From Functional Programs to Pipelined Dataflow Networks

Richard Townsend
Martha A. Kim     Stephen A. Edwards

Columbia University

IBM PL Day, December 5, 2016
The Future of Hardware

40 Years of Microprocessor Trend Data

Transistors (thousands)
Single-Thread Performance (SpecINT x 10^3)
Frequency (MHz)
Typical Power (Watts)
Number of Logical Cores

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Designing Specialized Hardware
Designing Specialized Hardware

Stage in the Design Process

Abstraction

Combination logic

Register
Designing Specialized Hardware
Designing Specialized Hardware

Abstraction

Stage in the Design Process

Programming
Designing Specialized Hardware

Abstraction

Stage in the Design Process
Overview

Floh (Functional Language on Hardware)

```
data List = Nil | Cons Int List

recMath :: List → Int → Int
recMath l x =
  case l of
    Nil → add x 1
    Cons y xs → recMath xs (mul x y)
```
Floh (Functional Language on Hardware)

```haskell
data List = Nil | Cons Int List

recMath :: List → Int → Int
recMath l x =
  case l of
    Nil      → add x 1
    Cons y xs → recMath xs (mul x y)
```

- True recursion → stack
Floh (Functional Language on Hardware)

```haskell
data List = Nil Go | Cons Int List
data Go = Go

recMath :: List -> Int -> Go -> Int
recMath l x g =
  case l of
    Nil _ -> add x (1 g)
    Cons y xs -> recMath xs (mul x y) g
```

- True recursion → stack
- Trigger constants with Go
Floh (Functional Language on Hardware)

```
data List = Nil Go | Cons Int ListPtr
data Go = Go
data ListPtr = ListPtr Int

recMath :: ListPtr → Int → Go → Int
recMath lp x g =
    case readList lp of
        Nil _ → add x (1 g)
        Cons y xs → recMath xs (mul x y) g
```

- True recursion → stack
- Trigger constants with Go
- Recursive types → explicit memory operations
Flosh (Functional Language on Hardware)

```haskell
data List = Nil Go | Cons Int ListPtr
data Go = Go
data ListPtr = ListPtr Int

recMath :: ListPtr → Int → Go → Int
recMath lp x g =
  case readList lp of
    Nil _ → add x (1 g)
    Cons y xs → recMath xs (mul x y) g
```

- True recursion $\rightarrow$ stack
- Trigger constants with Go
- Recursive types $\rightarrow$ explicit memory operations
Floh (Functional Language on Hardware)

```
data List  = Nil Go | Cons Int ListPtr
data Go    = Go
data ListPtr = ListPtr Int

recMath :: ListPtr → Int → Go → Int
recMath lp x g =
    case readList lp of
        Nil _ → add x (1 g)
        Cons y xs → recMath xs (mul x y) g
```

- True recursion → stack
- Trigger constants with Go
- Recursive types → explicit memory operations

Strictness Policies
Floh (Functional Language on Hardware)

```haskell
data List = Nil Go | Cons Int ListPtr
data Go = Go
data ListPtr = ListPtr Int

recMath :: ListPtr → Int → Go → Int
recMath lp x g =
  case readList lp of
    Nil _ → add x (1 g)
    Cons y xs → recMath xs (mul x y) g
```

- True recursion → stack
- Trigger constants with Go
- Recursive types → explicit memory operations

Strictness Policies
- Data Constructors: strict – evaluate all arguments
Floh (Functional Language on Hardware)

```haskell
data List = Nil Go | Cons Int ListPtr
data Go = Go
data ListPtr = ListPtr Int

recMath :: ListPtr → Int → Go → Int
dercMath lp x g =
    case readList lp of
    Nil _ → add x (1 g)
    Cons y xs → recMath xs (mul x y) g
```

- True recursion → stack
- Trigger constants with \texttt{Go}
- Recursive types → explicit memory operations


Strictness Policies

- Data Constructors: \texttt{strict} – evaluate all arguments
- Functions: \texttt{non-strict} – evaluate first argument
Floh (Functional Language on Hardware)

data List = Nil Go | Cons Int ListPtr

data Go = Go

data ListPtr = ListPtr Int

recMath :: ListPtr → Int → Go → Int
recMath lp x g =
    case readList lp of
    Nil _ → add x (1 g)
    Cons y xs → recMath xs (mul x y) g

Strictness Policies

- Data Constructors: **strict** – evaluate all arguments
- Functions: **non-strict** – evaluate first argument
- Enables pipeline parallelism!
From Floh to Dataflow

\[
\text{recMath \ lp \ x \ g} = \begin{cases} 
\text{readList \ lp} & \text{Nil \ } \_ \rightarrow \text{add \ } x \ (1 \ \text{g}) \\
\text{Cons \ } y \ \text{xs} & \rightarrow \\
\text{recMath \ xs} & \text{(mul \ } x \ y) \ \text{g}
\end{cases}
\]
recMath lp x g =
  case readList lp of
    Nil _ → add x (1 g)
    Cons y xs →
      recMath xs (mul x y) g
From Floh to Dataflow

\[
\text{recMath } lp \times g = \\
\text{case } \text{readList } lp \text{ of } \\
\text{Nil } \_ \rightarrow \text{add } x \ (1 \ g) \\
\text{Cons } y \ xs \rightarrow \\
\text{recMath } xs \ (\text{mul } x \ y) \ g
\]
From Floh to Dataflow

\[
\text{recMath } lp \times g = \\
\quad \text{case readList } \; lp \; \text{of} \\
\quad \quad \text{Nil } _{\_} \rightarrow \text{add } x \; (1 \; g) \\
\quad \quad \text{Cons } y \; xs \rightarrow \\
\quad \quad \quad \text{recMath } xs \; (\text{mul } x \; y) \; g
\]
recMath \( lp \times g \) =
\[\text{case } \text{readList} \ \text{lp} \ \text{of} \]
\[\text{Nil } _{-} \rightarrow \text{add} \times (1 \ g) \]
\[\text{Cons } y \ \text{xs} \rightarrow \]
\[\text{recMath} \ \text{xs} \ (\text{mul} \times y) \ g\]
From Floh to Dataflow

\[
\text{recMath } lp \times g = \begin{cases} 
\text{readList } lp & \text{of} \\
\text{Nil } _{} & \rightarrow \text{add } x \times (1 \ g) \\
\text{Cons } y \ \text{xs} & \rightarrow \\
\text{recMath } \text{xs} \ (\text{mul } x \ y) \ g
\end{cases}
\]
From Floh to Dataflow

```
recMath lp x g =
  case readList lp of
  Nil _  →  add x (1 g)
  Cons y xs →
       recMath xs (mul x y) g
```
From Floh to Dataflow

```
recMath lp x g =
  case readList lp of
    Nil _ → add x (1 g)
    Cons y xs →
      recMath xs (mul x y) g
```
recMath lp x g =
  case readList lp of
    Nil _   → add x (1 g)
    Cons y xs →
      recMath xs (mul x y) g
From Floh to Dataflow

\[
\text{recMath} \; \text{l}p \; x \; g = \begin{cases} 
\text{add} \; x \; (1 \; g) & \text{Nil} \\
\text{Cons} \; y \; x s \rightarrow 
\text{recMath} \; x s \; (\text{mul} \; x \; y) \; g & \text{Cons} 
\end{cases}
\]
From Floh to Dataflow

\[
\text{recMath } lp \times g = \begin{cases} 
\text{add } x \times (1 + g) & \text{Nil} \\
\text{recMath } xs \times (\text{mul } x \times y) \times g & \text{Cons } y \times xs
\end{cases}
\]
From Floh to Dataflow

\[
\text{recMath } lp \times g = \begin{cases} \text{add } x (1 \ g) & \text{if Nil} \\ \text{recMath } xs (\text{mul } x \ y) \ g & \text{if Cons } y \ xs \end{cases}
\]
From Floh to Dataflow

\[
\text{recMath } \text{lp} \times \text{g} = \begin{cases} 
\text{readList } \text{lp} \ 	ext{of} \\
\text{Nil } _{-} \rightarrow \text{add } \times (1 \text{ g}) \\
\text{Cons } y \text{ xs} \rightarrow \text{recMath } \text{xs}(\text{mul } x y) \text{ g}
\end{cases}
\]
recMath lp x g =
  case readList lp of
    Nil _ → add x (1 g)
    Cons y xs →
      recMath xs (mul x y) g
From Floh to Dataflow

recMath lp x g = case readList lp of
  Nil _ → add x (1 g)
  Cons y xs → recMath xs (mul x y) g
From Floh to Dataflow

\[
\text{recMath} \ lp \ x \ g = \begin{cases} 
\text{case} \ \text{readList} \ lp \ \text{of} \\
\text{Nil} \ _ \ \rightarrow \ \text{add} \ x \ (1 \ g) \\
\text{Cons} \ y \ xs \ \rightarrow \ \text{recMath} \ xs \ (\text{mul} \ x \ y) \ g
\end{cases}
\]
From Floh to Dataflow

\[
\text{recMath } l p \times g = \begin{cases} 
\text{case } \text{readList } l p \text{ of} \\
\text{Nil } _- \rightarrow \text{add } x (1 \ g) \\
\text{Cons } y \ xs \rightarrow \\
\text{recMath } xs (\text{mul } x \ y) \ g
\end{cases}
\]
From Floh to Dataflow

\[ \text{recMath} \ l p \times g = \begin{cases} \text{add} \ x \ (1 \ g) & \text{Nil} \ _ \ \rightarrow \ x \ \text{Nil} \ Cons \ g \ \rightarrow \ \text{recMath} \ xs \ (\text{mul} \ x \ y) \ g \\ \end{cases} \]
From Floh to Dataflow

\[ \text{recMath } lp \times g = \begin{cases} \text{case } \text{readList } lp \text{ of} \\
\text{Nil } \_ \rightarrow \text{add } x \times (1 \ g) \\
\text{Cons } y \ x s \rightarrow \text{recMath } x s \times (\text{mul } x \ y) \ g \\
\end{cases} \]
From Floh to Dataflow

\[
\text{recMath } lp \times g = \left\{ \begin{array}{ll}
\text{Nil } & \rightarrow \text{add } x \times (1 \ g) \\
\text{Cons } y \ xs & \rightarrow \text{recMath } xs \ (\text{mul } x \ y) \ g
\end{array} \right.
\]
recMath lp x g =
  case readList lp of
    Nil _ → add x (1 ^ g)
    Cons y xs → recMath xs (mul x y) g
From Floh to Dataflow

$$\text{recMath } lp \times g =$$

\[
\text{case } \text{readList } lp \text{ of } \\
\text{Nil } \_ \rightarrow \text{add } x (1 \ g) \\
\text{Cons } y \ xs \rightarrow \text{recMath } xs (\text{mul } x \ y) \ g
\]
From Floh to Dataflow

\[
\text{recMath } lp \times g = \begin{cases} 
\text{case readList } lp \ of \\
\Nil \_ & \rightarrow \text{add } x \ (1 \ g) \\
\text{Cons } y \ xs & \rightarrow \text{recMath } xs \ (\text{mul } x \ y) \ g 
\end{cases}
\]
Non-strictness Exploits Pipeline Parallelism

Completion Cycles (Relative to Strict)
Non-Strict with Finite Buffers
Non-strict with Infinite FIFOs

TreeMapMergeSortMapFilterDFSAppend

× speedup
× speedup
Non-strictness Exploits Pipeline Parallelism

Graph showing the completion cycles of various operations:

- **Non-Strict with Finite Buffers**
- **Non-strict with Infinite FIFOs**

Key findings:

- **1.3× speedup** for certain operations.
- **2× speedup** for other operations.

Operations compared: Append, DFS, Filter, Map, MergeSort, TreeMap.
Non-strictness Exploits Pipeline Parallelism

Completion Cycles (Relative to Strict)

- Non-Strict with Finite Buffers
- Non-strict with Infinite FIFOs

- 1.3× speedup
- 2× speedup

Tasks:
- Append
- DFS
- Filter
- Map
- MergeSort
- TreeMap