EURASIP Journal on Applied Signal Processing 2005:17, 2888–2902
© 2005 Hindawi Publishing Corporation

Generic Hardware Architectures for Sampling and Resampling in Particle Filters

Akshay Athalye
Department of Electrical and Computer Engineering, Stony Brook University, Stony Brook, NY 11794-2350, USA
Email: athalye@ece.sunysb.edu

Miodrag Bolić
Department of Electrical and Computer Engineering, Stony Brook University, Stony Brook, NY 11794-2350, USA
Email: mbolic@ece.sunysb.edu

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

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

Received 18 June 2004; Revised 11 April 2005; Recommended for Publication by Markus Rupp

Particle filtering is a statistical signal processing methodology that has recently gained popularity in solving several problems in signal processing and communications. Particle filters (PFs) have been shown to outperform traditional filters in important practical scenarios. However their computational complexity and lack of dedicated hardware for real-time processing have adversely affected their use in real-time applications. In this paper, we present generic architectures for the implementation of the most commonly used PF, namely, the sampling importance resampling filter (SIRF). These provide a generic framework for the hardware realization of the SIRF applied to any model. The proposed architectures significantly reduce the memory requirement of the filter in hardware as compared to a straightforward implementation based on the traditional algorithm. We propose two architectures each based on a different resampling mechanism. Further, modifications of these architectures for acceleration of resampling process are presented. We evaluate these schemes based on resource usage and latency. The platform used for the evaluations is the Xilinx Virtex II pro FPGA. The architectures presented here have led to the development of the first hardware (FPGA) prototype for the particle filter applied to the bearings-only tracking problem.

Keywords and phrases: particle filters, hardware architectures, memory schemes, real-time processing, bearings-only tracking.

1. INTRODUCTION

Particle filters (PFs) [1, 2] are used to perform filtering for models that are described using the dynamic state-space approach [1]. Many problems in signal processing and communications can be described using these models [3]. In most practical scenarios, these models are nonlinear, the states are high-dimensional, and the densities involved are non-Gaussian. Traditional filters like the extended Kalman filter (EKF) are known to perform poorly in such scenarios [4]. PFs on the other hand are not affected by the conditions of nonlinearity and non-Gaussianity and handle high-dimensional states better than traditional filters [5].

PFs are Bayesian in nature and their goal is to find an approximation to the posterior density of a state of interest (e.g., position of a moving object in tracking, or transmitted symbol in communications) based on corrupted observations which are inputs to the filter. In the traditional PFs known as sample importance resample filters (SIRFs), this posterior is represented by a random measure consisting of a weighted set of samples (particles). The particles are drawn or sampled from a density known as the importance function (IF) using the principle of importance sampling (IS) [1]. This sampling step is followed by the importance computation step which assigns weights to the drawn particles based on received observations using IS rules to form the weighted set of particles. Another important process called resampling is generally needed in PFs to avoid weight degeneracy [2]. Various estimates of the state like MMSE or MAP estimates can be calculated from this weighted set of particles. The number of particles used to compute the posterior depends upon the nature of application, dimension of the state, and the performance requirements. From here on, we will refer to the number of particles used as M and the dimension of the state as N_s.

The main drawback of SIRFs is their computational complexity. For each observation received, all the M particles need to be processed through the steps of sampling, importance computation, and resampling. The sampling and importance computation steps typically involve transcendental exponential and nonlinear operations. Once all the M particles have been processed through the above-mentioned steps, the estimate of the state at the sampling instant is calculated and the next input can be processed. These operations present significant computational load even on a state-of-the-art DSP. The performance of SIRF for the bearings-only tracking problem [6] with $N_s = 4$ was evaluated on TI TMS320C54x generation DSP [7]. With $M = 1000$ particles, the inputs to the filter could be processed at the rate of only 1 kHz. Clearly this speed would prevent the use of PFs for online signal processing in real-time applications where higher sampling rates and/or higher number of particles are needed for processing. Thus, design of dedicated hardware for SIRF is needed if real-time applications are to become feasible.

SIRF is a recursive algorithm. The sampling step uses the resampled particles of the previous instant to compute the particles of the current instant. This requires the particles to be stored in memories. We will see later that a straightforward implementation of the traditional SIRF algorithm would have a memory requirement of $2N_s \times M$ since sampled and resampled particles need to be stored in different memories. Most practical applications involve nonlinear models and high-dimensional states (large N_s) which implies a large number of particles M for SIRFs applied to these problems [8]. This would make the total memory requirement of $2N_s \times M$ very large. The architectures proposed in this paper reduce this memory requirement to N_s memories of depth M (i.e., $N_s \times M$). This not only reduces the hardware resource requirement of the SIRF but also makes it more energy-efficient due to reduced memory accesses [9].

The specifics of an SIRF implementation depend upon the properties of the model to which the SIRF is applied. However, from a hardware viewpoint, the high-level data flow and control structure remain the same for every model. This paper focuses on the development of efficient architectures for these generic operations of the SIRF. They include the resampling step and memory-related operations of the sample step. The other model-dependent operations are data driven and involve mathematical computations. They can be easily incorporated into the proposed architectures for any model. We develop two architectures, one using the traditional systematic resampling (SR) algorithm and the other using the new residual systematic resampling (RSR) algorithm. These architectures are referred to as scheme 1 and scheme 2, respectively. As we will see later, the resampling operation in the SIRFs presents a bottleneck since it is inherently sequential and also cannot be executed concurrently (pipelined) with other operations. Scheme 1 has a low complexity and simple control structure, but is generically slow since SR involves a *while* loop inside an outer *for* loop. As opposed to this, the RSR algorithm has a single *for* loop and hence scheme 2 is faster than scheme 1. We also propose modifications of these schemes which bring about partial parallelization of resampling and reduce the effect of the resampling bottleneck on execution throughput.

The rest of the paper is organized as follows. In Section 2 we briefly describe the theory behind the SIRF, the traditional SR algorithm, and the new RSR algorithm. Section 3 presents the proposed architectures and their modification for increased speed. Evaluation of resource utilization and latency of the two schemes on FPGA platform along with an example application is presented in Section 4. Section 5 concludes the paper.

2. BACKGROUND

2.1. The SIRF algorithm

Dynamic state space (DSS) models to which SIRFs can be applied are of the form

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

$$\mathbf{y}_n = g_n(\mathbf{x}_n, \mathbf{v}_n), \tag{2}$$

where f_n and g_n are possibly nonlinear functions describing the DSS model. The symbol $\mathbf{x}_n$ represents the dynamically evolving state of the system, $\mathbf{q}_n$ represents the process noise, and $\mathbf{y}_n$ is the observation vector of the system which is corrupted by the measurement noise $\mathbf{v}_n$ at instant n. The SIRF algorithm estimates the state of the system $\mathbf{x}_n$ based on the received corrupted observations. The algorithm proceeds through the following steps.

(1) Sampling step (S). In this step, samples (particles) of the unknown state are drawn from the IF. In our implementation we choose the prior probability density function (pdf) of the state, given by $p(\mathbf{x}_n \mid \mathbf{x}_{n-1})$ to be the IF. This prior pdf can be deduced from (1). The sampling step can be thought of as propagation of particles at time instant $n-1$ into time instant n through (1). The sampled set of particles at instant n is denoted by $\{\mathbf{x}_n^{(m)}\}_{m=0}^{M-1}$. The SIRF is initialized with a prior set of particles at time instant 0 to start the recursion. These particles are then successively propagated in time.

(2) Importance (weight) computation step (I). This step assigns importance weights $w_n^{(m)}$ to the particles $\mathbf{x}_n^{(m)}$ based on the received observations. This step is the most computationally intensive and generally involves the computation of transcendental trigonometric and exponential functions. Since we use the prior IF for our implementation, the weight assignment equation is

$$w_n^{(m)} = w_{n-1}^{(m)} \cdot p\left(\mathbf{y}_n \mid \mathbf{x}_n^{(m)}\right). \tag{3}$$

```
for $m = 0$ to $M - 1$
    (1) Sampling step. Draw sample $\mathbf{x}_n^{(m)}$ from $p(\mathbf{x}_n \mid \mathbf{x}_{n-1}^{(m)})$.
    (2) Importance computation step. Calculate the weight
        $w_n^{(m)} = p(y_n \mid \mathbf{x}_n^{(m)})$.
end
    (3) Resampling step. Determine the resampled set of
        particles $\{\tilde{\mathbf{x}}_n^{(m)}\}_{m=0}^{M-1}$
    (4) Output calculation. Calculate the desired (like
        MMSE, MAP) estimate of the state $\hat{\mathbf{x}}_n$.
```

Pseudocode 1: SIRF algorithm.

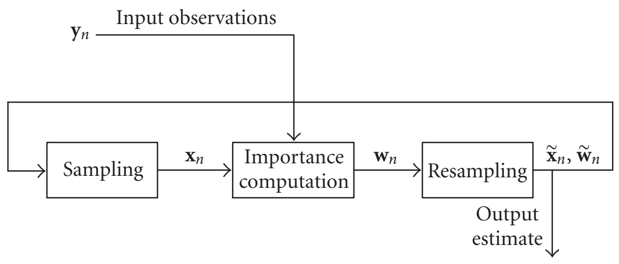


Figure 1: Overall structure of the SIRF.

The implementation of this step is largely dependent on the nature of the DSS model for the particular problem at hand and therefore is not discussed in this paper.

(3) Resampling step (R). Resampling prevents degeneration of particle weights by eliminating particles with small weights and replicating particles with large weights to replace those eliminated. This replication or discarding is done based on some function of the particle weights. The resampled set of particles is denoted by $\{\tilde{\mathbf{x}}_n^{(m)}\}_{m=0}^{M-1}$, and the weights of these particles after resampling are denoted by $\{\tilde{w}_n^{(m)}\}_{m=0}^{M-1}$, which are typically $1/M$. The resampled particles along with their weights form a random measure $\{\tilde{\mathbf{x}}_n^{(m)}, \tilde{w}_n^{(m)}\}_{m=0}^{M-1}$ which is used to represent the posterior $p(\mathbf{x}_n \mid \mathbf{y}_{1:n})$ and calculate estimates of the state.

The SIRF algorithm is summarized in Pseudocode 1. Figure 1 shows the overall structure of the SIRF. The recursive nature of the filter can be seen from the presented data flow. The sample and importance steps can be pipelined in operation. Resampling requires knowledge of sum of all particle weights. Hence it cannot begin before the weights of all the particles have been calculated. The sample step of time $n + 1$ cannot begin until resample step of time n has been completed. Thus, resampling cannot be executed concurrently with any other operation and presents a bottleneck which limits the sampling rate of the SIRF. The architectures presented in this paper reduce the effect of this bottleneck by using efficient memory schemes and modified resampling algorithms.

2.2. Systematic resampling in SIRF

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

$$Pr\left(\tilde{\mathbf{x}}_n^{(i)} = \mathbf{x}_n^{(j)}\right) = w_n^{(j)}. \tag{4}$$

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

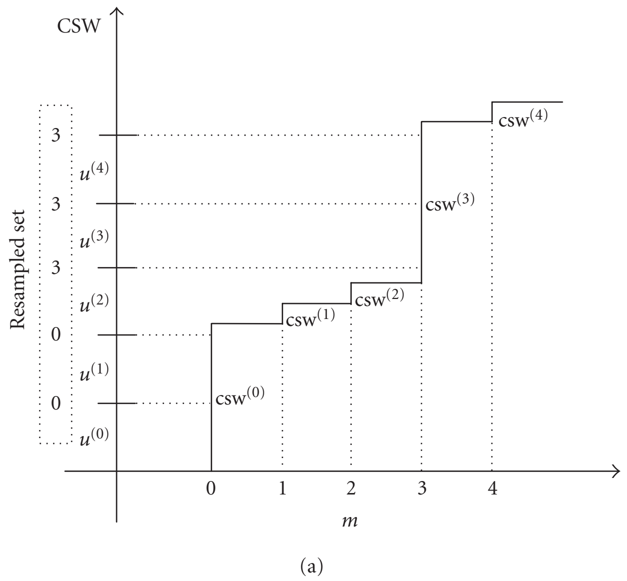
The SR concept for a PF that used 5 particles is shown in Figure 2a. First the cumulative sum of weights (CSW) of sampled particles is computed. This is presented in the figure for the case of 5 particles ($M = 5$) with weights $w^{(0)}, \ldots, w^{(4)}$. Then, as shown on the y axis of the graph, a function $u^{(m)}$ called the resampling function is systematically updated [1] and compared with the CSW of the particles. The corresponding particles are replicated to form the resampled set which for this case is $\{\mathbf{x}^{(0)}, \mathbf{x}^{(0)}, \mathbf{x}^{(3)}, \mathbf{x}^{(3)}, \mathbf{x}^{(3)}\}$. In the traditional SR algorithm, it is essential for the weights to be *normalized* such that their sum is one. However we use a modified resampling algorithm that avoids weight normalization by incorporating the sum of weights into the resampling operation [10]. This avoids M divisions per SIRF recursion which is very advantageous for hardware implementation.

The determination of the resampled set of particles is done sequentially as is shown in Figure 2b. In each cycle, depending on the results of comparison between the two numbers U and CSW, which represent the current value of the resampling function and the CSW respectively, the relevant particle is replicated or discarded and the value of either the resampling function U or the cumulative sum of weights CSW is updated. As shown in the figure, in the first cycle, $u^{(0)}$ and $\text{csw}^{(0)}$ are compared. Since CSW > U, particle 0 is replicated and the *resampling function* is updated, while in cycle 4, since CSW < U, particle 1 is discarded and the CSW is updated. This process is repeated till the replicated set of particles is obtained.

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

2.3. Residual systematic resampling algorithm

In spite of the low hardware complexity, the low speed of the SR algorithm may not be tolerable in case of high-speed applications. For these cases, the residual systematic resampling (RSR) algorithm proposed in [11] can be used. This algorithm has a single *for* loop of M iterations and hence is twice faster than SR in terms of number of cycles. This algorithm is based on the traditional residual resampling algorithm [12]. In residual resampling (RR) the number of replications of a specific particle $\mathbf{x}^{(m)}$ is determined by truncating the product of the number of particles M and the particle weight $w^{(m)}$. The result is known as a replication factor. The sum of the replication factors of all particles, except for some special cases, is less than M. These remaining particles are



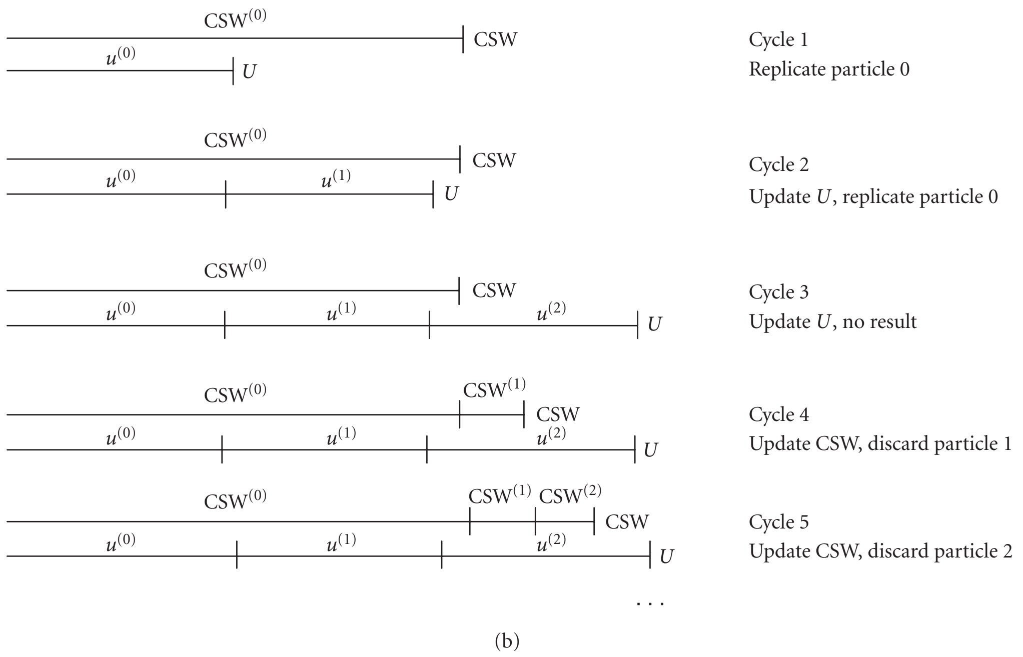


FIGURE 2: The concept of systematic resampling. (a) Resampling using cdfs. (b) Resampling done sequentially.

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

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

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

PSEUDOCODE 2: Residual systematic resampling (RSR) algorithm.

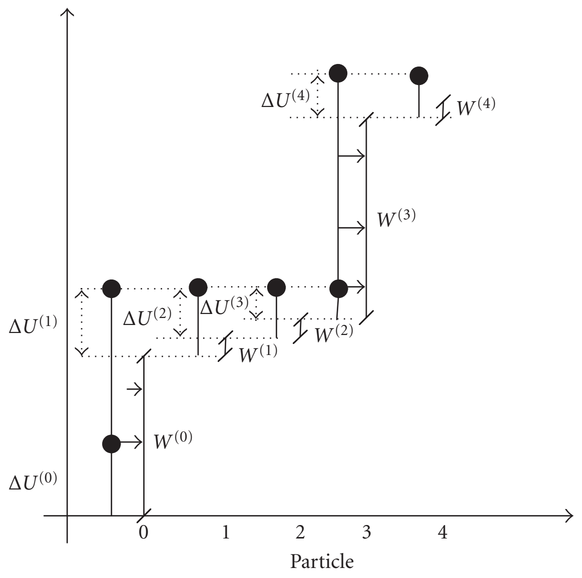


FIGURE 3: Residual systematic resampling for an example with $M = 5$ particles.

3. ARCHITECTURES AND MEMORY SCHEMES

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

3.1. Reduction of memory requirement

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

$$\mathbf{x}_n^{(m)} \sim p\left(\mathbf{x}_n \mid \tilde{\mathbf{x}}_{n-1}^{(m)}\right), \quad m = 0, 1, \ldots, M-1. \tag{5}$$

A straightforward implementation of the SIRF would require $2 \times N_s$ memories of depth M, N_s for storing the sampled particles $\{\mathbf{x}_n^{(m)}\}_{m=0}^{M-1}$, and N_s for storing the resampled particles $\{\tilde{\mathbf{x}}_n^{(m)}\}_{m=0}^{M-1}$. This implementation is shown in Figure 4a. At time instant n, the sampled particles $\{\mathbf{x}_n^{(m)}\}_{m=0}^{M-1}$ will be stored in the memory labelled SMEM. Their weights will be calculated in the importance computation step. Once all the weights have been determined, the resampling unit determines the resampled set of particles $\{\tilde{\mathbf{x}}_n\}_{m=0}^{M-1}$, which are written to the memory labelled RMEM. The sample unit then reads particles from RMEM for propagation. These memories are shown in Figure 4a for $N_s = 1$.

The memory schemes proposed here reduce this requirement to N_s memories of depth M. In our implementation, the resampling unit returns the set of indexes (pointers) of the replicated particles instead of the particles themselves. Then indirect addressing [13] can be used to read the set $\{\tilde{\mathbf{x}}_n\}_{m=0}^{M-1}$ from the sample memory SMEM itself for propagation. This means that the particles are propagated in the following manner:

$$\mathbf{x}_n^{(m)} \sim p\left(\mathbf{x}_n \mid \mathbf{x}_{n-1}^{\mathrm{ind}(m)}\right), \tag{6}$$

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

This execution can be speeded up by using *dual-port* memories [14] which are readily available on an FPGA platform.[1] This enables reading of $\{\tilde{\mathbf{x}}_{n-1}\}_{m=0}^{M-1}$ and writing of $\{\mathbf{x}_n^{(m)}\}_{m=0}^{M-1}$ to be executed concurrently. Hence, the sample step for M particles can be done in $M + L_S$ cycles. The memory scheme is shown in Figure 4b where the single dual-port memory labelled PMEM replaces the memories SMEM and RMEM of Figure 4a. Thus, use of index addressing reduces the memory requirement of the SIRF and use of dual-port memories reduces the execution cycle time.

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

Consider the following one-dimensional random walk model:

$$\begin{aligned} x_n &= x_{n-1} + q_n, \\ y_n &= x_n + v_n. \end{aligned} \tag{7}$$

Here x_n represents the one-dimensional state of the system and y_n is a noisy measurement. The symbols q_n and v_n are the process and the measurement noises, respectively. Consider 5 sampled particles at instant $n - 1$ (i.e., $\{x_{n-1}^{(m)}\}_{m=0}^{4}$). In the implementation of Figure 4a, these

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

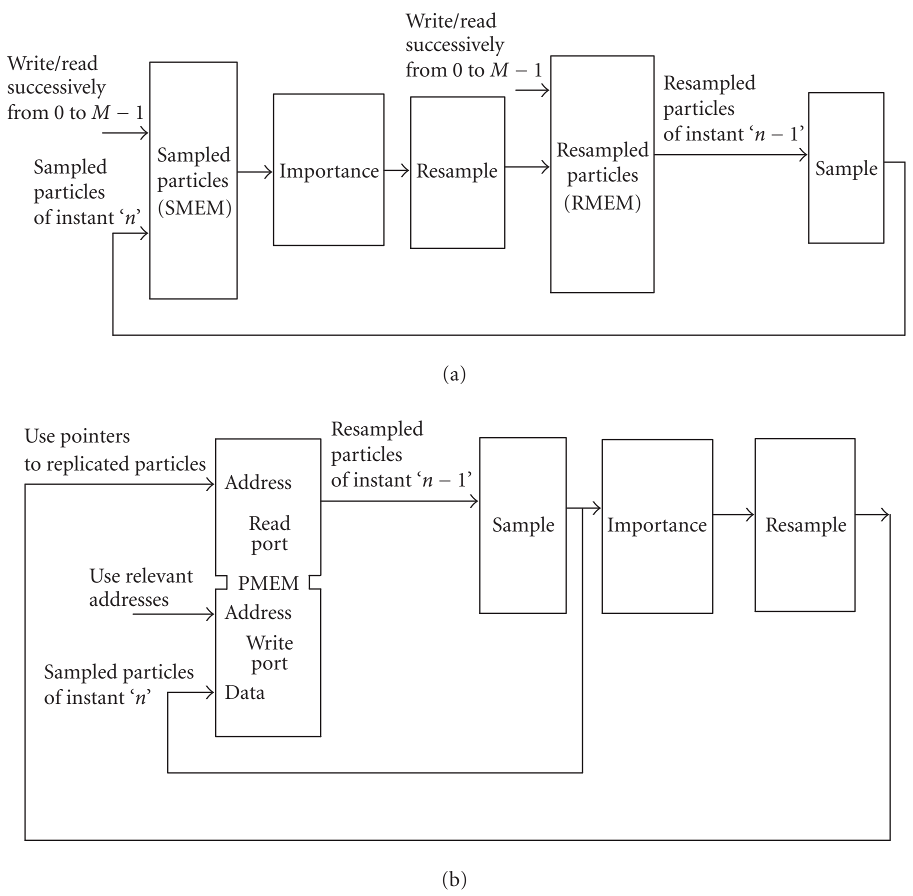

Write/read successively from 0 to $M-1$
Sampled particles of instant 'n'
Sampled particles (SMEM)
Importance
Resample
Write/read successively from 0 to $M-1$
Resampled particles (RMEM)
Resampled particles of instant '$n-1$'
Sample
(a)
Use pointers to replicated particles
Address
Read port
Use relevant addresses
PMEM
Address
Write port
Sampled particles of instant 'n'
Data
Resampled particles of instant '$n-1$'
Sample
Importance
Resample
(b)

FIGURE 4: Memories for storing particles. In the traditional implementation two memories would be needed. These are replaced by a single dual-port memory. (a) Resampling requiring two memories. (b) Modified architecture needing only one dual-port memory.

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

(a)

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

(b)

FIGURE 5: Memory operations in sample step, (a) for implementation with two memories and (b)for implementation with one memory.

particles will be stored in the memory SMEM at locations SMEM[0], ..., SMEM[4]. Suppose that after resampling, particle $x_{n-1}^{(0)}$ is replicated twice, $x_{n-1}^{(3)}$ three times, and that particles $x_{n-1}^{(1)}$, $x_{n-1}^{(2)}$, and $x_{n-1}^{(4)}$ are discarded. In the implementation with two memories, the resampled particles will be written to memory RMEM. The operations performed in the sample step at instant n for this case are shown in Figure 5a.

As seen from the figure, the sampled particles are written to the memory SMEM. In the reduced memory implementation of Figure 4b, the replicated particles are read out of the same single memory (PMEM in this case) using resampled indexes. For this example, the set of indexes of replicated particles is $\{0, 0, 3, 3, 3\}$. Thus the operations of the sample step will be as shown in Figure 5b. However, if sampled particles are written to successive locations of PMEM as in the previous case, the particle $x_n^{(m)}$ will overwrite the resampled particle $\tilde{x}_{n-1}^{(m)}$ causing an error if this particle has been replicated multiple times. In the above example, if the particle $x_n^{(0)}$ is written to PMEM[0], then for the next particle, we will get

$$x_n^{(1)} = x_n^{(0)} + q_n^{(1)} \tag{8}$$

which is incorrect. Thus different strategies for writing sampled particles to the memory need to be devised for the

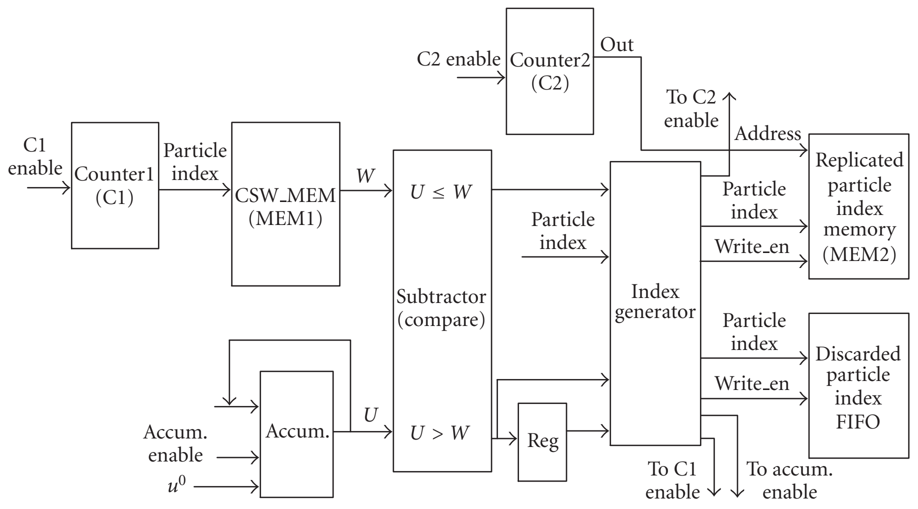


FIGURE 6: Architecture of resampling unit implementing SR.

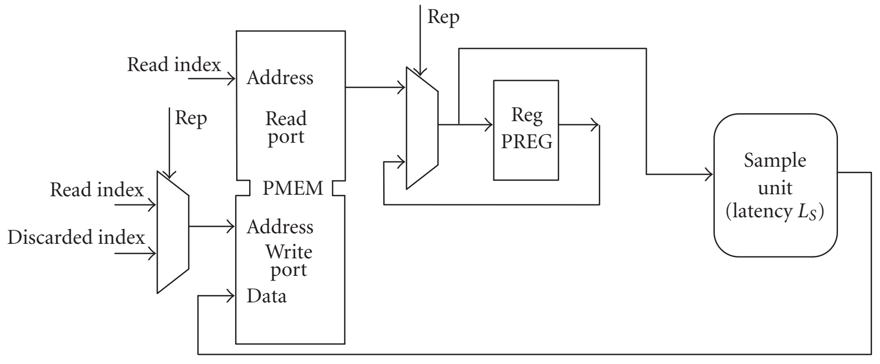


FIGURE 7: Architecture of sample unit.

reduced memory design to function correctly. In the following sections, we will explain the architectures developed using SR and RSR and how they handle reading and writing of the particle memory.

3.2. SIRF using systematic resampling: scheme 1

Figure 6 shows the architecture for the resampling unit implementing the SR mechanism. The CSW is stored in the memory labelled MEM1 at locations corresponding to the ordinal number of the particle in the sampled set. The resampling function $u^{(m)}$ is generated using an accumulator as shown. Both the memory and the accumulator are controlled by enable signals. The outputs of the accumulator and the memory (U and W) are compared for conditions $U \leq W$ and $U > W$ by using a subtractor. The values of the CSW are read from the memory using one counter (C1). The results of the comparison are passed to the *index generator* unit which determines whether to replicate or discard the particle (i.e., the index of the particle). The indexes of the replicated particles are stored in the memory MEM2.

This scheme also records the indexes of the discarded particles. These indexes are used while writing the sampled (propagated) particles back to the memory. A particle, which has been generated by a replication, is written to the location of a discarded particle in the memory. The number of particles before and after resampling is the same. This means that for every replicated particle there will be a discarded particle. Hence this scheme can be used effectively for writing particles to the memory. Since the number of particles that will be discarded is nondeterministic, we use a FIFO buffer of depth M to store the discarded particle indexes. The output of counter C1 at an instant is the index of the particle that is currently being processed. The comparator and the index generator unit bring about the resampling as in Figure 2. If the particle is replicated, its index is written to MEM2 whose locations are addressed by counter C2, and the accumulator is enabled so as to update the value of the resampling function. If the particle is discarded or when all its replications are found, counter C1 is enabled CSW of the next particle is read from the memory. When an index is to be discarded, the write enable of the FIFO buffer is asserted and the index is written to it.

Figure 7 shows the architecture for the sample step under this scheme. Once resampling is done, the memory MEM2

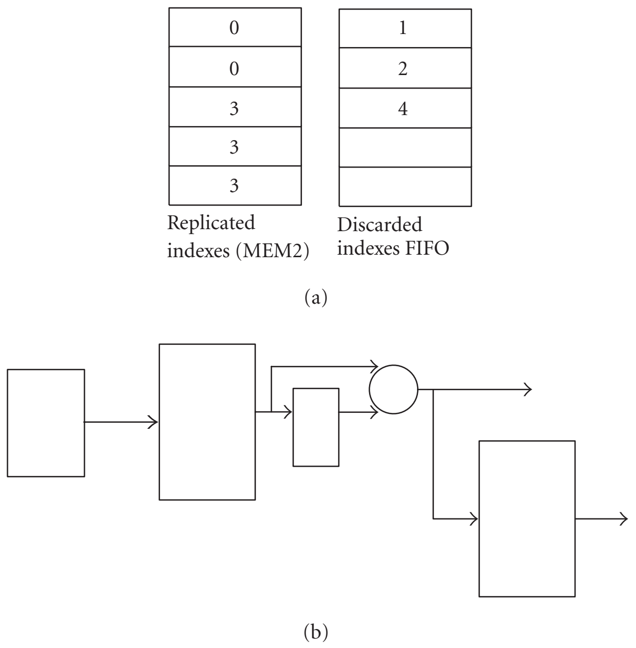


Figure 8: Addressing read and write ports of particle memory using stored indexes. (a) Contents of memory and FIFO after resampling. (b) Reading of replicated and discarded indexes.

Table 1: Function of sample step. Note that writing of particles is done L_S cycles after they are read, where L_S is the latency of the sample unit.

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

represents the array ind(m) containing M replicated indexes. This memory is read sequentially and the indexes are used as addresses to the read port of the dual-port memory (PMEM) storing the particles. The output of this memory is the set $\{\tilde{\mathbf{x}}_n\}_{m=0}^{M-1}$. Due to the nature of the SR algorithm, all the replications of a particular index will be written to successive locations in MEM2. Thus, since this memory is read sequentially, a replication can be detected by comparing the current read index with the previous one. When an index is read from MEM2 for the first time, the corresponding particle is read from the memory and stored in the temporary register PREG. After propagation this particle is written to its original location in the memory. When the same index is read from MEM2 in the following cycle, replication is detected, and the particle is read from the temporary register PREG rather than from the memory (since the location in the memory will be overwritten by the propagated particle). Also, the read enable of the FIFO is asserted high and a discarded index is obtained which is used as address to the write port of PMEM to write the replicated particle after propagation (see Figure 8b). We now further illustrate this scheme with our previous example.

Following the same case of the example, the contents of the replicated index memory MEM2 and discarded index FIFO in Figure 6 will be as shown in Figure 8a. The sample step starts by reading of the content of MEM2. The operations in various cycles are listed in Table 1. Figure 8b shows how the FIFO is read. A replication is detected by comparing the current read index with the previous one. From the index memory contents, we see that in this case a replication will be indicated when MEM2[1], MEM2[3], and MEM2[4] are read. The result of this comparison is used as read enable to the FIFO.

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

3.3. *Modification of scheme 1 for reduced execution time*

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

Due to the systematic nature of the resampling function update, the final value of the resampling function is fixed. This value is $u^{(0)} + (M - 1)/M$ for traditional SR and $u^{(0)} + S(M - 1)/M$ for our implementation of resampling using nonnormalized weights [10]. Also the final value of the CSW, S, is also known to us. We can use this property of SR to split the resampling shown in Figure 2a into two concurrent loops of $M/2$ iterations each. One loop determines the first $M/2$ resampled indexes by comparing $\text{csw}^{(0)}$ to $\text{csw}^{(M/2-1)}$ with $u^{(0)}$ to $u^{(M/2-1)}$ and the other loop determines the next $M/2$ resampled indexes by comparing $\text{csw}^{(M/2)}$ to $\text{csw}^{(M-1)}$ with $u^{(M/2)}$ to $u^{(M-1)}$. From a hardware viewpoint, this would require reading of two values of the CSW simultaneously from the memory which can be accomplished by storing the CSW values (MEM1 in Figure 6) in a dual-port memory. Also the replicated particle index memory MEM2 would need to be dual port and the discarded index FIFO would be replaced by two FIFOs of half the size. All other logic blocks in Figure 6 would be replicated. This would reduce the loop bound [15] of the SIRF recursion and increase its throughput. With this scheme, SR is split up into two parallel loops of $M/2$ iterations each. Execution time of SR is reduced to $2 \times M/2 - 1 = M - 1$ cycles at the cost of added hardware. As an extension of this concept, resampling can be split up into more than 2 loops of simultaneous comparisons due to the systematic update of the resampling function. However this would need more memory blocks and additional hardware. The tradeoff between added hardware and obtained speed is considered in Section 4.

3.4. *SIRF with residual systematic resampling: scheme 2*

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

Memory usage for the example described in Section 3.1 is shown in Figure 9, where the indexes are arranged in the memory using the above-mentioned method and the corresponding replication factors are stored in a separate memory.

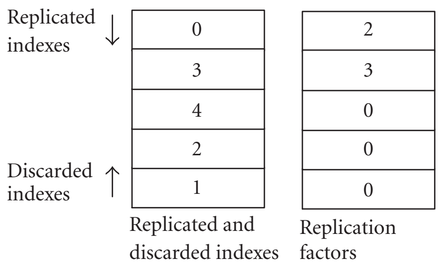


FIGURE 9: Contents of memories after the RSR method with particle allocation with arranged indexes.

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

PSEUDOCODE 3: Modified residual systematic resampling (RSR) algorithm.

From an implementation viewpoint, the RSR algorithm is beneficial since it has a single *for* loop. To make the RSR algorithm suitable for implementation, we make some changes in Pseudocode 2 in Section 2.3. The changes incorporate particle allocation with arranged indexes in the algorithm and also allow for resampling using *nonnormalized* weights. As in the case of SR, this saves M divisions at each instant.

The modified RSR algorithm is shown in Pseudocode 3.

In Pseudocode 3, there is one multiplication inside the loop and one division before the loop. The incorporation of the number K and generation of U from $\mathcal{U}[0,1]$ is done to allow nonnormalized weights in the resampling algorithm. These changes do not affect the correctness of the algorithm and the resampling results produced are the same as Pseudocode 2.

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

The memory-related operations in the sample step are shown in Pseudocode 4. First, the replicated indexes are read sequentially from the memory of arranged indexes as shown in the first *for* loop (line 2). The corresponding replication factors of the indexes are also read at the same time. If the particle has been replicated, then it is propagated repeatedly. This is shown by the second *for* loop (line 5) whose iterations equal the replication factor for that particular index. Then, $r^{(\text{ind}_r)} - 1$ sampled particles are written to the addresses of

```
    $(X)$ = Sampling $(i, r, X)$
(1) $\text{ind}_r = 0, \text{ind}_d = M - 1$
(2) *for* $\text{ind}_r = 1$ *to* $\text{length}(\text{ind}_r)$
(3)     $\text{Reg} = X(i(\text{ind}_r))$
(4)     $X(i(\text{ind}_r)) = \text{Sample (Reg)}, \text{ind}_r = \text{ind}_r + 1$
(5)     *for* $k = r(\text{ind}_r) - 1$ *down to* 1
(6)         $X(i(\text{ind}_d)) = \text{Sample (Reg)}, \text{ind}_d = \text{ind}_d - 1$
(7)     *end*
(8) *end*
```

Pseudocode 4: Memory-related operations of the sample step.

the discarded particles (line 6) by reading the arranged index memory from the bottom. The first sampled particle rewrites the original replicated particle (line 3). Hence the replicated particle has to be stored in a variable Reg.

3.5. Architecture for scheme 2

In this section, the architectures for the algorithms presented in Pseudocodes 3 and 4 are shown in Figures 10 and 11. In Figure 10, weights are stored in the memory MEM_w and addressed by the address counter that counts from 0 to $M - 1$ and corresponds to the variable m in Pseudocode 3. The index generator is the block in which the arithmetics from the lines 5, 6, and 7 from Pseudocode 3 are implemented. The other part of the figure represents the implementation of the particle allocation step (lines 8–12 of Pseudocode 3). MEM_i stores the arranged indexes and MEM_r stores the corresponding replication factors. Depending on whether a particle is replicated or not, its index is written to MEM_i at address pointed to by either the counter counting up (counter_r) or the counter counting down (counter_d). The appropriate replication factor is written to the corresponding location in MEM_r.

There are three main blocks in Figure 11: address generation, address control, and particle generation and storing. One dual-port memory PMEM is used for storing particles. The arithmetics of the sampling step is implemented in the sampling unit. The delay between read and write operations for the memory PMEM is determined by the pipeline latency of the sample unit (L_S). It is presented as Delay_1 in the figure. Counter_f represents the variable k in Pseudocode 4. The replication from the memory MEM_r is used as the initial value to the down counter counter_f. The other logic blocks are for generation of controls to bring about sampling as described in Pseudocode 4.

3.6. Modification of scheme 2 for reduced execution time

The RSR algorithm needs $M+L_{\text{RSR}}$ cycles for execution where the latency due to pipelining of the RSR datapath is L_{RSR} (2 in our case). Similar to the SR, the RSR algorithm can also be parallelized with addition of more hardware for reduced execution time. The RSR algorithm used for scheme 2 has only one loop in which the replication factor of a particle is determined and the value of the resampling function is systematically updated. This algorithm can also be modified for parallel execution by splitting the resampling process into multiple concurrent loops. The modified algorithm for 2 concurrent loops is shown in Pseudocode 5. The first loop does the usual RSR of Pseudocode 3 for the first $M/2$ particles from index 0 to $M/2 - 1$. The second loop does the same simultaneously for the remaining particles from index $M/2$ to M. This algorithm needs the cumulative sum of weights of the first $M/2$ particles. This is denoted as $S^{M/2}$. The initial value of the resampling function for the second loop is denoted by U^2 in the pseudocode. Once again the factor K is included so as to allow resampling using nonnormalized weights. This mechanism can also be directly extended to include more than two loops at the cost of adding more memory and hardware.

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

4. EVALUATION

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

4.1. Execution time

Figure 12 shows the timing of operations for one recursion of the SIRF. In the figure, L_S and L_I represent the startup latencies of the sample and importance unit, respectively. T_{res} is the number of cycles required for resampling. The total cycle time of the SIRF is then $T_{\text{SIRF}} = (M + L_S + L_I + T_{\text{res}})T_{\text{clk}}$, where T_{clk} is the system clock period.

As can be seen from the timing diagram, the resampling step is a bottleneck in the SIRF execution as it cannot be pipelined with other operations. Thus, T_{res} significantly affects the cycle time T_{SIRF}. Hence, developement of faster and more efficient resampling algorithms is vital to the implementation of real-time particle filters in high-speed applications. The architectures and their modifications that have been presented in this paper help to bring down T_{res} in different ways and hence reduce the effect of the resampling bottleneck. A resampling scheme should be chosen such that its time T_{res} satisfies the required T_{SIRF} for the application at hand. The modified versions of the two resampling schemes partially parallelize resampling and reduce execution time at the cost of added hardware. The times T_{res} and T_{SIRF} needed for resampling and one SIRF recursion respectively in terms of cycles are summarized in Table 2. k is the number of loops into which resampling is split as described in Sections 3.3 and 3.6 for modified schemes.

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

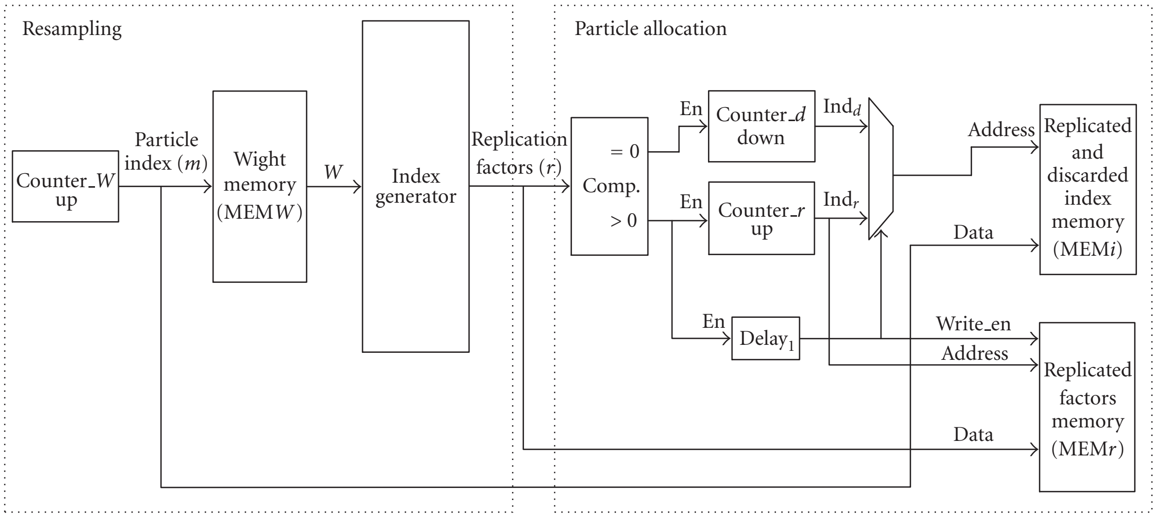


Figure 10: The architecture for the RSR algorithm combined with the particle propagation.

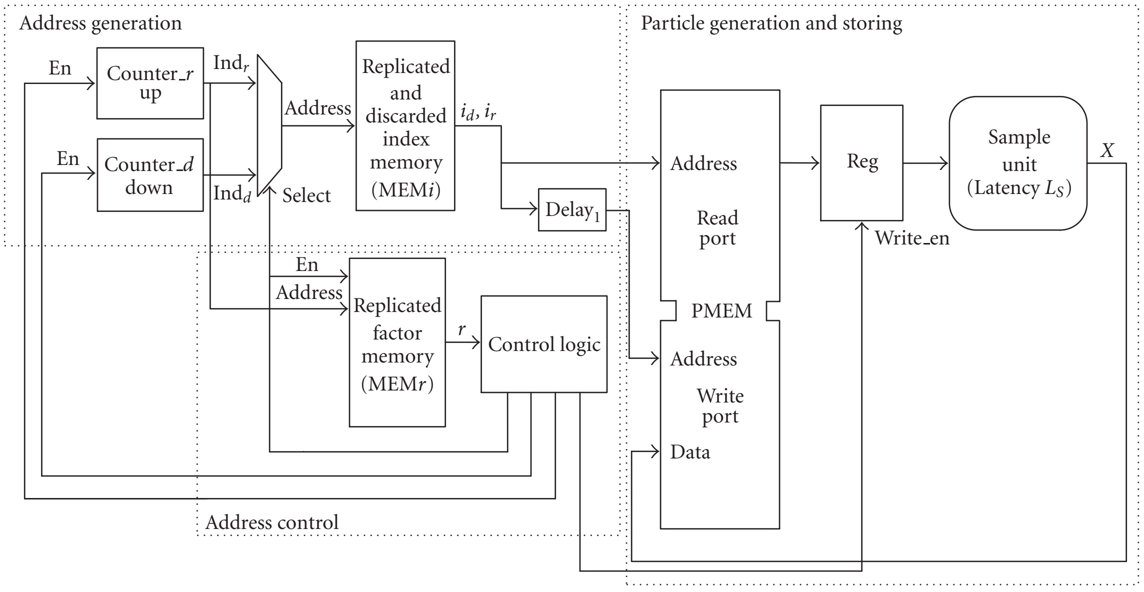


Figure 11: The architecture for memory-related operations of the sampling step.

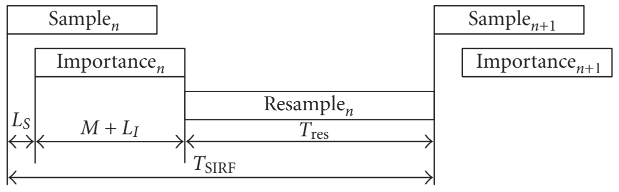


Figure 12: Timing of operations in SIRF.

4.2. *Resource utilization*

The architectures presented in the paper include all the memory-related operations of the generic SIRF. We use a Virtex 2 pro FPGA platform for evaluation [16]. All memory modules are mapped to the 18 kb block RAMs available on the chip. The memory required by SIRF for processing a certain number of particles depends upon the dimension of the state and the number of bits used in the fixed point representation of the state, weights (or CSW), and indexes. The number of 18 kb blocks needed on the Virtex 2 pro device

Table 2: Timing of SIRF using the different proposed architectures.

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

Table 3: Resource utilization for the two schemes on the XC2VP50-ff1152 device.

Resource	Scheme 1 (implemented)	Scheme 2 (implemented)	Modified scheme 1 (estimated)	Modified scheme 2 (estimated)
Slices	199	294	$k\cdot 199$	$k\cdot 294$
Slice registers	130	224	$k\cdot 130$	$k\cdot 224$
4 input LUTs	232	348	$k\cdot 232$	$k\cdot 348$
Block RAMS	15	14	$11+\left\lceil\frac{k}{2}\right\rceil$	$10+\left\lceil\frac{k}{2}\right\rceil$
Block multipliers	0	1	0	k

Table 4: Comparison of memory requirement and cycle time with a straightforward implementation.

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

for storing M particles (also weights, CSWs, or indexes), B_M, is given by

$$B_M = N_s\left\lceil\frac{M\cdot b}{18\times 1024}\right\rceil, \tag{9}$$

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

Table 3 summarizes the total utilization of the proposed architectures. The model we have chosen for our evaluation is the $N_s = 4$ dimensional bearings-only tracking (BOT) problem [6]. The fixed point widths in the design of this prototype were chosen using a methodology similar to that in [17]. Here a statistical analysis of all the variables in the algorithm over several runs is performed. The first 4 moments of the value that a particular variable takes are used to determine a fixed point format for representing that variable in hardware within a tolerable error of 10% to 15% for different variables. This analysis will need to be done for every model before the SIRF is applied to it. The chosen bit widths will not only affect the memory utilization as in (9), but will also increase latencies L_S and L_I. However, these latencies are small as compared to M and hence T_{SIRF} will not be significantly affected by bit widths. For the BOT model, we have used an 18-bit representation for the $M = 2048$ particles and the weights and 25-bit representation for the CSW. The indexes and replication factors are whole numbers and are represented using $\log_2 M = 11$-bits. The indexes and weights are obviously one-dimensional and the particles for the BOT problem are 4-dimensional. Accordingly, the number of memory blocks needed for each of these can be calculated from (9). The total memory requirement of the two schemes *for the mentioned bit widths* is shown in Table 3. Scheme 1 requires more memory than scheme 2 since it needs to store the CSW which has a wider fixed point representation. The amount of block RAMs available on a particular FPGA will determine the number of particles that an SIRF realization on that device can process. Thus, (9) can be used as a guideline for selecting a device for a particular implementation. The mathematical units in scheme 2 like the adders and multiplier were chosen to be 18-bits wide in input and output. The table also gives an estimate of the resources needed for the modified implementations of the two schemes with resampling being split into k parallel loops.

Finally, Table 4 shows the comparison of cycle time and memory requirement (in terms of words) for the proposed schemes with a straightforward implementation starting from the traditional algorithm. This approach requires the sampled and resampled particles to be stored in different memories. Also if the index-addressing schemes presented here are not used, another M cycles are added to the SIRF execution time since resampled particles first need to be read from one memory and written to another before the sample step can begin.

Table 5: Resource utilization for entire SIRF for the bearings-only tracking problem using scheme 1.

Resource	Random no. generation	Sample	Importance computation	Resample	Top level logic	Total
Slices	300	411	1535	199	623	3068 (13%)
Slice registers	364	568	2578	130	752	4392 (10%)
4 input LUTs	196	404	2674	232	342	3848 (8%)
Block RAMs	2	8	1	7	0	18 (8%)
Block multipliers	6	0	3	0	4	13 (6%)

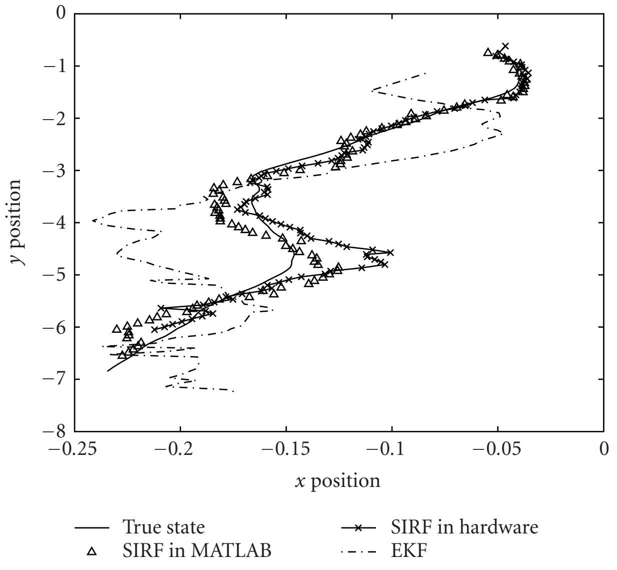


Figure 13: Results of tracking for the BOT problem.

4.3. Example application

The entire SIRF along with the computational units of sampling and importance for the above-mentioned bearings-only tracking problem was implemented on a Xilinx Virtex II pro device (XC2VP50FF1152). This FPGA prototype used the architecture described in scheme 1 with a resampling time $T_{\text{res}} = 2M - 1$ cycles as explained earlier. The state in this model is 4-dimensional and incorporates the position coordinates and velocities in x and y directions, respectively. The input to the filter is the time varying angle (bearing) of the moving object with respect to a fixed measurement point. This input is sampled by the filter at rate $1/T_{\text{SIRF}}$. Each input sample is processed by the SIRF to produce an estimate of the unknown state at that sampling instant.

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

$$T_{\text{SIRF}} = \left[(2048 + 8 + 53) + ((2 \times 2048) - 1)\right] T_{\text{clk}}. \qquad (10)$$

Thus one recursion of the SIRF needs 6024 cycles. The designed hardware can support clock frequencies of upto 118 MHz. Using a clock frequency of 100 MHz, we get the speed at which new samples can be processed $1/T_{\text{SIRF}} =$ 16 kHz. Table 5 shows the resource utilization of the entire SIRF for the BOT problem. Random number generation is needed for the sample step in accordance with the state space model. We have used a quantized version of the Box-Muller method for generating the random numbers. The importance step in the model involves exponential and arctangent operations and hence has a high resource requirement. These are implemented using 2 pipelined CORDIC units. The results of the tracking are shown in Figure 13. The position estimates are obtained from a post place and route simulation of the SIRF. The outputs are 32-bits wide and are in fixed point format. Their values are interpreted using SystemC fixed point data types and plotted using MATLAB. The figure also shows the tracking results obtained by the SIRF run in MATLAB using floating point representation for all variables. These results are also compared with tracking results obtained with the traditional EKF which for this model has execution speed of 10 kHz on a DSP platform (TMS320C54x).

This performance figure of 16 kHz is for 2048 particles. In practice a much larger number of particles is needed for tracking in noisy environments. This makes the SIRF computationally very intensive, and real-time processing using any software platform or DSP is not possible even for low sample rates. The FPGA hardware SIRF on the other hand can process input samples at rates of upto 3.5 kHz even with 10 000 particles using the basic scheme 1. Large number of particles will lead to increased memory requirement. But a large FPGA like the one chosen for our evaluation will support a high number of particles. Thus, the hardware realization of the SIRF not only allows for increased sample rates but also enables real-time processing even with a very large number of particles.

By using the other schemes introduced in the paper, the SIRF can be made even faster. Currently, work is being done on the development of parallel architectures for the SIRF which perform sampling and importance computation for groups of particles simultaneously. The goal is to use these architectures along with the high-speed resampling schemes presented here to increase the speed of the SIRF to the MHz range. This will enable the application of SIRF to wireless communications applications [5].

5. CONCLUSION

In this paper, we have presented a generic architectural framework for the hardware realization of SIRFs applied to any model. The architectures reduce the memory requirement of the filter in hardware and make efficient use of the

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

        Loop 1
        Initialize $\mathrm{ind}_r^1 = 0$, $\mathrm{ind}_d^1 = M/2 - 1$
        for $m = 0$ to $M/2 - 1$
            Perform steps 5–12 of Pseudocode 3
        end

        Loop 2
        Initialize $\mathrm{ind}_r^2 = M/2$, $\mathrm{ind}_d^2 = M - 1$
        for $m = M/2$ to $M - 1$
            Perform steps 5–12 of Pseudocode 3
        end
```

Pseudocode 5: Modified implementation of the RSR algorithm.

dual-port memories available on an FPGA platform. Two architectural schemes, scheme 1 and scheme 2 were proposed based on the SR and RSR algorithms, respectively. The resampling process cannot be pipelined with other operations and is a bottleneck in the filter execution. Hence for high-speed applications, the high latency of SR in scheme 1 is unacceptable. Scheme 2 uses the faster but more complicated RSR algorithm which allows for lower SIRF cycle times. We also introduced modifications of the two schemes involving parallelization of the resampling process by splitting it up into multiple concurrent loops. This allows for reducing the resampling latency and thus the SIRF cycle time at the cost of added hardware. The tradeoff between speed and hardware cost will dictate the choice of architecture for an SIRF realization. The results of this work have been used to develop the first FPGA prototype of the SIRF (using scheme 1 in this paper) for the bearings-only tracking problem in [6]. For 2048 particles, this prototype can process input observations at 16 kHz which is about 32 times faster than speed achieved for the same problem with the same number of particles on a state-of-the-art DSP.

ACKNOWLEDGMENT

This work has been supported by the NSF under Awards CCR-9903120 and CCR-0220011.

REFERENCES

[1] A. Doucet, S. Godsill, and C. Andrieu, "On sequential Monte Carlo sampling methods for Bayesian filtering," *Statistics and Computing*, vol. 10, no. 3, pp. 197–208, 2000.
[2] A. Doucet, N. de Freitas, and N. Gordon, Eds., *Sequential Monte Carlo Methods in Practice*, Springer, New York, NY, USA, 2001.
[3] P. J. Harrison and C. F. Stevens, "Bayesian forecasting," *Journal of the Royal Statistic Society Series B*, vol. 38, pp. 205–247, 1976.
[4] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking," *IEEE Trans. Signal Processing*, vol. 50, no. 2, pp. 174–188, 2002.
[5] P. M. Djurić, J. Kotecha, J. Zhang, et al., "Particle filtering," *IEEE Signal Processing Mag.*, vol. 20, no. 5, pp. 19–38, 2003.
[6] N. J. Gordon, D. J. Salmond, and A. F. M. Smith, "Novel approach to nonlinear/non-Gaussian Bayesian state estimation," *IEE Proceedings F, Radar and Signal Processing*, vol. 140, no. 2, pp. 107–113, 1993.
[7] *TMS320C54x DSP Library Programmers Reference*, Texas Instruments, August 2002.
[8] F. Daum and J. Huang, "Curse of dimensionality and particle filters," in *Proc. 5th ONR/GTRI Workshop on Target Tracking and Sensor Fusion*, Newport, RI, USA, June 2002.
[9] J. M. Rabaey, *Digital Integrated Circuits: A Design Perspective*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1996.
[10] M. Bolić, A. Athalye, P. M. Djurić, and S. Hong, "Algorithmic modification of particle filters for hardware implementation," in *Proc. 12th European Signal Processing Conference (EUSIPCO '04)*, Vienna, Austria, September 2004.
[11] M. Bolić, P. M. Djurić, and S. Hong, "Resampling algorithms for particle filters: A computational complexity prespective," *EURASIP Journal of Applied Signal Processing*, vol. 2004, no. 15, pp. 2267–2277, 2004.
[12] J. S. Liu and R. Chen, "Sequential Monte Carlo methods for dynamic systems," *Journal of the American Statistical Association*, vol. 93, no. 443, pp. 1032–1044, 1998.
[13] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, Menlo Park, Calif, USA, 3rd edition, 2002.
[14] *Understanding Synchronous and Asynchronous Dual Port RAMs*, Cypress Semiconductor Corporation, July 2001, Application Note, available from "www.cypress.com".
[15] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*, John Wiley & Sons, New York, NY, USA, 1999.
[16] *Virtex – II Pro^{TM} Platform FPGA User Guide*, v2.6 ed. , Xilinx, San Jose, Calif, USA, April 2004.
[17] S. Kim, K.-I. Kum, and W. Sung, "Fixed-point optimization utility for C and C++ based digital signal processing programs," *IEEE Trans. Circuits Syst. II*, vol. 45, no. 11, pp. 1455–1464, 1998.

Akshay Athalye received his B.S. degree in electrical engineering from the University of Pune, India, in May 2001. Since then he has been pursuing the Ph.D. degree in the Department of Electrical and Computer Engineering at the Stony Brook University, NY, USA. His primary research interests lie in the development of dedicated hardware for intensive signal processing applications. His work encompasses algorithmic modifications, architecture development, and implementation and use of reconfigurable SoC design for real-time signal processing applications. His secondary research interests lie in design and implementation of algorithms for efficient signal processing in RFID systems. He has served as an External Reviewer for various journals and conferences affiliated to the IEEE and EURASIP. He is a Student Member of the IEEE.

Miodrag Bolić received the B.S. and M.S. degrees in electrical engineering from the University of Belgrade, Yugoslavia, in 1996 and 2001, respectively, and his Ph.D. degree in electrical engineering from Stony Brook University, NY, USA. He is currently with the School of Information Technology and Engineering at the University of Ottawa, Canada. From 1996 to 2000 he was a Research Associate with the Institute of Nuclear Sciences, Vinĉa, Belgrade. 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.

Sangjin Hong received the 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 at the State University of New York, Stony Brook. Before joining SUNY, he has worked at Ford Aerospace Corp. Computer Systems Division as a systems engineer. He also worked at Samsung Electronics in Korea as a Technical Consultant. His current research interests are in the areas of low-power VLSI design of multimedia wireless communications and digital signal processing systems, reconfigurable SoC design and optimization, VLSI signal processing, and low-complexity digital circuits. He served on numerous technical program committees for IEEE conferences. He is a Senior Member of the 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 his Ph.D. degree in electrical engineering from the University of Rhode Island, in 1990. From 1981 to 1986 he was a Research Associate with the Institute of Nuclear Sciences, Vinĉa, Belgrade. Since 1990, he has been with 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. He 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 a member of editorial boards of several professional journals.