Optimizing Stream Programs Using Linear State Space Analysis

Sitij Agrawal[1,2], William Thies[1], and Saman Amarasinghe[1]

[1]Massachusetts Institute of Technology
[2]Sandbridge Technologies

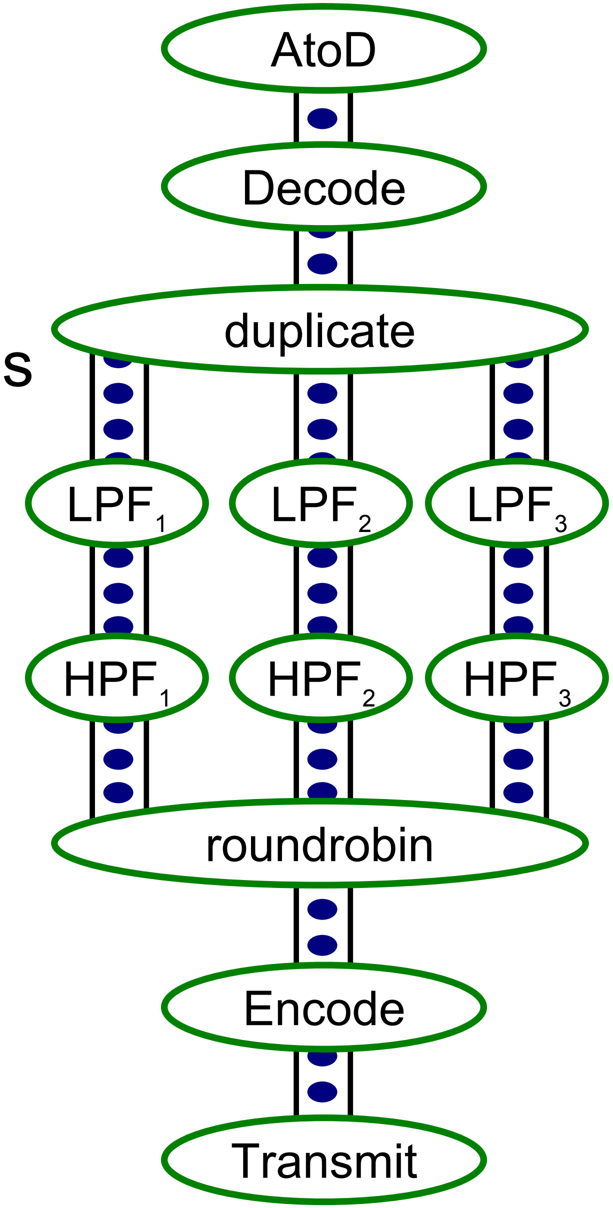
CASES 2005

StreamIt

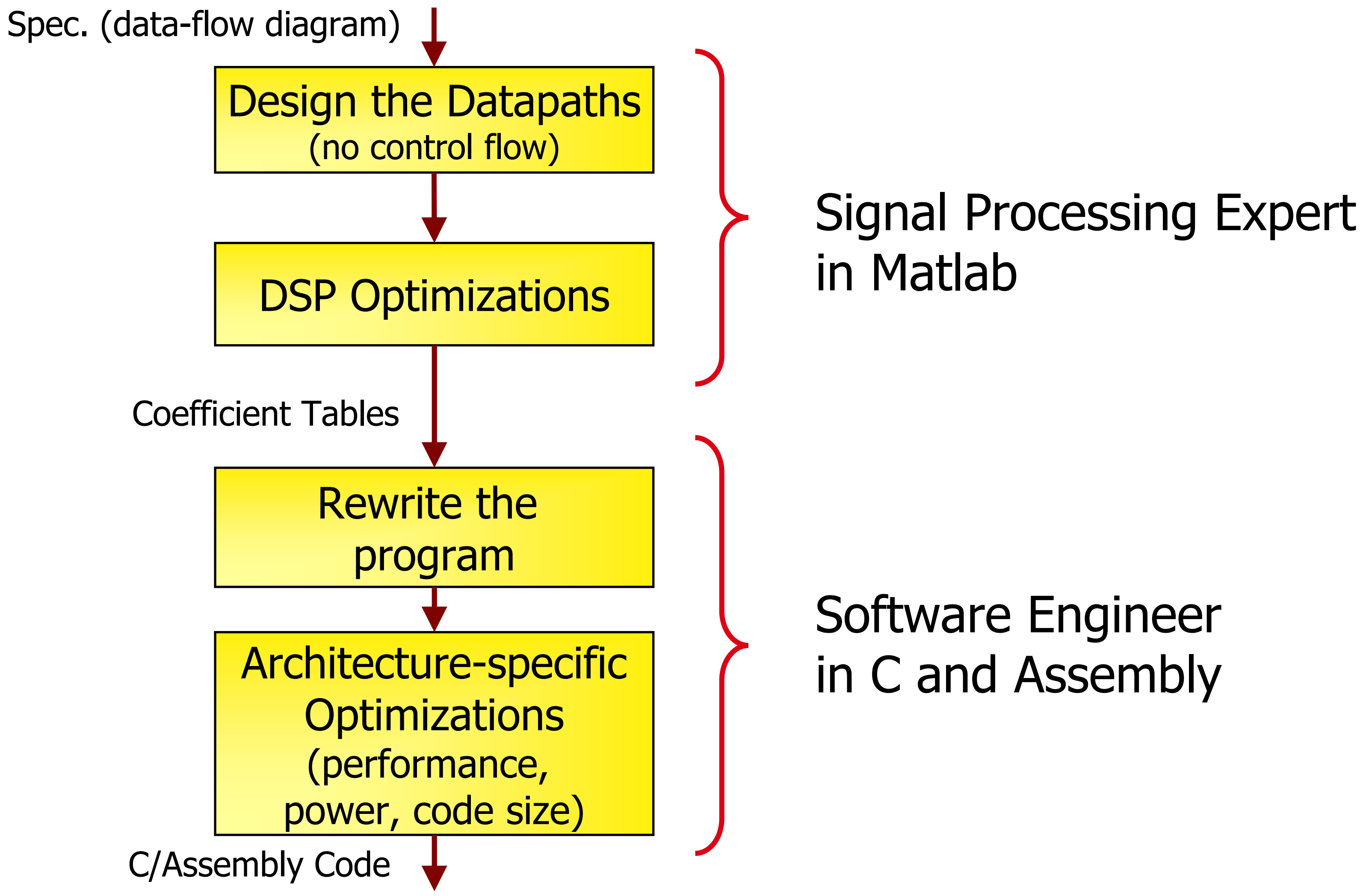
http://cag.lcs.mit.edu/streamit

Streaming Application Domain

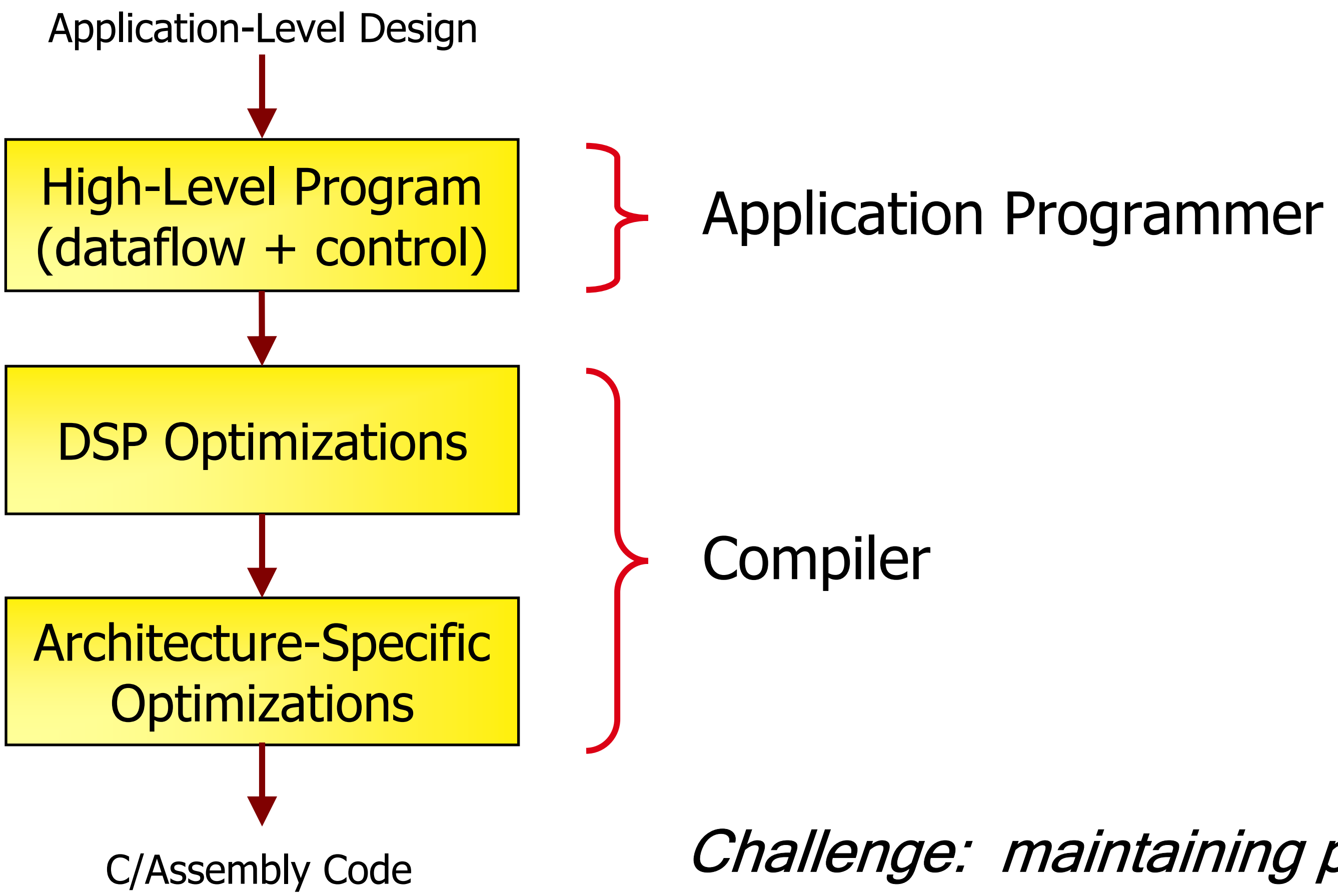
- Based on a stream of data
  - Graphics, multimedia, software radio
  - Radar tracking, microphone arrays, HDTV editing, cell phone base stations
- Properties of stream programs
  - Regular and repeating computation
  - Parallel, independent actors with explicit communication
  - Data items have short lifetimes



Conventional DSP Design Flow



Ideal DSP Design Flow



Challenge: maintaining performance

The StreamIt Language

- Goals:
  - Provide a high-level stream programming model
  - Invent new compiler technology for streams

- Contributions:
  - Language design [CC ’02, PPoPP ’05]
  - Compiling to tiled architectures [ASPLOS ’02, ISCA ’04, Graphics Hardware ’05]
  - Cache-aware scheduling [LCTES ’03, LCTES ’05]
  - Domain-specific optimizations [PLDI ’03, CASES ‘05]

Programming in StreamIt

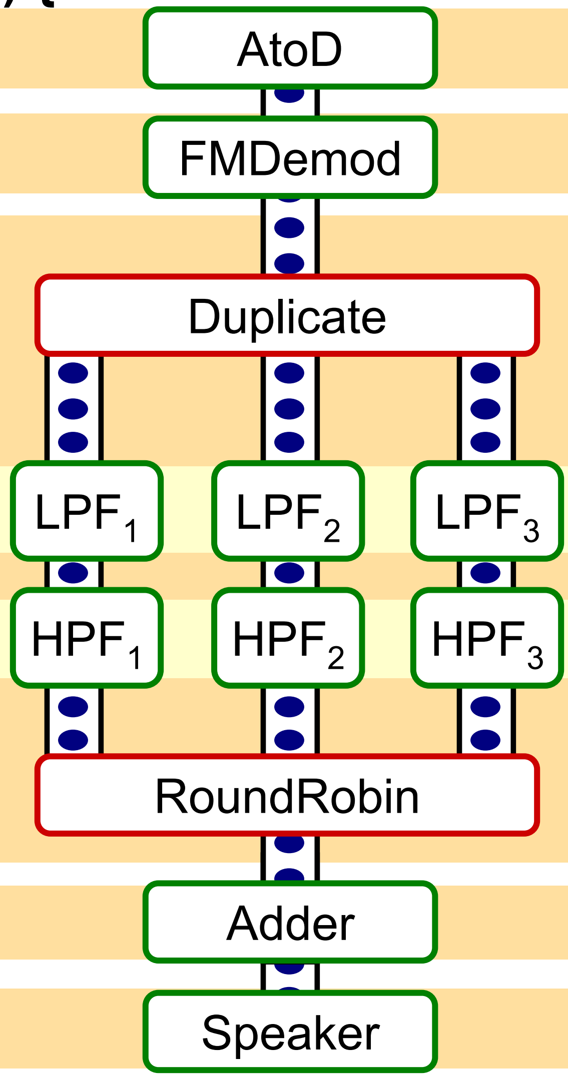
```
void->void pipeline FMRadio(int N, float lo, float hi) {
  add AtoD();

  add FMDemod();

  add splitjoin {
    split duplicate;
    for (int i=0; i<N; i++) {
      add pipeline {
        add LowPassFilter(lo + i*(hi - lo)/N);

        add HighPassFilter(lo + i*(hi - lo)/N);
      }
    }
    join roundrobin();
  }
  add Adder();

  add Speaker();
}
```

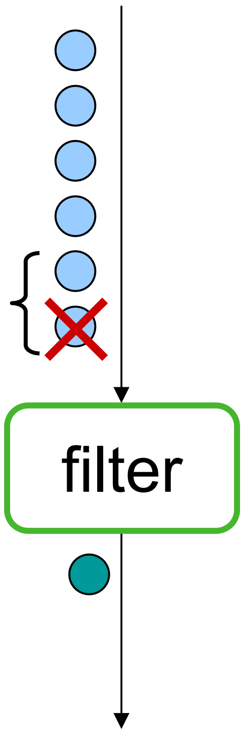


Example StreamIt Filter

```
float->float filter LowPassButterWorth (float sampleRate, float cutoff) {
  float coeff;
  float x;

  init {
      coeff = calcCoeff(sampleRate, cutoff);
  }

  work peek 2 push 1 pop 1 {
     x = peek(0) + peek(1) + coeff * x;
     push(x);
     pop();
  }
}
```



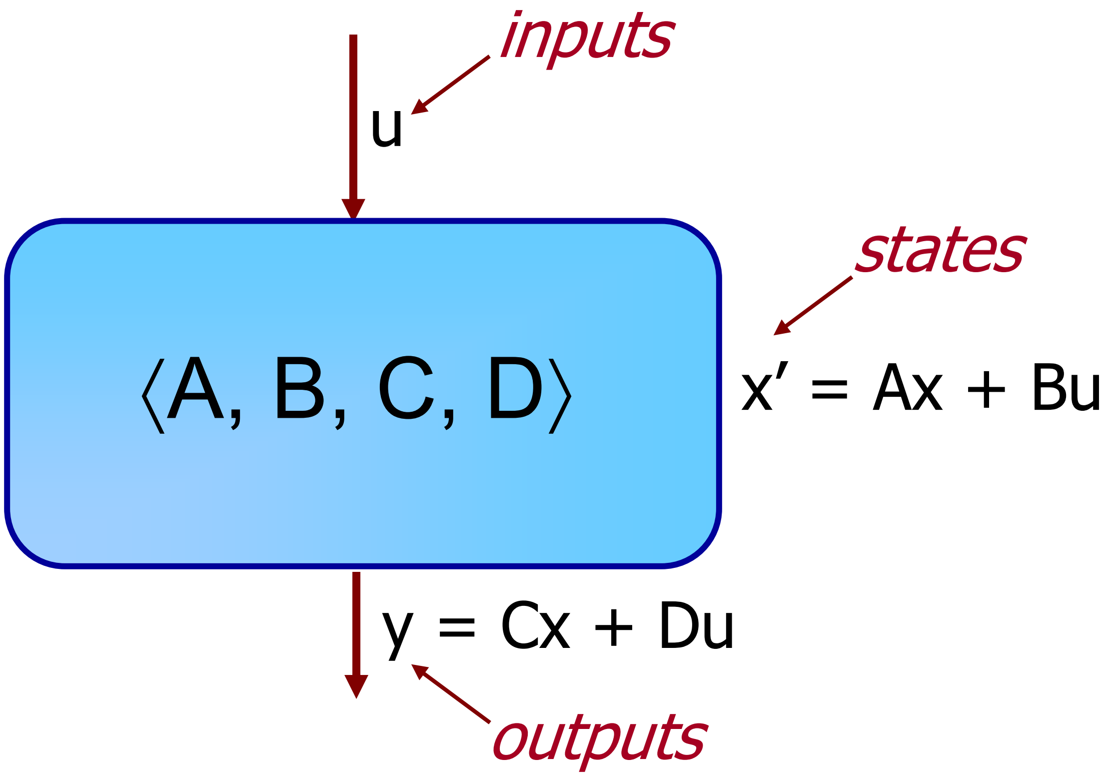
Focus: Linear State Space Filters

- Properties:
  1. Outputs are linear function of inputs and states
  2. New states are linear function of inputs and states

- Most common target of DSP optimizations
  - FIR / IIR filters
  - Linear difference equations
  - Upsamplers / downsamplers
  - DCTs

Representing State Space Filters

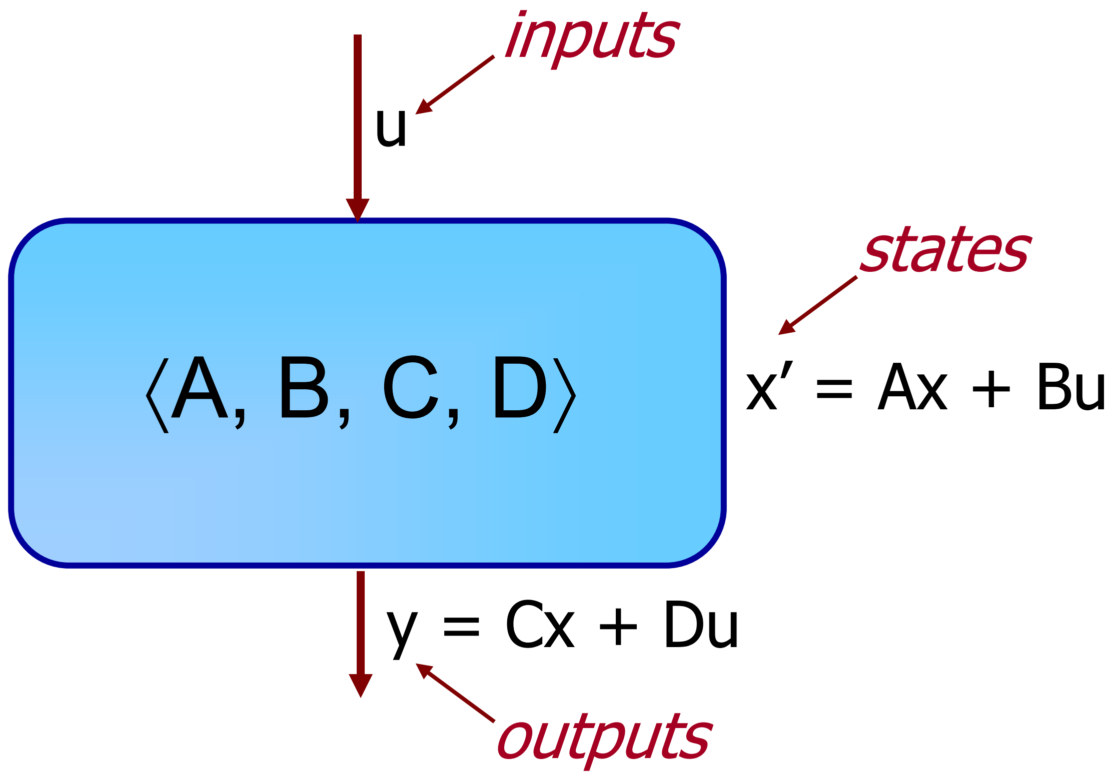
- A state space filter is a tuple $\langle A, B, C, D \rangle$



Representing State Space Filters

- A state space filter is a tuple ⟨A, B, C, D⟩

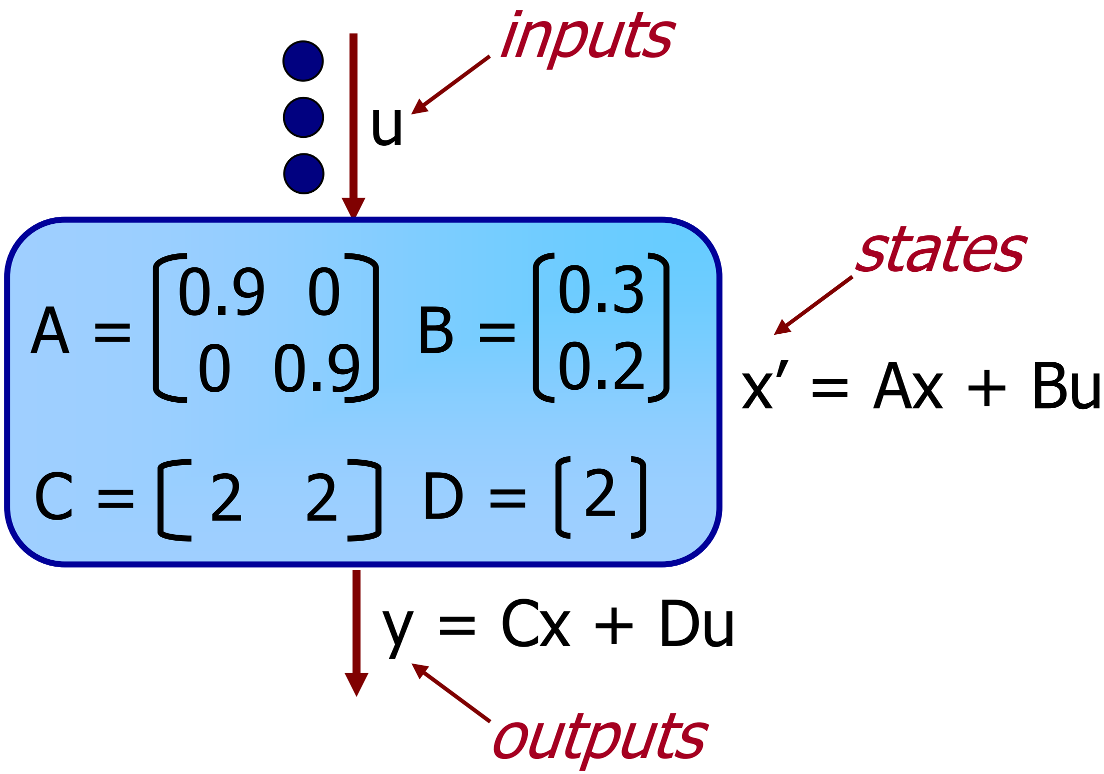
```
float->float filter IIR {
  float x1, x2;
  work push 1 pop 1 {
    float u = pop();
    push(2*(x1+x2+u));
    x1 = 0.9*x1 + 0.3*u;
    x2 = 0.9*x2 + 0.2*u;
} }
```



Representing State Space Filters

- A state space filter is a tuple $\langle A, B, C, D \rangle$

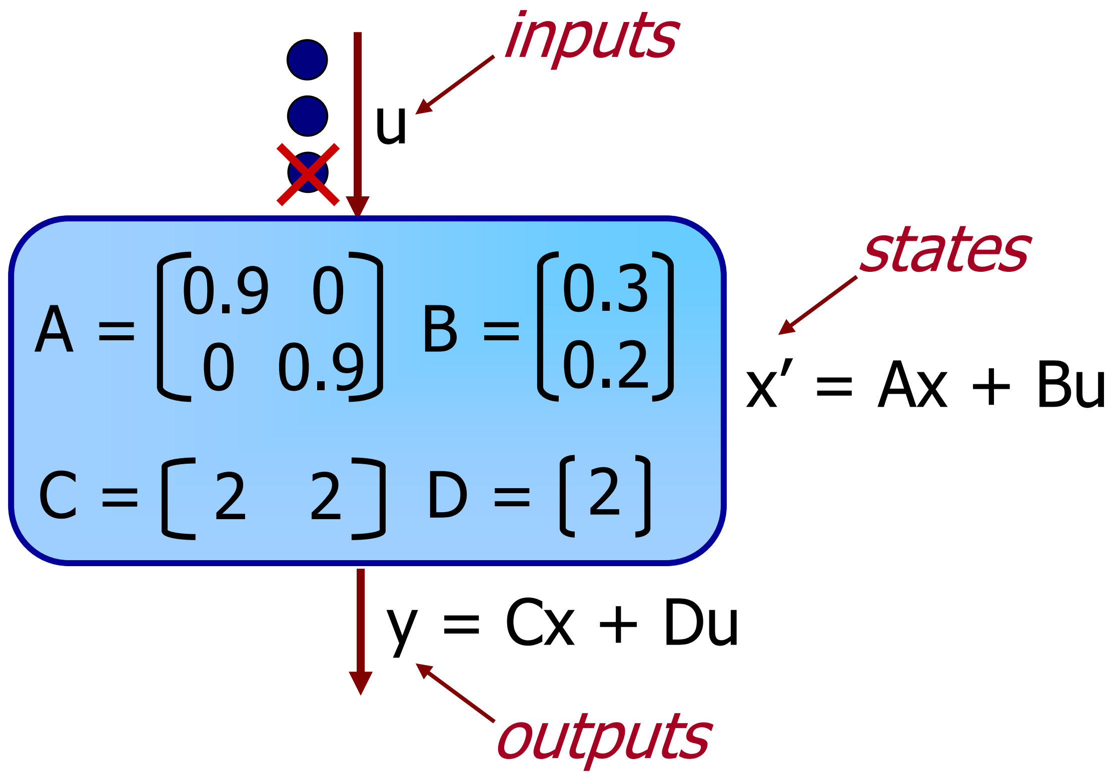
```
float->float filter IIR {
  float x1, x2;
  work push 1 pop 1 {
    float u = pop();
    push(2*(x1+x2+u));
    x1 = 0.9*x1 + 0.3*u;
    x2 = 0.9*x2 + 0.2*u;
} }
```



Representing State Space Filters

- A state space filter is a tuple ⟨A, B, C, D⟩

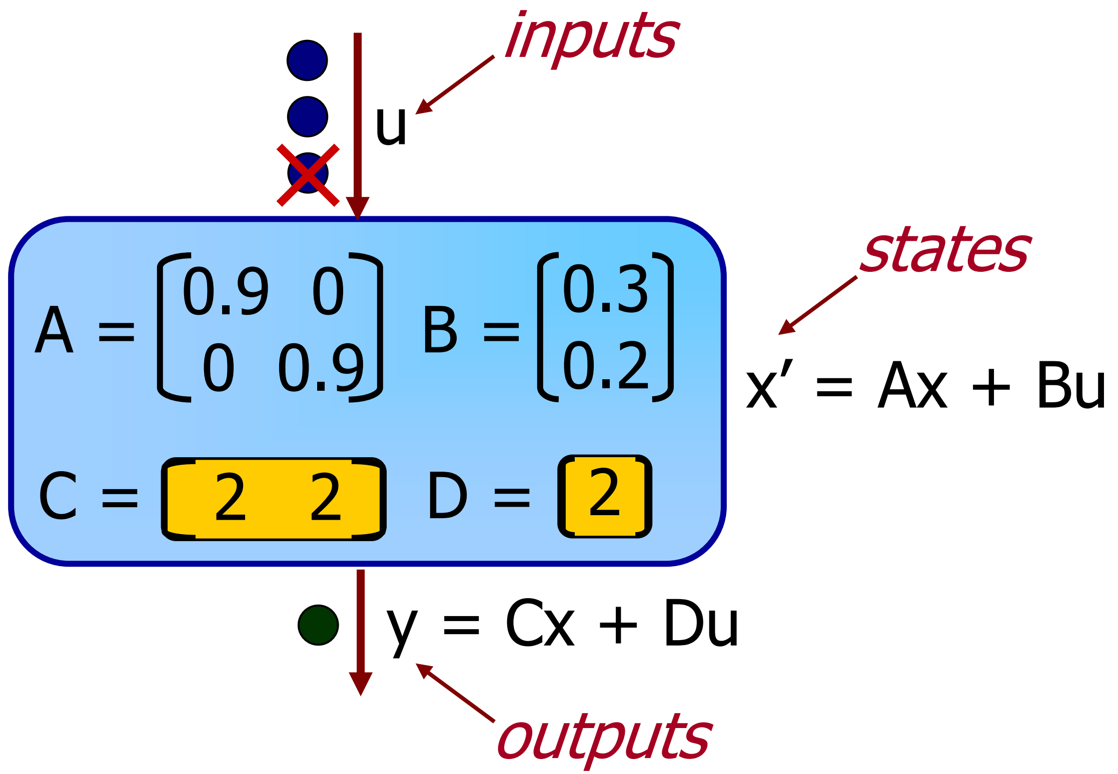
```
float->float filter IIR {
  float x1, x2;
  work push 1 pop 1 {
    float u = pop();
    push(2*(x1+x2+u));
    x1 = 0.9*x1 + 0.3*u;
    x2 = 0.9*x2 + 0.2*u;
} }
```



Representing State Space Filters

- A state space filter is a tuple ⟨A, B, C, D⟩

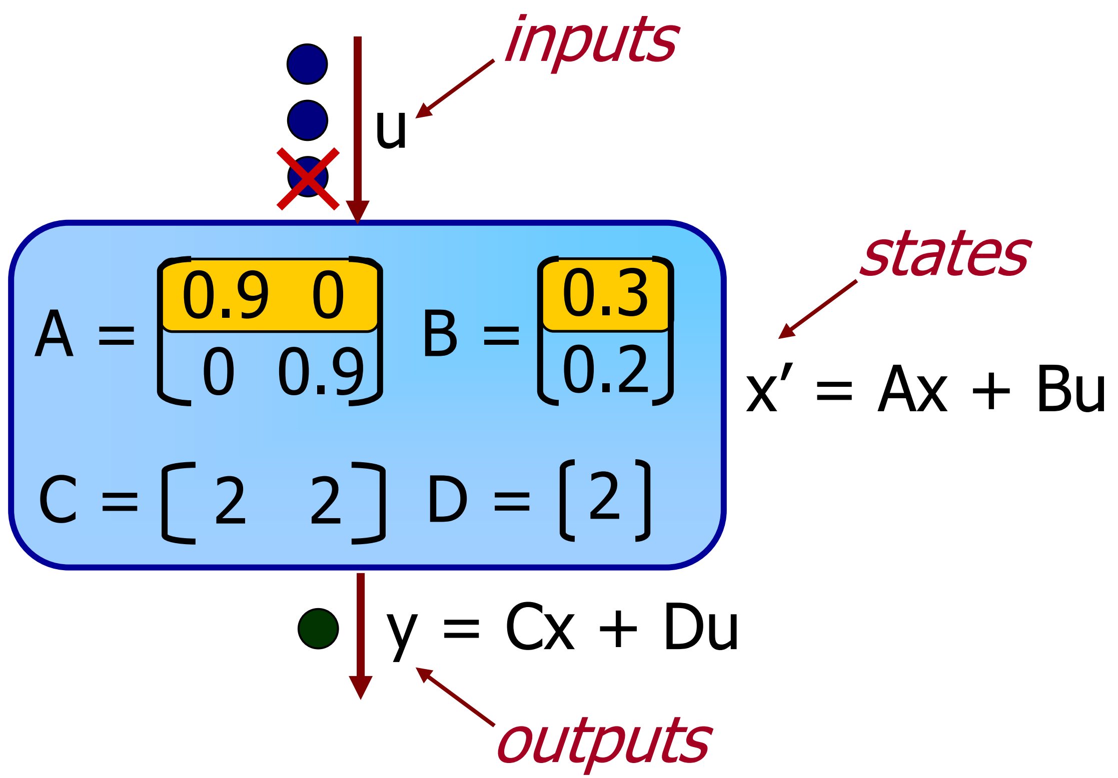
```
float->float filter IIR {
  float x1, x2;
  work push 1 pop 1 {
    float u = pop();
    push(2*(x1+x2+u));
    x1 = 0.9*x1 + 0.3*u;
    x2 = 0.9*x2 + 0.2*u;
} }
```



Representing State Space Filters

- A state space filter is a tuple ⟨A, B, C, D⟩

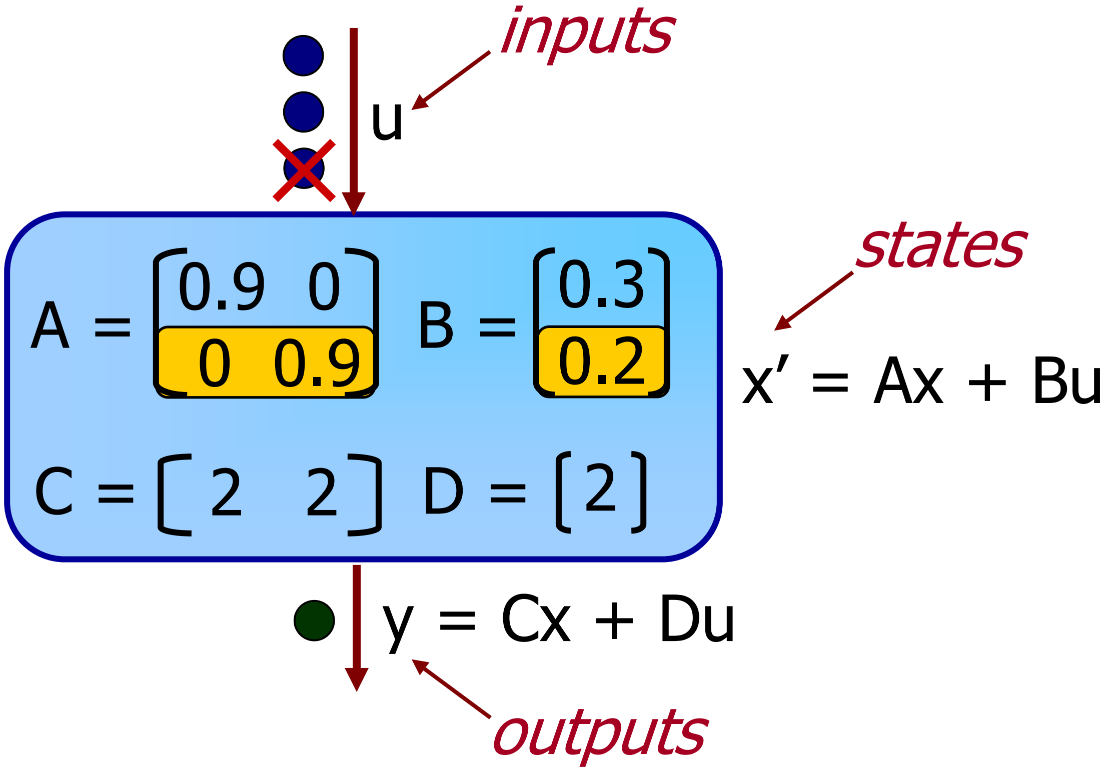
```
float->float filter IIR {
  float x1, x2;
  work push 1 pop 1 {
    float u = pop();
    push(2*(x1+x2+u));
    x1 = 0.9*x1 + 0.3*u;
    x2 = 0.9*x2 + 0.2*u;
} }
```



Representing State Space Filters

- A state space filter is a tuple ⟨A, B, C, D⟩

```
float->float filter IIR {
  float x1, x2;
  work push 1 pop 1 {
    float u = pop();
    push(2*(x1+x2+u));
    x1 = 0.9*x1 + 0.3*u;
    x2 = 0.9*x2 + 0.2*u;
} }
```



$$A = \begin{bmatrix} 0.9 & 0 \\ 0 & 0.9 \end{bmatrix} \quad B = \begin{bmatrix} 0.3 \\ 0.2 \end{bmatrix}$$

$$C = \begin{bmatrix} 2 & 2 \end{bmatrix} \quad D = \begin{bmatrix} 2 \end{bmatrix}$$

$$x' = Ax + Bu$$

$$y = Cx + Du$$

Representing State Space Filters

- A state space filter is a tuple ↑A, B, C, D⟩

```
float->float filter IIR {
  float x1, x2;
  work push 1 pop 1 {
    float u = pop();
    push(2*(x1+x2+u));
    x1 = 0.9*x1 + 0.3*u;
    x2 = 0.9*x2 + 0.2*u;
} }
```

inputs u

$$A = \begin{bmatrix} 0.9 & 0 \\ 0 & 0.9 \end{bmatrix} \quad B = \begin{bmatrix} 0.3 \\ 0.2 \end{bmatrix}$$

$$C = \begin{bmatrix} 2 & 2 \end{bmatrix} \quad D = \begin{bmatrix} 2 \end{bmatrix}$$

states $x' = Ax + Bu$

$y = Cx + Du$ *outputs*

Linear dataflow analysis

State Space Optimizations

1. State removal
2. Reducing the number of parameters
3. Combining adjacent filters

Change-of-Basis Transformation

$$x' = Ax + Bu$$
$$y = Cx + Du$$

Change-of-Basis Transformation

$$x' = Ax + Bu$$
$$y = Cx + Du$$

↓ T = invertible matrix

$$Tx' = TAx + TBu$$
$$y = Cx + Du$$

Change-of-Basis Transformation

$$x' = Ax + Bu$$
$$y = Cx + Du$$

T = invertible matrix

$$Tx' = TA(T^{-1}T)x + TBu$$
$$y = C(T^{-1}T)x + Du$$

Change-of-Basis Transformation

$$x' = Ax + Bu$$
$$y = Cx + Du$$

↓ T = invertible matrix

$$Tx' = TAT^{-1}(Tx) + TBu$$
$$y = CT^{-1}(Tx) + Du$$

Change-of-Basis Transformation

$$x' = Ax + Bu$$
$$y = Cx + Du$$

T = invertible matrix, $z = Tx$

$$Tx' = TAT^{-1}(Tx) + TBu$$
$$y = CT^{-1}(Tx) + Du$$

Change-of-Basis Transformation

$$x' = Ax + Bu$$
$$y = Cx + Du$$

T = invertible matrix, $z = Tx$

$$z' = TAT^{-1}z + TBu$$
$$y = CT^{-1}z + Du$$

Change-of-Basis Transformation

$$x' = Ax + Bu$$
$$y = Cx + Du$$

T = invertible matrix, $z = Tx$

$$z' = A'z + B'u$$
$$y = C'z + D'u$$

$A' = TAT^{-1}$ $B' = TB$

$C' = CT^{-1}$ $D' = D$

Change-of-Basis Transformation

$$x' = Ax + Bu$$
$$y = Cx + Du$$

↓ T = invertible matrix, z = Tx

$$z' = A'z + B'u$$
$$y = C'z + D'u$$

$A' = TAT^{-1}$ $B' = TB$
$C' = CT^{-1}$ $D' = D$

Can map original states x to transformed states z = Tx without changing I/O behavior

1) State Removal

- Can remove states which are:
  a. Unreachable – do not depend on input
  b. Unobservable – do not affect output

- To expose unreachable states, reduce [A | B] to a kind of row-echelon form
  – For unobservable states, reduce $[A^T \mid C^T]$
- Automatically finds minimal number of states

State Removal Example

$$x' = \begin{bmatrix} 0.9 & 0 \\ 0 & 0.9 \end{bmatrix} x + \begin{bmatrix} 0.3 \\ 0.2 \end{bmatrix} u$$

$$y = \begin{bmatrix} 2 & 2 \end{bmatrix} x + 2u$$

$$T = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

$$x' = \begin{bmatrix} 0.9 & 0 \\ 0 & 0.9 \end{bmatrix} x + \begin{bmatrix} 0.3 \\ \mathbf{0.5} \end{bmatrix} u$$

$$y = \begin{bmatrix} \mathbf{0} & 2 \end{bmatrix} x + 2u$$

```
float->float filter IIR {
  float x1, x2;
  work push 1 pop 1 {
    float u = pop();
    push(2*(x1+x2+u));
    x1 = 0.9*x1 + 0.3*u;
    x2 = 0.9*x2 + 0.2*u;
} }
```

State Removal Example

$$x' = \begin{bmatrix} 0.9 & 0 \\ 0 & 0.9 \end{bmatrix} x + \begin{bmatrix} 0.3 \\ 0.2 \end{bmatrix} u$$

$$y = \begin{bmatrix} 2 & 2 \end{bmatrix} x + 2u$$

$$T = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

$$x' = \begin{bmatrix} 0.9 & 0 \\ 0 & 0.9 \end{bmatrix} x + \begin{bmatrix} 0.3 \\ \mathbf{0.5} \end{bmatrix} u$$

$$y = \begin{bmatrix} \mathbf{0} & 2 \end{bmatrix} x + 2u$$

x1 is unobservable

```
float->float filter IIR {
  float x1, x2;
  work push 1 pop 1 {
    float u = pop();
    push(2*(x1+x2+u));
    x1 = 0.9*x1 + 0.3*u;
    x2 = 0.9*x2 + 0.2*u;
} }
```

State Removal Example

$$x' = \begin{bmatrix} 0.9 & 0 \\ 0 & 0.9 \end{bmatrix} x + \begin{bmatrix} 0.3 \\ 0.2 \end{bmatrix} u$$

$$y = \begin{bmatrix} 2 & 2 \end{bmatrix} x + 2u$$

$$T = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

$$x' = 0.9x + 0.5u$$

$$y = 2x + 2u$$

```
float->float filter IIR {
  float x1, x2;
  work push 1 pop 1 {
    float u = pop();
    push(2*(x1+x2+u));
    x1 = 0.9*x1 + 0.3*u;
    x2 = 0.9*x2 + 0.2*u;
} }
```

```
float->float filter IIR {
  float x;
  work push 1 pop 1 {
    float u = pop();
    push(2*(x+u));
    x = 0.9*x + 0.5*u;
} }
```

State Removal Example

$$\frac{\text{9 FLOPs, 12 load/store}}{\text{output}}$$ ☹

$$\frac{\text{5 FLOPs, 8 load/store}}{\text{output}}$$ ☺

```
float->float filter IIR {
  float x1, x2;
  work push 1 pop 1 {
    float u = pop();
    push(2*(x1+x2+u));
    x1 = 0.9*x1 + 0.3*u;
    x2 = 0.9*x2 + 0.2*u;
} }
```

```
float->float filter IIR {
  float x;
  work push 1 pop 1 {
    float u = pop();
    push(2*(x+u));
    x = 0.9*x + 0.5*u;
} }
```

2) Parameter Reduction

- Goal:
  Convert matrix entries (parameters) to 0 or 1

- Allows static evaluation:

1*x → x	Eliminate 1 multiply
0*x + y → y	Eliminate 1 multiply, 1 add

- Algorithm (Ackerman & Bucy, 1971)
  - Also reduces matrices [A | B] and [A^T | C^T]
  - Attains a canonical form with few parameters

Parameter Reduction Example

$$x' = 0.9x + 0.5u$$
$$y = 2x + 2u$$

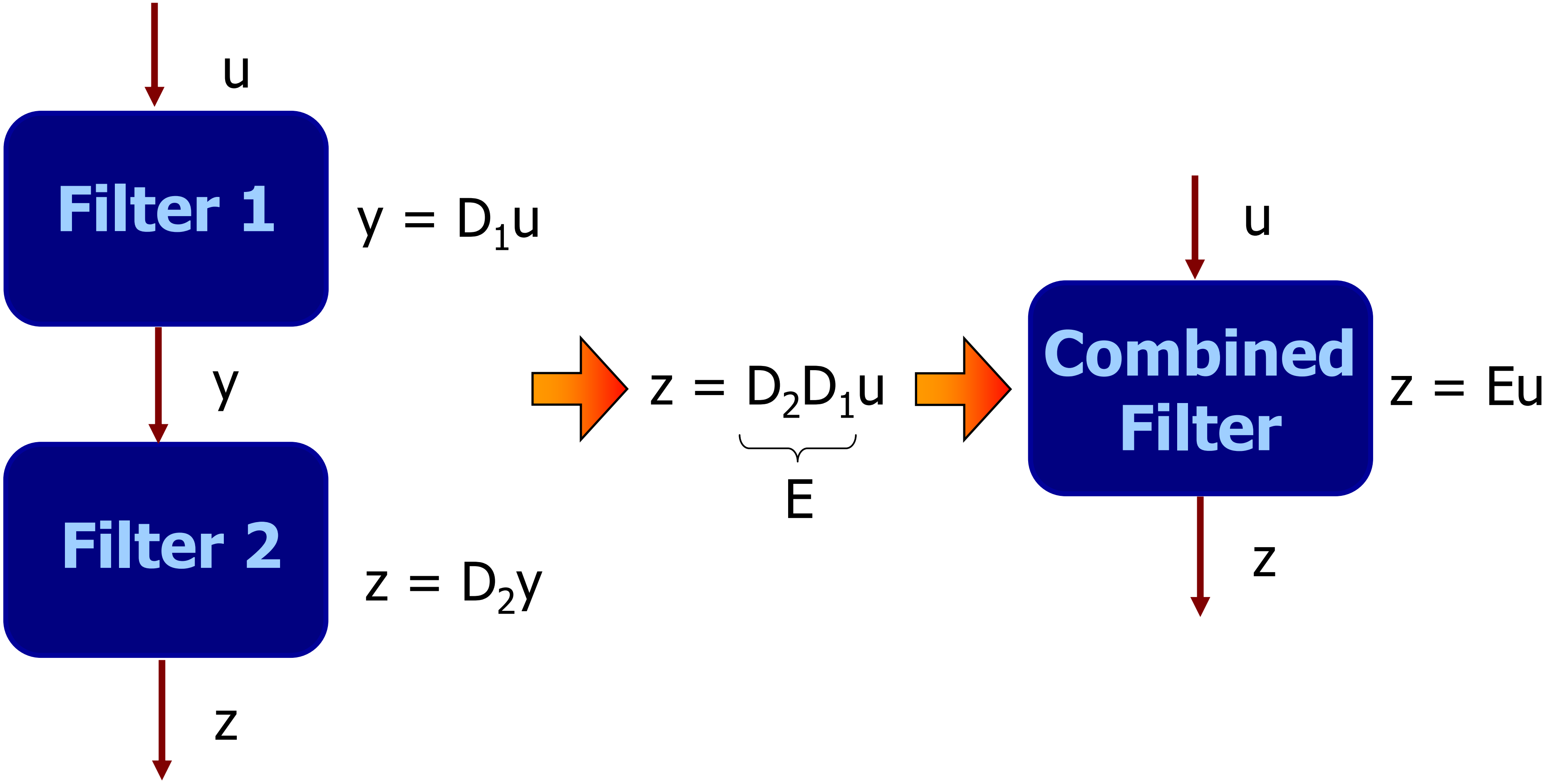
$T = [2]$

$$x' = 0.9x + \mathbf{1}u$$
$$y = \mathbf{1}x + 2u$$

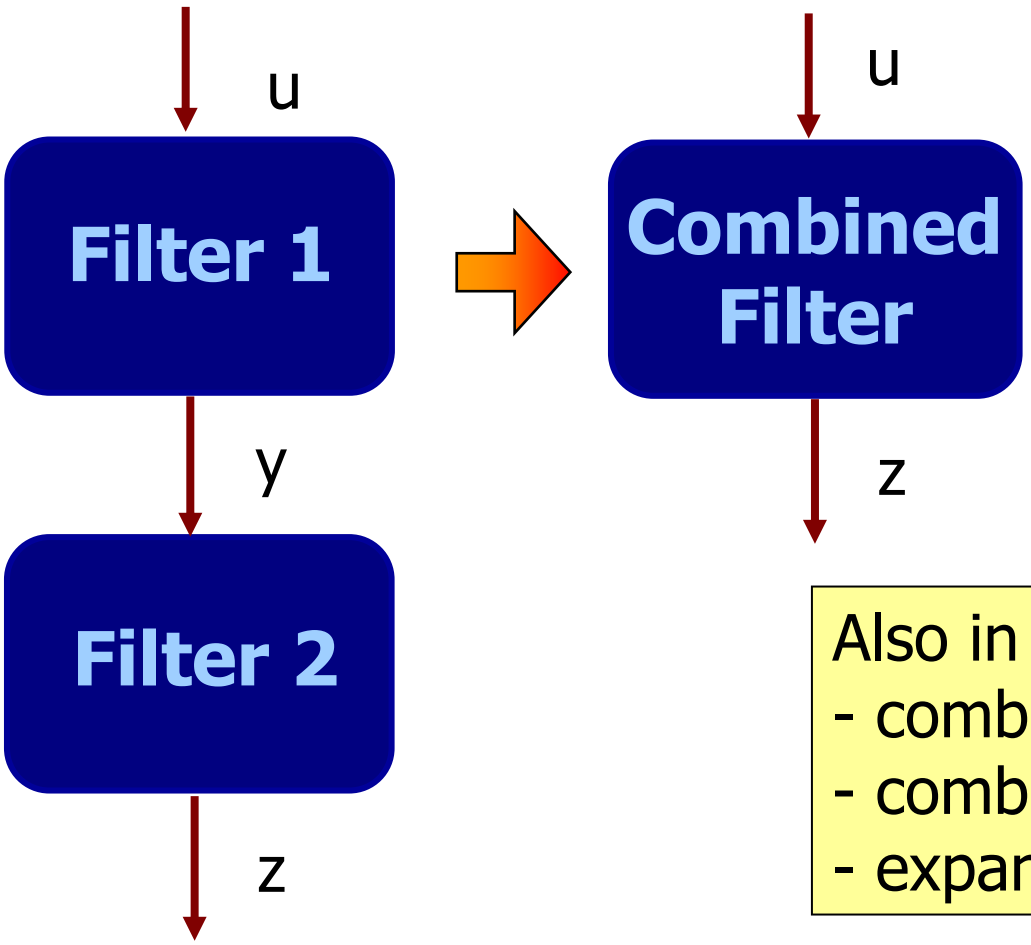
$$\frac{\text{6 FLOPs}}{\text{output}}$$

$$\frac{\text{4 FLOPs}}{\text{output}}$$

3) Combining Adjacent Filters



3) Combining Adjacent Filters



$$x' = \begin{bmatrix} A_1 & 0 \\ B_2C_1 & A_2 \end{bmatrix} x + \begin{bmatrix} B_1 \\ B_2D_1 \end{bmatrix} u$$

$$z = \begin{bmatrix} D_2C_1 & C_2 \end{bmatrix} x + \begin{bmatrix} D_2D_1 \end{bmatrix} u$$

Also in paper:
- combination of parallel streams
- combination of feedback loops
- expansion of mis-matching filters

Combination Example

IIR Filter

$$x' = 0.9x + u$$
$$y = x + 2u$$

Decimator

$$y = [1\ 0]\begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

IIR / Decimator

$$x' = 0.81x + [0.9\quad 1]\begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$
$$y = x + [2\ 0]\begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

$$\frac{\text{8 FLOPs}}{\text{output}}$$ ☹

$$\frac{\text{6 FLOPs}}{\text{output}}$$ ☺

Combination Example

IIR Filter

$$x' = 0.9x + u$$
$$y = x + 2u$$

Decimator

$$y = [1\ 0]\begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

IIR / Decimator

$$x' = 0.81x + [0.9\ \ 1]\begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$
$$y = x + [2\ 0]\begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

As decimation factor goes to ∞,
eliminate up to 75% of FLOPs.

Combination Hazards

- Combination sometimes increases FLOPs
- Example: FFT
  - Combination results in DFT
  - Converts O(n log n) algorithm to $O(n^2)$
- Solution: only apply where beneficial
  - Operations known at compile time
  - Using selection algorithm, FLOPs never increase
    - See PLDI '03 paper for details

Results

- Subsumes combination of linear components
  - Evaluated previously [PLDI ’03]
    - **Applications**: FIR, RateConvert, TargetDetect, Radar, FMRadio, FilterBank, Vocoder, Oversampler, DtoA
  - Removed 44% of FLOPs
  - Speedup of 120% on Pentium 4
- Results using state space analysis

	Speedup (Pentium 3)
IIR + 1:2 Decimator	49%
IIR + 1:16 Decimator	87%

Ongoing Work

- Experimental evaluation
  - Evaluate real applications on embedded machines
  - In progress: MPEG2, JPEG, radar tracker
- Numerical precision constraints
  - Precision often influences choice of coefficients
  - Transformations should respect constraints

Related Work

- Linear stream optimizations [Lamb et al. ’03]
  - Deals with stateless filters
- Automatic optimization of linear libraries
  - SPIRAL, FFTW, ATLAS, Sparsity
- Stream languages
  - Lustre, Esterel, Signal, Lucid, Lucid Synchrone, Brook, Spidle, Cg, Occam , Sisal, Parallel Haskell
- Common sub-expression elimination

Conclusions

- Linear state space analysis:
  An elegant compiler IR for DSP programs
- Optimizations using state space representation:
  1. State removal
  2. Parameter reduction
  3. Combining adjacent filters
- Step towards adding efficient abstraction layers that remove the DSP expert from the design flow

StreamIt

http://cag.lcs.mit.edu/streamit