Design space exploration of a particle filter using higher-order functions

Rinse Wester and Jan Kuper

University of Twente, Drienerlolaan 5, Enschede, Netherlands
{r.wester,j.kuper}@utwente.nl

Abstract. This paper presents a design space exploration methodology based on higher-order functions to facilitate the tradeoff between execution time and area usage on FPGAs. Higher-order function are transformed, resulting in parameterized nodes where the amount of parallelism and thereby performance, can be controlled. For composition and scheduling of operations, dataflow principles are used. To show the validity of the approach, a particle filter has been transformed and synthesized for FPGA. The resulting architecture is parameterizable and achieves good performance.

Keywords: Higher-order functions, tradeoff methodology, Particle filter, FPGA

1 Introduction

Particle filtering is a popular, Monte Carlo based technique, to perform state space estimation e.g. tracking [1]. Since particle filtering is computationally intensive, a proper tradeoff between time and space is necessary for FPGa implementation. In this paper, we propose a novel design space exploration methodology that exploits the mathematical structure in particle filters, resulting in a tradeoff between execution-time and FPGA area usage i.e. between time and space. Higher-order functions, a key abstraction technique used in functional programming, are translated into dataflow nodes using transformation rules that perform a tradeoff between time and space.

The tradeoff is explored in a particle filter written in plain Haskell [2] consisting only of normal and higher-order functions (functions that take a function as argument). Using a set of transformation rules, these higher-order functions are transformed into parameterizable CλaSH [3] hardware components. The CλaSH language is a subset of Haskell that is translated to hardware (VHDL) by the CλaSH compiler. To simplify simulation, the particle filter is implemented in both Haskell and CλaSH. Haskell is used for the reference implementation using floating point numbers while CλaSH is used for the actual fixed point hardware implementation. For composition of the resulting CλaSH hardware, dataflow principles are used by adding logic that performs synchronization and scheduling.

The rest of this paper is structured as follows. First related work is presented in Section 2. In Section 3.1, some background information is given on hardware design using the functional language Haskell. Particle filtering is introduced in Section 3.2. The design methodology is presented in Section 4 while simulation and hardware results are given in Section 5. Finally, in Section 6, conclusions are drawn and possible directions for future work are discussed.

2 Related work

Particle filters have become a subject of intensive research since the publication of [1]. Hardware implementations of particle filters using FPGAs for acceleration is extensively covered in [4] and [5] while hardware design methodologies can be found in [6] and [7]. In [6] a generic method is presented to implement different particle filters using a single model. [7] incorporates dataflow principles (data triggered execution) into a particle filtering architecture.

The main difference between the aforementioned papers and the methodology presented in this paper is that the tradeoff is directly applied to the mathematical definition (in Haskell) of a particle filter instead of C source code. As was shown in [8], there exists a one-to-one relation between higher-order functions and the resulting structure of components on the FPGA. It is therefore interesting to explore the transformations of higher-order functions involving a tradeoff between time and space.

A lot of research exists on using functional languages for hardware design [9], [10] including hardwarde design using higher-order functions [11]. However, compared to a direct register transfer level (RTL) approach, the transformations presented in this paper are applied on a higher abstraction level by exploiting the regularity of higher-order functions i.e. the transformations produce RTL style hardware.

3 Background

3.1 Hardware design using Haskell

All designs presented in this paper are written in Haskell or CλaSH. Haskell [2] is a functional language supporting abstraction techniques like type derivation, partial application and higher-order functions. Especially higher-order functions (functions accepting a function as argument or returning a function as result) is a very useful abstraction because it allows the designer to express the mathematical regularity of the application very concisely and semantically clear [8].

To design real hardware we use the functional hardware description language CλaSH, a subset of Haskell that is translated to VHDL by the CλaSH compiler. The language features that make Haskell very attractive for hardware design, like higher-order functions, are also available in CλaSH. Among others, the higher-order functions *map*, *zipWith* and *foldl* are supported by CλaSH, allowing a direct implementation of the components resulting from the design methodology.

In CλaSH, all components are expressed in the form of a Mealy machine (the output and new state are a function of the current state and input). Listing 1.1 shows a small CλaSH code example of a circuit adding all elements in a vector (a list with constant length).

Listing 1.1: CλaSH code example

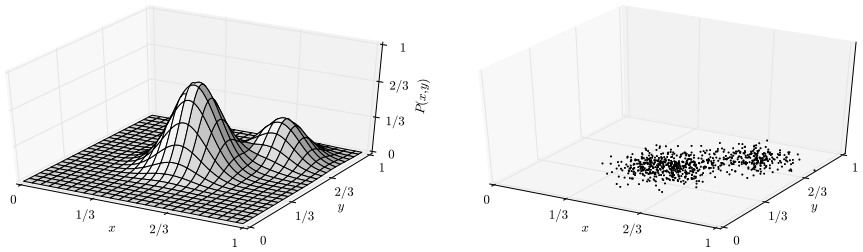
```
sum (State s) xs = (State s', out)
  where
    s'   =  vfoldl (+) 0 xs
    out  =  s
```

As shown Listing 1.1, the function describing the Mealy machine of *sum* accepts two arguments (the current state s and vector of values xs) and returns a new state s' and output *out*. Using the higher-order function *vfoldl*, the sum of the vector xs is determined and assigned to s'. *vfoldl* accepts the binary addition function (+), an initial value 0 and the vector of numbers xs to be summed. *vfoldl* determines the sum incrementally adding elements from xs starting with the initial value 0, thereby forming a chain of adders. In the last line, the value of the internal state register s is assigned to the output *out*, the result is therefore one cycle delayed. More information on hardware design using CλaSH can be found in [3].

3.2 Particle filtering

Particle filtering is a Bayesian filtering technique to estimate the state of a system recursively using noisy measurements [1]. The state of the system is a set of properties that should be tracked, examples are speed, position and angular momentum. For each measurement (a radar image for example), the current estimate of the real state vector is updated resulting in a more and more precise estimate. Since measurements contain noise, the resulting state will be in the form of a Probability Density Function (PDF). Analytically finding this PDF is often mathematically intractable (the integrals can not be solved) which is why approximation methods are used. Particle filters approximate this PDF by a set of N particles $\boldsymbol{x}_k^{(i)}$ where $i = 1 \ldots N$ is the index of a particle and k the iteration of the filter. A higher density of particles represents a higher probability in the continuous state space (Figure 1). We focus on a commonly used type of particle filter, the Sequential Importance Resampling Filter (SIRF) which consists of four steps: *prediction*, *update*, *normalization* and *resampling* [12].

During *prediction*, the next state is derived from the current state using the known dynamics of the system that is being observed. This is implemented by evaluating the system dynamics function f for all N particles, $\boldsymbol{x}_k^{(i)} = f(\boldsymbol{x}_{k-1}^{(i)}, u_k)$. f consist of a deterministic and non-deterministic part. For each particle, the deterministic part depends only on the previous state $\boldsymbol{x}_{k-1}^{(i)}$ while the non-deterministic part u_k requires a sample from a known distribution. For example, a ship moves in a straight line (deterministic) while the position might fluctuate a bit due to waves (non-deterministic).



(a) (b)

Fig. 1: Continous PDF and particle representation

In the *update* step, every particle $\boldsymbol{x}_k^{(i)}$ is assigned a weight $\omega_k^{(i)}$, using the a likelihood function g, representing the importance of a particle given a *measurement* z_k. The function g returns a weight $\omega_k^{(i)}$ given a particle $\boldsymbol{x}_k^{(i)}$, a measurement z_k and optionally noise v_k.

$$\omega_k^{(i)} = g(\boldsymbol{x}_k^{(i)}, z_k, v_k), \quad \text{for} \quad i = 1 \ldots N \tag{1}$$

The remaining two steps in particle filtering are normalization and resampling. During normalization the weights are scaled such that the sum is equal to one, preparing them for resampling. To prevent degeneracy of weights the resampling step replicates particles zero, one or more times depending on their weight $\tilde{\omega}^{(i)}$, while keeping the total number of particles constant i.e. particles with a high weight are replicated while particles with a low weight are discarded. More information on different resampling techniques and hardware implementations can be found in [4].

4 Design methodology

As already elaborated in [8], the whole Haskell description of the particle filter can be divided into two groups, higher-order functions and normal functions. Higher-order functions are used to express structure and repetition with other functions as argument. Normal functions on the other hand are used as discrete components and correspond to combinatorial circuits like an adder for example. We say that a function is normal when the base type contains no function-arguments i.e. the type of every argument and result may not contain an arrow.

The design space exploration methodology consists of three phases: it starts out with 1. a definition of the particle filter in Haskell 2. applying transformation rules to higher-order functions, and 3. composition using dataflow principles.

4.1 Particle filter in Haskell

Throughout this paper, a simple example of a particle filter is used to evaluate the design methodology. This filter performs tracking of a white square moving over

a dark background using 32 particles. Every frame is considered a measurement that is used in a complete cycle of the particle filter. Based on the color of a pixel in this frame pointed at by a particle, a weight is calculated. Pixels with a light color are probably in the square and are therefore assigned a high weight while darker pixels are assigned a low weight.

The square tracking particle filter is first implemented in Haskell to simplify simulation (no low level hardware details like fixed point numbers). Each step (*prediction*, *update*, *normalization* or *resampling*) consists of normal and higher-order functions. Transformations are applied to these higher-order functions such that a tradeoff between time and space is made. The next step is to exploit the regular structure of higher-order functions to derive more efficient hardware.

4.2 Space/time tradeoff rules

Using the CλaSH compiler, higher-order functions like *foldl* can directly be translated to hardware. However, if the lists/vectors being mapped or folded over are very long, a lot of area is needed and the clock frequency will be limited. Therefore, transformation rules are applied to these higher-order functions to limit the area usage and length of the combinatorial paths. These transformation rules reduce the combinatorial complexity by distributing the operations over several clock cycles making a tradeoff between time and space. In the remainder of this section, the focus will be on the higher-order function *foldl* which is used in the normalization step. However, the rules are very similar for the higher-order functions *map*, *zipWith* and *scanl* used in other parts of the particle filter.

Listing 1.2 and Figure 2 show the transformation of *foldl*. First the list to be processed xs is split into P sublists of size M such that $M \times P = N$. Each sublist is processed in a single cycle using $foldl_s$ (space) while the whole list is processed sequentially using $foldl_t$ (time). The definitions of *foldl* are equivalent ($g = h$). The amount of replication on hardware can now be controlled by the parallelization factor M, a parameter introduced by the transformation rule. A larger M results in larger sublists and therefore a higher throughput in a single clock cycle at the cost of more hardware. A smaller M, on the other hand, requires more clock cycles to process the complete list but uses less hardware.

Listing 1.2: Transformation of *foldl*

```
g f xs = foldl f a xs

h f xs = y
  where
    xss = split M xs
    y   = foldl_t (foldl_s f) a xss
```

Figure 2 shows the transformation of *foldl* visually. As shown in Figure 2c, the final architecture requires an additional register to store intermediate results from a previous cycle. Again, the size of the sublists and the amount of parallelism in controlled by M. The aforementioned transformation rules are applied to all higher-order functions in the square tracking filter.

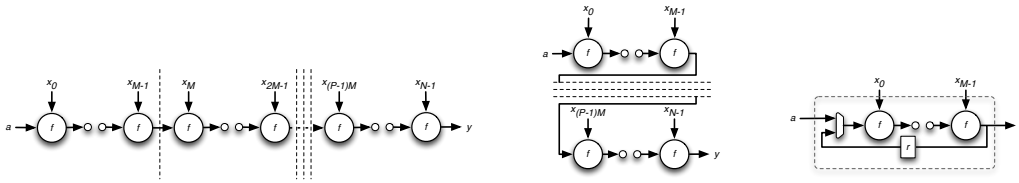


Fig. 2: Transformation of higher-order function *foldl*

4.3 Composition using dataflow

When all higher-order functions are transformed, the resulting components are wrapped into a dataflow node [13] for sycnhronization and scheduling. All these nodes are then connected together using FIFO buffers for storage of intermediate results. The data triggered behavior is implemented using a firing rule (start execution when all required data is available). When a node fires, arguments are removed from the input FIFOs while the result are written into an other FIFO. After all transformations have been applied to the particle filter, a graph is formed consisting of dataflow nodes and FIFOS.

5 Results

check flow and factor M

Before the VHDL generated by CλaSH is synthesized, the design is thoroughly simulated to verify its correctness. Since the CλaSH description of the dataflow particle filter is a valid Haskell program, simulation can be performed by just executing the code. A small framework has been built where a reference particle filter in plain Haskell is compared with the dataflow particle filter implemented using CλaSH. This framework produces a stream of grayscale images (256×256 pixels) for both particle filters to track. For each image, both particle filters generate a set of particles which are averaged to produce an estimate of the square position. The resulting tracks are displayed in Figure 3. Both filters are able to track the square on the Lissajous path within a few pixels. However, the CλaSH particle filter deviates sometimes a few pixels more from the path than the Haskell reference. This is caused by the fixed-point implementation of the arithmetic operations since the dataflow particle filter uses 18 bits to represent particle weight while the plain Haskell implementation uses double precision floating point operations.

The throughput is determined by looking at activity of the *write* signal of the FIFO between the replicator and the predictor.. With parallelization factor M=4, the resampled particles are sent in groups of 8 tokens to the predictor where each token contains 4 particles. Averaging over the differences between arrival times of each first token results in an average cycle time of 69 clock cycles. This cycle time gives a throughput of $32/69 \approx 0.46$ particles per clock cycle.

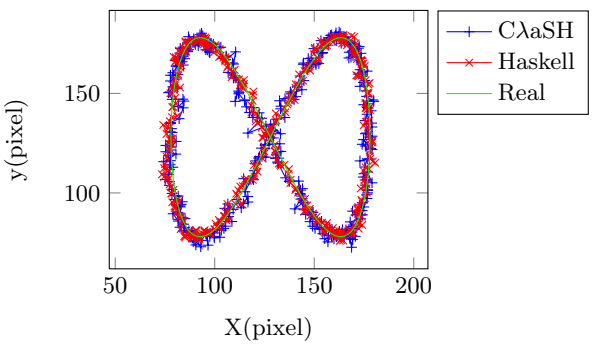


Fig. 3: Tracking of a Lissajous curve

After successfully simulating the particle filter, it has been translated to VHDL by the CλaSH compiler and synthesized for a Virtex 6 XC6VLX240T FPGA on an ML605 development board. Three instantiations are synthesized: a filter with parallelization factor M=2, 4 and 8 respectively. All instantiations are able run at a clock frequency of approximately 25MHz (currently limited reciprocal operation in the normalization step). Table 1 shows the number of LUTs used for the dataflow based particle filter. Figure 4 shows a graphical representation of numbers in Table 1 in which the number of LUTs required scales more or less linear with M. Similarly, M DSP48E1 multipliers are required for each instantiation.

	$M=2$		$M=4$		$M=8$	
Component	**LUTs**	**FFs**	**LUTs**	**FFs**	**LUTs**	**FFs**
Noisegen	70	64	138	128	274	256
Predict	37	-	69	-	133	-
Update	44	28	44	50	61	94
Sum	81	22	116	21	187	20
Recipr	923	-	923	-	923	-
Norm	20	4	29	3	48	2
Ws2Rs	204	30	333	29	592	28
Replicate	70	42	126	76	214	142
FIFOs	5210	4021	4650	3707	4435	3538
Total:	6659	4211	6428	4014	6867	4080

Table 1: Resource usage of dataflow based

Compared to the architectures presented in [5] and [6], the performance of the architecture presented in this paper is in the same order of magnitude. The throughput is also very similar to performance of the fully parallel particle filter in [8] but requires approximately a factor 6 fewer LUTs. Therefore, this design space exploration methodology is adequate for particle filtering.

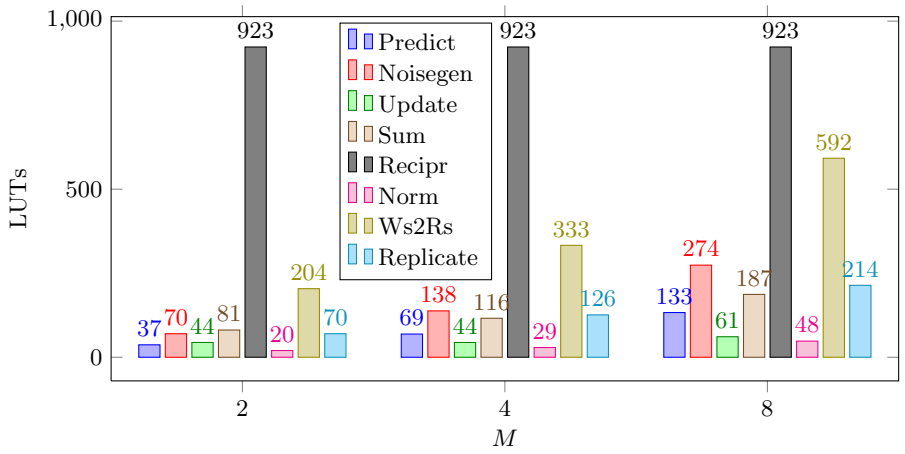


Fig. 4: LUTs used by components of particle filter

6 Conclusions and future work

A design methodology based on transformation of higher-order functions has been presented and applied to a particle filter application. The transformation rules produce dataflow nodes with a parallelization parameter M. By choosing a proper value for M, a tradeoff between execution time and FPGA area is made. For composition of the resulting components, dataflow principles are used. When applied to the particle filter example, the methodology produces scalable hardware in terms of throughput and FPGA area consumption. Higher-order functions are therefore an adequate abstraction level to express mathematical dependencies.

All transformations and implementations of dataflow nodes have now been done by hand, the next step is to automate this. The idea is to develop an embedded language to easily express designs using higher-order functions. A transformation algorithm then applies the transformation rules presented in this paper after which the hardware can be generated using CλaSH.

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

References

1. Arulampalam, M., Maskell, S., Gordon, N., Clapp, T.: A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking. Signal Processing, IEEE Transactions on **50**(2) (feb 2002) 174 –188
2. Jones, S.P., ed.: Haskell 98 Language and Libraries. Volume 13 of Journal of Functional Programming. (2003)

3. Baaij, C.P.R., Kooijman, M., Kuper, J., Boeijink, W.A., Gerards, M.E.T.: 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) 714–721
4. Bolić, M., Djurić, P.M., Hong, S.: Resampling algorithms for particle filters: a computational complexity perspective. EURASIP J. Appl. Signal Process. **2004** (January 2004) 2267–2277
5. Cho, J.U., Jin, S.H., Pham, X.D., Jeon, J.W., Byun, J.E., Kang, H.: A real-time object tracking system using a particle filter. In: Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on. (2006) 2822–2827
6. Saha, S., Bambha, N.K., Bhattacharyya, S.S.: Design and implementation of embedded computer vision systems based on particle filters. Computer Vision and Image Understanding **114**(11) (2010) 1203–1214
7. Hong, S., Liang, X., Djuric, P.: Reconfigurable particle filter design using dataflow structure translation. In: Signal Processing Systems, 2004. SIPS 2004. IEEE Workshop on. (2004) 325–330
8. Wester, R., Baaij, C.P.R., Kuper, J.: A two step hardware design method using CλaSH. In: 22nd International Conference on Field Programmable Logic and Applications, FPL 2012, Oslo, Norway, USA, IEEE Computer Society (August 2012) 181–188
9. Sheeran, M.: mufp, a language for vlsi design. In: Proceedings of the 1984 ACM Symposium on LISP and functional programming. LFP '84, New York, NY, USA, ACM (1984) 104–112
10. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: hardware design in Haskell. In: Proceedings of the third ACM SIGPLAN international conference on Functional programming. ICFP '98, New York, NY, USA, ACM (1998) 174–184
11. Sheeran, M.: Designing regular array architectures using higher order functions. In Jouannaud, J.P., ed.: Functional Programming Languages and Computer Architecture. Volume 201 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1985) 220–237
12. Cappe, O., Godsill, S., Moulines, E.: An overview of existing methods and recent advances in sequential monte carlo. Proceedings of the IEEE **95**(5) (may 2007) 899 –924
13. Lee, E., Messerschmitt, D.: Synchronous data flow. Proceedings of the IEEE **75**(9) (1987) 1235–1245