A Generalized Algorithm for Flow Table Optimization

Charles Marsh (BSE '15) with Prof. David Walker (Advisor)

Princeton University, Department of Computer Science

Goals

In the broader context of building high-level tools for Software Defined Networking (SDN), our goal is to allow programmers to optimize flow tables in network switches without being constrained by target hardware. We approach the problem by constructing an algorithm parameterized on the specification of the target hardware. This accomplishes the dual task of:

- **Compressing** network policies as much as possible.
- **Modularizing** the process of programming on network switches, allowing programmers to target different chipsets by merely altering function arguments.

Introduction

A network is a set of connected switches. These switches take in packets, observe the packets' *header fields*, and match them against patterns (or *predicates*) stored in their *flow tables*. Depending on the matching predicate, the switches perform actions, such as forwarding packets onward.

Switch hardware is expensive, which pushes programmers to use as few rules (i.e., as little memory) as possible.

Exact	*Range*	*Ternary*	*Prefix*
{00,01,10}	[0,2]	{*0,01}	{0*,10}

Meanwhile, flow tables can match on any combination of four predicate types: *range*, *exact*, *prefix*, and *ternary*. In industry, different switches employ different combinations of these predicates on tables of different sizes. This makes it difficult for programmers to optimize their flow tables, as solutions are often constrained by the target hardware.

Packet

Src. IP	Dst. IP
000	001

$\longrightarrow$

Flow Table

Src. IP	Dst. IP	
x 11*	11*	*Fwd*
✓ 00*	001	*Fwd*
- ***	***	*Drop*

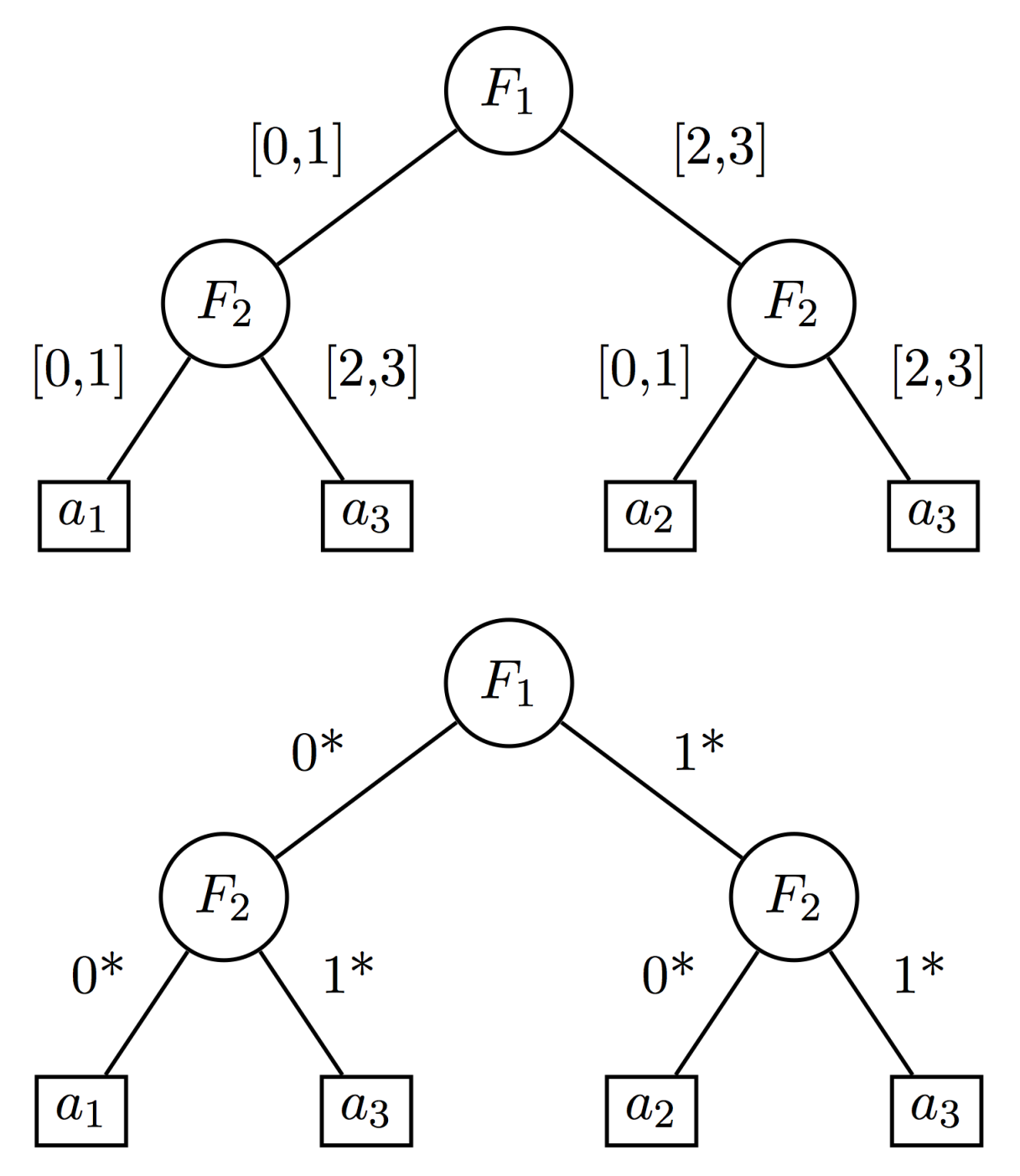
Rules are evaluated in priority order

Firewall Decision Diagrams

A key concept in compression is that of a Firewall Decision Diagram (FDD). This is essentially a tree in which each level corresponds to a field of a classifier, and the edges of the tree correspond to predicates. FDDs are generated using ranges (construction requires the intersection operation). We then convert to the appropriate predicate using efficient algorithms.

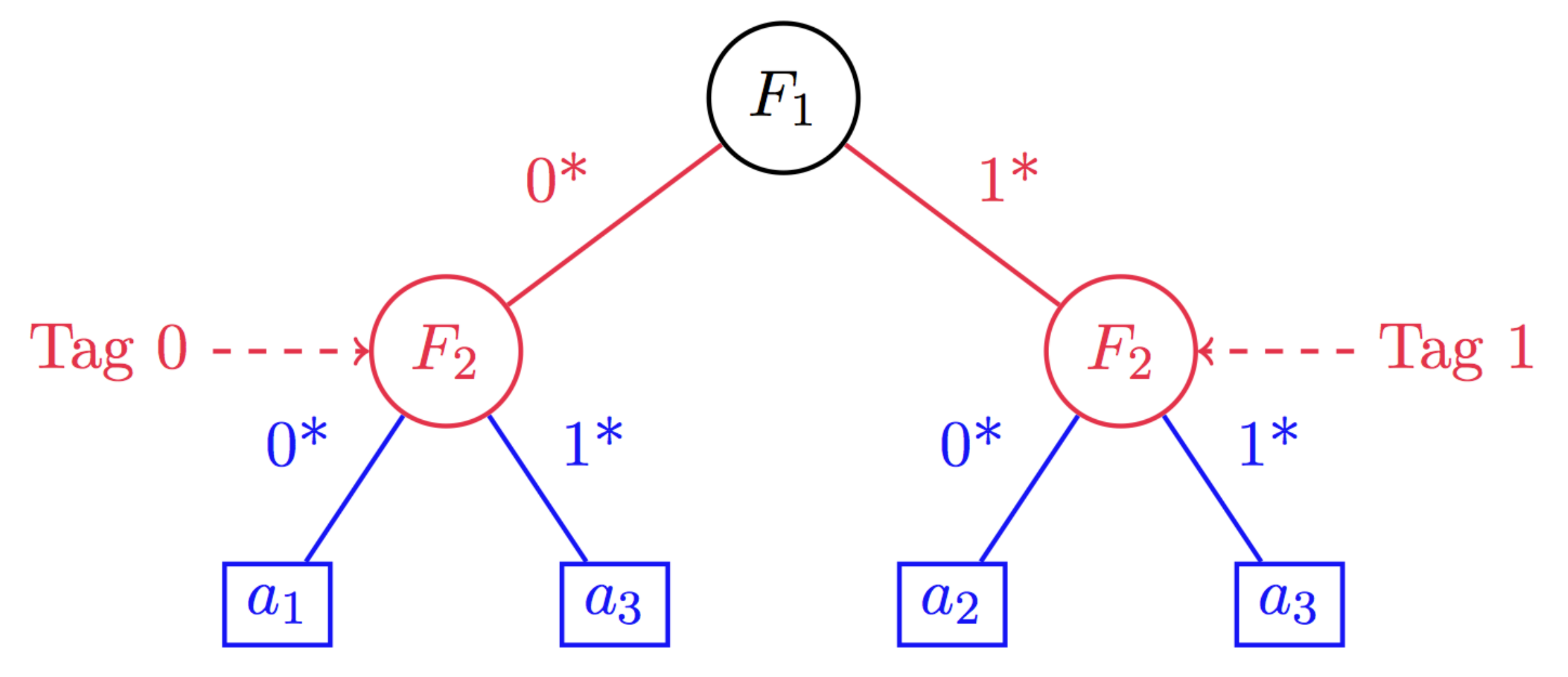
F_1	F_2	
0*	0*	a_1
1*	0*	a_2
**	**	a_3

$\longrightarrow$



Tagging

We allow the programmer to break a single flow table into multiple sub-tables. To do so, we first create the FDD for the input table, then convert all edges to the desired predicate type. Finally, we run BFS from each node at which we'd like a new table, to a depth equal to the number of fields in the desired table.



F_1	
0*	Tag 0
1*	Tag 1

Tag	F_2	
0	0*	a_1
0	1*	a_3
1	0*	a_2
1	1*	a_3

Contribution

We present an algorithm for optimizing flow tables that is **parameterized on**: the **number of tables** in a network switch; the **number of fields** in each table; the **type of predicate** used on each field; and the **widths of the fields** themselves.

Compression

We compress each node in the FDD using the appropriate one-dimensional compression algorithm, based on the type of predicate at that node:

1. **Prefix:** dynamic programming techniques
2. **Ternary:** conversion to prefixes via index swaps
3. **Range:** map to optimal scheduling problem, dynamic programming techniques
4. **Exact:** no compression possible

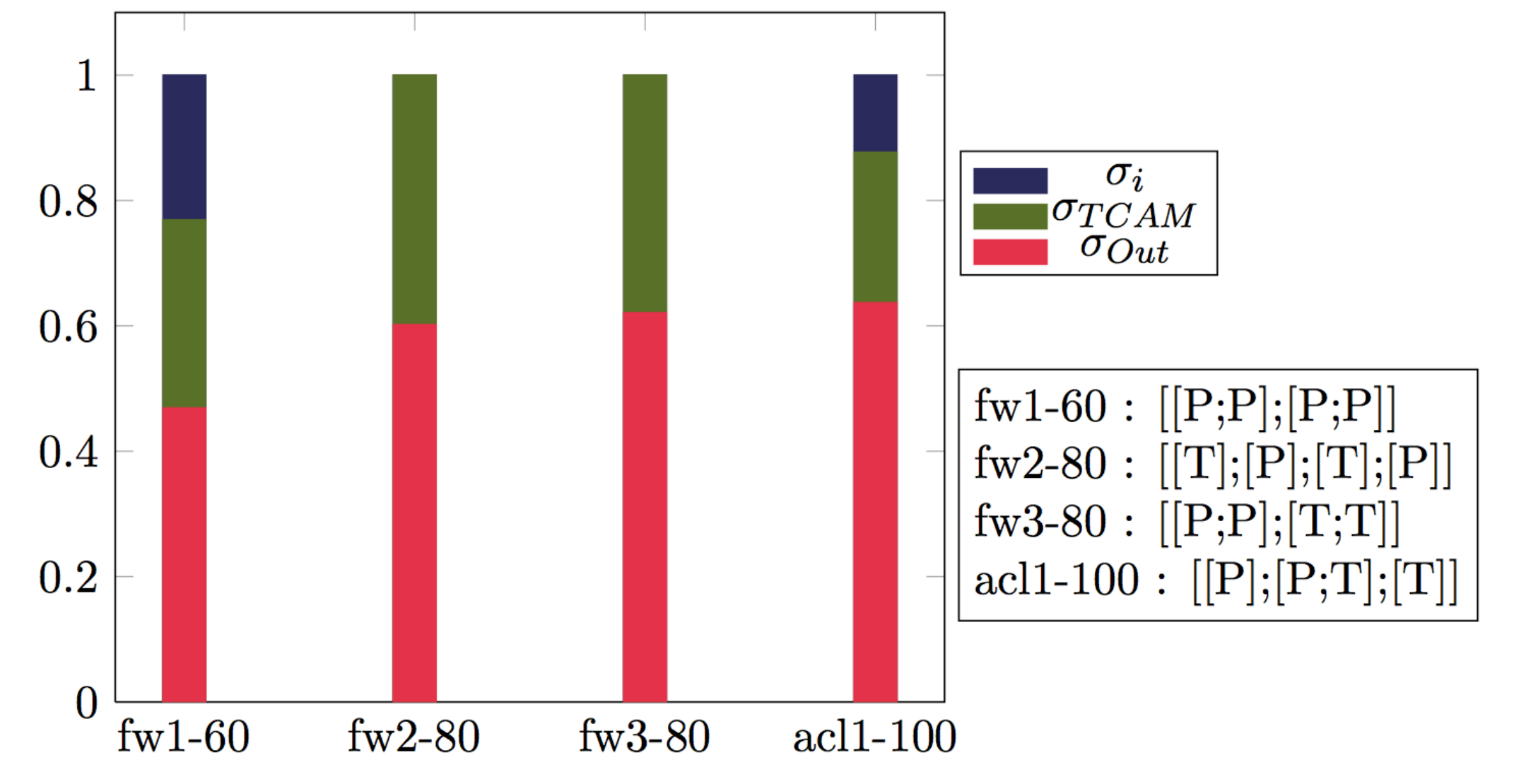
Tag	F_2	
0	0*	a_1
0	1*	a_3
1	0*	a_2
1	1*	a_3

$\longrightarrow$

Tag	F_2	
*	1*	a_3
0	**	a_1
1	**	a_2

Results

Size(P) = (# rules in P)(sum of widths of fields)

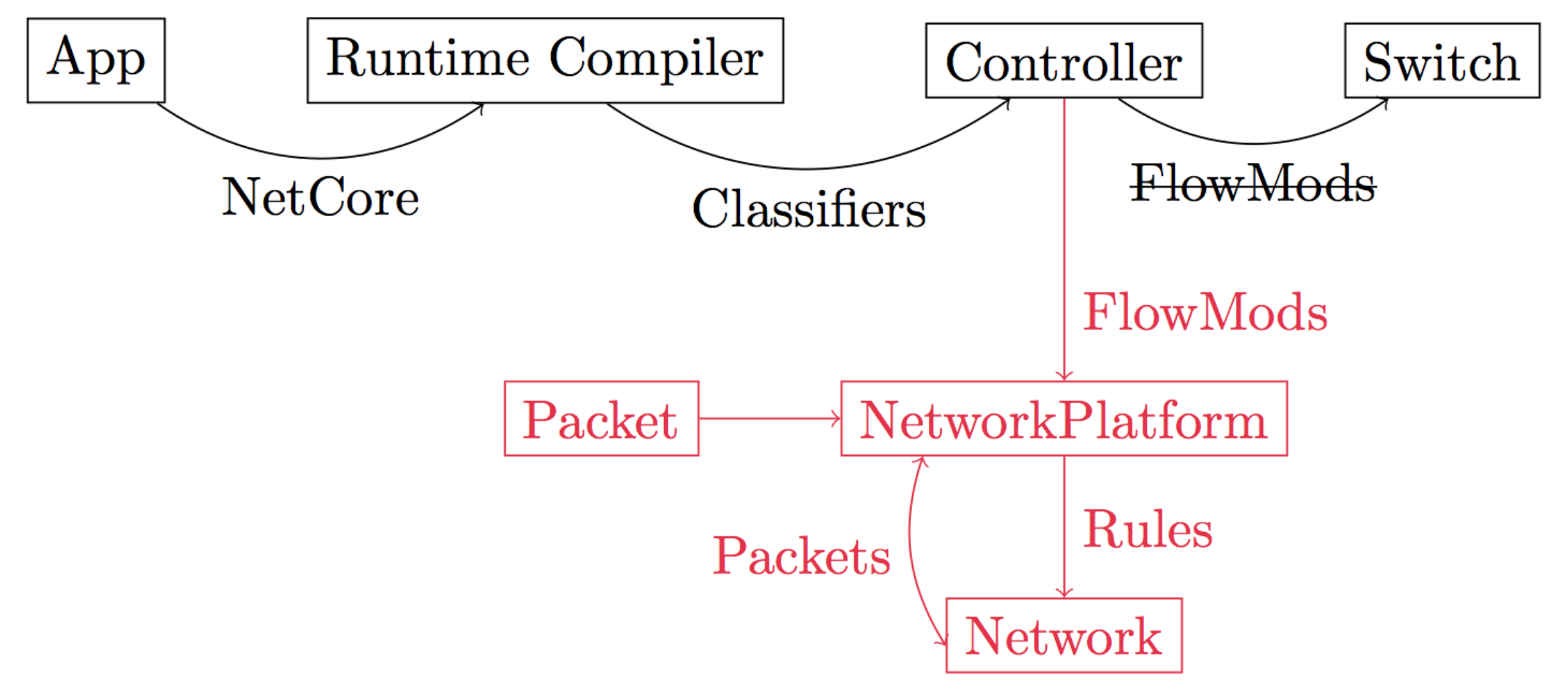


Results on ClassBench rulesets with sample user specifications. σ indicates the compression factor *initial/output*. σ_{TCAM} is a benchmark result using TCAM Razor, and σ_{Out} is our compression ratio. Ex) [[P;T]; [T; T]], indicates two tables, the first on prefix and ternary, the second solely on ternary.

Integration with Desmoines

We created a basic network simulator, integrated with the Desmoines compiler (part of the larger Frenetic Project) to simulate our algorithms. The red portion below indicates our contribution.

The Desmoines Architecture



Conclusion

We present a generalized algorithm for optimizing flow tables. Although our library is flexible as to the order of operations, our results section is based on the following (typical) workflow:

- Generate a policy using ClassBench.
- Convert this policy to a prefix-based FDD.
- Optimize this FDD locally on each node with the appropriate algorithm.
- Decompose this FDD into multiple tables using the tagging method.

This process allows the programmer to modularly adjust their programs based on the target hardware architecture with a simple change of parameters, while ensuring a reasonable level of compression.

Acknowledgements

A special thanks to Professor David Walker for being attentive and supportive throughout the process. In addition, thanks to the following individuals for their contributions: Professor Jennifer Rexford, Nanxi Kang (Princeton), Professor Nate Foster, Arjun Guha, and Mark Reitblatt (Cornell).

PRINCETON UNIVERSITY