up the resampling step, we selected $L = M$. This is feasible since the number of particles in our example is around 100 particles and our platform is FPGA. Since $M = L$, there is one particle for each resampling unit, and resampling takes place in just one clock cycle. The implemented RSR resampling eliminates the use of memory for storing the normalized weights.

Table 2 shows the resource utilization for hardware PF resampling in the case of single, 8, and 64 parallel resampling loops.

2.3.2.2. The execution time The timing diagram of the two-step architecture with parallel resampling SIRF is shown in Figure 9. The TL_g, TL_W, and a have the same values as for the two-step architecture with sequential resampling SIRF. T_{SIRF} (in cycles) = $(aM + TL_W + TL_{\text{res}})$, and TL_{res} is the resampling time. For $L = M$, TL_{res} equals two cycles (one for calculation + one for modification of M - N).

3. The distributed implementation of PF
In the following sections, we propose a distributed architecture for efficient implementation of the PFs. The main design goal of this architecture is to minimize the execution time. This goal is achieved by using multiple PEs with a CU. First, we review the published study on distributed PFs. We, also, review the different networks to connect the PEs. Next, we introduce our proposed distributed SIRF.

3.1. Algorithms and architectures for distributed PFs
In this section, we discuss the different resampling algorithms for distributed PFs.

3.1.1. Centralized resampling [27]
Centralized resampling is a straightforward approach to implement the SIRFs based on the architecture presented in Figure 1. The particles are sampled and the weights computed in parallel by the PEs. The CU carries out the resampling and particle routing as well as the overall control. Figure 10a shows the sequence of operations and the directions of communication. The CU is responsible for full resampling, which is performed sequentially. The communication requirements of this implementation are immense. The CU collects M weights in order to perform the resampling, and returns M indexes and replication factors to the PEs, with the assumption that particle allocation with arranged

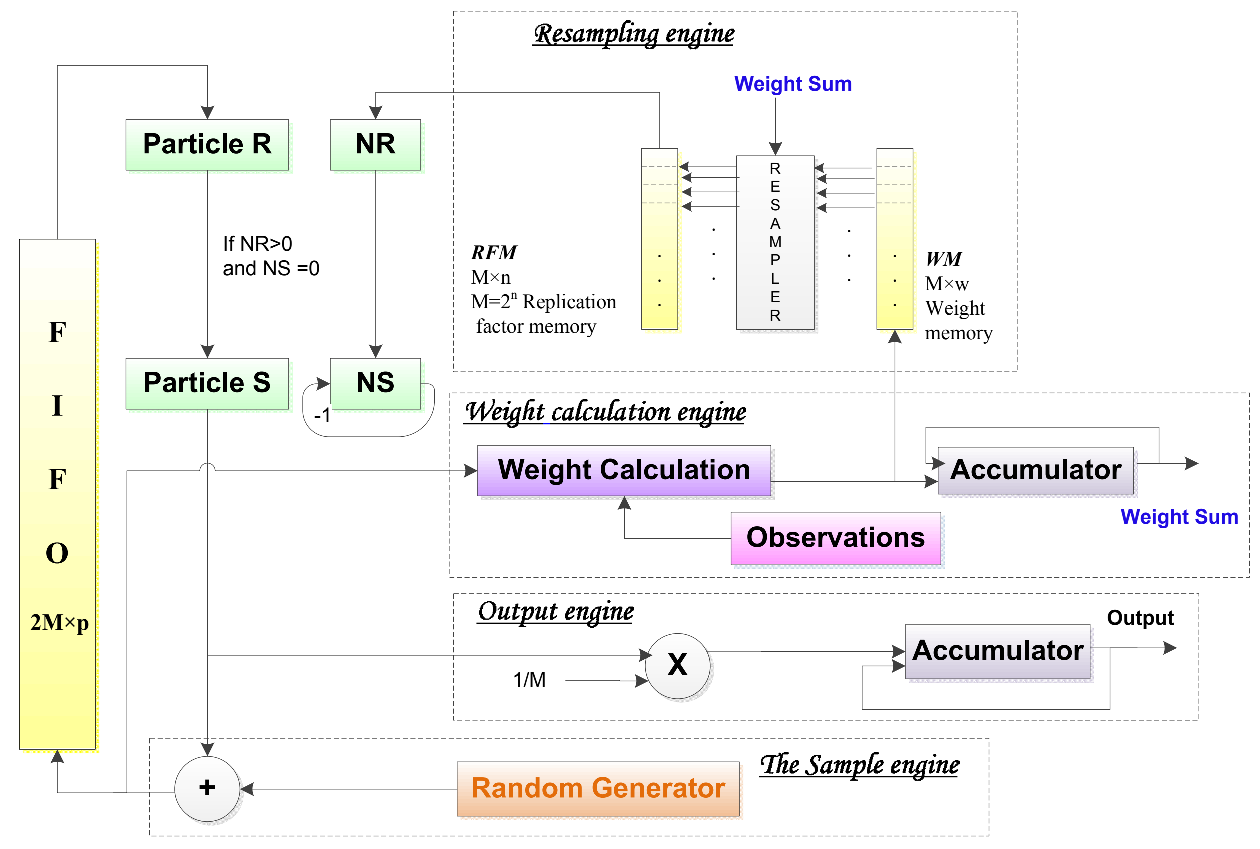


Figure 8 The two-step architecture with parallel resampling.

indexes is applied. While the communication of the weights and indexes is deterministic, the particles are routed in a non-deterministic fashion.

3.1.2. Distributed RPA [27]

Let the number of particles be M and number of PEs is K, so each PE is assigned $N = M/K$ particles. After the weight calculation step, each PE calculates the sum of the weights of its particles, i.e., $\mathrm{W}^k = \sum_{i=1}^{N} w^{i,k}, k = 1, 2, ..., K$. Each PE, then, sends only this sum to the CU. Next, the CU treats the individual PEs as particles, and carries out a resampling step between these K PEs according to their total weight sums. This resampling step is termed "Inter-resampling". The CU sends to each PE its share of the replicated particles. Each PE would subsequently carry out "intra-resampling" amongst the particles share assigned to it by the CU. The sequence of operations performed by the PE and CU are shown in Figure 10b. Obviously, particles should be redistributed among PEs after each cycle of inter- and intra-resampling. Thus, each cycle of inter- and intra-resampling is followed by a PEs' communication phase to rebalance the particles distribution among PEs. To speed up this process, PEs are divided into groups, and particles are locally redistributed–in parallel–within each group. PEs are next regrouped via several alternative schemes until the goal of equal particle redistribution among PEs is achieved.

The same resampling results are obtained via RPA or via sequential resampling. RPA is obviously superior to centralized resampling due to the time savings in the parallelized intra-resampling step as well as the reduced PE-CU communications. Also, CU design is simpler.

Table 1 Pseudo-code of splitting the RSR algorithm into *L* loops: *Generate a random number ΔU^1 ~ u*[0, 1]

```
D = M/W // where W is the total sum of weights.
– For l = 1 to L
    r(l) = int((W(l) -ΔU(l))*D);
    ΔU(l+1) = ΔU(1) + r(l)- W(l) * D;
    Loop l
        - For m = (l-1)*M/L to l*M/L-1
            Perform RSR
        - End For
– End For
```

Table 2 Comparison between the resources for different cases of the two-step sequential architecture with parallel resamplin (Xilinx FPGA xc5vlx50t-3-ff1136)

Resource	Sequential $L = 1, M = 8$	Parallel RSR $L = M = 8$	Parallel RSR $L = M = 64$
Slice register	11	11	131
Slice LUTs	77	78	1692
Number of slices used as logic	77	78	1692
Number of DSP48Es	1	8	48

The time for the resampling procedure in the distributed RPA is reduced $\frac{M}{M/K + K}$ times, where M/K corresponds to the intra-resampling time and K is a time for inter-resampling.

3.1.3. Distributed resampling with RNA [27]

The problems of the particle routing and the delay introduced by the global pre-processing step (inter-resampling) can be solved by using the RNA algorithm. In the RNA algorithm, the number of particles within a group (one or more PEs) is fixed and equals to the number of particles per group ($N^k = N$). So, full independent sampling is performed by each group. The inter-resampling step is eliminated completely. Again, intra-resampling leads to imbalance in particle distribution among the PEs groups. Thus, group communications is again needed post intra-sampling. Bolic et al. [27] proposed several schemes for speeding up this communications, which they termed "regrouping". The main advantage of RNA is the routing of particles which is deterministic and is planned by a designer. The other characteristic of RNA is that the weights after resampling are not equal to $1/M$; instead, they are equal inside the groups.

3.2. The proposed distributed architecture for SIRF

We propose a variant of the distributed RPA architecture with K PEs and CU as shown in Figure 11. Our proposed distributed RPA architecture implements a simple ring network for particle routing. Yet, it enhances the performance of this simple communication scheme by implementing a by-pass mechanism that allows particles to be routed quickly across several nodes (PEs). Each PE in our architecture is the same as to the proposed two-step architecture with sequential resampling described in Section 2.3.1. The CU function is to collect the partial sums of the weights from the PEs and to calculate the outputs, as well as to perform the inter-resampling.

The particles that are replicated as a result of the resampling for PE^k are stored into the local FIFO^k for N ($N = M/K$) particles. When there is a surplus of particles, $N^k > N$, these particles are routed to the neighboring processor PE^{k+1} through the local interconnections.

If there is a shortage of particles in PE^k, $N^k < N$, then PE^k reads particles from the neighboring processor PE^{k-1} through the local interconnections.

The sequence of the SIRF operations is carried out in the PEs and the CU as follows:

1- Each PE^k performs the sample step, the importance step to N particles ($N = 8$ in our design

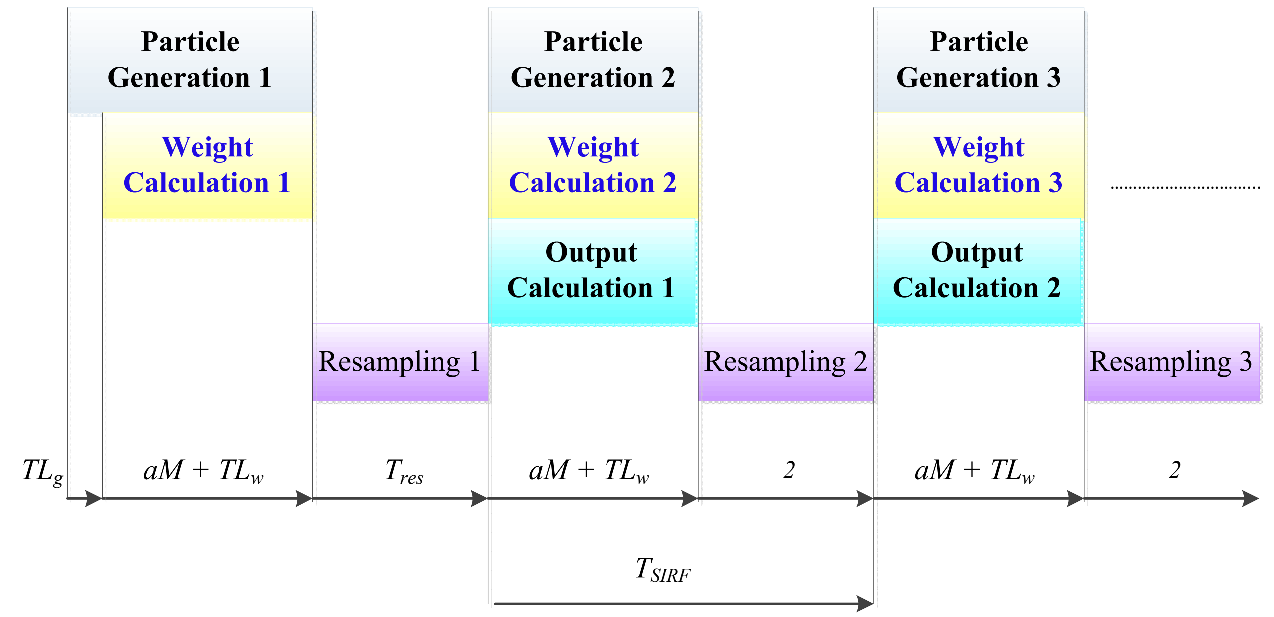


Figure 9 Timing diagram for the two-step architecture with parallel resampling.

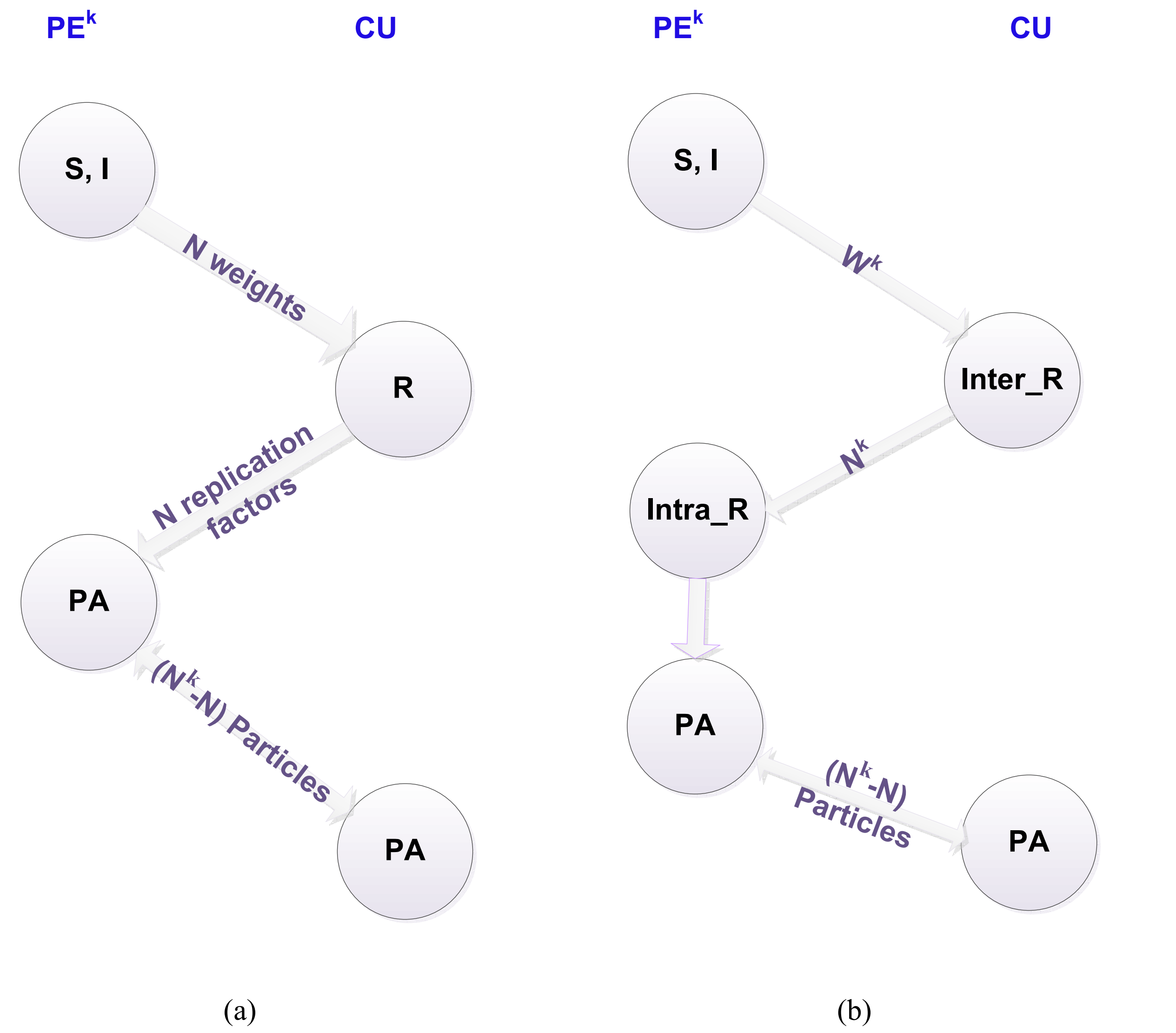


Figure 10 Sequence of operations performed by PE^k and CU for (a) centralized resampling and (b) RPA. S, sample; I, importance, R, resampling; PA, particle allocation [23].

example) and accumulates PE total weight $W^k = \sum_{i=1}^{N} w^{i,k}, k = 1, 2, ..., K$.

2- The CU receives the partial sums of the processors weights from the PEs, performs the inter-resampling, and sends back the replication number of particles N^k *to* PE^k for $k = 1,2,..., K$.

3- The CU also, calculates the inverse of W^k $(1/W^k)$ for $k = 1, 2,..., K$ and sends them to PE^k for $k = 1,2,..., K$, respectively, to be used in the intra-resampling in each PE.

4- The PEs perform the intra-resampling in parallel. The particles are allocated to the local corresponding FIFO for N particles.

5- All the PEs with $N^k > N$ will send the surplus $(N^K - N)$ particles to the local network in parallel.

6- All the PEs with $N^k < N$ will take the remainder $(N - N^k)$ particles from the local network in parallel.

7- The CU calculates the output X-mean, Y-mean position of the object as well as I-mean gray level intensity of the object as cumulative sums.

3.2.1. The resampling step

We use the modified RSR algorithm described in Section 2.3.1.2 in both the inter-resampling in the CU and intra-resampling in the PEs.

3.2.1.1. The inter-resampling The CU performs the partial resampling using the partial weights $W^k = \sum_{i=1}^{N} w^{i,k}, k = 1, 2, ..., K$ to produce N_k, $k = 1, 2, ...,$

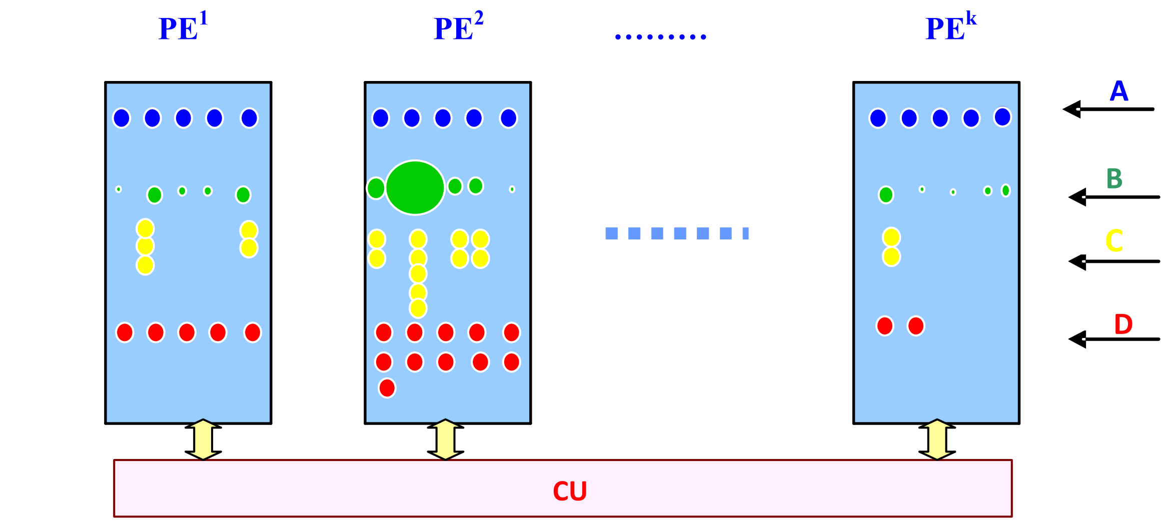


Figure 11 The distributed SIRF architecture, where the A group particles represent the uniformly distributed particles, the B group particles represent the weighted particles, the C group particles represent the uniformly replicated particles, and finally, the D group particles represent the uniformly sampled particles.

K using the modified RSR algorithm [25]. Since the resampling produces replication factors equal or less than M ($M \geq \sum_{i=1}^{K} N^k$), we opted to add the remaining particles to the PE with the largest weight. The index of this PE is stored during the calculation of the replication factors.

Using the LUTs, we calculate the inverse of the total weight WT and the individual weights W^k of each PE. For our example with $M = 64$ and $K = 8$, simple reflection would reveal that the same LUT can be used to calculate both WT and W^k, provided that a simple selectable shift operation is added at the output of this LUT.

3.2.1.2. The Intra-Resampling We implement the RSR resampling described in Section 2.3.1.2. After the CU calculates the number of particles that each PE replicates (N^k, $k = 1,....,K$), the intra-resampling is performed inside each PE. Each PE calculates one replication factor for each particle of its M/K particles per clock cycle, leading to K replication factors calculated by the K PEs per clock cycle.

3.2.2. The local interconnect network

It is important to notice that N^k is a random number, which depends on the overall distribution of the weights. The PEs with $N^k > N$ have surplus of particles and they need to exchange particles with the other PEs having a shortage of particles ($N^k < N$). The N^k numbers change after each sampling period, so that it is necessary to connect different PEs in order to perform particle routing. In summary, the communication pattern is non-deterministic and the connections among the PEs are changed after each sampling period.

Depending on the value of N^k, each PE^k has one of the three possible states shown in Figure 12.

1- State "00" where $N^k = N$. So, any incoming particle from PE^{k-1} is sent to PE^{k+1}.
2- State "01" where $N^k > N$. The processor is a source element. As soon as PE^k replicates the first N particles, it sends the reminder N^k - N to the neighbor PE^{k+1}. After PE^k completes the replication of N^k particles and any stored particles in the local net memory, the state transfers to state "00". If during replication of the particles, a particle from PE^{k-1} is coming, then PE^k stores this value in local net memory until it completes the replication of all its internal generated particles.
3- State "10" where $N^k < N$. The PE^k is a sink element. First, it replicates N^k and reads any incoming particles through the network to complete its FIFO to N particles. As soon as the reminder N - N^k particles are read from the network, the PE^k state transfers to state"00".

The particles sent to the network from source PE^k ($N^k > N$) are sent from the register *Particle S* (i.e., before performing the sample in the source processor). Also, the particles received at the sink processor PE^j ($N^j < N$, *where* $k \neq j$) are received in register *Particle S* (i.e., just before sampling in the sink processor). Figure 13 shows the local communication between two consecutive PEs.

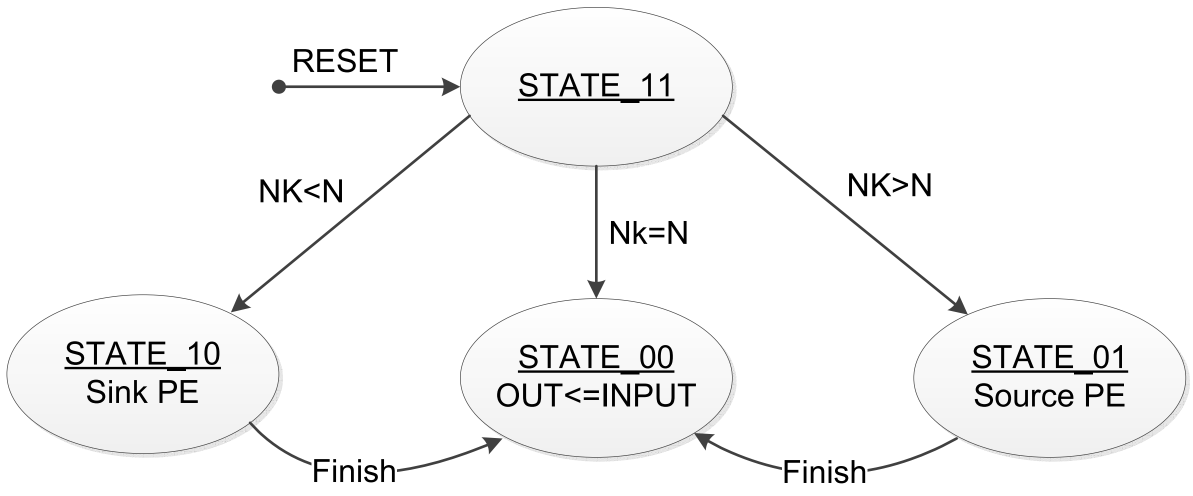


Figure 12 The possible states of the processor element during routing.

Note that the use of the local net memories inside PEs together with the control FSM of Figure 12 allows fast exchange between non-neighboring PEs. If, for example, $N^5 = N + 1$, $N^6 = N^7 = N$, and $N^8 = N - 1$, then the local net memories allow PE^5 to send the surplus particle to PE^8 in 2 cycles. Also, the routing step and the next sampling step are overlapped allowing pipelining of their operations. Consequently, the particle routing time adds only partially to the total execution time of the PF.

Figure 14 shows a ModelSim simulation run illustrating the operation of the local interconnect network. In the shown simulation, the PE^2, PE^4, PE^5, PE^6, and PE^8 are sink elements, while processors PE^1, PE^3, and PE^7 are source elements. As soon as the sources PE^k, k = 1, 3 and 7 complete writing N particles to their FIFOs, they start sending the reminder N^k-N, k = 1, 3, and 7 to the neighboring PE^{k+1} at the same time. The sinks PE^k, k = 2, 4, 5, 6, and 8 accept the incoming particles from the

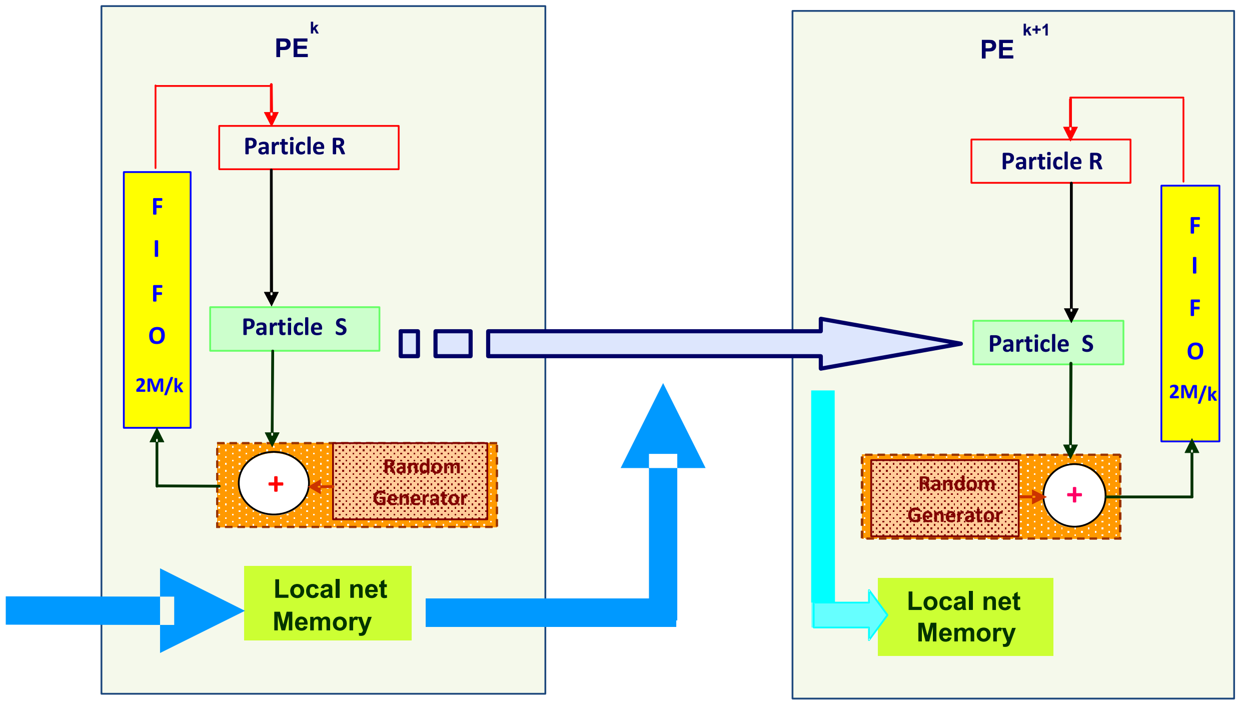
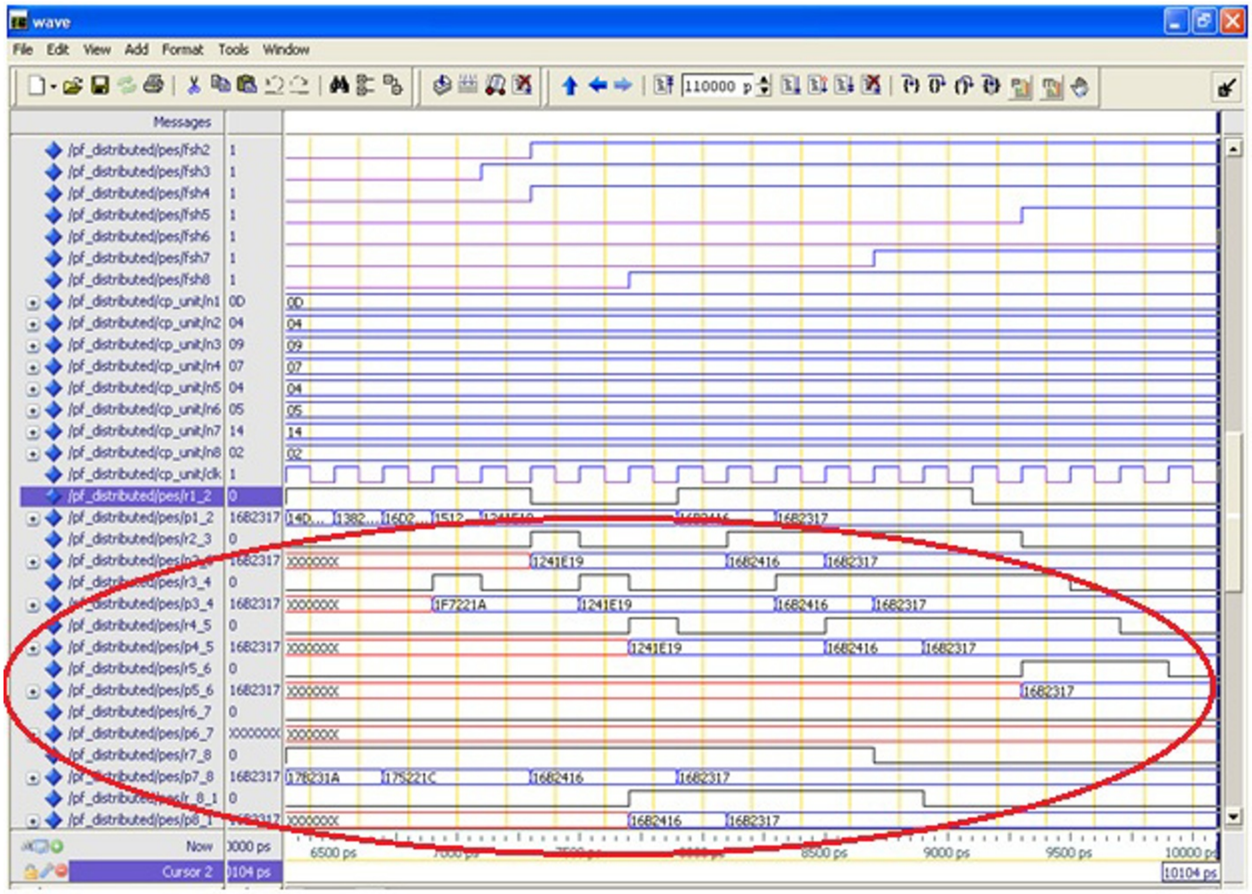


Figure 13 The local communication between two consecutive processor elements.



```
/pf_distributed/pes/fsh2      1
/pf_distributed/pes/fsh3      1
/pf_distributed/pes/fsh4      1
/pf_distributed/pes/fsh5      1
/pf_distributed/pes/fsh6      1
/pf_distributed/pes/fsh7      1
/pf_distributed/pes/fsh8      1
/pf_distributed/cp_unit/n1    0D
/pf_distributed/cp_unit/n2    04
/pf_distributed/cp_unit/n3    09
/pf_distributed/cp_unit/n4    07
/pf_distributed/cp_unit/n5    04
/pf_distributed/cp_unit/n6    05
/pf_distributed/cp_unit/n7    14
/pf_distributed/cp_unit/n8    02
/pf_distributed/cp_unit/clk   1
/pf_distributed/pes/r1_2      0
/pf_distributed/pes/p1_2      16B2317
/pf_distributed/pes/r2_3      0
/pf_distributed/pes/p2_3      16B2317
/pf_distributed/pes/r3_4      0
/pf_distributed/pes/p3_4      16B2317
/pf_distributed/pes/r4_5      0
/pf_distributed/pes/p4_5      16B2317
/pf_distributed/pes/r5_6      0
/pf_distributed/pes/p5_6      16B2317
/pf_distributed/pes/r6_7      0
/pf_distributed/pes/p6_7      XXXXXXX
/pf_distributed/pes/r7_8      0
/pf_distributed/pes/p7_8      16B2317
/pf_distributed/pes/r_8_1     0
/pf_distributed/pes/p8_1      16B2317
Now                           10000 ps
Cursor 2                      10104 ps
```

Figure 14 The parallel routing of the particles in the local interconnection network. PE[2], PE[4], PE[5], PE[6], and PE[8] are sinks elements, while processors PE[1], PE[3], PE[7] are sources elements. Where the waveform pk_k+1 represents the particles transferred from PE^k to PE^{k+1}. The waveforms *nk* and *fshk* represents the N^k and finish indicator written signal for *N* particles in the local $FIFO^k$, respectively (*N = 8* in our case).

network to complete their FIFOs to *N* particles, and send the incoming particles to the neighboring PE^{k+1}.

The local net memory size is designed to handle the worst case, in which two sequential processor elements PE^k, PE^{k+1} have half the replicated particles, i.e., $N^k = N^{k+1} = M/2$.

So, the local net memory should be enough to store $M/2 - N$ ($= M(K - 2)/2K$). In our case, the local net memory size has 24 particle locations. If the number of PEs *K* is less than two, then the local net memory is zero. This is logical since the net memory is used inside each PEs to store the particles passing through the PEs. If we have one PE or two PEs, then there is no passing-through particles, and hence, there is no need to the net memory.

The distributed RPA PF is captured using VHDL and verified using ModelSim simulator.

3.2.3. The execution time of the distributed SIRF

The timing diagram of the implemented distributed SIRF is shown in Figure 15. TL_W *and* TL_{res} have the same meaning and values in the first implemented SIRF described in Section 2.2.1. After the first instant, the total cycle time required is T_{SIRF} (in cycles) = ($aM/K + TR + TL_W + TL_{\text{res}}$), where *TR* is the routing time and equals $b(k - 1)M/k$ *clock cycles. b* assumes values between 0 and 1. *b* = 0 if no routing is needed; i.e., $N^k = N$, $k = 1, 2, ..., K$. $b = 1$ if only one processor element PE^j has $N^j = M$, *and* $N^k = 0$ for *all other k's*. That PE^j has only one particle with replication factor $r^j = M$ at location *N* of its local $FIFO^j$. The TL_{res} is the resampling time and equals the CU resampling time $K + 1$, in addition to the PEs resampling time $M/K + 1$. Totally, TL_{res} equals $K + M/K + 2$. So, $T_{\text{SIRF}} = (aM/K + K + M/K + 2 + b(K - 1)M/K + TL)$ clock cycles, assuming the same latencies *TL* for the single and distributed SIRF.

4. Comparison of the execution time between the three different implementations for SIRF

In summary, the following are the total execution times of SIRF in our three different implementations:

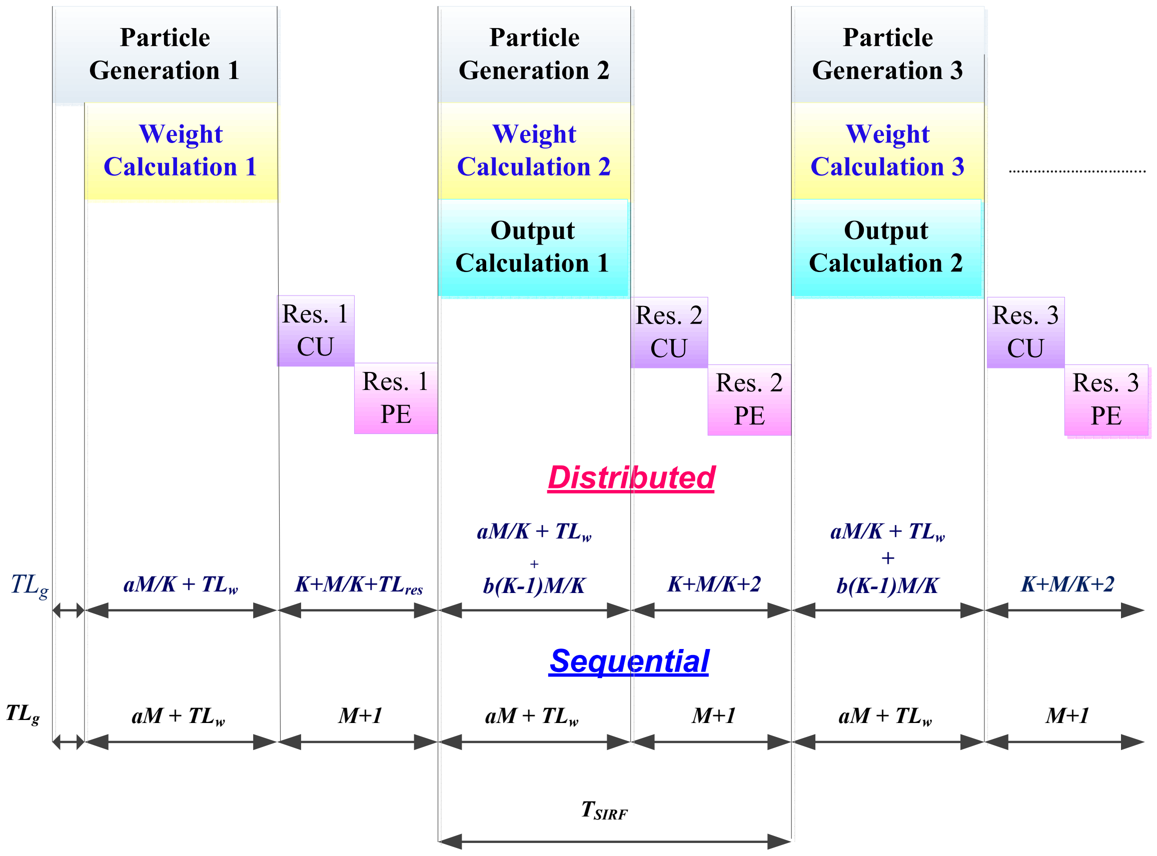


Figure 15 The timing diagram for the distributed SIRF with comparison versus the two-step architecture with sequential resampling.

- The total time of the two-step architecture with sequential resampling:
($aM + M + TL$) clock cycles
- The total time of the two-step architecture with parallel resampling:
($aM + 2 + TL$) clock cycles
- The total time of the distributed PF: ($aM/K + K + M/K + 2 + b(K\text{-}1)M/K + TL$) clock cycles

Table 3 compares the best case and the worst case timing for the three implementations. $TL = 3$ clock cycles. Consider, for example, the two-step PF with parallel resampling. Table 4 indicates that the worst case T_{SIRF} = 132 cycle. Full compilation of this PF on the Xilinx Virtex 5 xc5vlx50t-3-ff1136 FPGA indicated that this hardware design can support clock frequencies up to 36 MHz. Therefore, we can attain a throughput of $36/132 \times 10^6 = 270 \times 10^3$ PF iterations/s on this platform.

5. Synthesis comparison between the three different SIRF implementations

All the proposed architectures are captured using VHDL, and are synthesized using Xilinx environment on device xc5vlx50t-3-ff1136. Table 5 depicts the resources utilization for the proposed architectures. The total memory requirements for the first and second architectures are one dual-port FIFO of $2M$ words to store the particles vectors of width p (equals to 28 bits), $M \times n$ memory to store the replication factor where $M = 2^n$, and $M \times w$ memory to store the weights of the particles where w equals to 8 bits used in our fixed point representation.

The third implemented distributed RPA with local network, uses $2M/K$ FIFOs for each processor element, plus $(K\text{-}2)M/2K$ memory locations as the local net memory in each processor elements. Totally each processor elements requires a total of $(M/2+M/K)$ location memory to store the particles. The overall required memory is $(M+KM/2)$. In the comparison, the architecture presented in [6] needs M memory locations for each processor elements (M/K inside the processor elements plus $(K\text{-}1)M/K$ memory locations at the CU). Thus, the total memory locations required in [6] design is MK. Thus, our implementation has a resource reduction advantage of $M(K - 2)/2$ in addition to a speed advantage without compromising the SIRF performance.

Table 3 The timing comparison in the worst case and the best case

HW implementation	Worst case	Best case
The two-step architecture with sequential resampling	$a = 2$ $3M + 1+TL$ *196 cycle*	$a = 1$ $2M + 1+TL$ *132 cycle*
The two-step architecture with parallel resampling	$a = 2$ $2M + 2 + TL$ *132 cycle*	$a = 1$ $M + 2 + TL$ 69 cycle
The distributed PF	$a = 2$ and $b = 1$ $2M/K + K+M +2+TL$ *93 cycle*	$a = 1$ and $b = 0$ $2M/K + K +2 +TL$ *29 cycle*

For several hundred particles, the ratio between the latencies cycles and the number of particles affects the total execution time. For our object tracking application, the moving object moves between two consecutive frames in an area of 32 × 32 pixels. So, the number of particles, needed to represent this state space, is of moderate value. That is why it is important to consider the latencies cycles. For example, in the implementation of Athalye et al. [7], the latency of sampling and importance computation units are 8 and 53 cycles, respectively, giving a total value for $TL = 61$ cycles. If $M = 10000$ (as in case of reference [7]) this TL value is insignificant. However, if $M = 64$ (as in our case) this TL value is close to 100% of the computation time of the total sampling time. In our implementations, TL is reduced to *3* cycles, leading to a latency time overhead of *3/64*, less than *5%*.

6. Conclusion

In this article, three novel architectures for SIRF are proposed. The first architecture is a two-step architecture with sequential resampling, where particle sample, weight calculation, and output calculation are carried out in parallel during the first step, followed by sequential resampling in the second step. This first architecture serves as a core unit for the next proposed architectures. The second architecture speeds up the resampling step (second step) via a parallel, rather than a serial, architecture. The third architecture implements the SIRF as a distributed PF composed of several PEs and a CU. Each PE is in fact the two-step core of the first architecture.

Table 4 Worst case throughputs of proposed architectures (M = 64)

HW implementation	Worst case	Maximum clock frequency (MHz)	Throughput
The two-step architecture with parallel resampling	132 cycle	36	270×10^3 iterations/s
The distributed PF	93 cycle	74	795×10^3 iterations/s

Table 5 The resources utilization for the three proposed architectures on the Xilinx Virtex 5 FPGA xc5vlx50t-3-ff1136

Device utilization summary				
Resource	**Single PE with sequential resampling**	**Single PE with parallel resampling**	**Distributed PF 8 PE**	**FPGA available resources**
Slice registers	891	967	9049	28800
Slice LUTs	1203	4430	15017	28800
Fully used Bit Slices	364	746	2517	21549
Bonded IOBs	28	28	148	480
Block RAM/FIFO	1	1	4	60
BUFG/ BYFGCTRLs	2	2	2	32
DSP48Es	5	48	17	48

All the proposed architectures are captured using VHDL and are synthesized using Xilinx environment and verified using Modelsim simulator.

The main features of the proposed architectures are

- Memory addressing is eliminated via the efficient use of dual-port FIFOs to store the particles' state vectors. Consequently, significant speed up of the PF process is achieved. There is no need to add memories to store the particles addresses or indexes as in [2,7]. The overall memory size is $2M$ for the sequential PF. In comparison, required memory sizes for the PFs of references [2,7] are $3M$ and $4M$, respectively. The FIFO eliminates the need to read, write, or store the states addresses/indexes in separate memory, and hence, reduces the execution cycle time.
- The use of linearized weight likelihood functions instead of the exponential functions. This feature captures the localization of the likelihood function while reducing the hardware resources needed to evaluate it.
- The second architecture speeds up the resampling by a factor of M via a parallel, rather than a serial, architecture.
- The third distributed PF architecture is implemented by several PEs with simple but efficient ring interconnection network. The local routing of the particles among the PEs executed in parallel with the particle generation, weight, and output calculations pipeline step. The proposed ring interconnection network delay is in the range of zero cycle in the best case to $(K-1)M/K$ cycle in the worst case when only one particle has a non-zero weight. The proposed distributed PF achieves a total execution

time of ($2M/K + K + 2 + TL$) in the best case to ($2M/K + K + M + 2 + TL$) in the worst case. In comparison, the PF designs of references [2,6] require $3M/4 + 2$ cycles and ($2M/K + K + Mr + TL$) cycles, respectively. Mr is the delay due to particles routing assuming the same latencies TL. In addition our distributed PF gain a reduction of memory size of $M(K-2)/2$ when compared with the parallel PF proposed in [6].

Author details
[1]Nuclear Research Center, Atomic Energy Authority, Cairo, Egypt [2]Faculty of Engineering, Cairo University, Giza, Egypt

Competing interests
The authors declare that they have no competing interests.

Received: 23 April 2011 Accepted: 23 January 2012
Published: 23 January 2012

References
1. J Miguez, Analysis of parallelizable resampling algorithms for particle filtering. Signal Process J. **87**, 3155–3174 (2007). doi:10.1016/j.sigpro.2007.06.011
2. S Hong, S Chin, PM Djurc, M Bolic, Design and implementation of flexible resampling mechanism for high-speed parallel particle filters. J VLSI Signal Process. **44**, 47–62 (2006). doi:10.1007/s11265-006-5919-9
3. D Marimon, Y Maret, Y Abdeljaoued, T Ebrahimi, Particle filter-based camera tracker fusing marker and feature point cues. in *Proceedings of IS&T/SPIE Conference on Visual Communication and Image Processing*. **6508** (2007)
4. N de Freitas, Rao-Blockwellised particle filtering for fault diagnosis. in *IEEE Aerospace Conference Proceedings*. **4**, 493 (2002)
5. B Zhang, W Tian, Z Jin, Robust appearance-guided particle filter for object tracking with occlusion analysis. Int J Electron Commun. **62**, 24–32 (2008). doi:10.1016/j.aeue.2007.01.006
6. M Boli'c, Architectures for efficient implementation of particle filters. Ph D thesis, Stony Brook University (August 2004)
7. A Athalye, M Bolic, S Hong, PM Djuric, Generic hardware architectures for sampling and resampling in particle filter. EURASIP J Appl Signal Process. **17**, 2888–2902 (2005)
8. J Alarcon, I Lopez, A new real-time hardware architecture for road line tracking using a particle filter. in *IEEE Industrial Electronics IEC Annual Conference*, Paris 736–741 (2006)
9. J Uk Cho, JE Byun, H Kang, A real-time object tracking system using a particle filter. in *Proceedings of intelligent robots and Systems Conf. IEEE/RSJ*, Beijing, China 2822–2827 (October 2006)
10. J Uk Cho, SH Jin, XD Pham, JW Jeon, Object tracking circuit using particle filter with multiple features. in *Proceedings of SICE-ICASE International Joint Conf.*, Bexco, Busan, Korea 1431–1436 (October 2006)
11. R Velmurugan, Implementation strategies for particle filter based target tracking. Ph D Thesis, Georgia Institute of Technology (May 2007)
12. H Medeiros, X Gao, J Park, A parallel implementation of the color based particle filter for object tracking. in *Computer Vision and Pattern Recognition Workshops, CVPRW '08, IEEE Comput Soc Conf* 1–8 (June 2008)
13. S Saha, NK Bambha, SS Bhattacharyya, Parameterized design framework for hardware implantation of particle filters. in *Proceedings of the International Conf on Acoustics, Speech and Signal Processing*, Las Vegas, Nevada 1449–1452 (March 2008)
14. G Hendeby, R Karlsson, F Gustafesson, Particle filtering: the need for speed. EURASIP J Adv Signal Process. **2010** (2010). Article ID 181403, 9
15. Howida A. Abd El-Halym, II Mahmoud, SED Habib, Efficient hardware architecture for particle filter based object tracking. in *Proceeding of IEEE International Conference on Image Processing (ICIP), Hong Kong* 4497–4500 (September 2010)
16. MS Arulampalam, S Maskell, N Gordon, T Clapp, A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. IEEE Trans Signal Process. **50**, 174–188 (2002). doi:10.1109/78.978374
17. NJ Gordon, DJ Salnoda, AFM Smith, Novel approach to nonlinear/non-Gaussian Bayesian state estimation. IEE Proc F. **140**(2), 107–113 (1993)
18. HW Sorenson, DL Alsach, Recursive-Bayesian estimation using gaussian sums. Automatica. **7**, 465–479 (1971). doi:10.1016/0005-1098(71)90097-5
19. M Bolic, S Hong, PM Djurc, Performance and complexity analysis of adaptive particle filtering. in *Proceedings of the conference on Signals, Systems and Computers, CA* 853–857 (November 2002)
20. J Cho, IE Byun, H Hang, A real-time object tracking system using particle filter. in *Proceedings of Intelligent Robots and Systems Conf IEEE/RSJ*, Beijing, China 2822–2827 (October 2006)
21. K Bai, W Liu, Improved object tracking with particle filter and mean shift. in *Proceedings of the IEEE International Conf on Automation and Logistics* Jinan, China 431–435 (2007)
22. Howida A. Abd El-Halym, II Mahmoud, A AbdelTawab, SED Habib, Appraisal of an enhanced particle filter for object tracking. in *Proceedings of IEEE International Conference on Image Processing (ICIP)*, Cairo, Egypt 4105–4107 (November 2009)
23. Howida A. Abd El-Halym, II Mahmoud, A AbdelTawab, SE-D Habib, Particle filter versus particle swarm optimization for object tracking. in *Proceedings of ASAT Conf*, Cairo, Egypt (May 2009)
24. II Mahmoud, A Abd ElTawab, Hardware genetic algorithm for robot motion path planning. in *Proceedings of 4th STCEX* Riyad, KSA. **2**, 359–366 (2006)
25. M Bolic, A Athalye, PM Djuric, S Hong, Algorithmic modification of particle filters for hardware implementation. Eur Signal Process Conf Vienna, Austria 1641–1644 (September 6–10, 2004)
26. http://www.mathworks.com/matlabcentral/fileexchange/17960
27. M Bolic, PM Djuri, S Hong, Resampling algorithms and architectures for distributed particle filters. IEEE Trans Signal Process. **53**, 2442–2450 (2005)

doi:10.1186/1687-6180-2012-17
Cite this article as: Abd El-Halym *et al.*: **Proposed hardware architectures of particle filter for object tracking.** *EURASIP Journal on Advances in Signal Processing* 2012 **2012**:17.