Microsoft Research

Microsoft

Optimizing reconfigurable pipelines in ZIRIA

Gordon Stewart (Princeton), Mahanth Gowda (UIUC),
Geoff Mainland (Drexel), Cristina Luengo (UPC), Anton Ekblad (Chalmers),
Bozidar Radunovic (MSR), Dimitrios Vytiniotis (MSR)

What is ZIRIA*

- A programming language for bit stream and packet processing
- Programming abstractions well-suited for wireless PHY implementations in software (e.g. 802.11a/g)
- Optimizing compiler that generates real-time code
- Developed @ MSR Cambridge, open source under Apache 2.0

  www.github.com/dimitriv/Ziria

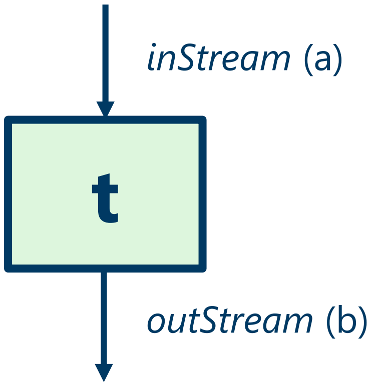
  http://research.microsoft.com/projects/Ziria
- Repo includes a protocol compliant line-rate WiFi RX & TX PHY implementation

* In past presentations referred to as "WPL" and "Blink"

ZIRIA: A 2-level language

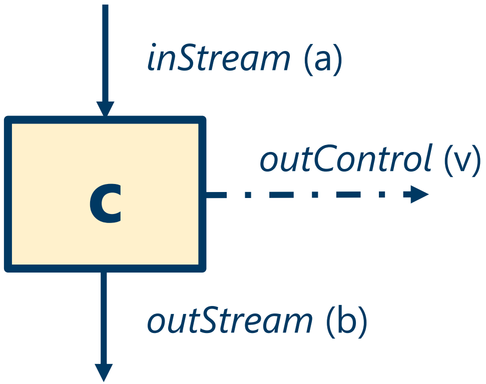
- Lower-level
  - Imperative C-like language for manipulating bits, bytes, arrays, etc.
  - Aimed at EE crowd (used to C and Matlab)
- Higher-level:
  - Monadic language for specifying and composing stream processors
  - Enforces clean separation between control and data flow
  - Intuitive semantics (in a process calculus)
- Runtime implements low-level execution model
  - inspired by stream fusion in Haskell
  - provides efficient sequential and pipeline-parallel executions

ZIRIA programming abstractions



A stream transformer t,
of type:

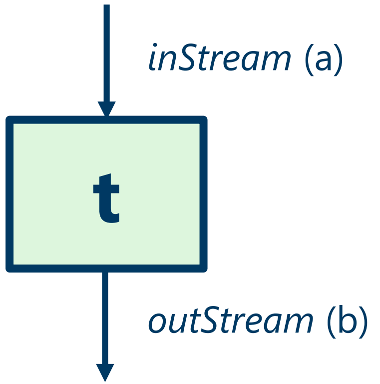
ST T a b



A stream computer c,
of type:

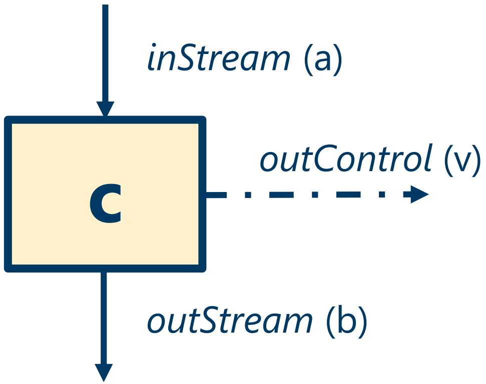
ST (C v) a b

ZIRIA programming abstractions



A stream transformer t,
of type:

ST T a b

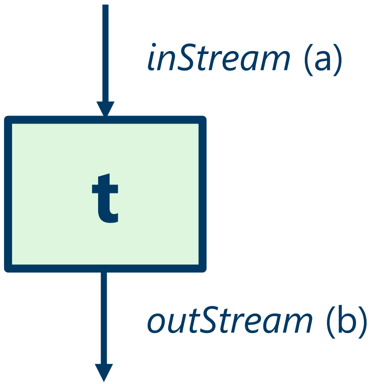


A stream computer c,
of type:

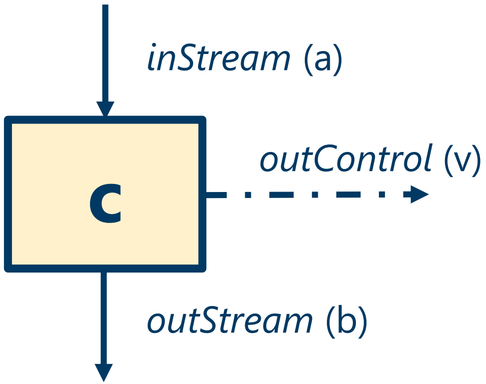
ST (C v) a b

* Types similar to (but a lot simpler than) Haskell Pipes types

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
```

Data- and control-path 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
```

Data- and control-path 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
```

* Like Yampa's `switch`, but using different channels for control and data

Data- and control-path 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)

Reinventing a classic:
The “Fudgets” GUI monad
[Carlsson & Hallgren, 1996]

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

* Like Yampa’s `switch`, but using different channels for control and data

Data- and control-path 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
```

Slightly unusual semantics

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

Composition along "data path" (like an arrow)

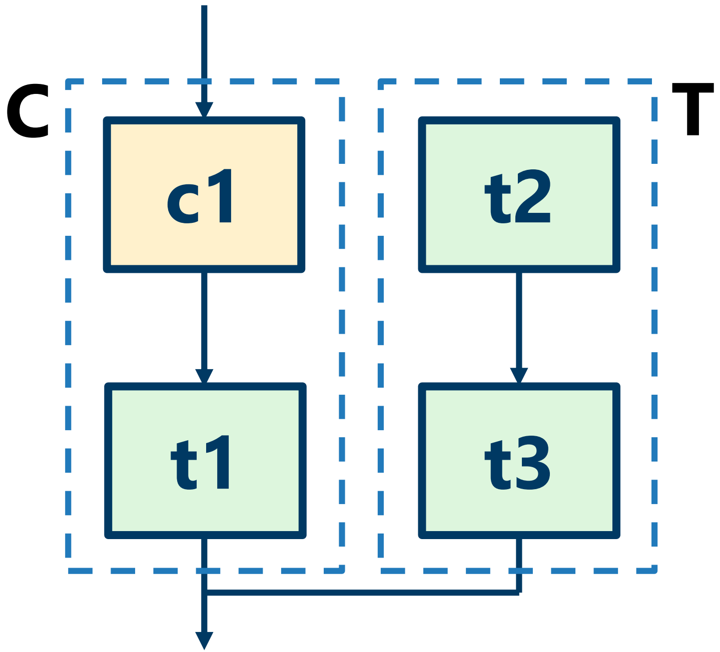
Reinventing a classic:
The "Fudgets" GUI monad
[Carlsson & Hallgren, 1996]

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

* Like Yampa's `switch`, but using different channels for control and data

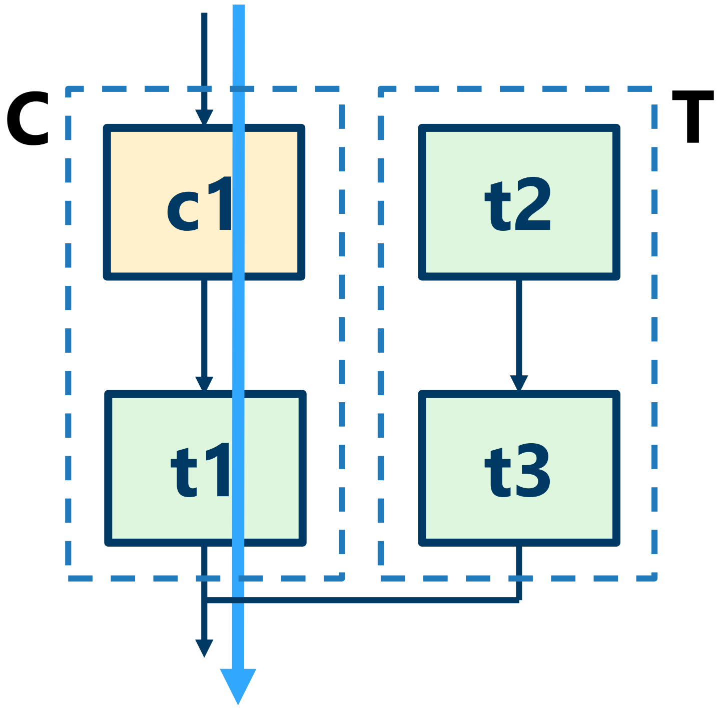
Composing pipelines, in diagrams

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



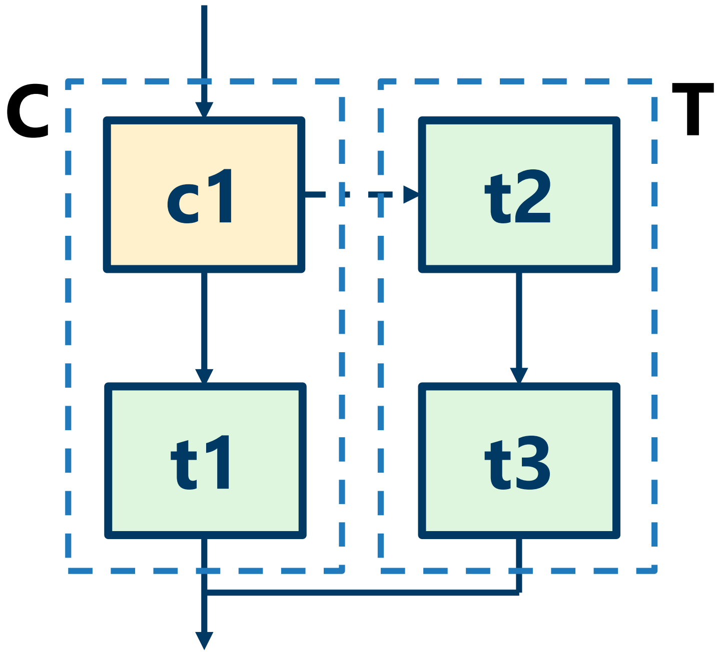
Composing pipelines, in diagrams

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



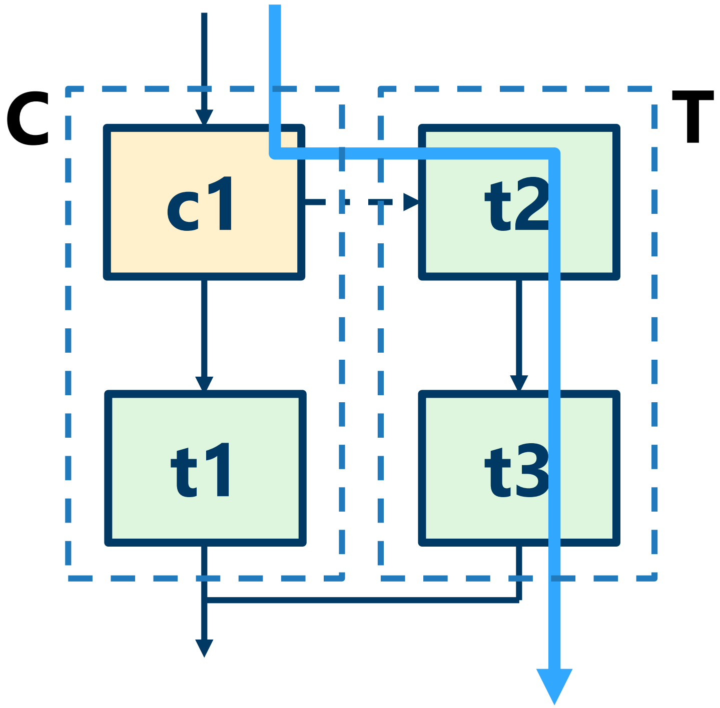
Composing pipelines, in diagrams

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

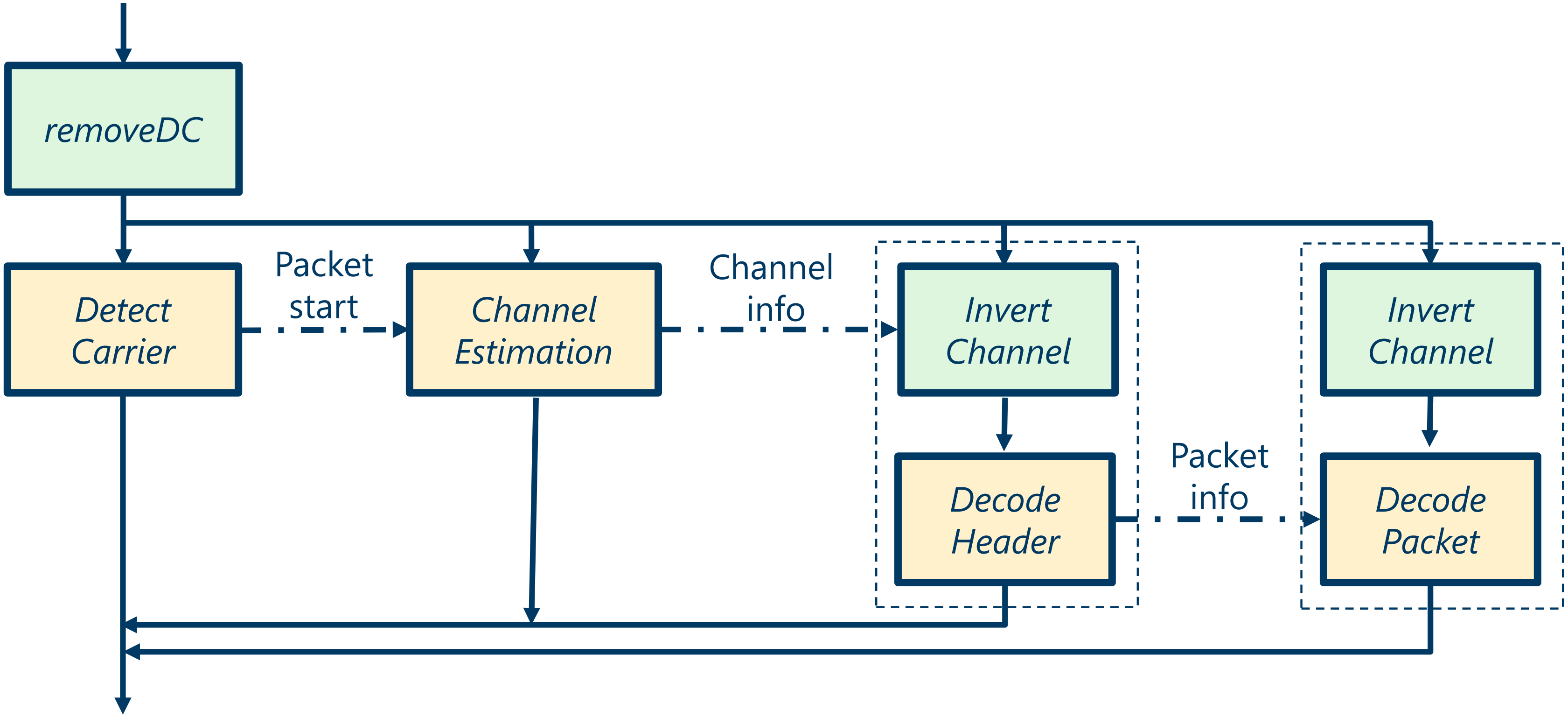


Composing pipelines, in diagrams

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



WiFi receiver (simplified)



Fitting together low and high-level parts

Low-level imperative code

```
let comp scrambler() =
  var scrmbl_st: arr[7] bit := {'1,'1,'1,'1,'1,'1,'1};
  var tmp,y: bit;

  repeat {
      (x:bit) <- take;
      do {
        tmp := (scrmbl_st[3] ^ scrmbl_st[0]);
        scrmbl_st[0:5] := scrmbl_st[1:6];
        scrmbl_st[6] := tmp;
        y := x ^ tmp
      };

      emit (y)
  }
```

Optimizing ZIRIA code

1. Exploit monad laws, partial evaluation
2. Fuse parts of dataflow graphs
3. Reuse memory, avoid redundant memcopying
4. Compile expressions to lookup tables (LUTs)
5. Pipeline vectorization transformation
6. Pipeline parallelization

Optimizing ZIRIA code

1. Exploit monad laws, partial evaluation
2. Fuse parts of dataflow graphs
3. Reuse memory, avoid redundant memcopying
4. Compile expressions to lookup tables (LUTs)
5. Pipeline vectorization transformation
6. Pipeline parallelization

The rest
of the talk

Pipeline vectorization

Problem statement: given (c :: ST x a b), automatically rewrite it to

c_vect :: ST x (arr[N] a) (arr[M] b)

for suitable N,M.

Pipeline vectorization

Problem statement: given (c :: ST x a b), automatically rewrite it to

c_vect :: ST x (arr[N] a) (arr[M] b)

for suitable N,M.

Benefits of vectorization

- Fatter pipelines => lower dataflow graph interpretive overhead
- Array inputs vs individual elements => more data locality
- Especially for bit-arrays, enhances effects of LUTs

Computer vectorization feasible sets

```
seq { x <- takes 80
    ; var y : arr[64] int
    ; do { y := f(x) }
    ; emit y[0]
    ; emit y[1]
    }
```

Computer vectorization feasible sets

```
seq { x <- takes 80
    ; var y : arr[64] int
    ; do { y := f(x) }
    ; emit y[0]
    ; emit y[1]
    }
```

1. Assume we have *cardinality info*: # of values the component takes and emits before returning (Here: ain = 80, aout = 2)
2. Feasible vectorization set:
   { (din,dout) | din `divides` ain,
   dout `divides` aout }

Computer vectorization feasible sets

```
seq { x <- takes 80
    ; var y : arr[64] int
    ; do { y := f(x) }
    ; emit y[0]
    ; emit y[1]
    }
```

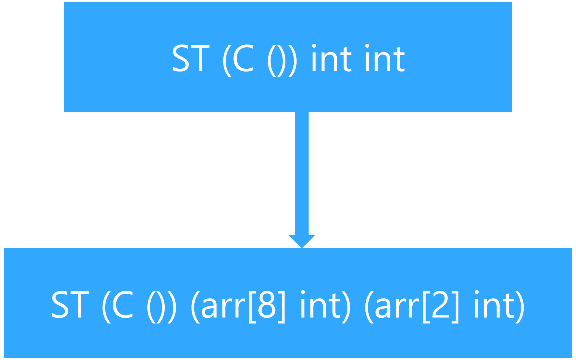
1. Assume we have *cardinality info*: # of values the component takes and emits before returning (Here: ain = 80, aout = 2)
2. Feasible vectorization set:
   { (din,dout) | din \`divides\` ain,
   dout \`divides\` aout }

e.g.
din = 8,
dout =2

```
seq { var x : arr[80] int
    ; for i in 0..10 {
          (xa : arr[8] int) <- take;
           x[i*8,8] := xa;
      }
    ; var y : arr[64] int
    ; do { y := f(x) }
    ; emit y }
```

Computer vectorization feasible sets

```
seq { x <- takes 80
    ; var y : arr[64] int
    ; do { y := f(x) }
    ; emit y[0]
    ; emit y[1]
    }
```



1. Assume we have *cardinality info*: # of values the component takes and emits before returning (Here: ain = 80, aout = 2)
2. Feasible vectorization set:
   { (din,dout) | din \`divides\` ain,
   dout \`divides\` aout }

e.g.
din = 8,
dout =2

```
seq { var x : arr[80] int
    ; for i in 0..10 {
          (xa : arr[8] int) <- take;
           x[i*8,8] := xa;
      }
    ; var y : arr[64] int
    ; do { y := f(x) }
    ; emit y }
```

Impl. keeps feasible *sets* and not just singletons

```
seq { x <- c1
    ; c2
    }
```

c1_v1 :: ST (C v) (arr[80] int) (arr[2] int)
c1_v2 :: ST (C v) (arr[16] int) (arr[2] int)
....

c2_v1 ::ST (C v) (arr[24] int) (arr[2] int)
c2_v2 :: ST (C v) (arr[16] int) (arr[2] int)
....

Well-typed choice:
c1_v1 and c2_v2
Hence: we must keep sets

Transformer vectorizations

Without loss of generality, every ZIRIA transformer can be treated as:

`repeat c`

where c is a computer

How to vectorize (**`repeat c`**)?

Transformer vectorizations in isolation

How to vectorize (`repeat c`)?

- Let c have cardinality info (ain, aout)
- Can vectorize to all divisors of ain (aout) [as before]
-

Transformer vectorizations in isolation

How to vectorize (**`repeat c`**)?

- Let c have cardinality info (ain, aout)
- Can vectorize to all divisors of ain (aout) [as before]
- Can also vectorize to all multiples of ain (aout)

Transformer vectorizations in isolation

How to vectorize (`repeat c`)?

- Let c have cardinality info (ain, aout)
- Can vectorize to all divisors of ain (aout) [as before]
- Can also vectorize to all multiples of ain (aout)

Why? It's a TRANSFORMER, it's supposed to always have data to process

Transformer vectorizations in isolation

How to vectorize (**`repeat c`**)?

- Let c have cardinality info (ain, aout)
- Can vectorize to all divisors of ain (aout) [as before]
- Can also vectorize to all multiples of ain (aout)

Why? It's a TRANSFORMER, it's supposed to always have data to process

```
repeat { x <- take
       ; emit f(x)
       }
```

ST T int int

```
repeat {
  (vect_xa : arr[8] int) <- take;
  times i 2 {
    times j 4 {
      do { vect_ya[j] := f(vect_xa[i*4 + j]) }
    }
    emit vect_ya;
  }
}
```

ST T (arr[8] int) (arr[4] int)

Transformers-before-computers

``It’s a
TRANSFORMER,
it’s supposed to
always have data to
process”

Transformers-before-computers

LET ME QUESTION THIS ASSUMPTION

`It’s a TRANSFORMER, it’s supposed to always have data to process”

```
seq { x <- (repeat c) >>> c1
    ; c2 }
```

Transformers-before-computers

LET ME QUESTION THIS ASSUMPTION

"It's a TRANSFORMER, it's supposed to always have data to process"

```
seq { x <- (repeat c) >>> c1
    ; c2 }
```

Assume c1 vectorizes to input (arr[4] int)

Transformers-before-computers

LET ME QUESTION THIS ASSUMPTION

"It's a TRANSFORMER, it's supposed to always have data to process"

Assume c1 vectorizes to input (arr[4] int)

```
seq { x <- (repeat c) >>> c1
    ; c2 }
```

ain = 1, aout =1

Transformers-before-computers

LET ME QUESTION THIS ASSUMPTION

"It's a TRANSFORMER, it's supposed to always have data to process"

Assume c1 vectorizes to input (arr[4] int)

```
seq { x <- (repeat c) >>> c1
    ; c2 }
```

ain = 1, aout =1

QUIZ: Is vect. (ST T (arr[8] int) (arr[4] int) correct?

Transformers-before-computers

- ANSWER: No! (repeat c) may consume data destined for c2 after the switch
- SOLUTION: consider (K*ain, N*K*aout), NOT arbitrary multiples°

Assume c1 vectorizes to input (arr[4] int)

```
seq { x <- (repeat c) >>> c1
    ; c2 }
```

ain = 1, aout =1

QUIZ: Is vect. (ST T (arr[8] int) (arr[4] int) correct?

Transformers-before-computers

- ANSWER: No! (repeat c) may consume data destined for c2 after the switch
- SOLUTION: consider (K*ain, N*K*aout), NOT arbitrary multiples°

(°) *caveat: assumes that* (repeat c) >>> c1 *terminates when* c1 *and* c *have returned. No "unemitted" data from* c

Assume c1 vectorizes to input (arr[4] int)

```
seq { x <- (repeat c) >>> c1
    ; c2 }
```

ain = 1, aout =1

QUIZ: Is vect. (ST T (arr[8] int) (arr[4] int) correct?

Transformers-after-computers

```
seq { x <- c1 >>> (repeat c)
    ; c2 }
```

Transformers-after-computers

Assume c1 vectorizes to output (arr[4] int)

```
seq { x <- c1 >>> (repeat c)
    ; c2 }
```

ain = 1, aout =1

Transformers-after-computers

Assume c1 vectorizes to output (arr[4] int)

```
seq { x <- c1 >>> (repeat c)
    ; c2 }
```

ain = 1, aout =1

QUIZ: Is vect. (ST T (arr[4] int) (arr[8] int) correct?

Transformers-after-computers

- ANSWER: No! (repeat c) may not have a full 8-element array to emit when c1 terminates!

- SOLUTION: consider (N*K*ain, K*aout), NOT arbitrary multiples [**symmetrically to before**]

```
seq { x <- c1 >>> (repeat c)
    ; c2 }
```

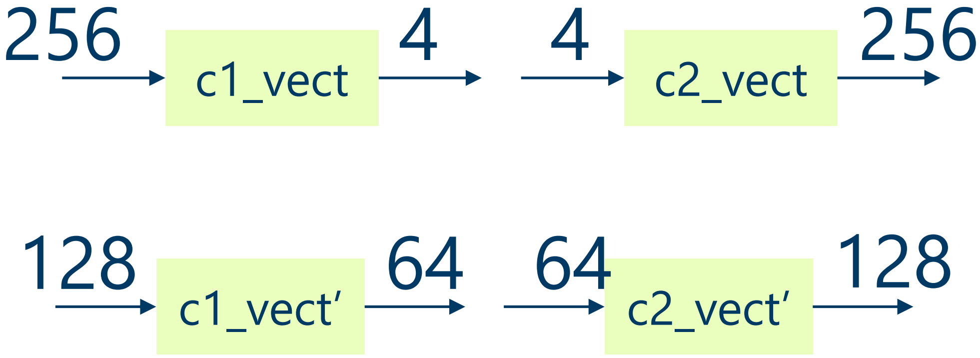
Assume c1 vectorizes to output (arr[4] int)

ain = 1, aout =1

QUIZ: Is vect. (ST T (arr[4] int) (arr[8] int) correct?

How to choose final vectorization?

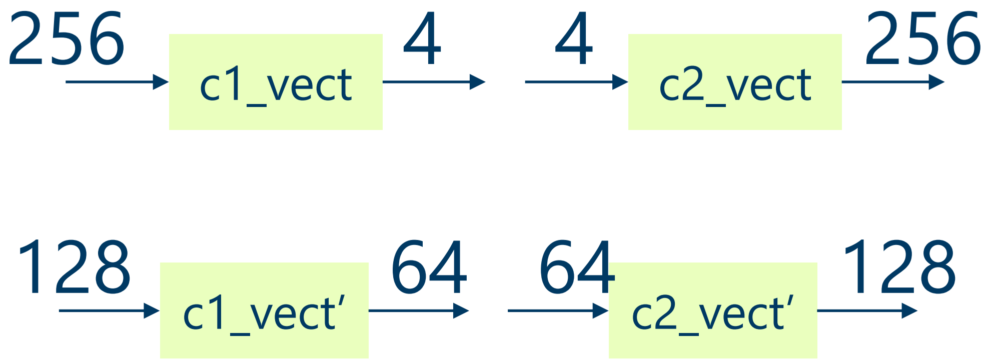
- In the end we may have very different vectorizations



- Which one to choose? Intuition: prefer fat pipelines
- Failed idea: maximize sum of pipeline arrays
- Alas it does not give <u>uniformly fat pipelines</u>: 256+4+256 > 128+64+128

How to choose final vectorization?

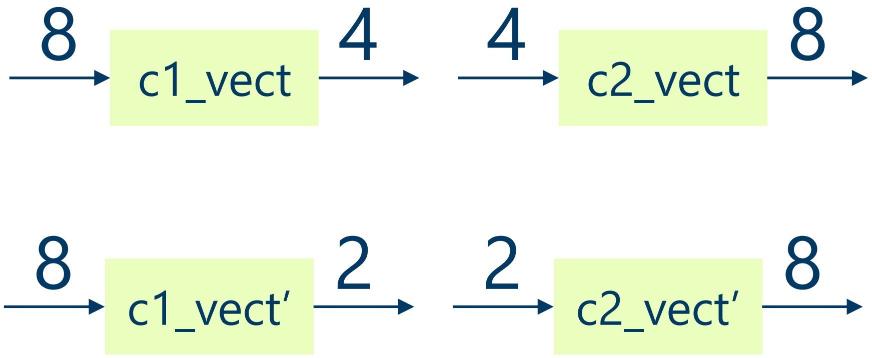
- Solution: From paper of Kelly et al. on *distributed optimization*



- Idea: maximize sum of a convex function (e.g. log) of sizes of pipeline arrays
- $\log 256+\log 4+\log 256 = 8+2+8 = 18 < 20 = 7+6+7 = \log 128+\log 64+\log 128$
- Sum of **log(.)** gives uniformly fat pipelines and can be computed **locally**

Final piece of the puzzle: pruning

- As we build feasible sets from the bottom up we ***must not discard vectorizations***
- But there may be multiple vectorizations with the same type, e.g:



- Which one to choose? [They have ***same type (ST x (arr[8] bit) (arr[8] bit)***]
- We must prune by choosing <u>one per type</u> to avoid search space explosion
- Answer: keep the one with maximum utility from previous slide

Vectorizing the Wifi TX

```
do { hInfo ← 8-{emitHeader_VECT(())}-8 >>>
             8-{scrambler_VECT (())}-8 >>>
             8-{encode12_VECT (())}-8 >>>
             8-{interleaver_bpsk_V(())}-8 >>>
             8-{modulate_bpsk_VECT (())}-8 >>>
             8-{map_ofdm_VECT (())}-64 >>>
             64-{tIFFT_VECT (())}-160;
     8-{scrambler_VECT (())}-8 >>>
     8-{encode12_VECT (())}-8 >>>
     8-{interleaver_qpsk_VECT (())}-8 >>>
     8-{modulate_qpsk_VECT (())}-4 >>>
     4-{map_ofdm_VECT (())}-64 >>>
     64-{tIFFT_VECT (())}-160
```

Vectorization and LUT synergy

```
let comp scrambler() =
  var scrmbl_st: arr[7] bit :=
          {'1,'1,'1,'1,'1,'1,'1};
  var tmp,y: bit;

  repeat {
      (x:bit) <- take;
      do {
        tmp := (scrmbl_st[3] ^ scrmbl_st[0]);
        scrmbl_st[0:5] := scrmbl_st[1:6];
        scrmbl_st[6] := tmp;
        y := x ^ tmp
      };

      emit (y)
  }
```

RESULT: ~ 1Gbps scrambler

Vectorization and LUT synergy

```
let comp scrambler() =
  var scrmbl_st: arr[7] bit :=
          {'1,'1,'1,'1,'1,'1,'1};
  var tmp,y: bit;

  repeat {
      (x:bit) <- take;
      do {
        tmp := (scrmbl_st[3] ^ scrmbl_st[0]);
        scrmbl_st[0:5] := scrmbl_st[1:6];
        scrmbl_st[6] := tmp;
        y := x ^ tmp
      };

      emit (y)
  }
```

Vectorization

```
let comp v_scrambler () =
  var scrmbl_st: arr[7] bit :=
          {'1,'1,'1,'1,'1,'1,'1};
  var tmp,y: bit;

  var vect_ya_26: arr[8] bit;
  let auto_map_71(vect_xa_25: arr[8] bit) =
    LUT for vect_j_28 in 0, 8 {
          vect_ya_26[vect_j_28] :=
             tmp := scrmbl_st[3]^scrmbl_st[0];
             scrmbl_st[0:+6] := scrmbl_st[1:+6];
             scrmbl_st[6] := tmp;
             y := vect_xa_25[0*8+vect_j_28]^tmp;
             return y
        };
        return vect_ya_26
  in map auto_map_71
```

RESULT: ~ 1Gbps scrambler

Vectorization and LUT synergy

```
let comp scrambler() =
  var scrmbl_st: arr[7] bit :=
         {'1,'1,'1,'1,'1,'1,'1};
  var tmp,y: bit;

  repeat {
      (x:bit) <- take;
      do {
        tmp := (scrmbl_st[3] ^ scrmbl_st[0]);
        scrmbl_st[0:5] := scrmbl_st[1:6];
        scrmbl_st[6] := tmp;
        y := x ^ tmp
      };

      emit (y)
  }
```

Vectorization

```
let comp v_scrambler () =
  var scrmbl_st: arr[7] bit :=
         {'1,'1,'1,'1,'1,'1,'1};
  var tmp,y: bit;

  var vect_ya_26: arr[8] bit;
  let auto_map_71(vect_xa_25: arr[8] bit) =
    LUT for vect_j_28 in 0, 8 {
          vect_ya_26[vect_j_28] :=
             tmp := scrmbl_st[3]^scrmbl_st[0];
             scrmbl_st[0:+6] := scrmbl_st[1:+6];
             scrmbl_st[6] := tmp;
             y := vect_xa_25[0*8+vect_j_28]^tmp;
             return y
        };
        return vect_ya_26
  in map auto_map_71
```

Automatic lookup-table-compilation
Input-vars = scrmbl_st, vect_xa_25 = 15 bits
Output-vars = vect_ya_26, scrmbl_st = 2 bytes
IDEA: precompile to LUT of 2^15 * 2 = 64K

RESULT: ~ 1Gbps scrambler

Conclusions and current work

- Similar correctness issues as in vectorization appear in pipeline parallelization. Currently in the workings
- Exploring process calculus semantics to help prove optimizations correct (or discover bugs ☺). For a long time our canonical semantics was the CPU execution model but that choice WAS JUST WRONG (too low-level)
- Ask me to see code, more optimizations, detailed evaluation of the optimizations and end-to-end performance numbers on our WiFi TX/RX implementation

Thanks!

www.github.com/dimitriv/Ziria