Journal of Signal Processing Systems
https://doi.org/10.1007/s11265-019-01489-y

Improved Particle Filter Resampling Architectures

Syed Asad Alam[1,2] · **Oscar Gustafsson**[1]

Received: 8 June 2018 / Revised: 18 June 2019 / Accepted: 12 September 2019

Abstract

The most challenging aspect of particle filtering hardware implementation is the resampling step. This is because of high latency as it can be only partially executed in parallel with the other steps of particle filtering and has no inherent parallelism inside it. To reduce the latency, an improved resampling architecture is proposed which involves pre-fetching from the weight memory in parallel to the fetching of a value from a random function generator along with architectures for realizing the pre-fetch technique. This enables a particle filter using M particles with otherwise streaming operation to get new inputs more often than $2M$ cycles as the previously best approach gives. Results show that a pre-fetch buffer of five values achieves the best area-latency reduction trade-off while on average achieving an 85% reduction in latency for the resampling step leading to a sample time reduction of more than 40%. We also propose a generic division-free architecture for the resampling steps. It also removes the need of explicitly ordering the random values for efficient multinomial resampling implementation. In addition, on-the-fly computation of the cumulative sum of weights is proposed which helps reduce the word length of the particle weight memory. FPGA implementation results show that the memory size is reduced by up to 50%.

Keywords Particle filter · Resampling architectures · Latency reduction · Hardware implementation · ASIC · FPGA · Pre-fetched resampling 22 minutes ago Create comment Create flashcard Jump to context Delete

1 Introduction

In problems that involve hidden Markov models (HMM) with unobserved states and cannot be evaluated analytically, filtering refers to estimation of that state from a set of observations that are corrupted by noise. In other words, it refers to determining the state probability distribution at time instance n, given observations up to n. The states evolve and capturing this evolution of state enables the prediction of the state in the future [3, 9, 11–14, 27]. It can be described by the following set of equations

$$\begin{aligned} x(n) &= f\left(x(n-1), z(n-1)\right) \\ y(n) &= g\left(x(n), v(n)\right), \end{aligned} \tag{1}$$

where $x(n)$ is the evolving state vector of interest, $y(n)$ is a vector of observations, $z(n)$ and $v(n)$ are independent noise vectors with known distributions, and $f(.)$ and $g(.)$ are known functions.

Particle filters track $x(n)$ by updating a random measure $\{x_k(n), w_k(n)\}_{k=1}^{M}$, which consists of M particles $x_k(n)$ and their weights $w_k(n)$ defined at time n, recursively. This random measure is used to approximate the posterior density of the unknown vector [3]. Particle filtering is used when the state transition and the observation models are non-linear and noise is non-Gaussian, in contrast to Kalman filters where they are linear and Gaussian, respectively [9]. Particle filters find application in a wide variety of complex problems including target tracking [15, 26, 32], computer vision [25], robotics [22], vehicle tracking [21, 28], acoustics [2] and channel estimation in digital communication channels [29] or any application involving large, sequentially evolving data-sets [11, 13].

There are three steps involved in particle filters, as illustrated in Fig. 1:

- Time-update (particle generation/sampling)
- Measurement-update (weight computation)
- Resampling

✉ Syed Asad Alam
syed.asad.alam@liu.se; syed.asad.alam@namal.edu.pk

Oscar Gustafsson
oscar.gustafsson@liu.se

[1] Department of Electrical Engineering, Linköping University, Linköping, Sweden

[2] Present address: Department of Electrical Engineering, Namal Institute, Mianwali-42250, Pakistan

Published online: 14 December 2019

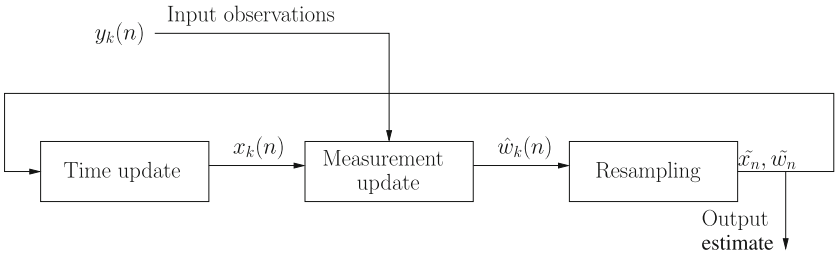

Figure 1 Different steps in particle filtering.

In the first step, samples (particles) of the unknown state are drawn or sampled from a density using the principle of importance sampling. This step can typically be considered as propagation of particles at time instant $n-1$ into n through (1). Thus, this step is also termed time-update. In the second step of particle filtering, particles $x_k(n)$ are assigned importance weights $w_k(n)$ based on the received observations. This is typically the most computationally intensive step and can also be referred to as measurement-update [12].

Among the three steps that accomplish particle filtering, resampling is the one which is hardest to obtain a high-speed implementation of since it cannot be straightforwardly parallelized. A number of contributions have focused on different resampling algorithms [4, 6–8, 16–20, 24, 27]. An overview of resampling algorithms can be found in [8, 17, 24] and is briefly discussed in Section 3. Algorithms which are focused on hardware implementation includes [8], where an algorithm combining systematic and residual resampling is proposed with the aim of providing a shorter and distribution independent latency. In [4], different architectures and memory schemes for the sampling and the resampling steps are discussed but there is no discussion and analysis of the word length requirement for the architecture and memories. A modified systematic resampling algorithm using non-normalized weights is proposed in [7]. Implementation aspects of different algorithms are discussed in [4, 8]. In particular, resampling algorithms applicable for distributed architectures are discussed which can enable a parallel execution of the resampling step with other steps of particle filtering [6]. Implementation of resampling step using general purpose computing techniques on a graphical processing unit (GPU) to obtain a parallel particle filter is reported in [16], while a compact resampling algorithm using only one threshold value and its associated hardware is presented in [19].

The current work looks into techniques for improving different aspects of the resampling step. The main contributions of this work are:

- Reduction in latency of the resampling step by pre-fetching weights in parallel to fetching of the next value from random function generator
- Associated hardware architectures for the above mentioned latency reduction technique
- Reduction in memory size of weights by doing cumulative sum computation on-the-fly
- A division-free architecture for multinomial, systematic, and stratified resampling, where the multinomial variant does not require explicit ordering of the values from the random function generator

Preliminary results concerning division free architecture and memory size reduction have been reported in [1].

The rest of the paper is organized as follows: in Sections 2 and 3, basic architectures for particle filters and the resampling process with different types of resampling algorithms are described, respectively. The main contributions of this work are presented in Section 4 and results showing the benefit and properties of the proposed approaches are presented in Section 5. Finally, in Section 6 some concluding remarks are given.

2 Architectures for Particle Filters

Particle filters consists of three main blocks; time-update, measurement-update and resampling. Resampling is considered a bottleneck because it cannot be executed in parallel with the other steps of particle filtering. A schedule depicting this situation over two iterations is shown in Fig. 2. Here, it is assumed that a new time-update computation can be started every clock cycle. After a latency of L_t cycles, the new particle is available and fed to the measurement-update. After yet L_m cycles the weight of the particle is available. After another M cycles all particles and particle weights are computed and the resampling can start. The resampling will output M particles and, as discussed in more detail later, will not output particles for a number of cycles, here denoted L_r, which is the latency of the resampling step. Hence, the total time for resampling is $L_r + M$. Here, it should be noted that although the resampling process is not finished, the next time-update step can start when there are M cycles remaining of the resampling.

It is common to implement the architecture using a memory that keeps track of which particles should be

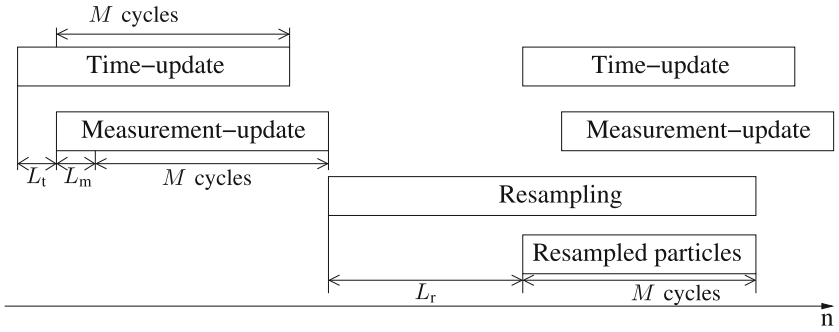

Figure 2 Basic scheduling of different steps in particle filter.

replicated and/or discarded, as shown in Fig. 3. In this way, a single particle memory can be used.

In case of the resampling unit and the time-update unit having separate memories to store particles, the time-update unit can read the resampled particle and write to its own memory, referred to as *particle allocation with index addressing* in [6]. However, it requires the resampling step to read from the time-update memory for the particle to be resampled. In this case, the time-update step, and as a consequence the measurement-update step, will not be executed in a continuous manner as shown in Fig. 2. Rather it will be distributed, as shown in, Fig. 4, where the resampling step finishes when the number of resampled particles is equal to the original number of particles. The latency of the resampling step, L_r, will be a sum of the cycles in which no particle is being replicated and hence will be distributed.

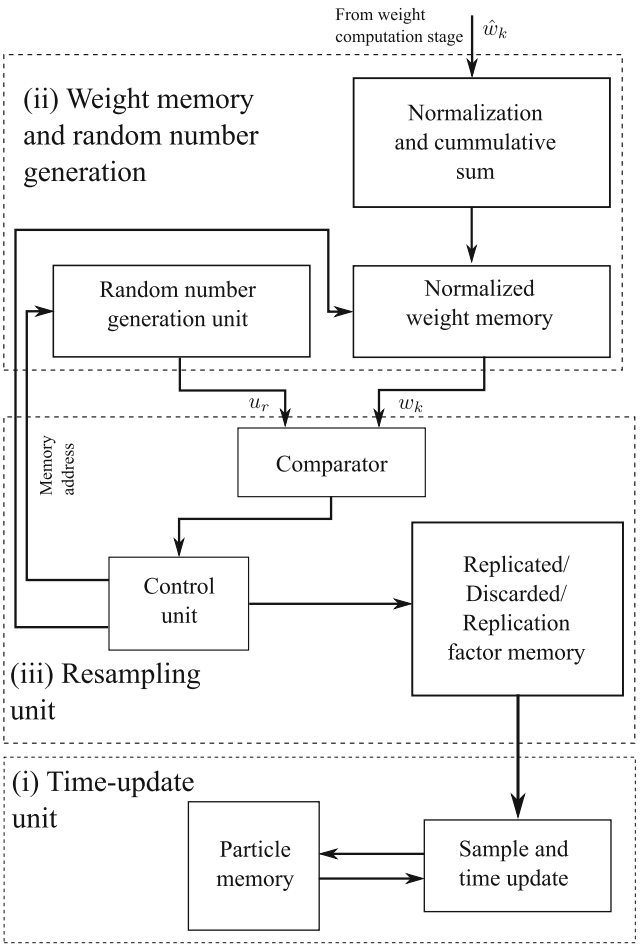

Figure 3 Basic architecture for the resampling step together with the time-update step of the particle filter.

If there is only one particle memory, the resampling unit will produce a set of indices of the resampled particles. However, this scheme requires a discarded indices memory as well so that the resampled particles can be written to the position of the discarded particles [4]. In such a scenario, the output of resampling cannot be immediately used until enough discarded indices have been produced to over write them. This is highly un-deterministic and in the worst case the resampling output can only be used after M cycles, after which the resampling output will be continuous and so will the execution of the subsequent steps, as earlier shown in Fig. 2.

Another alternative memory architecture is the use of alternating double buffers [6], also known as ping-pong memory [23], as shown in Fig. 5. In [6], this technique is shown to be applicable only to systematic resampling and requires an additional index memory of size $2M$. However, if the resampling output is allowed to be used in a distributed manner together with a distributed execution of the time-update and measurement-update step, as shown in Fig. 4, then there is no need of any additional index memory. This is also a natural consequence of most of standard resampling algorithms. As soon as it is known which is the next particle to resample, it is fed to the time-update unit and so on. The result is written back to the other memory. For the next iteration, the memories switch roles.

Even in the case of the resampling unit generating replication factors, the continuous or distributed execution will depend on the memory architecture. If the latency of the resampling step, L_r, can be reduced and together with the distributed execution, execution of each iteration can be started much earlier.

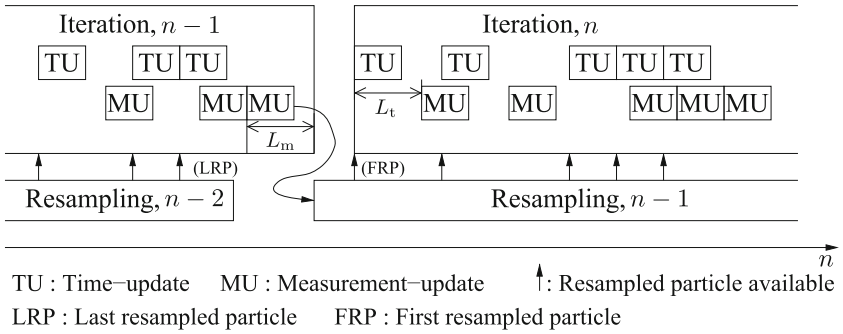

Figure 4 Distributed schedule of the particle filter.

3 Resampling in Particle Filters

Resampling prevents degeneration of particles, which improves the estimation of the state of the system by modifying the weighted approximate density $\pi(x_n)$ to an un-weighted density $\hat{\pi}(x_n)$. This is achieved by eliminating particles having low importance weights and replicating those having high importance weights. This will prevent poorer approximation on the posterior density and lead to better estimates. More formally

$$\pi(x_n) = \sum_{k=1}^{M} w_k(n)\delta(x - x_k(n)) \tag{2}$$

is replaced by

$$\hat{\pi}(x_n) = \sum_{i=1}^{M} \frac{1}{M}\delta(x - x_i(n)) = \sum_{k=1}^{M} \frac{c_k}{M}\delta(x - x_k(n)), \tag{3}$$

where c_k is the number of copies of the original particle $x_k(n)$ in the new set of particles $x_i(n)$ [17]. In these resampling schemes, the expected number of times the k^{th} particle is resampled, c_k, is proportional to its weight. For brevity, we assume that the number of particles is the same before and after resampling.

Different methods of resampling algorithms exist to generate $x_i(n)$ (3). However, all resampling algorithms use a normalized cumulative sum of weights which is given as:

$$w_k = \frac{W_k}{W_M}, \qquad k \in 1 \ldots M, \tag{4}$$

where $W_k = \sum_{i=1}^{k} \hat{w}_i$ and the time index is dropped for brevity. The cumulative sum can be calculated in parallel with the measurement-update step and will not add to the total execution time of particle filtering.

Different forms of distribution are used to generate random numbers to compare against normalized particle weights. In this paper, we are primarily interested in the algorithms that use various forms of random numbers from a single distribution to perform resampling [17, 24]:

- Multinomial
- Stratified
- Systematic
- Residual

Multinomial resampling is the basic approach which uses uniformly distributed random numbers to perform resampling. These numbers are generated according to [17]:

$$u_r = \tilde{u}_r, \text{ with } \tilde{u}_r \sim U[0, 1). \tag{5}$$

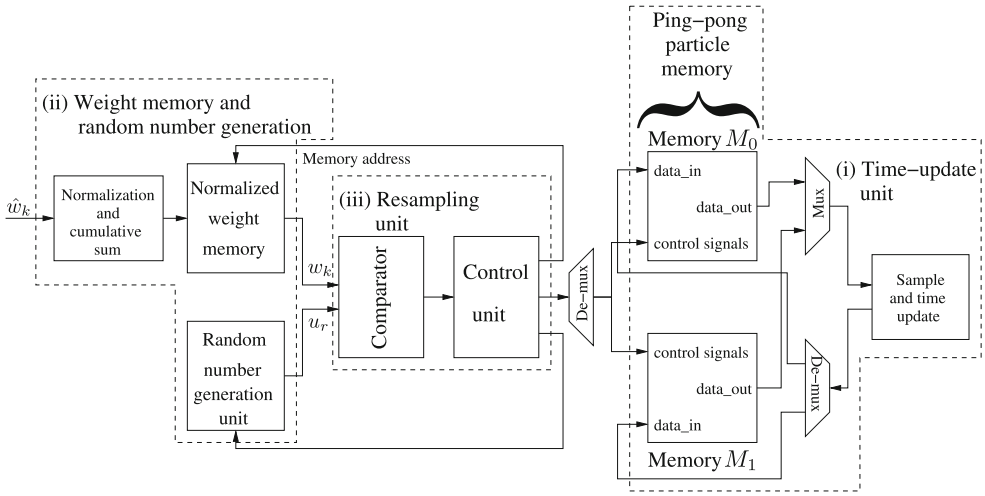

Figure 5 Architecture for the particle filter with ping-pong memory.

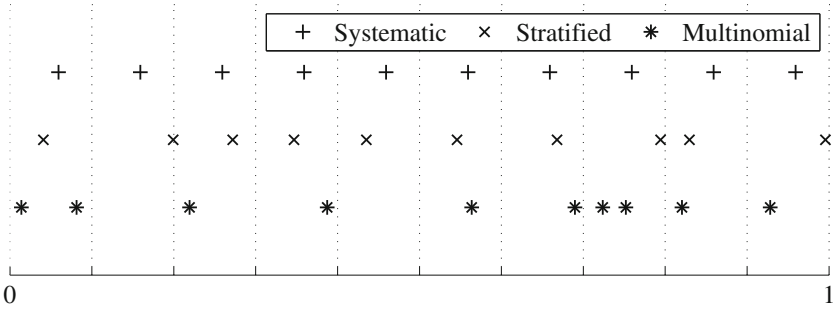

Figure 6 Standard uniformly distributed samples for $M = 10$.

The total time required for multinomial resampling is of the order of M^2 as for each iteration, up to M normalized weights must be compared against the random number [24]. However, if the random numbers are ordered, this is reduced to $2M$,[1] and, hence, the latency is $L_r = M$.

Stratified resampling divides the interval into uniform intervals, also termed as stratification, which helps the samples to be more uniformly distributed. One value is then picked from each interval with a random offset. This can be mathematically stated as [17]:

$$u_r = \frac{1}{M}(r-1) + \tilde{u}_r, \text{ with } \tilde{u}_r \sim U\left[0, \frac{1}{M}\right). \tag{6}$$

Systematic resampling is an extended view of stratified resampling where the offset is the same resulting in the following formulation for generation of random numbers [17]:

$$u_r = \frac{1}{M}(r-1) + \tilde{u}, \text{ with } \tilde{u} \sim U\left[0, \frac{1}{M}\right). \tag{7}$$

An example of the different random samples used by multinomial, stratified and systematic resampling methods are shown in Fig. 6.

As shown in Eq. 4, typically in the resampling process, the weights are normalized which results in M divisions per resampling process incurring high hardware cost. In [7], the authors proposed a modified systematic resampling scheme whereby M divisions are replaced by one division by drawing the uniform random number from a distribution spanning $\left[0, \frac{W_M}{M}\right)$, and then updating it by $\frac{W_M}{M}$, as given in Eq. 8. However, there is still one division, making it primarily attractive when M is a power of two, in which case the division is simplified to a binary shift.

$$u_r = \frac{W_M}{M}(r-1) + \tilde{u}, \text{ with } \tilde{u} \sim U\left[0, \frac{W_M}{M}\right). \tag{8}$$

The first three resampling methods mentioned above can be represented in a unified way using a flow chart, as shown in Fig. 7. In this figure, the random number generator can be based on any one of the three methods as long as the random numbers are ordered, which they automatically are for stratified and systematic resampling. In each iteration, the considered weight is compared against the current random value. If the weight is larger, then the corresponding particle is replicated and the next random value is fetched. If the weight is smaller, the next weight is fetched. It is the latter action that leads to the latency of M cycles, as the complete weight sequence must be traversed and no particle is output during that iteration.

Residual resampling calculates the replication factor for each particle in two steps [17]. In the first step, the number of replications of a particle is determined by the truncated product of M and w_k. Remaining particles are resampled in the second step using one of the earlier described resampling schemes. A modified version of this, called the residual systematic resampling (RSR), removes the second step by incorporating the systematic resampling principle to update the random number [8]. It uses the knowledge of the fixed intervals between two consecutive values generated in systematic resampling to calculate the replication factors which reduces the number of random numbers fetched. The comparison step shown in Fig. 7 is replaced by the

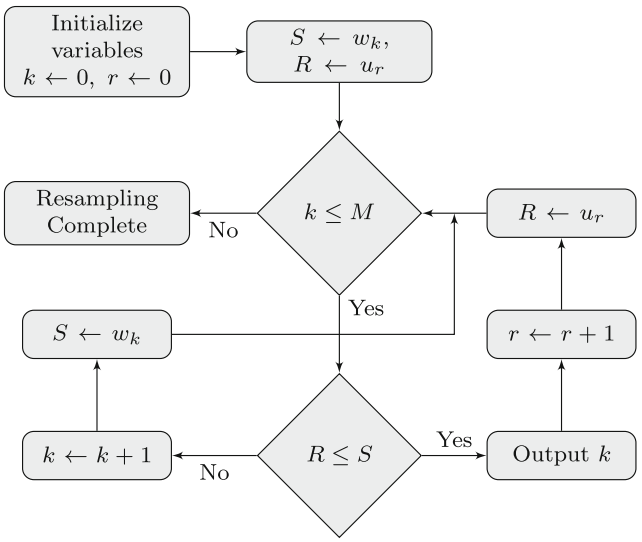

Figure 7 Flow chart of resampling.

[1] It is sometimes claimed to be $2M - 1$. However, the exact number of cycles depends on how they are counted and the exact implementation. For simplicity, we state $2M$.

replication factor generator which will help in reducing the number of iterations for the right sided loop in Fig. 7.

Based on the type of resampling algorithm, the random number generation unit will be modified. Multinomial resampling requires a random number generator and a memory to hold it during the resampling step while the control unit will then generate address for this memory. Stratified and systematic resampling only need a random number generator and an accumulator. For stratified, as given in Eq. 6, a new random number is produced every cycle while for systematic, in Eq. 7, one random number is produced at the start of the resampling process.

For the residual resampling algorithm the second step will require a random number generator and memory while for RSR, a similar random number generator is required as used by systematic resampling. However, for these two algorithms, the resampling unit will be different to multinomial, stratified, and systematic resampling algorithms.

4 Proposed Techniques

The main contributions in this work are techniques to improve two of the three units shown in Fig. 3:

1. Reduction in latency of the resampling unit, L_r as indicated in Fig. 2
2. A generic double multiplier division free architecture for the resampling unit and a compact memory structure for the weight and random number generation unit

The latency reduction technique is primarily applicable to the architecture shown in Fig. 5 but it can be easily adapted for the architecture shown in Fig. 3. The division free architecture and compact memory structure technique, however, is applicable to both architectures shown. Each of these technique is discussed in the proceeding sections.

4.1 Reduction in Resampling Latency – Pre-Fetch

In a typical resampling process, there is a comparison between the normalized cumulatively summed weight, w_k (4), of a particle and a random number (or a variation of it) as given by Eqs. 5–8. For simplicity, w_k will only be referred to as weight. The result of each comparison either makes the system fetch a new weight or a new random number as shown in the flow graph in Fig. 7. During the fetching of the random number, the weight of a particle is not fetched from the weight memory and as mentioned earlier these cycles contribute to the overall latency of the resampling.

This results in a variable total latency which can rise to at most M cycles and a total computation time of at most $2M$, where M is the number of particles. The latency and hence the total computation time can be reduced if the cycle during which the random number is fetched is used to fetch more weights and perform a parallel comparison between the random number and all the fetched weights. This is referred to here as *pre-fetch*. In other words, the number of iterations in the left sided loop of Fig. 7 is reduced by pre-fetching weights. This is unlike RSR, where the number of iteration in the right sided loop in reduced. In this way, the number of dedicated cycles required to fetch the weights will be significantly reduced.

In order to illustrate the pre-fetch technique, the case of pre-fetch by two is explained first. Here, at most only one extra weight is fetched leading to a three-way comparison as given by the following equations:

$$u_r \leq w_k \tag{9}$$

$$w_k < u_r \leq w_{k+1} \tag{10}$$

$$w_{k+1} < u_r, \tag{11}$$

where w_{k+1} is the pre-fetched weight, fetched as a result of the operations indicated by Eqs. 9 and 10. The flow of operations during the whole process is shown in Fig. 8, where the pre-fetch is shown corresponding to Eqs. 9 and 10.

In case of Eq. 11, a new weight will be fetched, indicating that a weight is always fetched from the weight memory unless the condition in Eq. 9 is satisfied and a new weight has already been pre-fetched. In case of Eq. 10, the pre-fetched weight (indicated by S_1 in the figure) is retained (copied to S_0) and a new weight pre-fetched (into S_1).

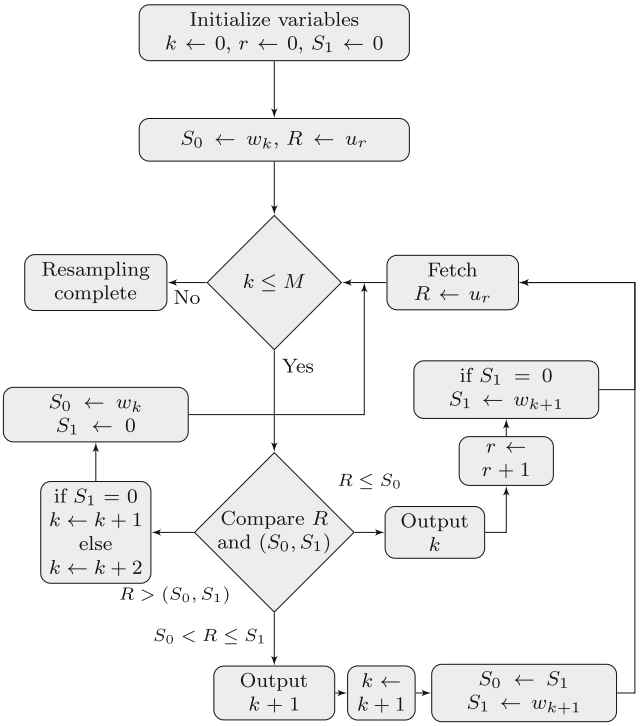

Figure 8 Flow chart of pre-fetch up to $D = 2$ weights from memory.

For a general pre-fetching scheme, weights need to be stored in a memory structure, depth of which will be equal to the maximum number of weights being pre-fetched. This number is indicated here as D. The memory structure will resemble the operation of a first in first out (FIFO) memory structure with the exception that multiple data can be removed from it. For expository convenience, this structure will be simply referred to as a buffer in the rest of the paper. For comparison with the value from the random number generator, one needs D parallel comparators. The relationship defining the operation can be generalized as given below and the flow chart describing the operations is given in Fig. 9, where $f \leq D$ is the number of filled buffer places and $S_0 \ldots S_{f-1}$ are the weights that fill up the buffer.

$$u_r \leq w_k \tag{12}$$

$$w_k < u_r \leq w_{k+1} \tag{13}$$

$$w_{k+1} < u_r \leq w_{k+2} \tag{14}$$

$$\vdots$$

$$w_{k+f-2} < u_r \leq w_{k+f-1} \tag{15}$$

$$w_{k+f-1} < u_r. \tag{16}$$

In case there is an $R \leq S_i$ where $i \in 0 \ldots f-1$, j is the smallest index such that $R \leq S_j$, all weights with index up to $j-1$ will be discarded and $k+j$ will be the output index. When all the weights in the buffer, $S_0 \ldots S_{f-1}$ are less than R, the buffer will be completely dequeued and a new weight fetched and enqueued in the buffer, as indicated by the left hand side loop in Fig. 9 and by Eq. 16. This is the cycle that contributes to the latency of the resampling step.

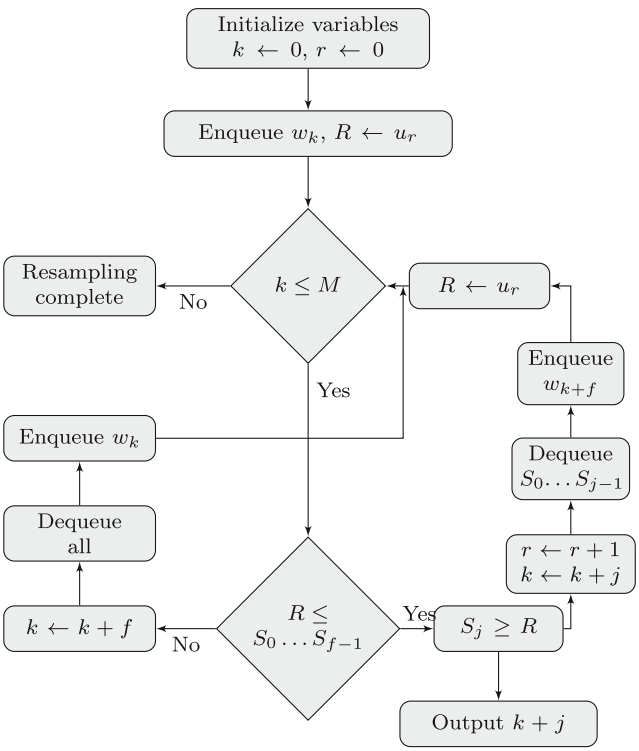

Figure 9 Flow chart of proposed pre-fetch resampling using a buffer to enqueue the pre-fetched values.

An example illustrating what happens to the buffer before and after the comparison for different cases is shown in Fig. 10 for $D = 5$. In Fig. 10a, R is less than the first value in the buffer, corresponding to Eq. 12 meaning S_0 will be resampled. A new weight, S_3, will be enqueued in the buffer, but no weights will be dequeued. In Fig. 10b with $f = 3$, since $R \leq S_2$, it means that $j = 2$ and $R > (S_0, S_1)$. S_0 and S_1 will be dequeued, S_2 will be moved to the top of buffer and S_3 will be enqueued. A similar operation occurs in Fig. 10c where $j = 1$ and $f = 5$ and only S_0 is dequeued while in d the whole buffer is dequeued, no particle is resampled and a new weight, S_5 is fetched to top of buffer, while f is set to one.

4.1.1 Hardware Architecture for Pre-Fetch Resampling

The generic architecture for the pre-fetch technique is shown in Fig. 11. The architecture primarily consists of a buffer with depth D which feeds a comparator array which compares each fetched w_k with u_r as given in Eqs. 12–16. Each comparator generates a logic 1 when $u_r \leq w_k$ and a logic 0 otherwise. This comparator array is connected to a $D{:}\lceil \log_2 D \rceil$ priority encoder which encodes the ones and zeros of the comparator array. The priority encoder is used because the comparator higher up the order have higher priority. In case of multiple ones, the priority needs to be resolved, hence the need of a priority encoder.

The buffer can either be implemented as a memory or as a network of multiplexers directly feeding the comparators. In the later case, one first needs a $1{:}D$ demultiplexer to direct the incoming fetched w_k to the correct comparator and move the already fetched weights to the appropriate comparators using multiplexers. The demultiplexer directs the incoming weight to one of its output and forces other outputs to zero. One technique will then be to use two layers of multiplexers with the first layer consisting of $S{:}1$ multiplexers with the number of inputs successively decreasing from D to two. The second layer will then have D, 2:1 multiplexers. This will produce a regular and scalar architecture.

An example of such an architecture is shown for pre-fetch by two in Fig. 12. Since this architecture scales with D, as D gets smaller, the control logic, size of multiplexers, the de-multiplexer and the priority encoder gets smaller as well. For a pre-fetch by two, the architecture gets significantly simplified. There is no need of a priority encoder and all multiplexers are reduced to having just two inputs as well as the de-multiplexer.

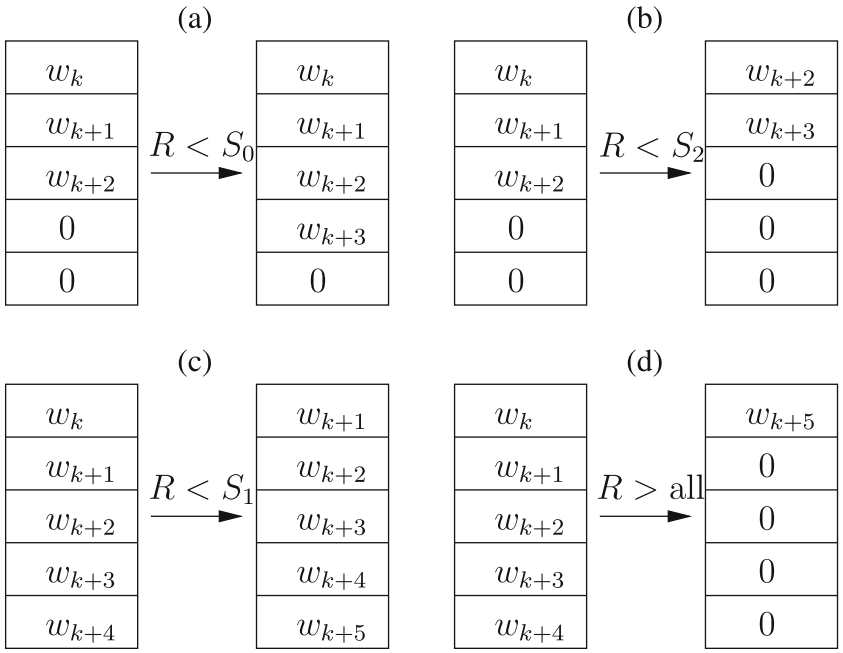

Figure 10 Different cases when pre-fetching for $D = 5$.

4.2 Generalized, Division-Free Resampling Architecture

To perform resampling, the normalized weight w_k from Eq. 4, will be compared to a random value u_r based on any of the schemes introduced earlier. For systematic resampling, using (6), this becomes

$$\frac{W_k}{W_M} \lessgtr \frac{1}{M}(r-1) + \tilde{u}. \tag{17}$$

This can now be rewritten as

$$W_k \lessgtr W_M \left(\frac{1}{M}(r-1) + \tilde{u} \right) \tag{18}$$

or, similar to the resampling methods earlier:

$$u_r = W_M \left(\frac{1}{M}(r-1) + \tilde{u} \right), \text{ with } \tilde{u} \sim U\left[0, \frac{1}{M}\right). \tag{19}$$

The result in Eq. 19 is similar to Eq. 8 from [7]. However, the new expression avoids any division as we can accumulate $\frac{1}{M}$ to get the $\frac{1}{M}(r-1)$ part and the random number generation (RNG) has a range independent of W_M. This is easily adaptable for stratified resampling, and, as will be shown later, can be adapted to multinomial resampling.

If the number of particles, M, is configurable, it may be required to store multiple different values for $\frac{1}{M}$ and modify the range of the random number generation accordingly (one way is to multiply a random number $\tilde{u} \sim U[0, 1)$ with $\frac{1}{M}$). Instead, it is possible to rewrite (18) as

$$M W_k \lessgtr W_M (r - 1 + \tilde{u}) \tag{20}$$

with $\tilde{u} \sim U[0, 1)$ leading to that the random number is always in a well defined interval and that the multiplication is with M instead. In this way, it is easy to change M while

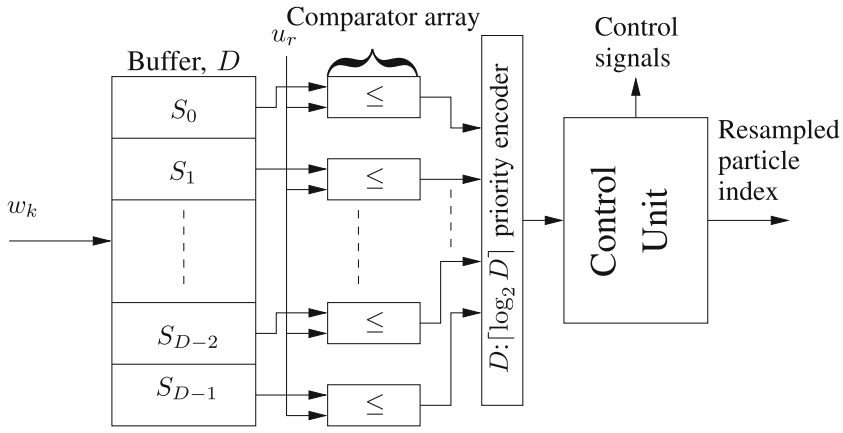

Figure 11 Architecture for the resampling unit using the pre-fetch by D technique.

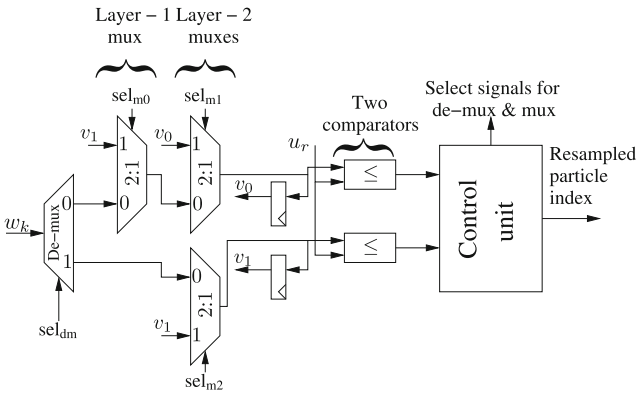

Figure 12 Detailed architecture for the resampling unit with buffer size $D = 2$.

avoiding any division and use a random number generator with a well defined and static range.

In earlier works, the cumulative sum of the weights has been stored. This leads to the word length increasing as the cumulative sum of the weights require a longer word length to be stored compared to the particle weights. Assuming that there are M particles and the particle weights have a word length of W_w, the required word length of the memory is

$$W_{\text{Mem},w} = \lceil \log_2 M \rceil + W_w \tag{21}$$

Hence, for, say, $M = 2048$, 11 more bits are required compared to W_w. Naturally, it is possible to reduce the word length by quantization of the cumulative sum, but this will come with increased errors.[2] So, in general, there will be a trade-off between accuracy and circuit area when determining the word length of the cumulative sum memory.

Instead, we suggest to compute the cumulative sum on the fly when using the W_k values. However, as W_M is required, it is required to perform this computation twice, once when storing the values in the cumulative sum memory and once when performing the resampling. An architecture combining (20) and the on-the-fly computation of cumulative sum is shown in Fig. 13. For comparison, an architecture using (18) with a static random number range without on-the-fly computation, is shown in Fig. 14.

It may be argued if multinomial resampling has any advantage over other methods [8, 10, 17]. One claimed issue is a higher computational complexity [17], either by traversing the memory for unordered random values or generating ordered random values. Here, we show that multinomial resampling can be implemented with a much lower complexity compared to what has been shown earlier.

[2]Clearly, these can be kept arbitrarily small, at the expense of memory size.

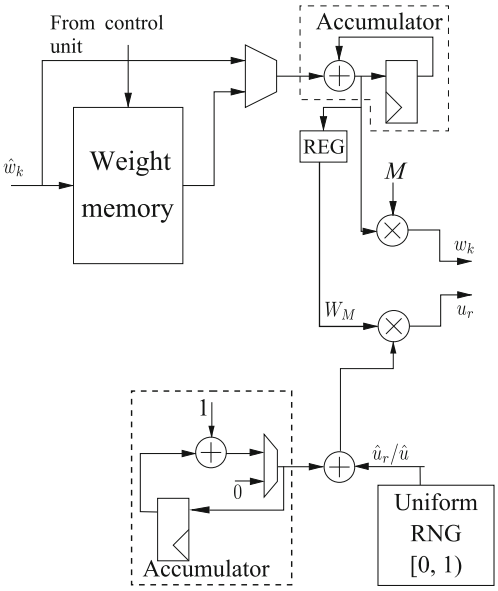

Figure 13 Weight and random number generation with on-the-fly cumulative sum for division-free stratified/systematic resampling.

Still, from a filtering perspective, it can be argued that some of the other schemes are still advantageous.

The first problem is to generate ordered random values. In [17], the equation

$$u_r = u_{r+1}\tilde{u}_r^{\frac{1}{r}},\ u_N = \tilde{u}^{\frac{1}{N}},\ \text{with } \tilde{u} \sim U\,[0,1) \tag{22}$$

was used. This require computation of the r:th root which in general is computationally demanding.

Instead, we suggest using a cumulative sum of random numbers as

$$u_r = \frac{\sum_{i=1}^{k} \hat{u}_i}{\sum_{i=1}^{M+1} \hat{u}_i} = \frac{U_r}{U_{M+1}}, \qquad k \in 1 \ldots M, \tag{23}$$

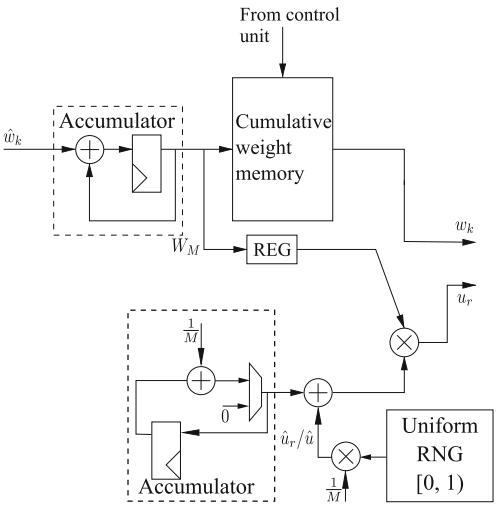

Figure 14 Weight and random number generation with cumulative sum weight memory for division-free stratified/systematic resampling.

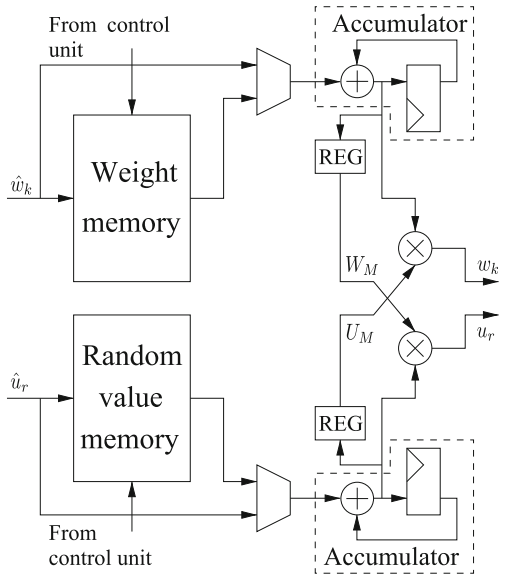

Figure 15 Weight and random number generation with on-the-fly cumulative sum for division-free multinomial resampling.

where $U_r = \sum_{i=1}^{k} \hat{u}_i$. If the random numbers are from an exponential distribution, the normalized cumulative sum will have a uniform distribution [5]. Note that the sum goes to $M+1$ to avoid that the last u_r value is always 1. For FPGA implementation, a random number generator for exponential distribution such as [30] is smaller compared to a uniform distribution [31].

Now, similar to Eq. 17, the comparison can be written as

$$\frac{W_k}{W_M} \lessgtr \frac{U_r}{U_M}. \tag{24}$$

To avoid divisions, it is simply rewritten as

$$W_k \times U_M \lessgtr U_r \times W_M, \tag{25}$$

leading to a redefinition of w_k and u_r as:

$$\begin{aligned} w_k &= W_k U_M \\ u_r &= U_r W_M. \end{aligned} \tag{26}$$

It should be noted that computing the random numbers and their cumulative sum can be executed in parallel with the particle weight computation stage. The resulting architecture for multinomial resampling with on-the-fly computation of U_r is shown in Fig. 15. A similar memory word length saving as for the cumulative sum memory is obtained by using on-the-fly-computation. Furthermore, doing the full cross multiplication, as given by Eq. 25, makes it equally efficient for non-powers-of-two number of particles. It also allows generation of random numbers in the interval $0 \rightarrow 1$ without an explicit ordering.

It should be noted that the division-free schemes will also be efficient in software implementations as the latency of a division is about three times longer compared to a multiplication, both for integer and floating-point operations.

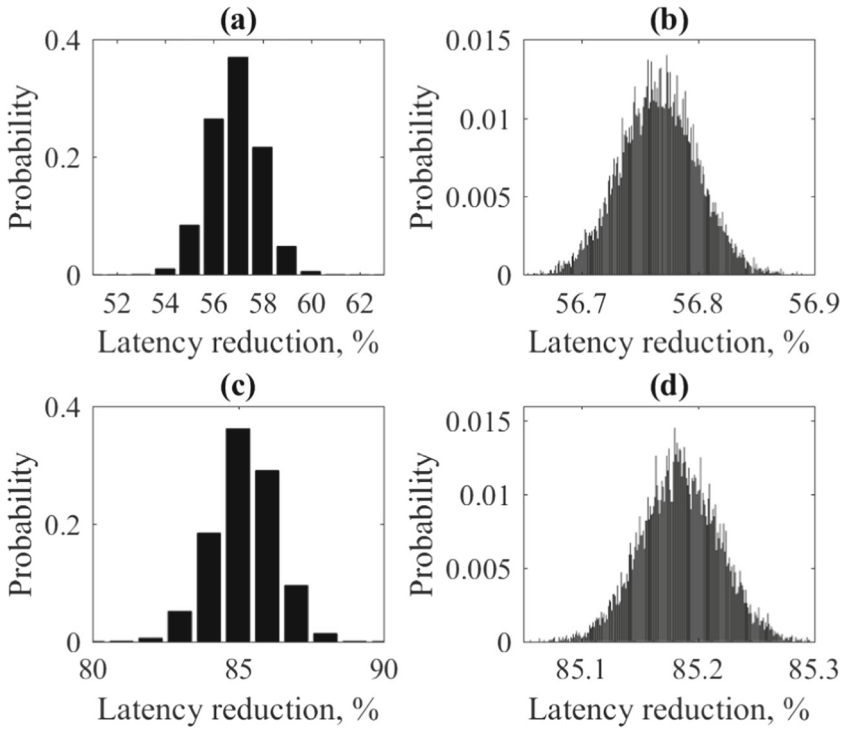

Figure 16 Latency reduction due to pre-fetch with $D = 2$, **a** $M = 10^3$ and **b** $M = 10^6$ and $D = 5$, **c** $M = 10^3$ and **d** $M = 10^6$.

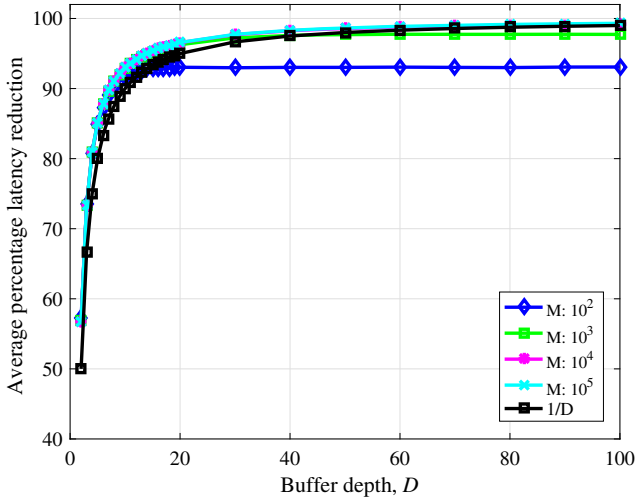

Figure 17 Average reduction in latency for pre-fetch for various values of D.

5 Results

5.1 Latency Reduction

As stated in Section 4.1, latency of the resampling step is defined as the cycles where no output is produced. The main aim of the pre-fetch technique proposed is to reduce the latency of the resampling step. To illustrate the savings obtained, the probability of latency reduction for various number of particles with buffer depth of two and five is shown in Fig. 16.

The average savings in latency is approximately 56% and 85% for $D = 2$ and $D = 5$, respectively. Also, with an increase in the number of particles, the variance of the latency reduction reduces significantly. However, increasing the buffer depth will not increase the savings proportional to the increase in the complexity to implement such a system and the rate of increase in latency reduction will decrease. This is shown in Fig. 17 for various number of particles with varying D.

As shown in Fig. 16, the latency reduction is a distribution with a certain variance. From a design perspective, there is a need of a worst case latency reduction due to the pre-fetch scheme so that the execution of the next iteration can be started. Based on the analysis of the latency reduction, a worst case value of $1/D$ is suggested and is shown in Fig. 17. It shows that for $D \ll M$, the $1/D$ is a safe approximation.

This reduction in the rate of increase in savings is because as D is increased, the degree of utilization of each place in the buffer decreases and the whole buffer is not utilized fully. This is shown in Fig. 18 for $M = 10^3$ and $M = 10^4$ for various buffer depths. There is high degree of utilization for initial places in the buffer but it decreases significantly for the later places.

The divergence in the average percentage latency reduction between different values of M, as D is increased in Fig. 17, is because the decrease in the buffer utilization is not linear, as shown in Fig. 18a and b. This non-linearity only appears for large values of D relative to M, meaning that the smaller the value of M, the earlier this non-linearity will appear, with respect to D. This non-linearity accounts

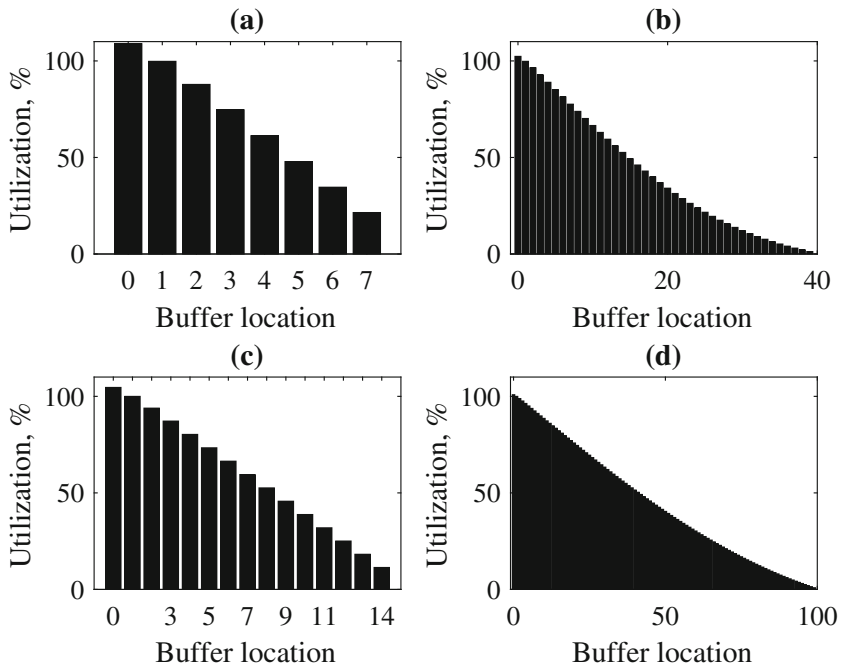

Figure 18 Buffer utilization for $M = 10^3$, **a** $D = 8$ and **b** $D = 40$ and $M = 10^4$, **c** $D = 15$ and **d** $D = 100$

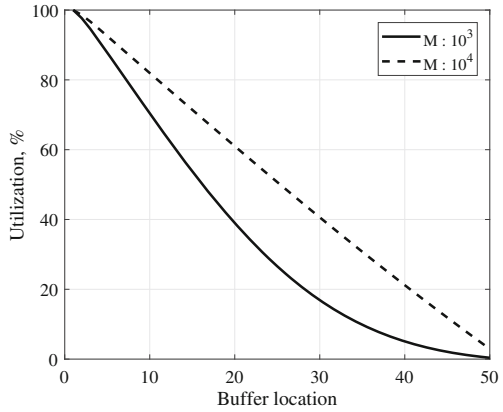

Figure 19 Buffer utilization for varying number of particles with $D = 50$.

for a lower utilization of the buffer with a given D for less number of particles, M. This is shown in Fig. 19, where it can be seen that for $D = 50$, $M = 10^4$ has a high utilization of the buffer places as compared to $M = 10^3$.

However, as D increases, so will the hardware complexity. This is because the size of the de-multiplexer and the size and the number of multiplexers, registers and comparators shown in Fig. 12 grows with the increase in D. The growth of the architecture, in terms of area, grows quadratically which is consistent with the increase in the size of the multiplexer. This increase is shown in Fig. 20 for $M = 2048$, when synthesized using standard cells on a 65 nm low power standard V_T CMOS process with a nominal working temperature of 125 °C and an operating voltage of $1.15V$. Also shown is the approximation that the hardware complexity increases quadratically as a function of the buffer depth, D.

It was shown earlier in Fig. 17 that the latency decreases as buffer depth is increased. However, the rate of increases drops as higher buffer depths are reached. Furthermore, increase in D also quadratically increases the area. In order to determine the right trade-off between reduction in latency and increase in the area, the area latency product was computed as a function of buffer depth D and plotted in Fig. 21 for $M = 2048$. It shows that the normalized area initially decreases up to $D = 5$ and then increases. Thus, a buffer depth of five gives the best trade-off between latency and area.

The consequence of the latency reduction is that the architecture can process new inputs faster. The previously fastest resampling scheme (not considering partial resampling), residual systematic resampling, RSR, can process a new sample every $2M + L$ cycles, where M is the number of particles and L is the pipeline latency of the computational units. Often $L \ll M$, so we assume a time of $2M$. To give a bit margin for the expected latency reduction, we assume for $D = 2$ that 50% of the cycles are removed and for $D = 5$, 80% of the cycles. The resulting time between input samples are then $1.5M$ and $1.2M$ for $D = 2$ and $D = 5$, respectively, compared to $2M$ for RSR. This can be used for increasing the sample rate or the number of particles that can be processed for a given sample rate.

5.2 Memory Usage by Generalized Division-Free Architecture for Multinomial Resampling

The original architecture proposed in [1], without on-the-fly cumulative sum will have a relatively larger memory compared to the one in Fig. 13, for multinomial resampling. However, the on-the-fly cumulative sum architecture has two extra multiplexers and associated control circuity. Both the architectures were implemented on a Xilinx Virtex-6 FPGA, as they have specialized blocks to implement multipliers and memories.

Table 1 illustrates the resource usage by the two architectures, when implemented on the mentioned FPGA. Designs for larger number of particles were realized to reflect more practical scenarios while also using non-powers-of-two

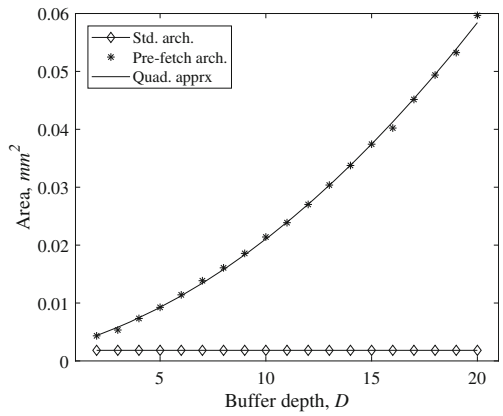

Figure 20 Area of the resampling step as a function of buffer depth, D with $M = 2048$.

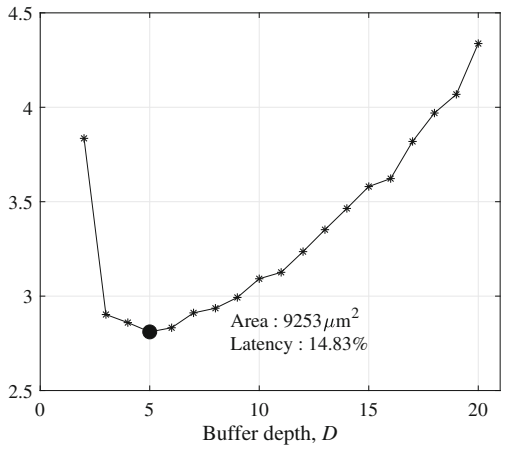

Figure 21 Area latency product as a function of buffer depth, D with $M = 2048$.

Table 1 Complexity, in Terms of FPGA resources, of Architectures Based on Stored and On-the-fly Cumulative Sum (CS).

Particle	Stored CS		On-the-fly CS	
count	LUT	Mem.	LUT	Mem.
512	375	2†	456	2*
1000	234	2†	299	2*
1024	210	2†	292	2*
2000	226	4†	317	2†
2048	231	4†	358	2†
3000	290	6† +2*	378	4†
4096	273	6† +2*	372	4†
8192	289	16†	381	8† +2*
10000	311	32†	410	18†
15000	312	32†	427	18†
16384	306	32†	421	18†
20000	330	66†	421	36†

*18k BRAM

†6k BRAM

number of particles to reflect the flexibility of the proposed architecture. Architecture based on on-the-fly cumulative sum reduces the memory requirement by half at the cost of extra logical complexity, in terms of LUT. This is due to the presence of multiplexers and associated control logic. However, memory blocks in an FPGA are far less in number than LUTs, making the on-the-fly cumulative sum option an attractive one.

6 Conclusion

In this work, a new technique of pre-fetch has been proposed to reduce the latency of the resampling step. In this technique, new particle weights are pre-fetched during the same cycle where random values are fetched. This helps in reducing the total number of cycles needed for resampling. An area latency product shows that $D = 5$ achieves the best trade-off between latency reduction and hardware cost, resulting in a latency reduction of about 85%. Hence, with a conservative design assuming an 80% reduction, an implementation will be able to process a new input every $1.2M$ clock cycle, where M is the number of particles, compared to $2M$ when using e.g. residual systematic resampling [8]. This additional speed can be used either to increase the sample rate or use more particles within the same sample rate.

Futhermore, it is shown that the size of the particle weight memory can be significantly reduced by calculating the cumulative sum on-the-fly before the resampling. When implemented on FPGA, a memory reduction of up to 50% is shown. Division-free resampling schemes for stratified, systematic, and multinomial resampling are proposed. These can be combined with the latency and memory reduction schemes, but may also be useful in software implementations as division has a higher latency compared to multiplication. As part of the division-free scheme for multinomial resampling, a new ordered random number scheme is also proposed, removing the need for n:th-root computation from earlier schemes.

Acknowledgments This work is funded by ELLIIT, project number 3.5: Parallel architectures for sampling based nonlinear filters.

Funding Information Open access funding provided by Linköping University.

References

1. Alam, S.A., & Gustafsson, O. (2015). Generalized resampling architecture and memory structure for particle filters. In *Proc. Europ. Conf. Circuit Theory Design*. Trondheim.
2. Ansari, N., & Chamnongthai, K. (2019). Particle filtering with adaptive resampling scheme for modal frequency identification and dispersion curves estimation in ocean acoustics. *Applied Acoustics*, *154*, 90–98.
3. Arulampalam, M.S., Maskell, S., Gordon, N., Clapp, T. (2002). A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing*, *50*(2), 174–188.
4. Athalye, A., Bolić, M., Hong, S., Djurić, P.M. (2005). Generic hardware architectures for sampling and resampling in particle filters. *EURASIP Journal on Advances in Signal Processing*, *2005*(17), 2888–2902.
5. Bentley, J.L., & Saxe, J.B. (1980). Generating sorted lists of random numbers. *ACM Transactions on Mathematical Software*, *6*(3), 359–364.
6. Bolić, M. (2004). Architectures for efficient implementation of particle filters PhD thesis. The Graduate School of Electrical Engineering, Stony Brook University.
7. Bolić, M., Athalye, A., Djurić, P.M., Hong, S. (2004). Algorithmic modification of particle filters for hardware implementation. In: *Proc. Europ. signal process. conf.* (pp. 1641–1644).
8. Bolić, M., Djurić, P.M., Hong, S. (2004). Resampling algorithms for particle filters: a computational complexity perspective. *EURASIP Journal on Advances in Signal Processing*, *2004*(15), 2267–2277.
9. Daum, F. (2005). Nonlinear filters: beyond the Kalman filter. *IEEE Aerospace and Electronic Systems Magazine*, *20*(8), 57–69.
10. Douc, R., & Cappé, O. (2005). Comparison of resampling schemes for particle filtering. In *Proc. Int. symp. image signal process. analysis* (pp. 64–69).
11. Doucet, A., & Wang, X. (2005). Monte Carlo methods for signal processing: a review in the statistical signal processing context. *IEEE Signal Processing Magazine*, *22*(6), 152–170.
12. Doucet, A., Godsill, S., Andrieu, C. (2000). On sequential Monte Carlo sampling methods for Bayesian filtering. *Statistics and Computing*, *10*(3), 197–208.

13. Doucet, A., de Freitas, N., Gordon, N. (Eds.) (2001). *Sequential Monte Carlo methods in practice*. New York: Springer.
14. Gordon, N.J., Salmond, D.J., Smith, A.F.M. (1993). Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *IEE Processing Radar Signal Process*, *140*(2), 107–113.
15. Gustafsson, F., Gunnarsson, F., Bergman, N., Forssell, U., Jansson, J., Karlsson, R., Nordlund, P.J. (2002). Particle filters for positioning, navigation, and tracking. *IEEE Transaction on Signal Process*, *50*(2), 425–437.
16. Hendeby, G., Karlsson, R., Gustafsson, F. (2010). Particle filtering: the need for speed. *EURASIP Journal on Advances in Signal Processing*, *2010*, 22.
17. Hol, J.D., Schön, T.B., Gustafsson, F. (2006). On resampling algorithms for particle filters. In *Proc. IEEE Nonlinear stat. signal process. workshop* (pp. 79–82).
18. Hong, S., Chin, S.S., Djurić, P., Bolic, M. (2006). Design and implementation of flexible resampling mechanism for high-speed parallel particle filters. *Journal of VLSI Signal Processing Systems*, *44*(1–2), 47–62.
19. Hong, S., Shi, Z., Chen, K. (2008). Compact resampling algorithm and hardware architecture for paticle filters. In *Proc. Int. conf. comm. circuits syst.* (pp. 886–890).
20. Hong, S.H., Shi, Z.G., Chen, J.M., Chen, K.S. (2010). A low-power memory-efficient resampling architecture for particle filters. *Circuits, Systems and Signal Processing*, *28*(1), 155–167.
21. Hostettler, R., & Djurić, P.M. (2015). Vehicle tracking based on fusion of magnetometer and accelerometer sensor measurements with particle filtering. *IEEE Transactions on Vehicular Technology*, *64*(11), 4917–4928.
22. Jiang, Z., Zhou, W., Li, H., Ni, W., Hang, Q. (2017). A new kind of accurate calibration method for robotic kinematic parameters based on extended Kalman and particle filter algorithm. *IEEE Transactions on Industrial Electronic*, *PP*(99), 1–1.
23. Joo, Y., & McKeown, N. (1998). Doubling memory bandwidth for network buffers. In *Proc. IEEE joint conf. IEEE comput. comm. soc.*, (Vol. 2 pp. 808–815).
24. Li, T., Bolić, M., Djurić, P.M. (2015). Resampling methods for particle filtering: classification, implementation and strategies. *IEEE Signal Processing Magazine*, *32*(3), 70–86.
25. Lin, S.D., Lin, J.J., Chuang, C.Y. (2015). Particle filter with occlusion handling for visual tracking. *IET Image Processing*, *9*(11), 959–968.
26. Ristic, B., Arulampalam, M.S., Gordon, N. (2004). *Beyond the Kalman filter: particle filters for tracking applications*. Artech House: Norwood.
27. Sankaranarayanan, A.C., Srivastava, A., Chellappa, R. (2008). Algorithmic and architectural optimizations for computationally efficient particle filtering. *IEEE Transactions on Image Processing*, *17*(5), 737–748.
28. Scharcanski, J., de Oliveira, A.B., Cavalcanti, P.G., Yari, Y. (2011). A particle-filtering approach for vehicular tracking adaptive to occlusions. *IEEE Transactions on Vehicular Technology*, *60*(2), 381–389.
29. Shi, Z.G., Hong, S.H., Chen, J.M., Chen, K.S., Sun, Y.X. (2008). Particle filter-based synchronization of chaotic colpitts circuits combating AWGN channel distortion. *Circuits Syst Signal Process*, *27*(6), 833–845.
30. Thomas, D.B., & Luk, W. (2008). Sampling from the exponential distribution using independent bernoulli variates. In *Proc. Int. conf. field-programmable logic applicat.* (pp. 239–244).
31. Thomas, D.B., Howes, L., Luk, W. (2009). A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In *Proc. ACM/SIGDA int. symp. field-programmable gate arrays FPGA '09* (pp. 63–72). New York: ACM.
32. Tian, M., Bo, Y., Chen, Z., Wu, P., Yue, C. (2019). Multi-target tracking method based on improved firefly algorithm optimized particle filter. Neurocomputing Available online.