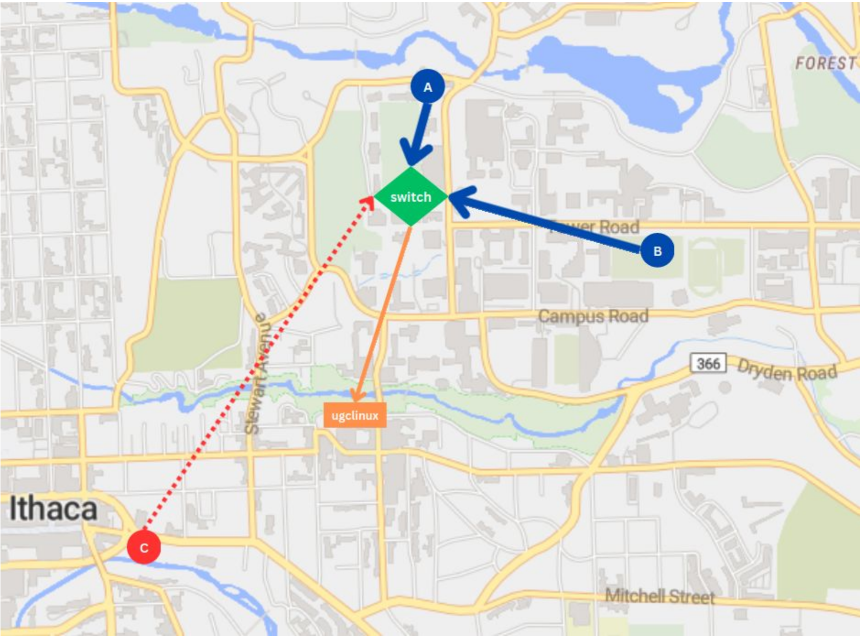
Programmable Packet Scheduling

Akash, Cassandra, Kabir

Context

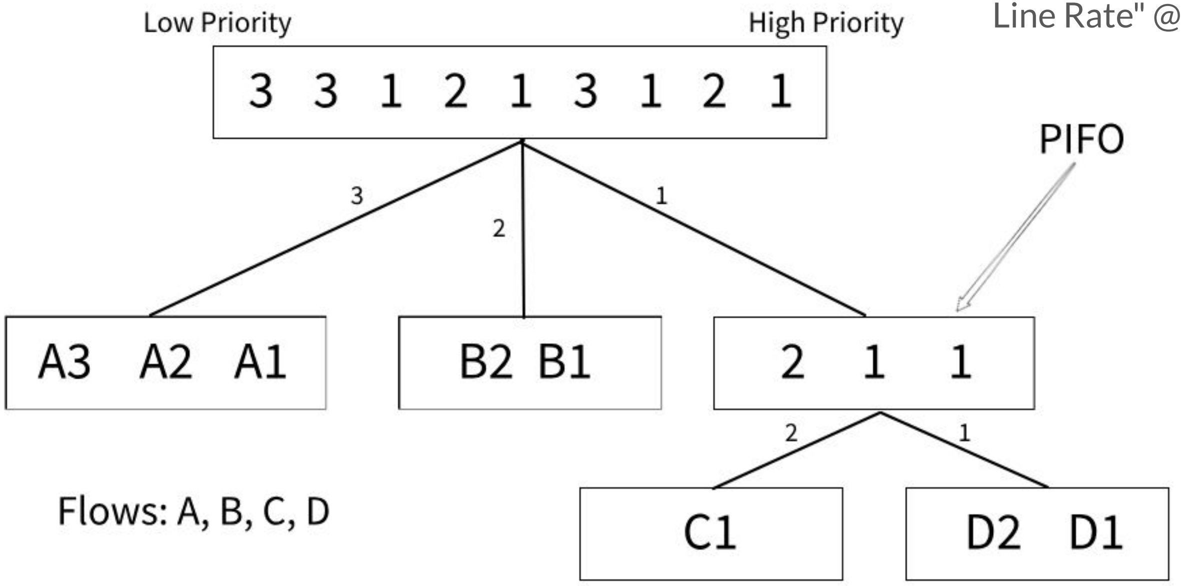


WANT: **Programmable and Fast!**

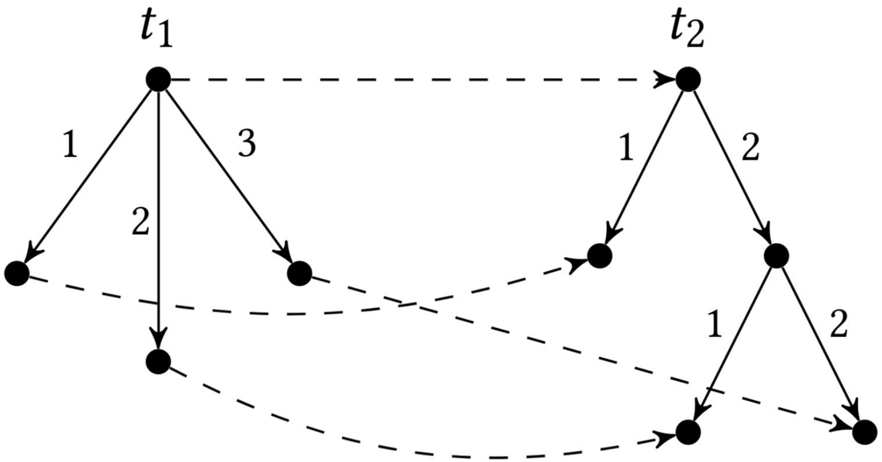
Background – PIFO Trees

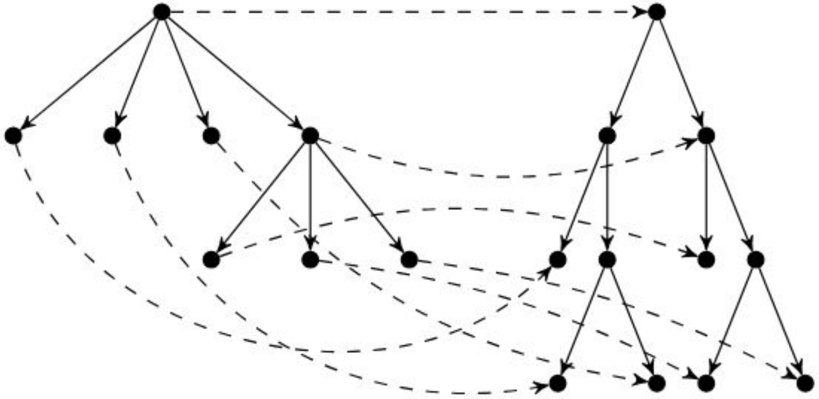
Priority Queue

Sivaraman's "Programmable Packet Scheduling at Line Rate" @ SIGCOMM'16



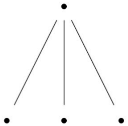
Background – Tree to Tree (T2T) Compilation





Mohan's "Formal Abstractions for Packet Scheduling" @ OOPSLA'2023

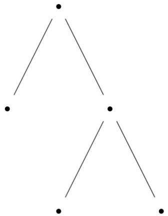
The Old Dream



```
classes A, B, C;
policy = strict[A, B, C];
return policy;
```

T2T

```
classes A, B, C;
policy = ????
return policy;
```



Rio

Rio++

```
component policy(@go go: 1, packet: 32) -> (@done done: 1) {
  cells {
    pifo_root = pifo();
    pifo_0 = pifo();
    pifo_1 = pifo();
    pifo_2 = pifo();
    ....
  }
  wires { ... }
  control { par { push; pop; } }
}
```

Calyx

T2T

```
component policy(@go go: 1, packet: 32) -> (@done done: 1) {
  cells {
    pifo_root = pifo();
    pifo_0 = pifo();
    pifo_1 = pifo();
    pifo_10 = pifo();
    pifo_11 = pifo();
    ....
  }
  wires { ... }
  control { par { push; pop; } }
}
```

Calyx

AMD XILINX FPGA

AMD XILINX FPGA

Partial FPGA programming? Nope.

New bitstream = full cost, every time :(

The New Dream – Checkpoints 1 & 2

```
classes A, B, C;
policy = strict[A, B, C];
return policy;
```

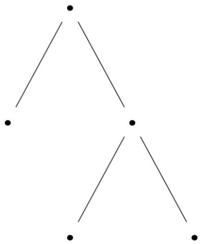
Calyx

SystemVerilog

1

AMD XILINX FPGA

```
classes A, B, C;
policy = strict[A, B, C];
return policy;
```



Calyx

SystemVerilog

2

AMD XILINX FPGA

Towards Checkpoint 1

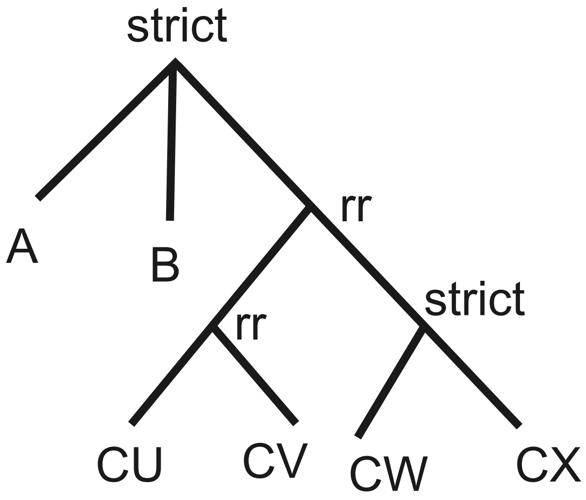
- ❑ Hierarchical policies networking folks care about
- ❑ Sivaraman's PIFO implementation
- ❑ Operational semantics for Rio

Overview

```
classes A, B, CU, CV, CW, CX;

c_policy = rr[rr[CU, CV], strict[CW, CX]];
policy = strict[A, B, c_policy];

return policy
```



Overview

Work conserving vs. non-work conserving policies

Work conserving - round robin, strict, weighted fair queueing (WFQ), FIFO order/FCFS, etc.

Non-work conserving - leaky bucket filter, rate controlled static priority, etc.

Policies people are interested in

- Not all hierarchical policy compositions make sense
- WFQ and Strict are popular

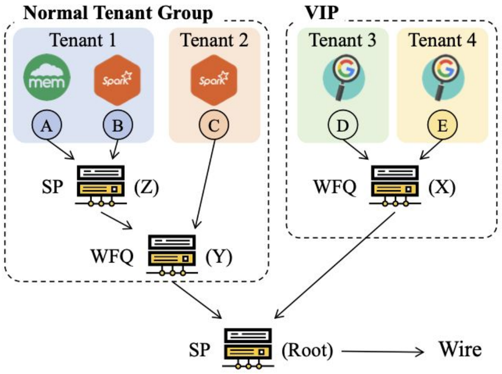


Figure 1: A Hierarchical Scheduling Policy in an MTDC.

Zhang et. al

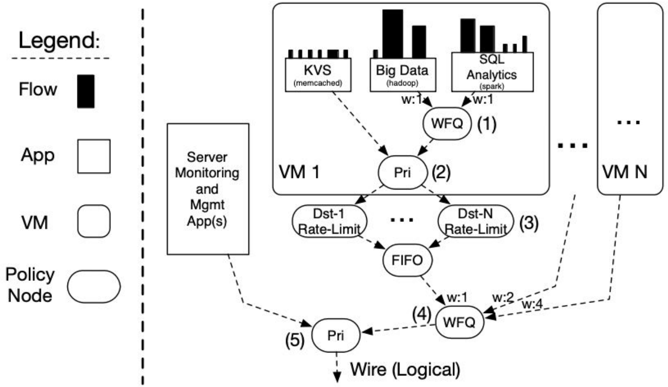


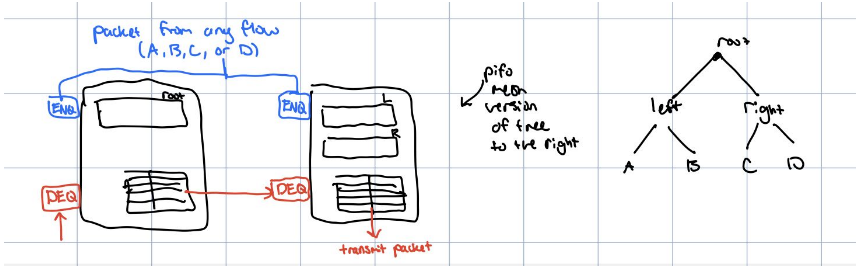
Figure 1: **A scheduling hierarchy for a server. Different parts of this hierarchy are specified by different entities (Section 3). The OS is responsible for dynamically enforcing this policy.**

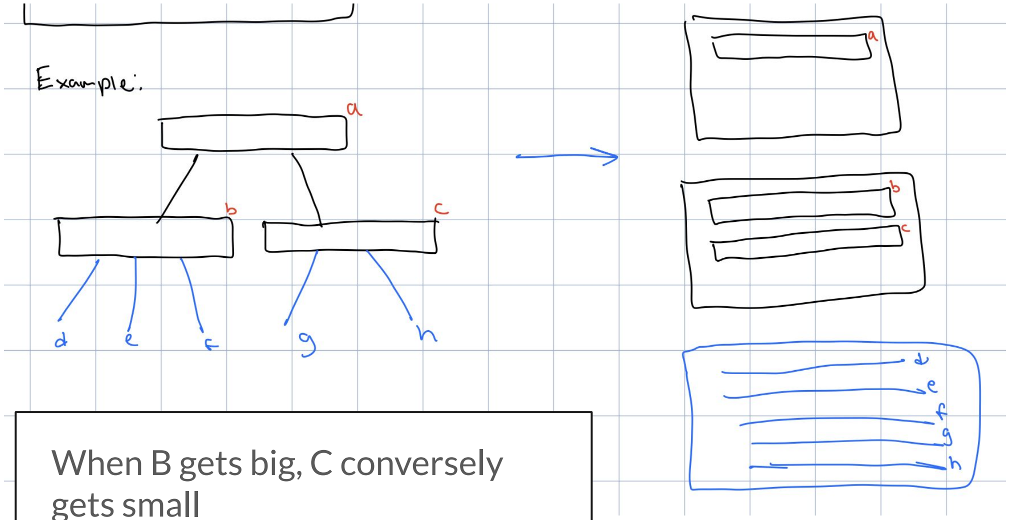
Loom paper (Stephens et. al)

Sivaraman et. al PIFOs

Overview of idea

- Represent PIFOs as a block
    - Each block corresponds to a different level on the tree
    - Queues in a block share memory





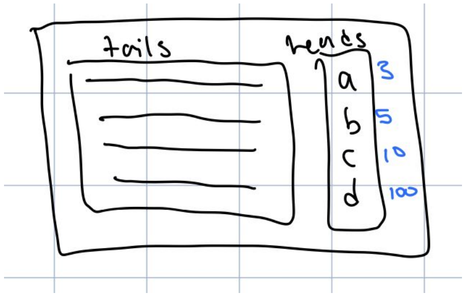
When B gets big, C conversely gets small

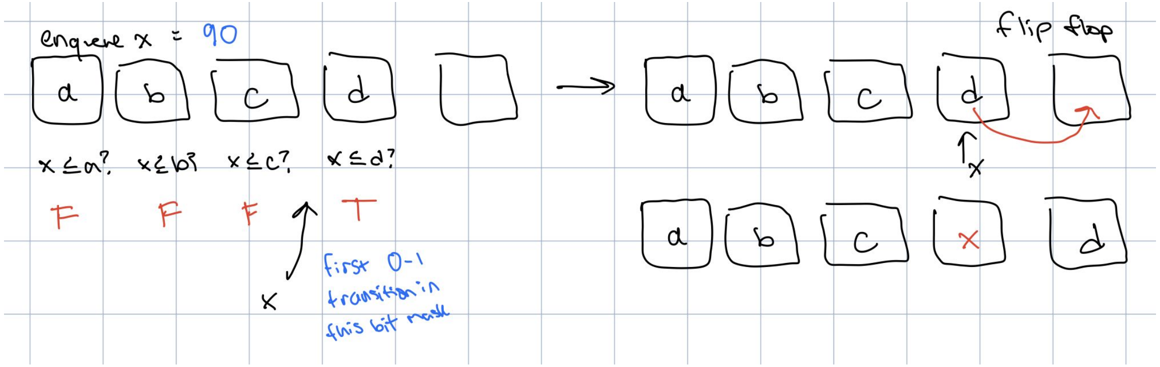
Hardware Implementation

Enqueue

- Incoming element is compared against all existing elements in parallel

Flow scheduler - makes it feasible to sort based on flows rather than individual packets





Policy Classification

Distinction Between Policies

Our project has looked at a variety of different policies, to determine *which packet to pop next.*

- **First-In First-Out** *(Which packet was enqueued first?)*
- **Earliest-Deadline First** *(Which packet needs to be processed first?)*
- **Round-Robin** *(Switch out between a series of different* ***flows****)*
- **Strict** *(Stick one one flow until it falls silent)*
- And more!

However, there's a key distinction:

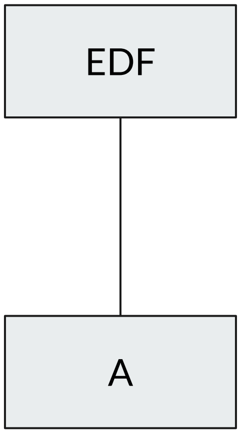
A policy arbitrates between different **classes** (sources of packets). Hierarchical policies may in turn arbitrate between others.

- Some policies (FIFO, EDF and others) want us to look at *every packet within every class under it.*

- Some policies (Round Robin, Strict and others) instead want us only to consider what their immediate children say, and arbitrate between **those candidates!**

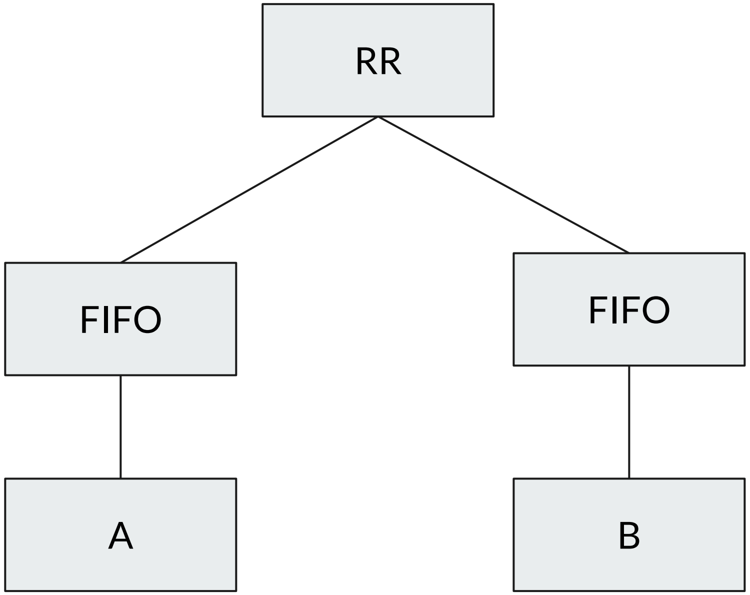
To get a better understanding:

Distinction Between Policies



Set-To-Stream

Here, every single packet within A is fair game.

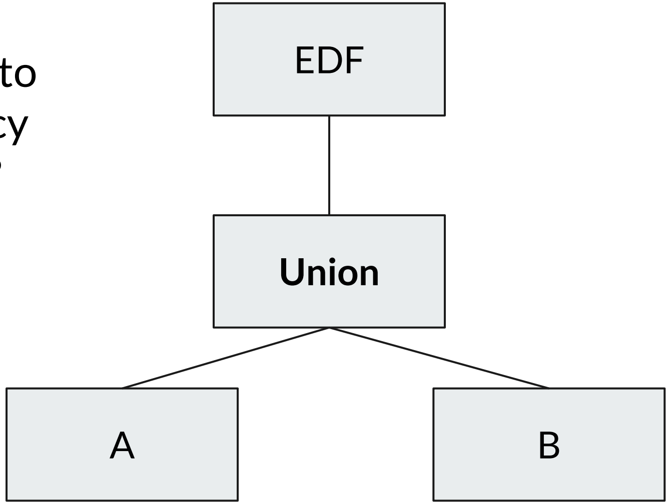


Stream-To-Stream

Here, we need to respect what the FIFOs say before RR can decide which packet to pop.

Union Operator

What happens if we want to have a **set-to-stream** policy look at a bunch of classes?



Now, every packet in both A and B is fair game with each pop.

Transformers

We propose two classifications of policies:

Set-To-Stream Transformers

- The policy node has **exactly one child** – a **set** (class or union of classes).
- When popping, *all packets within this are considered.*

Stream-To-Stream Transformers

- The policy node has as many children as it wants – **streams** (other policies).
- When popping, *only the pop candidates from each of its streams are considered.*

Representing Transformers As AST Nodes

```
type set =
  | Class of string
  | Union of set list

type stream =
  (* Set-to-Stream *)
  | Fifo of set
  | EarliestDeadline of set (* and others *)
  (* Stream-To-Stream *)
  | RoundRobin of stream list
  | Strict of stream list (* and others *)
```

- Programs take on the form of **streams**.
- Classes (and unions of classes) join the hierarchy via Set-To-Stream transformers.
- We build more complex programs via Stream-To-Stream Transformers

Encoding These Restrictions Into Semantics

To reason about these more formally, we can define operational semantics for **Rio programs** based on the types of policies they contain.

We denote a concept of **state –** a set of PIFOs containing our **packets** and a **program**. **State** tuples are subject to the following operations:

- `push : pkt *  queue *  state -> state`
- `pop : state -> pkt option * state`

Pushing takes in a packet and specific queue, an existing state tuple, and accordingly updates the state.

Popping takes in a state tuple, uses the program and semantics to determine the next packet to pop, and returns it along with the updated state.

Compilation Pipeline – PIFO Trees vs. Rio Programs

Compiled Rio Programs (PIFO Trees)

- Compile policies to PIFO Trees
- The queues used in state tuples become the leaves of the resultant PIFO Tree
- Packets are popped using standard logic of PIFO Trees

Semantically Evaluated Rio Programs

- Packets are directly popped based on the POP and PUSH rules of our operational semantics
- PIFO Trees play no role!

Next goal: Show equality between the two!

Current Work - Showing Equality Between The Two

- A big part of our work right now circulates around showing that compiled Rio programs are equivalent to how their semantics describe them.
- More formally – **popping** from any program should always give the same packet as popping from its corresponding PIFO Tree, once compiled.
- We are working on a mechanized proof of equality for a subset of the language, using the Coq proof assistant.