A two step hardware design method using CλaSH

Rinse Wester, Christiaan Baaij, Jan Kuper

University of Twente, Enschede
Contents

• Introduction

• Background

• Designing method applied to particle filter

• Results

• Conclusions & Future Work
Introduction

- What is C\texttt{\textasciitilde}aSH?
  - Functional Language and Compiler for Concurrent Digital Hardware Design

- Motivation?
  - Evaluate C\texttt{\textasciitilde}aSH and design method on complex application

- Why a particle filter?
  - Covers important aspects of digital hardware design: massive parallelism, feedback loop and data dependent processing.
Background

* C\lambda\text{aSH}

  * A functional language and compiler for digital hardware design

  * On the lowest level, everything is a Mealy machine \( f(s,i) = (s',o) \)

  * A C\lambda\text{aSH} description is purely structural i.e. all operations are performed in a single clock cycle

  * Simulation is cycle accurate
Background

\[
\text{mac (State } s \text{) (} a, b \text{) } = \text{(State } s', \text{ out)} \\
\text{where} \\
\quad s' = s + a \ast b \\
\quad \text{out} = s'
\]
Background

\[ \text{fir } cs \ (\text{State } us) \ \text{inp} = (\text{State } us', \ \text{out}) \]

where

\[ us' = \text{inp} \mathbin{\text{+++}} us \]
\[ ws = \text{vzipWith (*) } us \ cs \]
\[ \text{out} = \text{vfoldl (+) } 0 \ ws \]
State estimation

- Determine $p(x_k \mid z_k)$ recursively with noise
- State variables: position, speed, angle, ...
- Applications: tracking in radar and video

Requirements for estimator

- System dynamics
- Measurement function
Background

- Monte Carlo approximation of \( p(x_k \mid z_k) \) represented by concentration of points (particles)

- Applicable to non-linear, non-gaussian systems (tracking, robotics, ..)

- Parameterizable in and \( N, F_{\text{sys}}(x) \) and \( F_{\text{meas}}(x,m) \)
Background

- Prediction
  - Predict next state based on current
    \( F_{sys}(x) \rightarrow x' \)

- Update
  - Assign weights to particles based on measurement
    \( F_{meas}(x,m) \rightarrow \omega \)

- Normalize such that \( \sum \omega^{(i)} = 1 \)

- Resample

Particle Filter

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 in 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.

The particle filter described in this paper has not been implemented in VHDL. However, based on [6], it is expected that the resulting hardware is very similar. In [6], a dataflow processor has been designed using both VHDL and C-aSH based on a single specification and the same design decisions were made.

Although many particle filters have been built for FPGA, it is very hard to compare them to the results presented in this paper. This is due to the fact that the particle filter described in this paper is fully parallel (all particles are processed in single cycle) while other particle filters usually process a single particle per cycle resulting in very different hardware.
Background

- Prediction
  - Predict next state based on current $F_{sys}(x) \rightarrow x'$
- Update
  - Assign weights to particles based on measurement $F_{meas}(x,m) \rightarrow \omega$
  - Normalize such that $\sum \omega^{(i)} = 1$
- Resample

Particle Filter

1. **Prediction**
   - Predict next state based on current $F_{sys}(x) \rightarrow x'$

2. **Update**
   - Assign weights to particles based on measurement $F_{meas}(x,m) \rightarrow \omega$
   - Normalize such that $\sum \omega^{(i)} = 1$

3. **Resample**
   - Measurement
   - Predict
   - Update
   - Normalize
   - Resample
Background

- Prediction
  - Predict next state based on current \( F_{sys}(x) \rightarrow x' \)

- Update
  - Assign weights to particles based on measurement \( F_{meas}(x,m) \rightarrow \omega \)
  - Normalize such that \( \sum \omega^{(i)} = 1 \)

- Resample

Particle Filter

1. Related work
2. Background
3. Particle Filtering

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 in 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.

The particle filter described in this paper has not been implemented in VHDL. However, based on [6], it is expected that the resulting hardware is very similar. In [6], a dataflow processor has been designed using both VHDL and C-aSH based on a single specification and the same design decisions were made.

Although many particle filters have been built for FPGA, it is very hard to compare them to the results presented in this paper. This is due to the fact that the particle filter described in this paper is fully parallel (all particles are processed in single cycle) while other particle filters usually process a single particle per cycle resulting in very different hardware.
**Prediction**

- Predict next state based on current 
  \[ F_{sys}(x) \rightarrow x' \]

**Update**

- Assign weights to particles based on measurement 
  \[ F_{meas}(x,m) \rightarrow \omega \]

- Normalize such that \[ \sum \omega^{(i)} = 1 \]

**Resample**
Background

• Prediction
  • Predict next state based on current
    \( F_{sys}(x) \rightarrow x' \)

• Update
  • Assign weights to particles based on measurement
    \( F_{meas}(x,m) \rightarrow \omega \)

• Normalize such that \( \sum \omega^{(i)} = 1 \)

• Resample
Background Mathematical formulation of Particle Filter

* Prediction
  \[ x_k^{(i)} \sim p(x_k|x_{k-1}) \]

* System dynamics function
  \[ x_k^{(i)} = f(x_{k-1}^{(i)}, u_k) \]

* Update
  \[ \omega_k^{(i)} = p(z_k|x_k^{(i)}) \]

* Measurement function
  \[ \omega_k^{(i)} = g(x_k^{(i)}, z_k, v_k), \text{ for } i = 1 \ldots N \]

* Normalize
  \[ \tilde{\omega}^{(i)} = \frac{\omega^{(i)}}{\sum_{n=1}^{N} \omega^{(n)}} \text{ for } i = 1 \ldots N \]

* Resample
  \[ \{ \tilde{x}_k^{(1)}, \tilde{x}_k^{(2)} \ldots \tilde{x}_k^{(N)} \} = \big\| \text{replicate}(x_k^{(i)}, r_i) \big\|_{n=1}^{N} \]
Background

Simple tracking application
Background

Simple tracking application

- Tracking a square on a dark background
Background

Simple tracking application

• Tracking a square on a dark background

• Particle: $X^{(i)} = <x, y, \omega>$
**Background**

- Tracking a square on a dark background
  - Particle: $X^{(i)} = <x, y, \omega>$
  - System dynamics
Background

Simple tracking application

- Tracking a square on a dark background
  - Particle: $X^{(i)} = <x, y, \omega>$

- System dynamics
  - $(x', y') = (x + \delta_x, y + \delta_y)$
    where $\delta_x, \delta_y \sim U(-a,a)$
Tracking a square on a dark background

Particle: $X^{(i)} = <x, y, \omega>$

System dynamics

$(x', y') = (x + \delta_x, y + \delta_y)$
where $\delta_x, \delta_y \sim \mathcal{U}(-a, a)$

Measurement function
Background

• Tracking a square on a dark background

• Particle: $X^{(i)} = <x, y, \omega>$

• System dynamics

• $(x', y') = (x + \delta_x, y + \delta_y)$
  where $\delta_x, \delta_y \sim U(-a, a)$

• Measurement function

• $\omega = 1 / (1 + (255-\text{pxl})^2)$
Background

- Tracking a square on a dark background
  - Particle: \( X^{(i)} = \langle x, y, \omega \rangle \)
- System dynamics
  - \((x', y') = (x + \delta_x, y + \delta_y)\)
    where \(\delta_x, \delta_y \sim U(-a,a)\)
- Measurement function
  - \(\omega = 1 / (1 + (255-\text{pxl})^2)\)
Implementing the particle filter

- Design method
- Math to Haskell
- Haskell to C\texttt{aSH}
Design method

- First step
  - Reformulate the mathematics of Particle filtering into plain Haskell

- Second step
  - Apply small modifications to Haskell code such that it is accepted by the CλaSH compiler

![Diagram showing the design method flow from mathematics to hardware with Haskell and CλaSH as intermediate steps.](image)
Math to Haskell

- Apply the state space model to all particles
  \[
  x_k^{(i)} = f(x_{k-1}^{(i)}, u_k).
  \]

- All operations are performed independently
  \[
  f(x_k^{(i)}, u_k) = x_k^{(i)} + u_k
  \]

- Corresponding higher order function is `zipWith`

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

```haskell
f (x, y, \omega) (\delta_x, \delta_y) = (x', y', \omega)
where
  x' = x + \delta_x
  y' = y + \delta_y
```
Math to Haskell

- Determine sum of weights and apply to all particles
- Corresponding higher order functions are *foldl* and *zipWith*

\[
\bar{\omega}(i) = \frac{\omega(i)}{\sum_{n=1}^{N} \omega(n)} \quad \text{for} \quad i = 1 \ldots N
\]

```haskell
normalize ps = ps'
where
  tot\omega = sum (map weight ps)
  ps' = map (\(x, y, \omega\) \rightarrow (x, y, \omega / tot\omega)) ps
```
Haskell to C\text\alpha aSH

* Translate lists to Vectors

\[
predict \:: (Ptl \to Ns \to Ptl) \to [Ptl] \to [Ns] \to [Ptl]
predict \ f \quad ps \quad us = ps'
\]
\[
\text{where}
\]
\[
ps' = \text{zipWith} \ f \ ps \ us
\]

\[
predict \:: (Ptl \to Ns \to Ptl) \to (\text{Vector} \ D32 \ Ptl) \to (\text{Vector} \ D32 \ Ns) \to (\text{Vector} \ D32 \ Ptl)
predict \ f \quad ps \quad us = ps'
\]
\[
\text{where}
\]
\[
ps' = \text{vzipWith} \ f \ ps \ us
\]
Haskell to CλaSH

* Translate lists to Vectors

\[ \text{predict} :: (Ptl \rightarrow Ns \rightarrow Ptl) \rightarrow (\text{Vector D32 Ptl}) \rightarrow (\text{Vector D32 Ns}) \rightarrow (\text{Vector D32 Ptl}) \]

\[ \text{predict } f \]

\[ \text{where} \]

\[ ps' = \text{vzipWith } f \ p s \ u s \]
Haskell to CλaSH

- Translate lists to Vectors
- Use fixed point representation for weights

```haskell
normalize :: [Ptl] → [Ptl]
normalize ps = ps'
where
  totω = sum (map weight ps)
  ps = map (λ (x, y, ω) → (x, y, ω / totω)) ps
```

```haskell
normalize :: (Vector D32 Ptl) → (Vector D32 Ptl)
normalize ps = ps'
where
  totω = sum (vmap weight ps)
  totω_recip = fprecip totω
  ps = vmap (λ (x, y, ω) → (x, y, ω * totω_recip)) ps
```
Haskell to \texttt{C\&aSH}

\begin{itemize}
\item Translate lists to Vectors
\item Use fixed point representation for weights
\end{itemize}

\begin{verbatim}
normalize :: [Ptl] -> [Ptl]
normalize ps  = ps'
  where
    totomega = sum (map weight ps)
    ps      = map (\(x, y, \omega\) -> (x, y, \omega / totomega)) ps
\end{verbatim}

\begin{verbatim}
normalize :: (Vector D32 Ptl) -> (Vector D32 Ptl)
normalize ps  = ps'
  where
    totomega    = sum (vmap weight ps)
    totomega/reci = fprecip totomega
    ps      = vmap (\(x, y, \omega\) -> (x, y, \omega * totomega/reci)) ps
\end{verbatim}

Thursday, September 13, 12
Haskell to CλaSH

- Translate lists to Vectors
- Use fixed point representation for weights

```haskell
normalize :: [Ptl] → [Ptl]
normalize ps = ps'
where
  totω = sum (map weight ps)
  ps = map (λ (x, y, ω) → (x, y, ω / totω)) ps

normalize :: (Vector D32 Ptl) → (Vector D32 Ptl)
normalize ps = ps'
where
  totω = sum (vmap weight ps)
  totω_recip = fprecip totω
  ps = vmap (λ (x, y, ω) → (x, y, ω * totω_recip)) ps
```
Haskell to CλaSH

- Translate lists to Vectors
- Use fixed point representation for weights

```haskell
normalize :: [Ptl] → [Ptl]
normalize ps = ps'
  where
    totω = sum (map weight ps)
    ps = map (λ (x, y, ω) → (x, y, ω / totω)) ps
```

```haskell
normalize :: (Vector D32 Ptl) → (Vector D32 Ptl)
normalize ps = ps'
  where
    totω = sum (vmap weight ps)
    totω_recip = fprecip totω
    ps = vmap (λ (x, y, ω) → (x, y, ω * totω_recip)) ps
```
Results

- Parallel particle filter with 32 particles synthesized for FPGA
- Area = about 40k LUTs
- PF can be synthesized but is slow
- Resampling step is bottleneck in both area and clock frequency
- For larges PFs, we need a trade off between area and execution time
Conclusions

- A completely parallel Particle Filter has been implemented
- Higher order functions are a natural way to reason about structure in both the mathematical formulation and hardware
- Haskell code needs only small modifications before it is accepted by the ClaSH compiler
- Fully parallel resampling is a bottleneck in both area and clock frequency
Future Work

- Extend particle filter to more particles and more complex tracking
- Develop area vs time time trade off based on functional description
Questions?