nsdi '22

Enabling In-situ Programmability in Network Data Plane: From Architecture to Language

Yong Feng[1], Zhikang Chen[1], Haoyu Song[2],
Wenquan Xu[1], Jiahao Li[1], Zijian Zhang[1], Tong Yun[1], Ying Wan[1], Bin Liu[1]

[1]Tsinghua University; [2]Futurewei Technologies

Development of Switching ASIC

- Fixed-function Switch

- Programmable Switch
  - customize protocol, flow table and processing logic before deployment
  - Intel Tofino, Broadcom Trident4

- **I**n-situ **P**rogrammable **S**witch (**A**rchitecture) – **IPSA**
  - Customize protocol, flow table and processing logic ***at runtime***

Motivations

- Dynamic network telemetry or measurement
- Transitory in-network computing
- Table refactoring and repurposing
- Multi-tenant non-interrupted service
- State preserving

Motivations

- Dynamic network telemetry or measurement
- Transitory in-network computing
- Table refactoring and repurposing
- Multi-tenant non-interrupted service
- State preserving

Motivations

- Dynamic network telemetry or measurement
- Transitory in-network computing
- Table refactoring and repurposing
- Multi-tenant non-interrupted service
- State preserving

Motivations

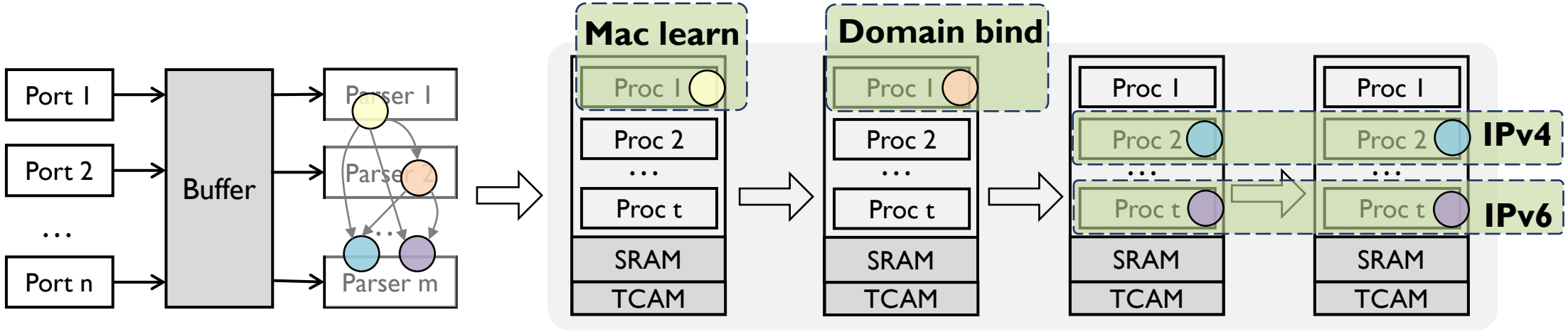
- Dynamic network telemetry or measurement
- Transitory in-network computing
- Table refactoring and repurposing
- Multi-tenant non-interrupted service
- State preserving

Motivations

- Dynamic network telemetry or measurement
- Transitory in-network computing
- Table refactoring and repurposing
- Multi-tenant non-interrupted service
- State preserving

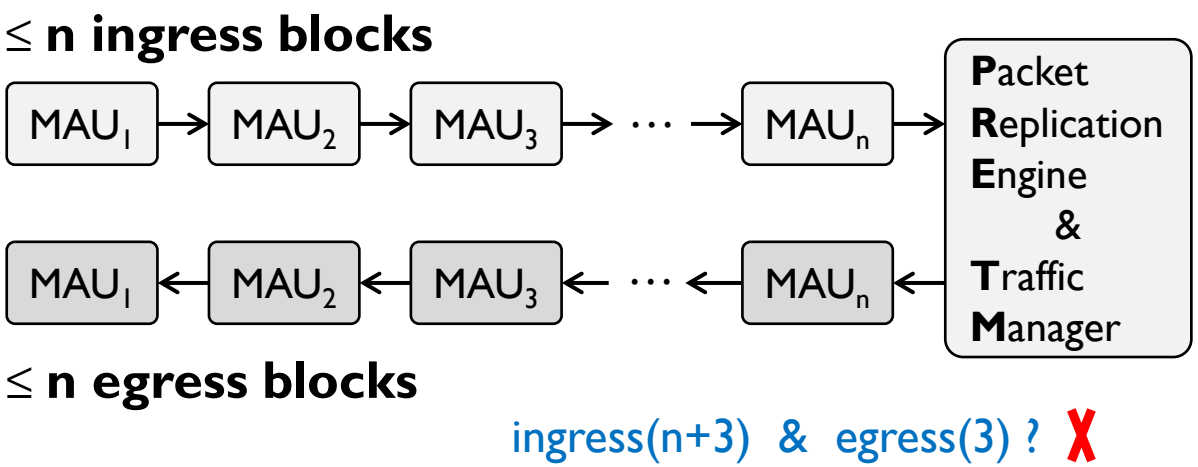
Limitations of implementing in-situ programming in PISA?

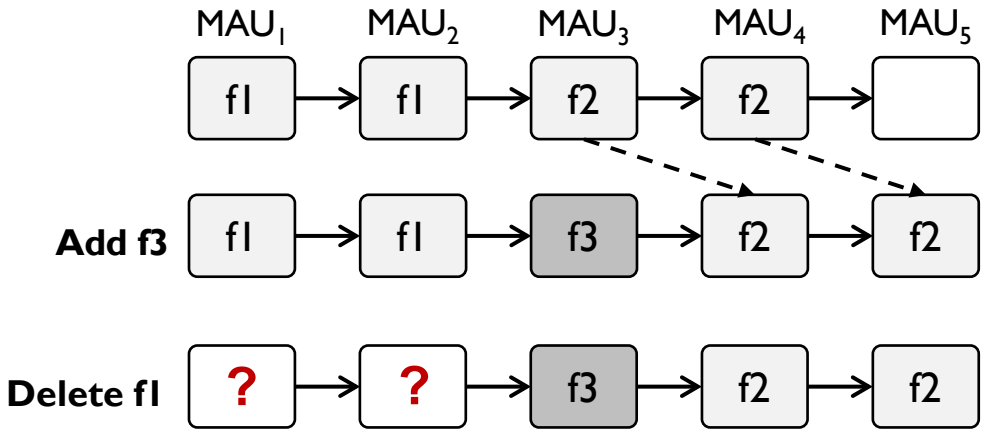
Programmability in PISA – Protocol (why not)



- **Decoupling may be something bad**
  - Self-contained?
  - Insert new protocol with fields?

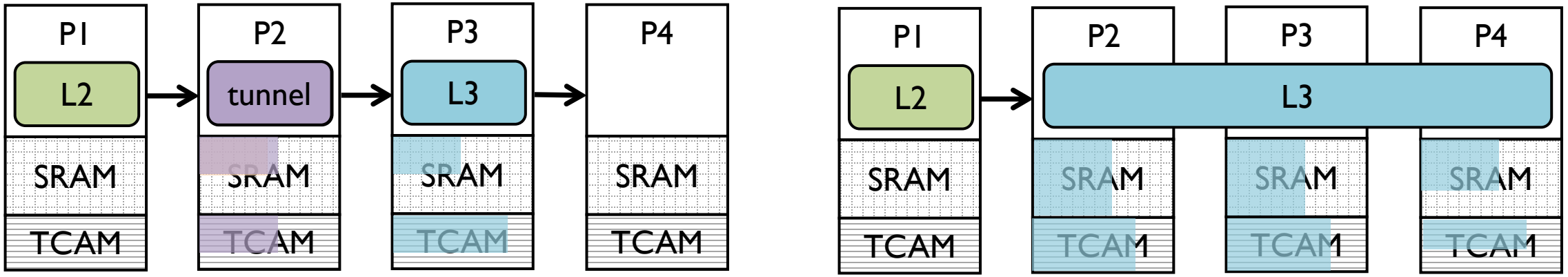
Programmability in PISA - Processing Logic (why not)





- **Flexibility of hardwiring processors**
  - Processors in Ingress and Egress pipelines ?
  - Insert new function blocks or delete blocks?

Programmability in PISA - Memory



- **Concerns in Memories**
  - Processing logic migration → table migration among different processors
  - One function with three processor memories → logic replication

IPSA vs. PISA (how we solve limitations of PISA)

- Modifying front parser blocks all operations

  Distributed, on-demand, self-contained **parser**

- Binary executable cannot be altered when running

  Template-based stage processor: *parser-matcher-executor* backbone

- Inserting or deleting functions blocks mean processing logic migration

  Virtual pipeline with crossbar (crossbar, CLOS, Bens, BB, and etc)

- Cannot do table refactoring

  Disaggregated memory pool: create, recycle dynamically

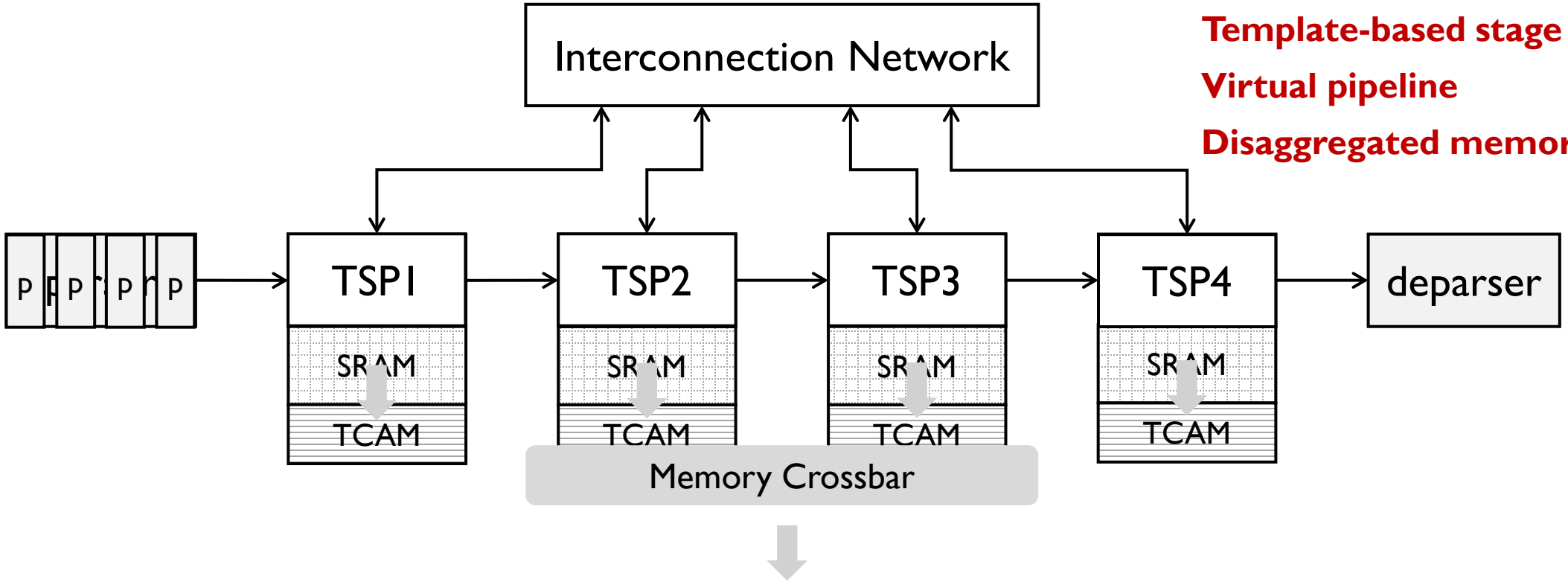
IPSA vs. PISA (how we solve limitations of PISA)

Distributed on-demand parser

Template-based stage processor

Virtual pipeline

Disaggregated memory pool



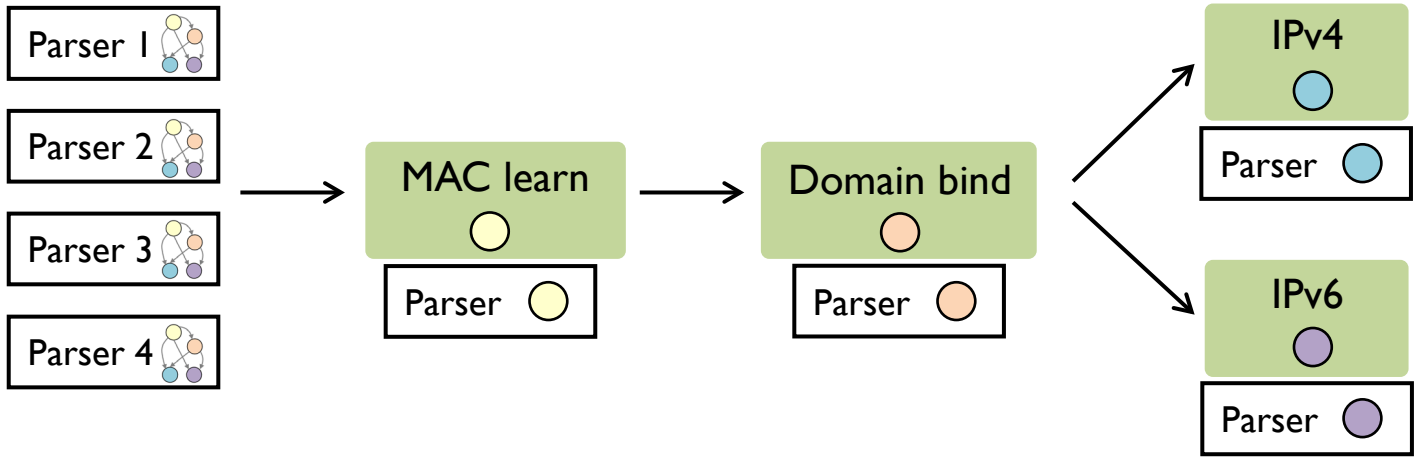
IPSA - Distributed on-demand parser (why can)

- Eliminate front parser and distribute it among processors
- Function block only parses its own needed protocols

~~PHV (packet header vector)~~

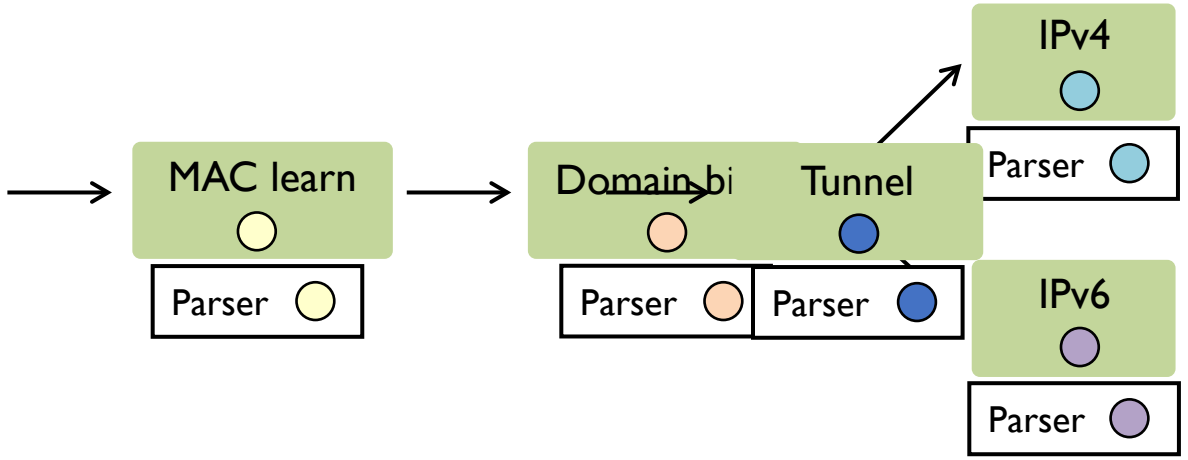
↓

Packet window



IPSA - Distributed on-demand parser (why can)

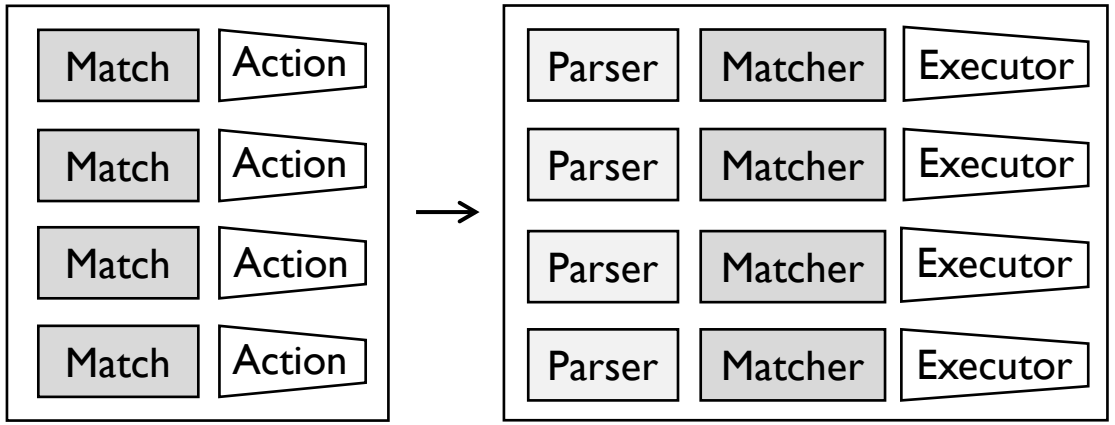
- Eliminate front parser and distribute it among processors
- Function block only parses its own needed protocols
- Simplify modularized function updates, minimize the effect to update function



- Distributing algorithm: see in paper.

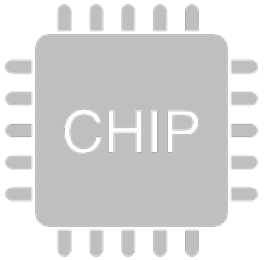
IPSA - Template-based Stage Processor (why can)

- Abstraction: Match-Action → Parser-Matcher-Executor
- Template (Backbone): Binary Executables → Logic Parameters



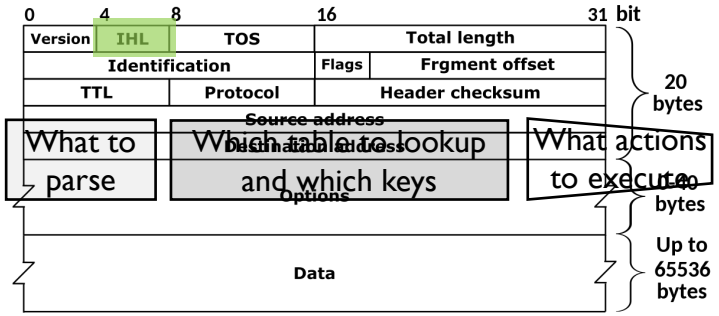
PISA Processor

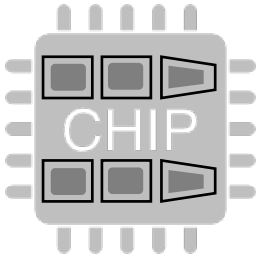
IPSA Template-based Stage Processor



Download binary executables (PISA)

POF: {ipv4_id, ihl_offset, ihl_length}



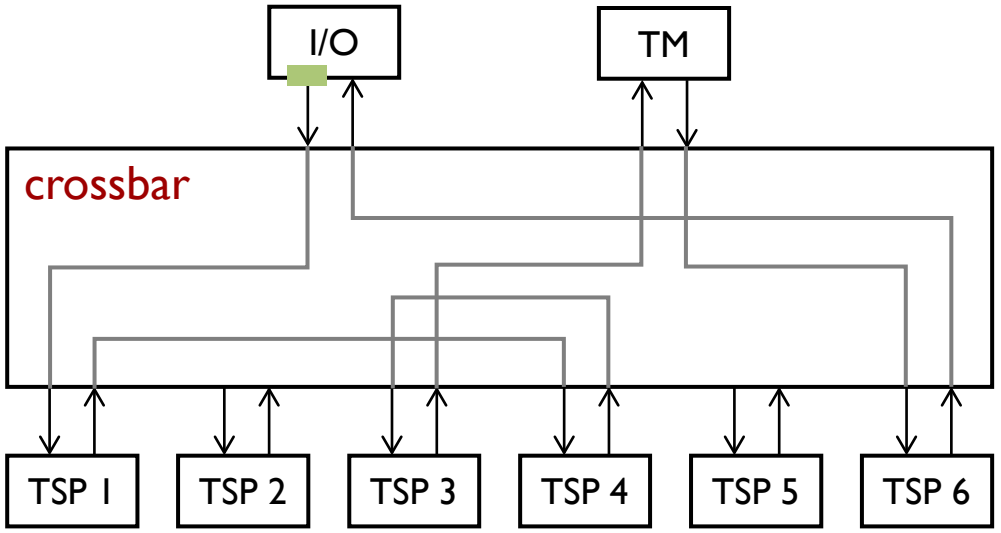


Download parameters (IPSA)

IPSA - Virtual Pipeline (why can)

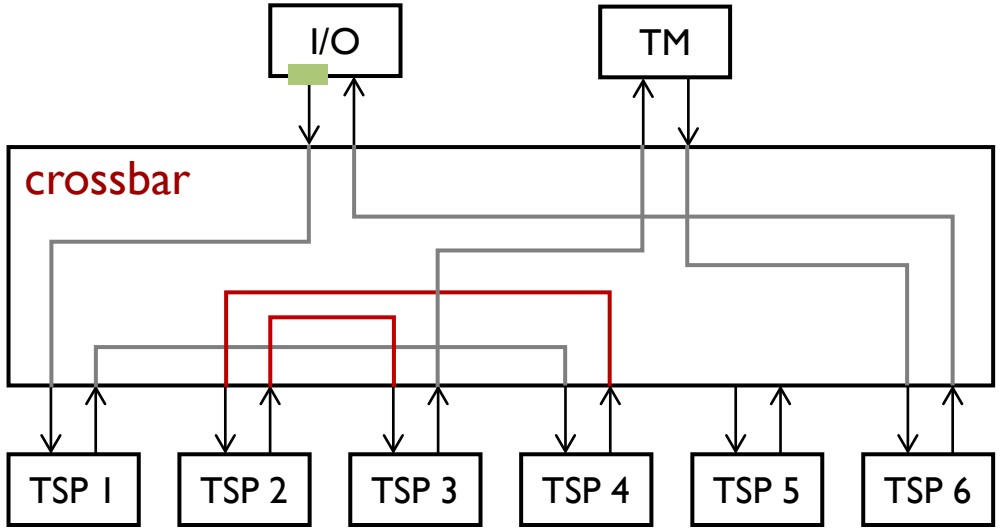
- Interconnection network (IN) : Insert/delete function stages flexibly
- Enlarge the design space of ingress and egress pipelines
- Analysis of different INs can be seen in paper.

- **Crossbar act as configurator, not scheduler !**
- **Full pipeline implementation**



Input → 1 → 4 → 3 → TM → 6 → Output

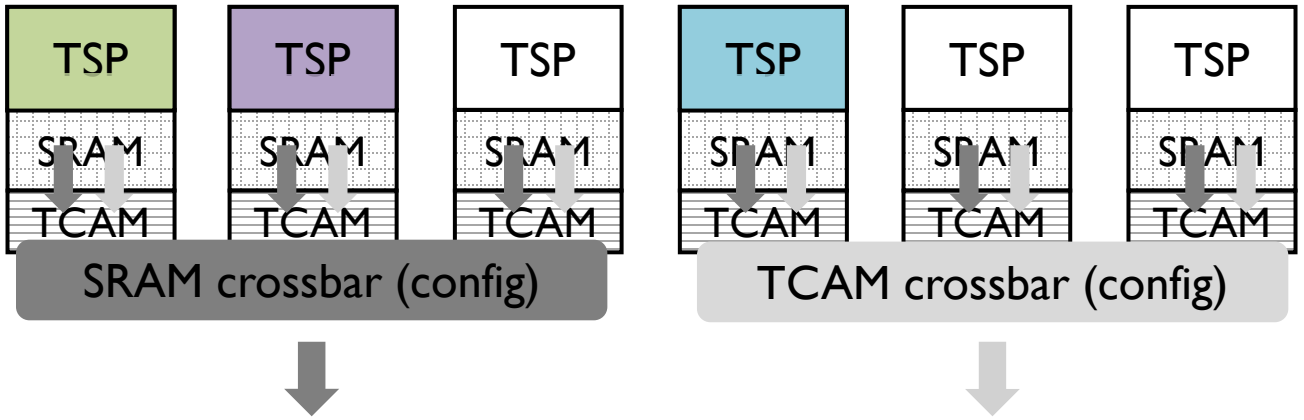
Insert function stage



Input → 1 → 4 → **2** → 3 → TM → 6 → Output

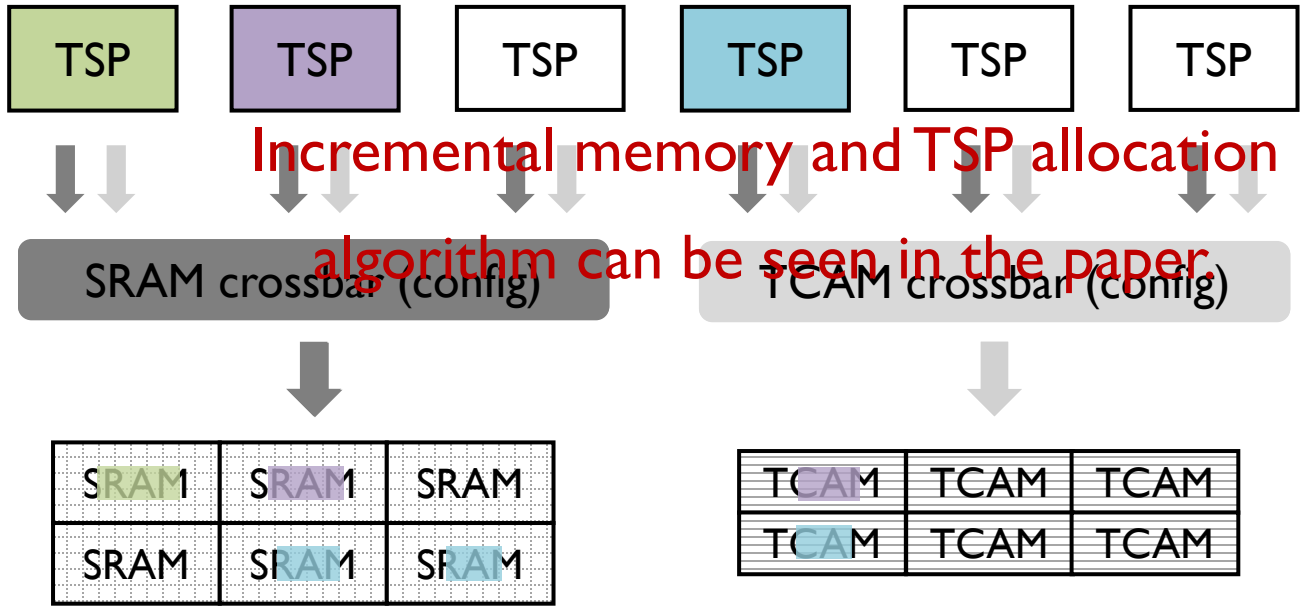
IPSA - Disaggregated Memory (why can)

- Recycle and create flow tables at runtime
- Cluster: trade off complexity for flexibility and scalability

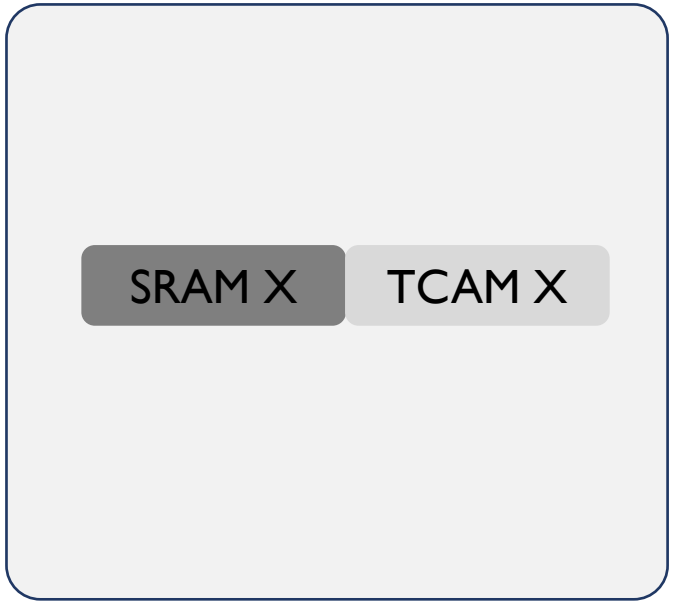


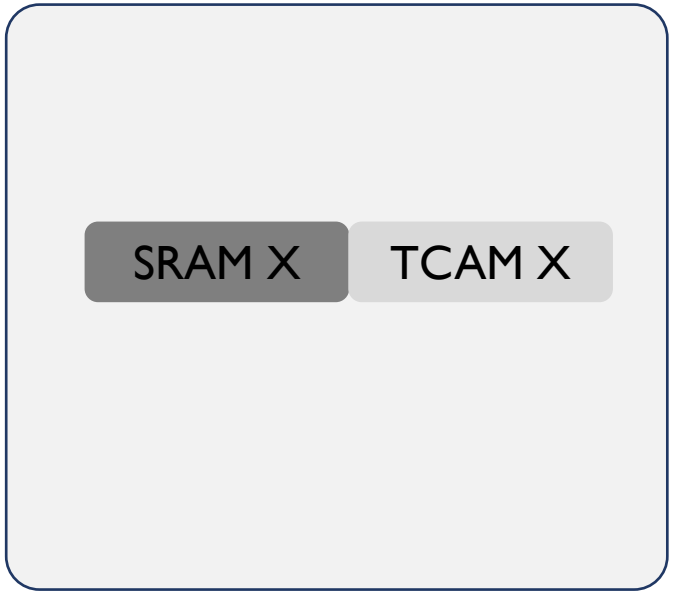
IPSA - Disaggregated Memory

- Recycle and create flow tables at runtime
- Cluster: trade off complexity for flexibility and scalability

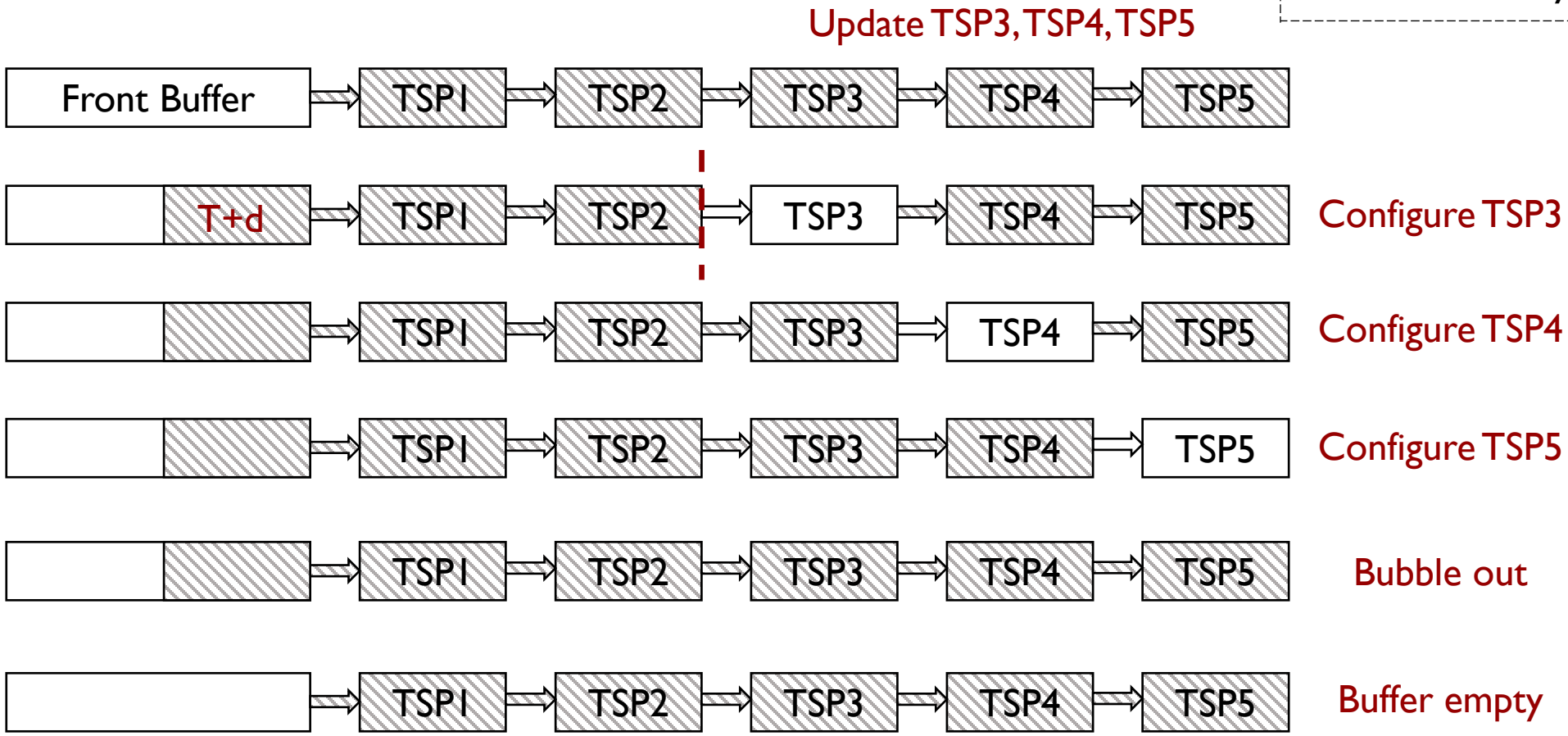


Incremental memory and TSP allocation algorithm can be seen in the paper

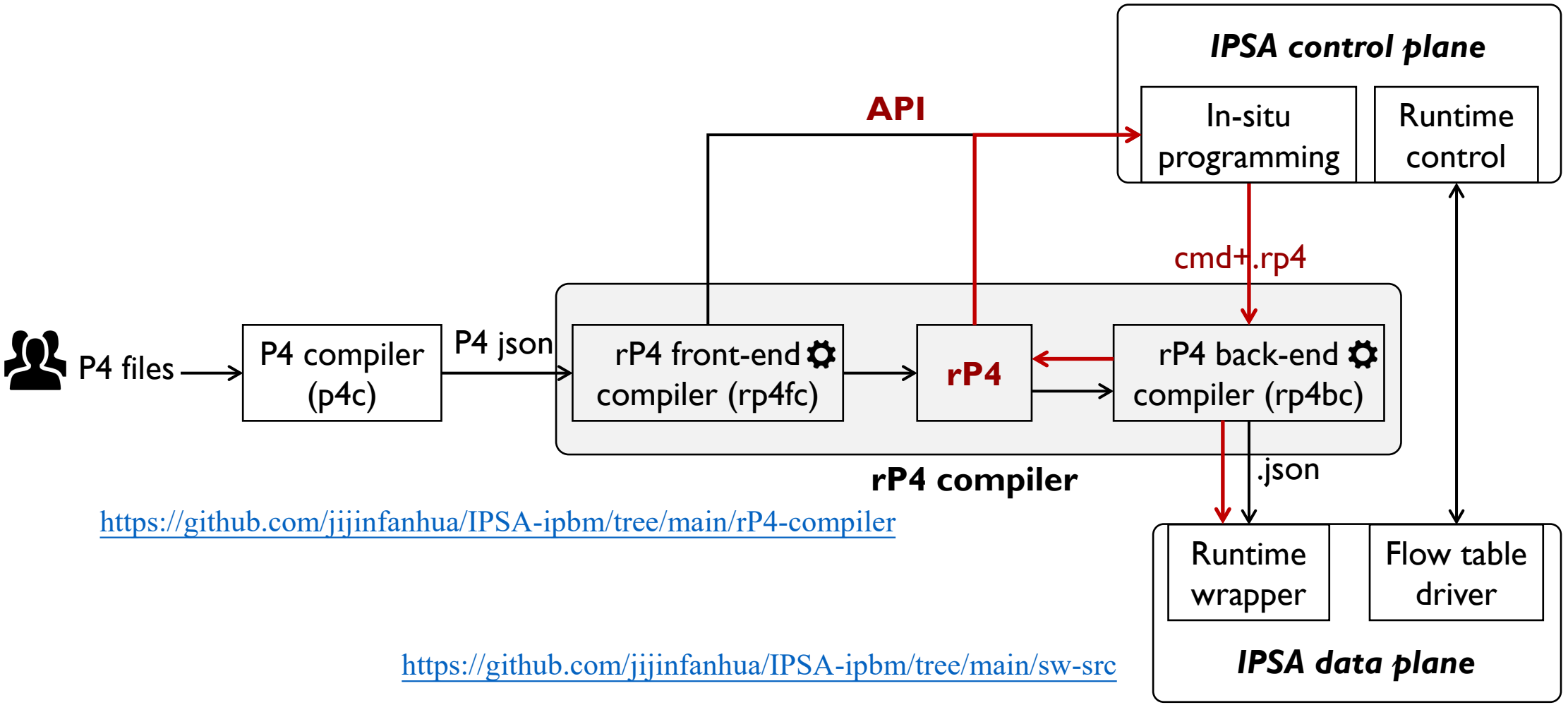




IPSA - Big Bubble Update



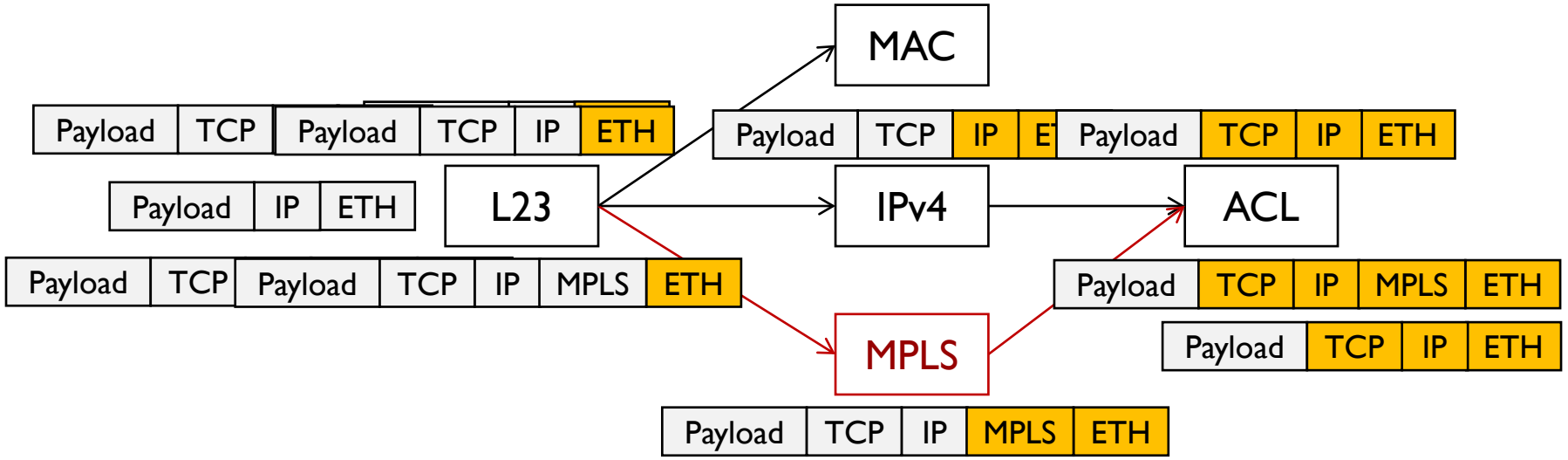
Workflow of rP4 (r - reconfigurable)



https://github.com/jijinfanhua/IPSA-ipbm/tree/main/rP4-compiler

https://github.com/jijinfanhua/IPSA-ipbm/tree/main/sw-src

Use Case



Script:

```
load mpls.p4 --func_name MPLS
add_link L23 MPLS
add_link MPLS ACL

load mpls_header.p4 --header_name mpls
add_child ethernet mpls ethertype 0x8847
add_child mpls mpls s 0
```

```
header mpls {
    bit<20> label;
    bit<3>  exp;
    bit<1>  s;
    bit<8>  ttl;
}
```

```
stage mpls_s{
    parser { mpls };
    matcher {
            if(mpls.isValid())
                mpls_table.apply();
    }
    executor {
            0: pop_mpls;
            1: swap_mpls;
    }
}
```

Evaluation - Resources

- PISA and IPSA config:
  - 12 processors, each with two sets of configuration
  - 256-byte PHV (PISA) and 192-byte packet window (IPSA)
  - 3 clusters for IPSA, each with 64 memory blocks
  - Every PISA processor has 16 memory blocks

12.09% more LUTs

9.69% more FFs

There should be more spaces to store the configurations!

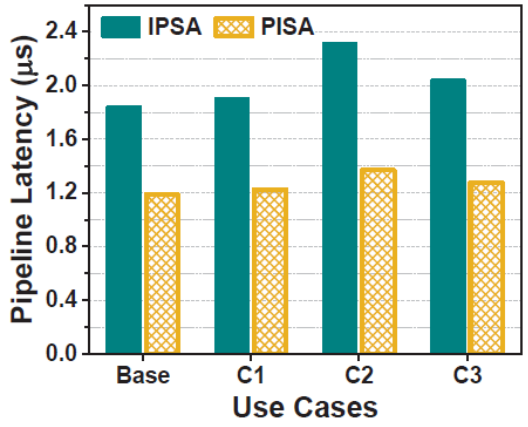
Prototype	**PISA**		**IPSA**	
Resource	**LUT**	**FF**	**LUT**	**FF**
Parsers/Deparsers	0.94%	1.54%	-	-
Processors	49.55%	3.52%	42.02%	13.72%
Crossbar	-	-	3.08%	0.01%
Inter-Network	-	-	17.48%	1.02%
Total	50.49%	5.06%	62.58%	14.75%

Evaluation - Throughput and Latency

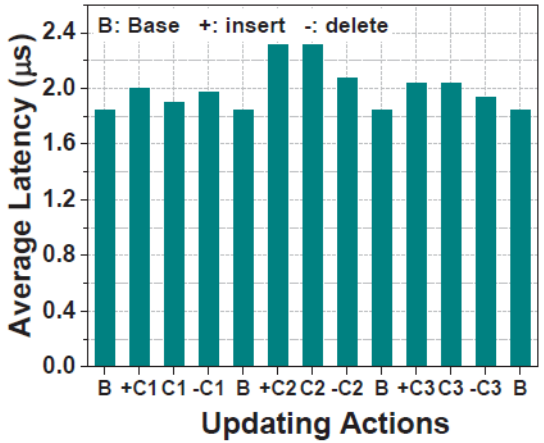
- Tool: Vivado Design Suite
- Platform: Alveo U280
- Synthesized Clock Frequency
  - PISA: 153.30MHz (235.47Gbps)
  - **IPSA**: 110.45MHz (169.65Gbps, limited by the multi-die in FPGA, which can be solved in ASIC chip)

Longer latency: field and flow table profile fetching, match key assembly, and primitive loading

Latency while updating: relatively steady



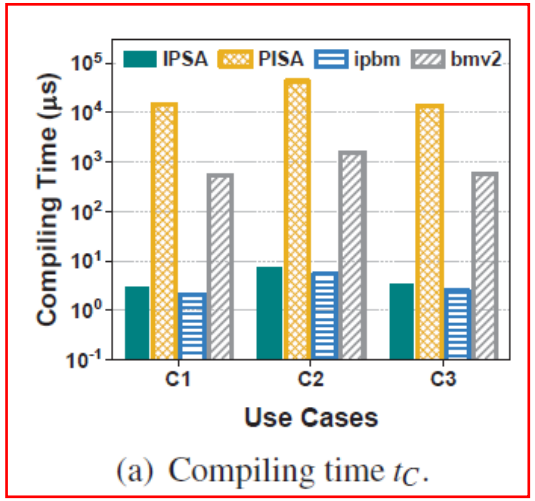
Forwarding latency of three cases



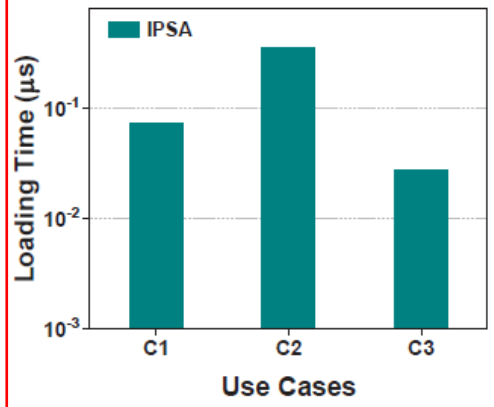
IPSA average pipeline latency while updating

Evaluation - Update Performance

IPSA only compiles the updated code segment, it takes much shorter time than PISA.



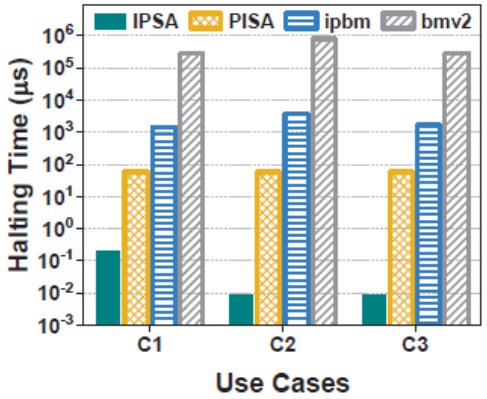
(a) Compiling time t_C.



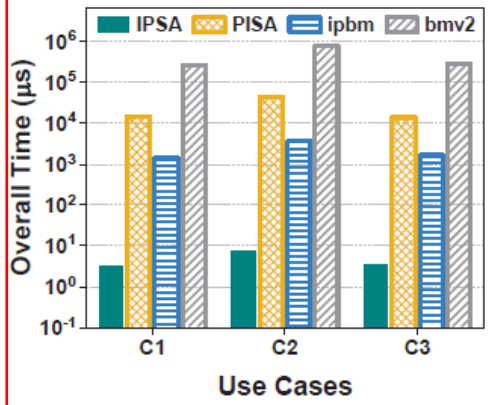
(b) Loading time t_L of IPSA.

Configuration loading time of IPSA.

IPSA's pipeline halting time is only 0.34% of PISA's.



(c) Pipeline halting time t_H.



(d) Overall update time.

IPSA's overall time is four orders of magnitude smaller than that of PISA.

Conclusion

- IPSA is proposed and is verified in theory and FPGA implementation.
  - support runtime reconfigurability in terms of protocol, processing logic and flow table.
- rP4 is extended from P4 to support user to reconfigure data plane at runtime.
- The behavioral model and corresponding compiler is open-sourced.
  - Provide researchers a new target to deploy their apps and support runtime reconfigurability.
  - https://github.com/jijinfanhua/IPSA-ipbm
  - Contact us if there exist problems when using the target. feng-y18@mails.tsinghua.edu.cn
- Ongoing work: provide users with GUI
- IPSA demo can be seen in the poster session.

Thank you!