Representing Dataflow with MLIR

Pedro Ciambra
pedro.ciambra@ic.unicamp.br

Advisors:

Hervé Yviquel (UNICAMP/IC/LSC)
Maxime Pelcat (INSA Rennes/IETR/Vaader)
Mickaël Dardaillon (INSA Rennes/IETR/Vaader)

Introduction

- Two part presentation
- 1: An overview of the work I did on a Dataflow compiler for my Master's degree at UNICAMP.
- 2: Focus on MLIR dialect:
  - needs & difficulties
  - solutions
  - discussion
  - a proposal
- I'll try to be quick to give time for discussion afterwards.

Part 1

IaRa SDF Compiler

- Motivation
- Dataflow basics
- My compiler
- Results

Dataflow ecosystem

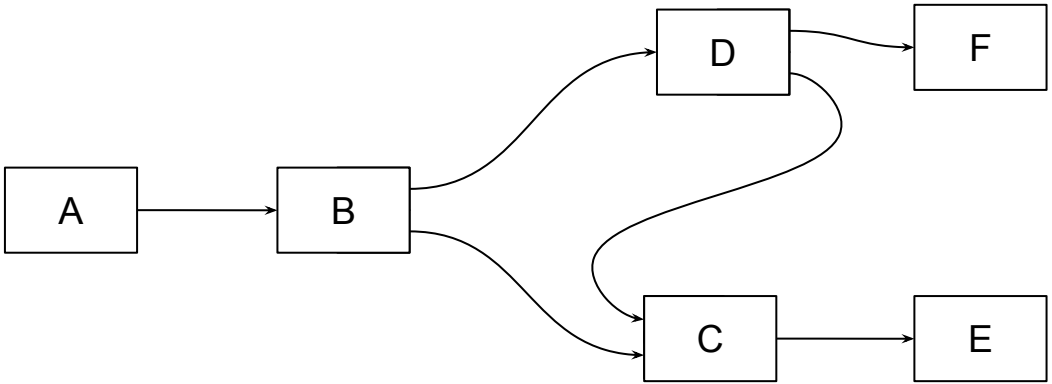
- Dataflow programming is a very mature research subject ('60s)
- It has been a successful approach for a diverse range of applications
  - Real-time systems, reactive systems, heterogenous networks, HSL, etc...
- Many different MoCs with different abstractions and constraints
- Solutions are generally built with specific application/architecture/MoC/philosophy in mind
- Not much focus on intercompatibility between tools, even if there is a good amount of overlap in the abstractions and techniques used.
- MLIR for the rescue

Motivation

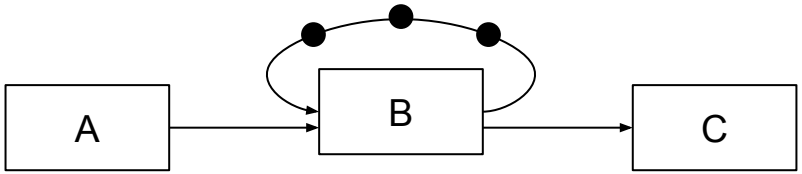
- Enable Dataflow researchers to more easily access the state of the art in compilers
- Facilitate collaboration and integration of projects
- Make the advantages of Dataflow more accessible to the compilation community
- Enable reuse of components such as actor implementations, schedulers, optimizations and runtimes
- **Problem:** what are the requirements for a common dataflow format that is useful within MLIR's constraints?

Dataflow programming paradigm

- Models the algorithm as a directed graph
- Vertices (actors) are computation units that act on tokens of data
- Edges are FIFO data structures
- Parallelizable operations correspond to parallel paths in the graph
- Data dependencies are explicit
- Easier to analyse when scheduling for multicore / generating HSL

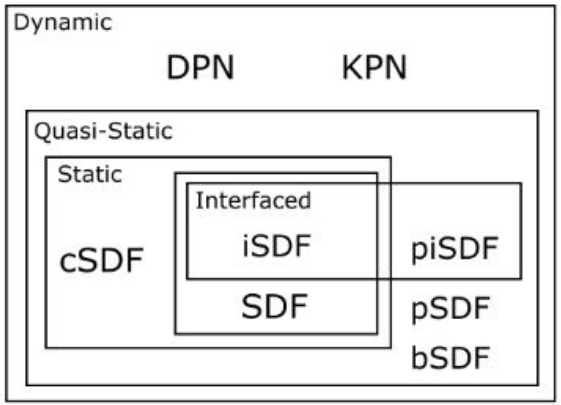


Cycles and delays



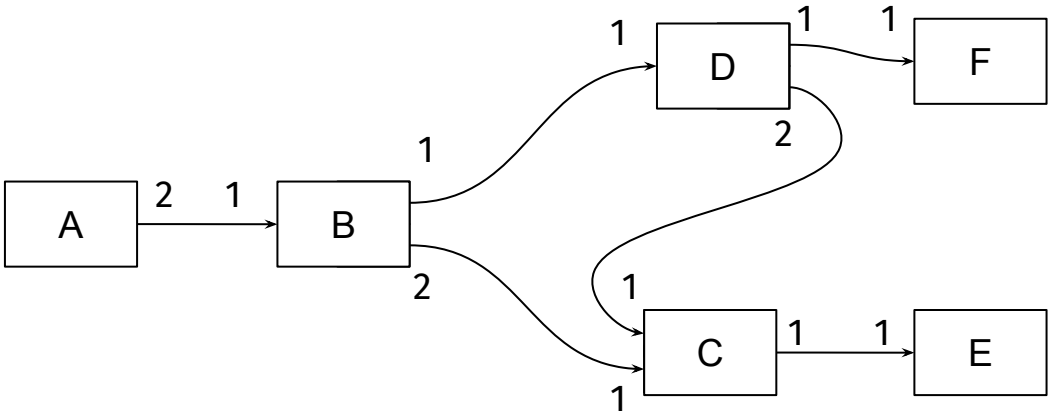
Dataflow Models of Computation

- There are many Dataflow MoCs, with different levels of strictness
  - Each level relaxes a constraint
  - Constrained models are easier to schedule at compile time
  - Unconstrained models are more expressive and easier to use



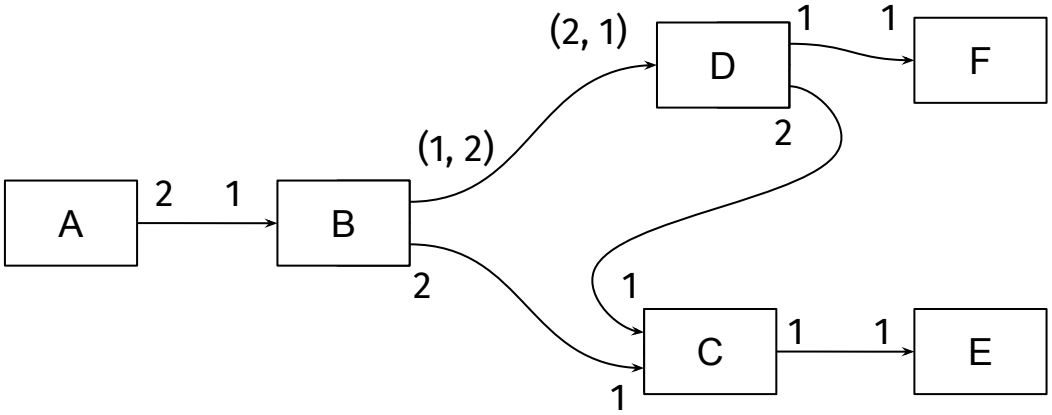
Synchronous Dataflow (SDF)

- Data rates are constrained to a constant number.
- Schedule can be generated statically, with no runtime cost
- Can detect inadmissible graphs at compile time

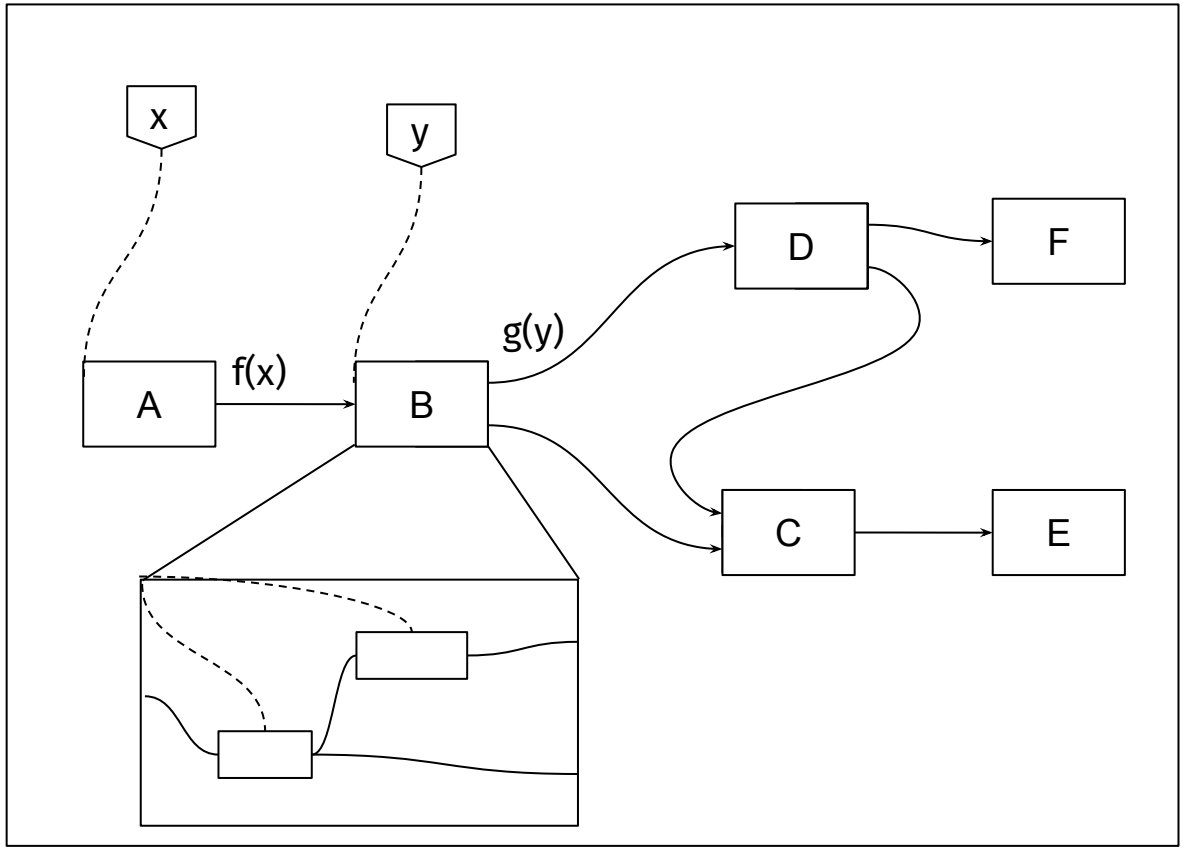


Cyclo-static Dataflow (CSDF)

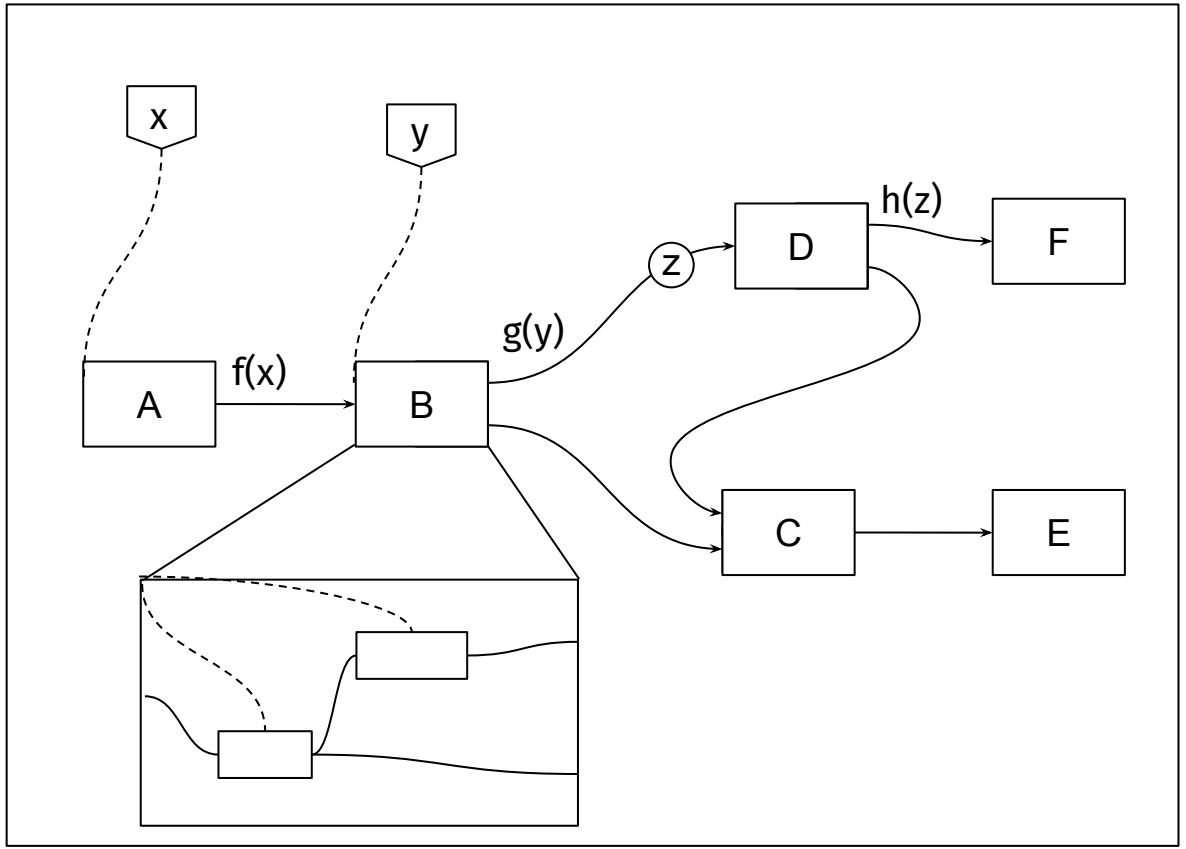
- Data rates are cyclic and predictable
- Still doesn't depend on runtime data



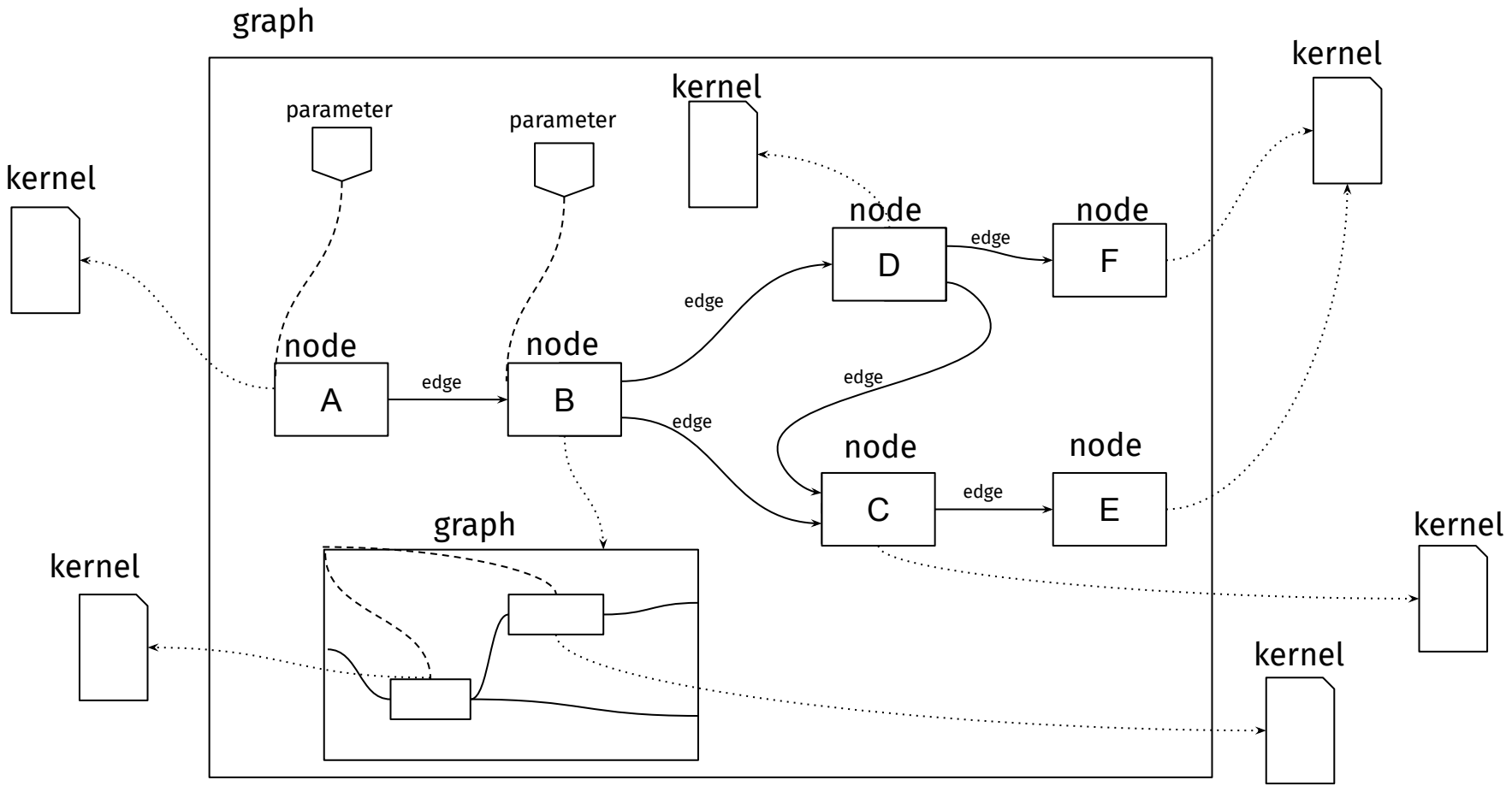
Parameterized/Interfaced (piSDF)



Dynamic MoCs (DPN, KPN)

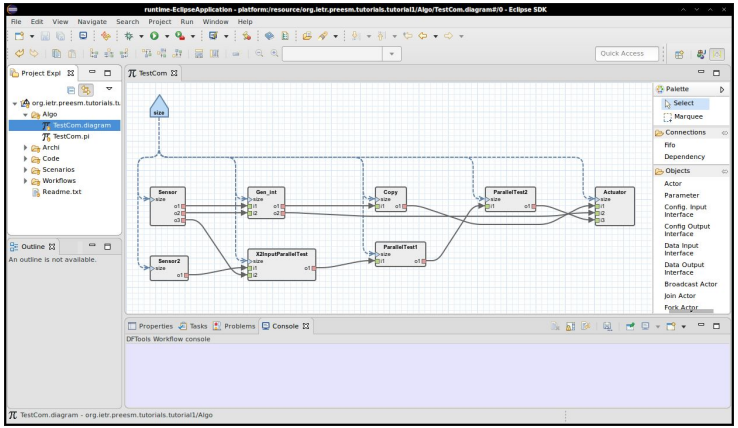


Terminology



Related works

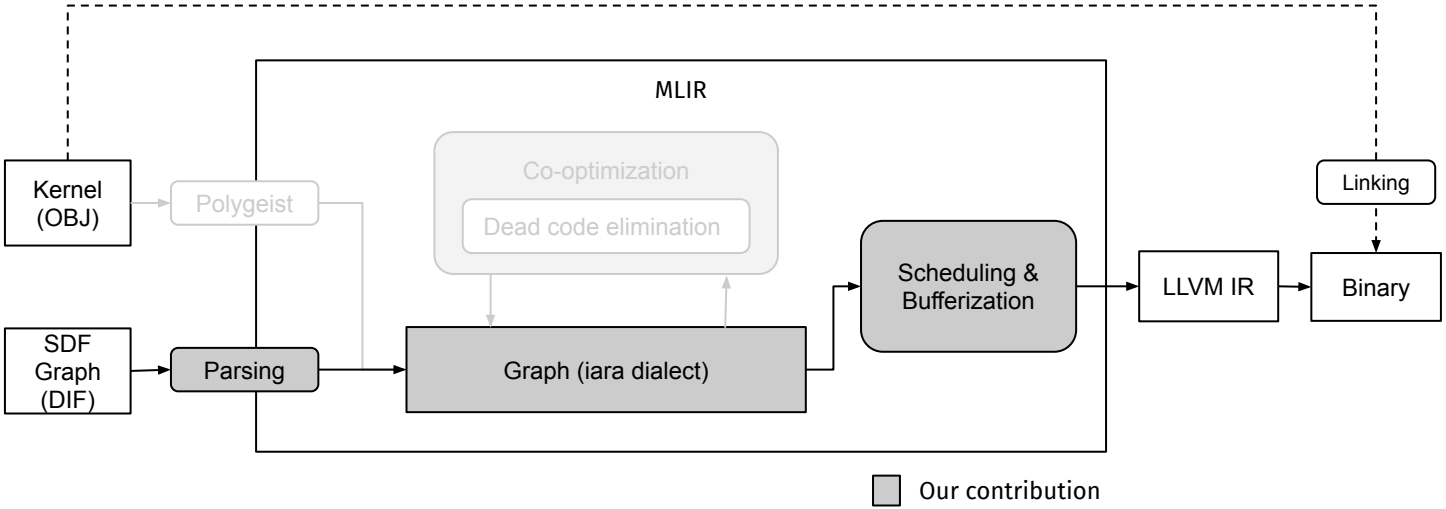
- Preesm is a dataflow IDE
  - Support for static and quasi-static MoC
  - Generates C code for multicore
  - Has a graphical interface for graph design
  - Has state-of-the-art optimizations
  - Stores graph in declarative XML format



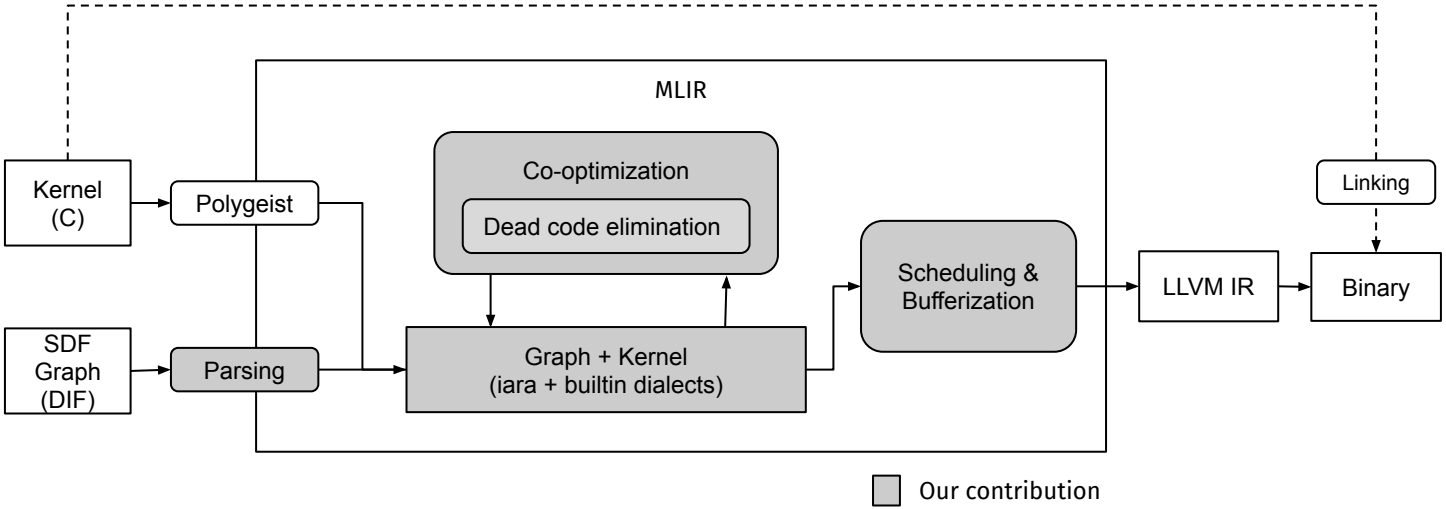
- Dataflow Interchange Format (DIF)
  - Part of a Java-based library
  - Simple, human-readable grammar
  - No assumptions about the MoC
  - Custom "attributes"

```
dif graph1_4 {
        topology {
        nodes = n1, n2, n3, n4;
        edges = e1 (n1, n2),
                e2 (n2, n1),
                e3 (n1, n3),
                e4 (n1, n3),
                e5 (n4, n3),
                e6 (n4, n4);
        }
        interface {
                inputs = p1, p2:n2;
                outputs = p3:n3, p4:n4;
        }
        parameter {
                param1;
                param2 = 1;
        ...
```

The IaRa compiler

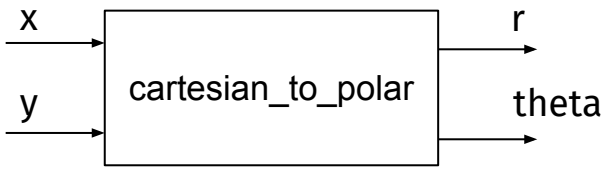


Exploiting MLIR



Dead code elimination for unused outputs

- Automatically remove operations that access only unused outputs
- Allows actor reuse and saves memory and cycles




```
void cartesian_to_polar(
  float *x, float *y, // input ports
  float *r, float *theta) // output ports {
  *r = sqrt(*x * *x + *y * *y);
  *theta = atan2f(*y, *x);
}
```

```
func @cartesian_to_polar(
    %arg0: llvm.ptr<f32>, %arg1: llvm.ptr<f32>,
    %arg2: llvm.ptr<f32>, %arg3: llvm.ptr<f32>){
  %1 = llvm.load %arg0 : !llvm.ptr<f32>
  %2 = arith.mulf %1, %1 : f32
  %3 = llvm.load %arg1 : !llvm.ptr<f32>
  %4 = arith.mulf %3, %3 : f32
  %5 = arith.addf %2, %4 : f32
  %6 = math.sqrt %5 : f32
  llvm.store %6, %arg2 : !llvm.ptr<f32>
  %9 = call @atan2(%3, %1) : (f32, f32) -> f32
  llvm.store %9, %arg3 : !llvm.ptr<f32>
  return
}
```

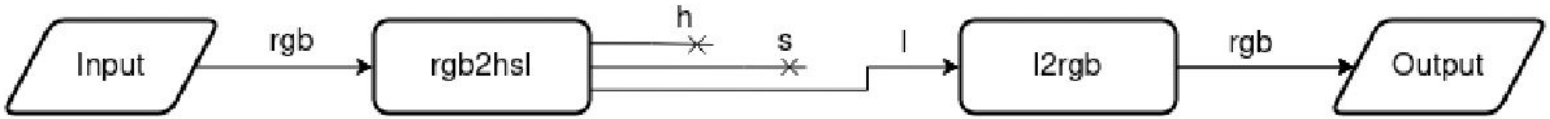
* Ciambra, Yviquel, Dardaillon, Pelcat. *Co-optimizing Dataflow Graphs and Actors with MLIR*. 2022 IEEE Workshop on Signal Processing Systems (SiPS)

Results

- Compared against existing SDF applications available in Preesm
  - Converted Preesm's XML graph format into DIF
- Compared performance with Preesm's single-core scheduler
- Got comparable results, except for very large graphs, where Preesm's memory optimizations made a big difference.

Results: RGB to Grayscale

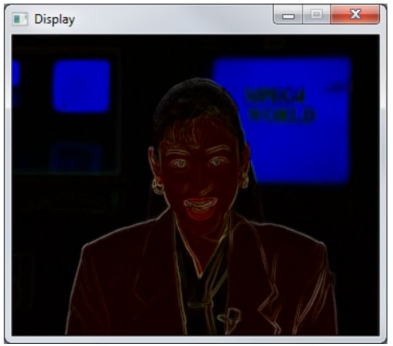
- 4 nodes, 3 edges, 4 kernels, 68 lines of code
- New application
  - Developed to show DCE
- PREESM does not support unused outputs
  - PREESM implementation writes into a scratch buffer

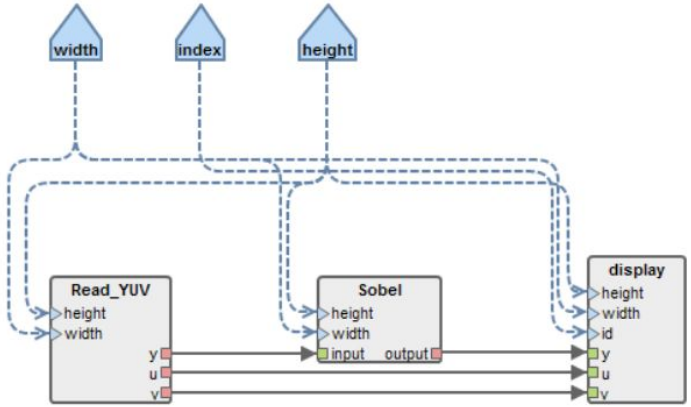


	Preesm	IaRa (No DCE)	IaRa (With DCE)
Time [s]	4.151	4.020	2.689
Max RSS [KB]	11,048	6,300	3,838
Speedup	1.0	1.03	1.54
Mem. Usage	1.0	0.56	0.34

Results: Sobel filter

- 3 nodes, 4 edges, 3 kernels, 644 lines of code
- Schedule is trivial





	Preesm	IaRa
FPS	1,398	1,402
Max RSS [KB]	33,404	33,232
Speedup	1.0	1.0
Mem. Usage	1.0	0.99

Results: Scale-invariant Feature Transform (SIFT)

- 57 nodes, 108 edges, 33 kernels, 5460 lines of code
- Several hierarchical layers, feedback loops with delays
- 4x more memory usage, 50% speed
- Many broadcast nodes, which are optimized out in PREESM.

	Preesm	IaRa
Time [s]	0.538	0.977
Max RSS [KB]	198,448	857,540
Speedup	1.0	0.55
Mem. Usage	1.0	4.32

Part 2

Dataflow dialect

- Requirements
- Approach
- Solution
- Discussion

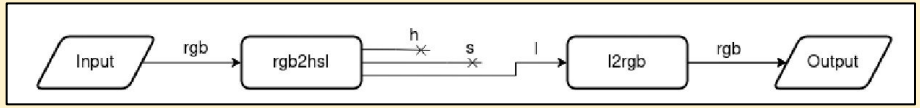
Requirements to represent a declarative Dataflow graph

- A list of actor interfaces (signatures)
  - Name, parameters, input and output ports
  - Each port has a name, type, and token rate information
- A list of nodes
  - Instances of actors, respecting signature
  - Reference to implementation (C function, link-time symbol, or sub-graph)
- A list of edges between ports
  - **Edges may have extra data, such as delays**
  - Needs **type checking** to ensure that producing and consuming ports match.
- Extensibility for different MoCs
- Notice the separation between graph and kernel
- Not very MLIR-like (no SSA to encode the graph)

First approach

- Two dialects:
  - **Declarative**: high-level topology that has same structure as the input languages
    - Everything is referred to by name (using SymbolName attributes)
    - Easy to generate directly from the AST of the input languages
    - Should remain human-readable
      - Keep the interface recognizable for debug purpuses
      - No name mangling, no parameter indexing etc
  - **SSA**: for middle-level dataflow analysis
    - Using MLIR type system and SSA semantics
    - Using MLIR's graph traversal utilities for analysis
    - Enables using the pattern matching framework for transformation

Declarative dialect




```
module {
  iara.graph @main : "dif"
   param_defaults [
        #iara.param<"width" = 640 : i32>,
        #iara.param<"height" = 480 : i32>]
   {
      iara.kernel @read_rgb_frame
         params [#iara.param<"width" = <<NULL ATTRIBUTE>>>, #iara.param<"height" = <<NULL ATTRIBUTE>>>]
         outputs [#iara.port<"rgb" : i8[921600 : i64]>] // width * height * 3
      iara.kernel @rgb_to_hsl
         params [#iara.param<"width" = <<NULL ATTRIBUTE>>>, #iara.param<"height" = <<NULL ATTRIBUTE>>>]
         inputs [#iara.port<"rgb" : i8[921600 : i64]>]
         outputs [#iara.port<"h" : f32[307200 : i64]>,  // width * height
                  #iara.port<"s" : f32[307200 : i64]>,
                  #iara.port<"l" : f32[307200 : i64]>]
      iara.kernel @l_to_rgb
         params [#iara.param<"width" = <<NULL ATTRIBUTE>>>, #iara.param<"height" = <<NULL ATTRIBUTE>>>]
         inputs [#iara.port<"l" : f32[307200 : i64]>]
         outputs [#iara.port<"rgb" : i8[921600 : i64]>]
      iara.kernel @write_rgb_frame
         params [#iara.param<"width" = <<NULL ATTRIBUTE>>>, #iara.param<"height" = <<NULL ATTRIBUTE>>>]
         inputs [#iara.port<"rgb" : i8[921600 : i64]>]

      iara.node @n2 : @rgb_to_hsl
      iara.node @n3 : @l_to_rgb
      iara.node @n4 : @write_rgb_frame
      iara.node @n1 : @read_rgb_frame

      iara.edge @e1 : @n1::"rgb" → @n2::"rgb"
      iara.edge @e2 : @n2::"l" → @n3::"l"
      iara.edge @e3 : @n3::"rgb" → @n4::"rgb" // <delays=[1,2,3 ... ]>
   }
 }
```

Kernels

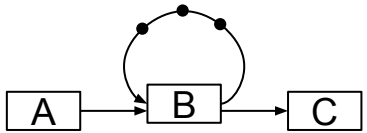
Nodes

Edges

Hypothetical middle-level "MLIR-like" dialect


```
%rgb       = iara.node ()     <sym_name="n1", impl="read_rgb_frame",  width=640, height=480> : ()→( ... );
%h, %s, %l = iara.node (%rgb) <sym_name="n2", impl="rgb_to_hsl",      width=640, height=480> : ( ... )→( ... );
%gs        = iara.node (%l)   <sym_name="n3", impl="l_to_rgb",        width=640, height=480> : ( ... )→( ... );
             iara.node (%gs)  <sym_name="n4", impl="write_rgb_frame", width=640, height=480> : ( ... )→();
```

- Ports represented by values, not attributes
- Names of ports are not preserved between passes (won't preserve debug info)
- Need to represent rate information (which could be very complex) in a custom type
- Need some sort of parameter indexing for more complex type information
  - For instance, reactive dataflow MoCs need to be able to represent associations between different ports/parameters in the type signature of actors
- Loss of source code location information and names of edges
- Would need a new operation anyways to add delays




```
%b_o1, %b_o2  = iara.node : (%b_delay, %a_o1)→( ... ); // Node B
%b_delay      = iara.delay (%b_o1) {delays=[1,2,3,4 ... ]};
```

Solutions chosen

- Used declarative IR directly
  - Did type-checking manually
    - Already had to walk the graph to check for SDF rate consistency
    - Could only get away with this because of SDF's simplicity
    - The ideal thing would be to find a way to use MLIR's type system.
- Not ideal for DF models that don't have kernel and graph separation
  - For instance, languages such as LUSTRE have the actor implementation mixed in with the graph information and signature.

A small proposal


```
%1... = some.op : () -> (.a: i32, .b:i32);
some.other_op (%1.a, %1.b);

// equivalent to

%1_a, %1_b = some.op : () -> (i32, i32);
some.other_op(%1_a, %1_b);
```

- Named outputs
  - All outputs of an operation share the same "namespace"
  - Names defined by the operation and ensured by the type system
  - Don't have to keep the index of the output
  - Persistent syntax between passes
  - Would facilitate several abstractions, not only dataflow
    - Would allow for struct-like abstractions
    - Wouldn't need member access operations
  - Any obvious reasons why it would be unfeasible/unnecessary?

Thank you!

Questions or feedback?