Microsoft Research

Microsoft

Wireless programming for hardware dummies

Compiling stream processors for software-based low-latency network processing

Gordon Stewart (Princeton), Mahanth Gowda (UIUC), Geoff Mainland (Drexel), Bozidar Radunovic (MSR), Dimitrios Vytiniotis (MSR)

Motivation

- Lots of innovation in PHY/MAC design
- Popular experimental platform: GnuRadio
  - Easy to program but slow, no real network deployment
- Modern wireless PHYs require high-rate DSP
- Real-time platforms (SORA, Warp, …)
  - Achieve protocol processing requirements, difficult to program, no code portability (lots of manual hand-tuning)

Issues for wireless researchers

- SMP platforms (e.g. SORA)
  - Manual vectorization, CPU placement
  - Cache optimizations
- FPGA platforms (e.g. Warp)
  - Latency-sensitive design, difficult for new CS students/researchers to break into
- Portability/readability
  - (Manually) highly optimized code is difficult to read and maintain
  - Practically impossible to target another platform

Difficulty in writing and reusing code hampers innovation

Our goal

- New wireless programming platform
  1. Code written in a high-level language
  2. Compiler deals with low-level code optimization
  3. Same code compiles on different platforms (not there just yet!)
- Challenges
  1. Design PL abstractions that are intuitive and expressive
  2. Design efficient compilation schemes (to multiple platforms)
- What is special about wireless
  1. … that affects abstractions: large degree of separation b/w data and control
  2. … that affects compilation: need low latency stream processing

Related works

- SMP: SORA bricks (MSRA), GnuRadio blocks
  - **Language extension (templates) and lots of libraries**
- FPGA: Airblue
  - **Programmer deals with hardware low-level stuff (sync, queues, etc)**
- Control and data separation: CodiPhy, OpenRadio (Stanford)
- Streaming languages: StreamIt (MIT)
- Functional reactive programming: e.g. Yampa (Yale), Fran
- Dataflow languages e.g. Lucid (but no clocks here)

WPL: A 2-layer design

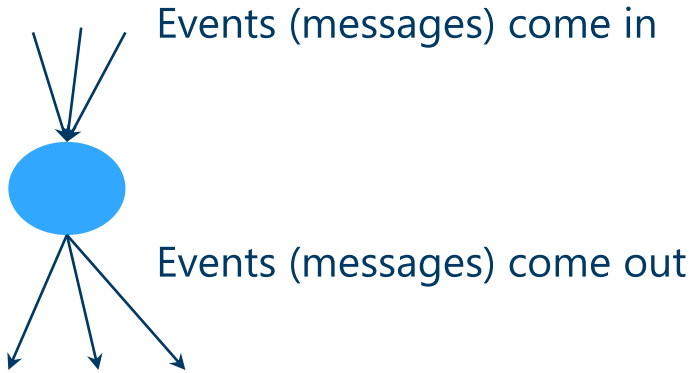
- Lower-level
  - Imperative C-like code for manipulating bits, bytes, arrays etc.
- Higher-level:
  - A monadic language for specifying and staging stream processors
  - Enforces clean separation between control and data flow
- Runtime implements low-level execution model
  - Inspired by stream fusion in Haskell
  - Provides efficient sequential and pipeline-parallel executions
- Monadic stream language enables aggressive compiler optimizations

Dataflow streaming abstractions

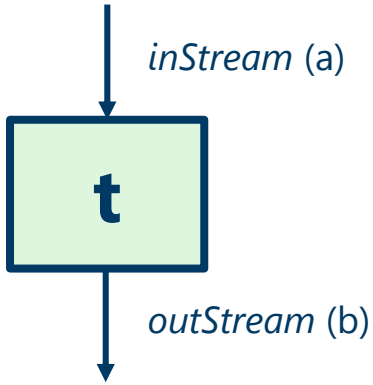
Predominant abstraction today (e.g. SORA, Streamlt, GnuRadio) is that of a "vertex" in a dataflow graph

- Reasonable as abstraction *of the execution model*
- Unsatisfactory as *programming and compilation* model

Why unsatisfactory? It does not expose:
(1) When is vertex state (re-) initialized?
(2) Under which external "control" messages can the vertex change behavior?
(3) How can vertex transmit "control" information to other vertices?

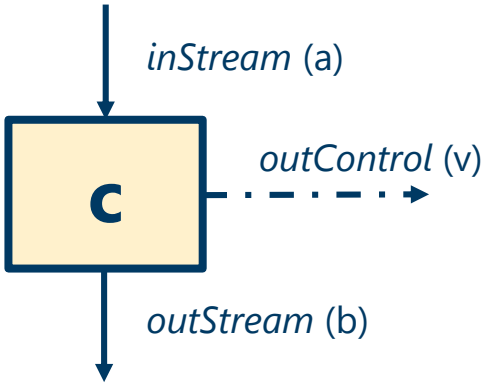


Control-aware streaming abstractions



A stream transformer t,
of type:

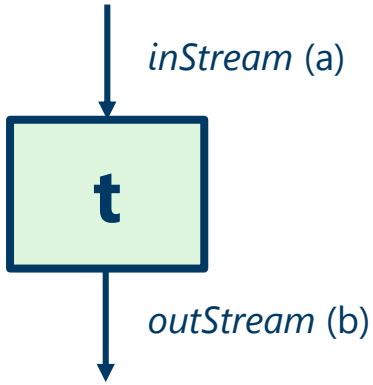
ST T a b



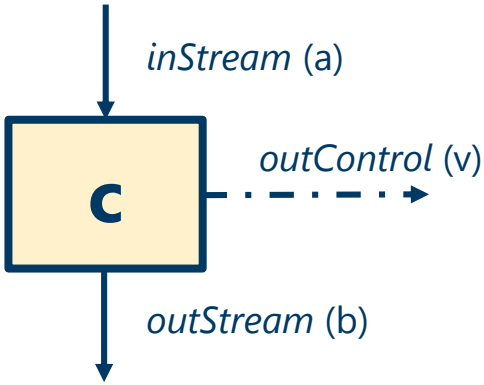
A stream computer c,
of type:

ST (C v) a b

Control-aware streaming abstractions




```
map    :: (a -> b) -> ST T a b
repeat :: ST (C ()) a b -> ST T a b
```




```
take :: ST (C a) a b
emit :: v -> ST (C ()) a v
```

Horizontal and vertical composition

```
(>>>) :: ST T a b     -> ST T b c     -> ST T a c
(>>>) :: ST (C v) a b -> ST T b c     -> ST (C v) a c
(>>>) :: ST T a b     -> ST (C v) b c -> ST (C v) a c
```

Composition along "control path" (like a monad*)

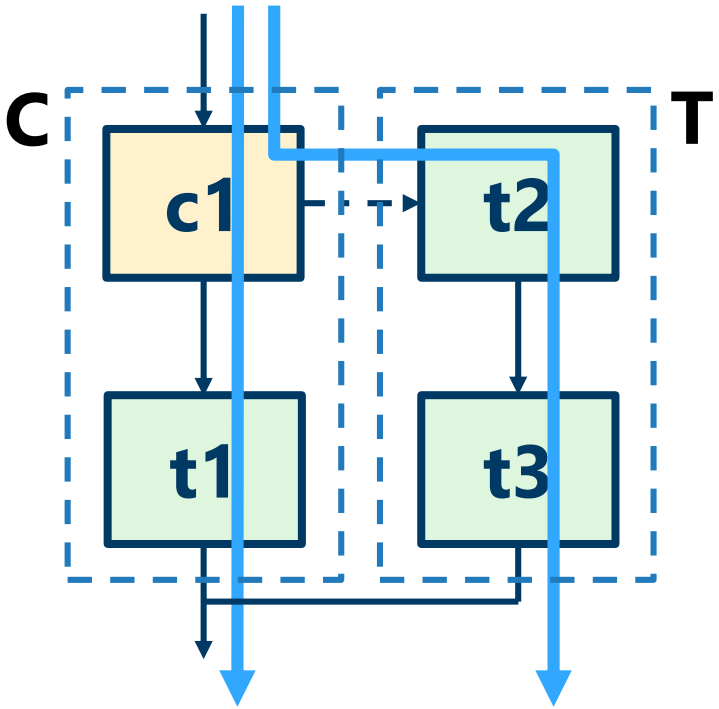
Composition along "data path" (like an arrow)

```
(>>=)  :: ST (C v) a b -> (v -> ST x a b) -> ST x a b
return :: v -> ST (C v) a b
```

* This is like Yampa's `switch`, but using different channels for control and data

Staging a pipeline, in diagrams

```
do { v <- (c1 >>> t1)
   ; t2 >>> t3
   }
```



Simple example: scrambler

```
let scrambler(u : unit) =
  var scrmbl_st: arr[7] bit;
  var tmp: bit;
  var y:bit;

  return(
    for i in 0,7 {
      scrmbl_st[i] := bit(1)
    }
  );

  repeat (
    x <- take1;

    return (
      tmp := (scrmbl_st[3] ^ scrmbl_st[0]);
      for i in 0,6 {
        scrmbl_st[i] := scrmbl_st[i+1]
      };
      scrmbl_st[6] := tmp;
      y := x ^ tmp
    );

    emit (y)
  )
in
```

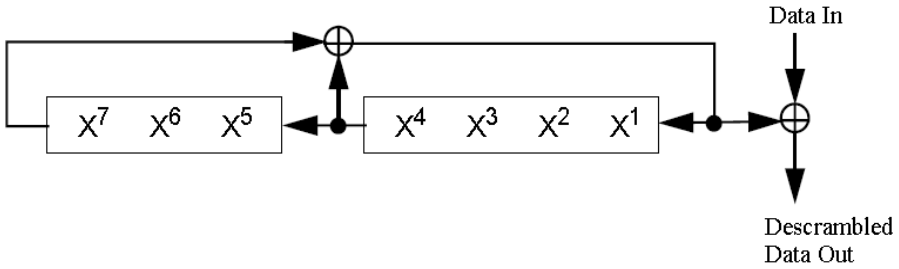
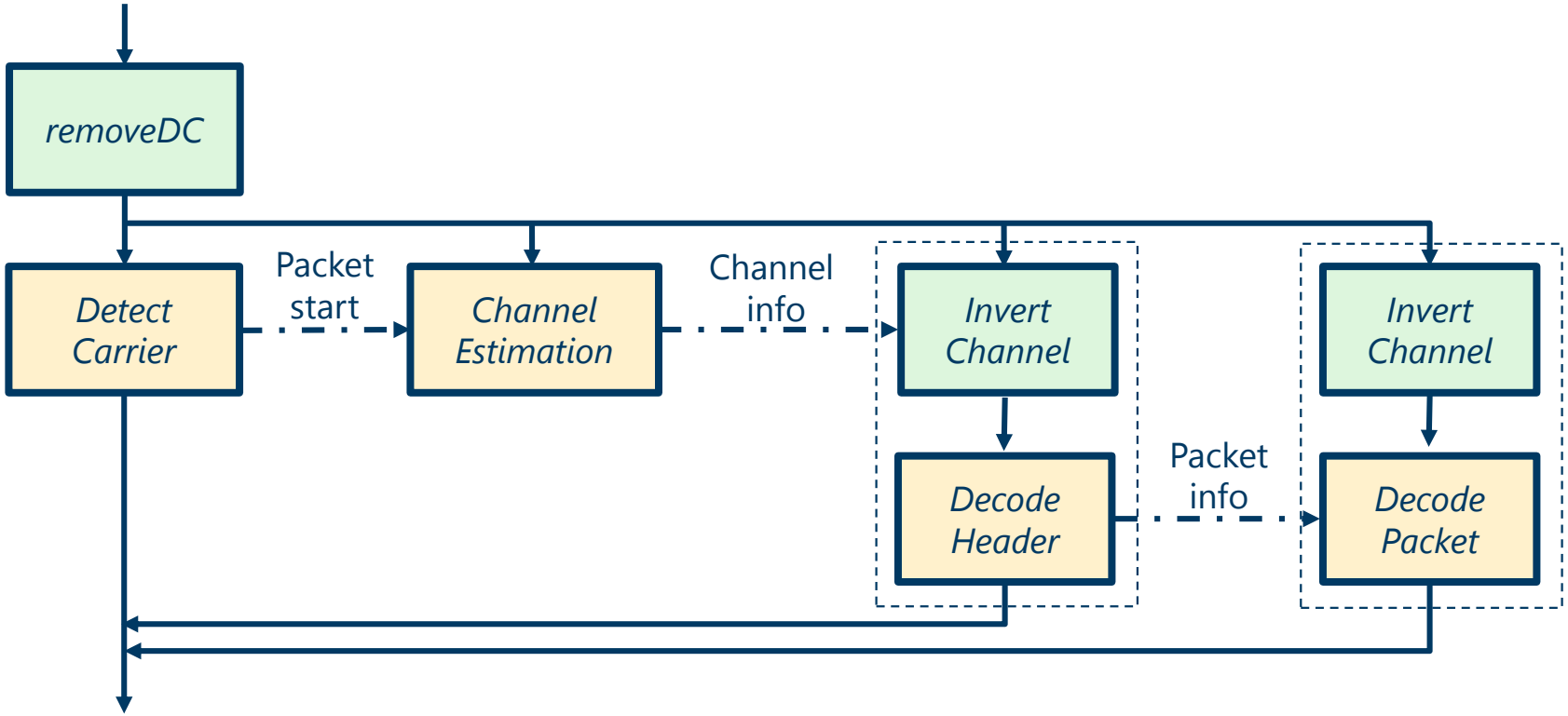


Figure 113—Data scrambler

WiFi receiver (simplified)



~~Semantics~~ Execution model (SMP)

Every component `(ST (C v) a b)` "compiles" to 2 functions:

$$\text{tick} : \text{Void} \rightarrow (\text{Result v a b} + \text{NeedInput})$$

$$\text{process} : \text{a} \longrightarrow \text{Result v a b}$$

$$\text{Result v a b} = \text{Skip} \mid \text{Yield b} \mid \text{Done v}$$

$$\text{NeedInput} = ()$$

Details more intricate: components have **state** and our execution model reflects that

Similar to the datatype used in stream fusion

Execution model (continued)

Runtime loop:

```
1: Let t = top-level-component
2: whatis := t.tick()
3: if whatis == Yield b
    then putBuf(b) ; goto 2
    else if whatis == Skip then goto 2
    else if whatis == Done then exit()
    else if whatis == NeedInput then
     c = getBuf(); whatis := t.process(); goto 3.
```

In reality:

- Very few function calls with a CPS-based translation: every “process” function knows its continuation
- Optimizations: never tick components with trivial tick(), never generate process() for tick()-only components
- Only indirection is for bind: at different points in times, function pointers point to the correct “process” and “tick”
- Slightly different approach to input/output

Ticking a bind / sequence

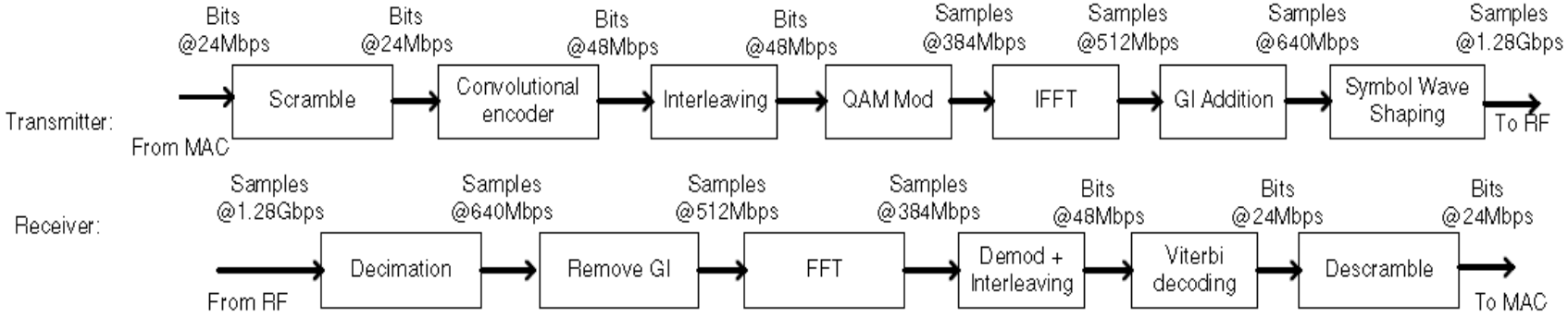
[[c1 >>= c2]] :=

```
{ init := c1.init();
 , tick := c1.tick()
 , process := \a -> case c1.process(a) of
       Skip -> Skip
       Yield b -> Yield b
       Done v -> (c2 v).init();
                 tick := (c2 v).tick()
                 process := (c2 v).process() }
```

[[c1 >>> c2]] :=

```
{ init := c1.init(); c2.init();
, tick := case c2.tick() of
            Result r -> Result r
            NeedInput -> case c1.tick() of
               Skip -> Skip
               Emit b -> c2.process(b)
               NeedInput -> NeedInput
, process := \a ->
         case c1.process(a) of
           Skip -> Skip
           Emit b -> c2.process(b) }
```

Speed! (Optimizations)



(b) IEEE 802.11a/g 24Mbps

* From the SORA paper, [NSDI 2009]

Auto-vectorization

- Convert pipelines automatically to work with arrays

  ST x a b ~~~> ST x (arr n a) (arr m b)

- Challenges: How to figure out the right multiplicities?
- Implemented "cardinality analysis"
- Searching space of vectorizations in two modes:
  - Scale-up vectorization
  - Scale-down vectorization

Scale-up vectorization

```
let block(u:unit) =
    var y:int;

    repeat(
      (x : int) <- take1;
      return(y := x+1);
      emit (y)
    )
```

```
let block_VECTORIZED (u: unit) =
          var y: int;
          repeat let vect_up_wrap_46 () =
                   var vect_ya_48: arr[4] int;
                   (vect_xa_47 : arr[4] int) <- take1;
                   __unused_174 <- times 4 (\vect_j_50. (x : int) <- return vect_xa_47[0*4+vect_j_50*1+0];
                                                        __unused_1 <- return y := x+1;
                                                        return vect_ya_48[vect_j_50*1+0] := y);
                   emit vect_ya_48
                 in
                 vect_up_wrap_46 (tt)
```

Program transformations

```
let block_VECTORIZED (u: unit) =
          var y: int;
          repeat let vect_up_wrap_46 () =
                   var vect_ya_48: arr[4] int;
                   (vect_xa_47 : arr[4] int) <- take1;
                   __unused_174 <- times 4 (\vect_j_50. (x : int) <- return vect_xa_47[0*4+vect_j_50*1+0];
                                                        __unused_1 <- return y := x+1;
                                                        return vect_ya_48[vect_j_50*1+0] := y);
                   emit vect_ya_48
                 in
                 vect_up_wrap_46 (tt)
```

Dataflow graph iteration converted to tight loop!
In this case we got x3 speedup

```
let block_VECTORIZED (u: unit) =
          var y: int;
          repeat let vect_up_wrap_46 () =
                    var vect_ya_48: arr[4] int;
                    (vect_xa_47 : arr[4] int) <- take1;
                    emit let __unused_174 = for vect_j_50 in 0, 4 {
                                                 let x = vect_xa_47[0*4+vect_j_50*1+0]
                                                 in let __unused_1 = y := x+1
                                                    in vect_ya_48[vect_j_50*1+0] := y
                                               }
                           in vect_ya_48
                 in vect_up_wrap_46 (tt)
```

Scale-down vectorization

For components that take/emit many elements

```
let t11aDataSymbol(u:unit) =
   repeat (
    (xp:arr[80] complex) <- take 80;
     emits xp[16:79]
   )
```

```
repeat let vect_dn_8 () =
                var vect_xa_9: arr[80] int;
                var vect_ya_10: arr[64] int;
                 __unused_33 <- times 20 (\vect_i_11. (xtemp_12 : arr[4] int) <- take1;
                                                      return vect_xa_9[vect_i_11*4:+4] := xtemp_12);
                let (xp : arr[80] int) = vect_xa_9[0:+80]
                in
                let vect_res_13 = vect_ya_10[0:+64] := xp[16:+64]
                in
                __unused_32 <- times 16 (\vect_i_11. emit vect_ya_10[vect_i_11*4:+4]);
                return vect_res_13
```

Vectorization boundaries and queues

```
if c then
   repeat (take 3 elements; emit 4 elements)
else
   repeat (take 2 elements; emit 3 elements)
```

First path can be vectorized to:
3*n*k – 4*k
Second path can be vectorized to:
2*n'*k' – 3*k'

Least "good" input queue = 72
TOO LARGE!

Solution: introduce queues as primitives, to take pressure off the vectorizer

read :: QueueId -> ST BUF a
write :: QueueId -> ST a BUF

```
if c then
   repeat (take 3 elements; emit 4 elements) >>> write(out)
else
   repeat (take 2 elements; emit 3 elements) >>> write(out)
```

Data paths now vectorize independently!

Requires type-directed compilation of read/write

This path vectorizes to 6-4

```
if c then
  repeat (take 3 elements; emit 4 elements) >>> write(out)
else
  repeat (take 2 elements; emit 3 elements) >>> write(out)
```

This path vectorizes to 6-3

Pipelining with SMPs

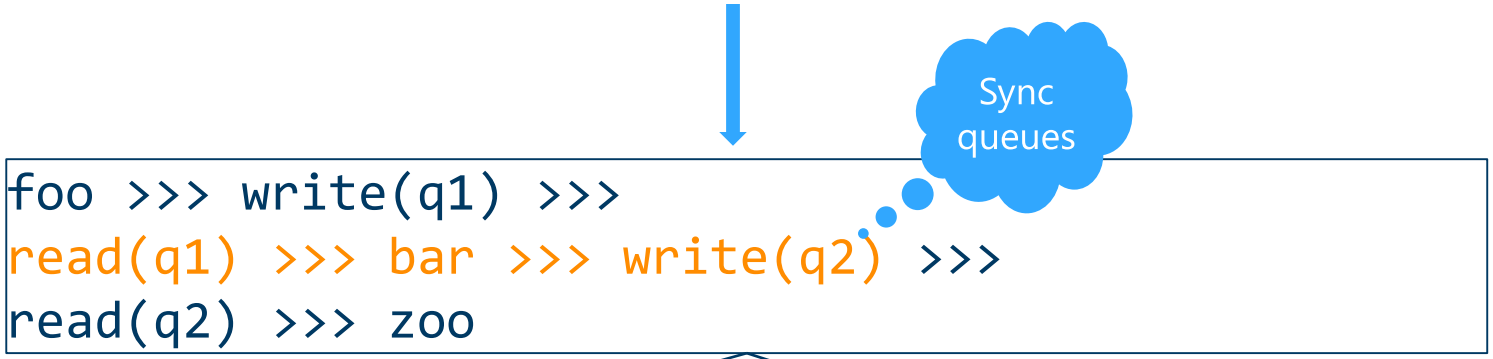
802.11a transmitter:

Opportunity to pipeline parallelize

```
read >>> (
   hInfo <- emitHeader(tt) >>> scrambler(tt) >>>
                               encode(12)    >>>
                               interleaver(bpsk) >>>
                               modulate(bpsk) >>> map_ofdm(tt)) ;
  scrambler(tt) >>> encode(hInfo[2])
                >>> interleaver(hinfo[1]) >>> modulate(hinfo[1]) >>> map_ofdm(tt)
   ) >>> write
```

Pipelining with SMPs

```
foo >>> bar >>> zoo
```




```
foo >>> write(q1) >>>
read(q2) >>> zoo
```

Thread 1, pin to Core 1

```
read(q1) >>> bar >>> write(q2)
```

Thread 2, pin to Core 2

Status report

- Fully working language and compiler implementation
- Other features: SIMD-programming library
- Interfacing with external C-functions
- Re-using SORA driver (for faster kernel-space run)
- Vectorizer *really* works: 2x faster on the complete Wifi receiver pipeline, up to 4x faster on individual components.
- Processing rate: single-CPU, SIMD+vectorized ~ 200ms/20MB = twice as fast as the protocol requirements

In the pipeline

Working on:

1. Finalizing pipeline parallelization
2. Detailed profiling and evaluation
3. Writing paper, implementing more challenging protocols (4G LTE)

Future:

1. Cost models of execution model and vectorizer
2. FPGA backend, heterogeneous compilation
3. Verification of arithmetic floating point errors
4. Resource bounds prediction or modeling

Join us!

As users, or developers …