Compiling Low-level Radio Protocols

Geoffrey Mainland

IFIP WG 2.11

DREXEL UNIVERSITY
College of
Computing &
Informatics

Motivation

- Lots of innovation in PHY/MAC design.
- Modern wireless PHYs require fast DSP.
- Easy to program? fast? portable?
  - GnuRadio: easy to program, but slow.
  - SORA, Warp: fast, but difficult to program, and code is non-portable.
- We want all three!

Problems for Implementors

- CPU platforms (SORA)
  - Vectorization, CPU placement, cache use.
- FPGA platforms (Warp, Zynq)
  - Latency-sensitive design, difficult for new students/ researchers to get started.
- Portability/readability
  - Manually highly-optimized code is difficult to read/ maintain/modify.
  - Impossible to target another platform.

What makes wireless special?

- Large degree of separation between *data* and *control*.
  - Makes providing the right abstractions challenging.
- Absolutely requires low-latency stream processing.
  - Makes (efficient) compilation challenging.

Research Goals

- Language Properties:
  - Easy to read/write (high-level).
  - Easy to reason about (useful as a specification language).
  - General domain-specific abstractions (makes portability possible).
- Implementation Properties:
  - Fast!
  - Multiple back-ends (makes portability a reality).

Ziria

- Wireless code written in a high-level language.
- Compiler deals with low-level code optimization.
- Provides language abstractions that are intuitive, expressive, and appropriate for the domain.
- Implements efficient compilation scheme(s).
- Original implementation was joint work with Gordon Stewart, Mahanth Gowda, Dimitrios Vytiniotis, and Bozidar Radunovic.
- Competitive with Sora, hand-written C++ 802.11 stack.

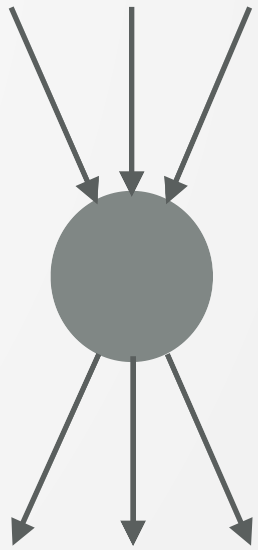
Ziria: A Two-layer Design

- Lower-level
  - Imperative C-like code for manipulating bits, bytes, arrays, etc.
- Higher-level
  - Monadic language for specifying and composing stream processors.
  - Enforces clean separation between control and data flow.
- Monadic stream language enables aggressive compiler optimizations.

Existing Abstractions

- Predominant abstraction is a dataflow graph where processing occurs at the vertices (GnuRadio, SORA, StreamIt).
- A reasonable *execution* model, but not a great *programming* model.

Events (messages) in

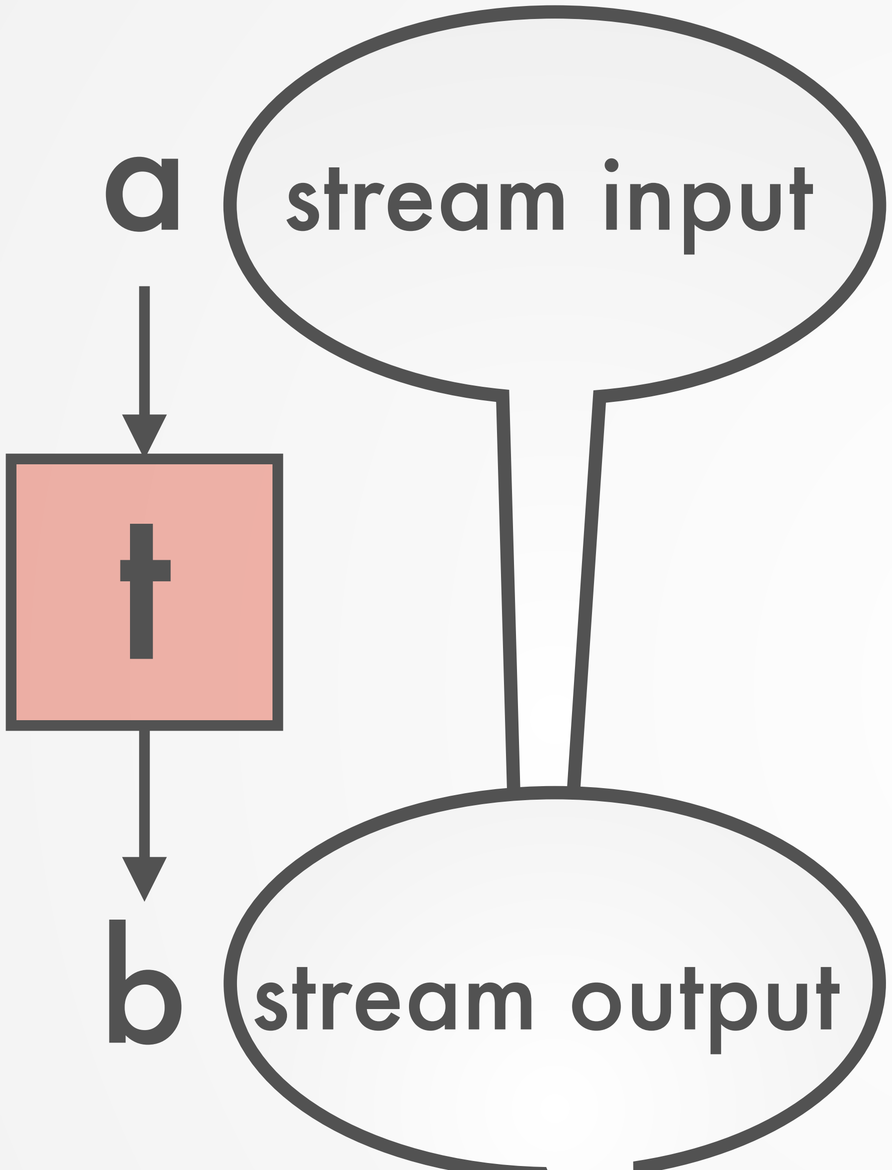


Events (messages) out

Why are data flow graphs unsatisfactory?

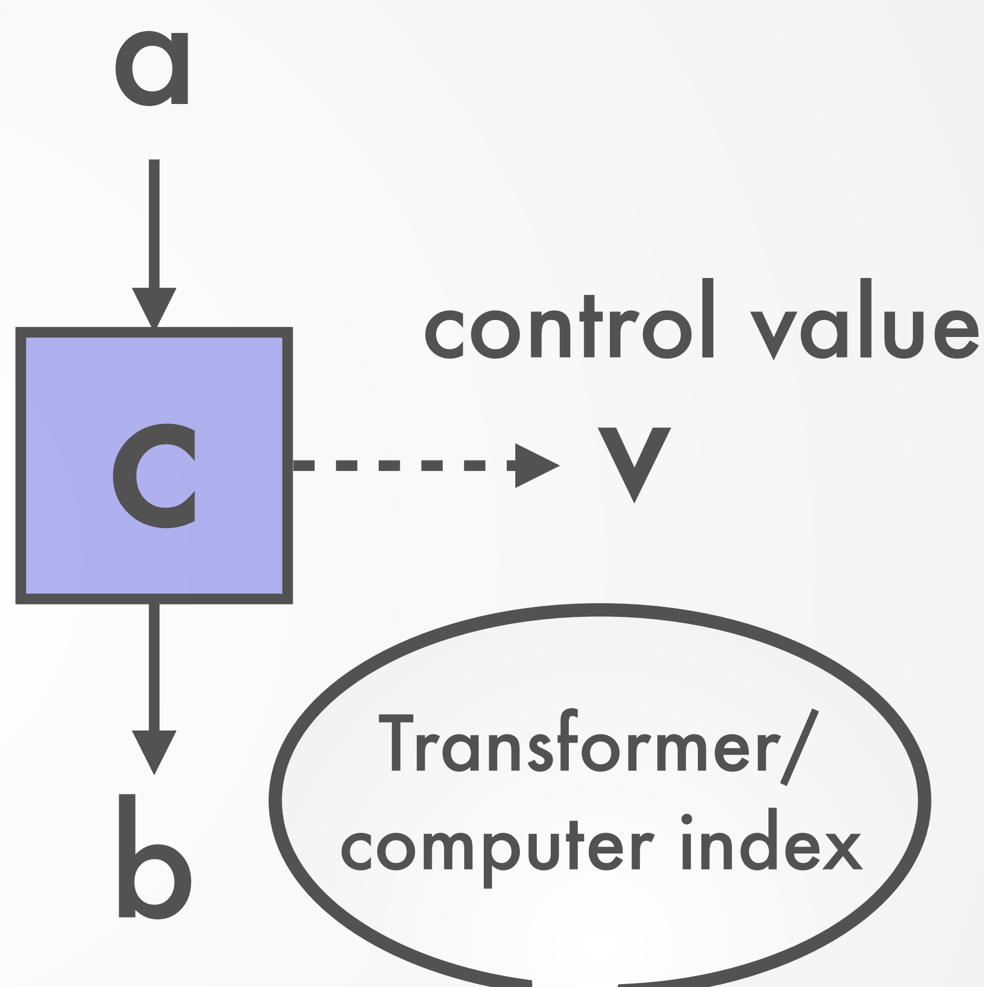
- When is vertex state initialized?
- How can "control" messages change a vertex's behavior?
- How can a vertex send a "control" message to another vertex, perhaps one to which it is not immediately connected?
- How can we jointly optimize interacting vertices' operations?

Control-Aware Stream Abstractions



Stream *transformer* t of type:

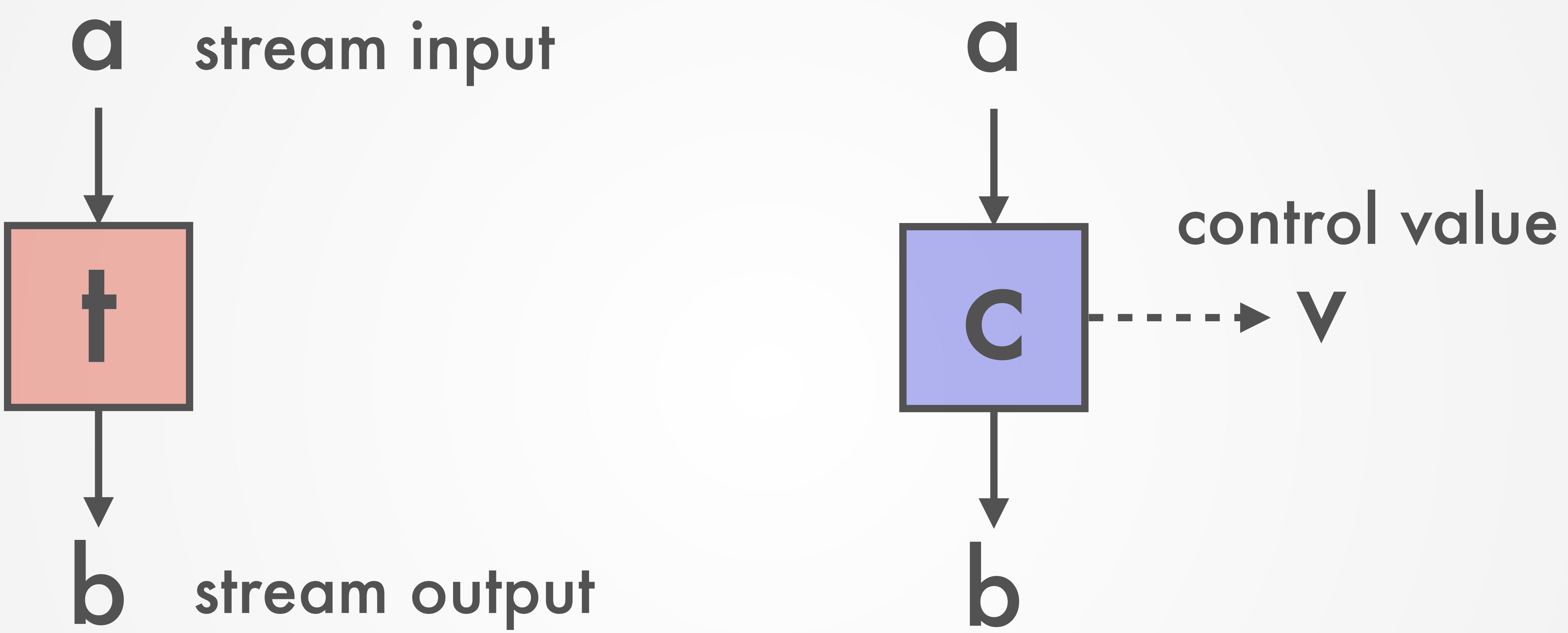
`ST T a b`



Stream *computer* c of type:

`ST (C v) a b`

Control-Aware Stream Abstractions




```
map    :: (a→b) → ST T a b
repeat :: ST (C ()) a b → ST T a b
```

```
take :: ST (C a) a b
emit :: b → ST (C ()) a b
```

“Horizontal” and “Vertical” Composition

```
(>>=)  :: ST (C v) a b → (v → ST ω a b)→ ST ω a b
return :: v →  ST (C v) a b
```

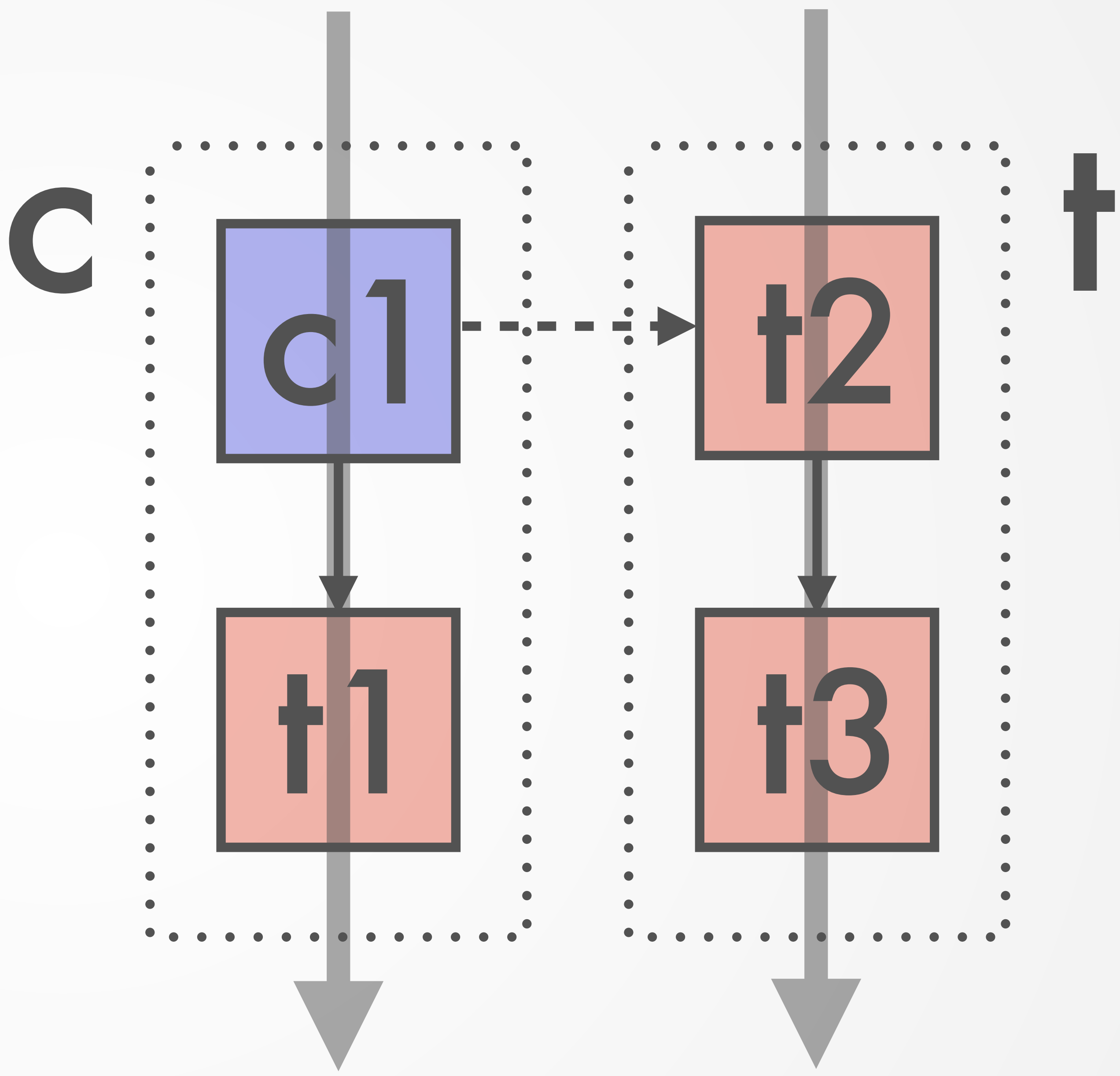
Composition along control path (like a monad)

Composition along data path (like an arrow)

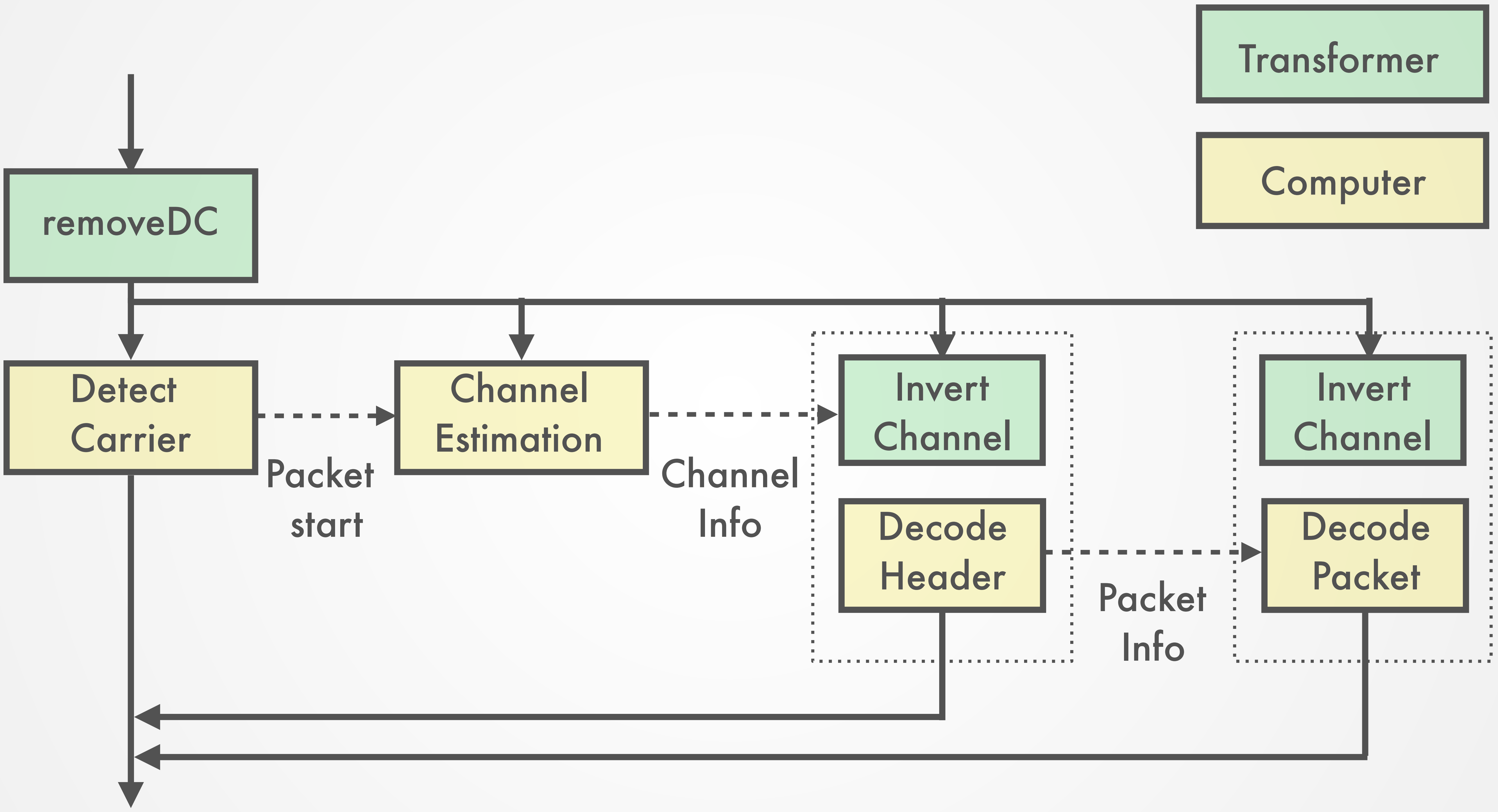
```
(>>>) :: ST T a b     → ST T b c     → ST T a c
(>>>) :: ST (C v) a b → ST T b c     → ST (C v) a c
(>>>) :: ST T a b     → ST (C v) b c → ST (C v) a c
```

Creating a Pipeline

```
{ v ← (c1 >>> t1)
; t2 >>> t3
}
```



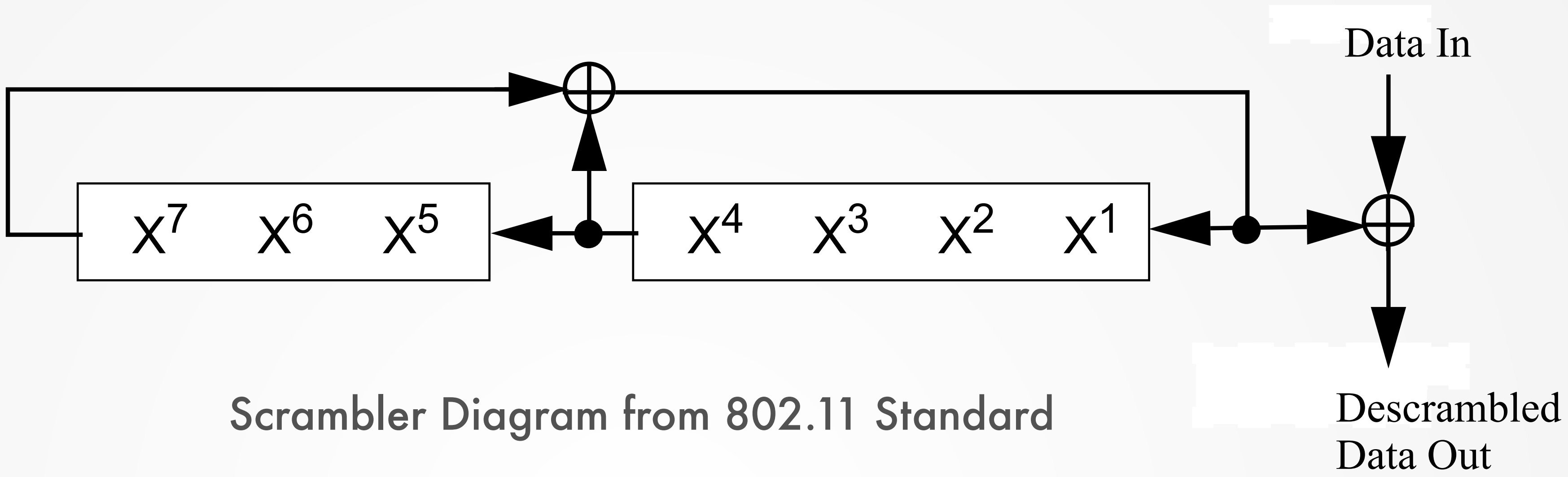
WiFi Receiver (simplified)



New Language/Compiler

- Uniform surface language (requires new types).
- Pure, monadic core (intermediate) language.
- Well-defined semantics for core language (PLT Redex).
- Compiler is a series of transformations on the typed intermediate language.
- More recently: generics and type-level Nat.

Example: Scrambler



Scrambler Diagram from 802.11 Standard

```
fun scrambler() {
  let mut tmp : bit;

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

Observations about Scrambler

```
fun scrambler() {
  let mut tmp : bit;

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

- Executable specification.
- Not very efficient to operate one bit at a time.
- If we could make the scrambler operate a byte at a time, we could convert it to a lookup table.

Let’s Fix the Scramble

Array of 8 bits

```
repeat {
  xs <- take;
  for i in 0..8
    emit xs[i];
}
>>>
scrambler()
>>>
{
  let mut xs : [bit;8]

  repeat {
    for i in 0..8 {
      x <- take;
      xs[i] = x;
    }
    emit xs;
  }
}
```

- Now the pipeline reads and writes bytes!
- If only we could somehow fuse these computations together...
- We can, with the *fusion transformation*.

Fusion

```
repeat { x ← take; emit f(x) } >>>
repeat { x ← take; emit g(x) }
```

Can be fused to:

```
repeat { x ← take; emit g(f(x)) }
```

- The original Ziria compiler went to great lengths to perform “auto-mapping.”
- Our fusion transformation can fuse much more, including repeat loops and for loops with known bounds.
- Fusion is “just” an abstract interpretation of the operational semantics.

Putting it All Together

```
repeat {
  xs <- take;
  for i in 0..8
    emit xs[i];
}
>>>
scrambler()
>>>
{
  let mut xs : [bit;8]

  repeat {
    for i in 0..8 {
      x <- take;
      xs[i] = x;
    }
    emit xs;
  }
}
```

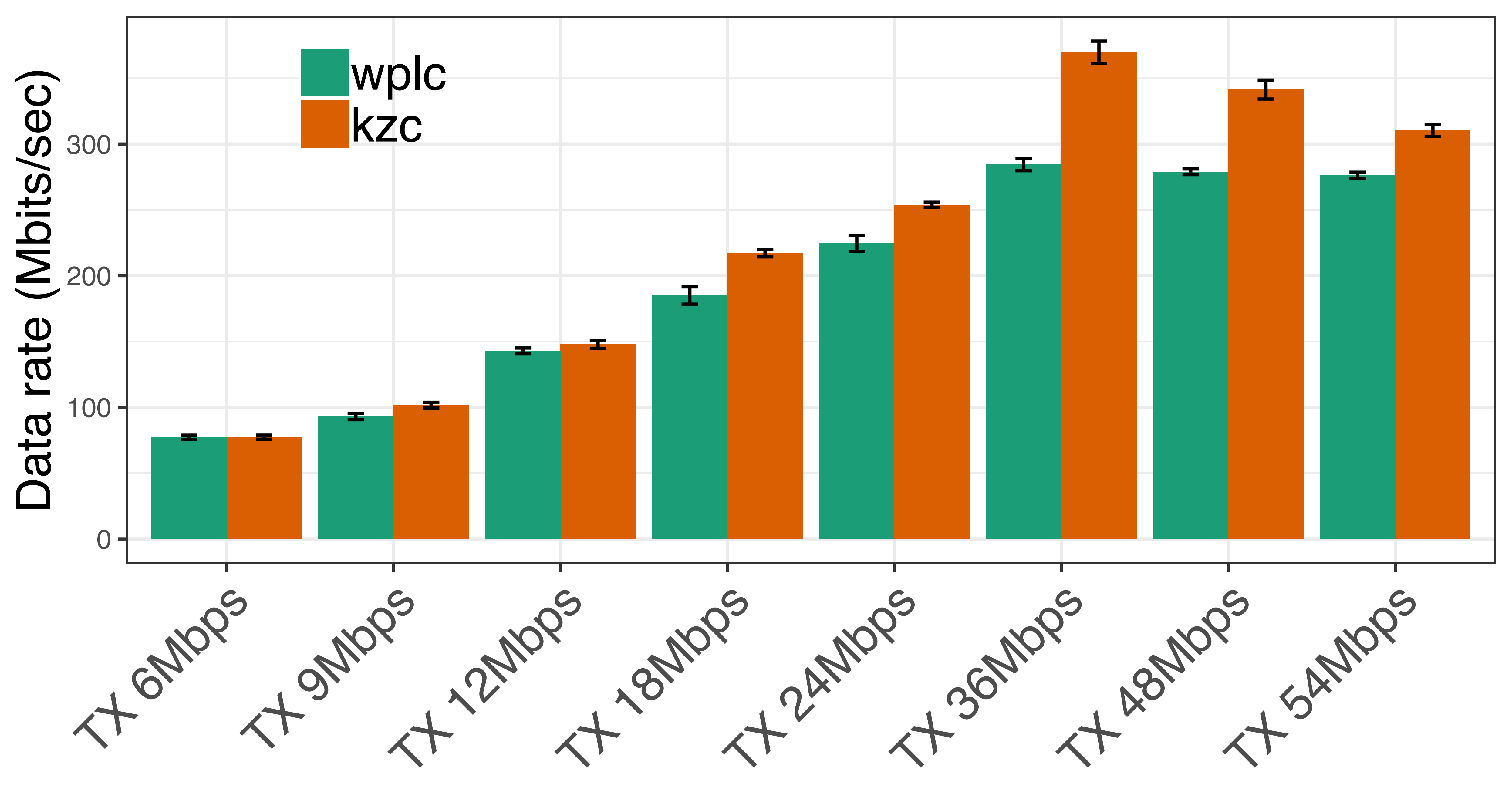
Now that we have fusion, how do we know where to place coercions like these?

Scrambler in the 6Mbps Pipeline

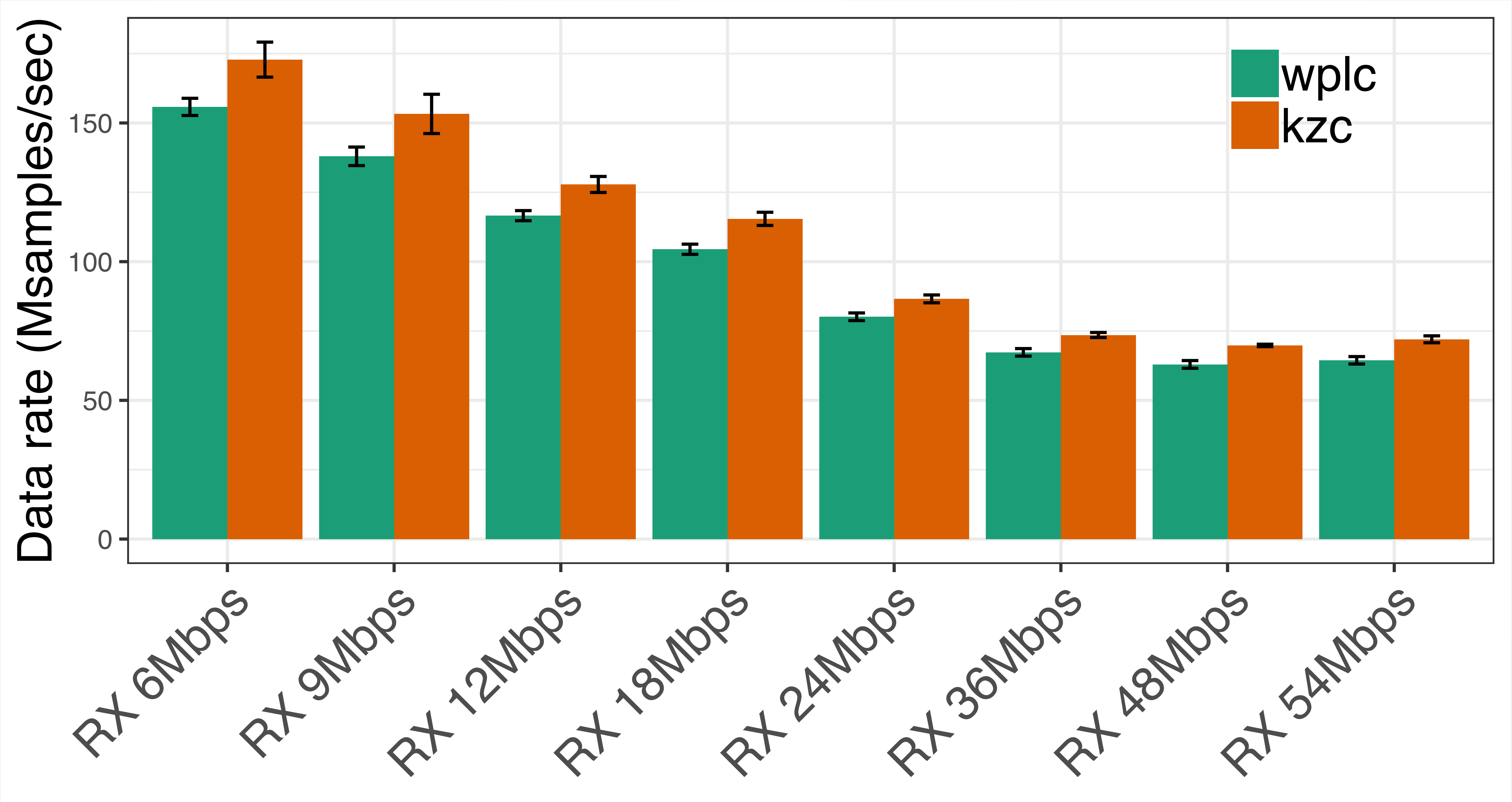
```
crc24(len, true) >>>                    [8,24]
scrambler(default_scrmbl_st) >>>        [1,1]
encode12() >>>                          [1,2]
interleaver_bpsk() >>>                  [48,48]
modulate_bpsk()                         [1,1]
```

- The compiler performs *rate analysis* to figure out the input/output “shape” of individual components. Previous compiler required annotations.
- The *pipeline coalescing* transformation inserts coercions to widen the pipeline, as with the scrambler on the previous slide, and to perform “impedance matching.”
- Finally, *fusion* eliminates >>>.

TX Performance



RX Performance



Where's the Magic?

1. The language
   - First-order (essentially).
   - `take` and `emit` are built-in to the language. Statically-known read/write sizes.
   - **No `zip` or `unzip`**. In contrast to, e.g., Lustre.
2. The application
   - No data dependencies once we know the data rate.
   - Constant loop bounds.

Continuing Challenges

- Provide better abstractions for signal processing operations (HSpiral, with Jeremy Johnson).
- Automate conversion of floating-point code to fixed-point code while maintaining "correctness" (Xiao Han).
- Compile to FPGAs (Mahshid Shahmohammadian).

Compiling to Hardware: Challenges

```
*   atan   … := conj(…*x, …*y)
```

- “Atomic” operations now have (space) costs we have to take into consideration.
- Compound operations now have (time) costs we have to take into consideration.
- Longest-latency operation now gates operating frequency.

Compiling to Hardware: Hope

- Simple consumer/producer model *matches* hardware model pretty well.
- ANF (already used in IR) leads to simple “instruction”-level “fission.”
- But when to fuse? For example, we still want to convert scramble to a LUT.