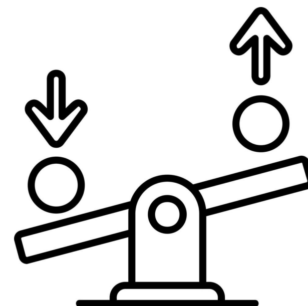
For LATTE'26

Jingtao Xia[1], Ayana Alemayehu[2], Sijie Kong[1], Ben Hardekopf[1], Rachit Nigam[2], Jonathan Balkind[1]

[1]UC Santa Barbara · [2]MIT



RTL

- precise timing
- explicit resources
- ❌ manual, tedious

HLS

- automated scheduling
- productive
- ❌ unpredictable, opaque

Designers must choose, or compromise

Can we **unify** them?

Timeline types ⇔ SDC scheduling constraints

- Timeline types encode when values are available
  - static guarantees of timing correctness

```
m := new Mult[32]<'G>( _ , _ );
// multiplier delay = 2
mx := new Mux[32]<'G+2>(op, …, m.out);
```

- HLS schedulers solve difference constraints (SDC)

T_a - T_b >= Delay(op)

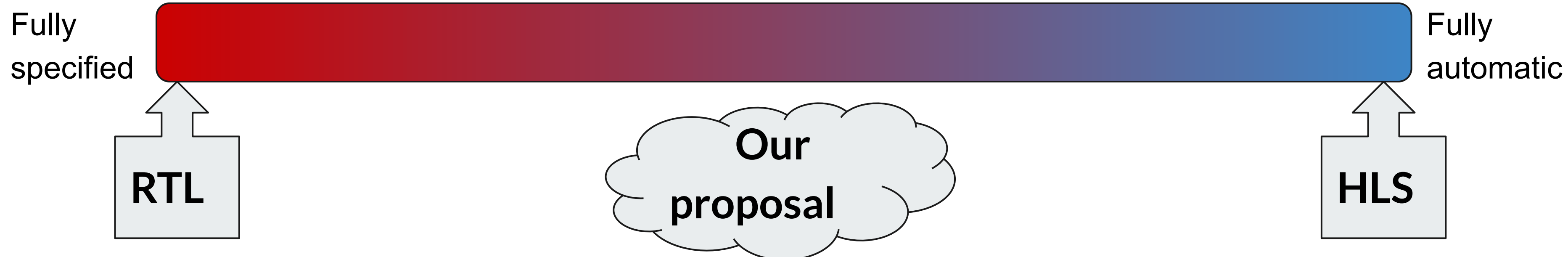
Same underlying structure

Language Constructs for HLS

- **Allocation**: pool M: Mult[32] * 3
- **Binding**: M[2]
- **Scheduling**: M[2]<’G+1>

Question Marks: Make constraints partially specified

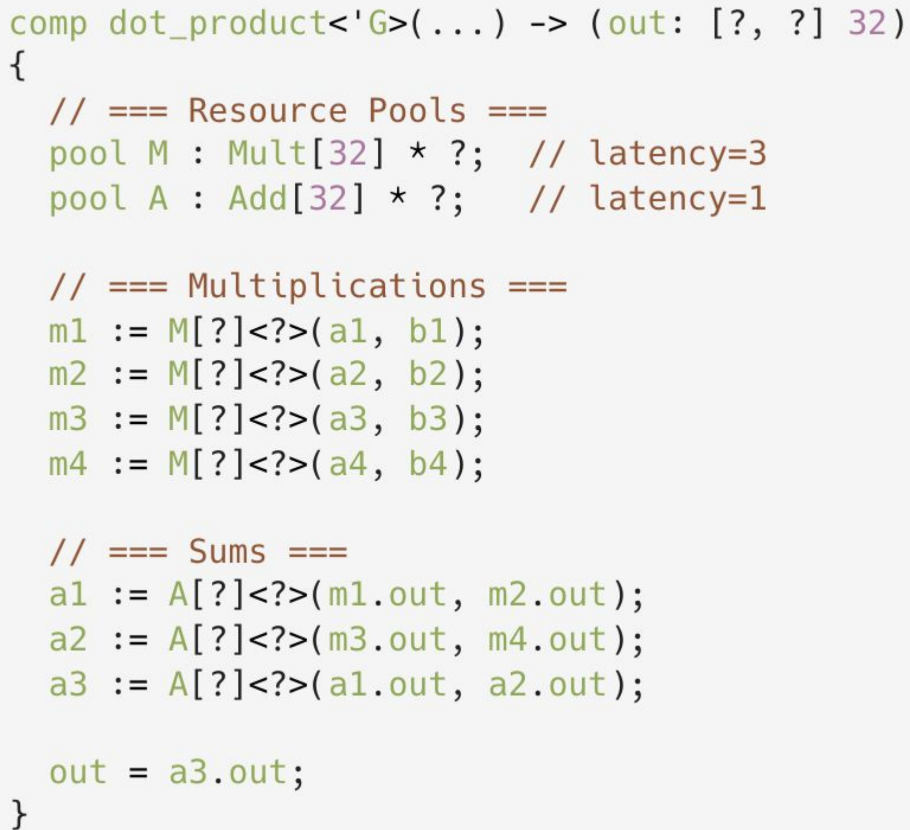
- **Allocation**: pool M: Mult[32] * 3→ pool M: Mult[32] * ?
- **Binding**: M[2]→ M[?]
- **Scheduling**: A<'G+1>→ A<?>



Example

```
fn dot_product(...) -> i32 {
    return (a1*b1 + a2*b2) + (a3*b3 + a4*b4);
}
```

- Starting point: HLS-style program



```
comp dot_product<'G>(...) -> (out: [?, ?] 32)
{
  // === Resource Pools ===
  pool M : Mult[32] * ?;  // latency=3
  pool A : Add[32] * ?;   // latency=1

  // === Multiplications ===
  m1 := M[?]<?>(a1, b1);
  m2 := M[?]<?>(a2, b2);
  m3 := M[?]<?>(a3, b3);
  m4 := M[?]<?>(a4, b4);

  // === Sums ===
  a1 := A[?]<?>(m1.out, m2.out);
  a2 := A[?]<?>(m3.out, m4.out);
  a3 := A[?]<?>(a1.out, a2.out);

  out = a3.out;
}
```

- Lower with full question marks

```
comp dot_product<'G>(...) -> (out: [?, ?] 32)
{
  // === Resource Pools ===
  pool M : Mult[32] * 2;   // latency=3, 2 multipliers
  pool A : Add[32] * 2;    // latency=1, 2 adders

  // === Multiplications ===
  m1 := M[?]<?>(a1, b1);
  m2 := M[?]<?>(a2, b2);
  m3 := M[?]<?>(a3, b3);
  m4 := M[?]<?>(a4, b4);

  // === Sums ===
  a1 := A[?]<?>(m1.out, m2.out);
  a2 := A[?]<?>(m3.out, m4.out);
  a3 := A[?]<?>(a1.out, a2.out);

  out = a3.out;
}
```

- User refine the design by extra constraints on allocation/binding/scheduling

```
comp dot_product<'G>(...) -> (out: [?, ?] 32)
{
  // === Resource Pools ===
  pool M : Mult[32] * 2;  // latency=3, 2 multipliers
  pool A : Add[32] * 2;   // latency=1, 2 adders

  // === Multiplications ===
  m1 := M[?]<?>(a1, b1);
  m2 := M[?]<?>(a2, b2);
  m3 := M[?]<?>(a3, b3);
  m4 := M[?]<'G+3>(a4, b4);       // hardcoded timing

  // === Sums ===
  a1 := A[0]<?>(m1.out, m2.out);        // hardcoded binding
  a2 := A[?]<?>(m3.out, m4.out);
  a3 := A[?]<?>(a1.out, a2.out);

  out = a3.out;
}
```

- User refine the design by extra constraints on allocation/binding/scheduling

```
comp dot_product<'G>(...) -> (out: [?, ?] 32)
{
  // === Resource Pools ===
  pool M : Mult[32] * 2;  // latency=3, 2 multipliers
  pool A : Add[32] * 2;   // latency=1, 2 adders

  // === Multiplications ===
  m1 := M[?u1]<?t_mul>(a1, b1);      // named binding + named timing
  m2 := M[?u1]<?t_mul+3>(a2, b2);    // same unit, relative timing
  m3 := M[?u2]<?>(a3, b3);
  m4 := M[?u2]<'G+3>(a4, b4);        // hardcoded timing

  // === Sums ===
  a1 := A[0]<?>(m1.out, m2.out);           // hardcoded binding
  a2 := A[?]<?>(m3.out, m4.out);
  a3 := A[?]<?>(a1.out, a2.out);

  out = a3.out;
}
```

- User refine the design by extra constraints on allocation/binding/scheduling

```
comp dot_product<'G>(...) -> (out: [?, ?] 32)
{
  // === Resource Pools ===
  pool M : Mult[32] * 2;  // latency=3, 2 multipliers
  pool A : Add[32] * 2;   // latency=1, 2 adders

  // === Multiplications ===
  m1 := M[?u1]<?t_mul>(a1, b1);      // named binding + named timing
  m2 := M[?u1]<?t_mul+3>(a2, b2);    // same unit, relative timing
  m3 := M[?u2]<?>(a3, b3);
  m4 := M[?u2]<'G+3>(a4, b4);        // hardcoded timing

  // === Sums ===
  a1 := A[0]<?t_add>(m1.out, m2.out);          // hardcoded binding
  a2 := A[?]<?t_add+[..2]>(m3.out, m4.out);  // relative timing w/ range
  a3 := A[?]<?>(a1.out, a2.out);

  out = a3.out;
}
```

- User refine the design by extra constraints on allocation/binding/scheduling

```
comp dot_product<'G>(...) -> (out: ['G+8, 'G+9] 32)
{
  // === Resource Pools ===
  pool M : Mult[32] * 2;  // latency=3, 2 multipliers
  pool A : Add[32] * 2;   // latency=1, 2 adders

  // === Multiplications ===
  m1 := M[1]<'G>(a1, b1);
  m2 := M[1]<'G+3>(a2, b2);
  m3 := M[0]<'G>(a3, b3);
  m4 := M[0]<'G+3>(a4, b4);

  // === Sums ===
  a1 := A[0]<'G+6>(m1.out, m2.out);
  a2 := A[1]<'G+6>(m3.out, m4.out);
  a3 := A[0]<'G+7>(a1.out, a2.out);

  out = a3.out;
}
```

- Scheduling result

What this enables: control and automation

- Smooth design space (not two paradigms)
- First-class scheduling (not pragmas)
- Compositional timing guarantees

Our Unified
Framework
HLS
RTL
imgflip.com

1. Language level

- Physical components vs abstract values and dependencies

2. Solver design

- Unified vs decomposed binding & scheduling?
- new constraint model → new algorithms

3. Usability

- how much control do designers want?