EFFICIENT HARDWARE ARCHITECTURE FOR PARTICLE FILTER BASED OBJECT TRACKING

Howida A. Abd El-Halym  
Nuclear Research Center,  
Atomic Energy Authority  
howidaaa@yahoo.com

Imbaby I. Mahmoud  
Nuclear Research Center,  
Atomic Energy Authority  
imbabvisma@yahoo.com

S. E.-D. Habib  
Faculty of Engineering, Cairo University  
seraged@ieee.org

1. INTRODUCTION

Particle Filters, as a powerful methodology with a wide range of applications, provide a general framework for estimation of nonlinear/ non-Gaussian dynamic systems. They are based on the idea of approximating the probability density functions of the state of a dynamic model by random samples (particles) and propagating the particles based on the probabilistic model of the state and the measurements. Particle filtering has been applied to several problems of tracking, navigation, detection [1-5]. A major limitation of using the particle filters in practice is related to the high computational load required to obtain estimation. As the particle filter (PF) is a Monte Carlo Simulation based estimation framework, the computations tends to be very slow. There are numerous software implementations of particle filtering scheme to various applications. Implementation of the particle filter applications on hardware or hardware/ software co-design platforms is further challenging due to the resources constraints on such platforms.

Regarding the parallel computation of the particle filter, many works have shown that the particle filter is immediately parallelizable since there are no data dependencies among particles. This is the case for most of the particle filter steps, except for the resampling step. Most of the published researches focus on the designing a resampling step suitable for parallel implementation [6, 7].

Atheyle et al. [8] presented generic architectures for the implementation of the PF namely, the Sampling Importance Resampling Filter (SIRF) to improve the resampling step. They proposed two architectures each based on different resampling mechanism and modified these architectures by splitting it up into multiple concurrent loops. Bolic et al. [9] showed that a particle filter with M particles could be computed on a SIMD machine with K processing units in \( (M/K + TL) \) steps, where TL is the latency for the first particle to be available. They presented different resampling methods and proposed architectures for effective implementation for these methods.

Cho et al. [10, 11] proposed a modified architecture for a circuit implementation of the particle filter. They sample the new position of the particles according to the resample particles and Gaussian white noises from the particle selector and Gaussian distribution Lookup Table. The particle selector selects the new particles among the current particles according to the uniform random distribution obtained from the random distribution Lookup Table and the cumulative function from the cumulative Lookup Table. Sankaranarayanan et al. [7] proposed two hardware architectures (sequential and parallel) for the particle filter algorithm and analyzed its latency and resource requirements. They replicated the sequential architecture, which works independently, several times to obtain parallel particle filter.

Medeiros et al. [12] worked on the parallel computation of the particle weights, as a major bottleneck in standard implementations of the color-based particle filter. They implemented the particle weights computations step in parallel using SIMD processor.

One of the main objectives of most of the published work in PF hardware implementation is to parallelize the resampling step, or simplify it via the use of Lookup Tables. It may be noted, however, that for a moderate number of particles, resampling itself is not computationally expensive [12, 13]. 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.

The proposed architecture performs the particle filter computations in two steps. In the first step, the particle sampling, weighting and output calculation are computed concurrently. Particle resampling is carried out in the second step. During this second steps, resampling for all particles is massively parallelized. At the same time, it uses less memory resources and makes efficient use of the dual-port FIFO available on an FPGA platform. The architecture is captured using VHDL and the design is synthesized using Xilinx environment and verified using Modelsim simulator.

2. PARTICLE FILTER OVERVIEW

The particle filter techniques are formulated on the concepts of the Bayesian theory and the sequential importance sampling which has
been found to be very effective in dealing with non-Gaussian and non-linear problems [14].

The particle filter approximates recursively the posterior distribution 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 particle filter 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 $t-1, t-2, ..., t_2, t_1$ are denoted as $X_{t-1}, X_{t-2}, ..., X_2$ and $X_t$. $p(X_t|X_{t-1})$ describes the transition for state vector $X_t$ (dynamic or motion model). Let all available observations be $Z_{t-1}, ..., Z_1$. The prediction uses the probabilistic system transition model to predict the posterior probability distribution at time $t$ as equation 1:

$$p(X_t|Z_{t-1}) = \int p(X_t|X_{t-1}) p(X_{t-1}|Z_{t-1}) \, dX_{t-1} \quad \text{equation 1}$$

Let $p(Z_t|X_t)$ be the observation model. At time $t$, the observation $Z_t$ is available, and hence the state can be updated using Bayes’s rule:

$$p(X_t|Z_t) = \frac{p(Z_t|X_t) p(X_t|Z_{t-1})}{p(Z_t|Z_{t-1})} \quad \text{equation 2}$$

Thus, a particle filter is an importance sampler for this posterior distribution. The weight calculation and accumulation engine, and an output calculation engine perform the particle generation and the output are carried out as shown in Figure 1.

3. HARDWARE IMPLEMENTATION OF THE PARTICLE FILTER

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.

3.1 The Proposed Architecture

The proposed architecture is shown in Figure 2. This architecture is composed of a FIFO to store particles, a resampling engine, a weight calculation and accumulation engine, and an output calculation engine. The FIFO is a 2MxP dual port FIFO to store the particles. $P$ is a five component state vector, given by $(x, y, vx, vy, Y)$. $L$ denote the $X$-position, $Y$-position, the velocity in the $X$ and $Y$ directions and the gray level intensity respectively. The resampling engine has two register files in addition to a massively parallel resample logic. These two registers are $M \times N$ register file for storing the replication factor for each particle (where $M=2^N$), and $M \times W$ register file for storing the particle weights where $W$ represents the length of weight bits. $M$ is chosen to equal 64 to cover a search area of 16x16 pixels for an object-tracking example.

The operations of the particle filter are carried out as follow:

1. The particles are generated and stored in the FIFO using the Random generator [15] with the known initial state vector of the object. As the particle is generated, it can be used for weight calculations.

2. During the process of weight calculation, the accumulation of the total sum of weights is done.

3. As the generation of the total number of particles finishes, the resample process starts and gives a replication factor for each particle according to its weight using the simplified resampling procedure described in sec 3.2.

4. At that time, the output calculation process starts and the FIFO contains $M$ particles.

- Read one particle from the FIFO to Particle R register and the corresponding replication factor to NR register; continue reading until NR is found to be greater than zero. At that time, the particle is translated to Particle-S register and the content of register NR is translated to register NS. NS is decremented at each clock cycle.

- 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 $Y$ position of the tracked object).

- During decrementing, the content of register NS to zero the FIFO reading does not stop until found NR value greater than zero.

5. As the M particles are read and repeated according to their replication factors, the resampling process calculates in parallel the replication factors for the all particles for the new instant.

6. Finally, again, the FIFO contains $M$ particles and so the replication factor memory contains $M$ corresponding replication 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 choose a simple parallel implementation for the resampling process described in the following section.

3.2 Residual Systematic Resampling Algorithm

The low speed of the systematic resampling algorithm SR may not be suitable in the case of the high-speed applications such as tracking. Therefore, we used a simplified RSR based algorithm that has a single loop of $M$ iterations, which is twice faster than SR algorithm in terms of the number of cycles. We used the modifications of Bolic et al. [8, 13], mainly the non-normalized weights, and the splitting of resampling into loops. The non-normalized weights are used to reduce the $M$ divisions in hardware. Instead, the modification uses one division to divide $M$ by the total sum of weights. 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 RSR for $M/L$ particles where $L$ is the number of loops [8]. This mechanism reduces the execution time of the resampling to $M/L$ cycles at the cost of adding more hardware. Exploiting recent FPGA devices for needs of speed the resampling step, we selected...
\( L = M \). This is feasible since our number of particles is within one hundred particles. Table 1 shows the resource utilization for hardware particle filter resampling in the case of single, 8, and 64 parallel resampling loops to calculate the replication factors at the same time. The implemented RSR resampling eliminates the use of memory for storing the normalized weights that are not needed to be calculated or even stored. In most cases, the sum of all replication factors is less than \( M \). The remaining particles are obtained using other mechanism, in which we added this remainder number to the replication factor of the highest weight particle.

Table 1 Comparison between the different resources in the case of sequential and parallel resampling for \( xc5vlx50t-3-ff1136 \)

<table>
<thead>
<tr>
<th>Resources</th>
<th>( L = 1, M = 8 )</th>
<th>( L = M = 8 )</th>
<th>( L = M = 64 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice Register</td>
<td>11</td>
<td>11</td>
<td>131</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>77</td>
<td>78</td>
<td>1692</td>
</tr>
<tr>
<td>No. of Slices used as logic</td>
<td>77</td>
<td>78</td>
<td>1692</td>
</tr>
<tr>
<td>No. of DSP48Es</td>
<td>1</td>
<td>8</td>
<td>48</td>
</tr>
</tbody>
</table>

3.3 Weight calculations

To get robust performance, we calculate the weight by using the likelihood presented in [16].

\[
LP(Z_{t}^{(i,j)}) = \exp\left(-\frac{(x - i)^2 + (y - j)^2}{2\sigma_p^2}\right) \quad \text{(6)}
\]

\[
LI(Z_{t}^{(i,j)}) = \exp\left(-\frac{(Z_{t}^{(i,j)} - Z_{t-1}^{(i,j)})^2}{2\sigma_t^2}\right) \quad \text{(7)}
\]

Where \((i,j)\) is the particle position and \((x, y)\) is the previous object position. \(\sigma_p\) is the variance of the position. \(Z_{t}^{(i,j)}\) is the measured gray level intensity value at position \((i, j)\) at time \(t\). \(\sigma_t\) is the variance of the gray level intensity. The overall likelihood is taken as \(L = LP \times LI\)

Taking into consideration the complexity of the hardware implementation of the exponential function, we can simplify the \(LP\) and \(LI\) likelihood calculations by the piecewise linear approximations shown in Figure 3. The piecewise linear function for the position likelihood is chosen to be 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 allow the tracker to be tolerant against the small intensity variations. Our MATLAB simulations indicate that this approximation does not affect the performance of the tracker.

4. EVALUATION

The architecture is captured using VHDL and synthesized on a Xilinx Virtex 5 FPGA chip. The design was verified using Modelsim simulation. Table 2 summarizes the total utilization of the proposed architecture.

4.1 Execution time

The timing diagram of the SIRF operations is shown in Figure 4. The \( TL_g \) is the particle generation latency and it equals to two clock cycles. After the first instant, the total cycle time required is \( TSIRF = (aM + TL_W + TL_{res}) \times T_{clk} \), Where \( TL_W \) is the start up latency of the weight calculation engine. \( TL_{res} \) equals one clock cycle for our implemented simplified piecewise linear functions. Therefore, the total latencies cycles are three. \( TL_{res} \) is the resampling time that for \( L = M \) loops (into which resampling is split) equals one. The value of \( a \) lies in between 1 and 2, and depends on the replication factors of the particles. The worst case takes place if \( M-I \) particles have zero replication factors and the only last stored particle in the \( FIFO \) has a replication factor that equals \( M \). At this time, the \( FIFO \) needs to read \( M-I \) particles first, until \( FIFO \) reaches the particle.
number $M$. Then the sampling, weighting, and the output calculations are started and repeated for those particle $M$ times. For all other cases (when some of the particles have non-zero replication factors) the value of $\nu$ is close to one. For hundreds 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 32x32 pixels. So, the number of particles, needed to represent this state space in this area, is of moderate value. That is why it is important to consider the latencies cycles. For example, in the implementation of Athalye et al. [8], the latency of sampling and importance computation units are 8 cycles and 53 cycles respectively, giving a total value for $TL = 61$ cycles. If $M=10000$ (as in case of Ref. [8]) 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 the proposed implementation, $TL$ is reduced to 3 cycles, leading to a latency time overhead of 3/64, less than 5%.

4.2 Resource utilization

The proposed architecture uses FIFO buffers. This implementation saves the logic needed to address memories and has a speed advantage over a straight memory implementation. Moreover, using 2M FIFO ensures that the process of updating the particles does not disturb the previous particles in its previous stored locations. The total memory requirements for this architecture are one dual port FIFO of 2M words to store the particles.

Table 2 The resources utilization for the proposed architecture on xc5vx50t-3-f1136

<table>
<thead>
<tr>
<th>Resource</th>
<th>The proposed architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Slice Register</td>
<td>934 out of 28800 3%</td>
</tr>
<tr>
<td>Number of Slice LUTs</td>
<td>4380 out of 28800 16%</td>
</tr>
<tr>
<td>Number used as logic</td>
<td>4352 out of 28800 16%</td>
</tr>
<tr>
<td>Number used as Memory</td>
<td>28 out of 7680 0%</td>
</tr>
<tr>
<td>No. of DSP48Exs</td>
<td>48 out of 48 100%</td>
</tr>
</tbody>
</table>

5. CONCLUSION

In the presented work, we have presented novel hardware architecture for the SIRF particle filter. This architecture carries out the particle generation, weight calculation and output calculation concurrently. Particle resampling is subsequently carried out in a massively parallel form. The memory addressing is eliminated which significantly speeds up the process. Resource saving is also attributed to the use of piecewise linear approximation rather than classical exponential weight function. The proposed architecture enables significantly smaller hardware implementation as is evidenced by our FPGA design case.

6. REFERENCES


Figure 3 simplified function for the weight calculations. (a)Simplified function for position likelihood, (b) simplified function for Intensity likelihood.

Figure 4 The timing diagram of SIRF operations

[aM + TLw] 1
Calculation 1
Particle Generation 1

[aM + TLw] 2
Calculation 2
Resampling 1
Particle Generation 2

[aM + TLw] 3
Calculation 3
Resampling 2

Output

[aM + TLw] 4
Calculation 4
Output

Particle Generation 3


