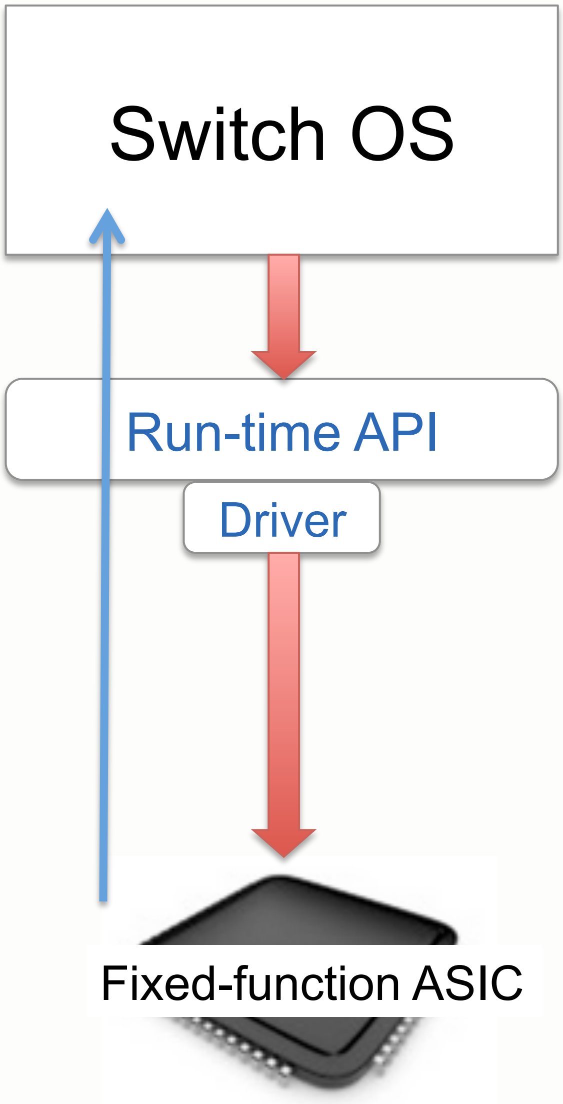
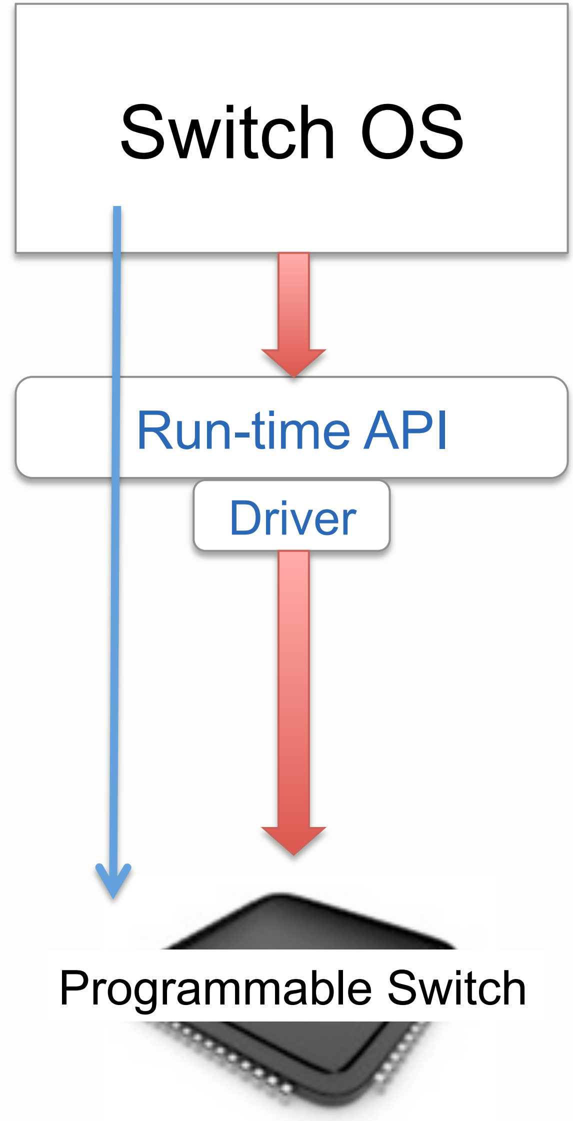
Compiling Packet Programs to Reconfigurable Switches

Lisa Yan and Lavanya Jose
Stanford University



“Bottom-Up”
“This is how I process packets”



“Top-down”
“This is how the switch must process packets”

Programmable switches

Some switches are more programmable than fixed-function ASICs

- CPU/NPU (OVS, Ezchip, Netronome, etc.)
- FPGA (Xilinx, Altera, Corsa)
- **Flexible Match+Action ASICs**
  (Intel Flexpipe, Cisco Doppler, Xpliant, …)

http://p4.org/

P4: A high-level language

Header Fields: VLAN

```
header_type vlan_tag_t {
  fields {
    ...
    vid         : 12;
    etherType   : 16;
  }
}
header vlan_tag_t vlan_tag[NUM];
```

Match+Action Table: VLAN

```
table port_vlan {
  reads {
    std_metadata.ingress_port : exact;
    vlan_tag[OUTER_VLAN].vid : exact;
  }
  actions {
    drop, ing_lif_extract;
  }
  size 16384;
}
```

Parser

```
parser parse_ethernet {
  extract(ethernet);
  return switch(latest.etherType) {
    ETHERTYPE_VLAN : parse_vlan;
    ETHERTYPE_MPLS : parse_mpls;
    ETHERTYPE_IPV4 : parse_ipv4;
    ETHERTYPE_IPV6 : parse_ipv6;
    ...
  }
}
parser parser_vlan {
  extract(vlan_tag[next]);
  ...
}
```

Control Flow: Ingress

```
control ingress {
  apply_table(mac_learning);
  if (valid(vlan_tag[0])) {
    apply_table(port_vlan);
  }
  apply_table(routable) {
    ucast_action {
      apply_table(ucast);
    }
    mcast_action {
      ...
  }
...
}
```

P4: A program

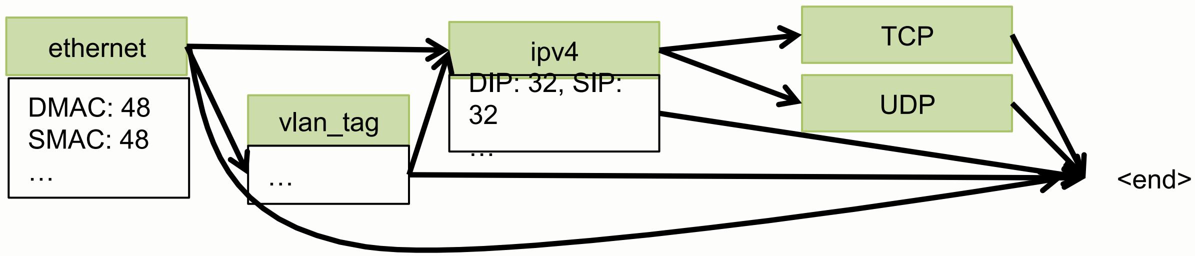
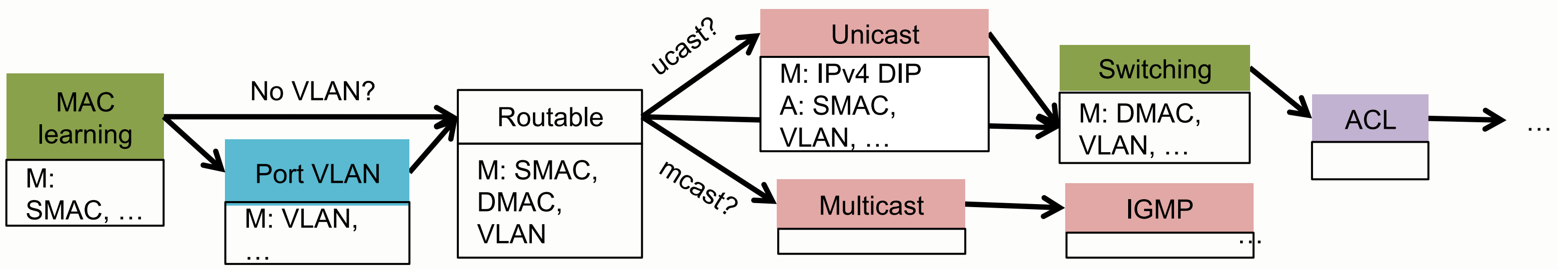
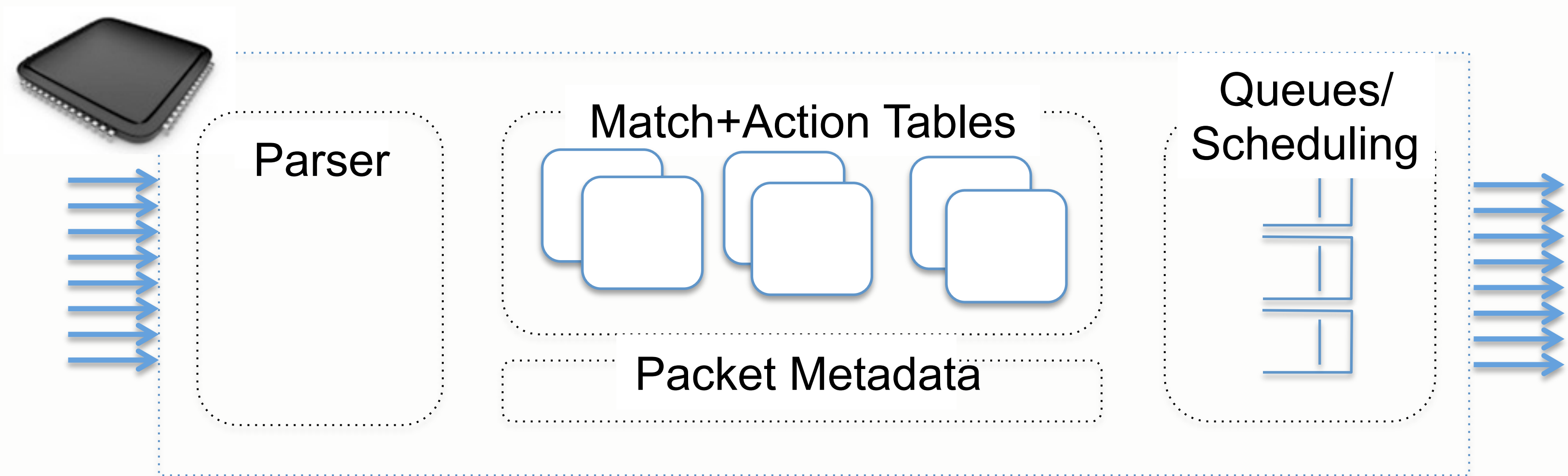
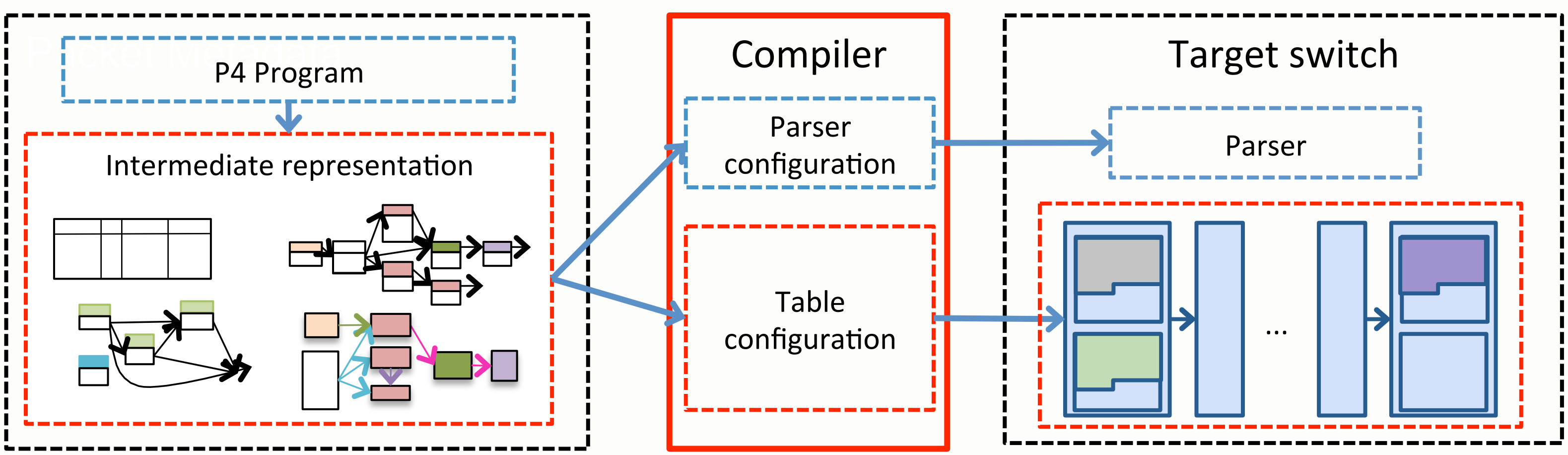


Table name	Match width	# entries	Type
MAC Learning	60 b	4000	exact
Unicast	32 b	2000	ternary
Switching	60b	4000	exact
ACL	110b	1000	Ternary
…	…	…	…



Why a compiler?



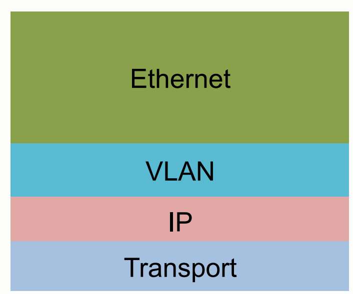


The Program: A representation

- Assignment constraint: all tables should be allocated somewhere in the pipeline

- How do we respect control flow while maximizing concurrency in a switch pipeline?

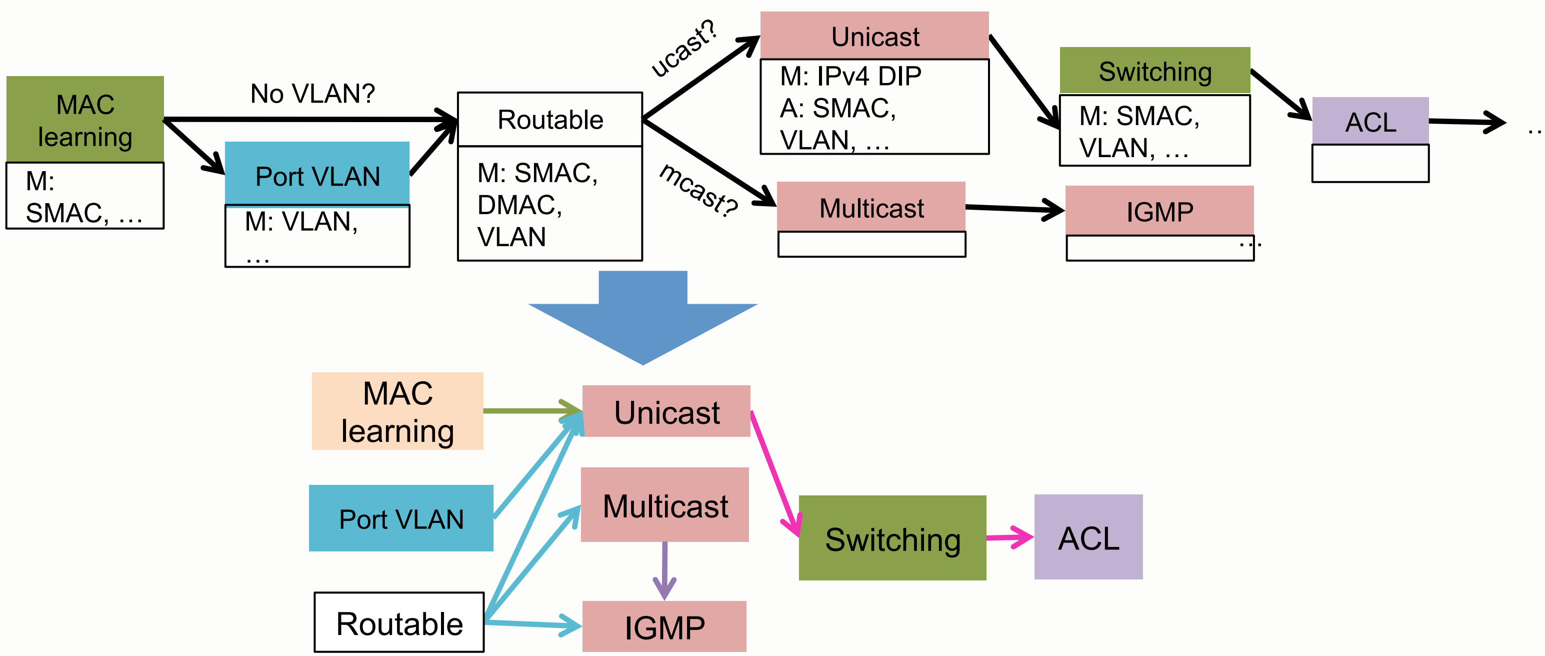
Table name	Match width	# entries	Type
MAC Learning	60 b	4000	exact
Unicast	32 b	2000	ternary
Switching	60b	4000	exact
ACL	110b	1000	Ternary
…	…	…	…



←MAC learning: reads SMAC
Switching: reads outgoing DMAC, modifies egress port
←Port VLAN: drops based on VLAN
←unicast: modifies SMAC, DMAC, and VLAN

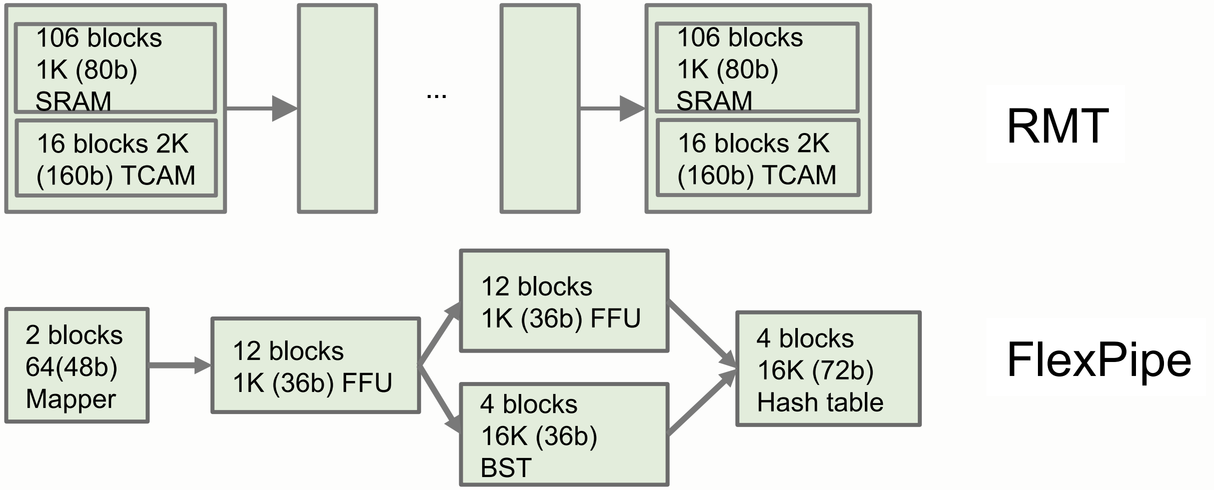
Representing Control Flow

- Table Dependency Graph (TDG)
- Directly generated from P4 program
- <u>Dependency constraint</u>: all tables assigned should respect the program control flow



The Switch

- Memory types
- Overhead memory: actions, statistics
- Capacity constraint: assignment of tables should not overflow memory per stage
- Switch-specific: input crossbars, match layout



Summary of the compiler problem

Goal: map a program to a switch's pipeline by exploiting concurrency

Constraints:

- – Assignment – all tables are somewhere
- – Dependency – respect program control flow
- – Capacity – obey memory limits of switch
- – Switch-specific

Approach 1: ILP

- Integer Linear Programming (ILP)
  - Make constraints into inequalities
  - Find the best valid solution to an objective function
- Constraints:
  assignment, capacity, dependency, switch-specific
- Objective functions:
  number of stages used, pipeline latency, power consumed

minimize: $4y$
subject to:
$$3x + 2y \geq 6$$
$$1 \leq y \leq 2$$
$$x \leq 1$$

ILP Example: Assignment

- All entries of a table must be assigned
- Table A: 5000 entries, exact match
- Memory types *m* = exact, ternary. …
- $W_{s,\mathrm{A},m}$ = # entries from Table A assigned to stage *s*, memory type *m*

$$\boxed{\sum_{s,m} W_{s,\mathrm{A},m} \geq 5000}$$

Approach 1: ILP

(+) global view of the problem

(+) returns best solution when possible

(–) NP-complete

(–) solver runtime is long

Approach 2: Greedy

1. Sort tables according to some heuristic
2. For each table:
   - Put in earliest stage possible
   - If constraints are violated, move to next stage
   - Repeat until all tables assigned

Approach 2: Greedy

(+) fast runtime

(+) control over sorting heuristic

(–) local view of resources

(–) may not find best (or any) solution

Experiments

- 2 different switches: RMT, FlexPipe
- 4 different benchmarks

Benchmark	Switch	Tables	Dependencies
L2L3 Complex	RMT	24	33
L2L3 Simple	RMT	16	19
	FlexPipe	13	16
L2L3 mTag	RMT	19	22
	FlexPipe	15	17
L3 DC	RMT	13	11

Results

- 3 greedy heuristics
- 3 ILP objective functions

Worst median greedy runtime: 0.33 seconds

Worst median ILP runtime: 233.84 seconds

L2L3 Complex Benchmark on RMT chip

	# stages	latency (ns)	power (W)
Best greedy	17	130	4.98
Optimal (ILP)	16	104	4.44

What did we learn?

Greedy:

- Much faster than ILP
- Choosing a heuristic is important

→ If you *need* the program to fit, use ILP

Greedy and ILP can help each other!

What else did we learn?

- Not all heuristics are created equal
    - ILP validates that dependency-based is best
- Use greedy estimates to speed up ILP
- Design decisions from compiler solutions
    - Some programs push the limits of the switch
    - Use compiler to validate memory dimensions based on expected program input

Thanks!

P4: http://p4.org/

RMT:

http://yuba.stanford.edu/~grg/docs/sdn-chip-sigcomm-2013.pdf

FlexPipe:

http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf

Advisors:

Nick McKeown (Stanford), George Varghese (Microsoft Research)