StreamIt: A Language for Streaming Applications

William Thies, Michal Karczmarek, Michael Gordon,
David Maze, Jasper Lin, Ali Meli, Andrew Lamb,
Chris Leger and Saman Amarasinghe

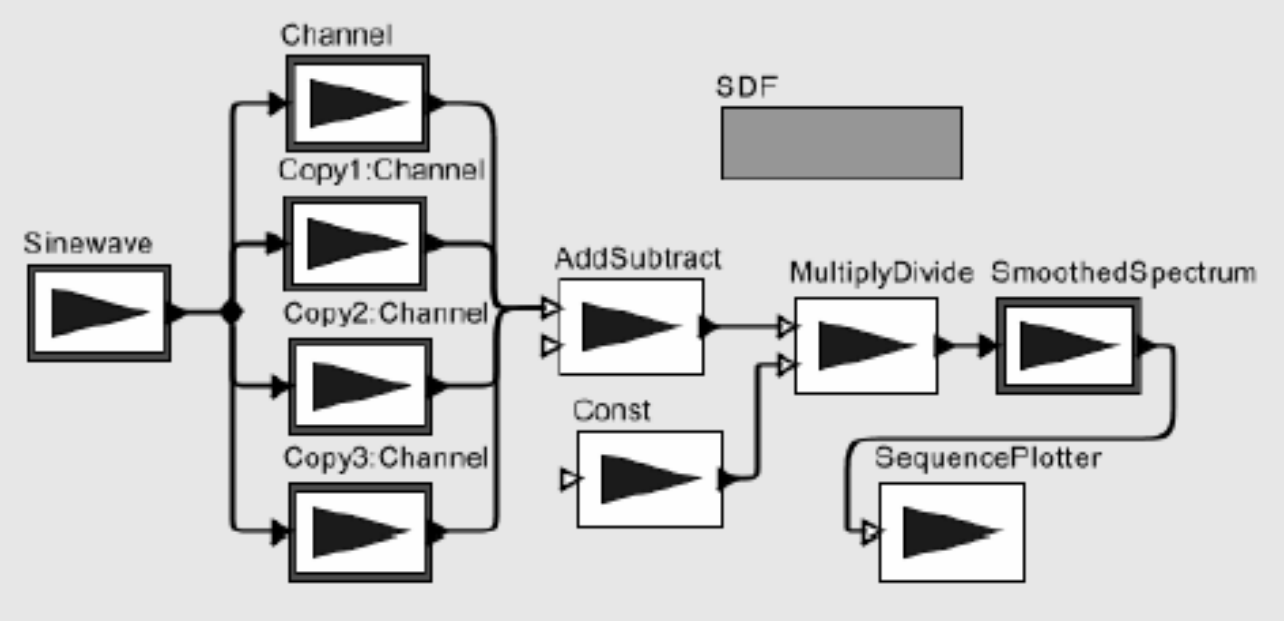
MIT Laboratory for Computer Science

New England Programming Languages and Systems Symposium
August 7, 2002

Streaming Application Domain

- Based on streams of data
- Increasingly prevalent and important
  - Embedded systems
    - Cell phones, handheld computers, DSP's
  - Desktop applications
    - Streaming media
    - Software radio
    - Real-time encryption
    - Graphics packages
  - High-performance servers
    - Software routers
    - Cell phone base stations
    - HDTV editing consoles

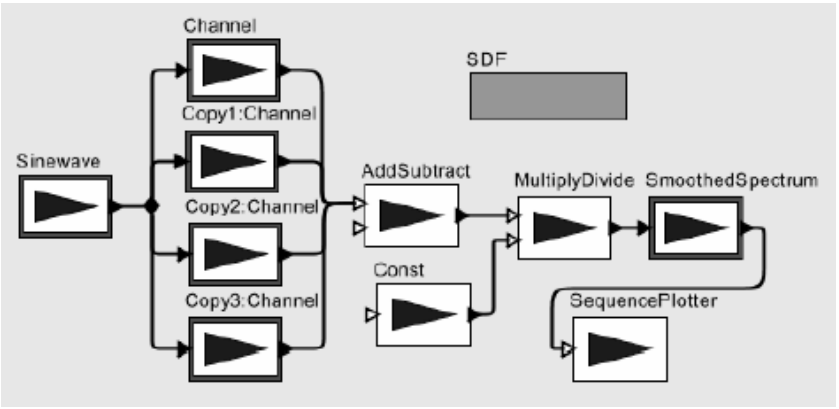
Synchronous Dataflow (SDF)

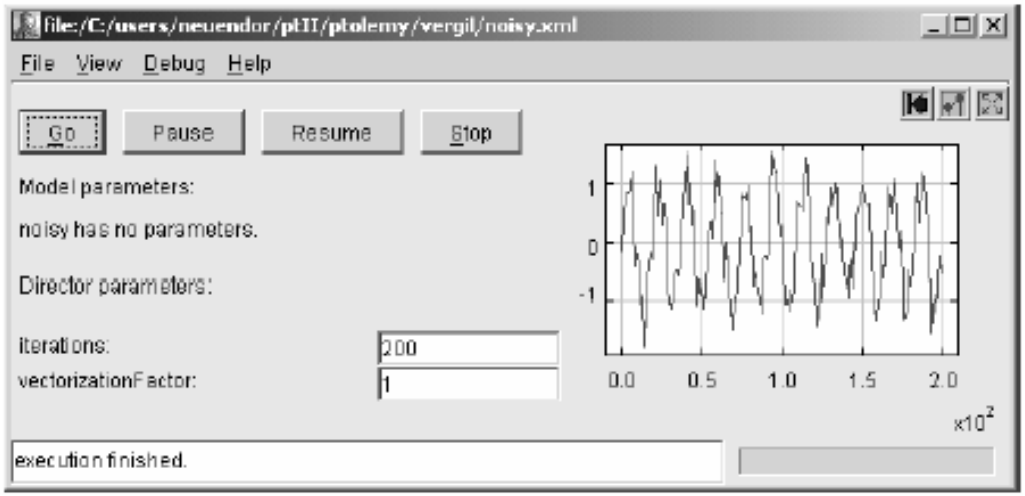


- Application is a graph of nodes
- Nodes send/receive items over channels
- Nodes have static I/O rates

  ⇨ Can construct a static schedule

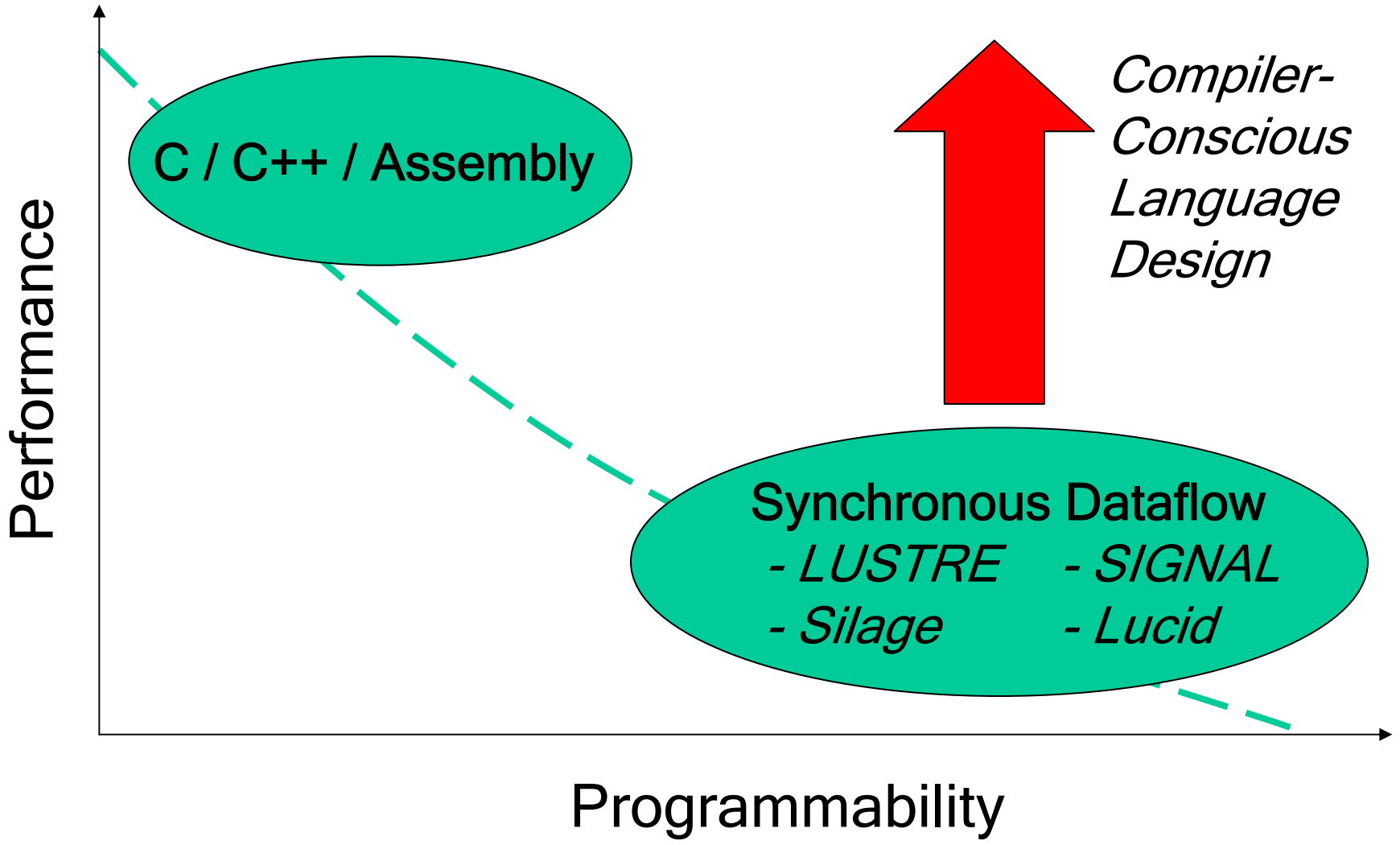
Prototyping Streaming Apps.





- Modeling Environments:
  - Ptolemy (UC Berkeley)
  - COSSAP (Synopsys)
  - SPW (Cadence)
  - ADS (Hewlett Packard)
  - DSP Station (Mentor Graphics)

Programming Streaming Apps.



The StreamIt Language

- Also a synchronous dataflow language
  - With a few extra features
- Goals:
  - High performance
  - Improved programmer productivity
- Language Contributions:
  - Structured model of streams
  - Messaging system for control
  - Automatic program morphing

} ENABLES Compiler Analysis & Optimization

Outline

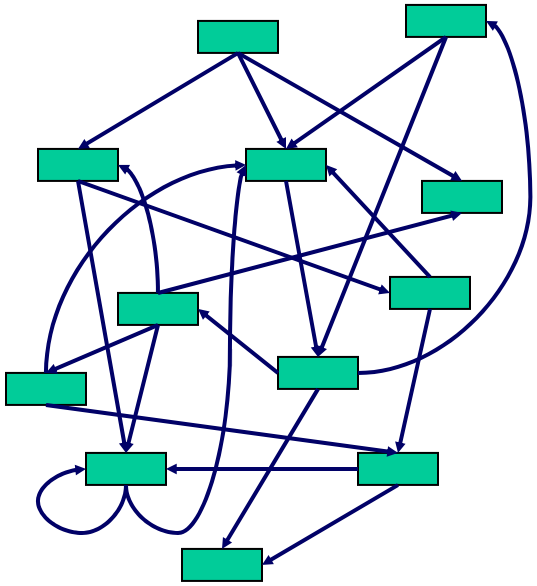
- Design of Streamlt
  - Structured Streams
  - Messaging
  - Morphing
- Results
- Conclusions

Outline

- Design of StreamIt
  - Structured Streams
  - Messaging
  - Morphing
- Results
- Conclusions

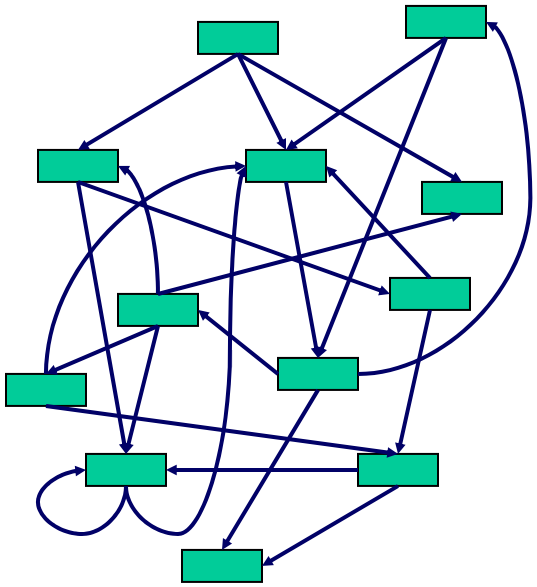
Representing Streams

- Conventional wisdom: streams are graphs
  - Graphs have no simple textual representation
  - Graphs are difficult to analyze and optimize

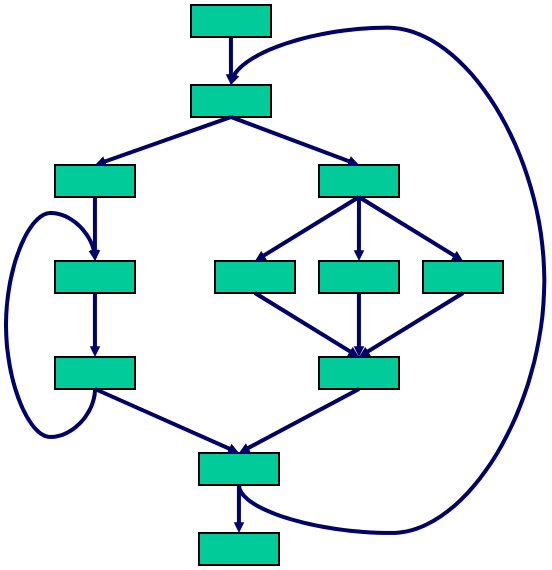


Representing Streams

- Conventional wisdom: streams are graphs
  - Graphs have no simple textual representation
  - Graphs are difficult to analyze and optimize
- Insight: stream programs have structure



unstructured

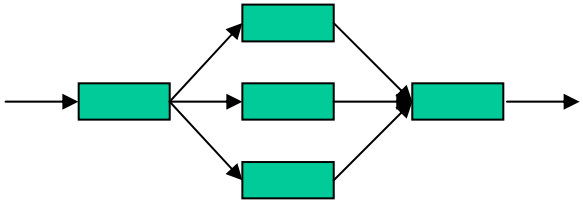


structured

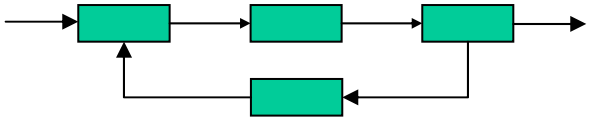
Structured Streams

- Hierarchical structures:
  - Pipeline

  - SplitJoin



  - Feedback Loop

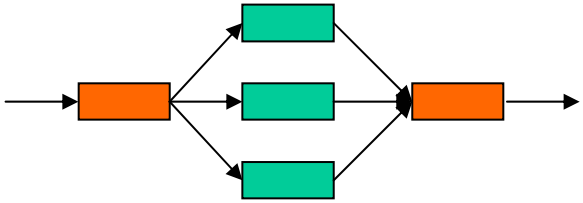


- Basic programmable unit: Filter

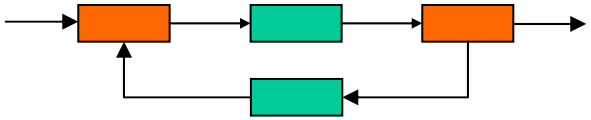
Structured Streams

- Hierarchical structures:
  - Pipeline

  - SplitJoin



  - Feedback Loop



- Basic programmable unit: Filter

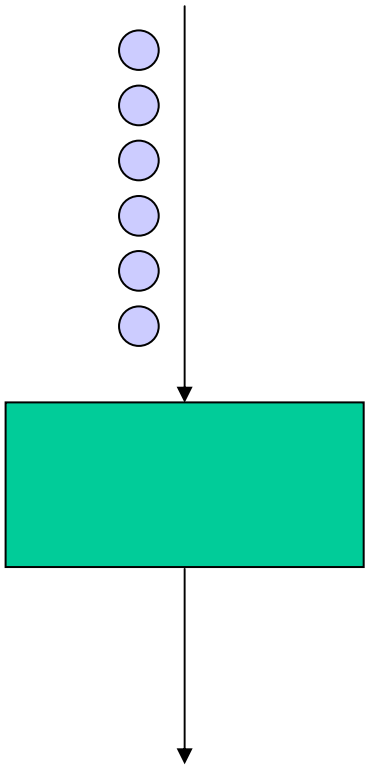
- Splits / Joins are compiler-defined

Representing Filters

- Autonomous unit of computation 
  - No access to global resources
  - Communicates through FIFO channels
    - pop() - peek(index) - push(value)
  - Peek / pop / push rates must be constant
- Looks like a Java class, with
  - An initialization function
  - A steady-state “work” function
  - Message handler functions

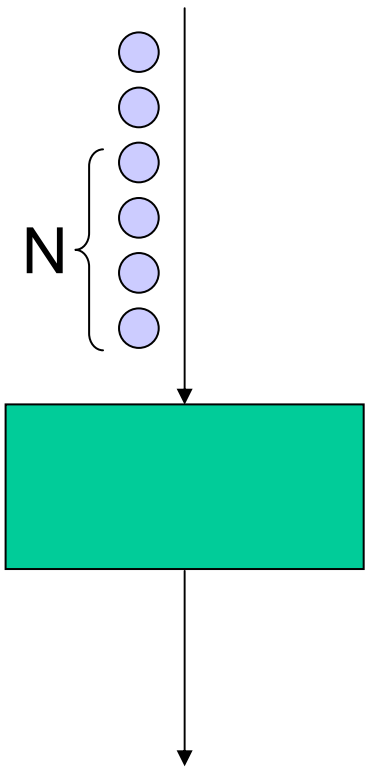
Filter Example: LowPassFilter

```
float->float filter LowPassFilter (float N) {
    float[N] weights;

    init {
        weights = calcWeights(N);
    }

    work push 1 pop 1 peek N {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }
}
```



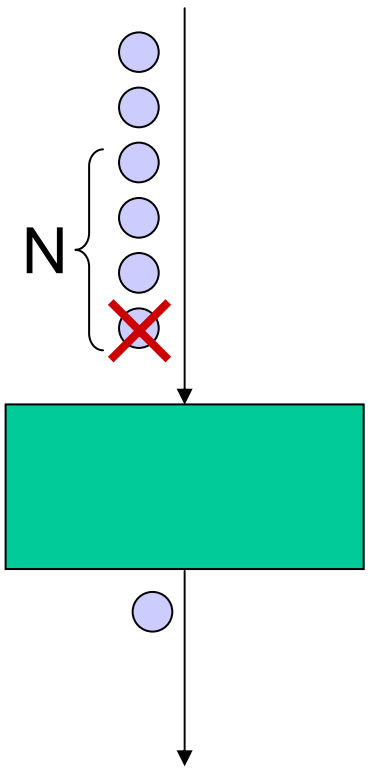
Filter Example: LowPassFilter

```
float->float filter LowPassFilter (float N) {
    float[N] weights;

    init {
        weights = calcWeights(N);
    }

    work push 1 pop 1 peek N {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }
}
```



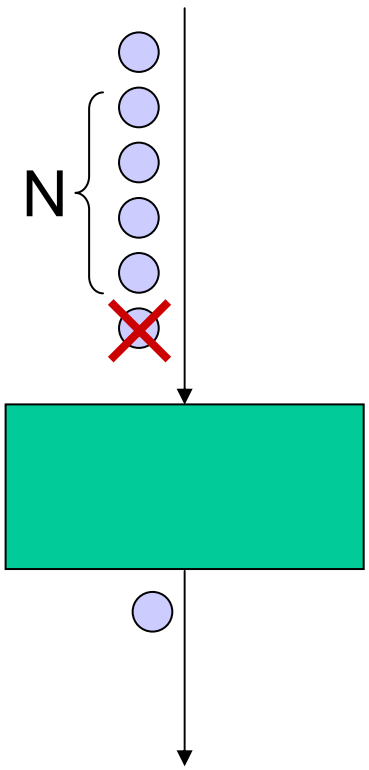
Filter Example: LowPassFilter

```
float->float filter LowPassFilter (float N) {
    float[N] weights;

    init {
        weights = calcWeights(N);
    }

    work push 1 pop 1 peek N {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }
}
```



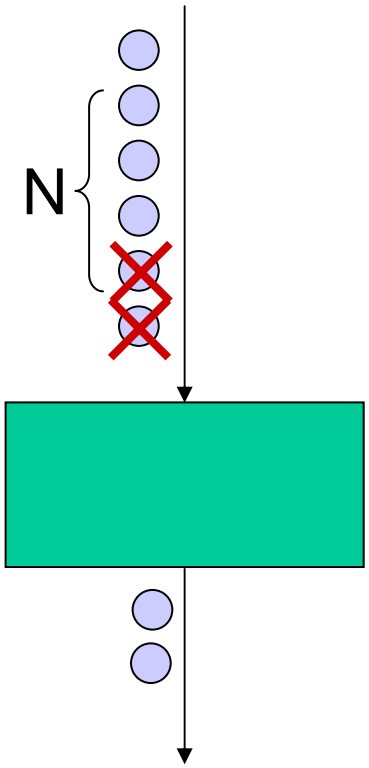
Filter Example: LowPassFilter

```
float->float filter LowPassFilter (float N) {
    float[N] weights;

    init {
        weights = calcWeights(N);
    }

    work push 1 pop 1 peek N {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }
}
```



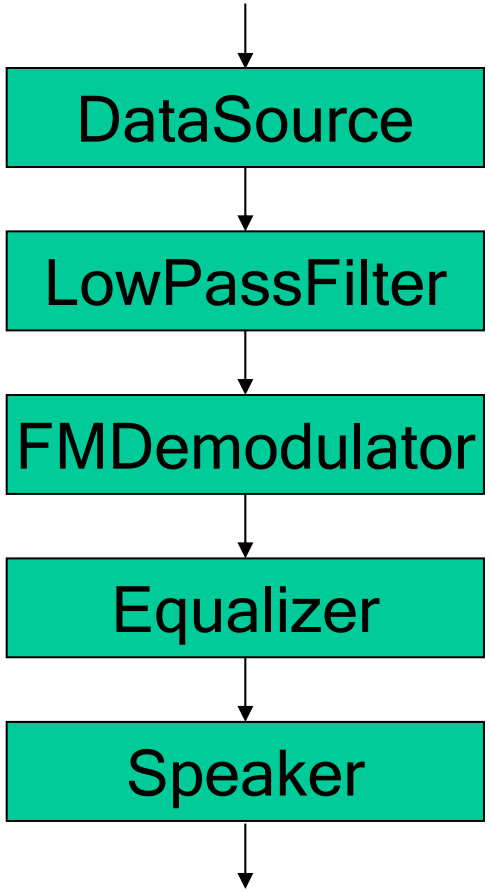
Filter Example: LowPassFilter

```
float->float filter LowPassFilter (float N) {
    float[N] weights;

    init {
        weights = calcWeights(N);
    }

    work push 1 pop 1 peek N {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }
}
```



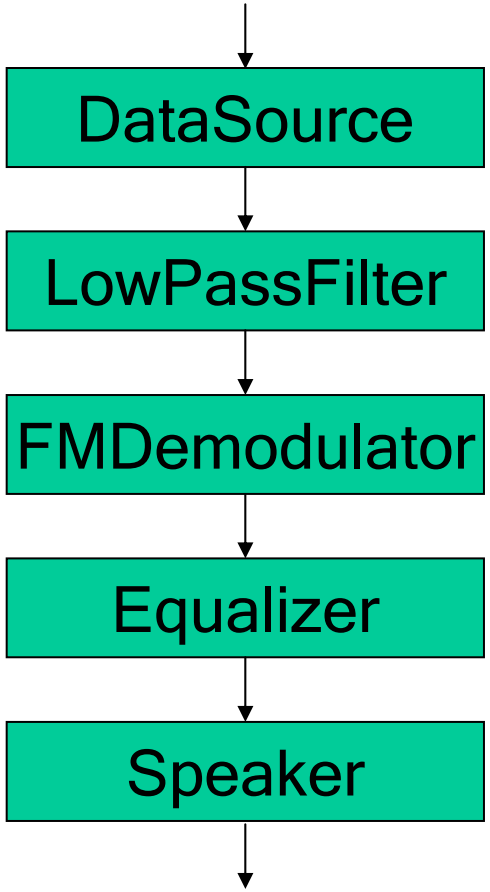
Pipeline Example: FM Radio

```
pipeline FMRadio {
    add DataSource();
    add LowPassFilter();
    add FMDemodulator();
    add Equalizer(8);
    add Speaker();
}
```



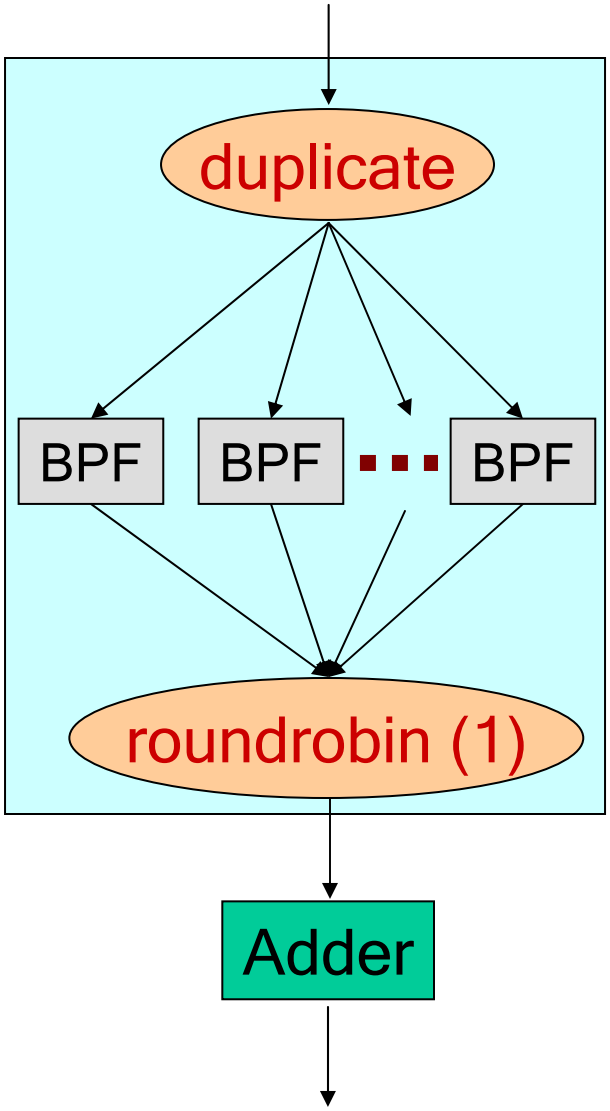
Pipeline Example: FM Radio

```
pipeline FMRadio {
    add DataSource();
    add LowPassFilter();
    add FMDemodulator();
    add Equalizer(8);
    add Speaker();
}
```



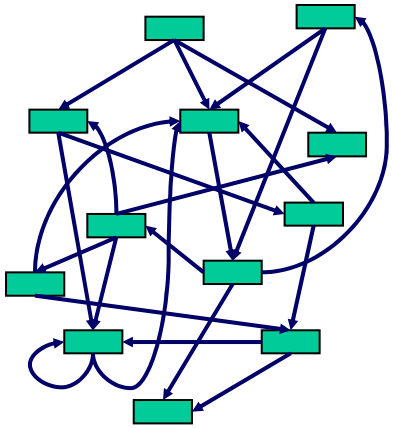
SplitJoin Example: Equalizer

```
pipeline Equalizer (int N) {
  add splitjoin {
    split duplicate;
    float freq = 10000;
    for (int i = 0; i < N; i ++, freq*=2) {
      add BandPassFilter(freq, 2*freq);
    }
    join roundrobin;
  }
  add Adder(N);
  }
}
```

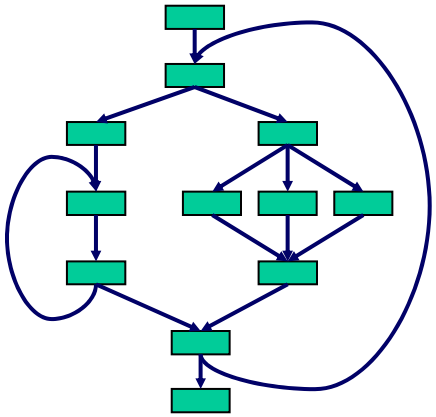


Why Structured Streams?

- Compare to structured control flow



GOTO statements



If / else / for statements

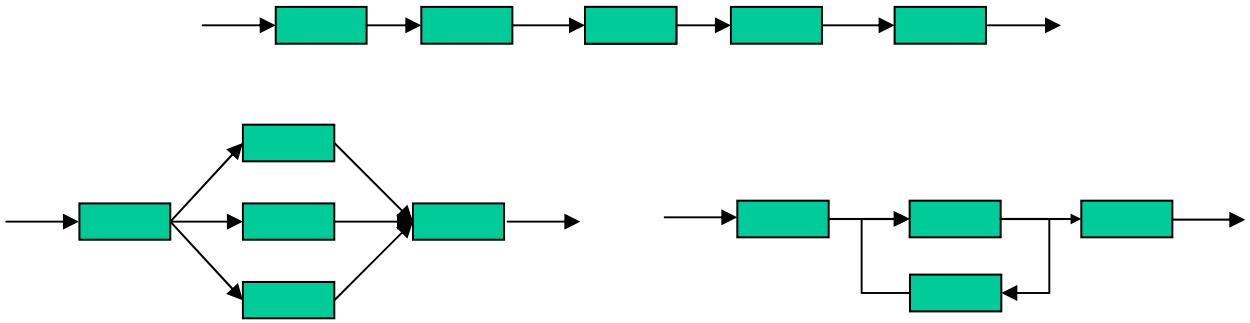
- Tradeoff:

  PRO: - more robust - more analyzable

  CON: - “restricted” style of programming

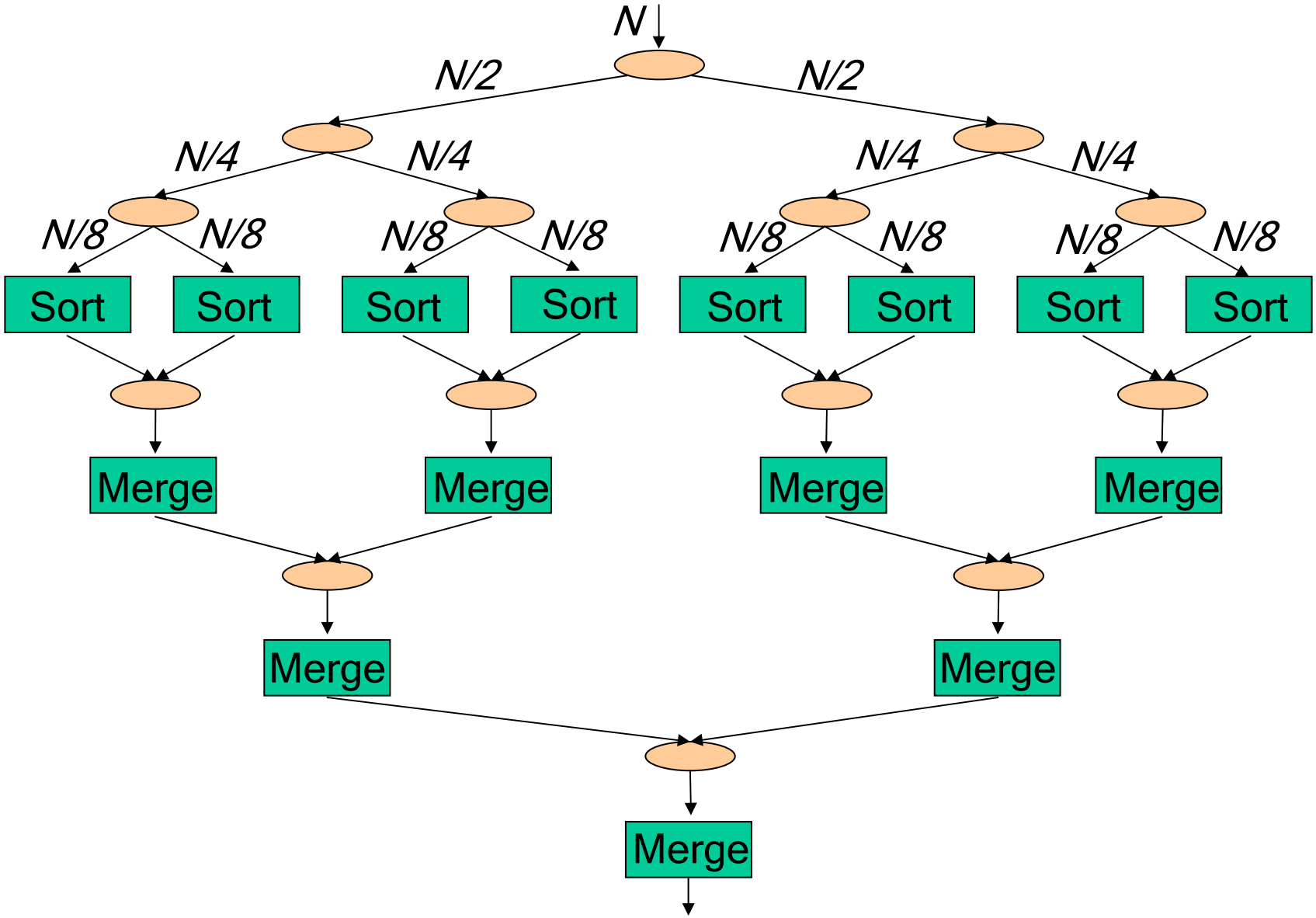
Structure Helps Programmers

- Modules are hierarchical and composable
  - Each structure is single-input, single-output



- Encapsulates common idioms
- Good textual representation
  - Enables parameterizable graphs

N-Element Merge Sort (3-level)

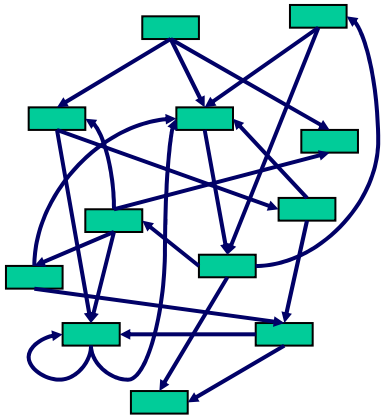


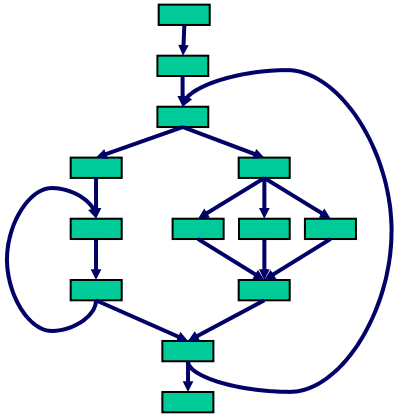
N-Element Merge Sort (K-level)

```
pipeline MergeSort (int N, int K) {
    if (K==1) {
        add Sort(N);
    } else {
        add splitjoin {
            split roundrobin;
            add MergeSort(N/2, K-1);
            add MergeSort(N/2, K-1);
            joiner roundrobin;
        }
    }
    add Merge(N);
  }
}
```

Structure Helps Compilers

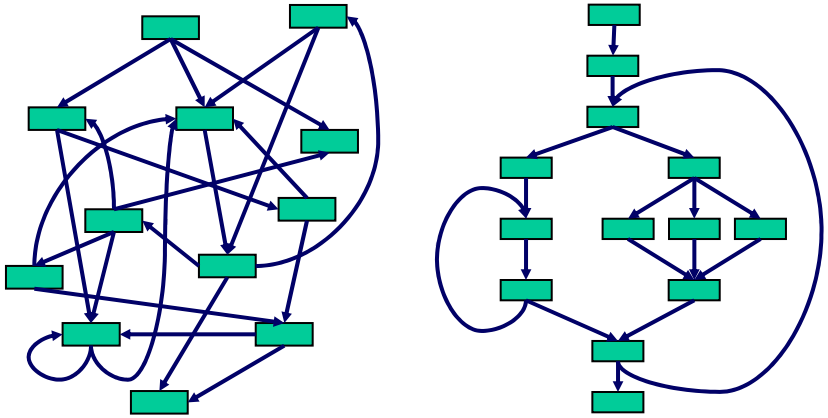
- Enables local, hierarchical analyses
  - Scheduling
  - Optimization
  - Parallelization
  - Load balancing



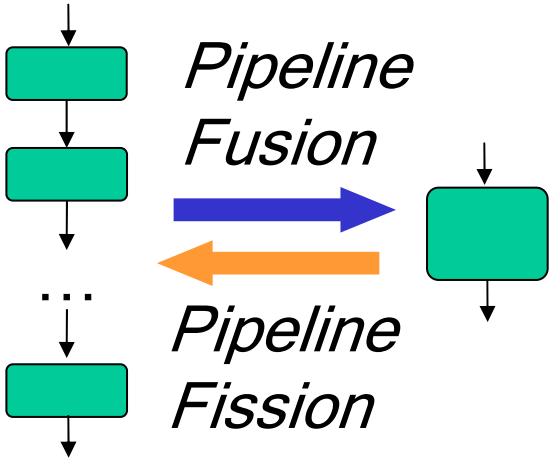


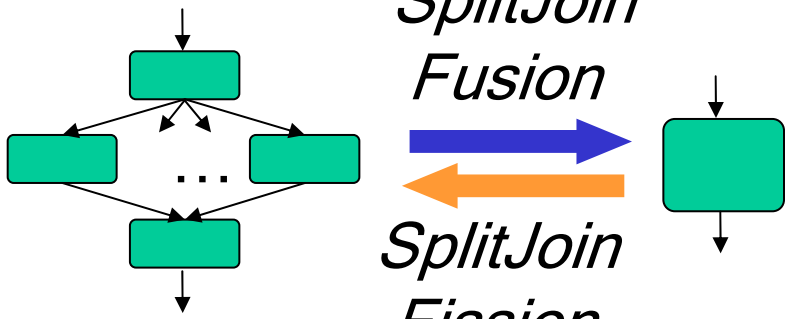
Structure Helps Compilers

- Enables local, hierarchical analyses
  - Scheduling
  - Optimization
  - Parallelization
  - Load balancing



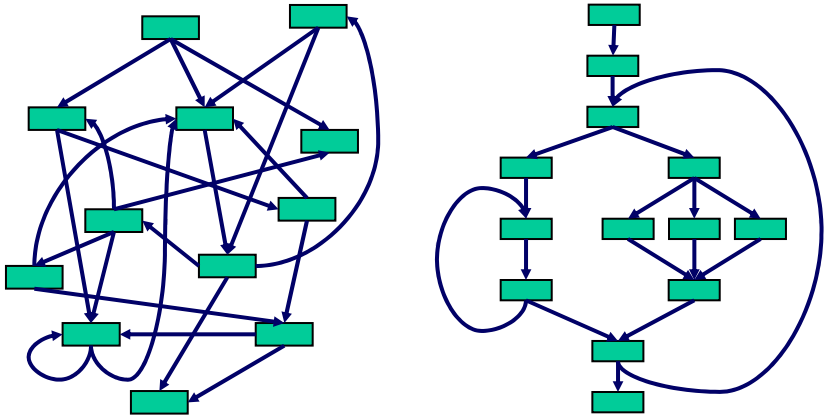
- Examples:



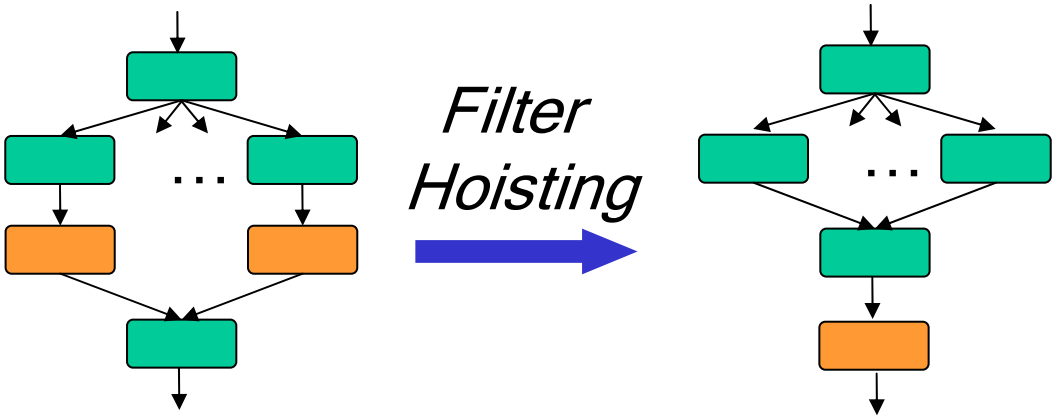


Structure Helps Compilers

- Enables local, hierarchical analyses
  - Scheduling
  - Optimization
  - Parallelization
  - Load balancing

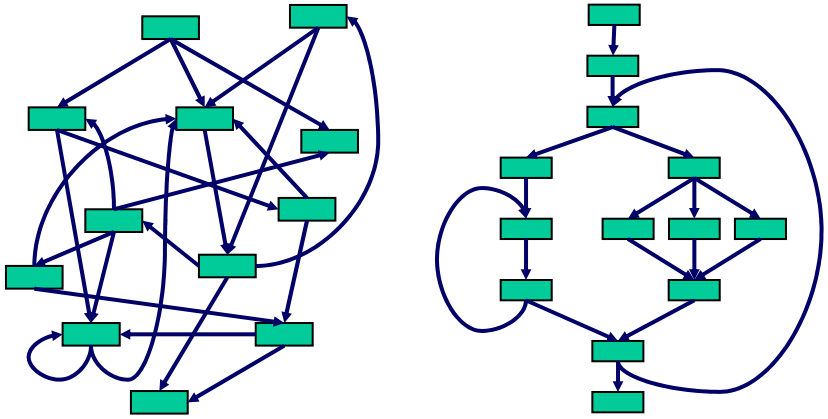


- Examples:



Structure Helps Compilers

- Enables local, hierarchical analyses
  - Scheduling
  - Optimization
  - Parallelization
  - Load balancing



- Disallows non-sensical graphs
- Simplifies separate compilation
  - All blocks single-input, single-output

CON: Restricts Coding Style

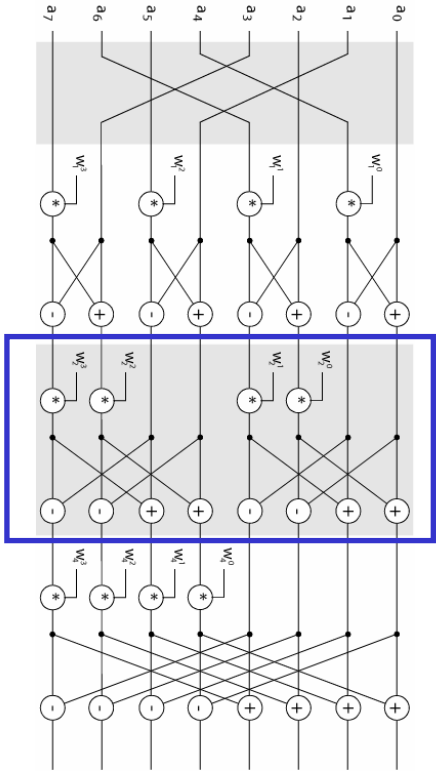
- Some graphs need to be re-arranged
- Example: FFT

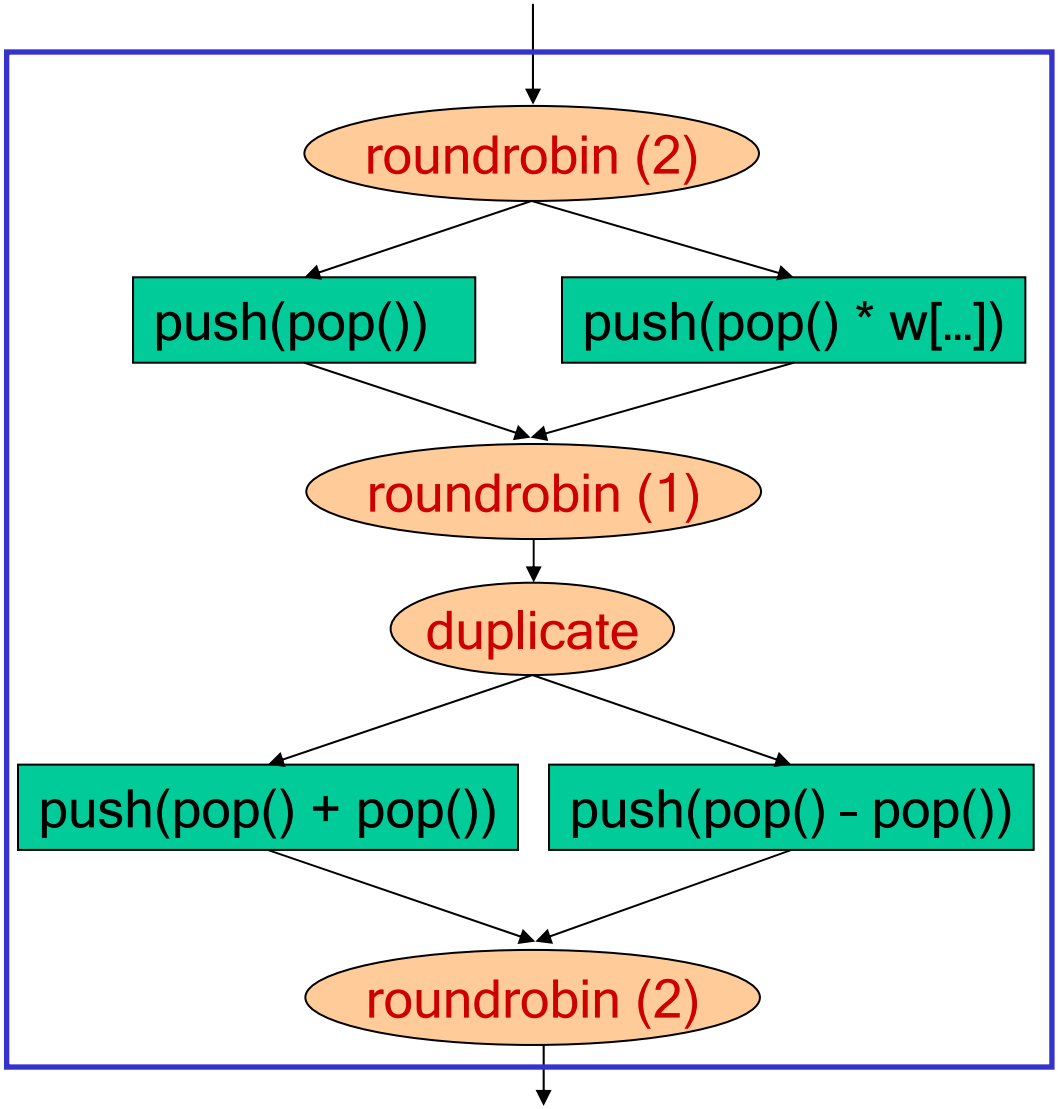
Bit-reverse order

Butterfly (2 way)

Butterfly (4 way)

Butterfly (8 way)



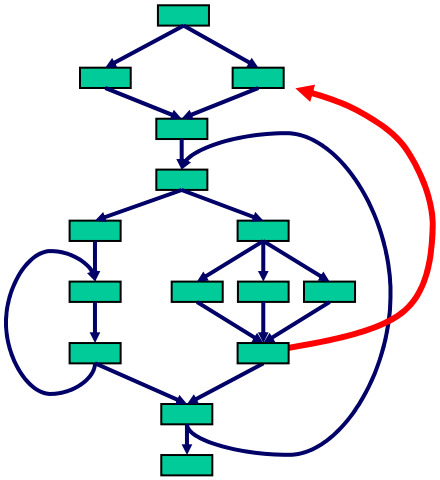


Outline

- Design of StreamIt
  - Structured Streams
  - Messaging
  - Morphing
- Results
- Conclusions

Control Messages

- Structures for regular, high-bandwidth data
- But also need a control mechanism for irregular, low-bandwidth events



  - Change volume on a cell phone
  - Initiate handoff of stream
  - Adjust network protocol

Supporting Control Messages

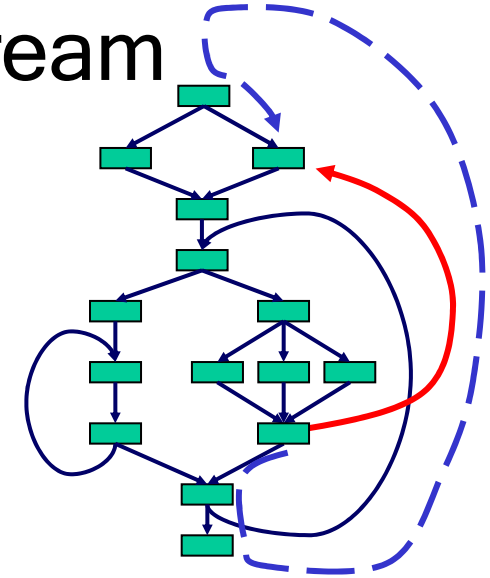
- Option 1: Embed message in stream

  PRO: - message arrives with data

  CON: - complicates filter code

  - complicates structure

  - runtime overhead

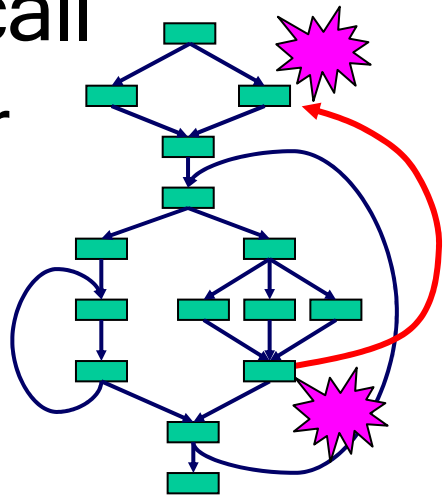


- Option 2: Synchronous method call

  PRO: - delivery transparent to user

  CON: - timing is unclear

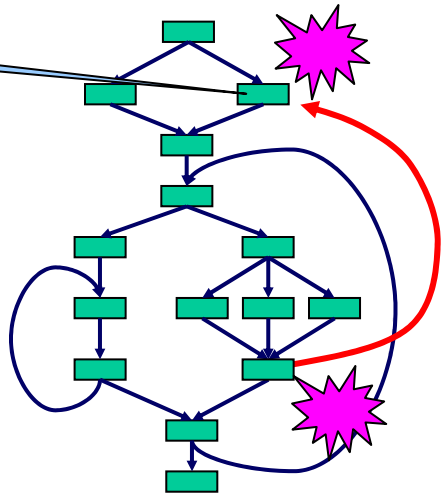
  - limits parallelism



StreamIt Messaging System

- Looks like method call, but semantics differ

```
void raiseVolume(int v)
  myVolume += v;
}
```

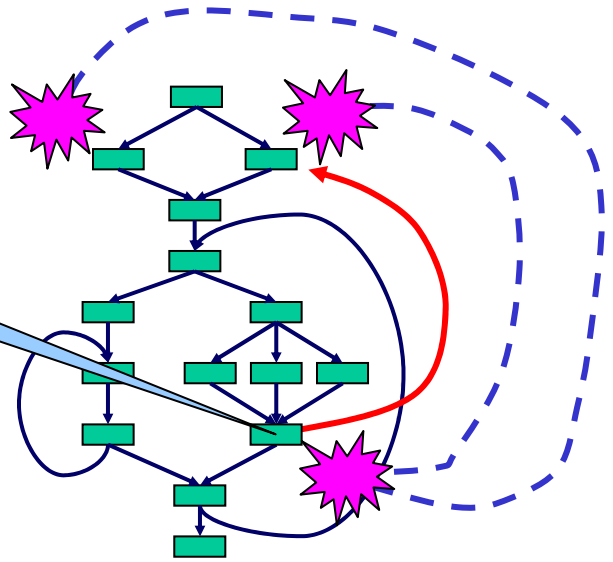


  - No return value
  - Asynchronous delivery
  - Can broadcast to multiple targets

StreamIt Messaging System

- Looks like method call, but semantics differ

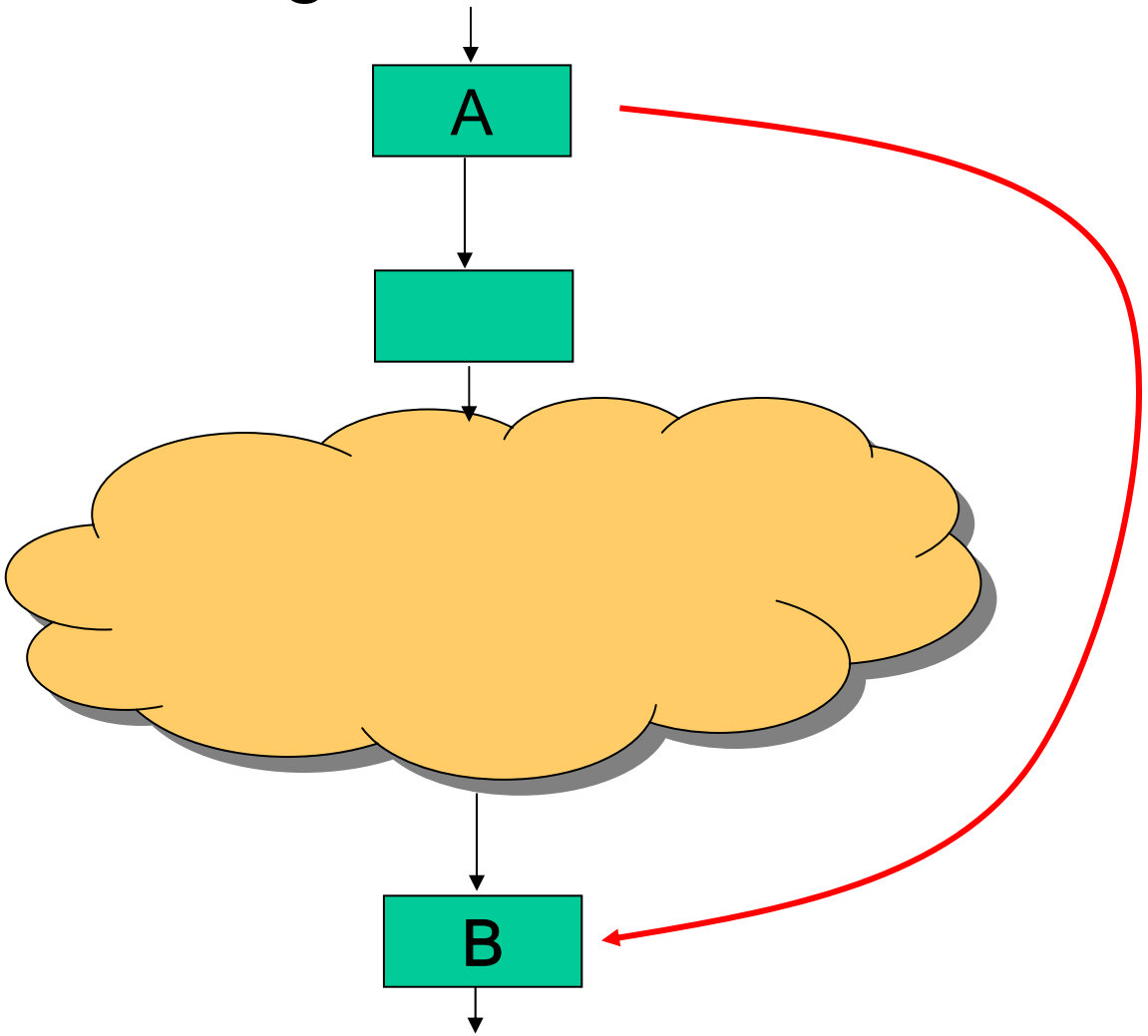
```
TargetFilter x;
work {
  …
  if (lowVolume())
     x.raiseVolume(10) at 100;
}
```



  - No return value
  - Asynchronous delivery
  - Can broadcast to multiple targets
- Timed relative to data
  - User gains precision; compiler gains flexibility

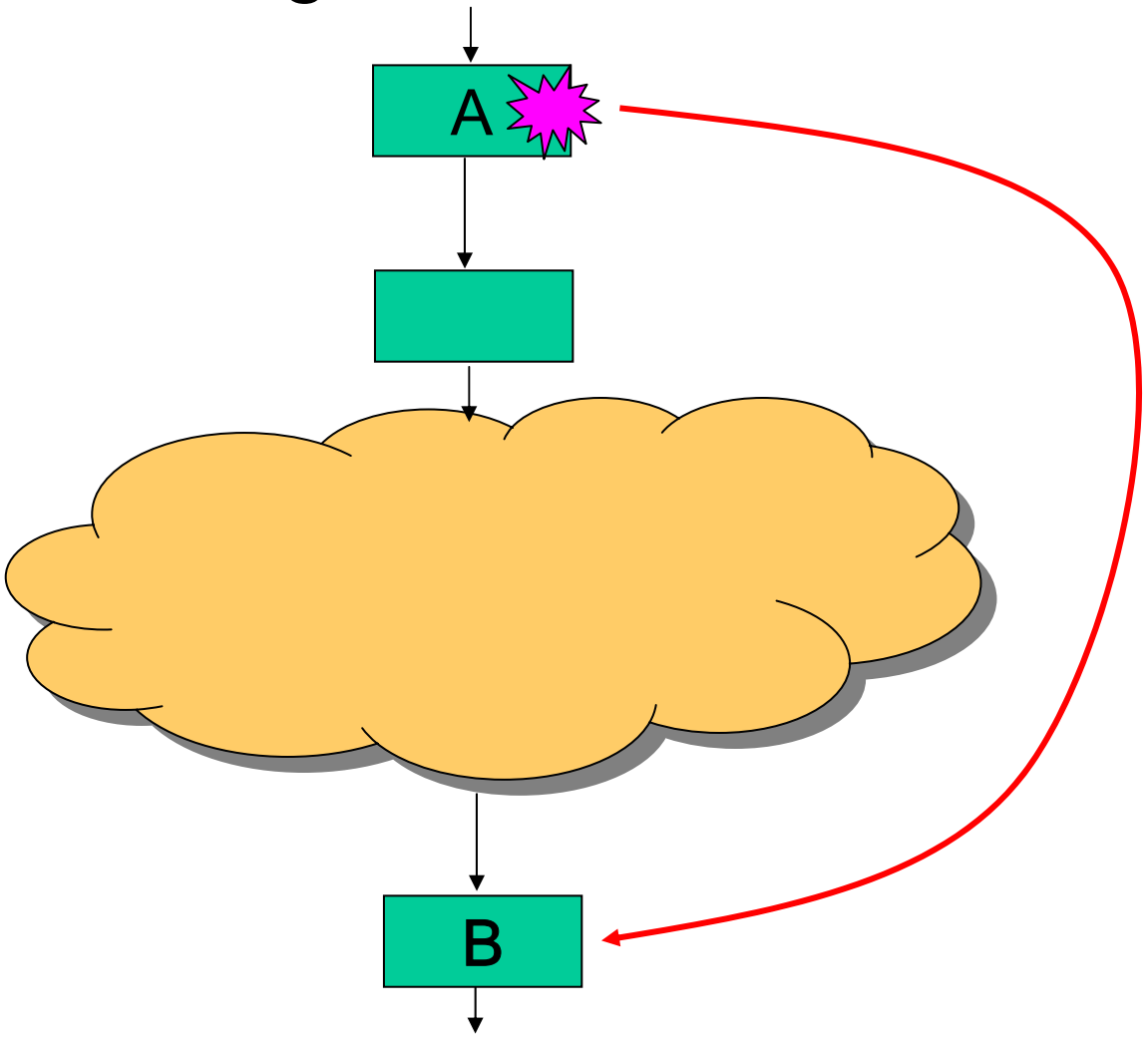
Message Timing

- A sends message to B with zero latency



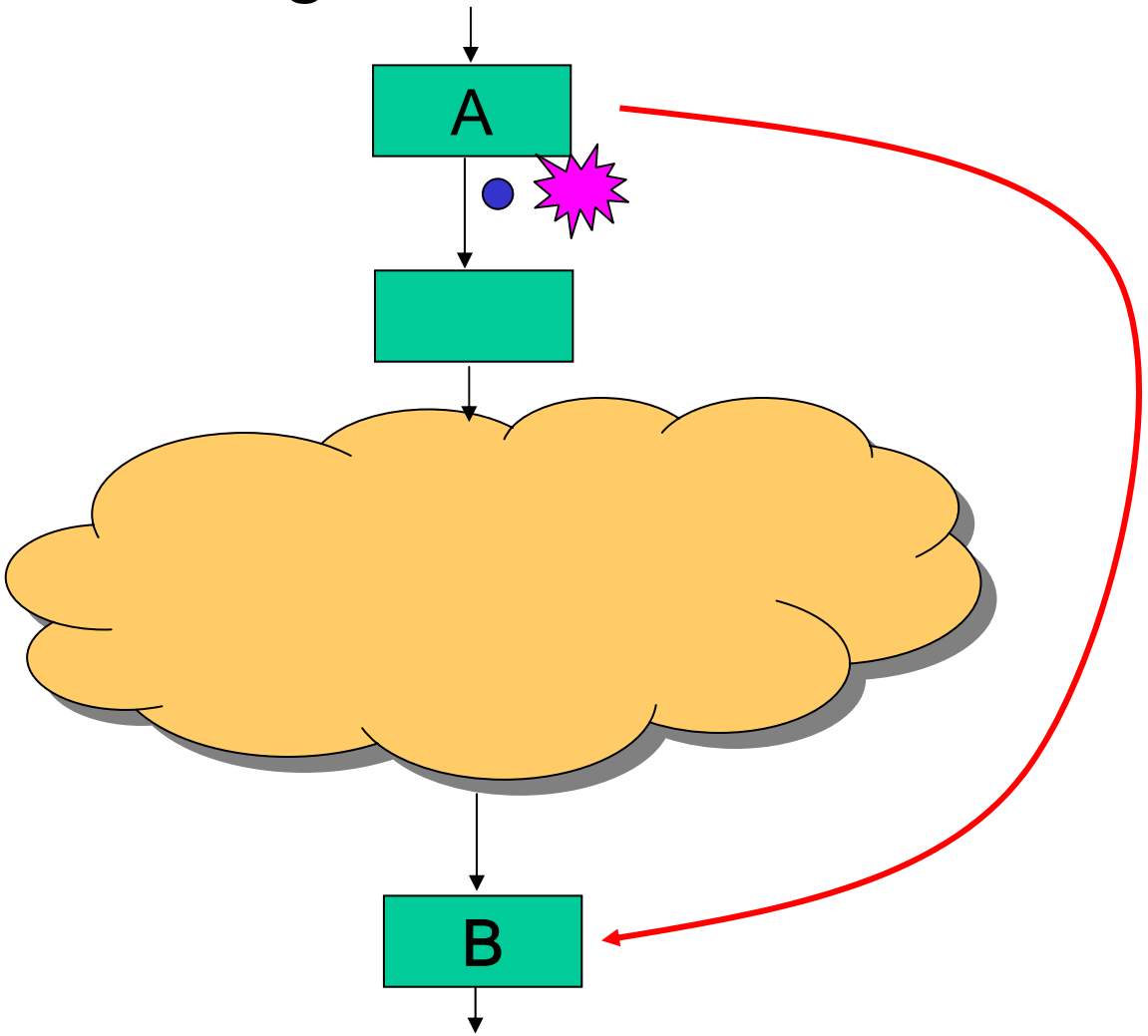
Message Timing

- A sends message to B with zero latency



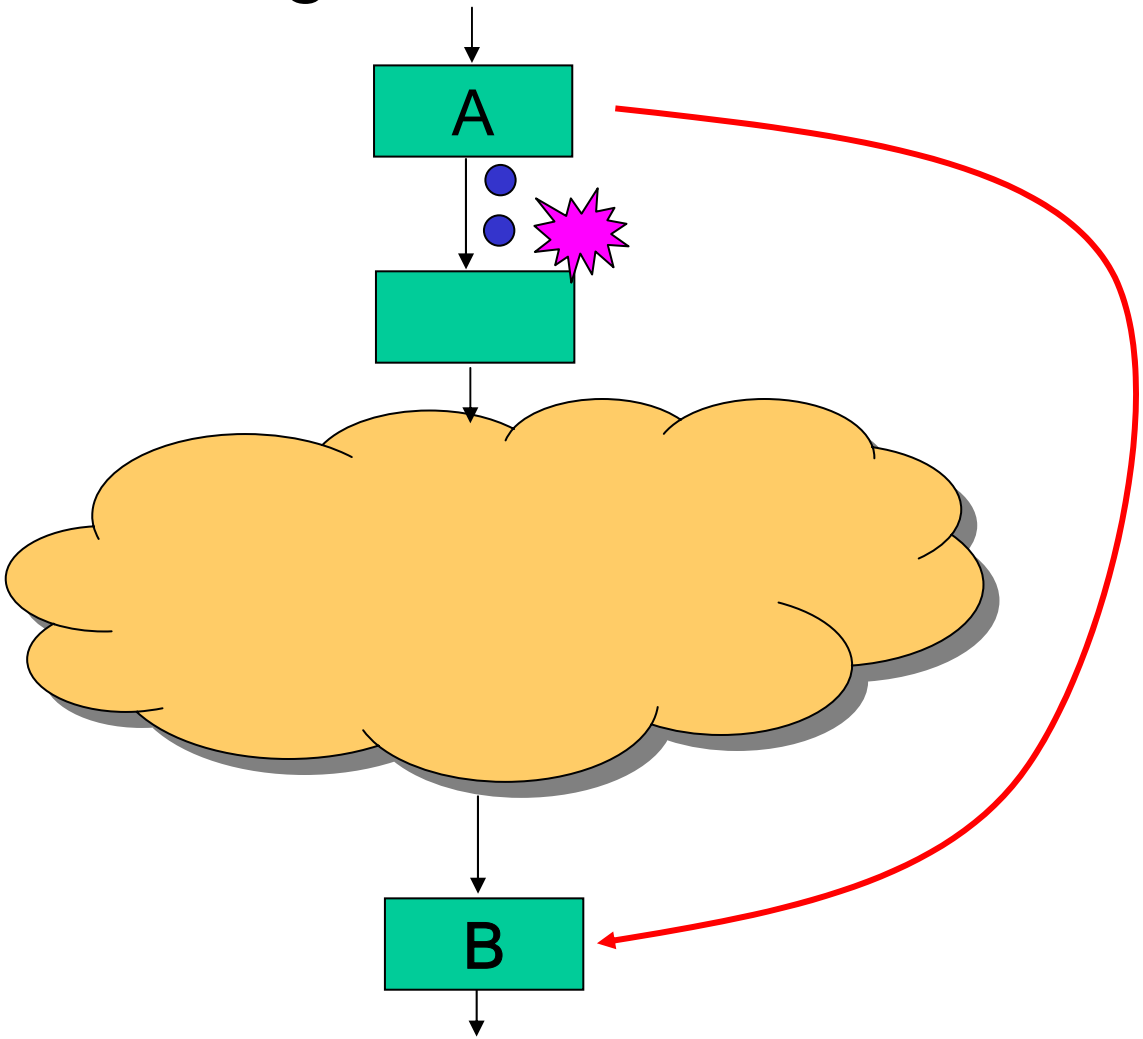
Message Timing

- A sends message to B with zero latency



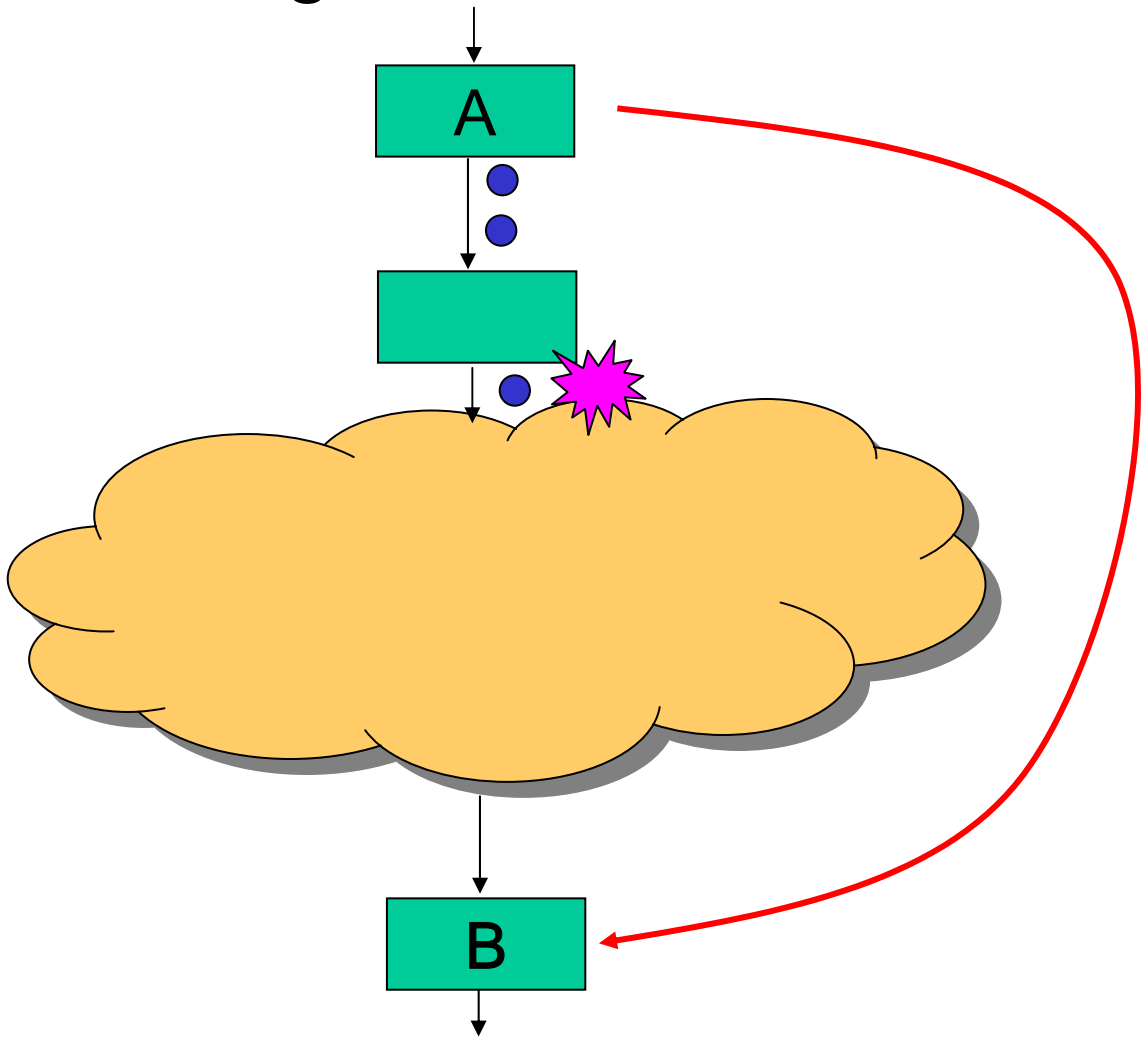
Message Timing

- A sends message to B with zero latency



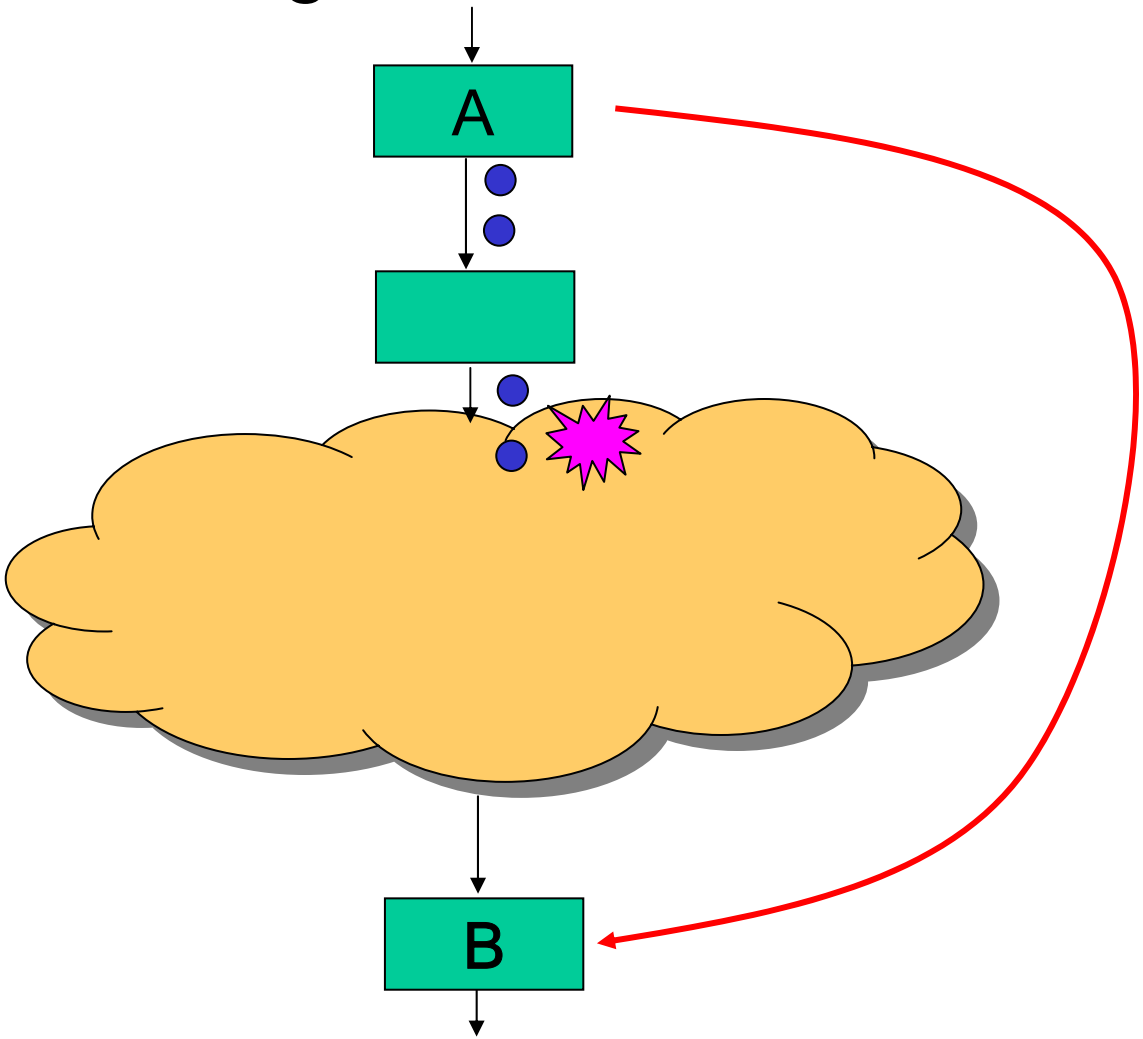
Message Timing

- A sends message to B with zero latency



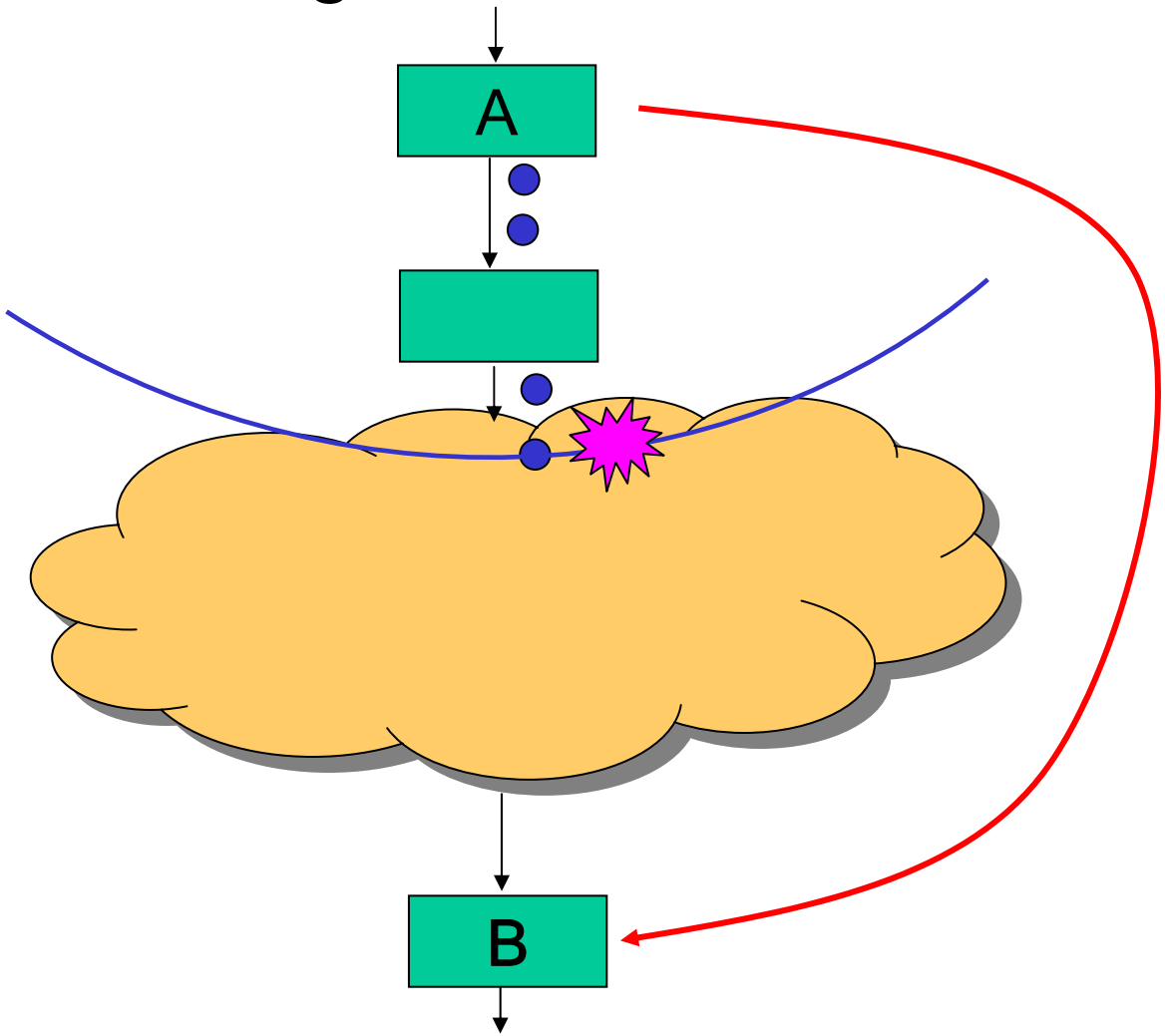
Message Timing

- A sends message to B with zero latency



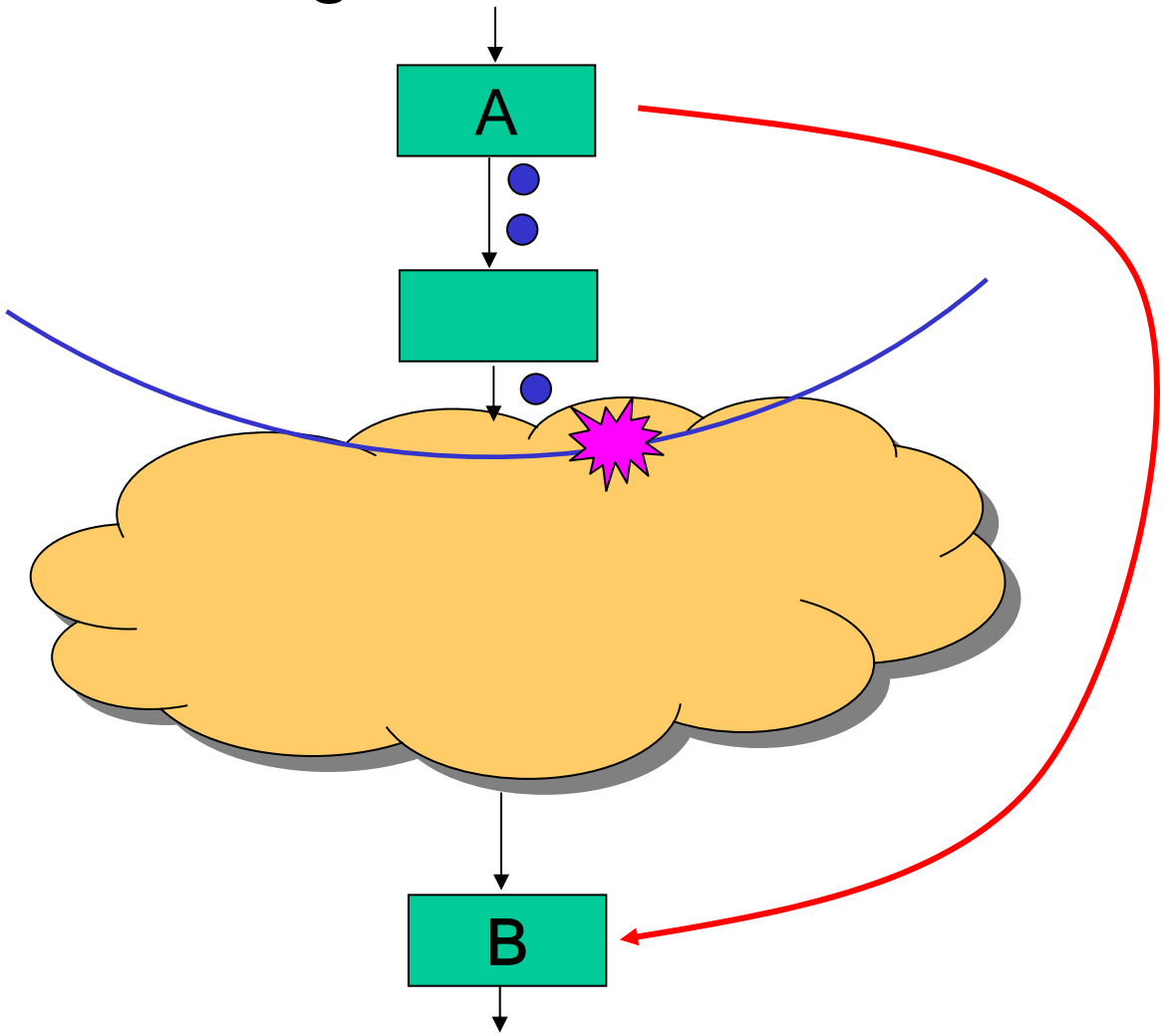
Message Timing

- A sends message to B with zero latency



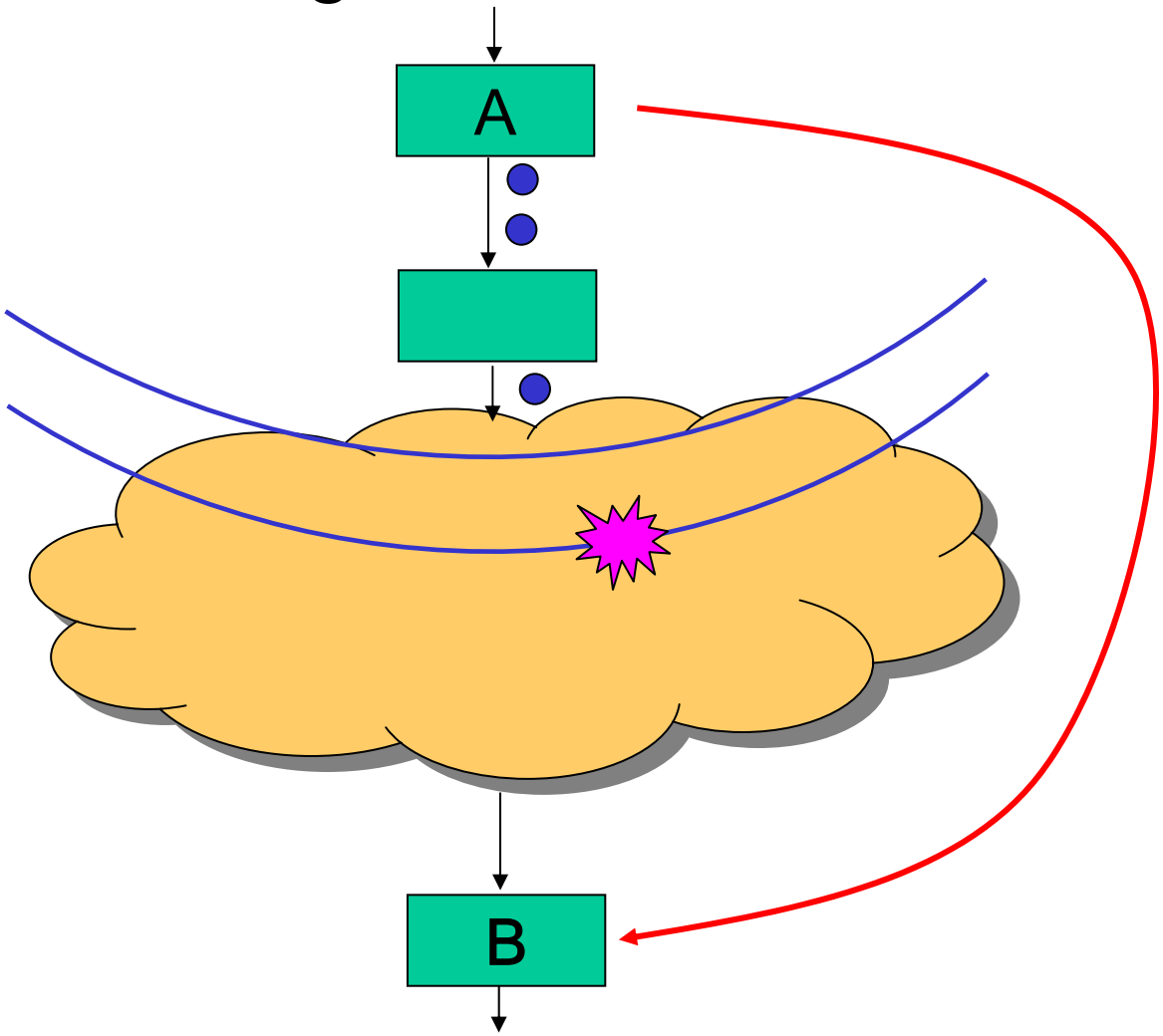
Message Timing

- A sends message to B with zero latency



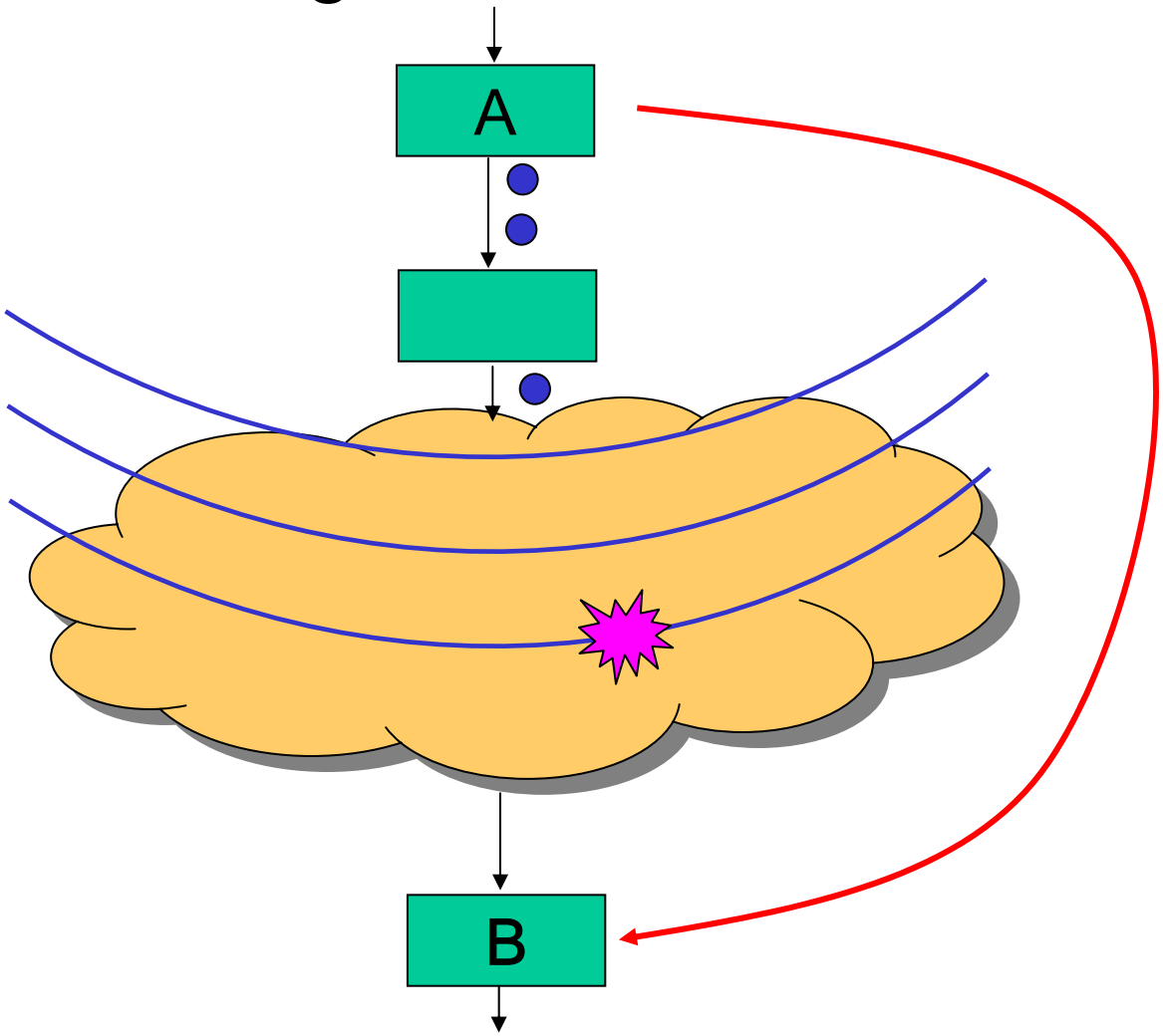
Message Timing

- A sends message to B with zero latency



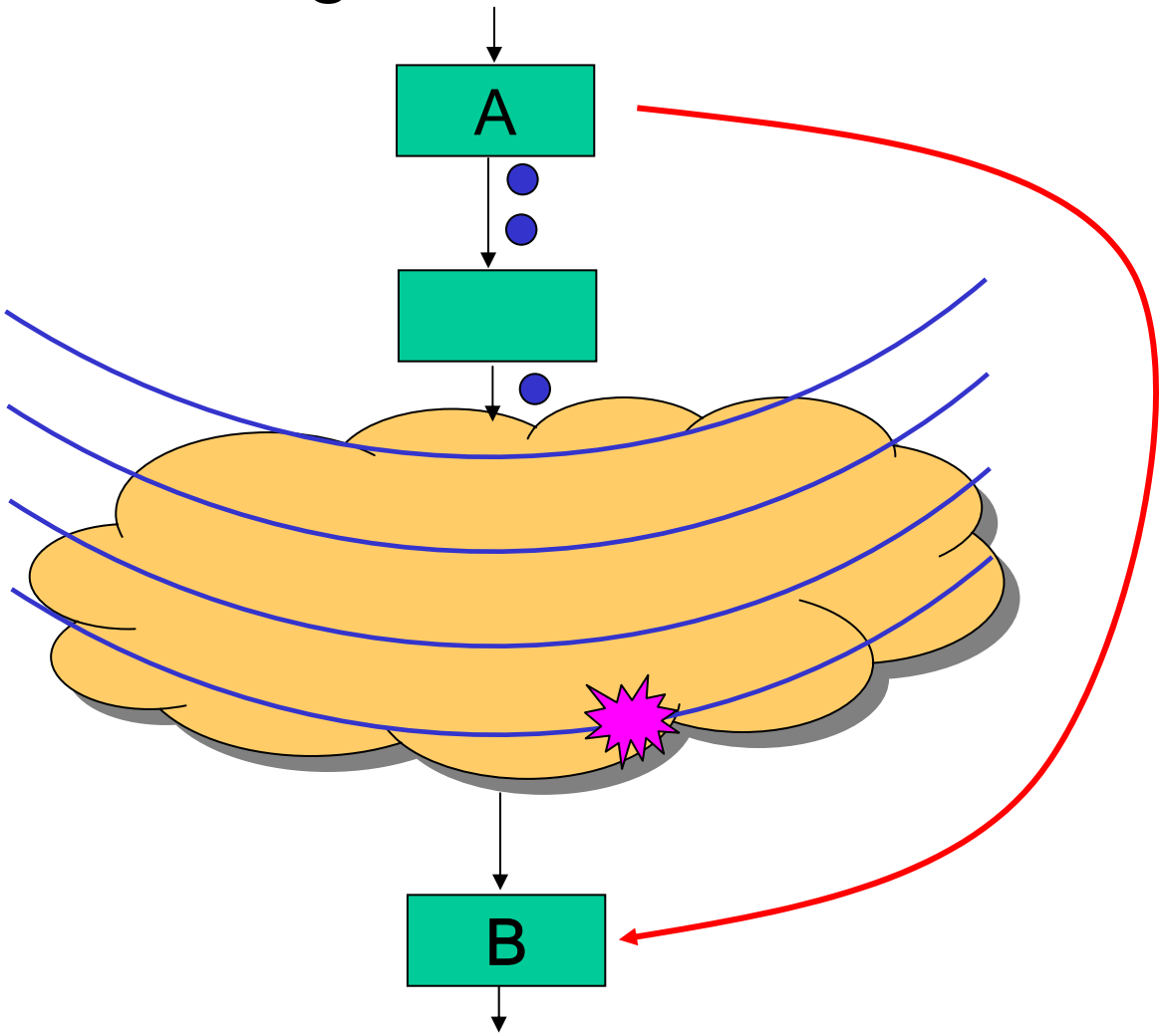
Message Timing

- A sends message to B with zero latency



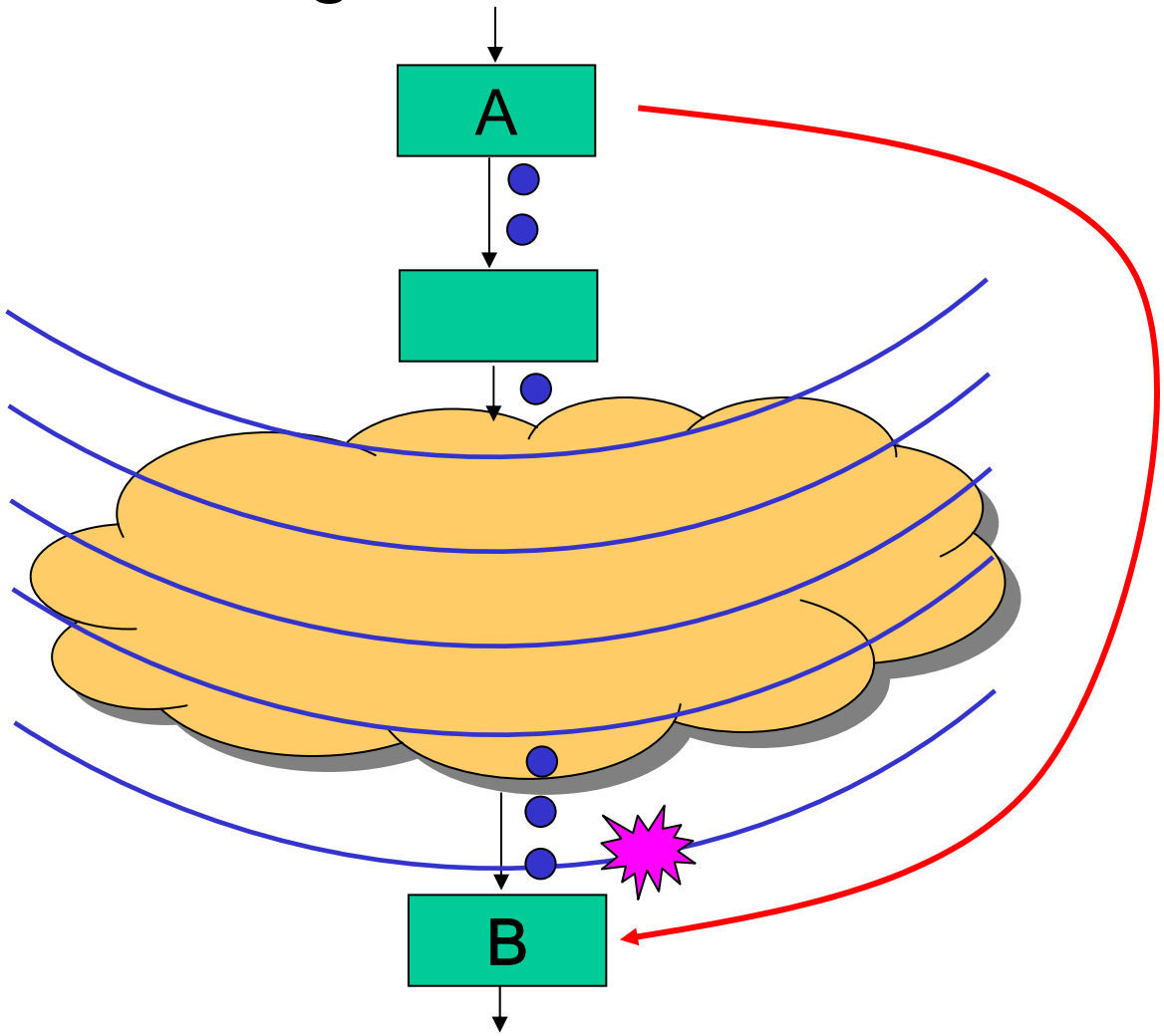
Message Timing

- A sends message to B with zero latency



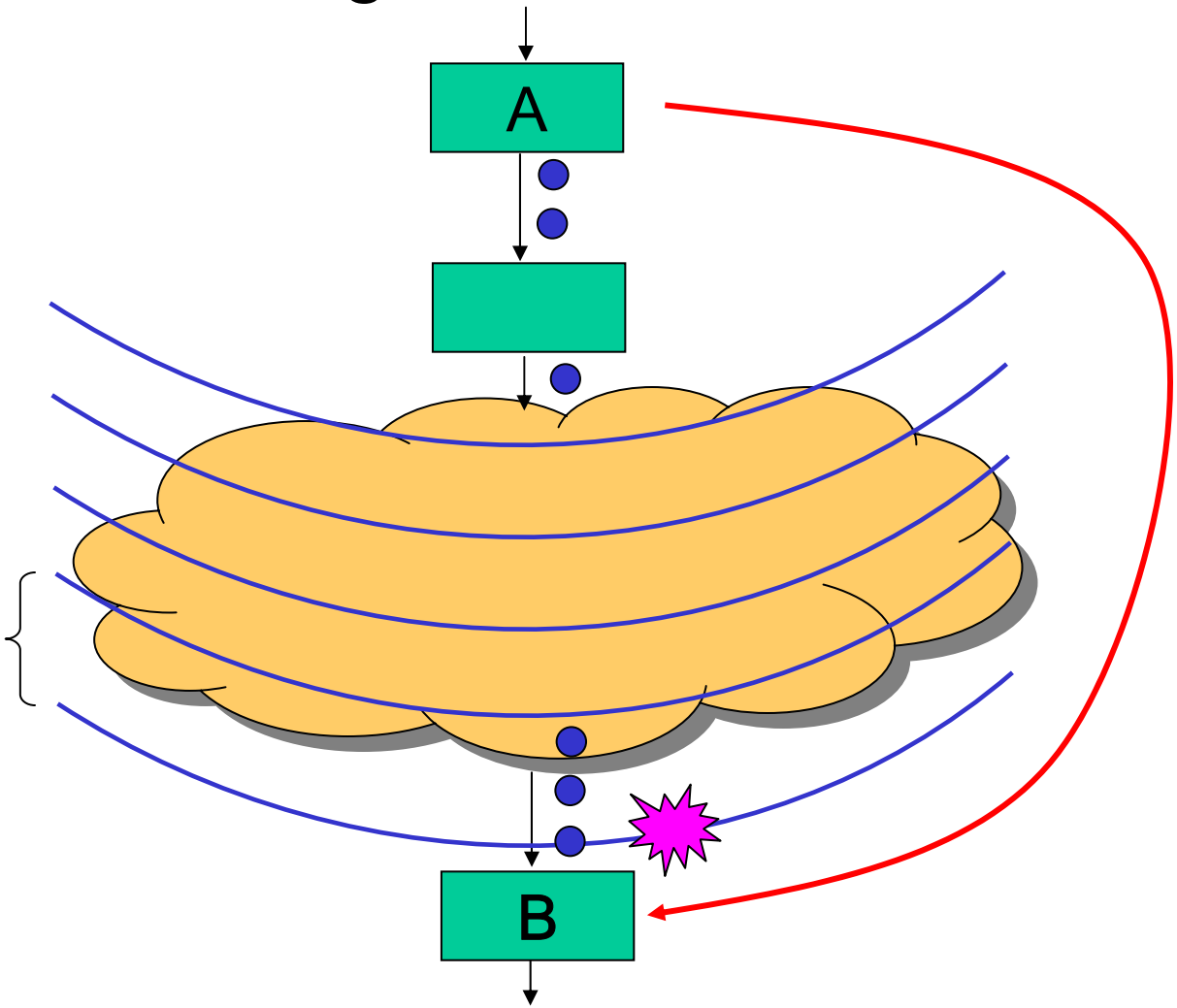
Message Timing

- A sends message to B with zero latency



Message Timing

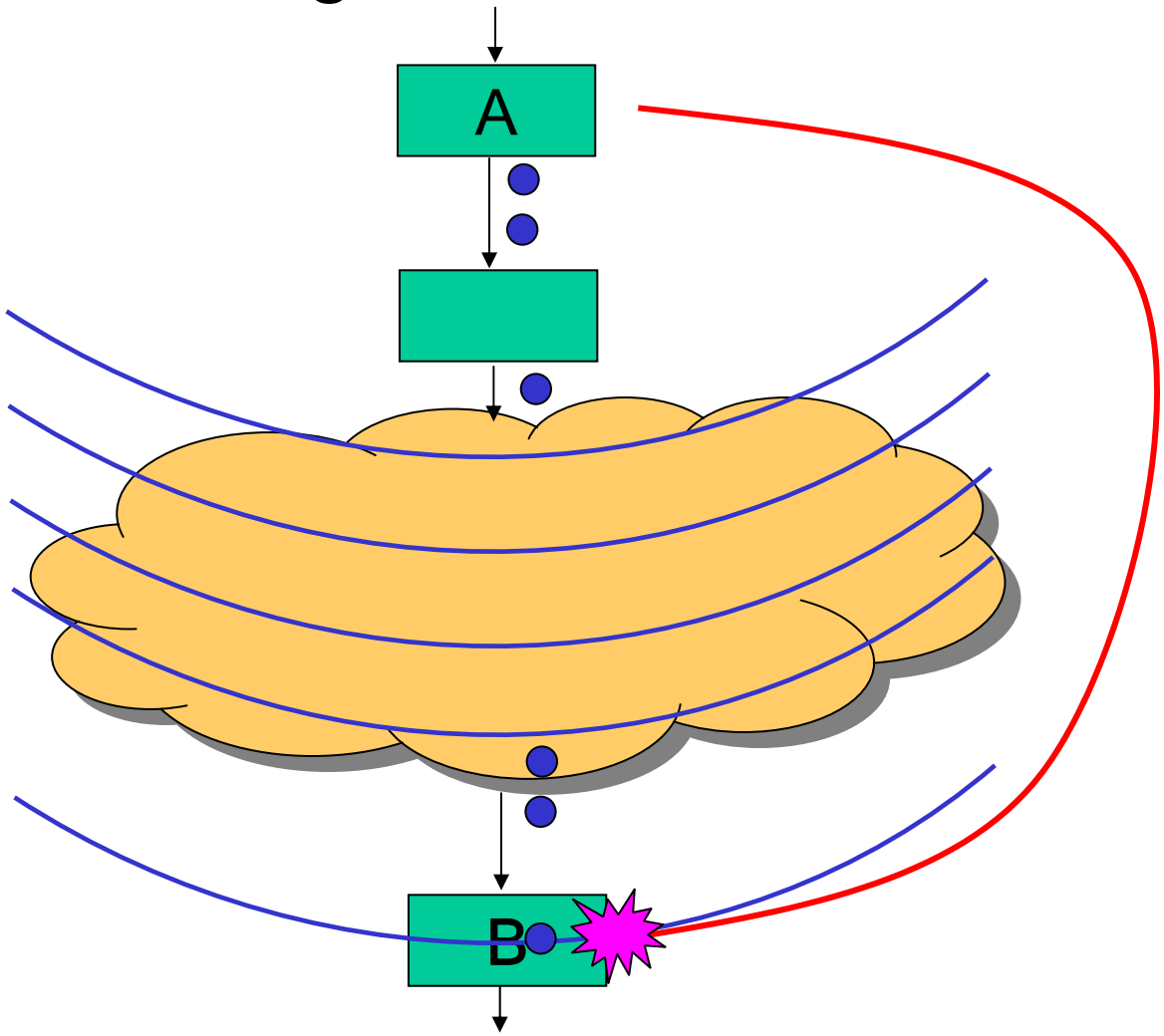
- A sends message to B with zero latency



Distance between wavefronts might have changed

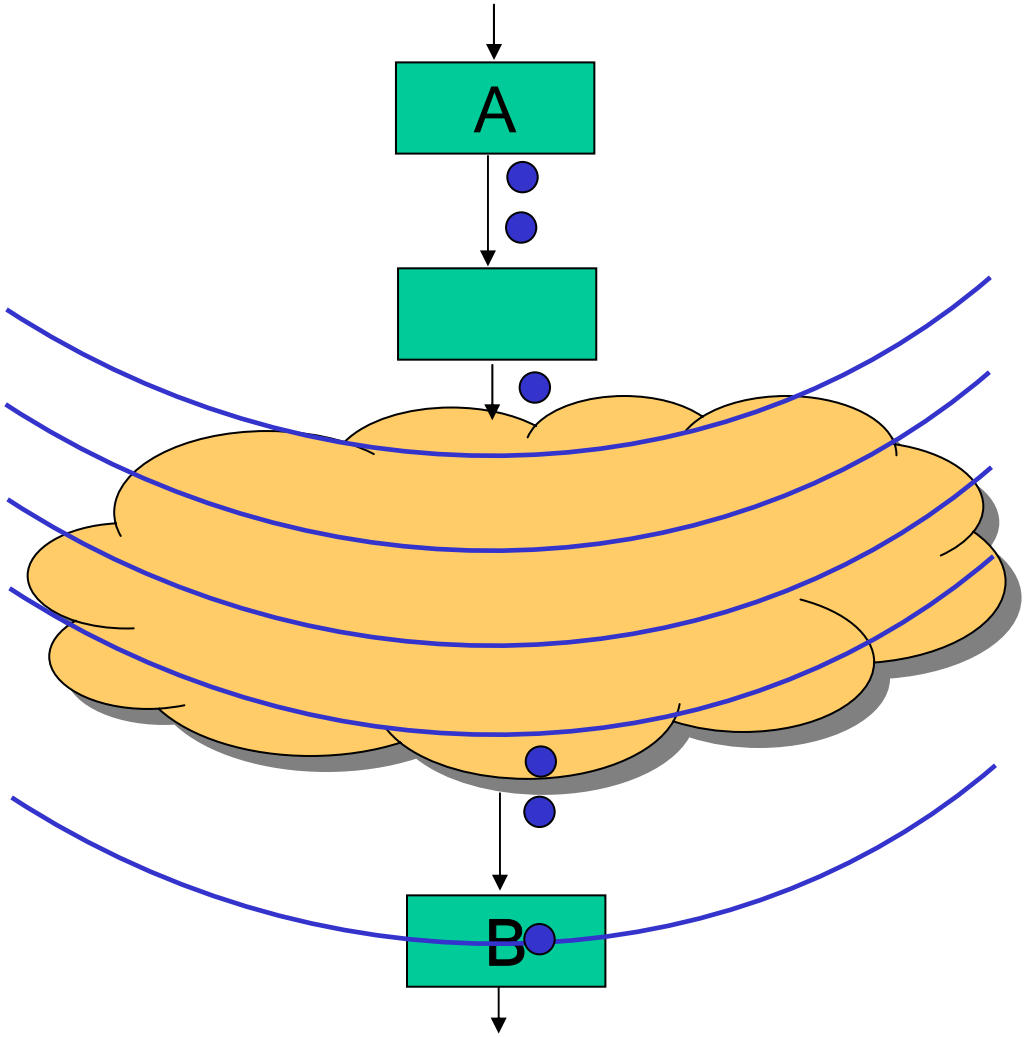
Message Timing

- A sends message to B with zero latency



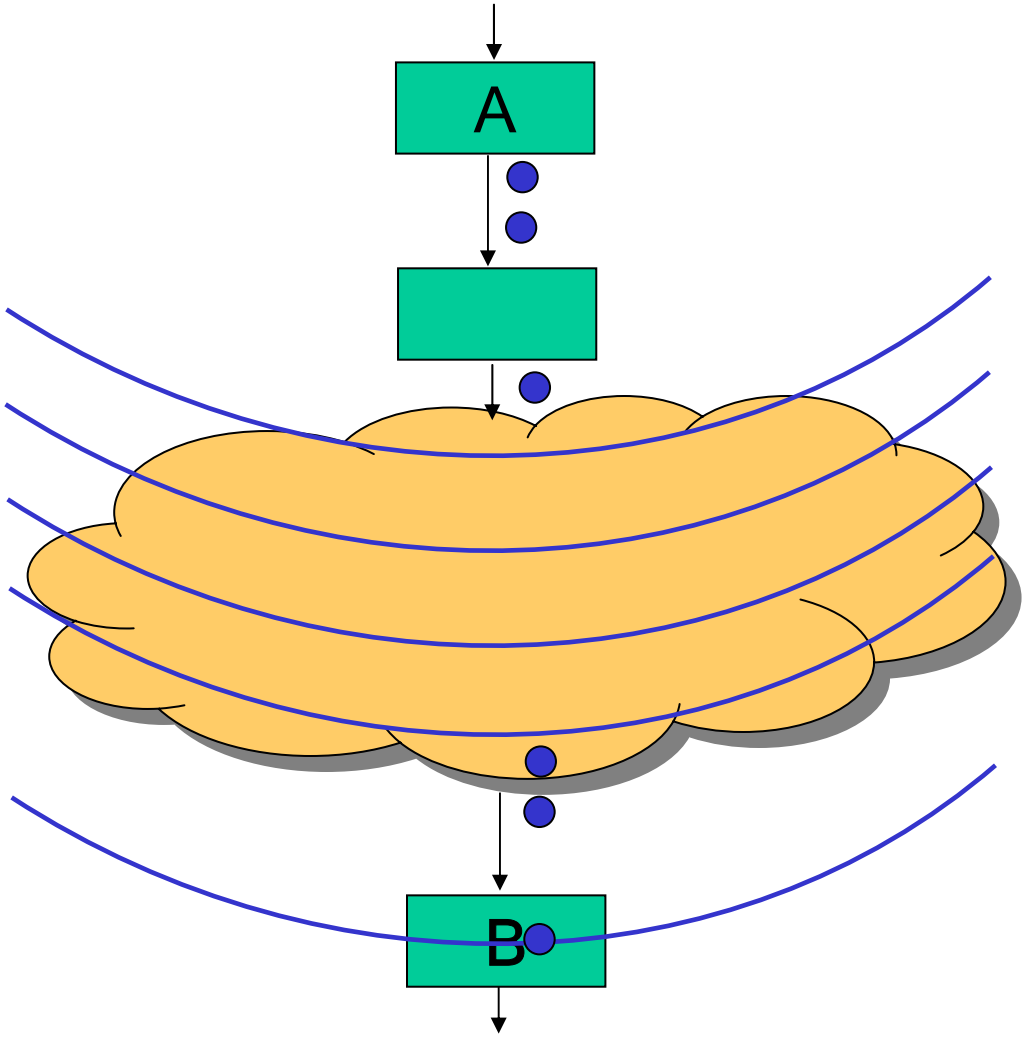
General Message Timing

- Latency of N means:
  - Message attached to wavefront that *sender* sees in N executions



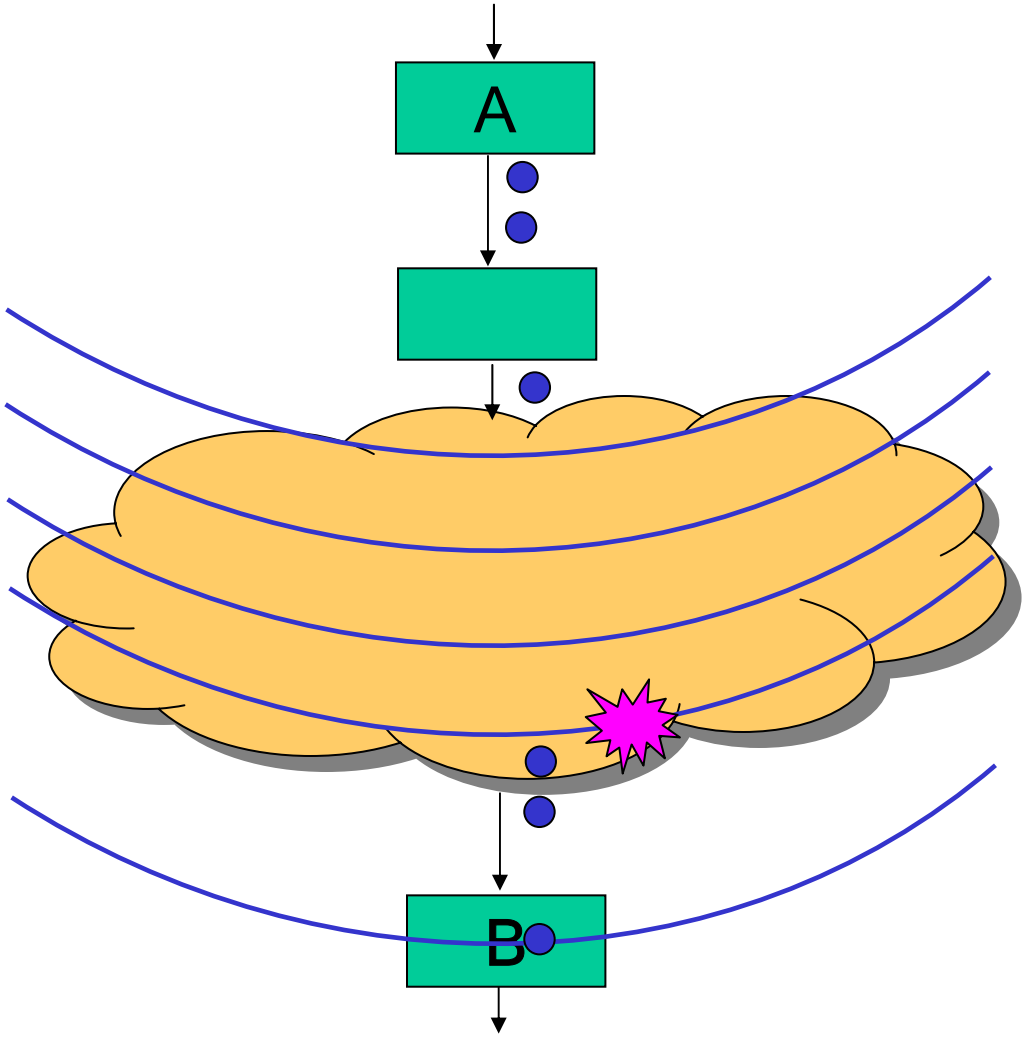
General Message Timing

- Latency of N means:
  - Message attached to wavefront that *sender* sees in N executions
- Examples:
  - A → B, latency 1



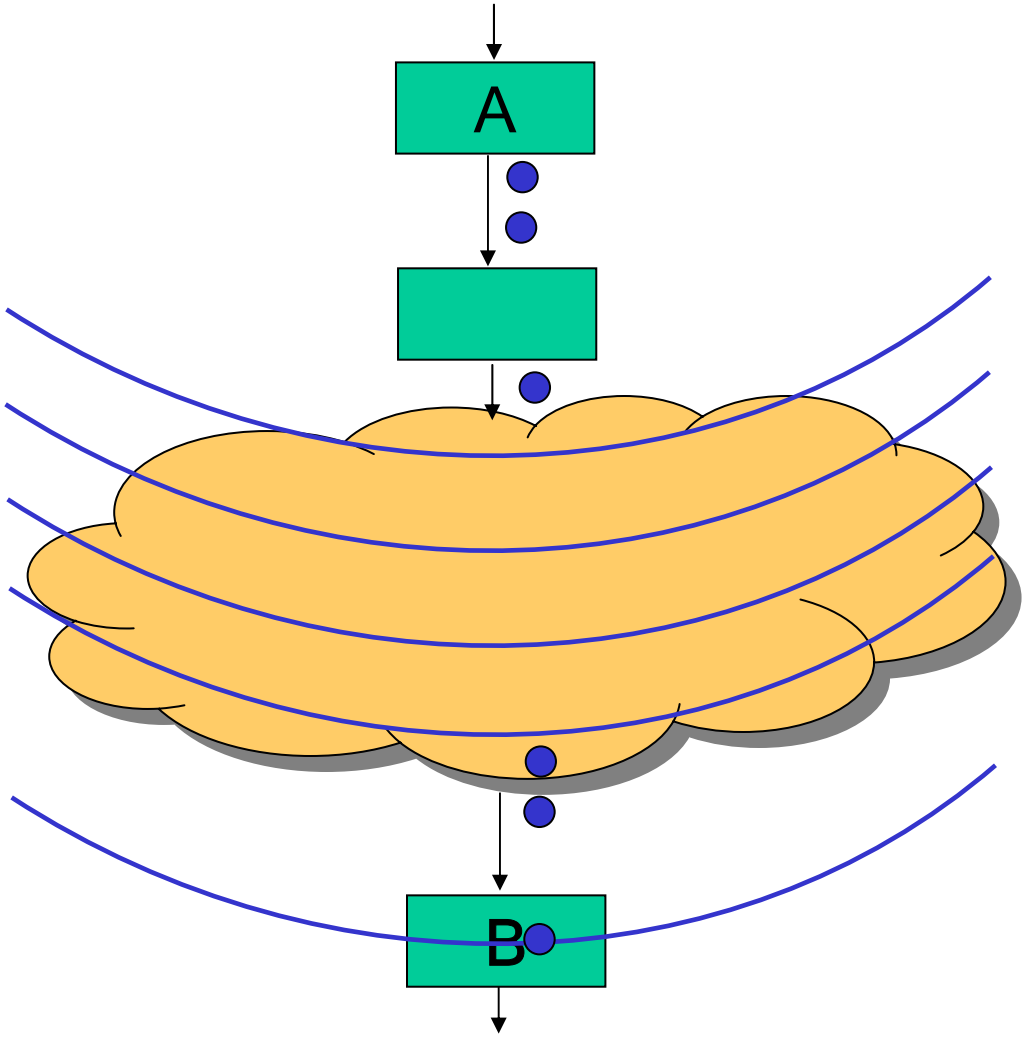
General Message Timing

- Latency of N means:
  - Message attached to wavefront that *sender* sees in N executions
- Examples:
  - A → B, latency 1



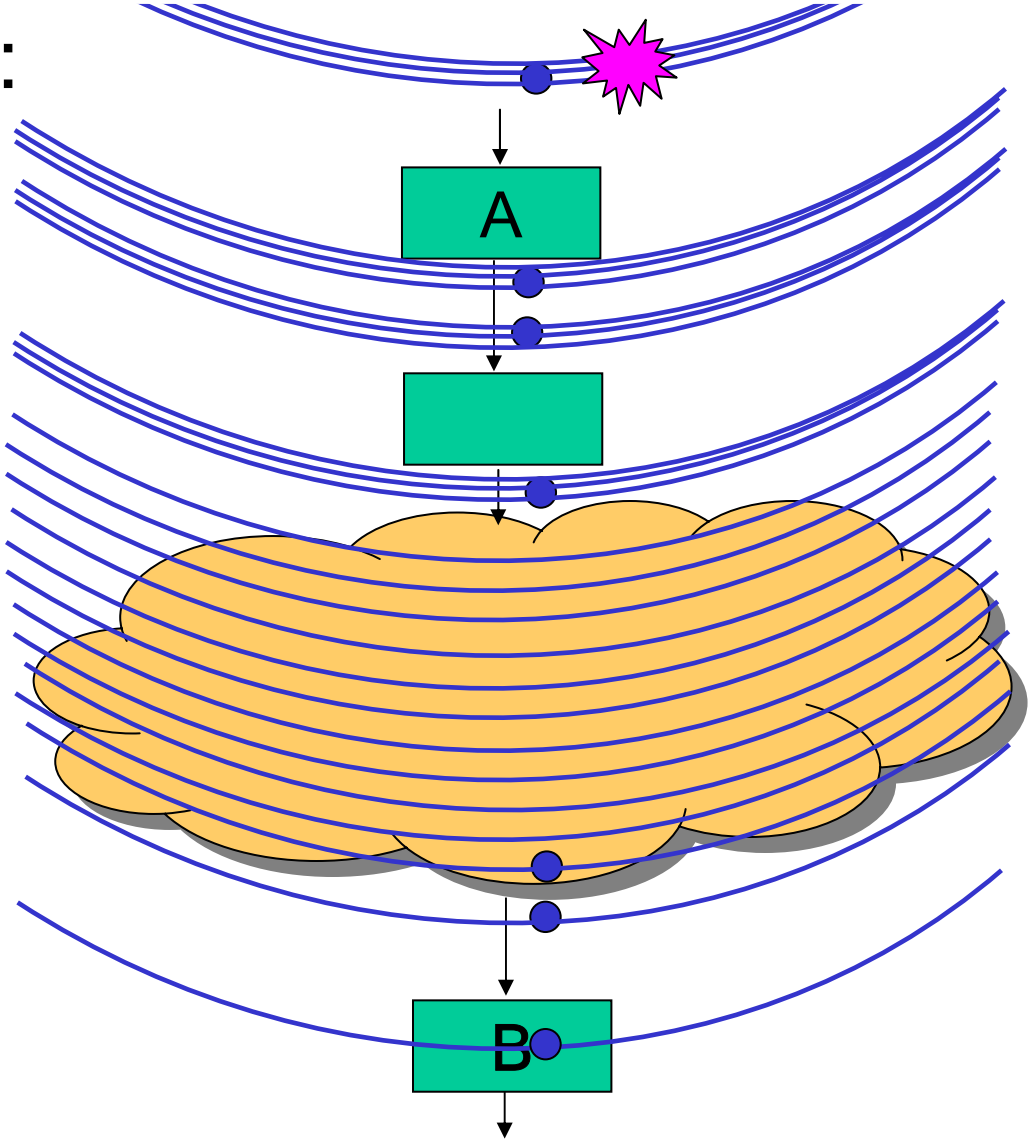
General Message Timing

- Latency of N means:
  - Message attached to wavefront that *sender* sees in N executions
- Examples:
  - A → B, latency 1
  - B → A, latency 25



General Message Timing

- Latency of N means:
  - Message attached to wavefront that *sender* sees in N executions
- Examples:
  - A → B, latency 1
  - B → A, latency 25



Rationale

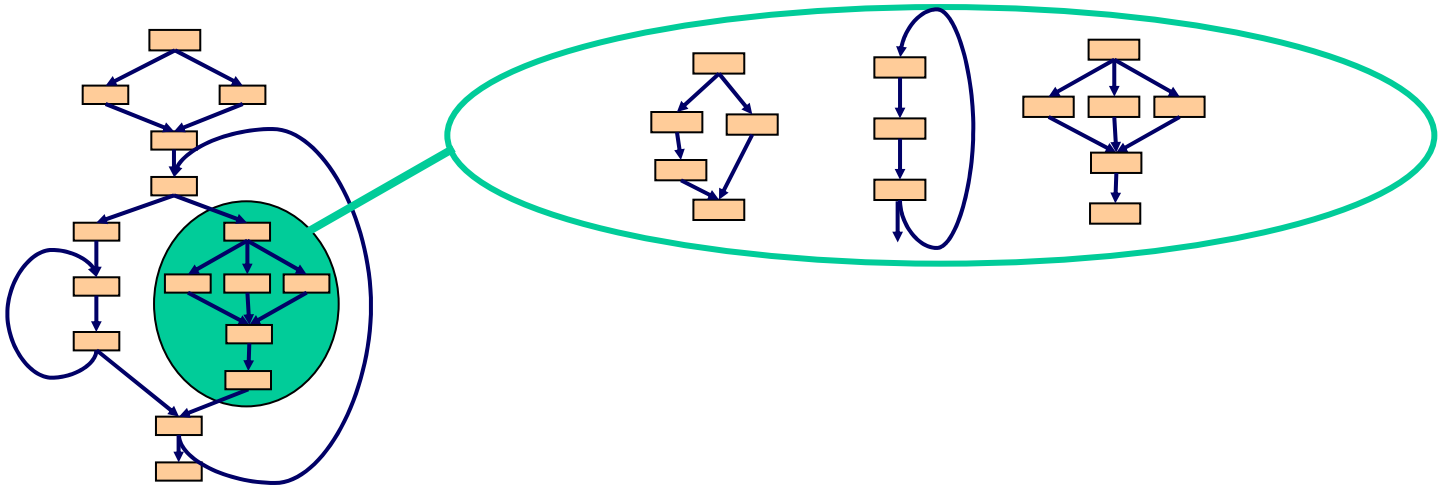
- Better for the programmer
  - Simplicity of method call
  - Precision of embedding in stream
- Better for the compiler
  - Program is easier to analyze
    - No code for timing / embedding
    - No control channels in stream graph
  - Can reorder filter firings, respecting constraints
  - Implement in most efficient way

Outline

- Design of Streamlt
  - Structured Streams
  - Messaging
  - Morphing
- Results
- Conclusions

Dynamic Changes to Stream

- Stream structure needs to change

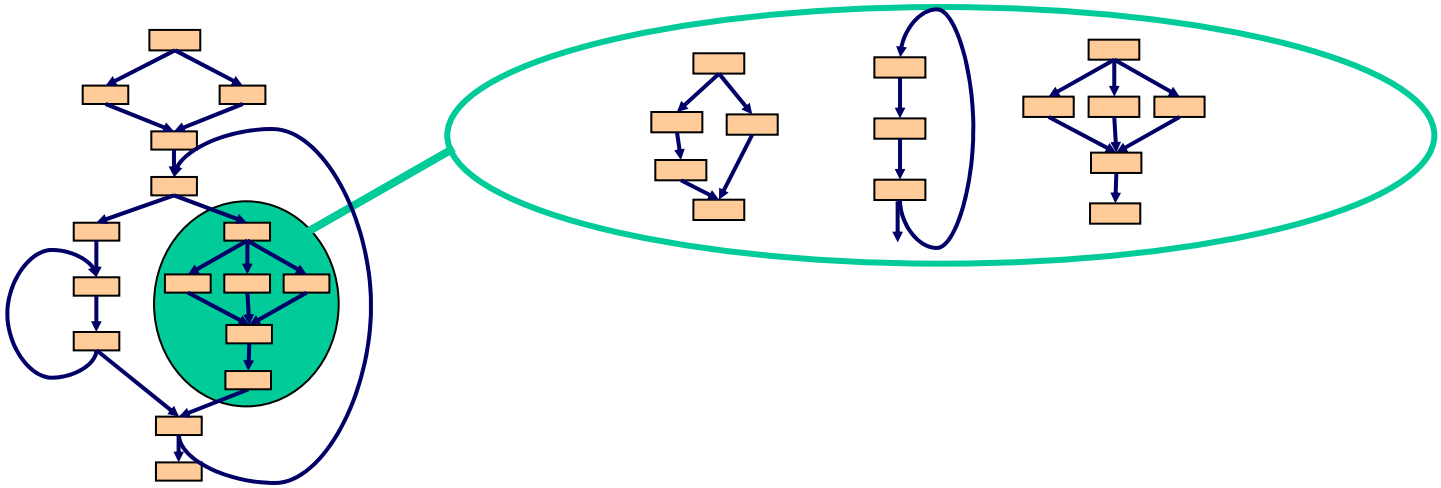


- Examples
  - Switch radio from AM to FM
  - Change from Bluetooth to 802.11

Program "Morphing"

Dynamic Changes to Stream

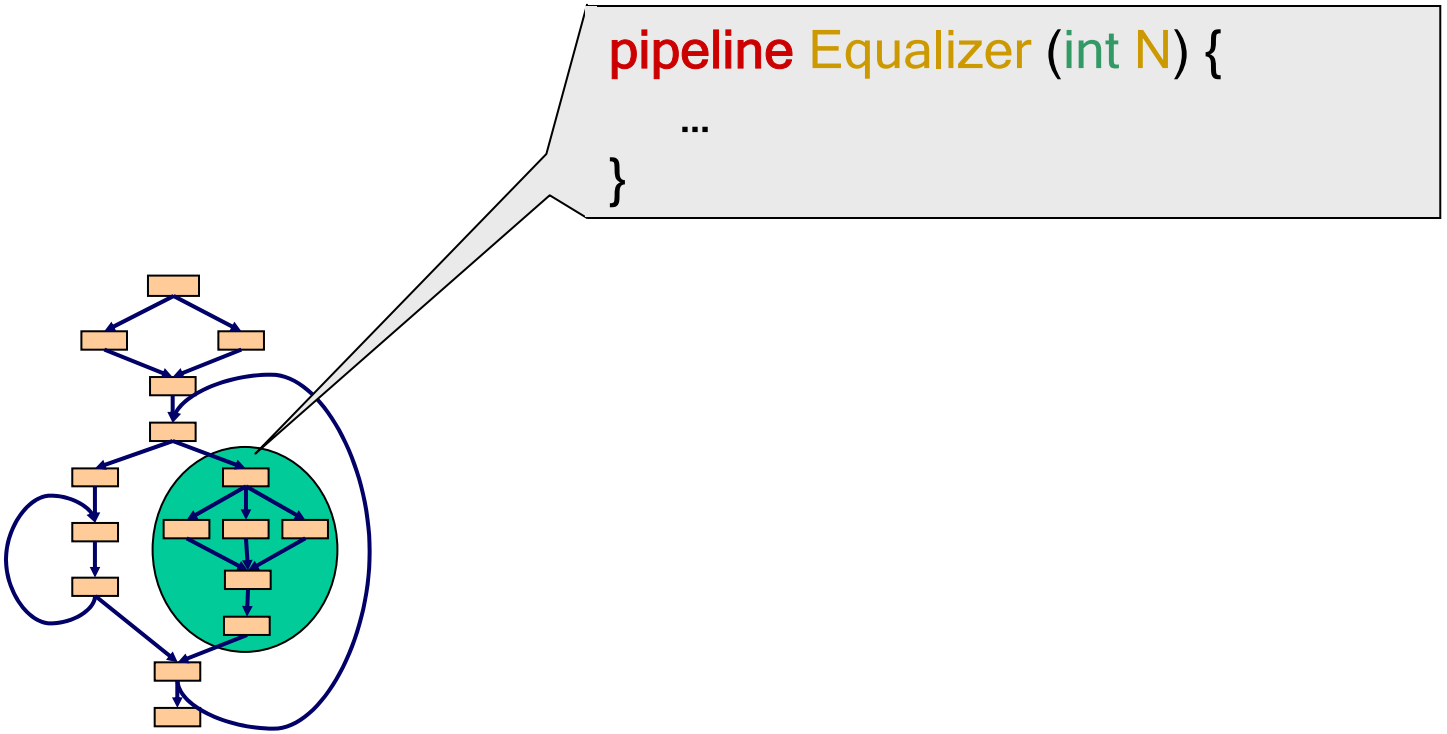
- Stream structure needs to change



- Challenges for programmer:
  - Synchronizing the beginning, end of morphing
  - Preserving live data in the system
  - Efficiency

Morphing in StreamIt

- Send message to “init” to morph a structure

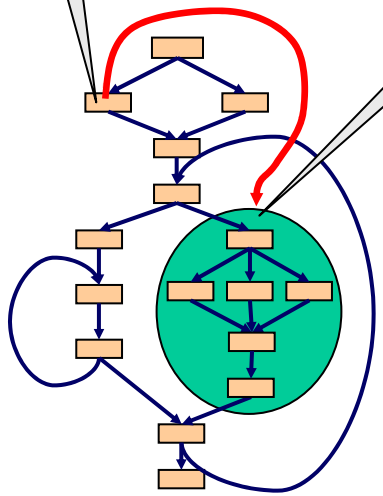


Morphing in StreamIt

- Send message to “init” to morph a structure

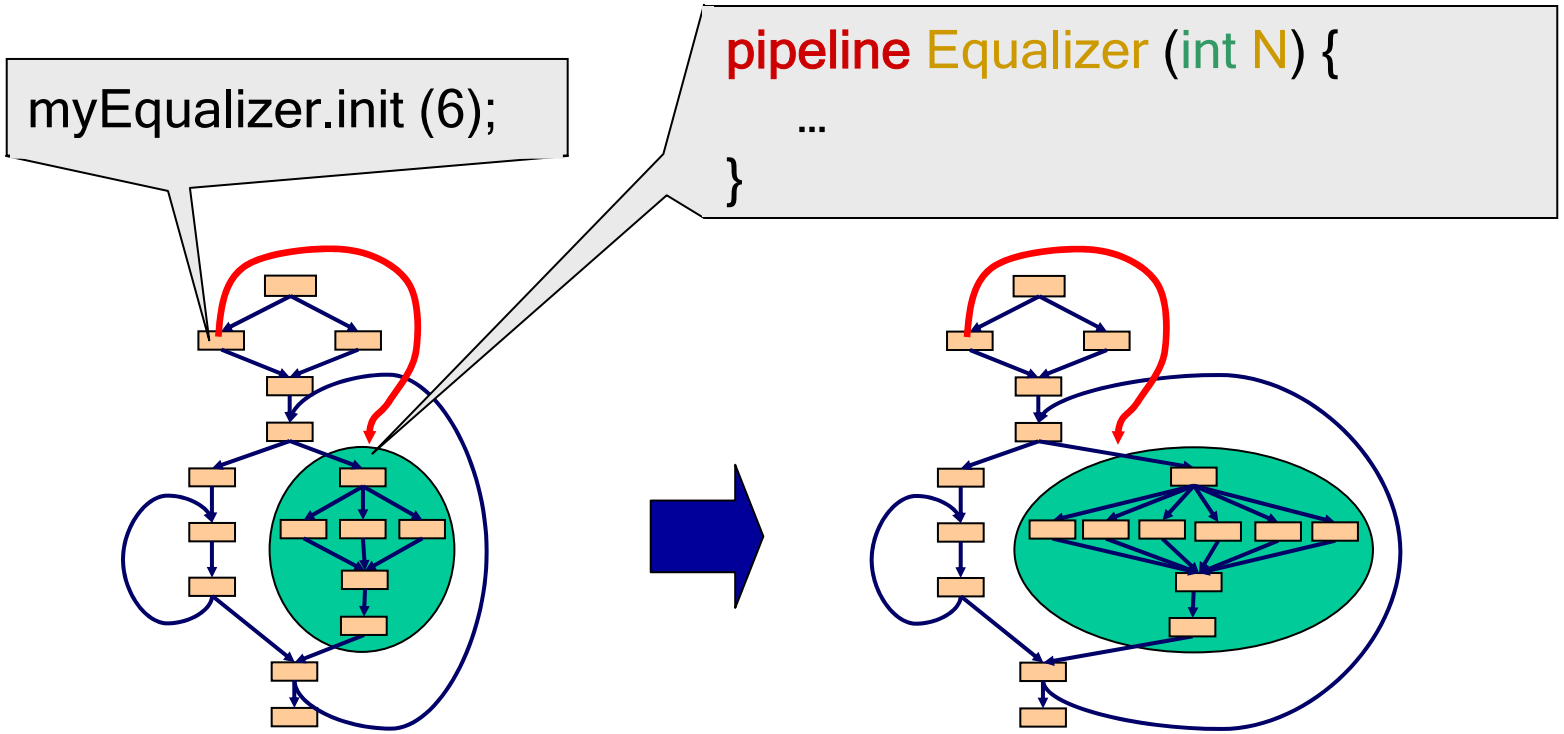
myEqualizer.init (6);

```
pipeline Equalizer (int N) {
    …
}
```



Morphing in StreamIt

- Send message to “init” to morph a structure



  - When message arrives, structure is replaced
  - Live data is automatically drained

Rationale

- Programmer writes “init” only once
  - No need for complicated transitions
- Compiler optimizes each phase separately
  - Benefits from anticipation of phase changes

Outline

- Design of Streamlt
  - Structured Streams
  - Messaging
  - Morphing
- Results
- Conclusions

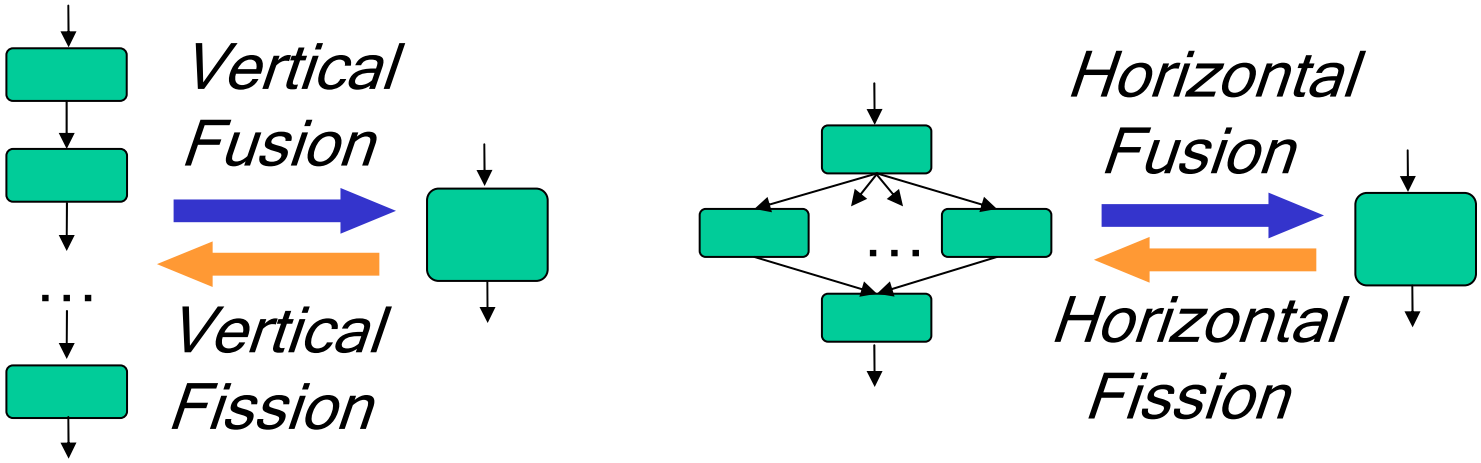
Implementation

- Basic StreamIt implementation complete
- Backends:
  - Uniprocessor
  - Raw: A tiled architecture with fine-grained, programmable communication

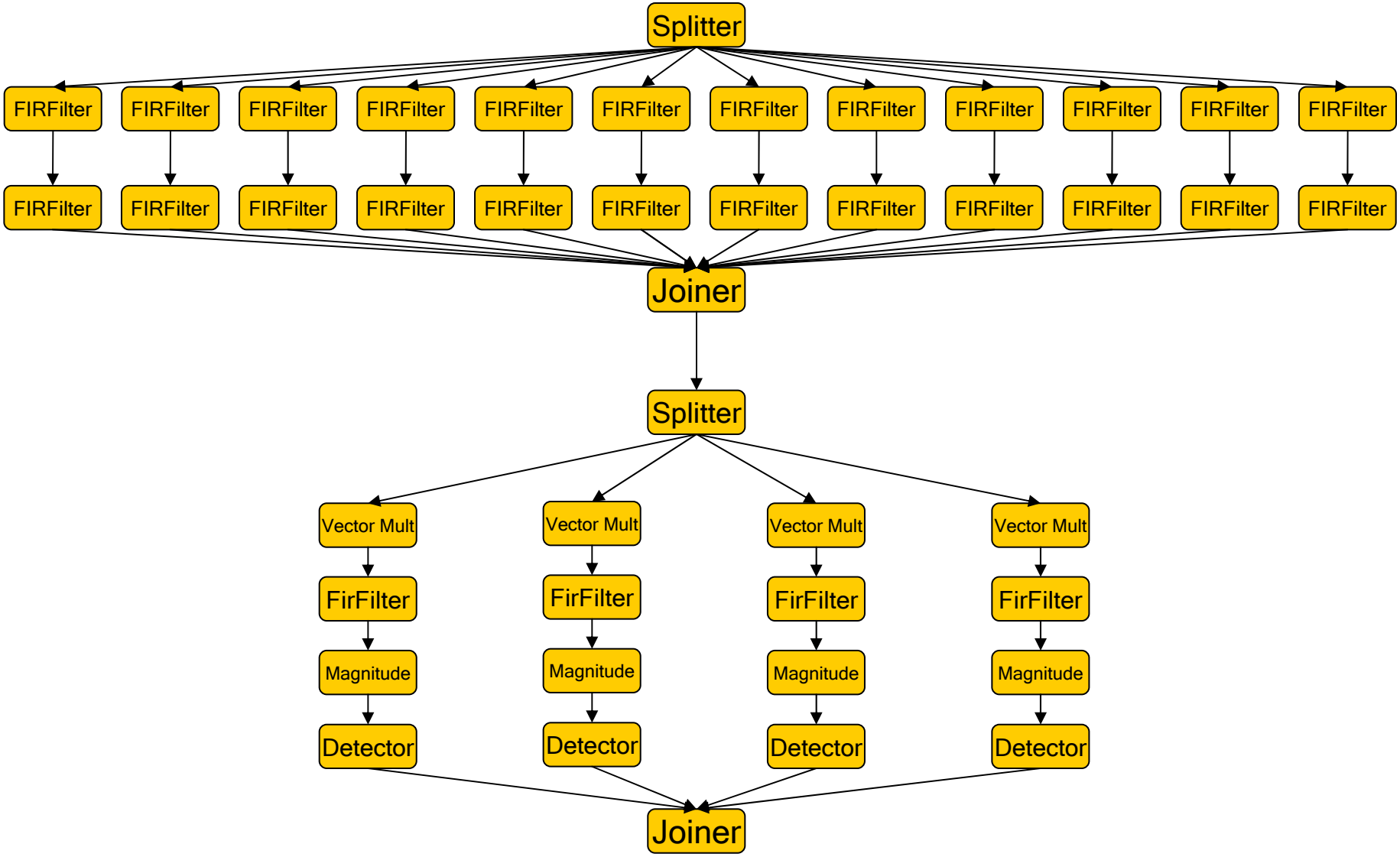
- Extended KOPI, open-source Java compiler

Results

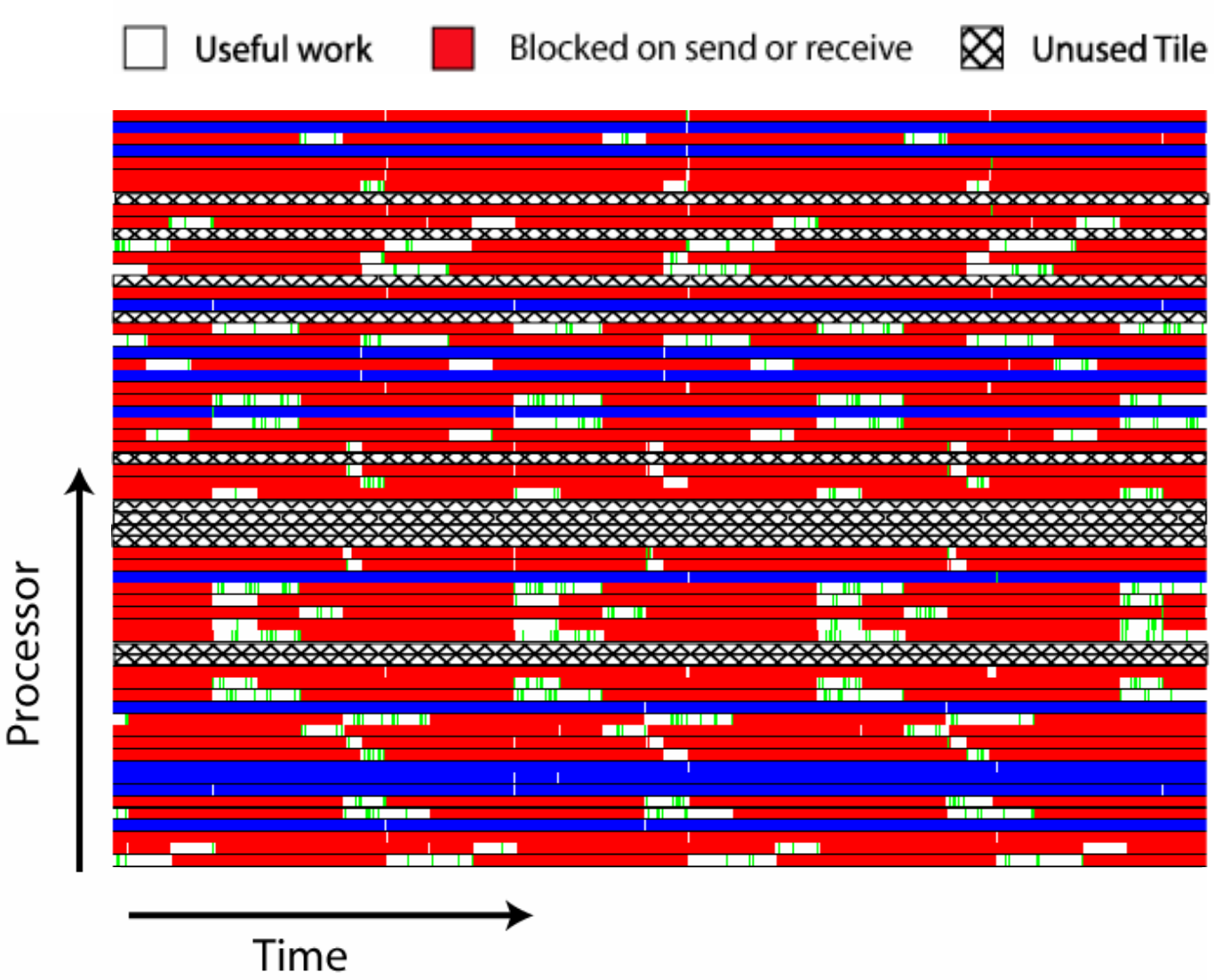
- Developed applications in StreamIt
  - GSM Decoder
  - FM Radio
  - Radar
  - FFT
  - 3GPP Channel Decoder
  - Bitonic Sort
- Load-balancing transformations improve performance on RAW



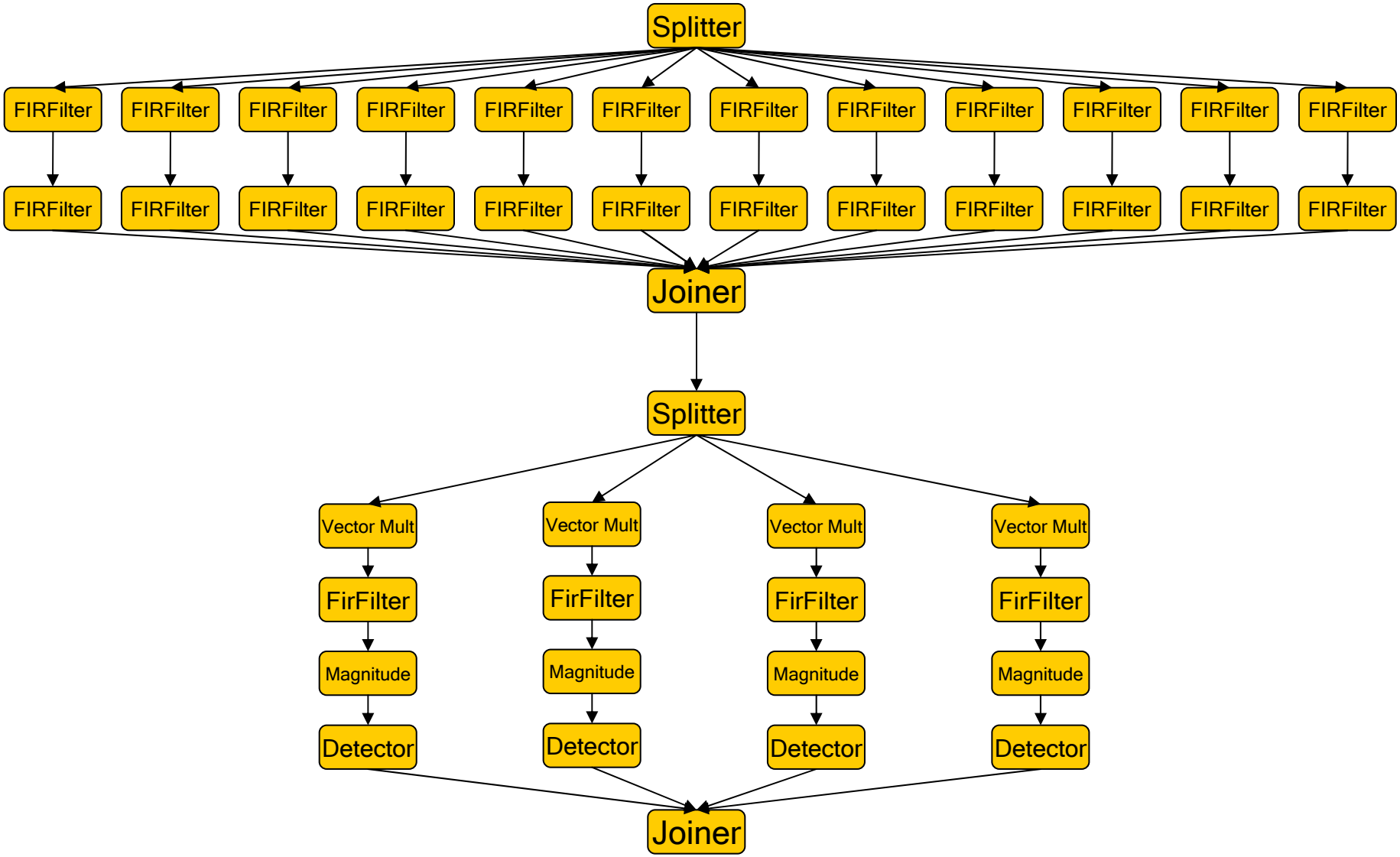
Example: Radar App. (Original)



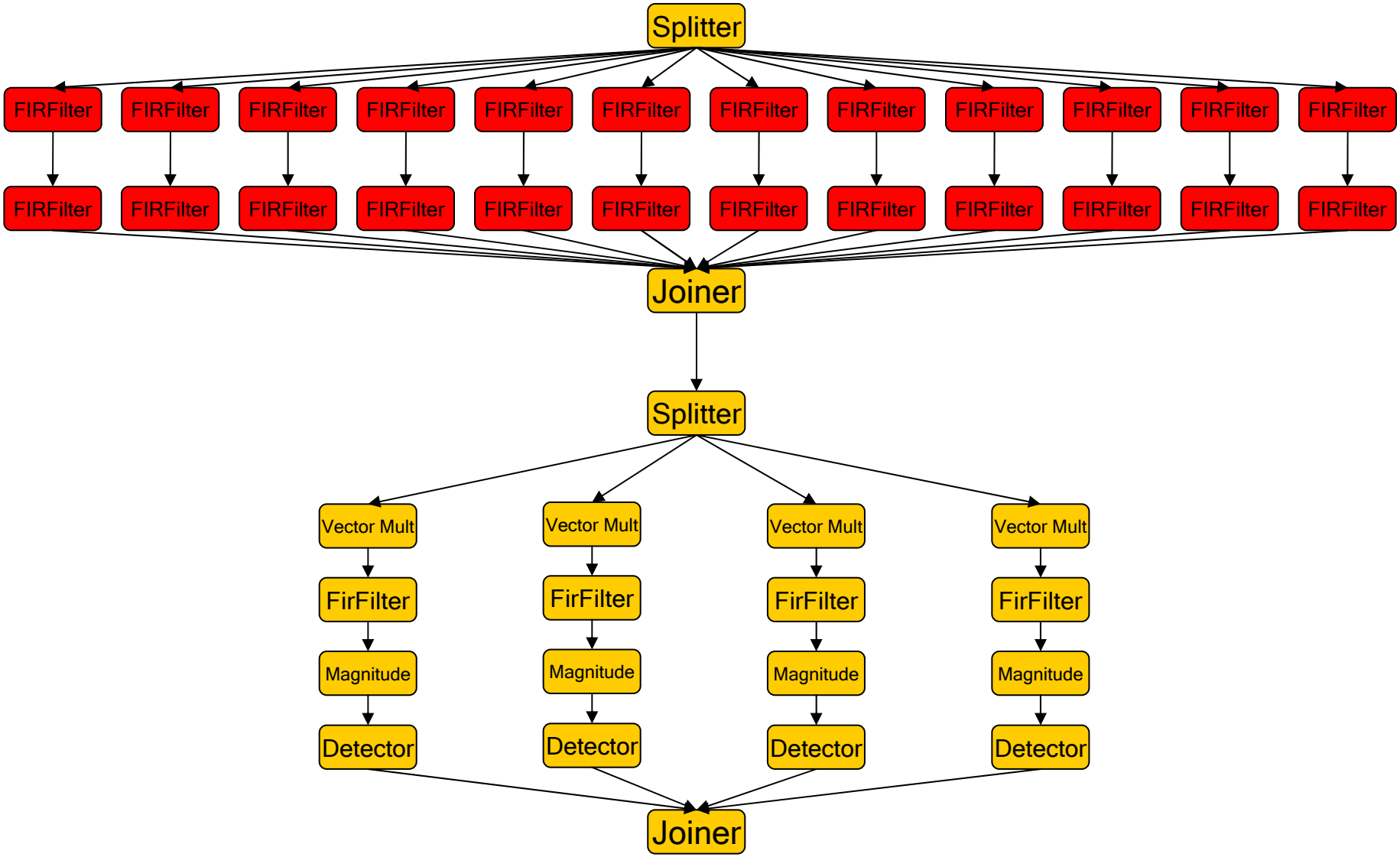
Example: Radar App. (Original)



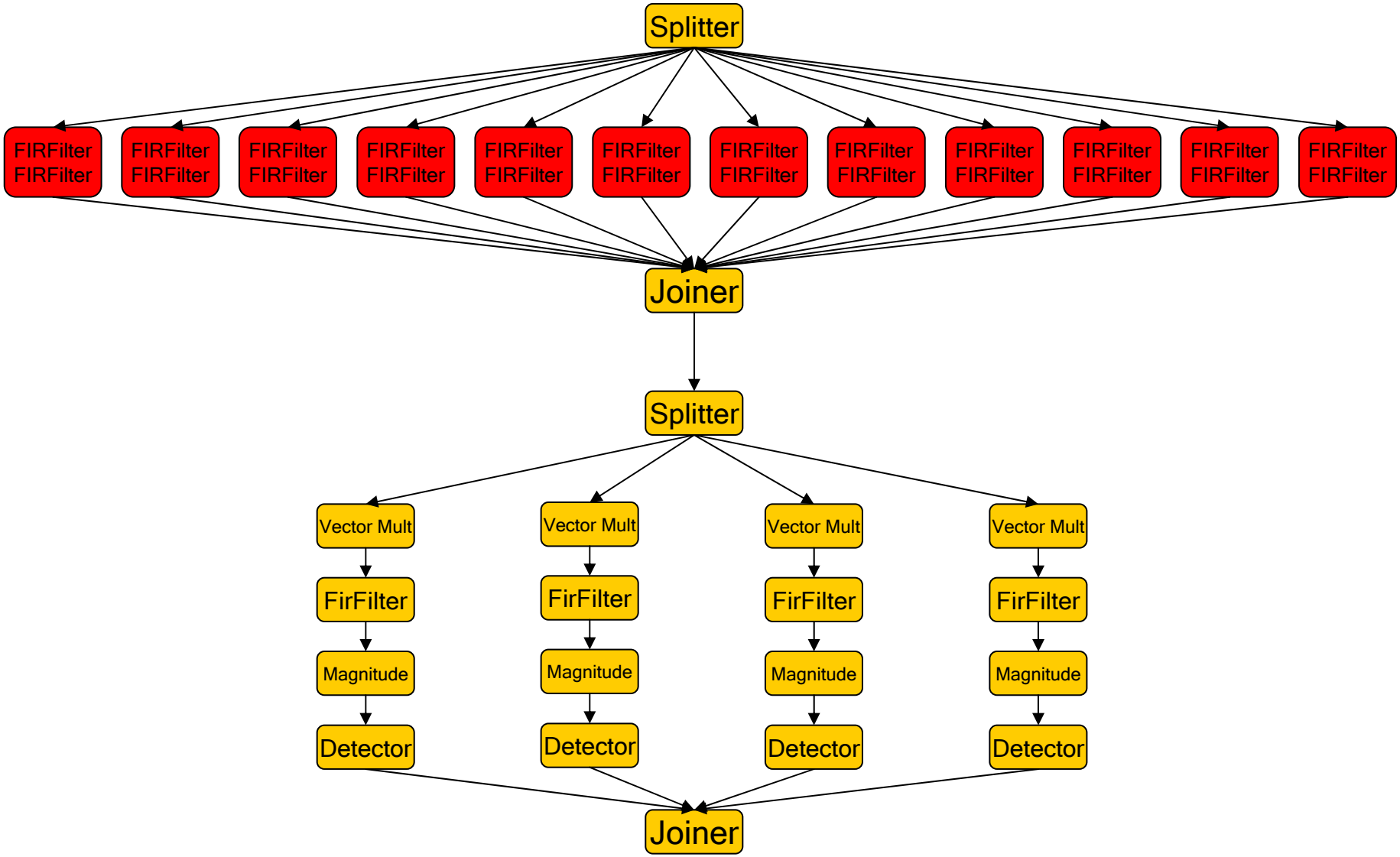
Example: Radar App. (Original)



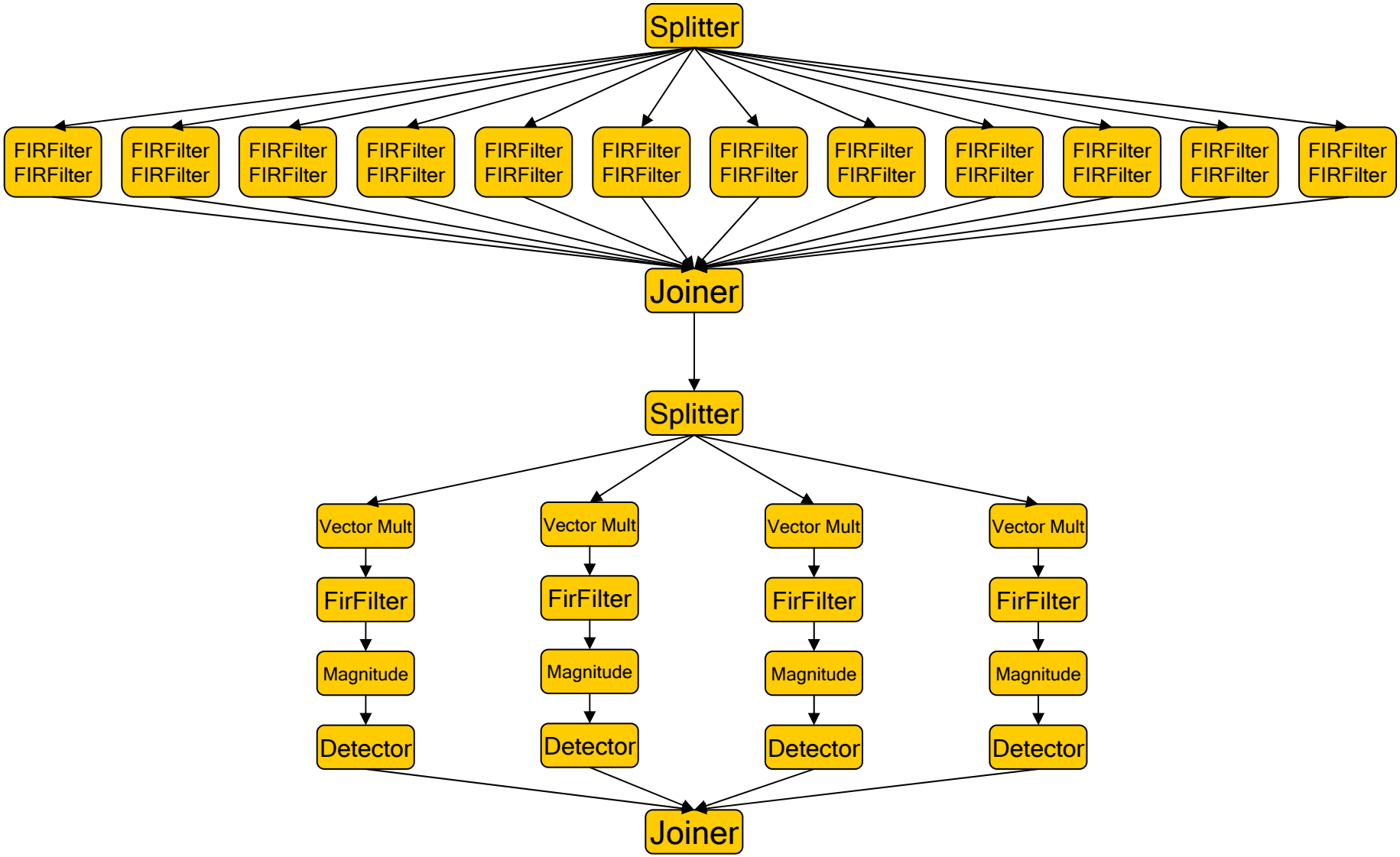
Example: Radar App. (Original)



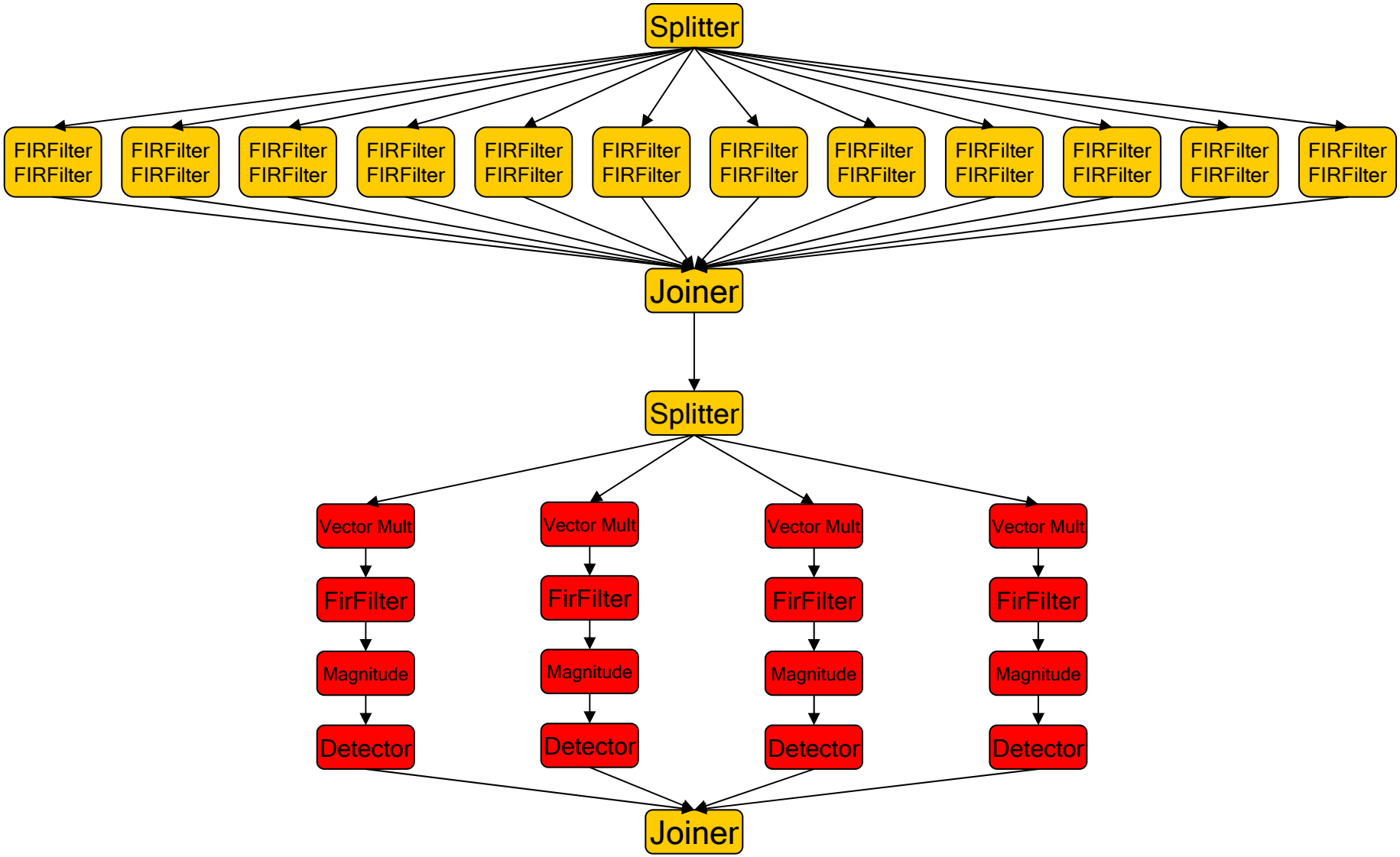
Example: Radar App.



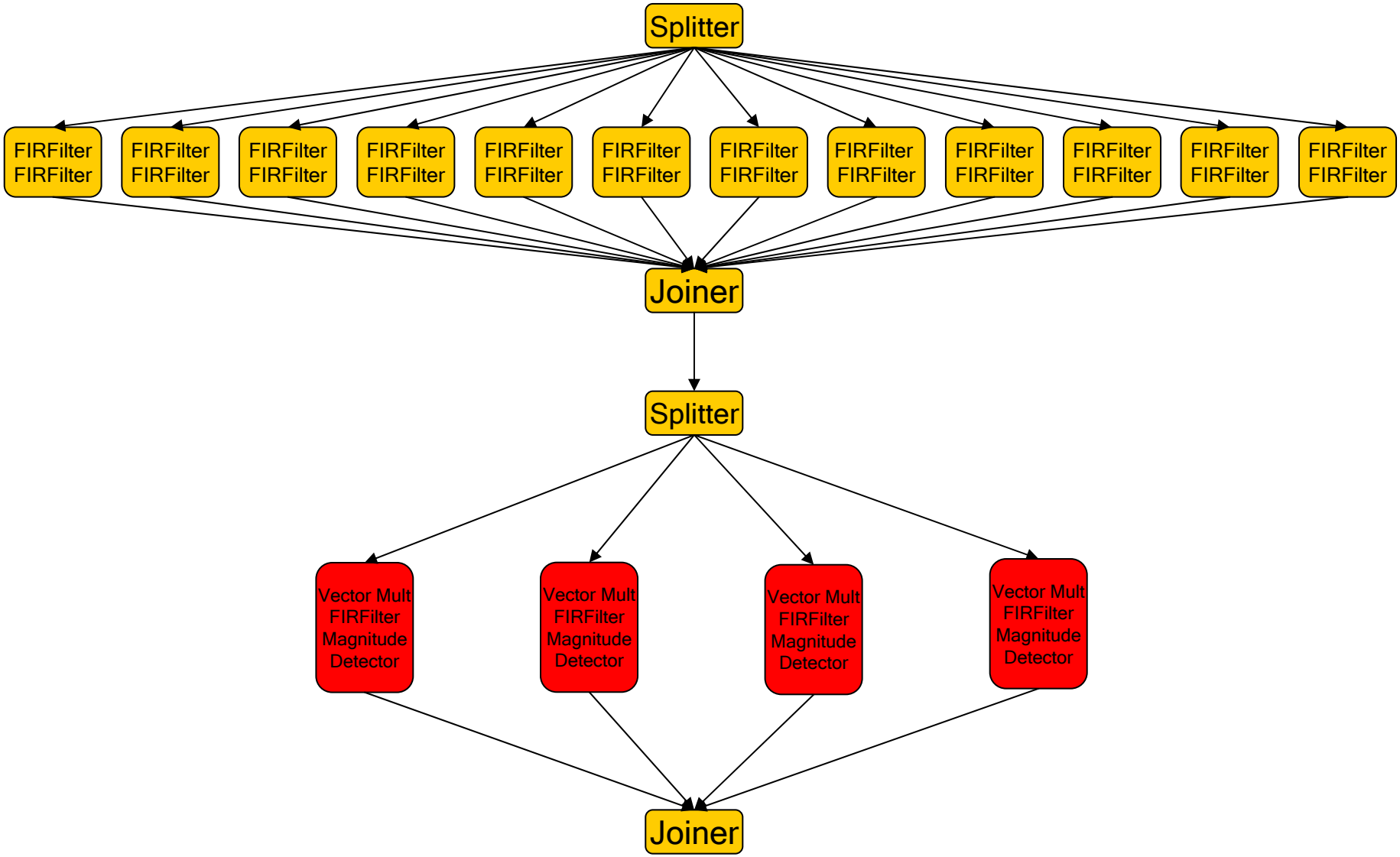
Example: Radar App.



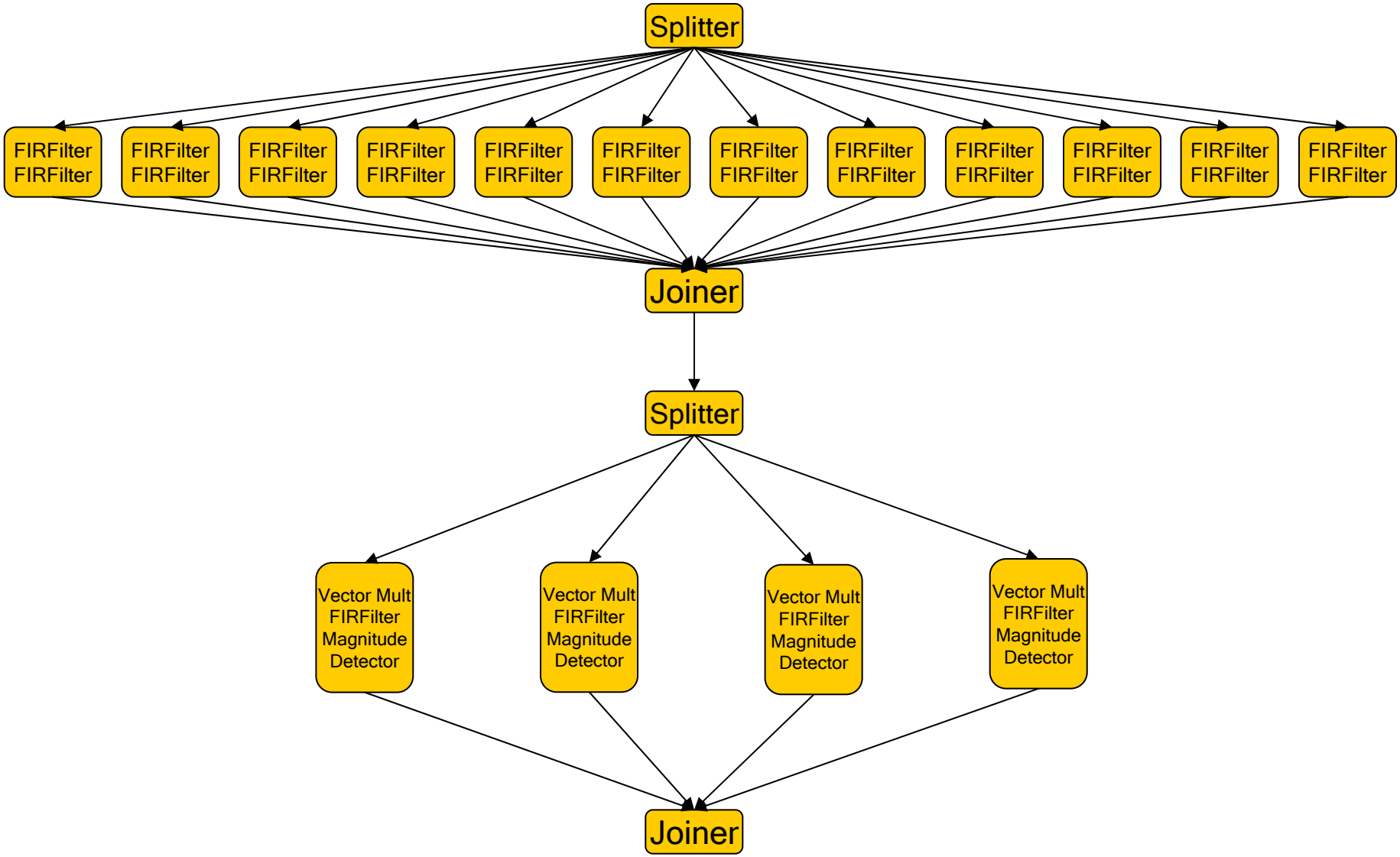
Example: Radar App.



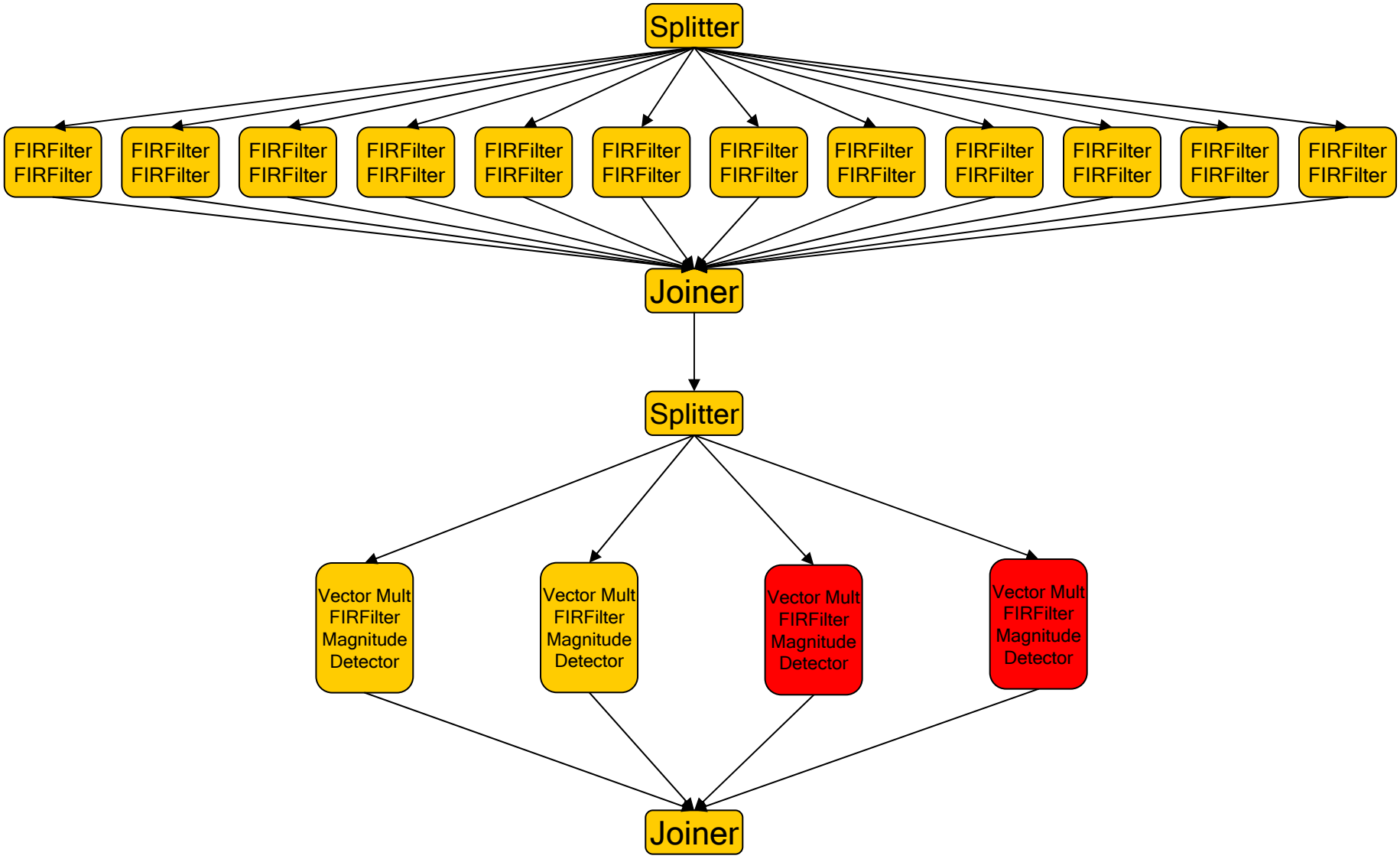
Example: Radar App.



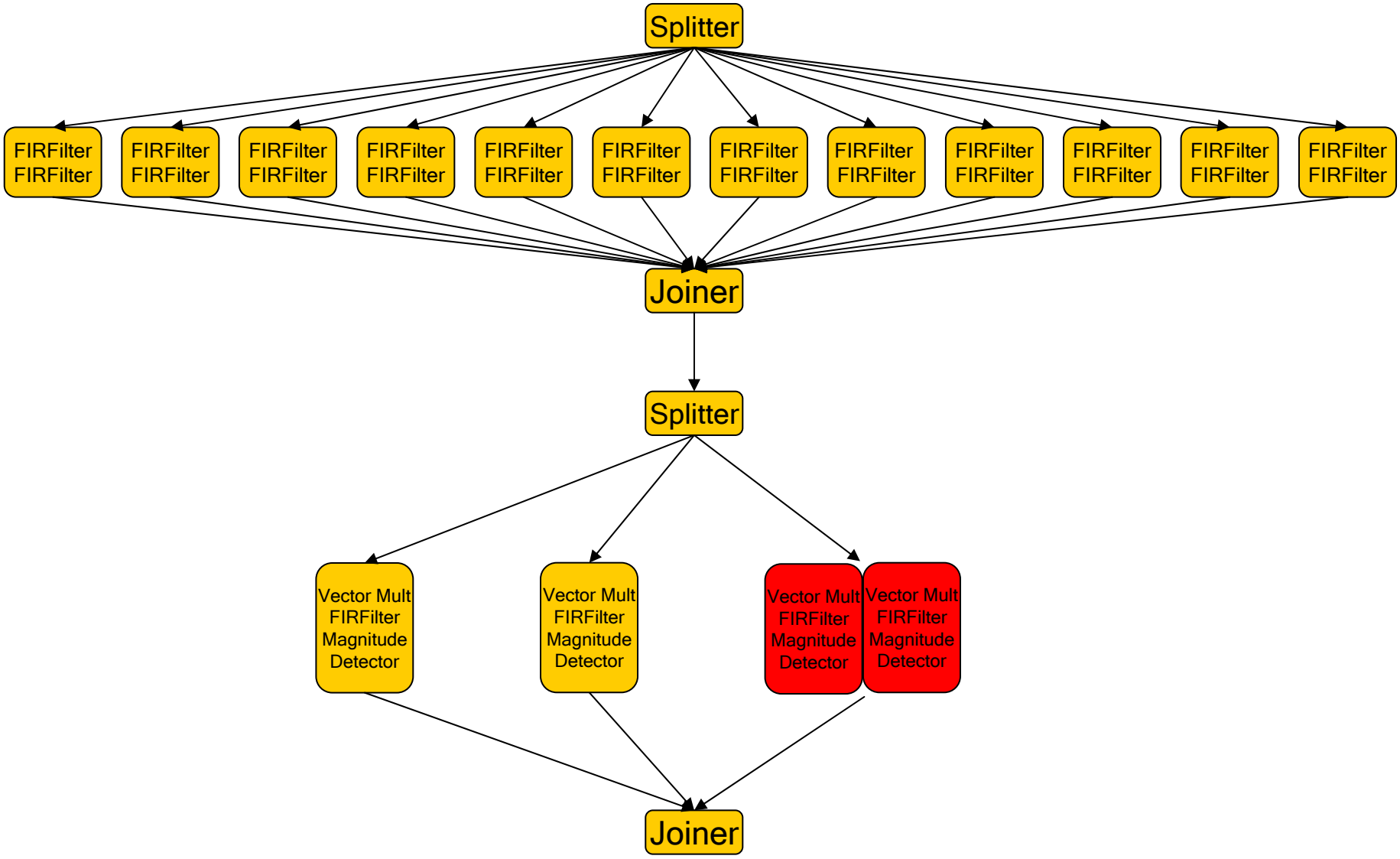
Example: Radar App.



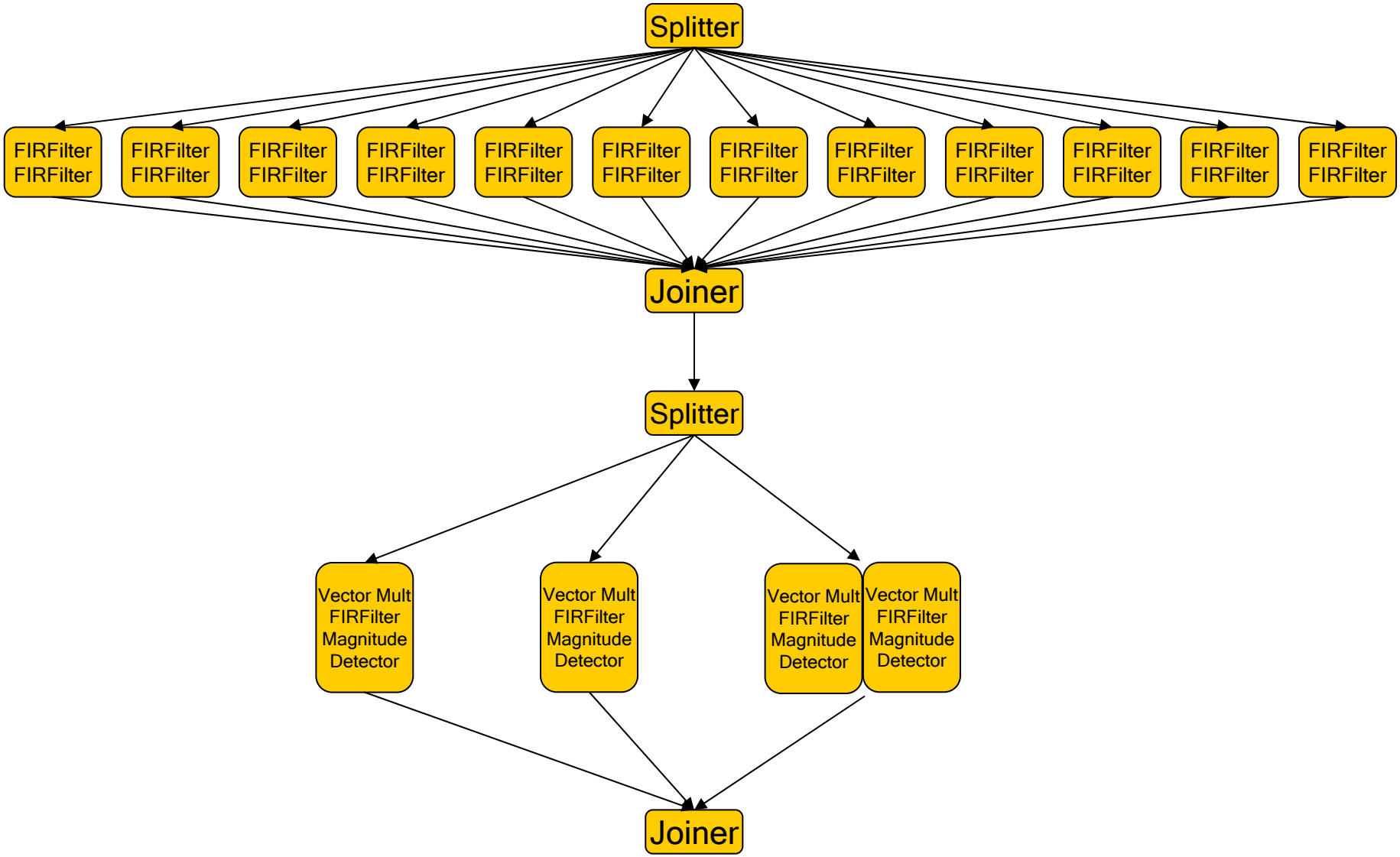
Example: Radar App.



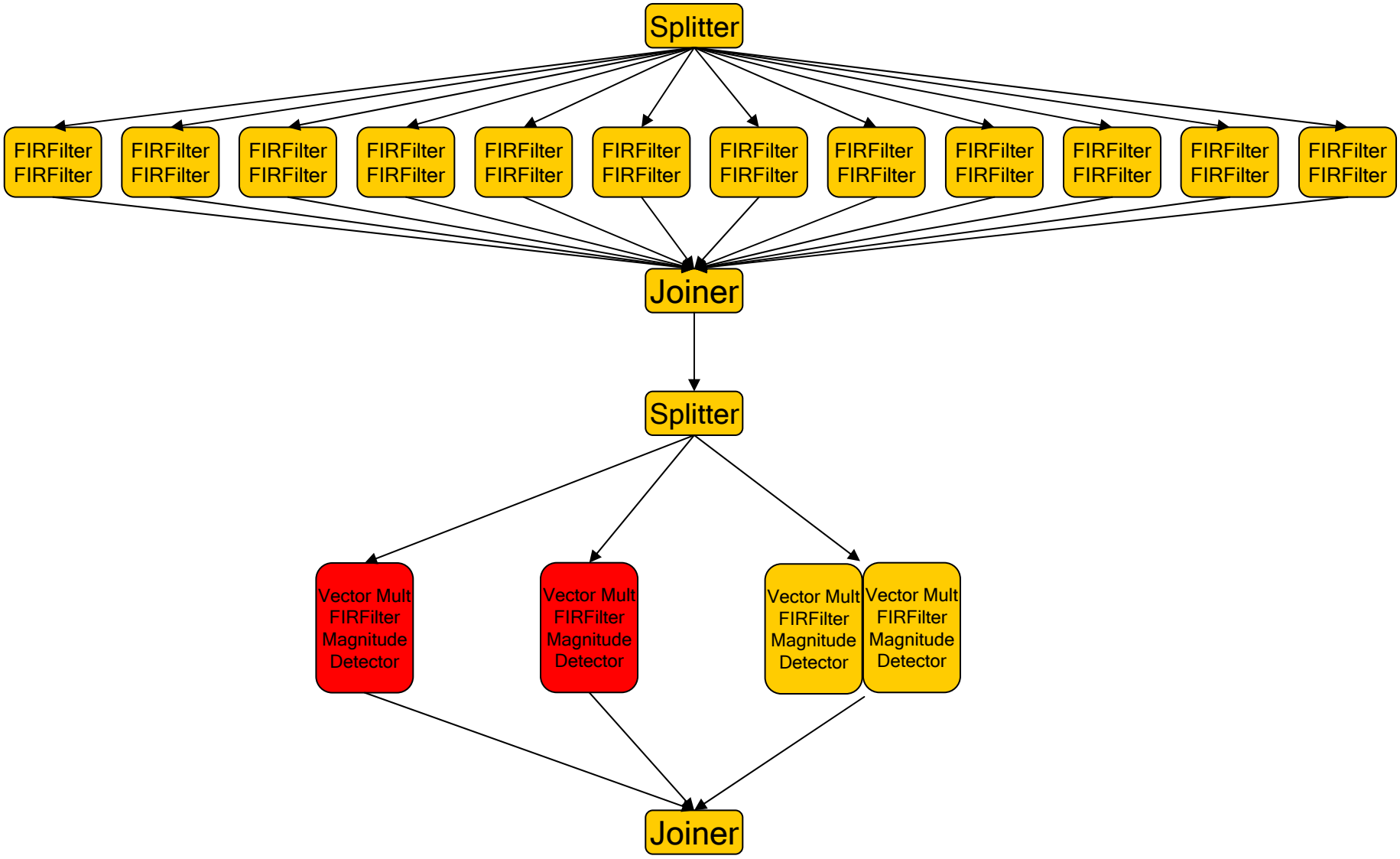
Example: Radar App.



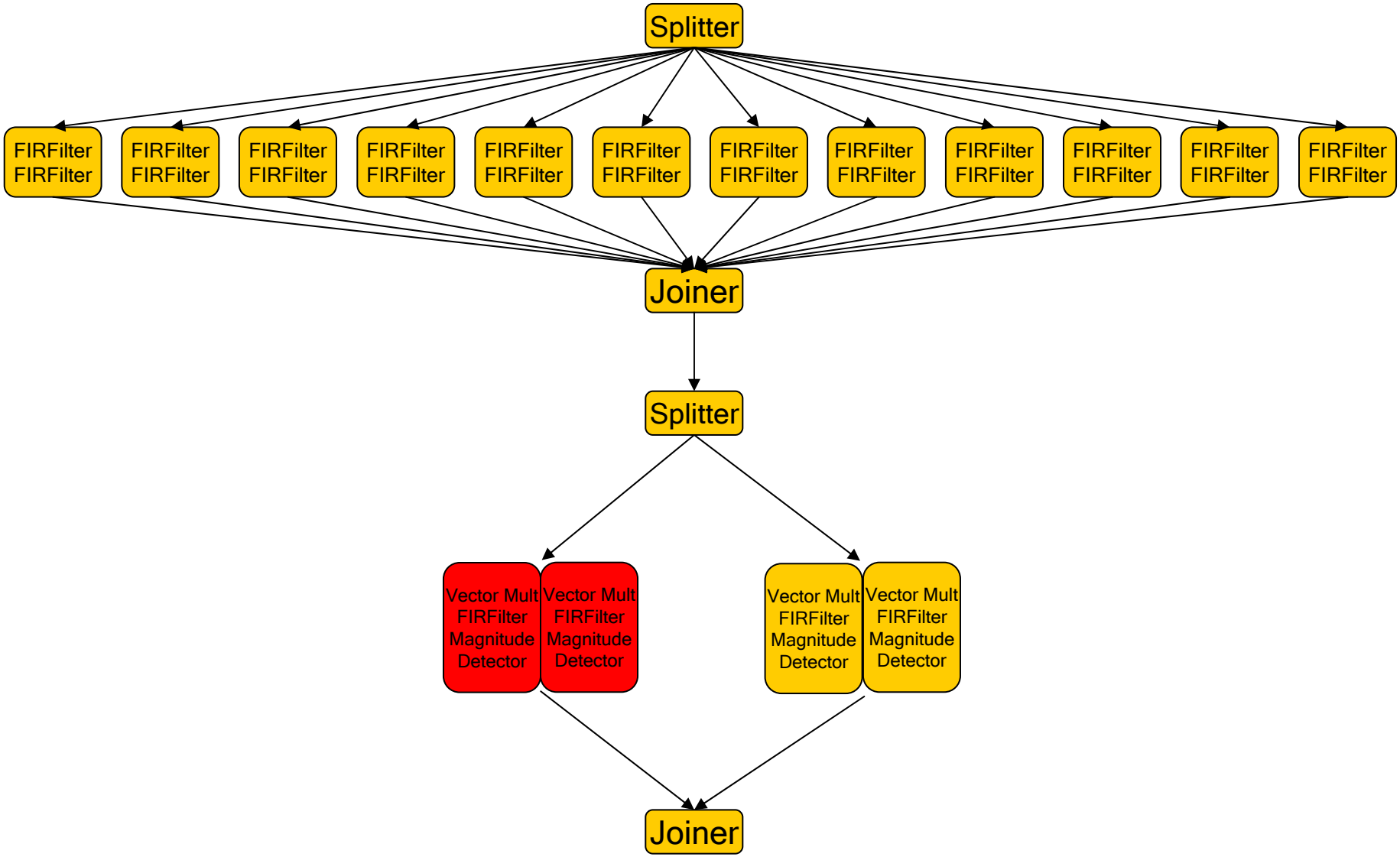
Example: Radar App.



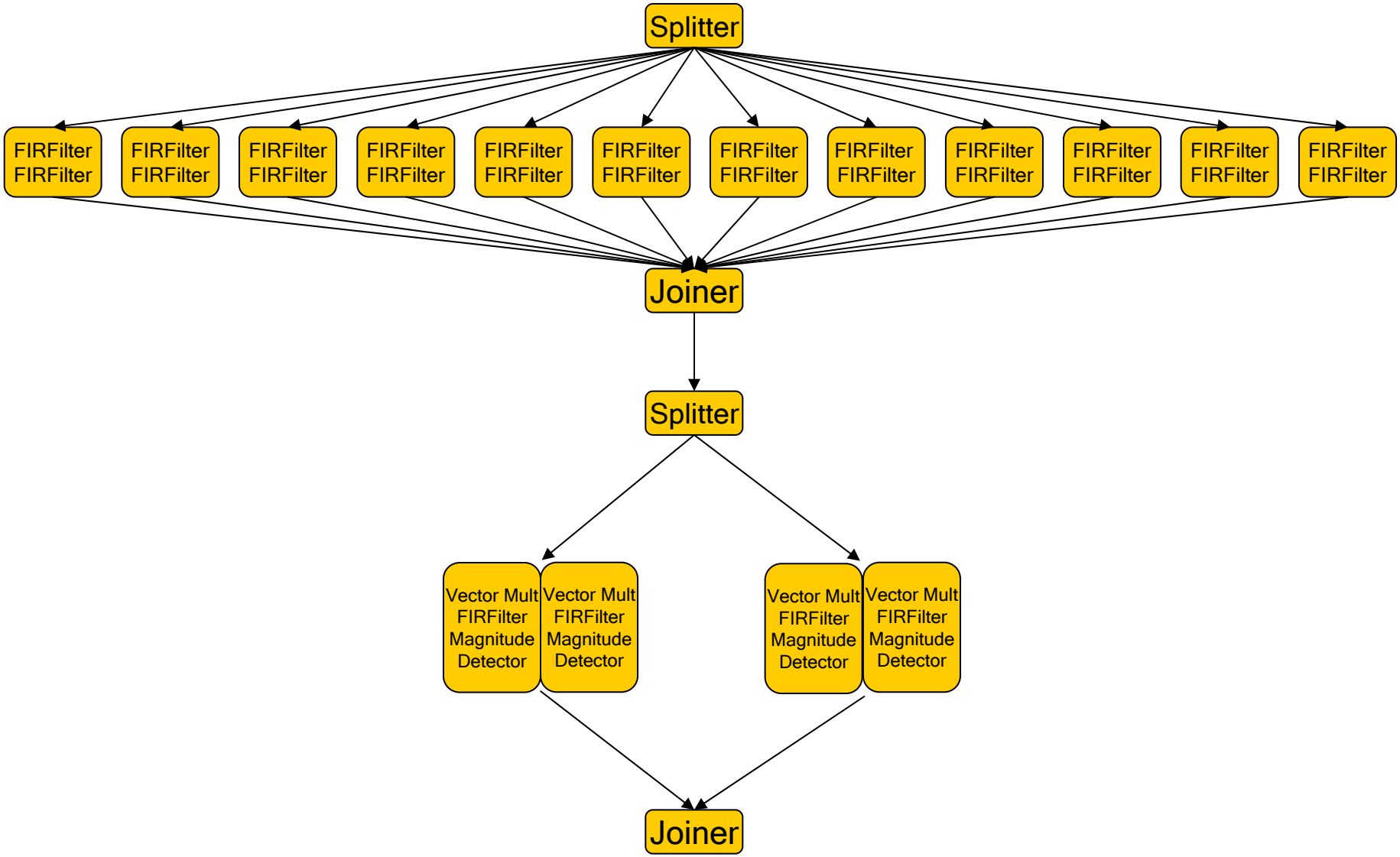
Example: Radar App.



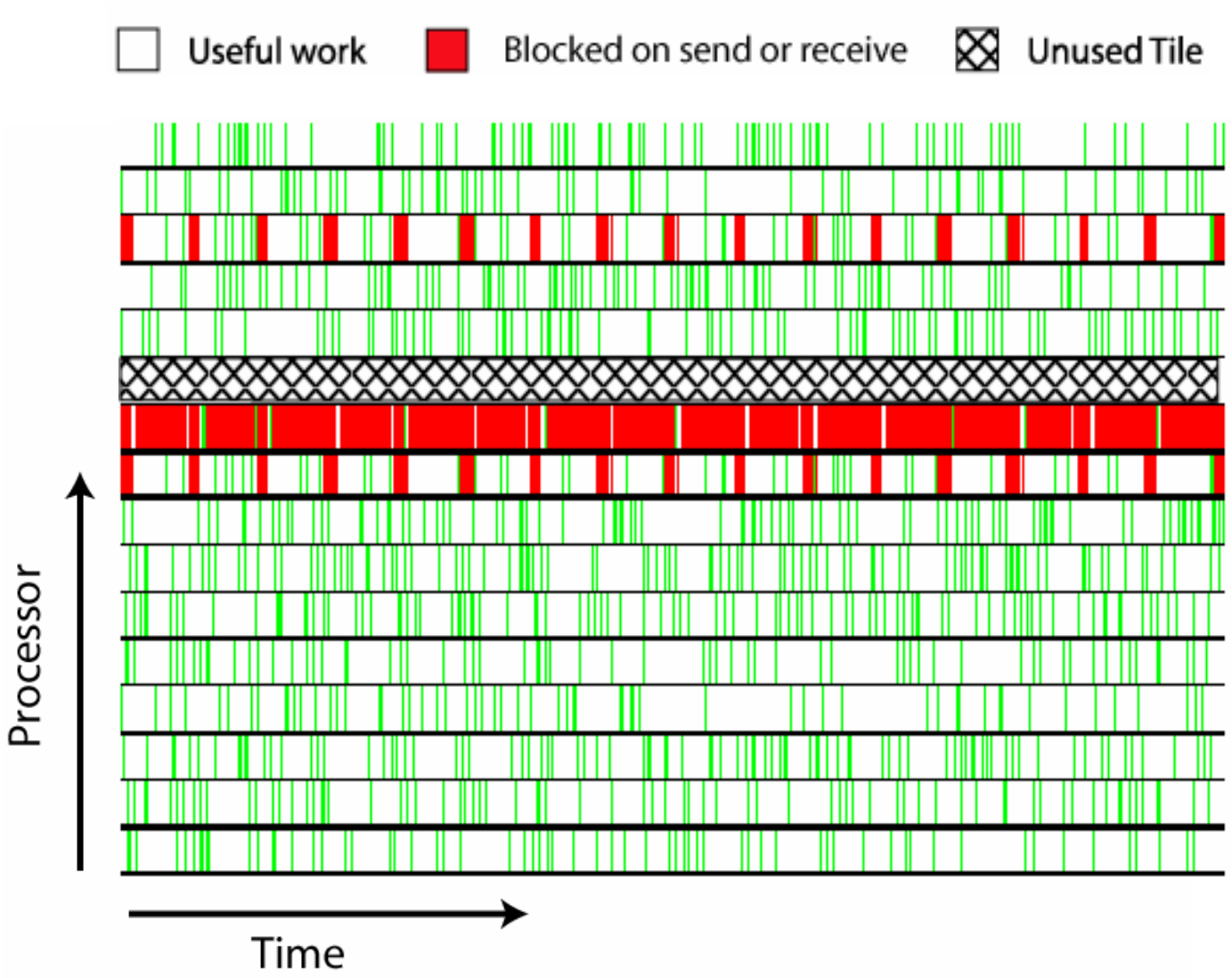
Example: Radar App.

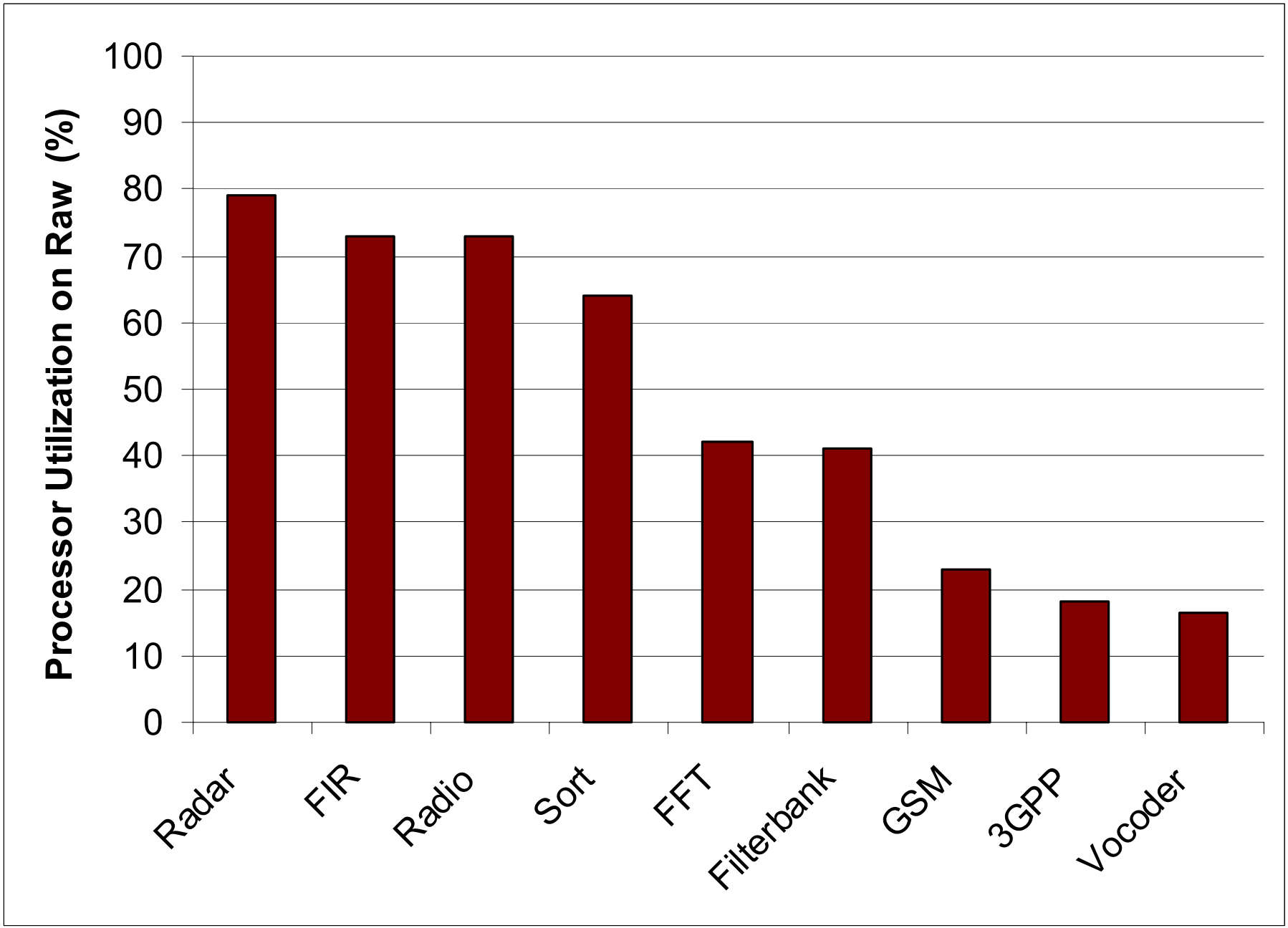


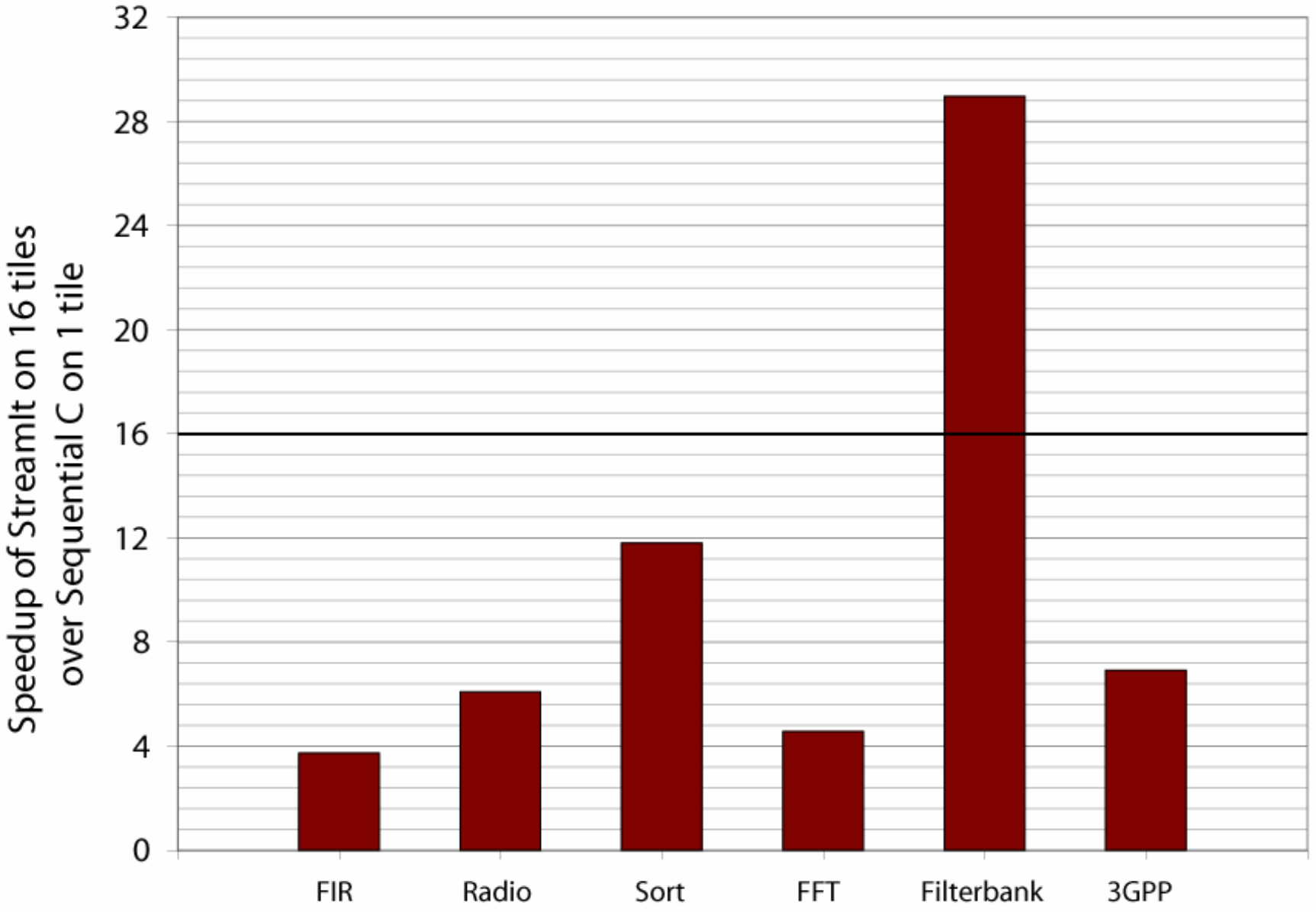
Example: Radar App. (Balanced)



Example: Radar App. (Balanced)




Processor Utilization on Raw (%)
100
90
80
70
60
50
40
30
20
10
0
Radar
FIR
Radio
Sort
FFT
Filterbank
GSM
3GPP
Vocoder


Speedup of StreamIt on 16 tiles over Sequential C on 1 tile
32
28
24
20
16
12
8
4
0
FIR
Radio
Sort
FFT
Filterbank
3GPP

Outline

- Design of Streamlt
  - Structured Streams
  - Messaging
  - Morphing
- Results
- Conclusions

Conclusions

- Compiler-conscious language design can improve both programmability and performance
  - Structure enables local, hierachical analyses
  - Messaging simplifies code, exposes parallelism
  - Morphing allows optimization across phases

- Goal: Stream programming at high level of abstraction without sacrificing performance

For More Information

StreamIt Homepage

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