A Stream Compiler for Communication-Exposed Architectures

Michael Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali Meli, Andrew Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, Saman Amarasinghe

Laboratory for Computer Science
Massachusetts Institute of Technology

The Streaming Domain

- Widely applicable and increasingly prevalent
  - Embedded systems
    - Cell phones, handheld computers, DSP's
  - Desktop applications
    - Streaming media
    - Software radio
    - Real-time encryption
    - Graphics packages
  - High-performance servers
    - Software routers (Example: Click)
    - Cell phone base stations
    - HDTV editing consoles
- Based on audio, video, or data stream
  - Predominant data types in the current data explosion

Properties of Stream Programs

- A large (possibly infinite) amount of data
  - Limited lifetime of each data item
  - Little processing of each data item
- Computation: apply multiple filters to data
  - Each filter takes an input stream, does some processing, and produces an output stream
  - Filters are independent and self-contained
- A regular, static computation pattern
  - Filter graph is relatively constant
  - A lot of opportunities for compiler optimizations

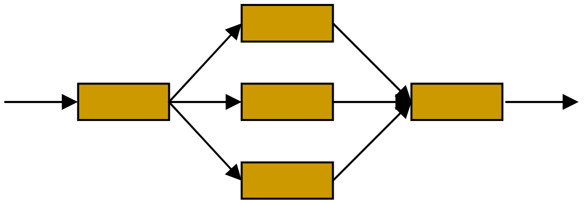
StreamIt: A spatially-aware Language & Compiler

- A language for streaming applications
  - Provides high-level stream abstraction
- Breaks the Von Neumann language barrier
  - Each filter has its own control-flow
  - Each filter has its own address space
  - No global time
  - Explicit data movement between filters
  - Compiler is free to reorganize the computation
- Spatially-aware Compiler
  - Intermediate representation with stream constructs
  - Provides a host of stream analyses and optimizations

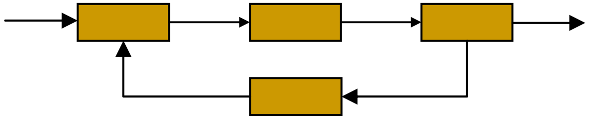
Structured Streams

- Hierarchical structures:
  - Pipeline

  - SplitJoin



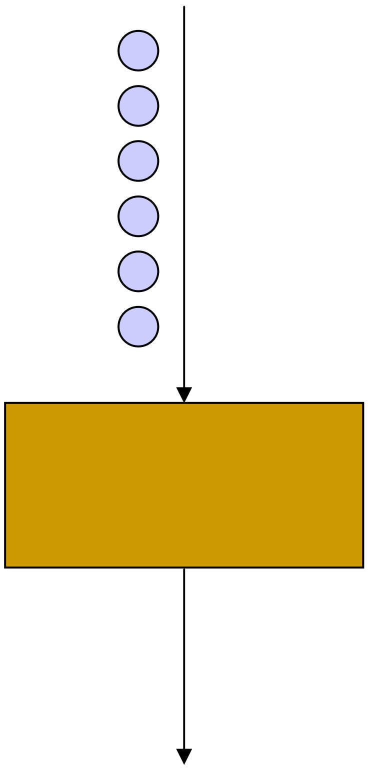
  - Feedback Loop



- Basic programmable unit: Filter

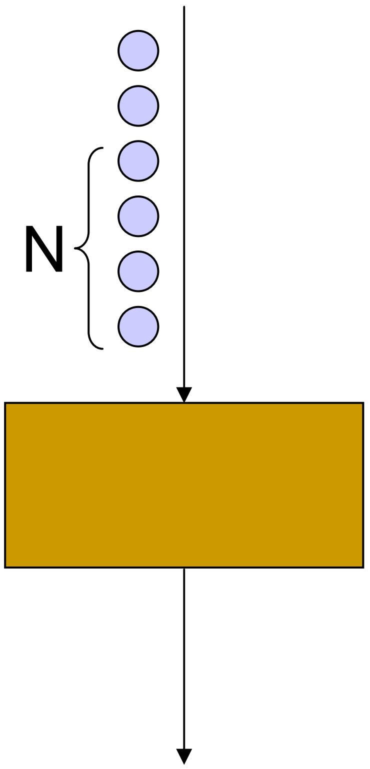
Filter Example: LowPassFilter

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

    init {
        for (int i=0; i<N; i++)
            weights[i] = calcWeights(i);
    }

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



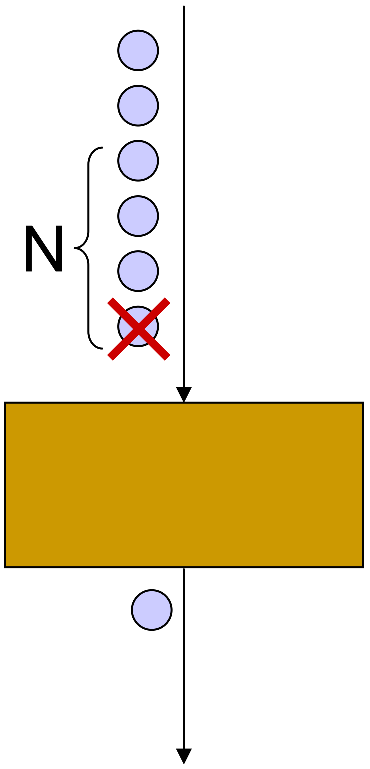
Filter Example: LowPassFilter

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

	init {
		for (int i=0; i<N; i++)
			weights[i] = calcWeights(i);
	}

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



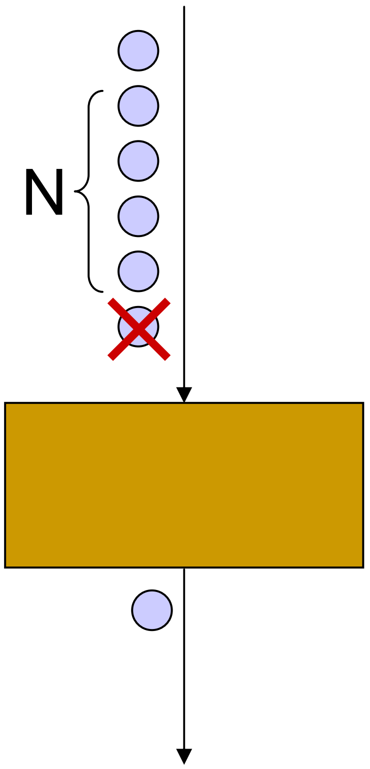
Filter Example: LowPassFilter

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

	init {
		for (int i=0; i<N; i++)
			weights[i] = calcWeights(i);
	}

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



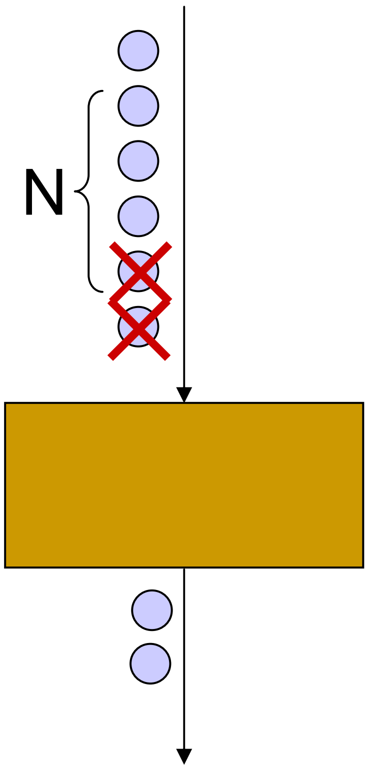
Filter Example: LowPassFilter

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

        init {
                for (int i=0; i<N; i++)
                        weights[i] = calcWeights(i);
        }

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



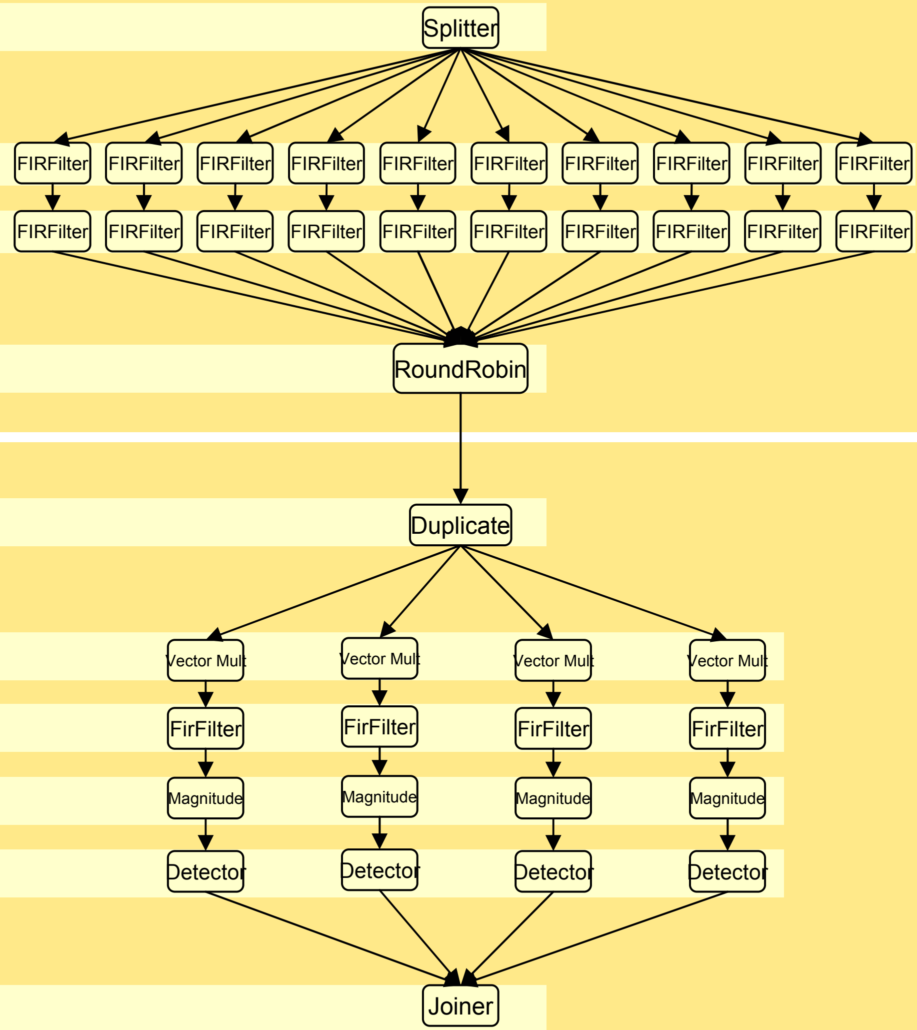
Filter Example: LowPassFilter

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

	init {
		for (int i=0; i<N; i++)
			weights[i] = calcWeights(i);
	}

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



Example: Radar Array Front End

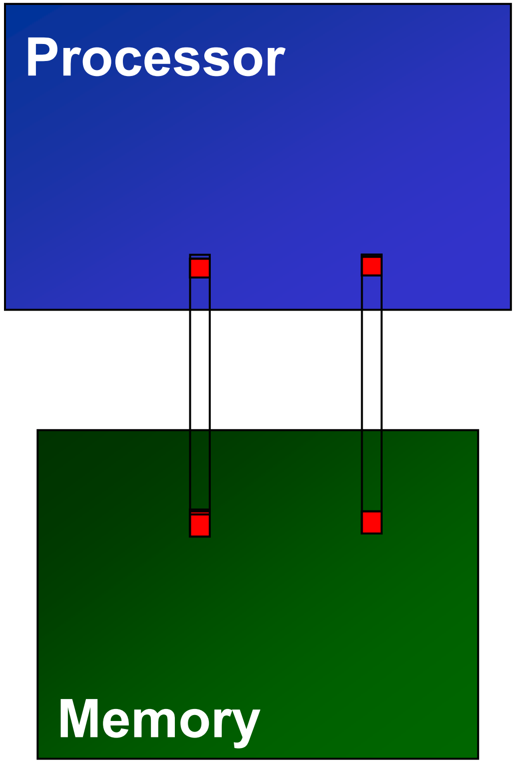
```
complex->void pipeline BeamFormer(int numChannels, int numBeams)
    {
        add splitjoin {
            split duplicate;
            for (int i=0; i<numChannels; i++) {
                add pipeline {
                    add FIR1(N1);
                    add FIR2(N2);
                };
            };
            join roundrobin;
        };
        add splitjoin {
            split duplicate;
            for (int i=0; i<numBeams; i++) {
                add pipeline {
                    add VectorMult();
                    add FIR3(N3);
                    add Magnitude();
                    add Detect();
                };
            };
            join roundrobin(0);
        };
}
```



How to execute a Stream Graph?
Method 1: Time Multiplexing

- Run one filter at a time

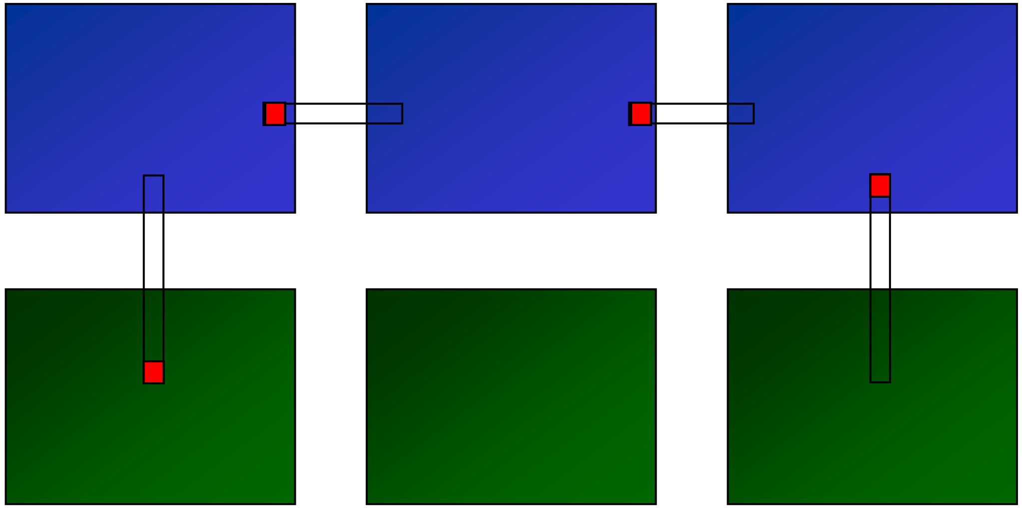
- Pros:
  - Scheduling is easy
  - Synchronization from Memory
- Cons:
  - If a filter run is too short
    - Filter load overhead is high
  - If a filter run is too long
    - Data spills down the cache hierarchy
    - Long latency
  - Lots of memory traffic
    - Bad cache effects
  - Does not scale with spatially-aware architectures



How to execute a Stream Graph? Method 2: Space Multiplexing

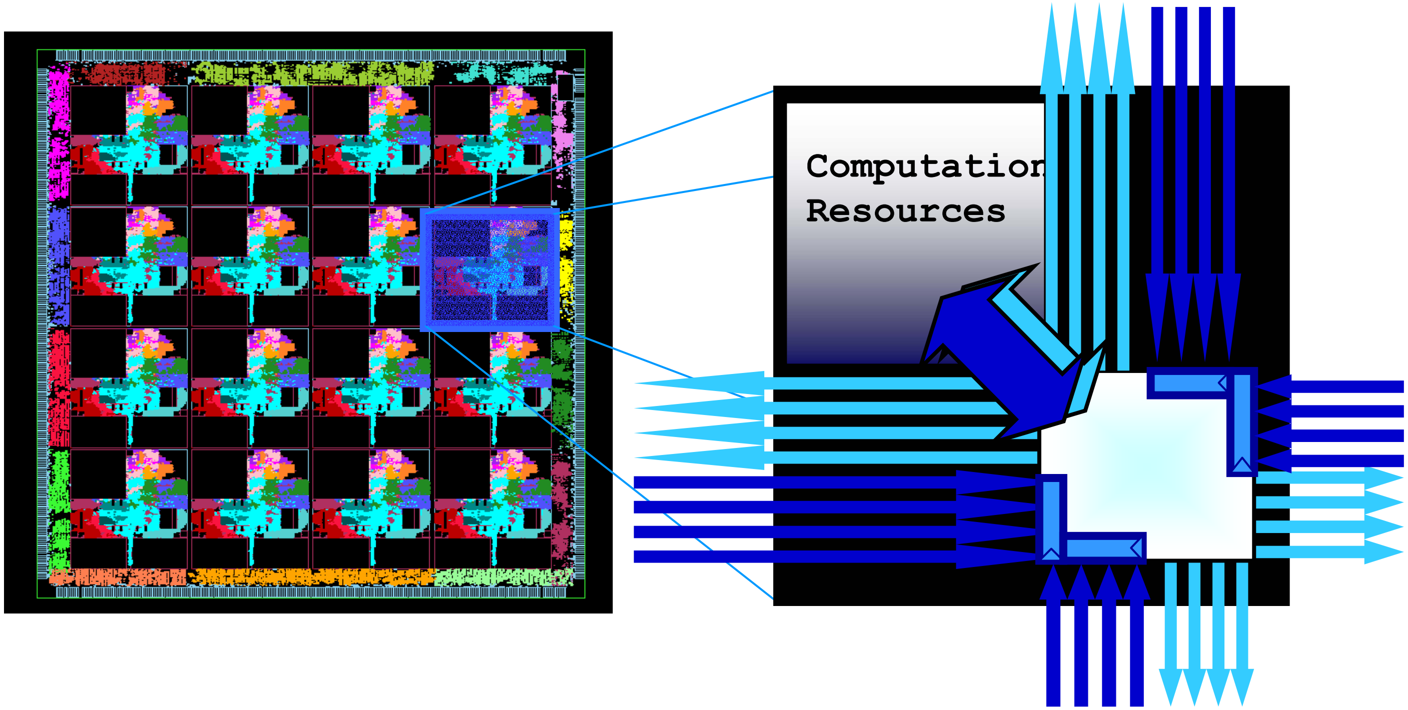
- Map filter per tile and run forever

- Pros:
  - No filter swapping overhead
  - Exploits spatially-aware architectures
    - Scales well
  - Reduced memory traffic
  - Localized communication
  - Tighter latencies
  - Smaller live data set



- Cons:
  - Load balancing is critical
  - Not good for dynamic behavior
  - Requires # filters ≤ # processing elements

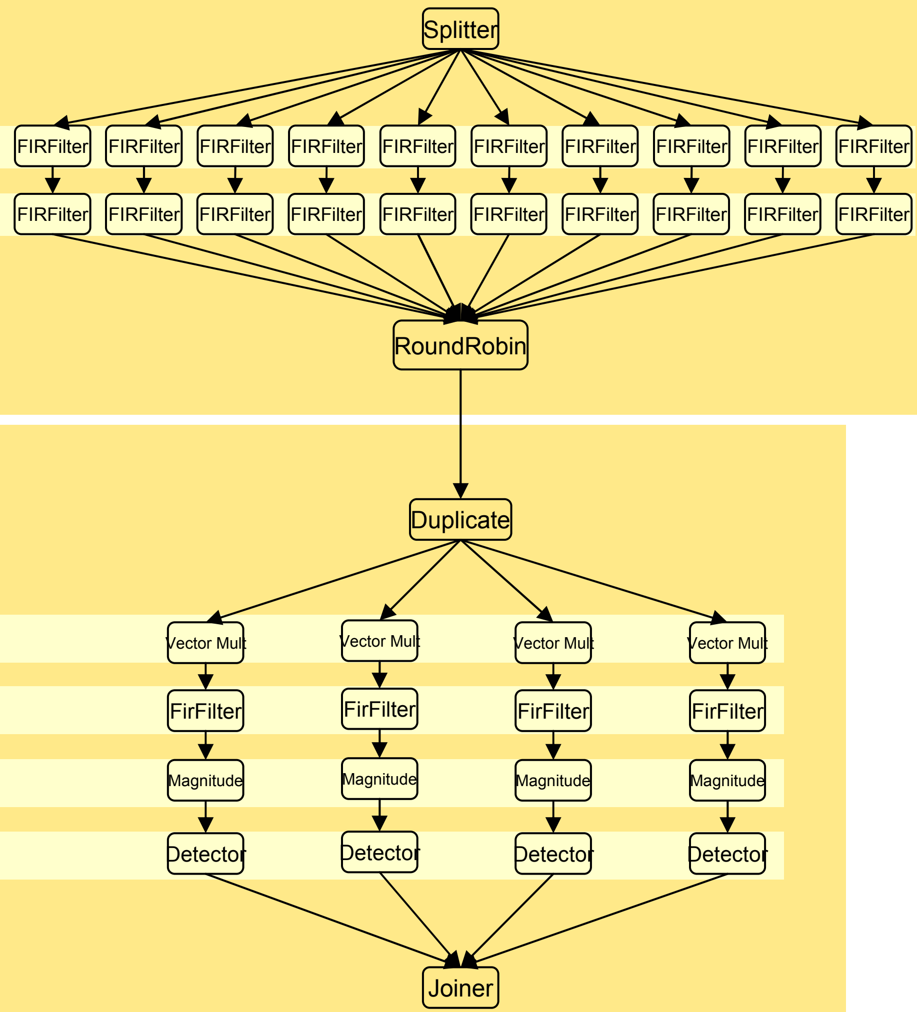
The MIT RAW Machine



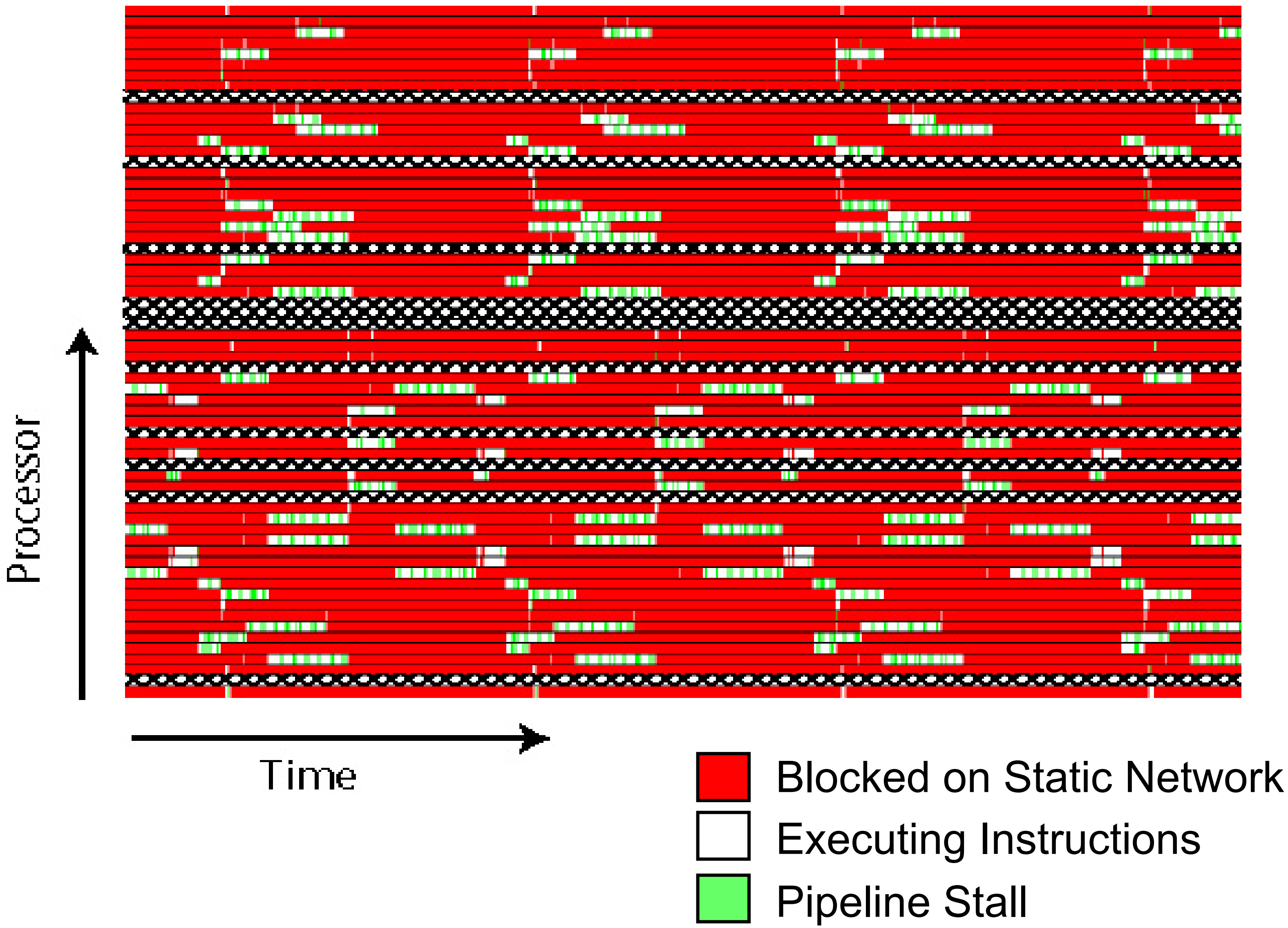
- A scalable computation fabric
  - 4 x 4 mesh of tiles, each tile is a simple microprocessor
- Ultra fast interconnect network
  - Exposes the wires to the compiler
  - Compiler orchestrate the communication

Example: Radar Array Front End

```
complex->void pipeline BeamFormer(int numChannels, int numBeams)
    {
      add splitjoin {
            split duplicate;
            for (int i=0; i<numChannels; i++) {
                add pipeline {
                    add FIR1(N1);

                    add FIR2(N2);
                };
            };
            join roundrobin;
      };
      add splitjoin {
            split duplicate;
            for (int i=0; i<numBeams; i++) {
                add pipeline {
                    add VectorMult();

                    add FIR3(N3);

                    add Magnitude();

                    add Detect();
                };
            };
            join roundrobin(0);
    };
}
```



Radar Array Front End on Raw



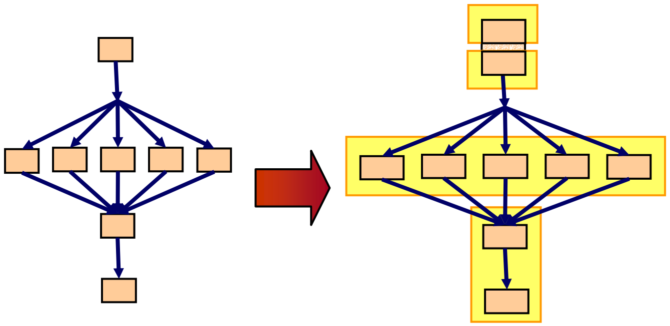
Bridging the Abstraction layers

- StreamIt language exposes the data movement
  - Graph structure is architecture independent
- Each architecture is different in granularity and topology
  - Communication is exposed to the compiler
- The compiler needs to efficiently bridge the abstraction
  - Map the computation and communication pattern of the program to the PE’s, memory and the communication substrate

Bridging the Abstraction layers

- StreamIt language exposes the data movement
  - Graph structure is architecture independent
- Each architecture is different in granularity and topology
  - Communication is exposed to the compiler
- The compiler needs to efficiently bridge the abstraction
  - Map the computation and communication pattern of the program to the PE's, memory and the communication substrate
- The StreamIt Compiler
  - Partitioning
  - Placement
  - Scheduling
  - Code generation

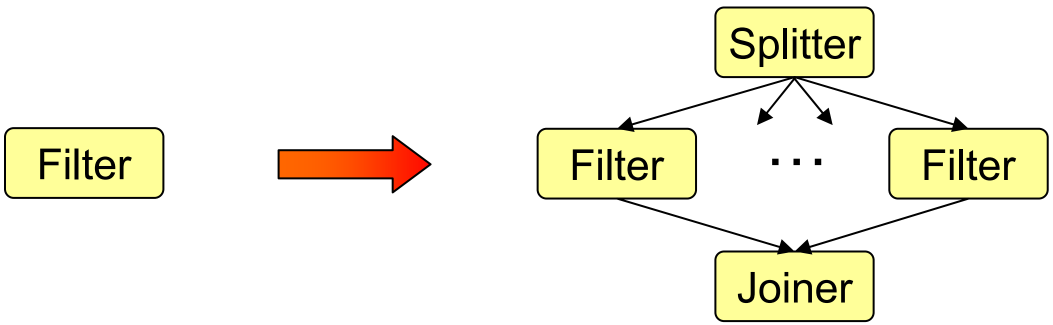
Partitioning: Choosing the Granularity



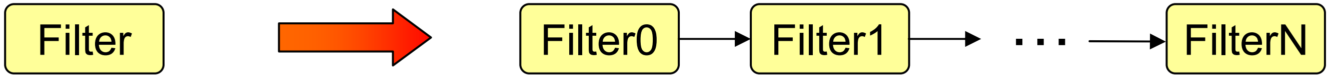
- Mapping filters to tiles
  - – # filters should equal (or a few less than) # of tiles
  - – Each filter should have similar amount of work
    - Throughput determined by the filter with most work
- Compiler Algorithm
  - – Two primary transformations
    - Filter fission
    - Filter fusion
  - – Uses a greedy heuristic

Partitioning - Fission

- Fission - splitting streams
  - Duplicate a filter, placing the duplicates in a SplitJoin to expose parallelism.

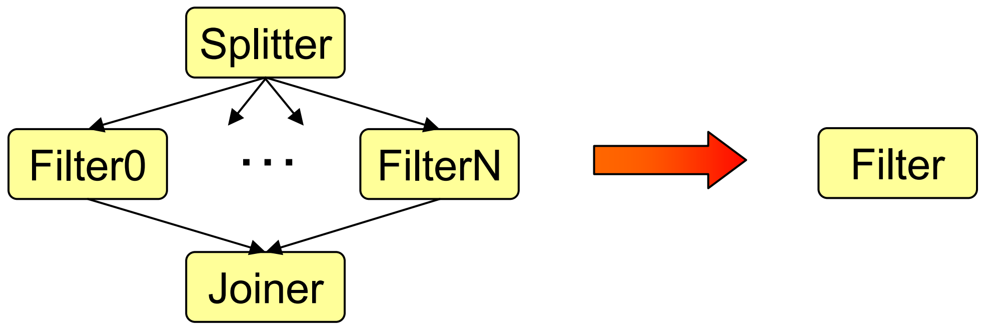


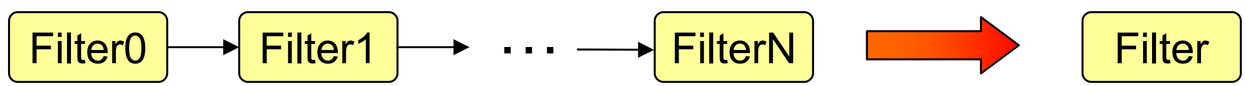
–Split a filter into a pipeline for load balancing



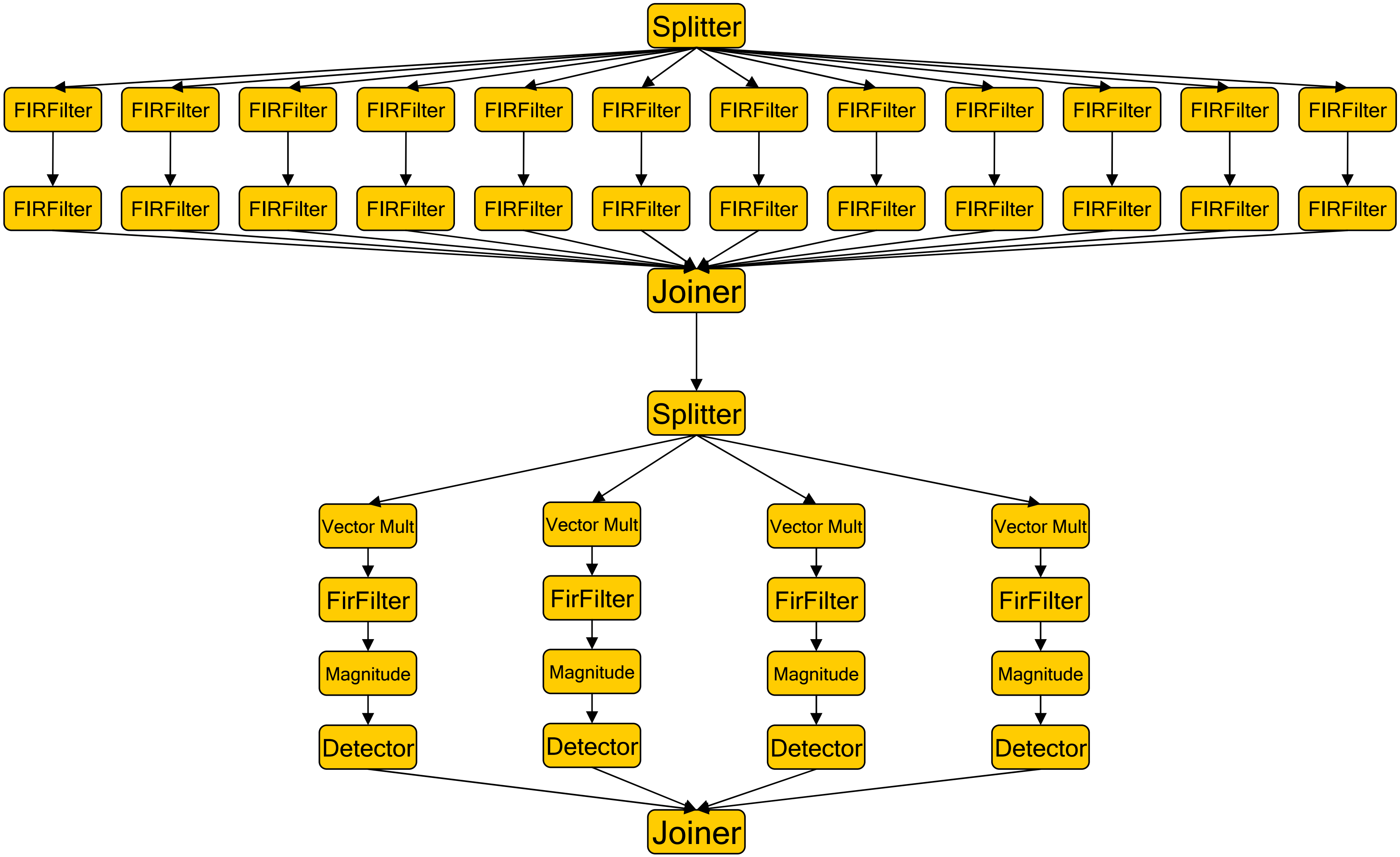
Partitioning - Fusion

- Fusion - merging streams
  - Merge filters into one filter for load balancing and synchronization removal

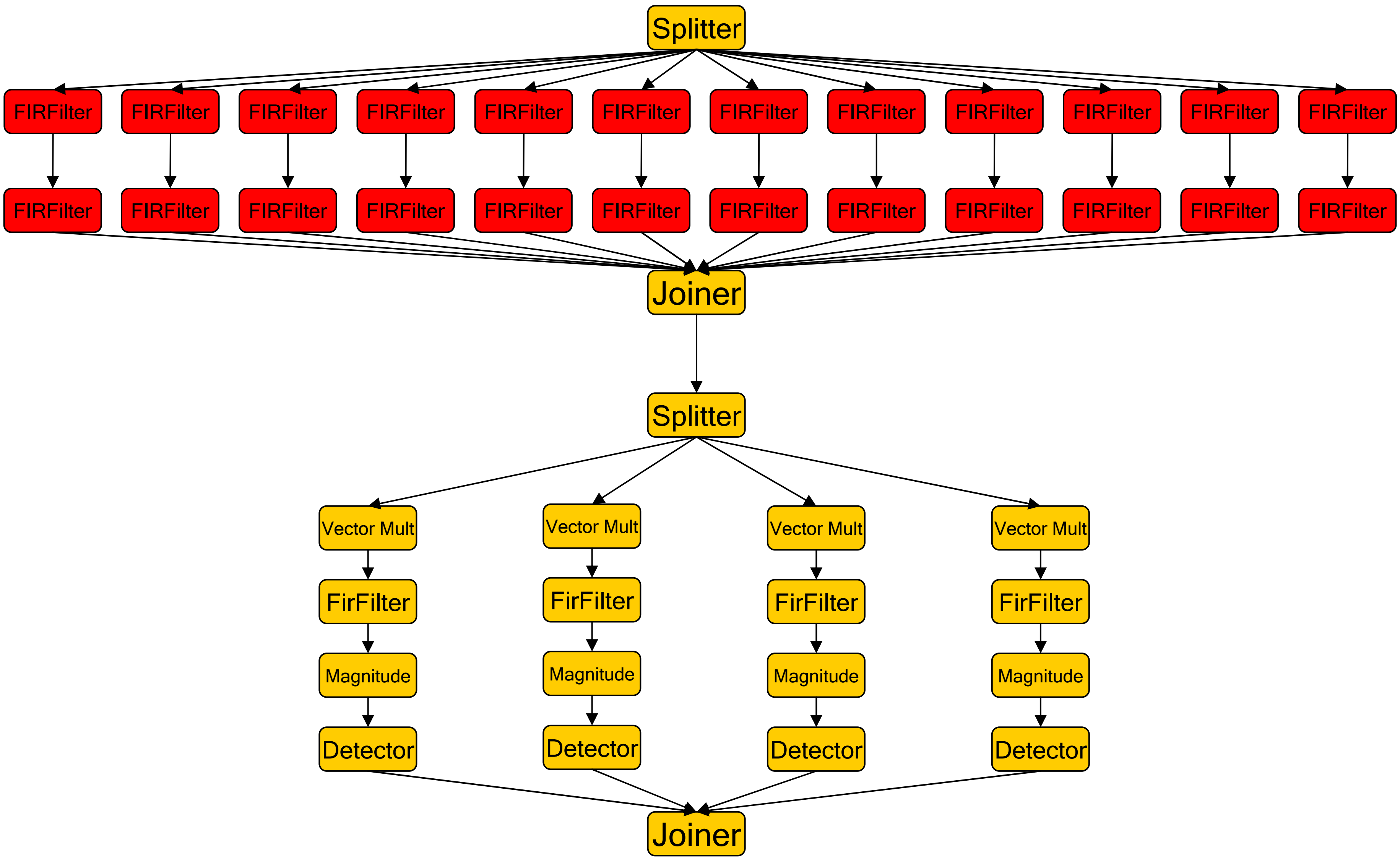




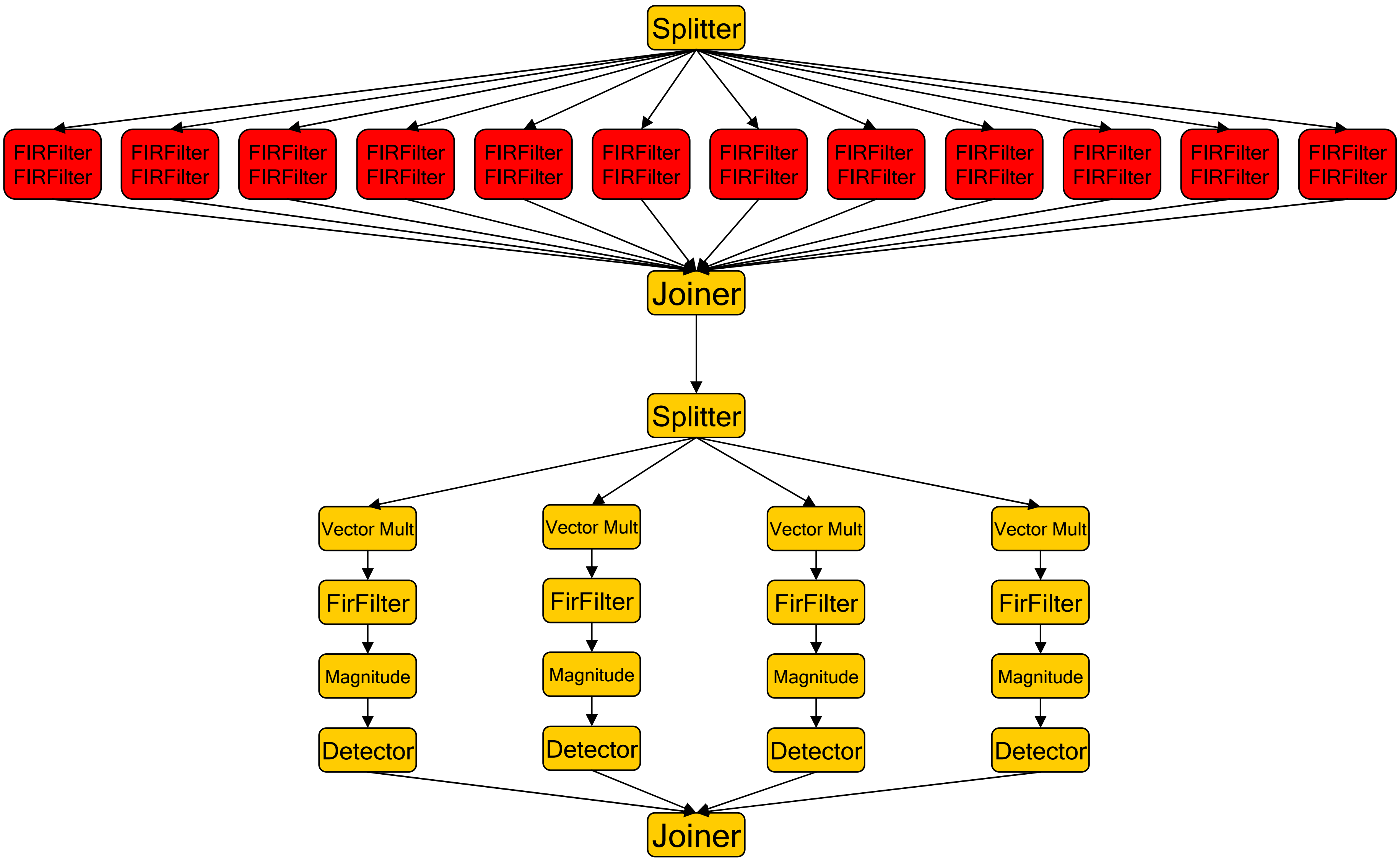
Example: Radar Array Front End (Original)



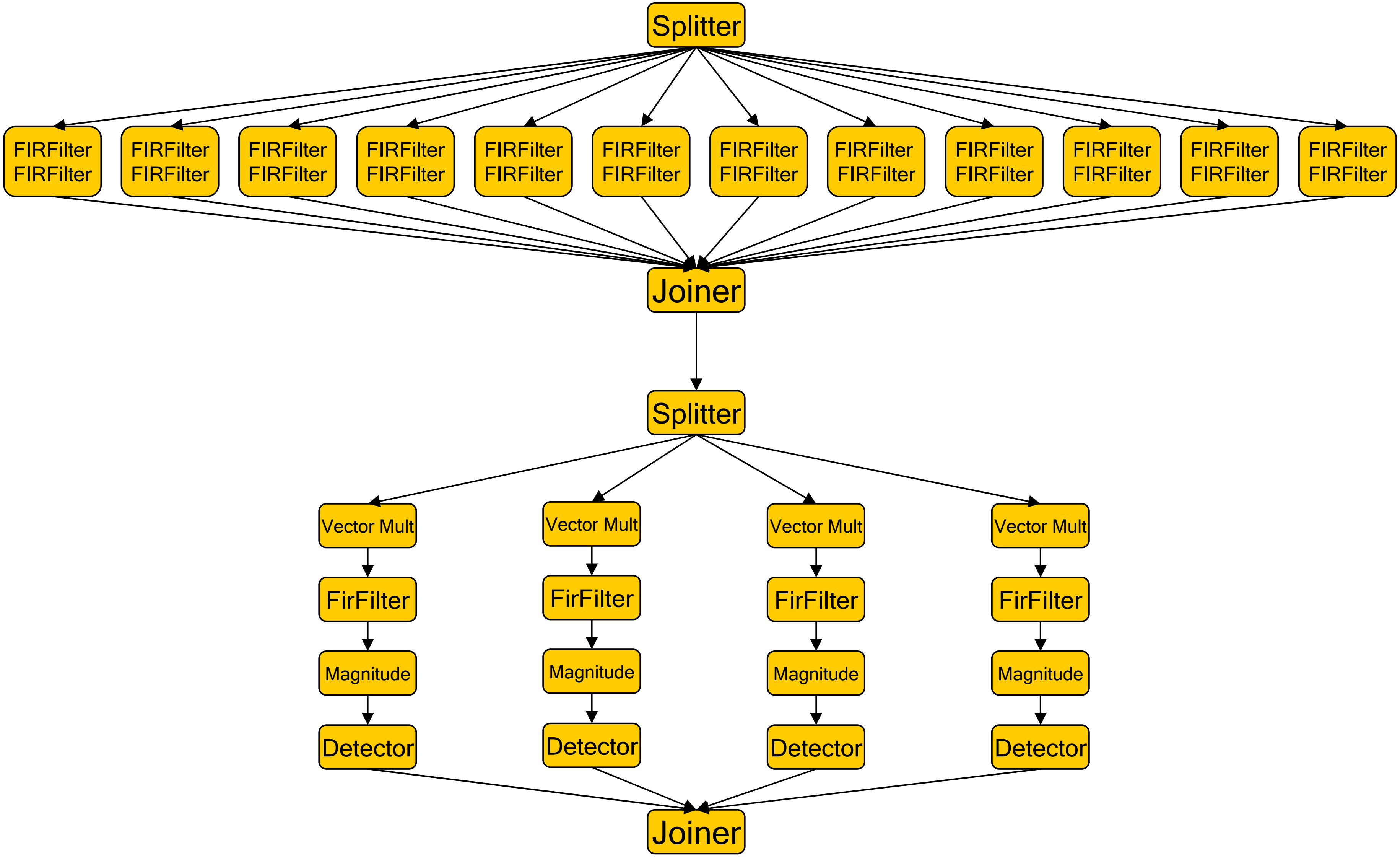
Example: Radar Array Front End



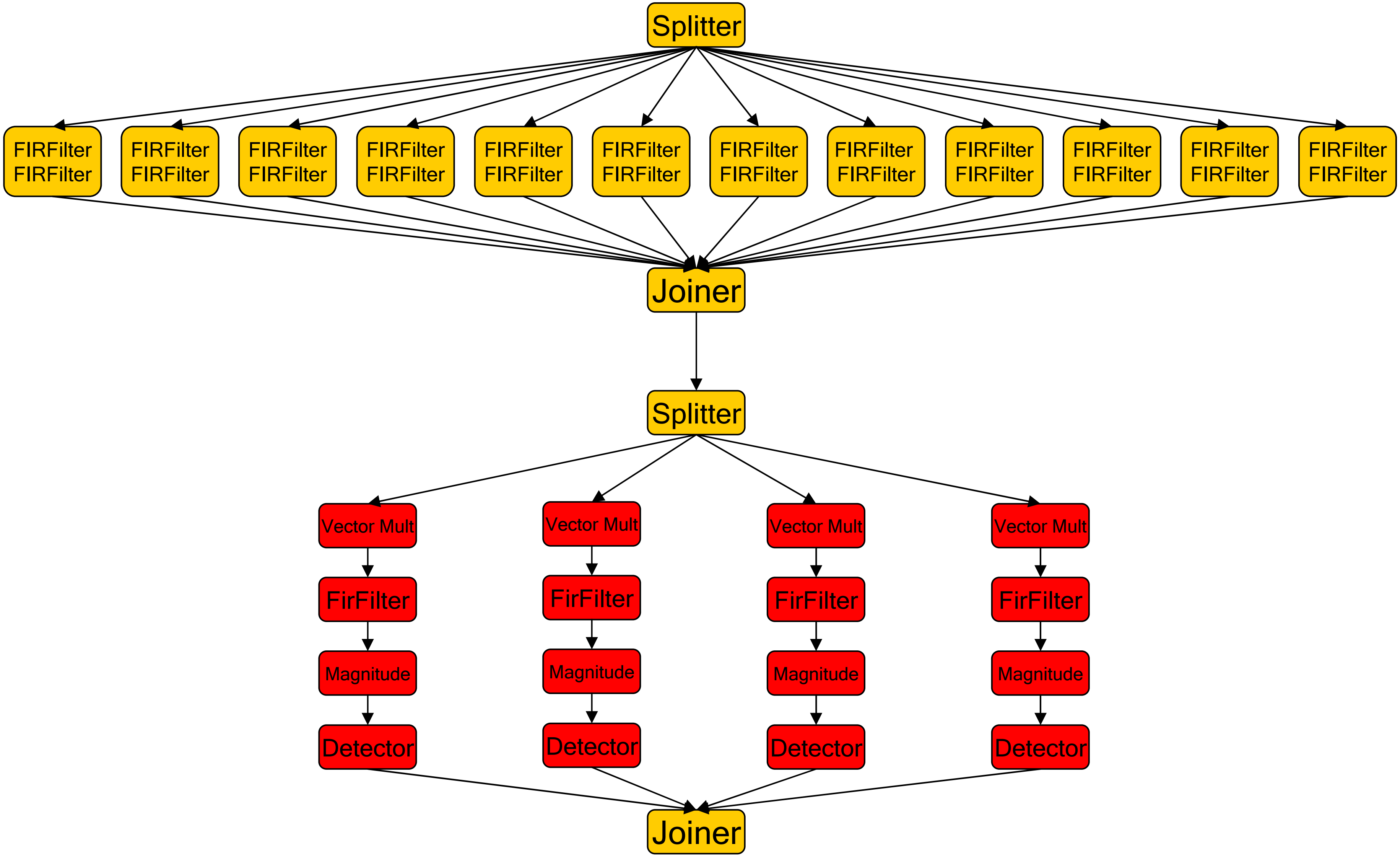
Example: Radar Array Front End



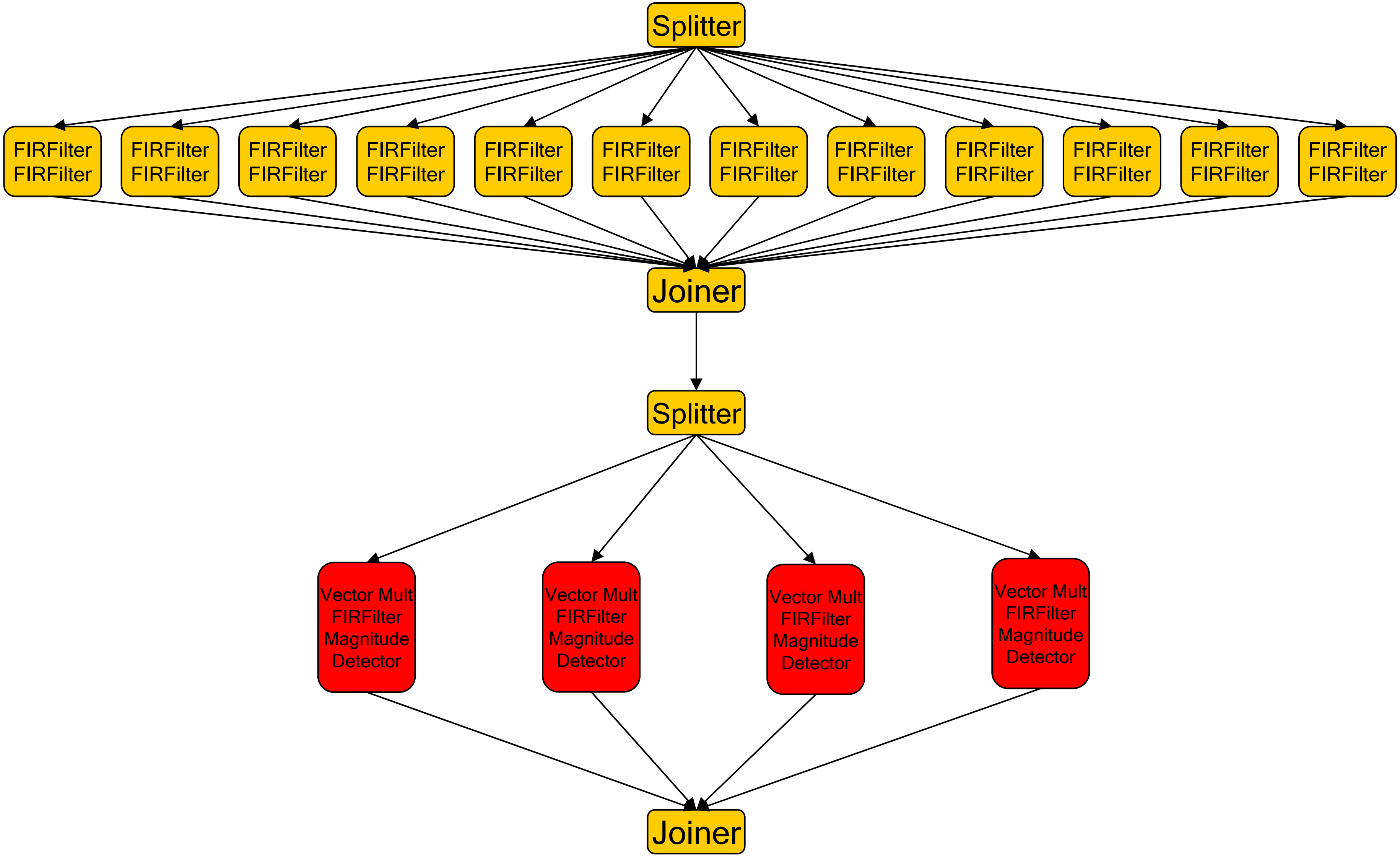
Example: Radar Array Front End



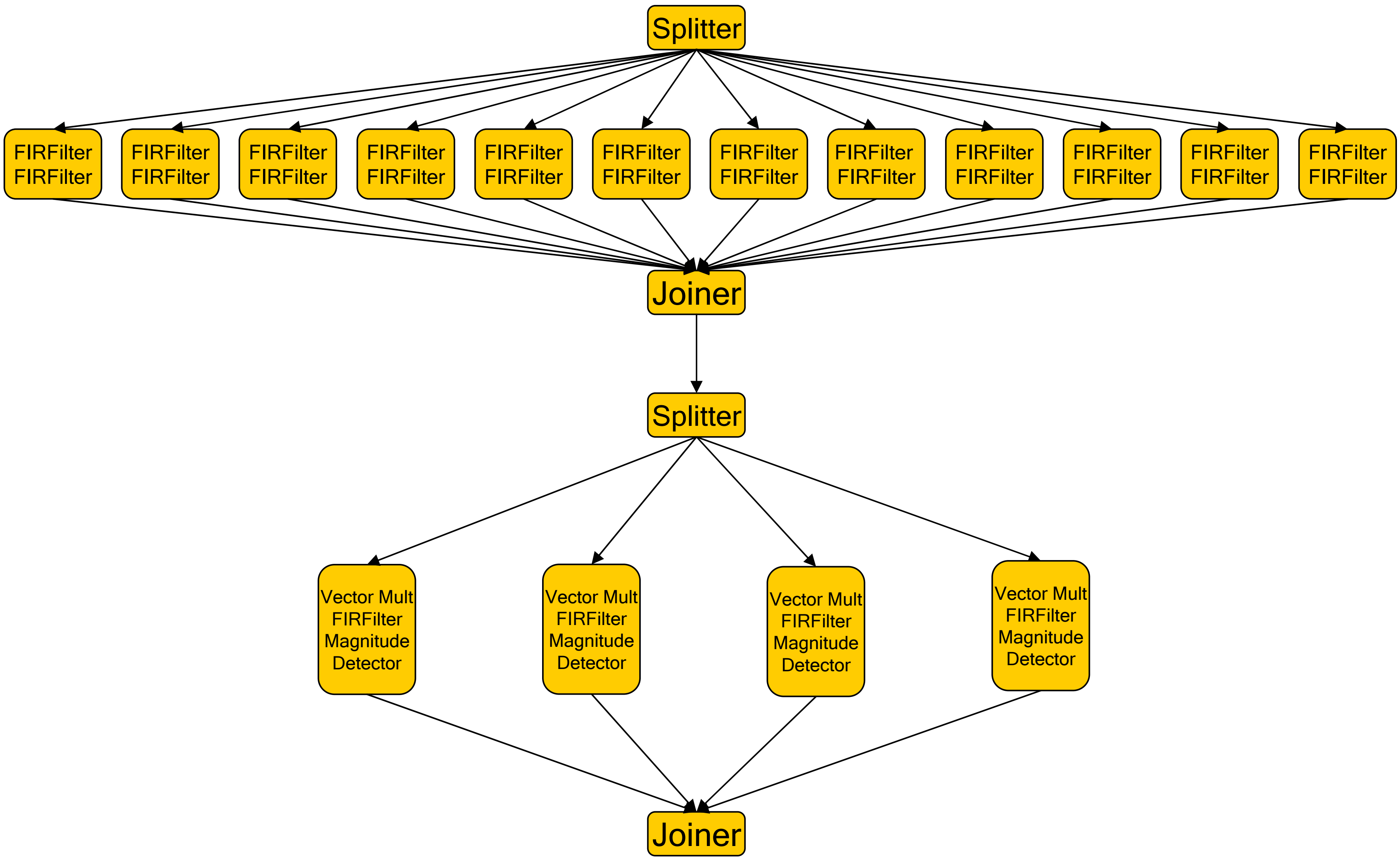
Example: Radar Array Front End



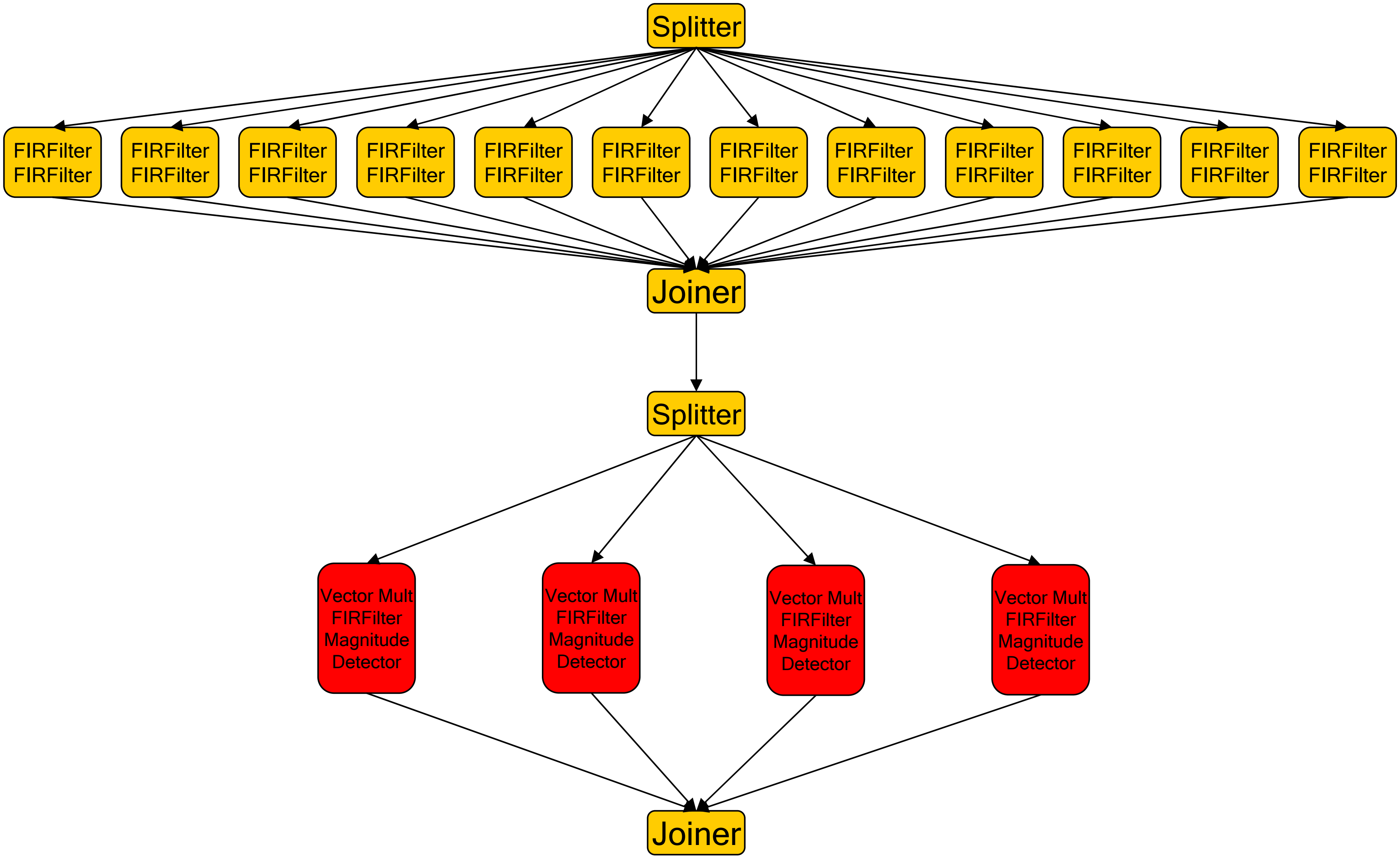
Example: Radar Array Front End



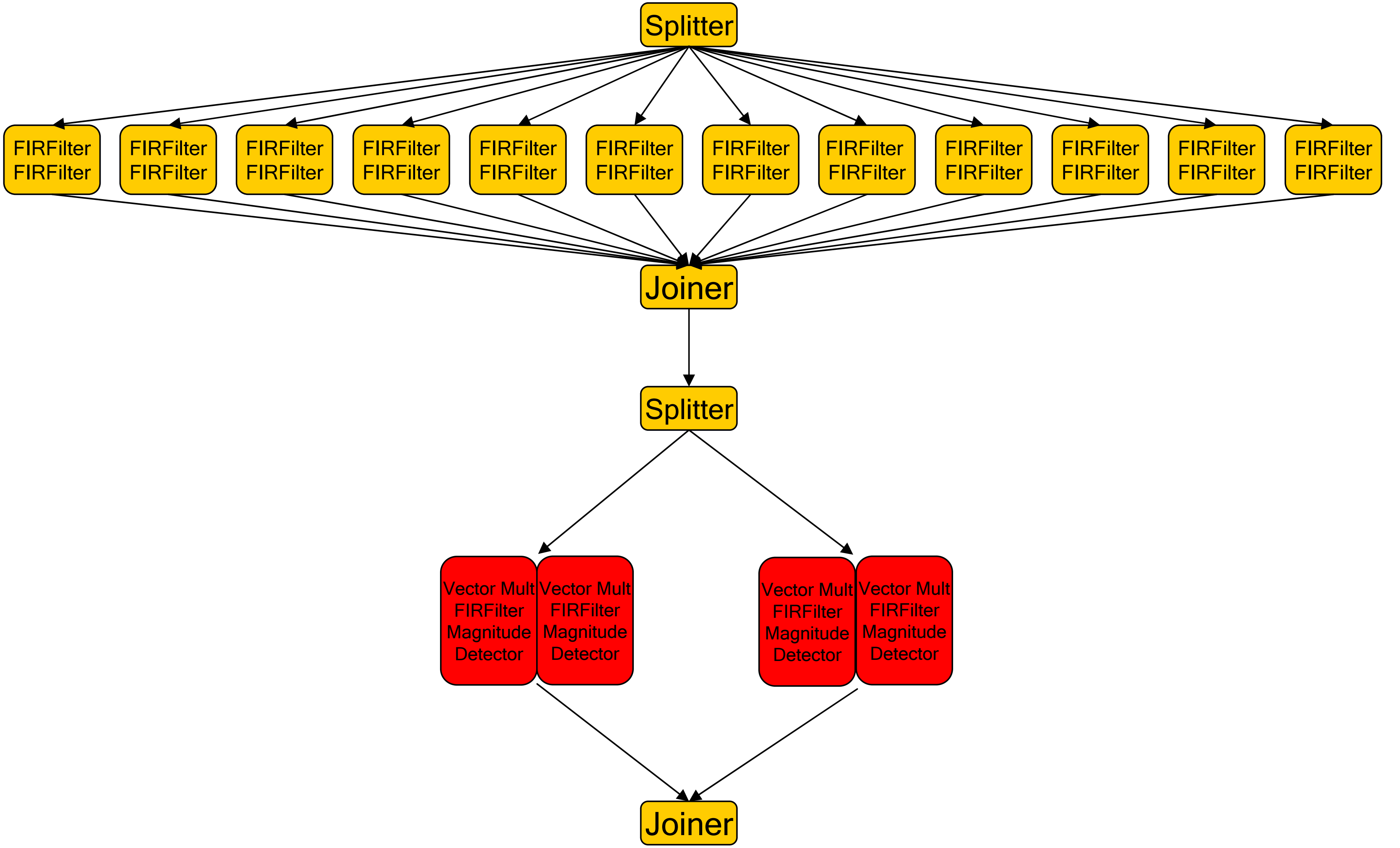
Example: Radar Array Front End



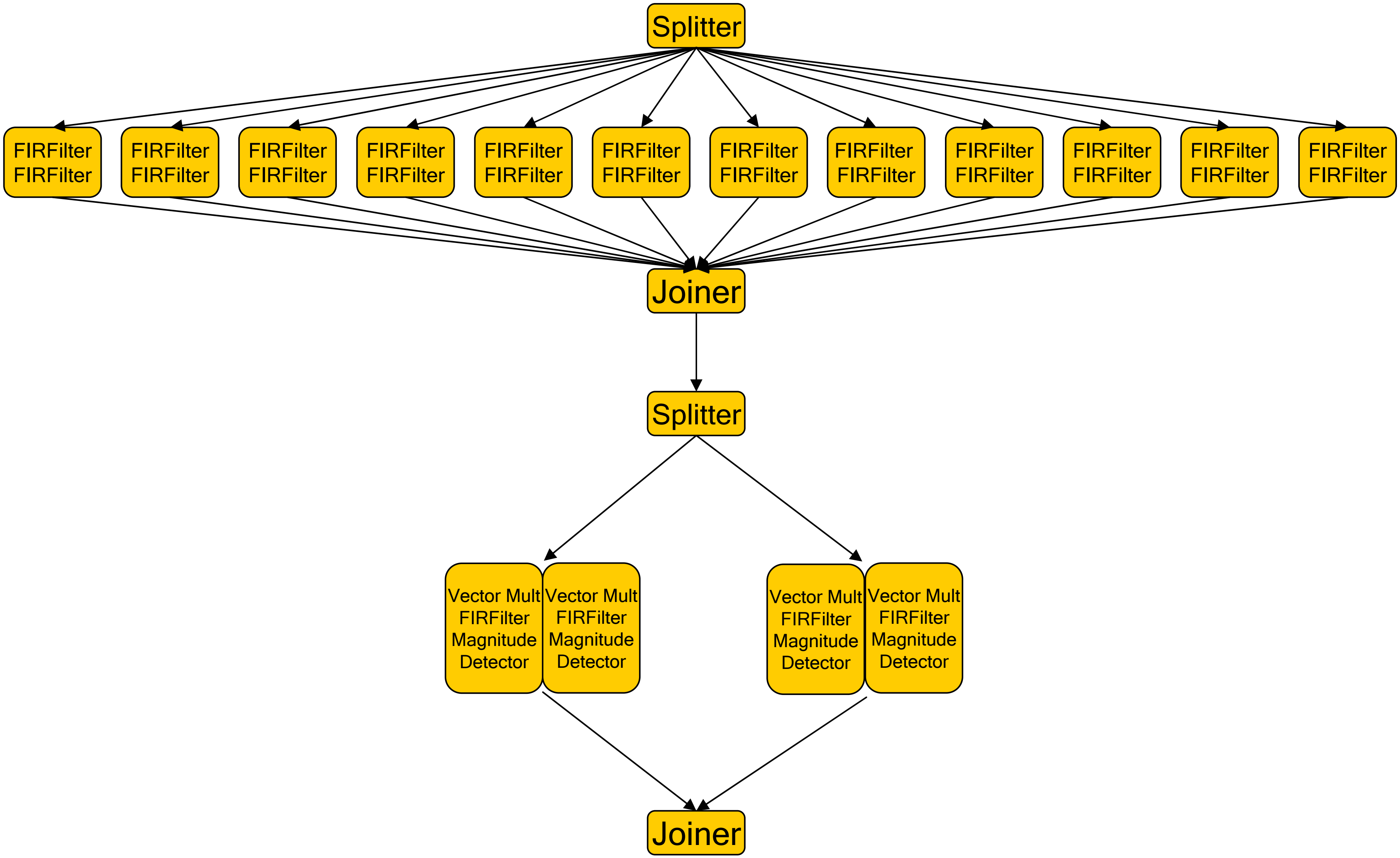
Example: Radar Array Front End



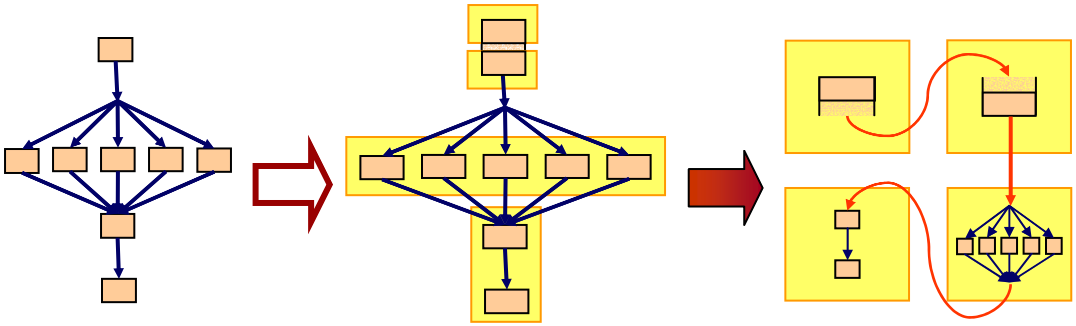
Example: Radar Array Front End



Example: Radar Array Front End (Balanced)

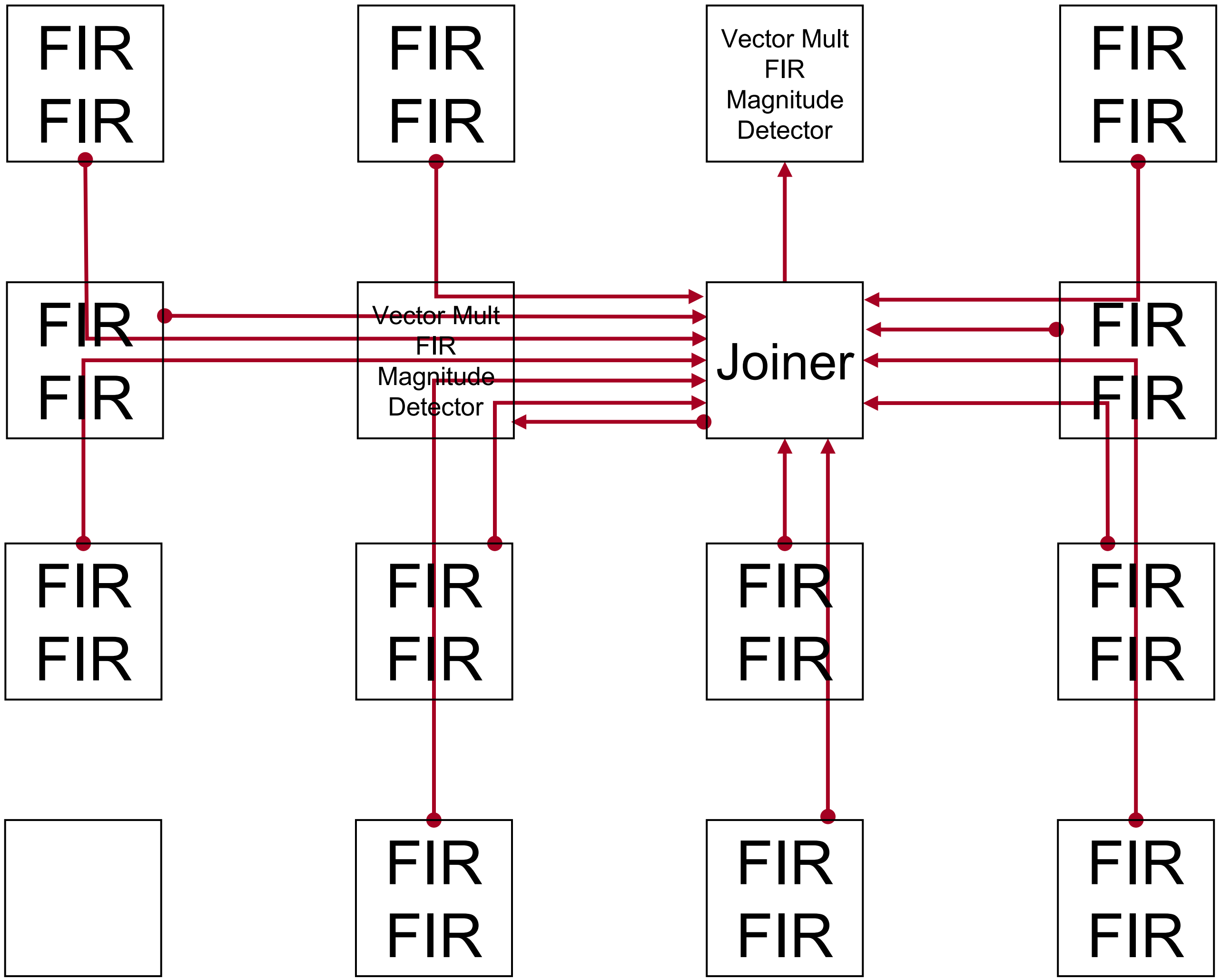


Placement: Minimizing Communication

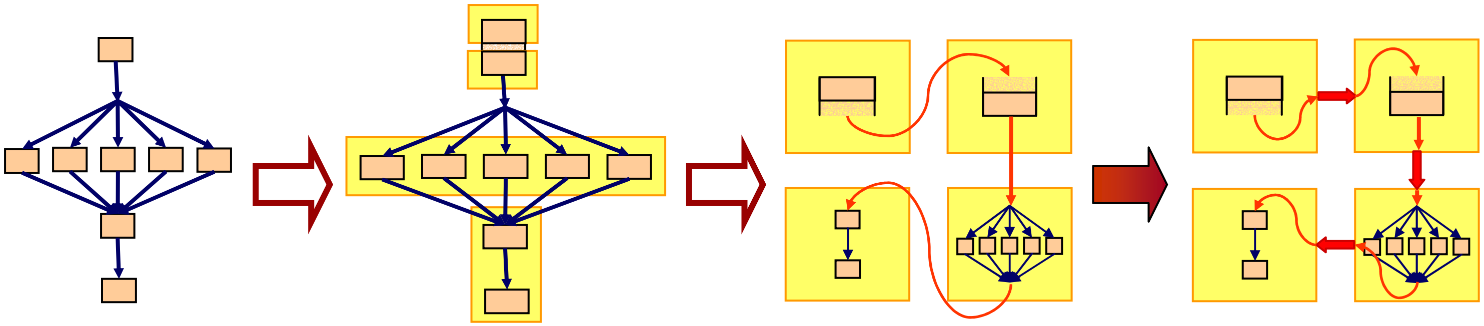


- Assign filters to tiles
  - Communicating filters → try to make them adjacent
  - Reduce overlapping communication paths
  - Reduce/eliminate cyclic communication if possible
- Compiler algorithm
  - Uses Simulated Annealing

Placement for Partitioned Radar Array Front End



Scheduling: Communication Orchestration

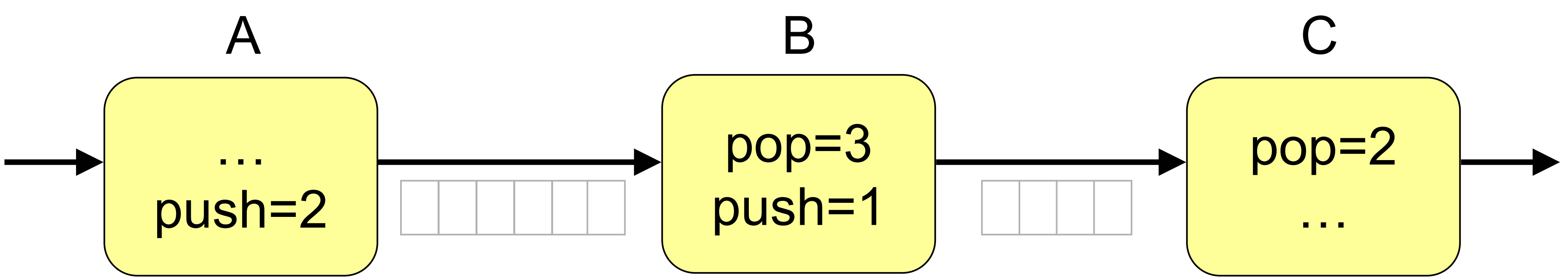


- Create a communication schedule
- Compiler Algorithm
  - Calculate an initialization and steady-state schedule
  - Simulate the execution of an entire cyclic schedule
  - Place static route instructions at the appropriate time

Steady-State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule

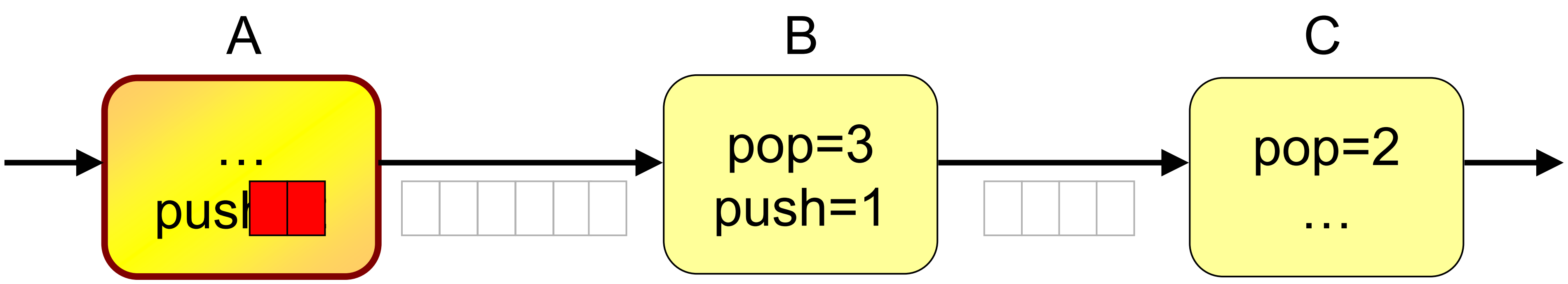
- Schedule = { }



Steady-State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule

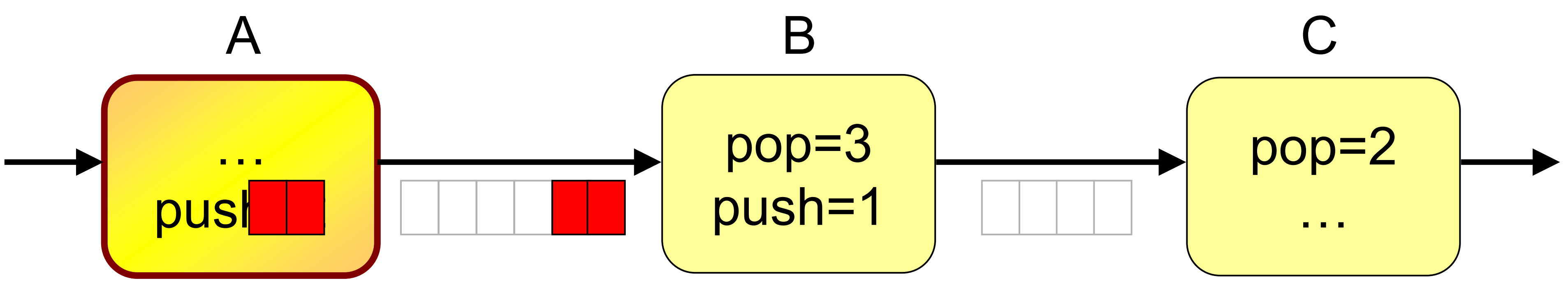
- Schedule = { A }



Steady-State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule

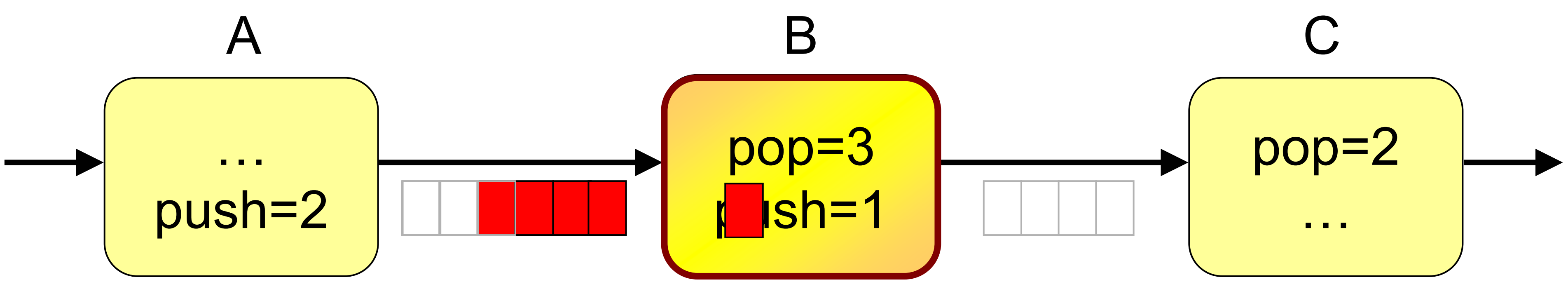
- Schedule = { A, A }



Steady-State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule

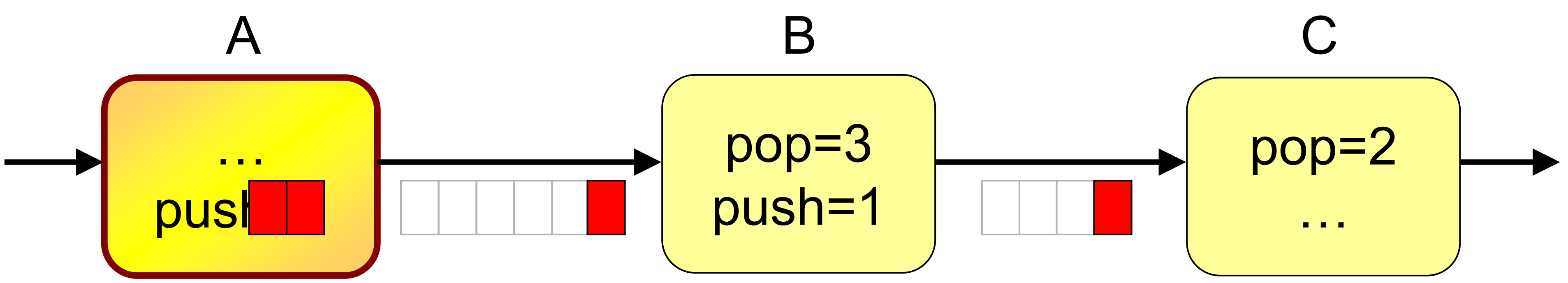
- Schedule = { A, A, B }



Steady-State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule

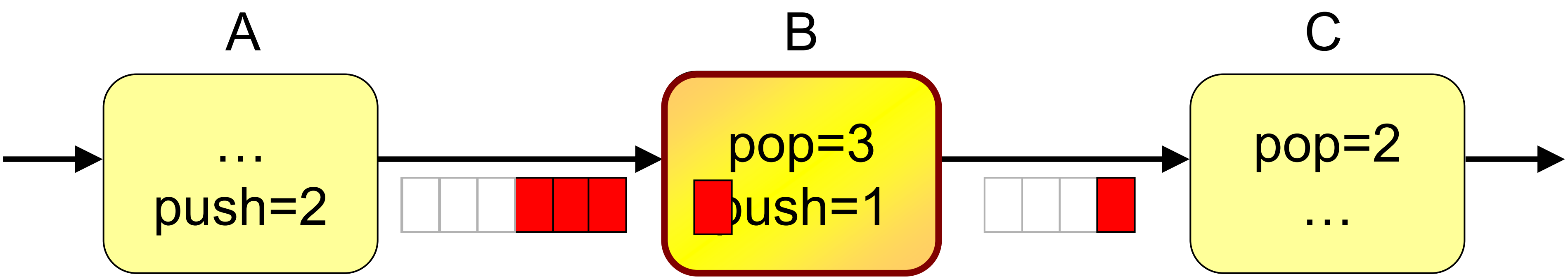
- Schedule = { A, A, B, A }



Steady-State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule

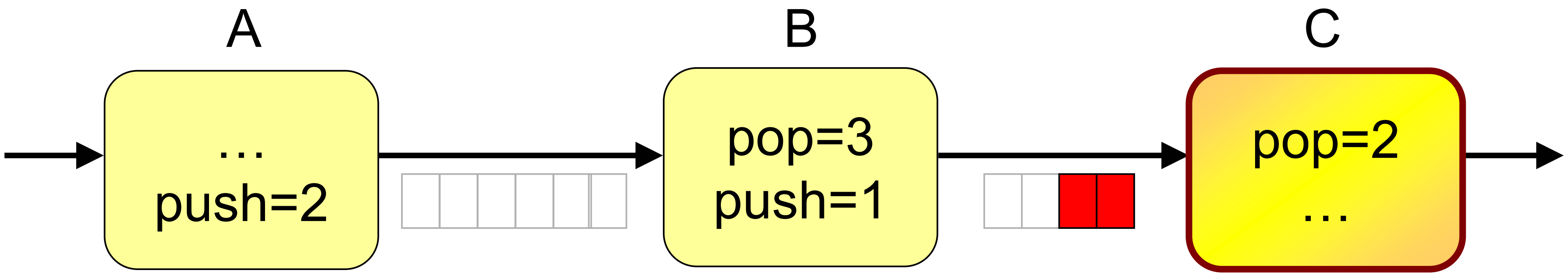
- Schedule = { A, A, B, A, B }



Steady-State Schedule

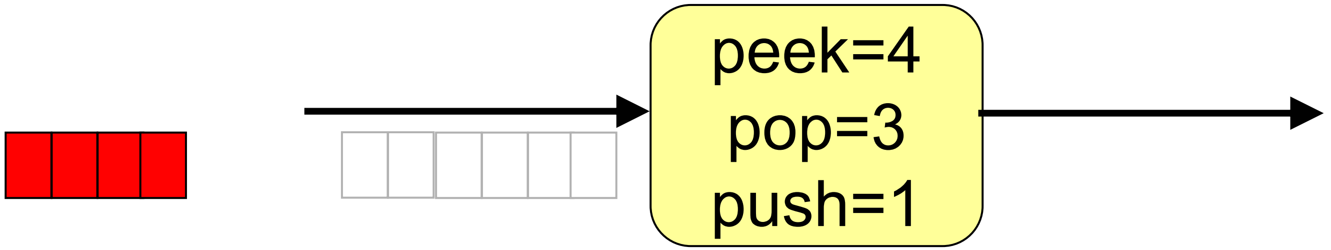
- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule

- Schedule = { A, A, B, A, B, C }



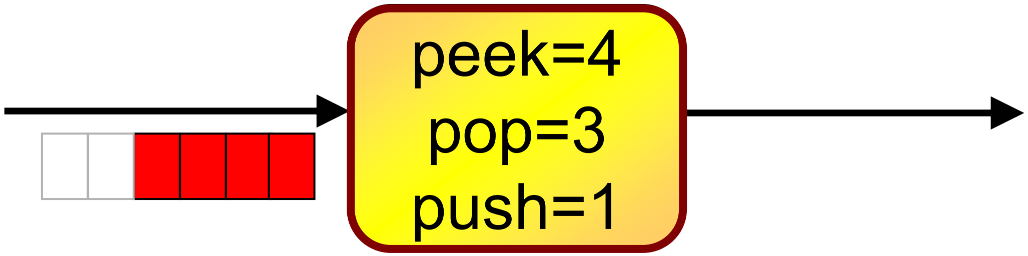
Initialization Schedule

- When peek > pop, buffer cannot be empty after firing a filter
- Buffers are not empty at the beginning/end of the steady state schedule
- Need to fill the buffers before starting the steady state execution

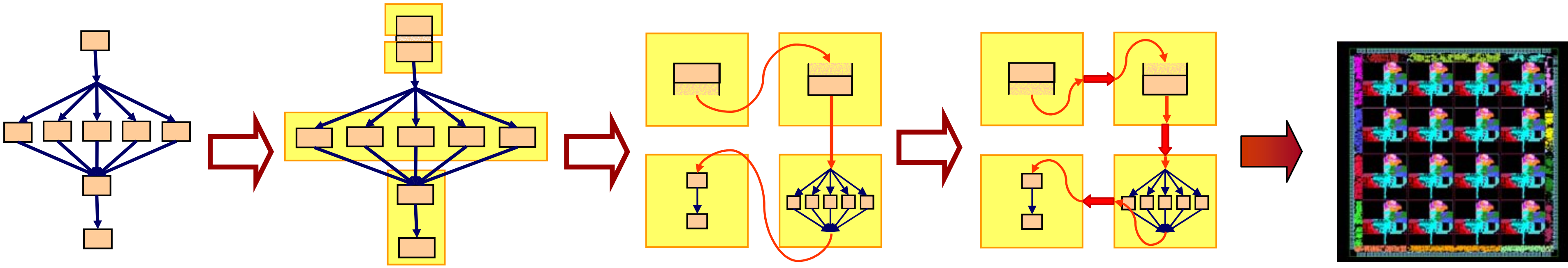


Initialization Schedule

- When peek > pop, buffer cannot be empty after firing a filter
- Buffers are not empty at the beginning/end of the steady state schedule
- Need to fill the buffers before starting the steady state execution

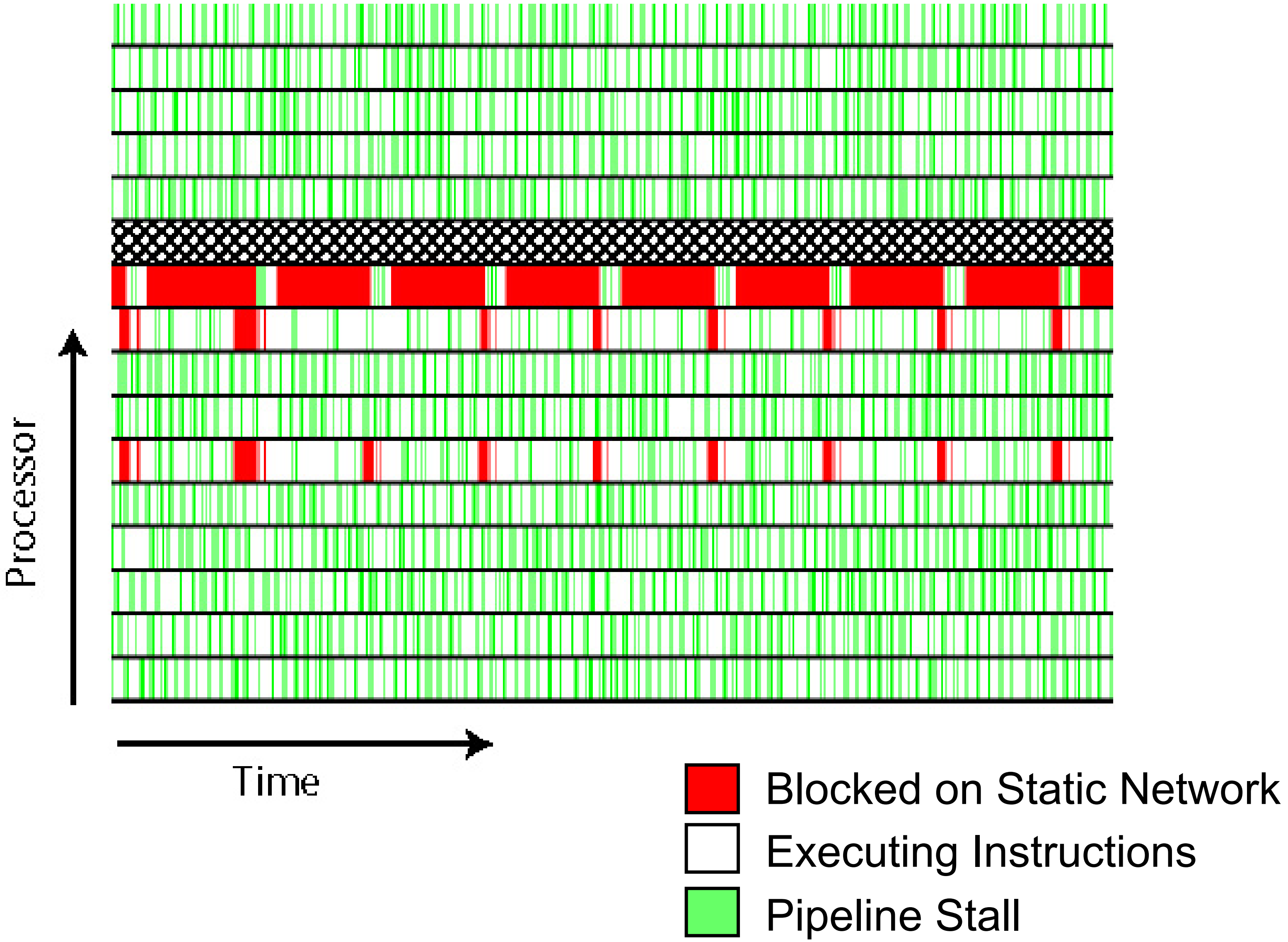


Code Generation: Optimizing tile performance

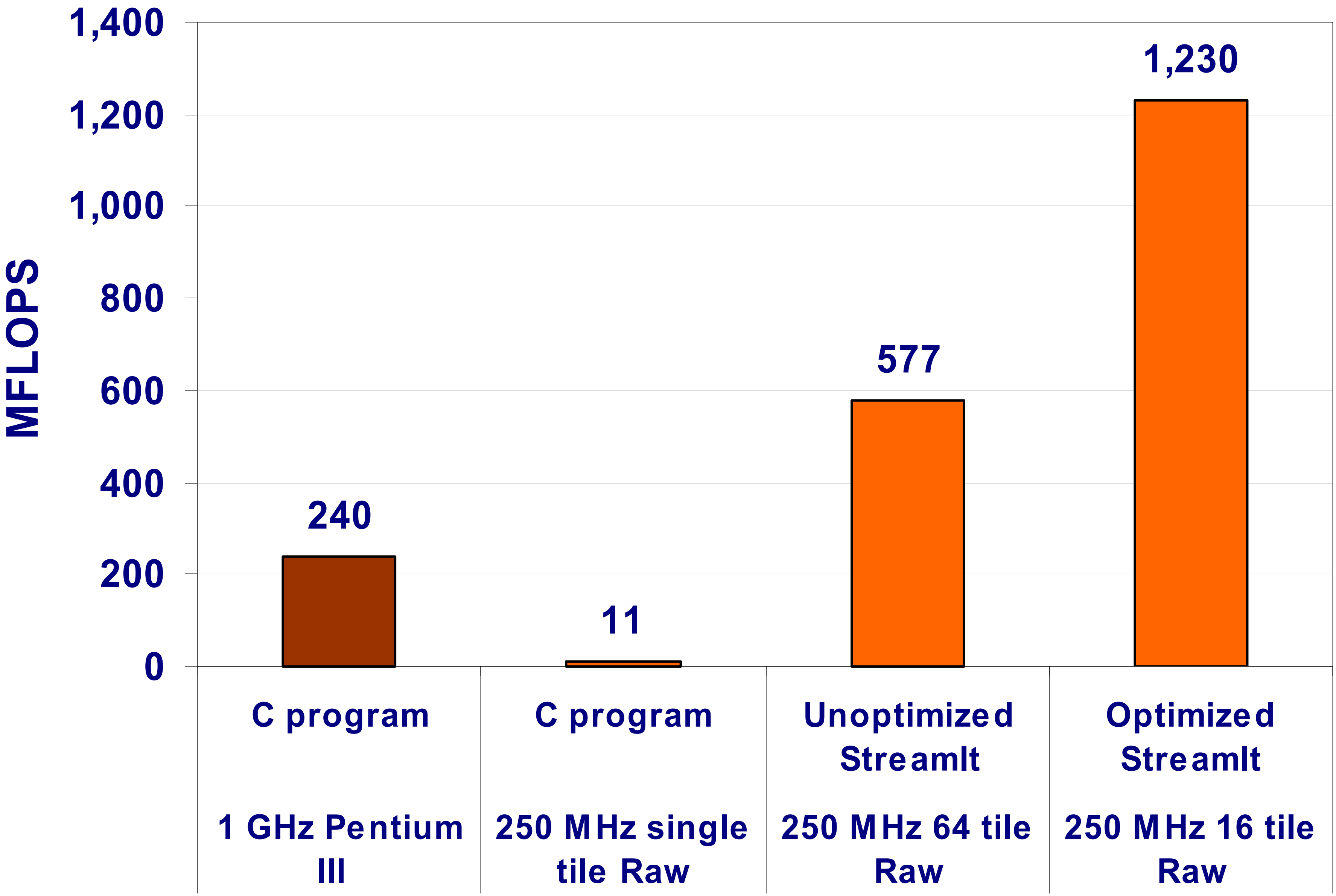


- Creates code to run on each tile
  - Optimized by the existing node compiler
- Generates the switch code for the communication

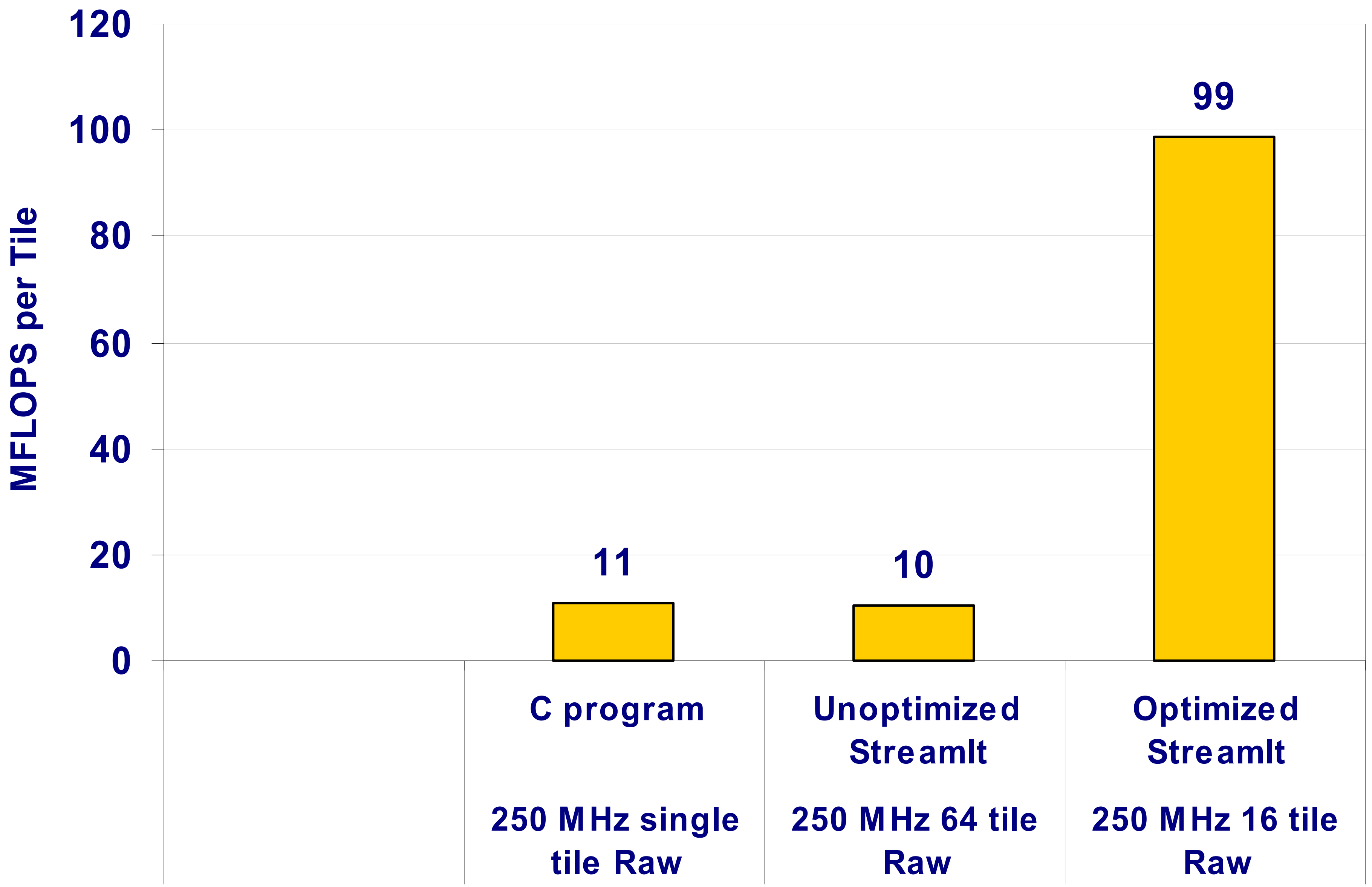
Performance Results for Radar Array Front End



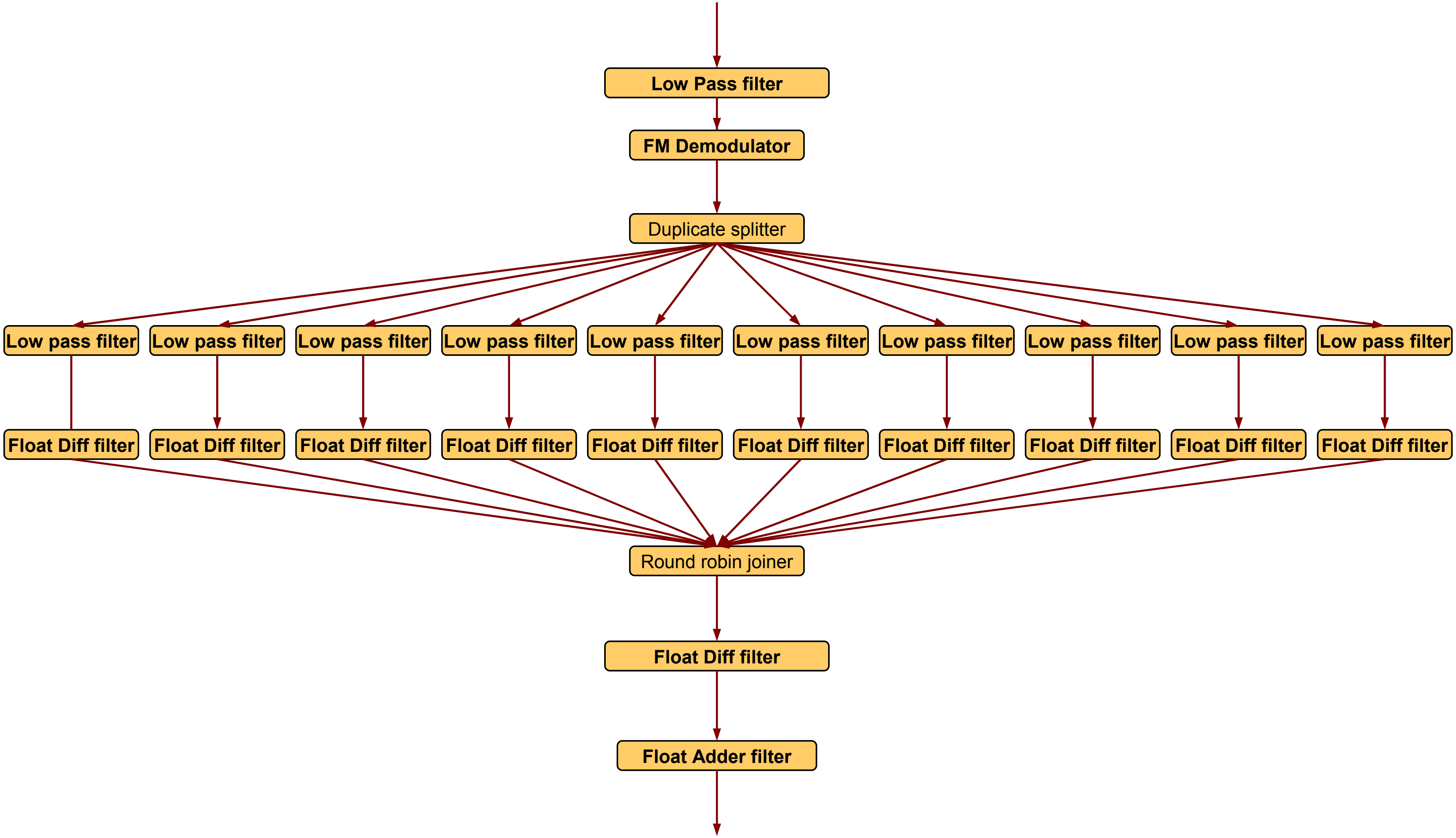
Performance of Radar Array Front End



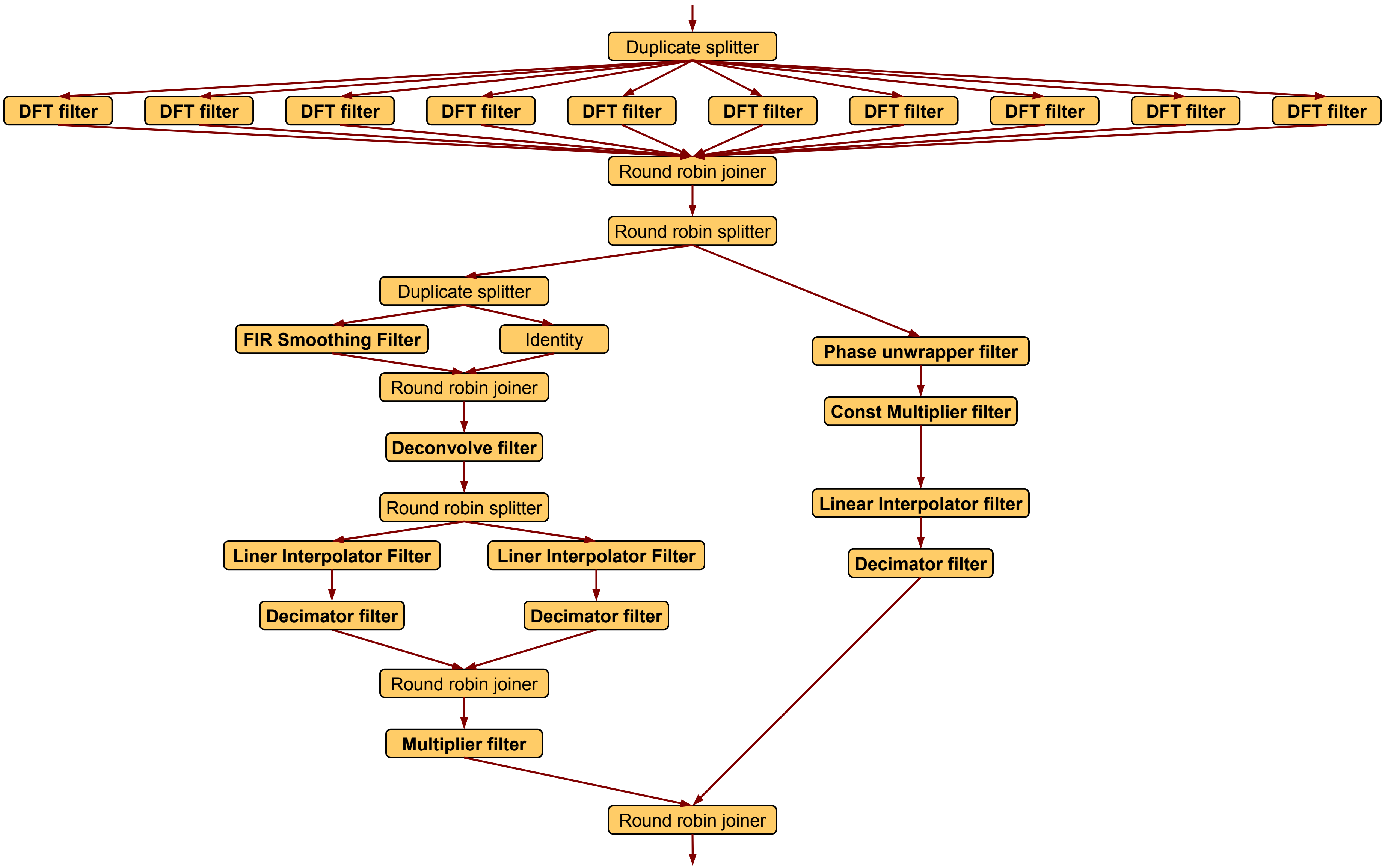
Utilization of Radar Array Front End



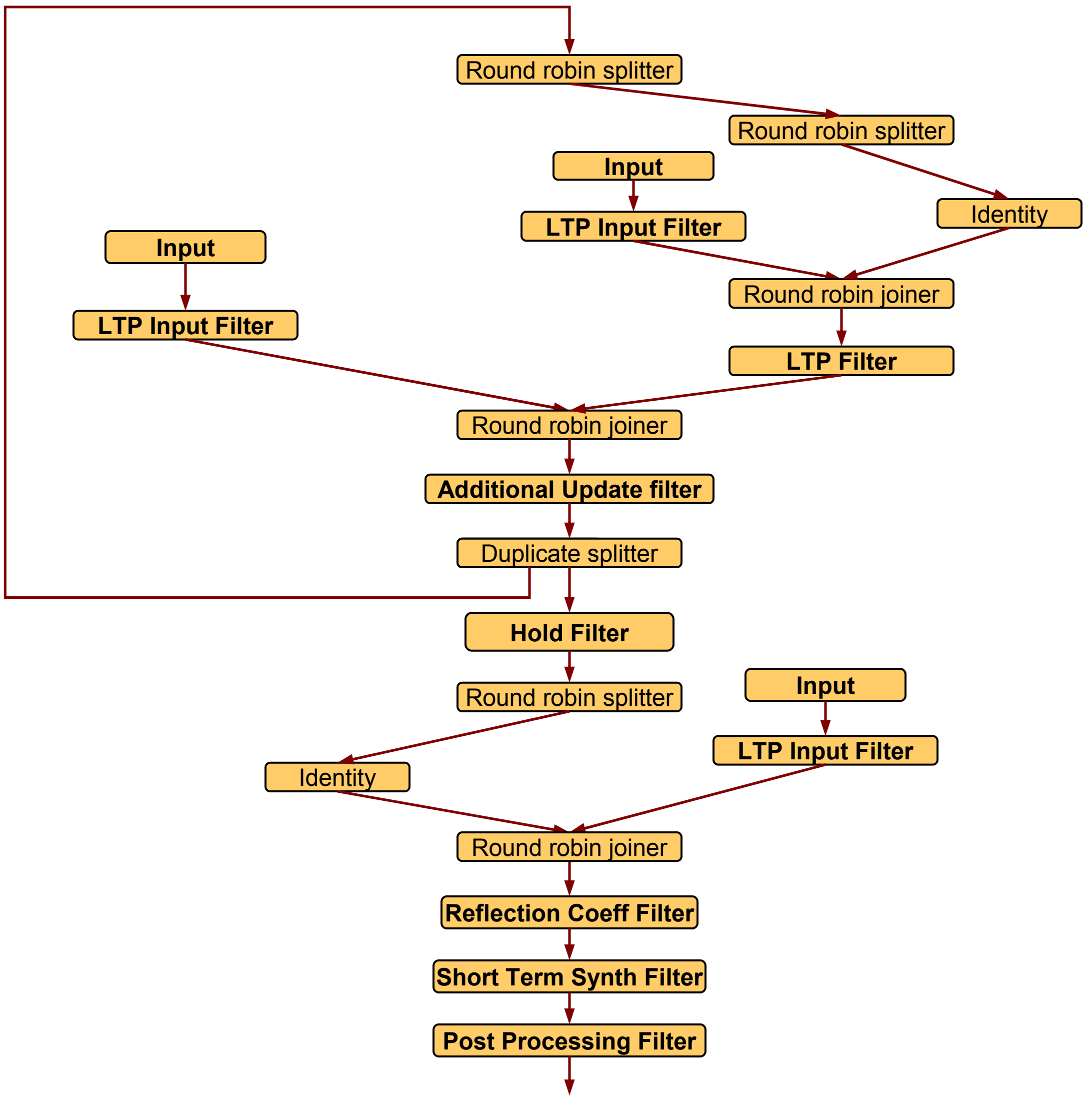
StreamIt Applications: FM Radio with an Equalizer



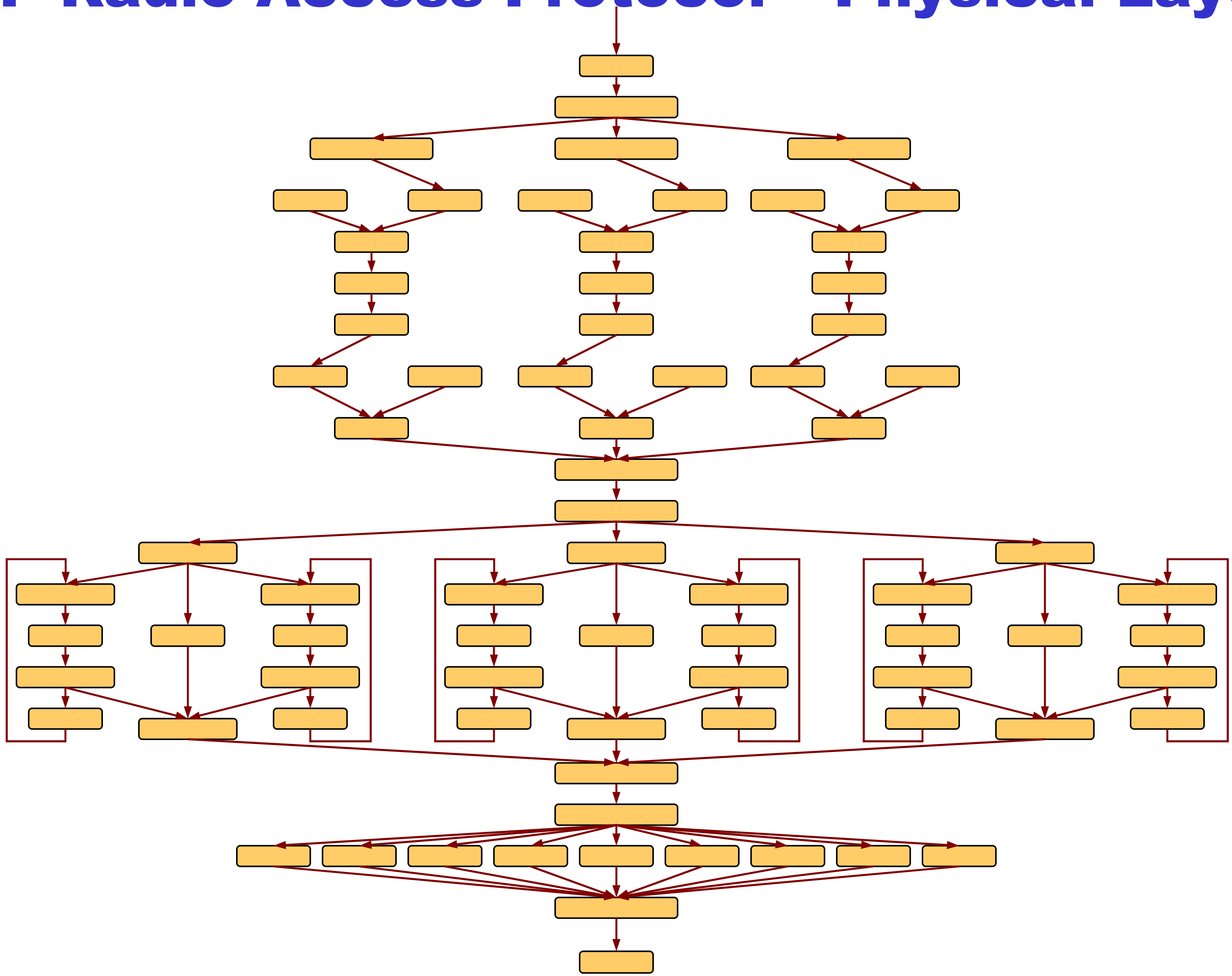
StreamIt Applications: Vocoder



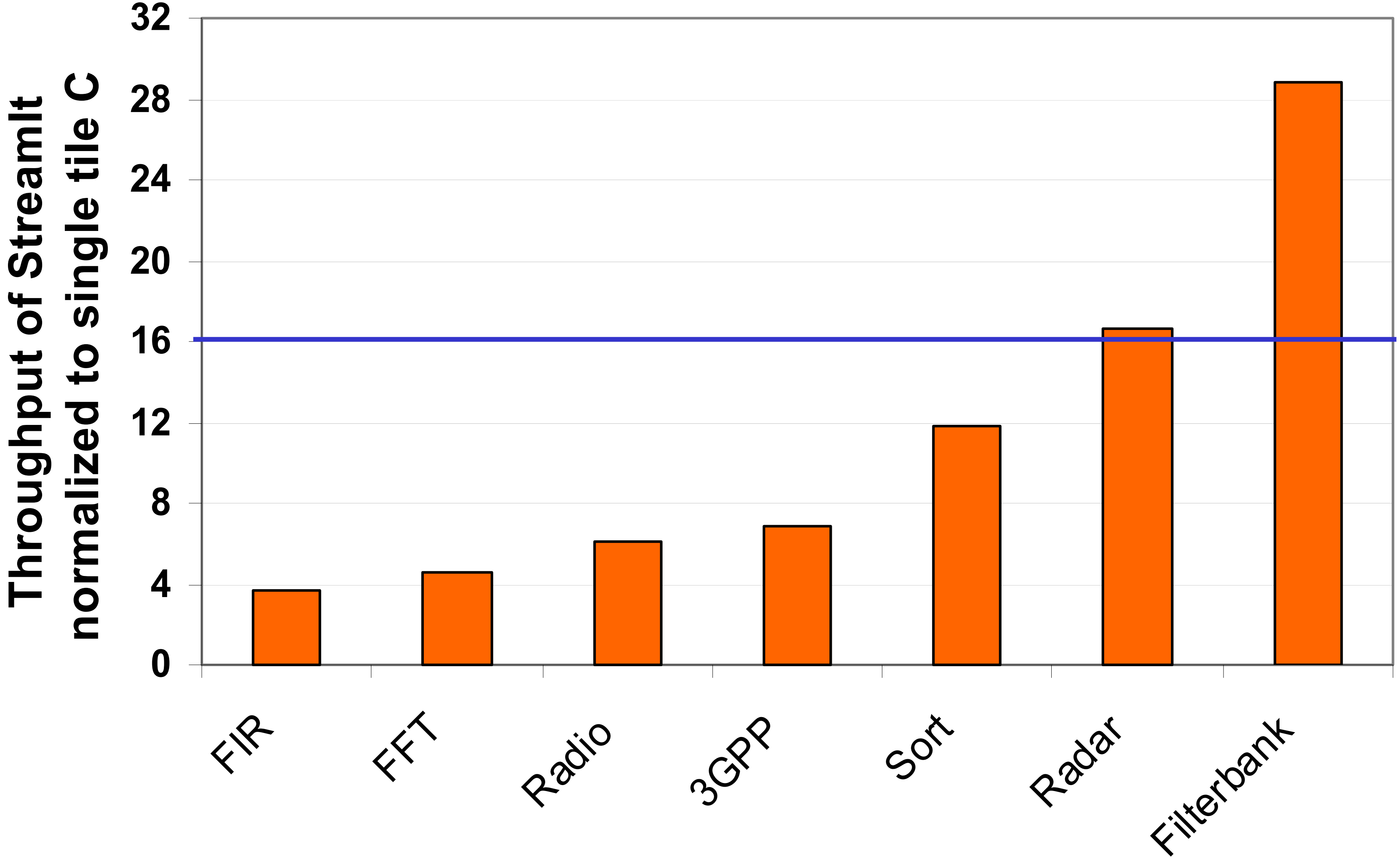
StreamIt Applications: GSM decoder



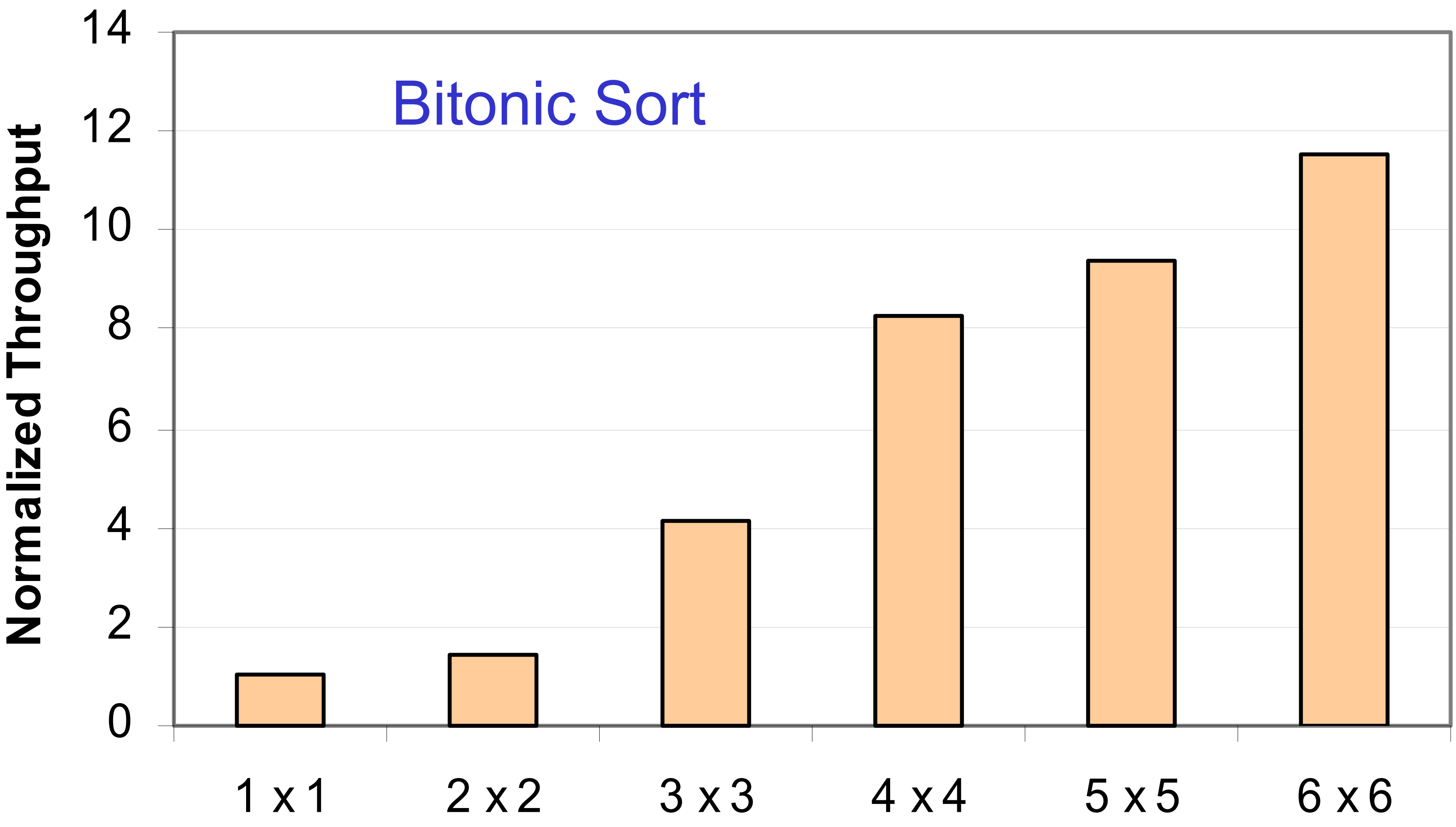
StreamIt Applications: 3GPP Radio Access Protocol – Physical Layer



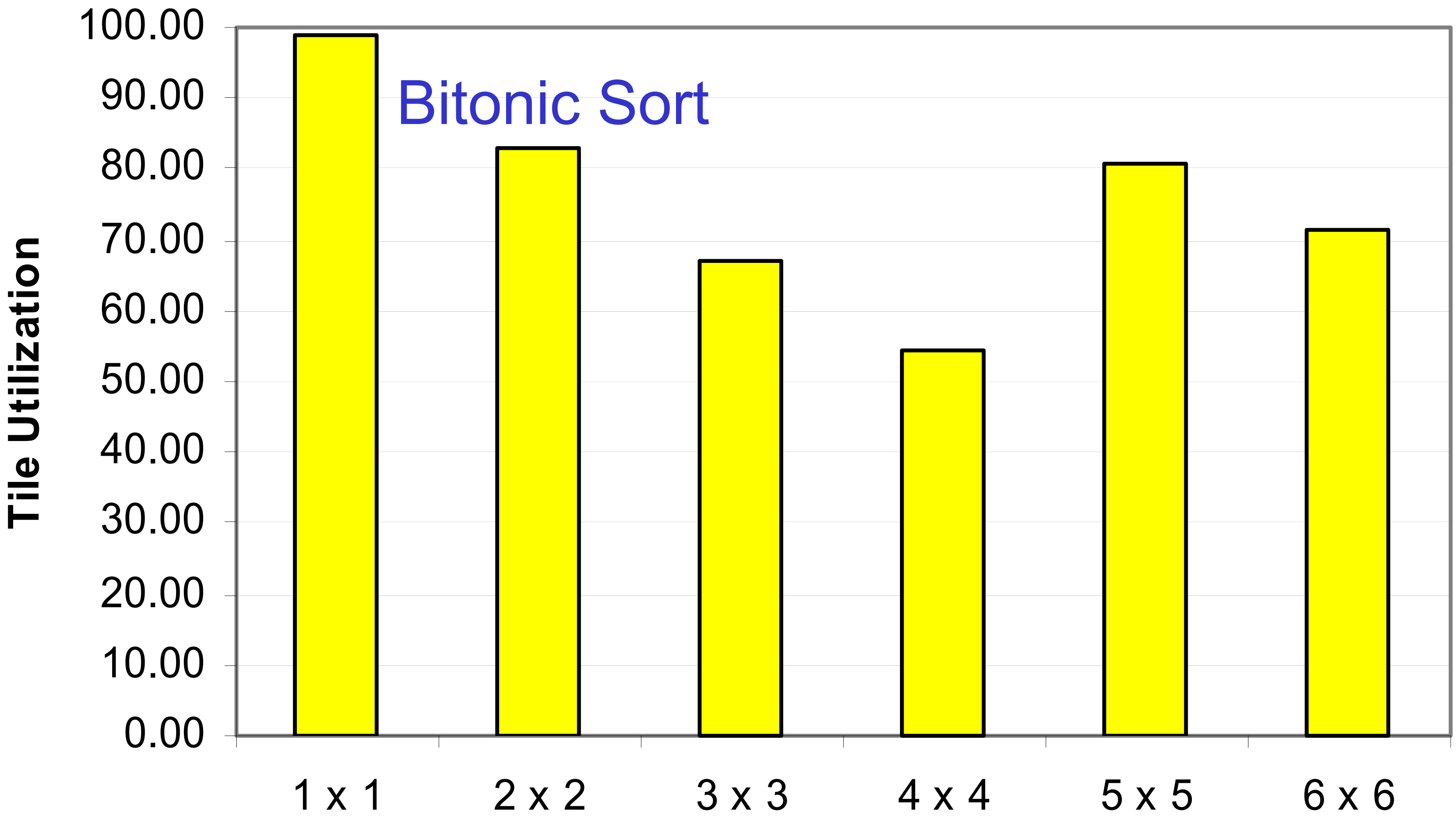
Application Performance



Scalability of StreamIt



Scalability of StreamIt



Related Work

- Stream-C / Kernel-C (Dally et. al)
  - Compiled to Imagine with time multiplexing
  - Extensions to C to deal with finite streams
  - Programmer explicitly calls stream “kernels”
  - Need program analysis to overlap streams / vary target granularity
- Brook (Buck et. al)
  - Architecture-independent counterpart of Stream-C / Kernel-C
  - Designed to be more parallelizable
- Ptolemy (Lee et. al)
  - Heterogeneous modeling environment for DSP
  - Many scheduling results shared with StreamIt
  - Don’t focus on language development / optimized code generation
- Other languages
  - Occam, SISAL – not statically schedulable
  - LUSTRE, Lucid, Signal, Esterel – don’t focus on parallel performance

Conclusion

- Streaming Programming Model
  - An important class of applications
  - Can break the von Neumann bottleneck
  - A natural fit for a large class of applications
  - Straightforward mapping to the architectural model
- StreamIt: A Machine Language for Communication Exposed Architectures
  - Expose the common properties
    - Multiple instruction streams
    - Software exposed communication
    - Fast local memory co-located with execution units
  - Hide the differences
    - Granularity of execution units
    - Type and topology of the communication network
    - Memory hierarchy
- A good compiler can eliminate the overhead of abstraction