A TWO STEP HARDWARE DESIGN METHOD USING CλaSH

Rinse Wester, Christiaan Baaij, Jan Kuper

Department of Electrical Engineering, Mathematics, and Computer Science
University of Twente
Enschede, The Netherlands
email: {r.wester, c.p.r.baaij,j.kuper}@utwente.nl

ABSTRACT

In order to effectively utilize the growing number of resources available on FPGAs, higher level abstraction mechanisms are needed to deal with increasing complexity resulting from large designs. Functional hardware description languages, like the CλaSH HDL, offer adequate abstraction mechanisms such as polymorphism and higher-order functions.

This paper describes a two step design method to implement a DSP application on an FPGA, starting from a mathematical specification, followed by an implementation in CλaSH. A non trivial application, a particle filter, is used to evaluate both the method and CλaSH. First, a straightforward translation is performed from the mathematical definition of a particle filtering to Haskell, a functional programming language with syntax and semantics similar to CλaSH. Secondly, minor changes are applied to the Haskell implementation so that it is accepted by the CλaSH compiler. The resulting hardware produced by our method is evaluated and shows that this method eases reasoning about structure and parallelism in both the mathematical definition and the resulting hardware.

1. INTRODUCTION

Traditional approaches to hardware implementations of signal processing applications require a lot of manual translations, as there is often a semantic mismatch between the 'golden' reference in the form of a C-program (sequential), the original algorithm (mathematical), and hardware (parallel). When starting from a golden reference in the form of a program written in Haskell [1], a functional language, you remove at least one semantic mismatch, as the Haskell functions can be seen as "the original mathematics equations in ASCII format". We use CλaSH [2], a functional Hardware Description Language (HDL), with syntax and semantics similar to Haskell, to create the actual hardware implementation. As a CλaSH description is an implicitly parallel description, there is no mismatch with the eventual circuit.

This research is conducted as part of the Sensor Technology Applied in Reconfigurable systems for sustainable Security (STARS) project www.starsproject.nl

We propose a two step design method using CλaSH applied to a signal processing application. The first step is reformulating the mathematical definition of the signal processing application in Haskell. This Haskell description is then modified slightly such that it is accepted by CλaSH and hardware can be generated. The design method is shown graphically in Figure 1.

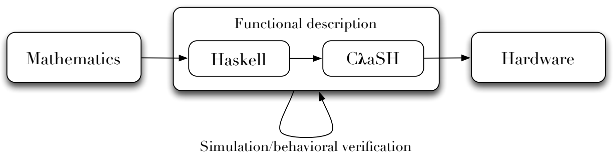

Fig. 1. Hardware design method

The reason for splitting the design into two steps is that fundamental changes in the mathematical definition are easier to change in the Haskell specification than in the CλaSH specification. The haskell specification uses double precision floating point operations while the implementation in CλaSH uses fixed point operations. Therefore, fundamental changes are performed in the Haskell definition while the hardware implementation details are covered by the CλaSH implementation. The benefit these two steps is a clear division between architectural design and low level hardware details like fixed point representation.

The application chosen for evaluation of the design method is particle filtering as it is challenging due to excessive parallelism, data dependencies and feedback.

The rest of this paper is structured as follows: first, the process of particle filtering is explained including the mathematical formulation. This is followed by a short introduction to the CλaSH HDL. In Section 3 we elaborate on our design method applied to particle filtering, where we first transform the mathematical definition into a Haskell reference design which then is transformed into a valid CλaSH description by small adaptations. Finally, we show the performance characteristics of our parallel implementation in Section 4, and end with conclusions and future work in Section 5.

1.1. Related work

The use of Haskell to design hardware is not new, the work by Gill and Farmer [3] uses Kansas Lava, a Domain Specific Language (DSL) embedded in Haskell, to implement an efficient FPGA implementation of an LPDC decoder. While their work focuses on applying many types of transformations on the reference Haskell specification to get an efficient implementation, we focus on trying to stay as close to the Haskell reference implementation as possible.

Much work on parallel particle filters using FPGAs has been done at Stony Brook University [4], covering generic architectures for different types of particle filters and techniques to increase the performance of resampling. In terms of parallelization, their approach is applying changes to the architecture to increase the performance while the approach taken is this paper is utilizing as much parallelism in the mathematical description as possible.

The need for abstraction in hardware design has led to a technique called high-level synthesis [5]. High-level synthesis takes a high-level language (usually C) and translates this to a hardware description language like VHDL or Verilog. The main difference between high-level synthesis and the approach taken in this paper is that our method uses a more mathematical oriented language (Haskell) instead of the inherently sequential language C.

2. BACKGROUND

The background information is divided into two subsections. First, Subsection 2.1 introduces (the mathematical structure of) particle filters. The last subsection (Subsection 2.2), introduces the CλaSH HDL including an example.

2.1. Particle Filtering

Particle filtering is a Bayesian filtering technique to find the state variables of a particular system based on noisy measurements [6]. For each measurement, the belief of the state is recursively updated resulting in a posterior belief about the state of the system. Since these measurements contain noise, the resulting belief will be in the form of a Probability Density Function (PDF). Several examples of these measurements are frames from video streams and range-Doppler images from radar. Analytically finding the posterior is often mathematically intractable (unable to solve the integrals) which is why approximation methods are used. A particle filter is a Monte Carlo approach that repeatedly generates random samples and eliminates them partially according to a selection function. Mathematically, the filtering problem is to find the PDF of the state vector x_k given the measurement z_k (k is the iteration number of the filter):

$$p(x_k|z_k) \tag{1}$$

In a particle filter, this PDF is approximated by a collection of particles $x_k^{(i)}$ where $i = 1 \ldots N$ is the index of a particle. A higher density of particles represents a higher probability in the continuous state space. Figure 2 shows both the continuous PDF and a particle filter approximation.

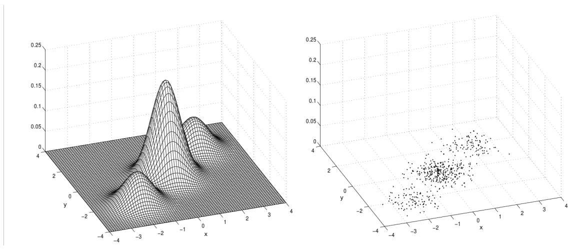

Fig. 2. Continuous PDF and particle filter approximation

A commonly used type of particle filter is the Sequential Importance Resampling Filter (SIRF) which consists of four steps: *prediction*, *update*, *normalization* and *resampling* [7]. Each time a measurement arrives (the sequential part), these four steps are performed and alter the particles for the next measurement forming the feedback loop shown in Figure 3.

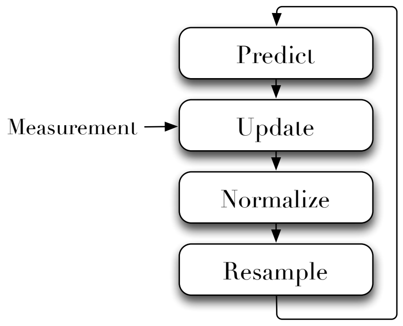

Fig. 3. Structure of particle filter

2.1.1. *Prediction*

During *prediction*, the next state is derived from the current state using the known dynamics of the system. Mathematically this comes down to evaluating Equation 2 which can be read as "draw particles from the PDF enforced by the system dynamics" (where $\sim$ is the sampling operator):

$$x_k^{(i)} \sim p(x_k|x_{k-1}) \tag{2}$$

Drawing particles from this distribution is performed by evaluating the System Dynamics function f for all particles,

$x_k^{(i)} = f(x_{k-1}^{(i)}, u_k)$. u_k is noise sampled from some probability distribution, not necessarily a Gaussian distribution.

2.1.2. Update

When a new prediction has been made, a measurement is used to update this prediction during the update step. In this step, weights $\omega_k^{(i)}$ are assigned to all particles representing the importance of a particular particle. Mathematically this is formulated in Equation 3. Note that determining the weights looks like an unconditional PDF but it is actually deterministic (expressed with $=$):

$$\omega_k^{(i)} = p(z_k | x_k^{(i)}) \tag{3}$$

The generic mathematical formulation of the update step is shown in (3). To find the actual weights, a likelihood function g is needed. This function returns, given a particle $x_k^{(i)}$, a single measurement z_k and noise sample v_k, a weight $\omega_k^{(i)}$ for each particle $x_k^{(i)}$:

$$\omega_k^{(i)} = g(x_k^{(i)}, z_k, v_k), \quad \text{for} \quad i = 1 \dots N \tag{4}$$

2.1.3. Normalization

The integral of any real PDF should be one, similarly this should also hold for the sum of all the weights. This is realized in the normalization step where every new weight $\tilde{\omega}^{(i)}$ is found by:

$$\tilde{\omega}^{(i)} = \frac{\omega^{(i)}}{tot\omega} \quad \text{for} \quad i = 1 \dots N$$
$$\text{where} \quad tot\omega = \sum_{n=1}^{N} \omega^{(n)} \tag{5}$$

2.1.4. Resampling

The last step performed in a particle filter iteration is the resampling step, which is needed to prevent degeneracy of weights [7]. Particles are replicated 0,1 or more times according to their weight $\tilde{\omega}^{(i)}$, while keeping the total number of particles constant. Mathematically, the resampling process is selecting particles as formulated in (6):

$$p\left(\tilde{x}_k^{(i)} = x_k^{(i)}\right) = \tilde{\omega}_k^{(i)} \quad \text{for} \quad i = 1 \dots N \tag{6}$$

The probability that a particle $x_k^{(i)}$ is replicated (the particle after resampling is denoted as $\tilde{x}_k^{(i)}$) proportionally to its weight $\tilde{\omega}_k^{(i)}$ is expressed in (6). Figure 4 shows the process of resampling, as expressed in (6), graphically.

As shown in Figure 4, particles with a low weight, those particles have a low $\tilde{\omega}_k^{(i)}$, are discarded (x) while particles with a high weight are replicated (o). Resampling is highly data dependent which is challenging for a parallel hardware implementation [8].

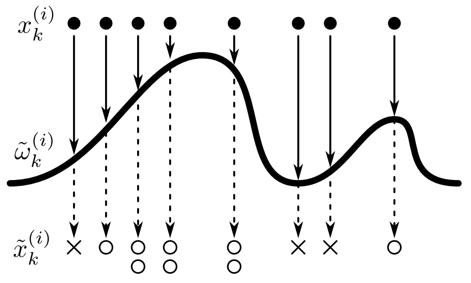

Fig. 4. Graphical representation of resampling

There exist several techniques to implement resampling [9], of which systematic resampling is commonly used. In short, Systematic Resampling replicates particles according to the amount of fixed intervals $\frac{1}{N}$ are within the range of a single weight given a random offset $0 < u_0 < \frac{1}{N}$. The resampling technique used in this paper is called Residual Systematic Resampling, a modified version of Systematic Resampling but mathematically equivalent [8].

Residual Systematic Resampling consists of two steps: first the replication factor is determined based on the weight of a particle, followed by the actual replication of particles. Equation 7 gives an expression to determine the replication factor r_i for a single weight $\omega^{(i)}$.

$$r_i = \lfloor (\omega^{(i)} - u_{i-1}) * N \rfloor + 1$$
$$u_i = u_{i-1} + \frac{r_i}{N} - \omega^{(i)}$$
$$\text{for} \quad i = 1 \dots N \quad \text{and} \quad u_0 \sim \mathcal{U}(0, \frac{1}{N}) \tag{7}$$

Figure 5 shows a graphical representation of RSR expressed in (7). Basically the replication factor is determined by the amount of arrows in the range of a weight. The randomization in resampling is implemented by the random offset u_0 sampled from the uniform distribution ($\mathcal{U}$).

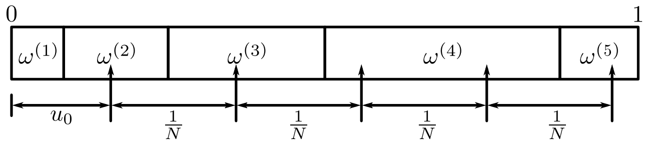

Fig. 5. Graphical representation of Residual Systematic Resampling

When all replication factors r_i have been found, the actual replication of particles can be performed. During replication, every particle $x_k^{(i)}$ is replicated r_i times and the resulting sets are merged into a single set of new particles $\tilde{x}_k^{(i)}$.

The mathematical formulation of replication using the concat operator ($\|$) is shown in (8).

$$\{\tilde{x}_k^{(1)}, \tilde{x}_k^{(2)} \ldots \tilde{x}_k^{(N)}\} = \mathop{\Big\|}_{n=1}^{N} replicate(x_k^{(i)}, r_i) \qquad (8)$$

Since the replication of particles is highly dependent on the replication factor r_i, the fully parallel hardware implementation of resampling is expected to be the most expensive component.

2.2. CλaSH

CλaSH [2] is a functional HDL, whose descriptions are translated to synthesizable VHDL by the CλaSH compiler. The CλaSH language has many advanced features such as polymorphism, higher-order functions, pattern matching and type derivation. Polymorphism and higher-order functions (functions that have functions as argument or result) allow circuit designers to describe parameterizable circuits in a natural way.

CλaSH is a synchronous HDL where, on the lowest level, everything is expressed as a Mealy machine i.e. every output and new state is therefore a function of the current state and input. Listing 1 shows an example of a discrete integrator expressed in CλaSH.

Listing 1 Integrator example in CλaSH.

```
integrator (State s) inp = (State s', out)
  where
    s'  = s + inp
    out = s'
```

All hardware components described using CλaSH have a structure as expressed in Listing 1. In order to distinguish between inputs for the component and data coming from registers, a keyword *State* is used. Every variable preceded by *State* will be translated to a register by the CλaSH compiler. In the integrator example of Listing 1, the first occurrence of state s is the current state (the output of the registers) while the second occurrence s' is the new state of the register (the input). Note that, in Haskell, arguments are generally written without brackets.

Even though types are very important in Haskell, we chose not to display these in the listings as this paper is about the structural correspondence between the mathematical formulation and the resulting hardware.

Every CλaSH description is also a valid Haskell program. This means that CλaSH descriptions can be simulated using a Haskell compiler or interpreter such as GHC [10]. Although every CλaSH design is a valid Haskell program, the reverse relation does not hold. Concepts such as recursive function definitions and recursive datatypes are for example supported in Haskell, but not (yet) in CλaSH.

3. DESIGN METHOD

The design method consists of two steps. First, the mathematical definition of particle filtering is reformulated to Haskell, while trying to preserve the original semantics of the equations. The second step is to perform minor changes on the Haskell description such that CλaSH can be used for translation to VHDL.

The following two sections describe how the two step method is applied to the simple particle filter described in [11].

3.1. Math to Haskell

To complete the mathematical specification as given in the background section, a state space model f and a likelihood function g still have to be defined. The application is a simple tracking particle filter for tracking a white square on a dark background. Since tracking the square results in a PDF representation of the position, each particle represents a possible position of the square. A single particle can be represented using a tuple with a position x, y and weight ω as (x, y, ω).

As a state space model, we use a uniform movement in an area of 32 × 32 pixels ($\mathcal{U}(-16, 16)$) resulting in the following expression for f.

$$\begin{aligned} f\left(x_k^{(i)}, u_k\right) &= x_k^{(i)} + u_k \\ \text{where} \quad u_k &= \langle \delta x, \delta y \rangle \to \delta x, \delta y \sim \mathcal{U}(-16, 16) \end{aligned} \qquad (9)$$

Weight assignment is done by the likelihood function g based only on the color of the pixel located at the position x, y from a single particle. Particles positioned inside the square should get a high weight while particles outside a low weight. This is implemented by finding the color distance as expressed in (10):

$$\begin{aligned} g\left(x_k^{(i)}, z_k^{(i)}\right) &= \frac{1}{1 + (255 - z_k^{(i)}[x, y])^2} \\ \text{where} \quad & x, y \in x_k^{(i)} \end{aligned} \qquad (10)$$

Now that all necessary functions have been defined, the mathematical definition of the prediction step expressed in (9) can be translated to Haskell. The prediction step in Haskell is generic i.e. the actual system dynamics function f is given as argument. Listing 2 shows how to express the prediction step in Haskell.

Listing 2 Prediction step in Haskell

```
predict $f$ $ps$ $us$ = $ps'$
  where
    $ps'$ = zipWith $f$ $ps$ $us$
```

As can be seen in Listing 2, the prediction step accepts a state space model function f, a set of particles ps and a list of random offsets us. Every particle and every offset is pairwise combined by f using *zipWith*.

The results of the prediction step are combined with a measurement in the update step. Again, the update step formulated in (3) is also generic by leaving the actual likelihood function as argument. As formulated in (3), every particle is combined with a single measurement to find the weight for each particle. In Haskell, this is expressed using the higher order function *map* (Listing 3):

Listing 3 Update step in Haskell

```
update $g$ $z$ $ps$ = $ps'$
  where
    $ps'$ = map ($g$ $z$) $ps$
```

As can be seen in Listing 3, the update step accepts three arguments: the likelihood function g, a measurement z and a list of particles ps. In the body, the likelihood function g is first assigned a measurement z after which it is applied to all particles (mapped over) ps.

Also translating the normalization step from Equation 5 is performed in a similar way as can be seen in Listing 4.

Listing 4 Normalization step in Haskell

```
$normalize$ $ps$ = $ps'$
  where
    $tot\omega$ = $sum$ (map $weight$ $ps$)
    $ps'$   = map ($\lambda$ $(x, y, \omega) \to (x, y, \omega\ /\ tot\omega)$) $ps$
```

As shown in Listing 4, the total weight $tot\omega$ is determined by first selecting only the weights of all particles ps using the $weight$ function. The $weight$ function is implemented as $weight(x, y, \omega) = \omega$. All weights are then accumulated in $tot\omega$. In the last line, a lambda expression is applied (mapped) to all particles ps. The lambda expression accepts a particle and replaces only the weight by the normalized weight.

The last step to formulate is the resampling step which consists of two steps: first the replication factor is determined based on the weight of a particle, followed by the actual replication of particles. Equation 7 gives an expression to determine the replication factor r_i for weight $\omega^{(i)}$.

The length of the recursion in Equation 7 depends only on the length of the weight list. We use a functional language feature called *pattern matching* to terminate the recursion. Listing 5 shows the two phases in the recursion. Either, not all weights have not been processed yet (line 1), or the last weight has been processed and an empty list [] is left (line 2). During processing, the list of weights $(\omega : \omega s)$ shrinks every time by taking the first element ω and calculating a replication factor based on that element. Calculation continues recursively with the remainder of the weights ωs until no weights are left [].

Listing 5 Haskell code to determine replication factors

```
ws2rfs $u$ [ ] = [ ]
ws2rfs $u$ $(\omega : \omega s)$ = $r$ : (ws2rfs $u'$ $\omega s$)
  where
    $r$  = $floor$ $((\omega - u) * N) + 1$
    $u'$ = $u + r\ /\ N - \omega$
```

Reformulating the replication of particles in Equation 7 to Haskell, comes down to translating the union operator $\cup$ to the Haskell variant $+\!\!+$. Each particle p is replicated r_i times and all those sets of particles are merged using the $+\!\!+$ operator (Listing 6):

Listing 6 Replication of particles

```
replps [ ]       [ ]       = [ ]
replps ($p$ : $ps$) ($r$ : $rs$) = (replicate $r$ $p$) ++ replps $ps$ $rs$
```

Replication of particles is performed recursively, using the $replps$ function (Listing 6). This function accepts two lists: a list of particles $(p : ps)$ and the corresponding replication factors $(r : rs)$. For every particle p and replication factor r, drawn from their respective lists, p is replicated r times. The final set of resampled particles is found by concatenating all replicated particles using the $+\!\!+$ operator.

The complete resampling step is formed by the composition of the function $ws2rfs$ and $replps$. All replication factors only depend on the weights ωs; these are extracted from the particles ps (first line in the *where* clause of Listing 7). The resulting list of replication factors rs is then used for replication by $replps$ (third line in the *where* clause). Finally, the last line replaces the weight by $\frac{1}{N}$ since all particles are of equal importance after resampling.

Listing 7 Complete resampling step in Haskell

```
$resample\ ps = ps'$
  **where**
    $\omega s \quad = \mathbf{map}\ weight\ pd$
    $rs \quad = \text{ws2rfs}\ \omega s$
    $ps\_r = \text{replps}\ ps\ rs$
    $ps' \quad = \mathbf{map}\ (\lambda\ (x, y, \_) \rightarrow (x, y, \frac{1}{N}))\ ps\_r$
```

3.2. CλaSH implementation

After creating the Haskell reference we continue with the CλaSH implementation by replacing only those parts that are not directly supported by the CλaSH compiler. The resulting implementation in CλaSH is fully parallel i.e. all operations in the Haskell code result in a distinct component on the FPGA.

The first difference between the Haskell reference and the CλaSH implementation is that we go from lists, which can have an arbitrary size at runtime, to vectors, which have a fixed length encoded in their type. This restriction does not change anything to the algorithm itself, as the number of particles is constant. In general, this does not hold and lists could be implemented be sequentially accessing elements of the list from a memory. Another minor change is that the *zipWith* from Listing 2 is replaced by a *vzipWith*. The *vzipWith* results in the same structure except that it works on vectors instead of lists. A similar replacement is used in the update and normalization step.

The resampling step, on the other hand, is more complicated to translate since it is expressed recursively. Like the Haskell reference, resampling starts by determining the replication factors, followed by the actual replication of particles.

As shown in the Haskell formulation of Listing 5, determining the replication factors is done using a tail recursive function. CλaSH does not support recursion (yet). Therefore, this recursion has to be reformulated using functions that are supported by CλaSH. Since the length of the recursion only depends on the amount of particles, it can be replaced by a vector based function called *vscanl*. Although the *scanl* is directly supported in CλaSH using *vscanl*, it is interesting to show the structural correspondence between the mathematical formulation and the resulting hardware. *vscanl* accepts a function *rf*, a starting value u_0, and a vector with weights ωs. The function argument of *vscanl* is applied to each element in the vector while accumulating intermediate values and sending this to the output thus being equivalent with the recursive definition with Listing 5. The CλaSH implementation of Equation 7 is shown in Listing 8 while Figure 6 shows the corresponding hardware structure.

Listing 8 Determining replication factors in CλaSH

```
$\text{ws2rfs}\ a\ \omega s = vscanl\ rf\ (0, u0)\ \omega s$
$rf\ (u, r)\ \omega = (u', r')$
  **where**
    $r' = floor\ ((\omega - u) * N) + 1$
    $u' = u + \frac{r}{N} - \omega$
```

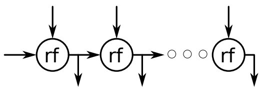

Fig. 6. Structural view of *vscanl*

The actual replication of particles is performed by N parallel multiplexers. Each multiplexer selects a single particle and puts this on the output depending on a list of multiplexer indices calculated from the set of replication factors.

Listing 9 Replication in CλaSH

```
$\mathbf{replicate}\ ps\ is = ps'$
  **where**
    $ps' = \mathbf{map}\ (ps!)\ is$
```

As shown in Listing 9, the *replicate* functions accepts two arguments, the list of particles ps and a list of indices is. Given the whole list of particles ps and a single index from is, a particle is selected using the index operator !. A shown in the CλaSH code, *map* is used to perform the multiplexing using each index in is. The code also shows an other powerful abstraction mechanism called partial application. With partial application, only a subset of the arguments are given to a function resulting in a function having only the remainder of arguments. This is applied in Listing 9 at the index operator !, the list of particles is already given since it is used for every index. The ! has only a single argument left which is supplied using the *map* function since it is applied to every index in is. Figure 7 shows the resulting hardware:

4. RESULTS

In the previous sections, the mathematical definition of a tracking particle filter has been reformulated in a Haskell program. Using this Haskell program, the minimum amount of particles for this particular application has been determined to be 32 particles using simulation. The number of

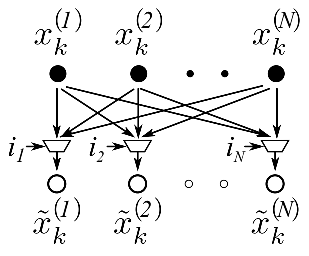

Fig. 7. Structural view replication

particles have been reduced until the particle filter was not able to track the object anymore. We then transformed all recursive function definitions to use higher-order functions e.g. *vzipWith* and *vscanl*, so that the code could be compiled by the CλaSH compiler. The resulting CλaSH description was verified to have the same external behavior as the reference description, using the testbench that was used to verify the functional correctness of the Haskell program. We determined the feasibility of the parallel CλaSH implementation by synthesizing the design for a Xilinx Virtex 6 FPGA (XC6VLX240T). An overview of the resource usage for the different parts of the particle filter are shown in Table 1.

Table 1. Area of components

Component	**LUTs**
Prediction	704
Update	954
Normalization	1402
Resampling	35978
Total	39038

In terms of performance, the synthesized particle filter achieves a throughput of 24 million particles per second. However, fully parallel resampling uses a lot of FPGA area and is therefore the biggest bottleneck in this design. Due to all data dependencies in the resampling step, all possible replications have to be considered, resulting in the large area. Although the particle filter has not been implemented in VHDL directly, similar resource consumption is expected based on experience in [12].

5. CONCLUSIONS AND FUTURE WORK

We created a fully parallel implementation of a particle filter using our two step design method with the functional HDL CλaSH. The use of higher-order functions and polymorphism allowed us to keep the mathematical definition and the resulting hardware structurally the same. We started with a reformulation from mathematics to Haskell, resulting in a design that closely matches the mathematical description of a particle filter. Only relatively minor adaptations had to be made to the Haskell code before the CλaSH compiler was able to translate the design to synthesizable VHDL code.

Although the process of creating a fully parallel implementation of a particle filter was straightforward, we believe there is room for improvement in the CλaSH compiler. Specifically, we think that the compiler should be able to translate a description of the particle filter with recursive function definitions, given that the recursion is both structural and finite in this case.

The final design of the particle filter is fully parallel, but has a large bottleneck in the resampling step. This bottleneck is the result of the large number of data dependencies present in the computation which are all solved combinatorially. The resampling stage also impedes scaling the design to a larger number of particles, due to large area costs inherent to a fully parallel design. In the future, we will try to pipeline the design so that we can make the design faster. To reduce the area, we will have to make a trade off between area and execution time based on the mathematical structure and formulation in Haskell.

6. ACKNOWLEDGEMENT

We would like to thank Hans Driessen for all the information on particle filtering and Koen Blom for his comments on the mathematical formulation and the writing.

7. REFERENCES

[1] S. P. Jones, Ed., *Haskell 98 Language and Libraries*, ser. Journal of Functional Programming, 2003, vol. 13, no. 1.

[2] C. P. R. Baaij, M. Kooijman, J. Kuper, W. A. Boeijink, and M. E. T. Gerards, "CλaSH: Structural Descriptions of Synchronous Hardware using Haskell," in *Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools, Lille, France*. USA: IEEE Computer Society, September 2010, pp. 714–721.

[3] A. Gill and A. Farmer, "Deriving an efficient FPGA implementation of a low density parity check forward error corrector," in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '11. New York, NY, USA: ACM, 2011, pp. 209–220.

[4] A. Athalye, M. Bolić, S. Hong, and P. M. Djurić, "Generic hardware architectures for sampling and resampling in particle filters," *EURASIP J. Appl. Signal Process.*, pp. 2888–2902, 2005.

[5] R. Gupta and F. Brewer, "High-level synthesis: A retrospective," in *High-Level Synthesis*, P. Coussy and A. Morawiec, Eds. Springer Netherlands, 2008, pp. 13–28, 10.1007/978-1-4020-8588-8_2. [Online]. Available: http://dx.doi.org/10.1007/978-1-4020-8588-8_2

[6] M. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking," *Signal Processing, IEEE Transactions on*, vol. 50, no. 2, pp. 174 –188, feb 2002.

[7] O. Cappe, S. Godsill, and E. Moulines, “An overview of existing methods and recent advances in sequential monte carlo,” *Proceedings of the IEEE*, vol. 95, no. 5, pp. 899 –924, may 2007.

[8] M. Bolić, P. M. Djurić, and S. Hong, “Resampling algorithms for particle filters: a computational complexity perspective,” *EURASIP J. Appl. Signal Process.*, vol. 2004, pp. 2267–2277, January 2004. [Online]. Available: http://dx.doi.org/10.1155/S1110865704405149

[9] J. D. Hol, T. B. Schon, and F. Gustafsson, “On resampling algorithms for particle filters,” in *Nonlinear Statistical Signal Processing Workshop, 2006 IEEE*, sept. 2006, pp. 79 –82.

[10] The GHC Team. The Glasgow Haskell Compiler, 2012. [Online]. Available: http://www.haskell.org/ghc/

[11] “Particle filter, python cookbook, 2012.” [Online]. Available: http://www.scipy.org/Cookbook/ParticleFilter

[12] A. Niedermeier, R. Wester, C. P. R. Baaij, J. Kuper, and G. J. M. Smit, “Comparing cλash and vhdl by implementing a dataflow processor,” in *Proceedings of the Workshop on PROGram for Research on Embedded Systems and Software (PROGRESS 2010), Veldhoven, The Netherlands*. Utrecht: Technology Foundation STW, November 2010, pp. 216–221.