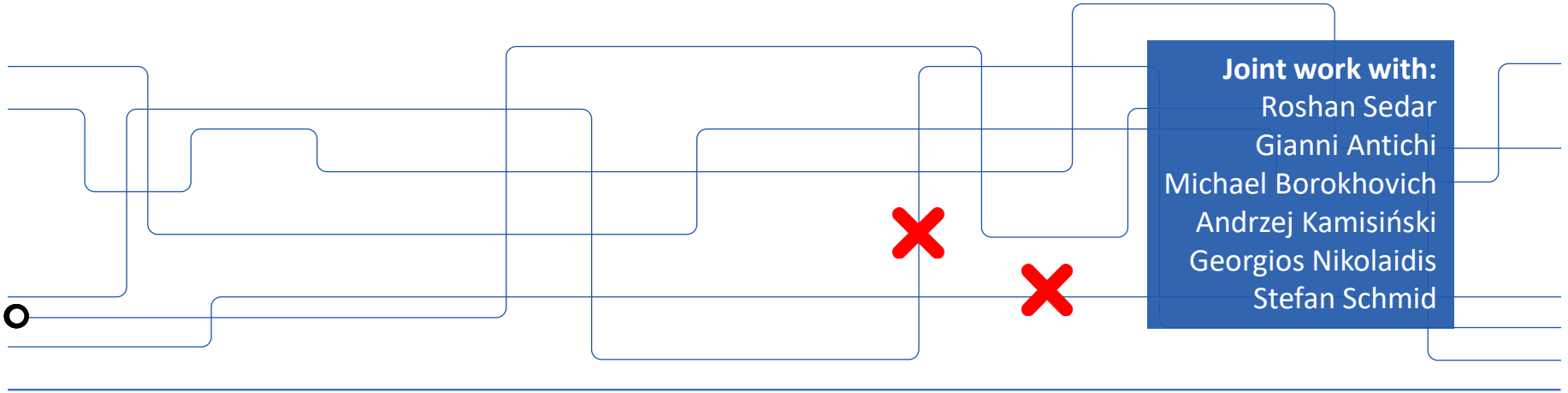
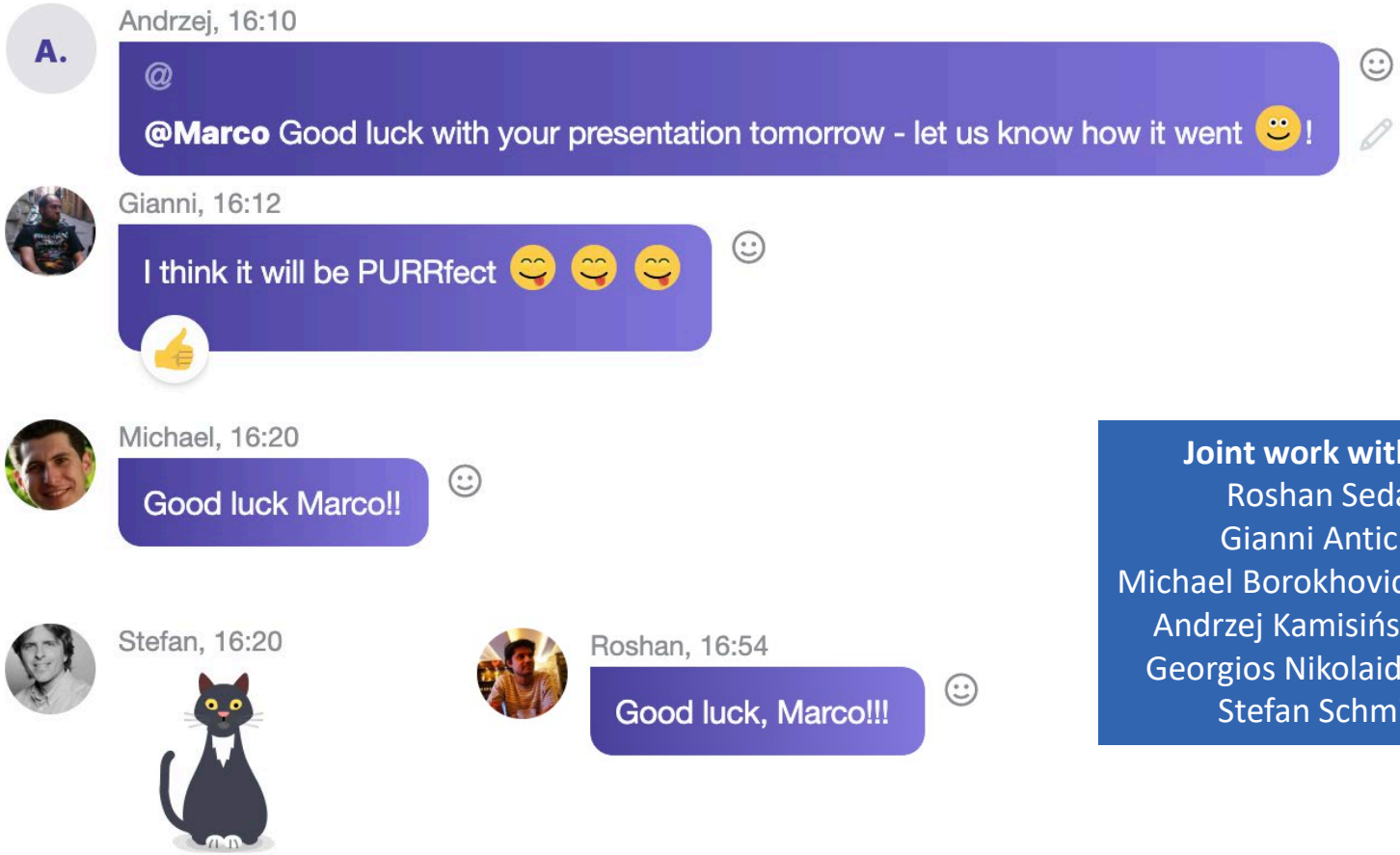
PURR: A Primitive for Reconfigurable Fast Reroute

Marco Chiesa
KTH Royal Institute of Technology
Code: bitbucket.org/marchiesa/purr



Joint work with:
Roshan Sedar
Gianni Antichi
Michael Borokhovich
Andrzej Kamisiński
Georgios Nikolaidis
Stefan Schmid

PURR: Thanks to my coauthors!



Joint work with:
Roshan Sedar
Gianni Antichi
Michael Borokhovich
Andrzej Kamisiński
Georgios Nikolaidis
Stefan Schmid

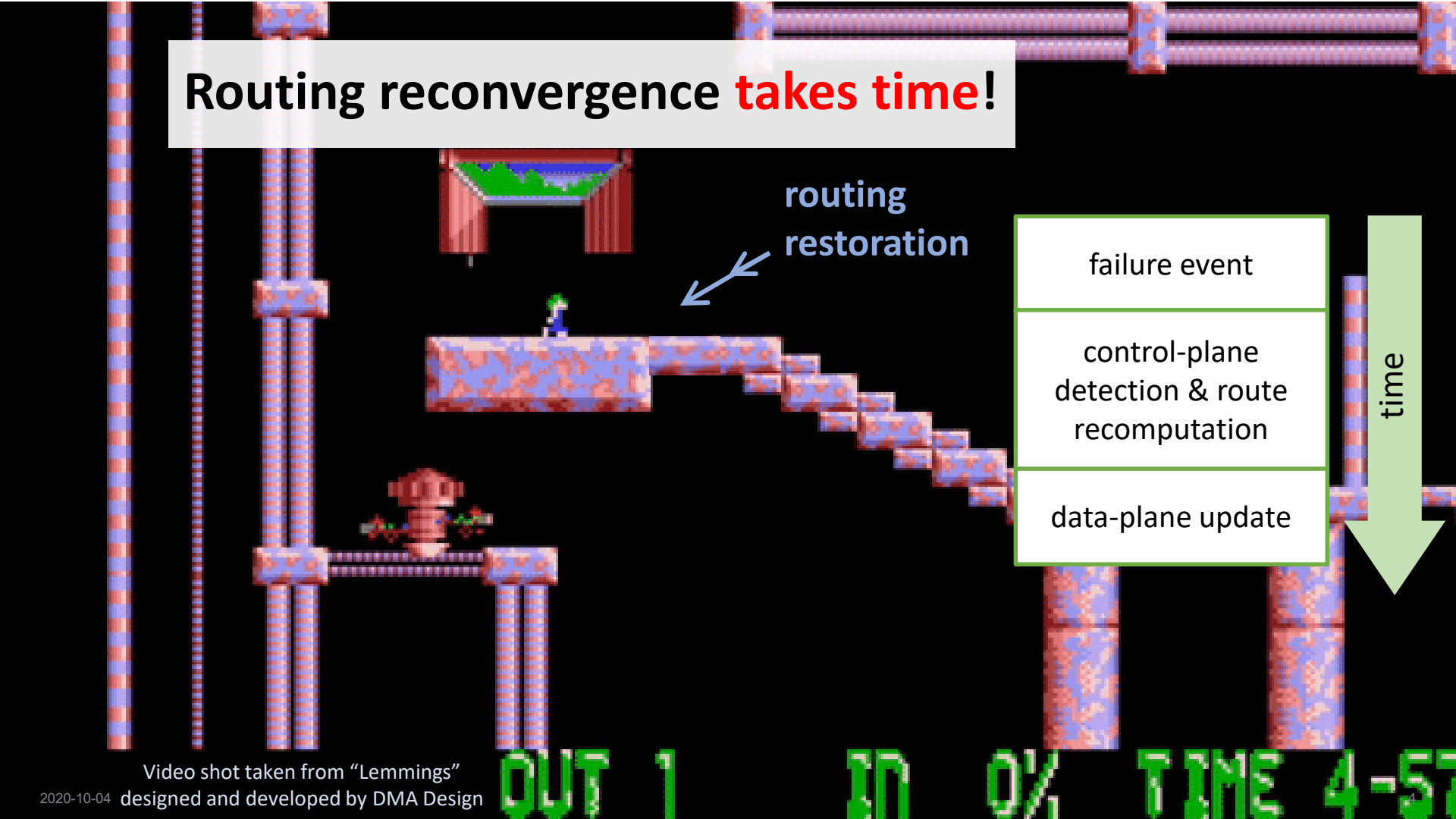
Network resilience is a key yet challenging property

High resilience

Why is it that hard?

Image credits: route by Philipp Petzka from the Noun Project

Routing reconvergence takes time!



Video shot taken from “Lemmings”
designed and developed by DMA Design

Centralized controllers make reconvergence slower

Advanced SDN routing

e.g., intent-based logically **centralized** SDN

High resilience

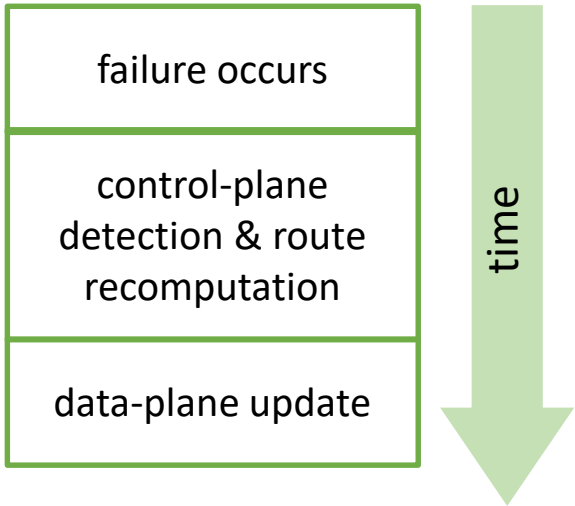


Image credits: route by Philipp Petzka from the Noun Project

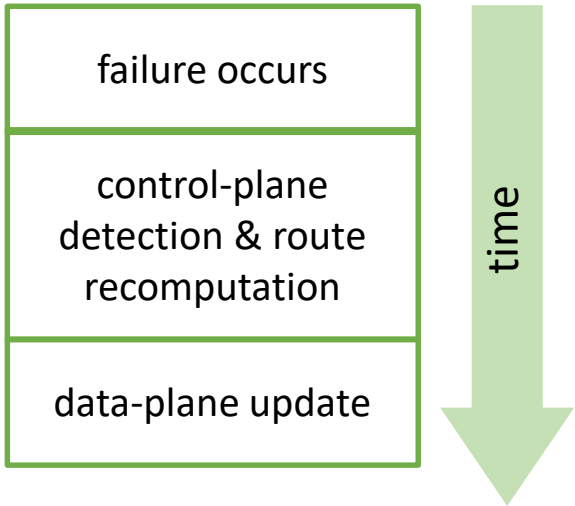
Hard to recover within 50ms!

Advanced SDN routing

e.g., intent-based logically **centralized** SDN

High resilience

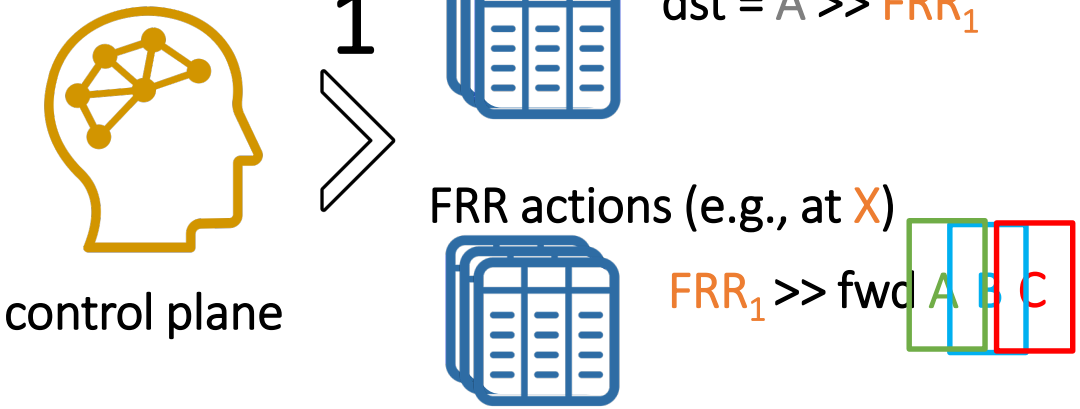
*"[carrier-grade networks] level of availability requires substantial **over-provisioning** and **fast reroute local recovery**, e.g., within 50 milliseconds" [1]*

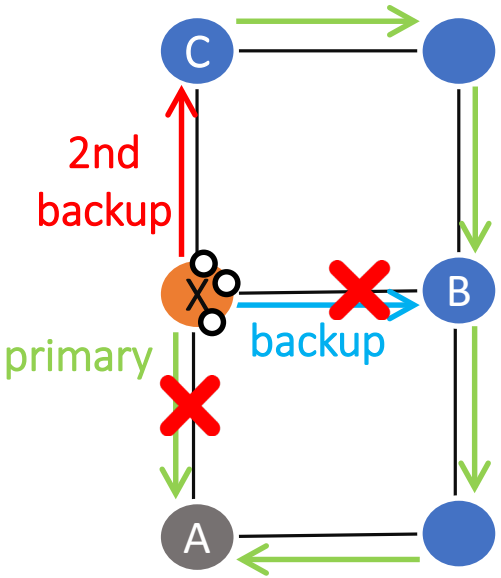


[1] On low-latency-capable topologies, and their impact on the design of intra-domain routing. In SIGCOMM 2018

Image credits: route by Philipp Petzka from the Noun Project

Fast Reroute (FRR): pre-computing failover paths





FRR entails solving two orthogonal problems:

1. control-plane: compute **network-wide** primary/backup forwarding rules

Fast Reroute (FRR): pre-computing failover paths

control plane

1

match/actions (e.g., at X)

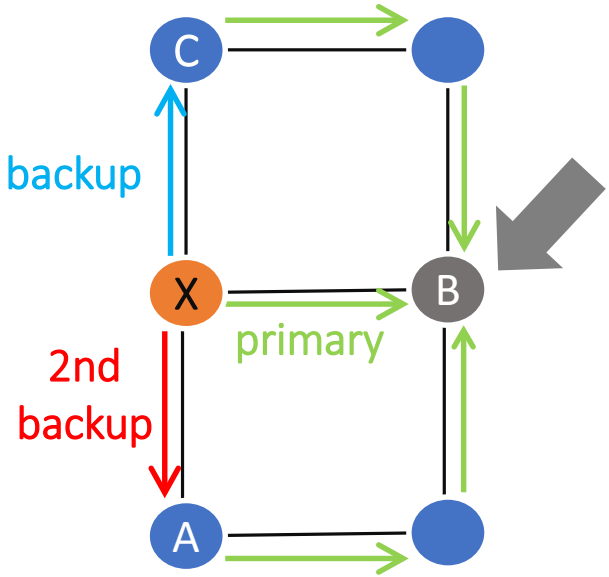
~~dst = A >> FRR_1~~

dst = B >> FRR_2

FRR actions (e.g., at X)

~~FRR_1 >> fwd A B C~~

FRR_2 >> fwd B C A



FRR entails solving two orthogonal problems:

1. control-plane: compute **network-wide** primary/backup forwarding rules

Fast Reroute (FRR): pre-computing failover paths

control plane

1

match/actions (e.g., at X)

dst = A >> FRR_1

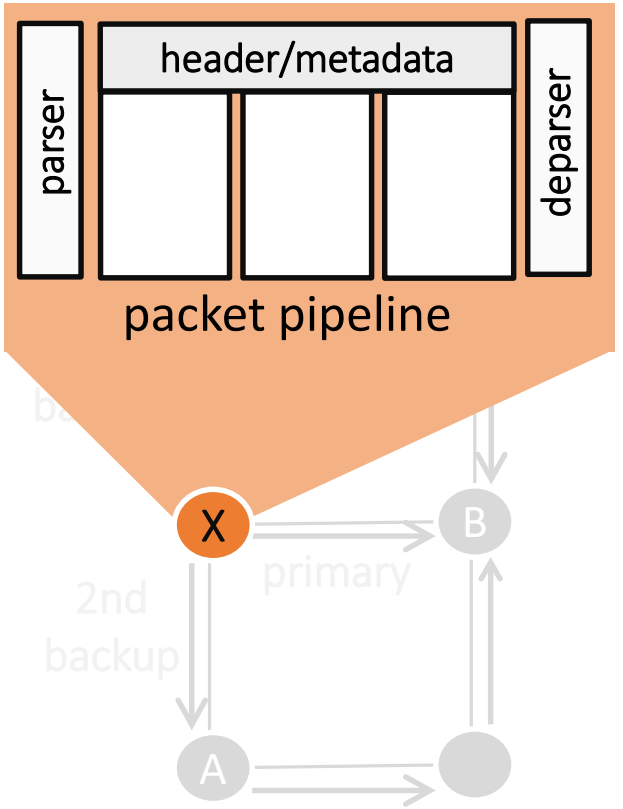
dst = B >> FRR_2

FRR actions (e.g., at X)

FRR_1 >> fwd A B C

FRR_2 >> fwd C B A

2

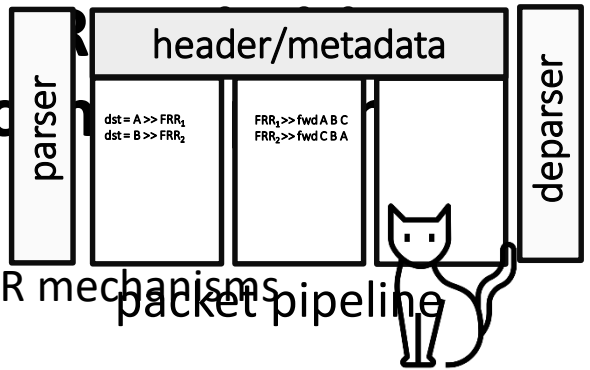


FRR entails solving two orthogonal problems:

1. control-plane: compute **network-wide** primary/backup forwarding rules

2. data-plane: support conditional forwarding in each **switch**

The goal of this talk: implement a R that minimizes pipeline resource c



PURR: A **building block** for implementing arbitrary FRR mechanisms

2

Input

FRR actions (e.g., at X)

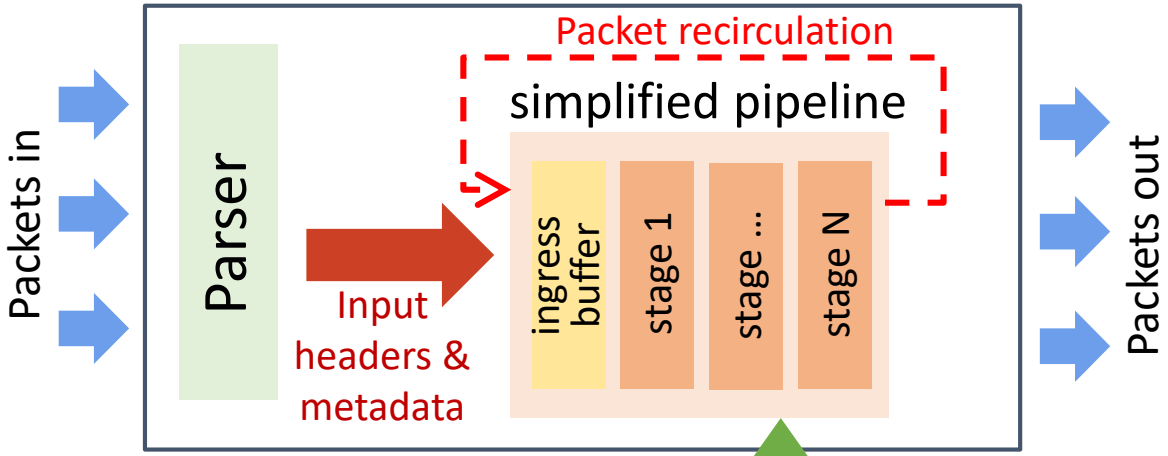
FRR_1 >> fwd A B C

FRR_2 >> fwd C B A

Output

Cat by dDara from the Noun Project

PURR applies to P4 programmable switches



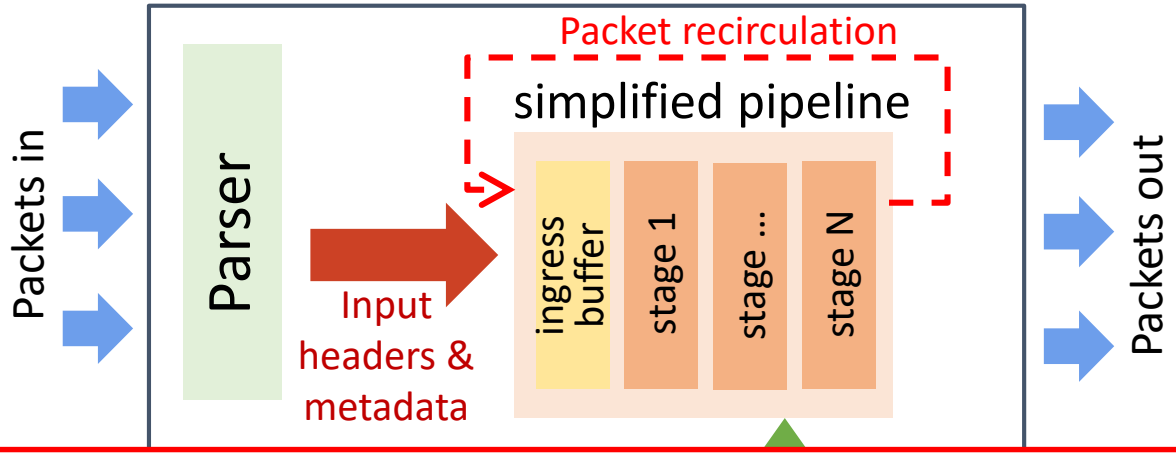
Resources:

- SRAM (exact matches), *e.g.*, 2M entries
- TCAM (wildcard matches), *e.g.*, 100K entries
- ALU

Runtime P4
(Control plane)

Image credits: route by Philipp Petzka from the Noun Project

PURR applies to P4 programmable switches



How do we realize FRR in P4?

No P4 built-in FRR primitive but…

a **programmable** pipeline: many approaches are possible!

Image credits: route by Philipp Petzka from the Noun Project

First approach: packet recirculation

Input

FRR_1 = 1 2 3 4

match	action	
tag	fwd →	write & recirculate ↻
1	~~1~~	tag := 2
2	~~2~~	tag := 3
3	3	tag := 4
4	4	-

port 1 fails

port 2 fails

throughput reduction
latency increase

FRR recirculation has high memory occupancy

Input

FRR_1 = 1 2 3 4

FRR_2 = 4 3 2 1

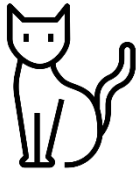
match	action	
tag	fwd →	write & recirculate ↻
1	1	tag := 2
2	2	tag := 3
3	3	tag := 4
4	4	-

port 4 fails

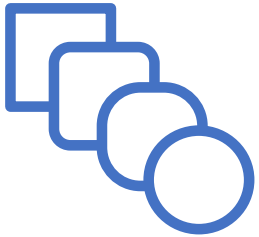
port 3 fails

throughput reduction
latency increase
high memory overhead

PURR: a primitive for reconfigurable FRR



A **building block** for implementing arbitrary FRR mechanisms



high throughput | **low forwarding latency** | **efficient reroute** | **flexibility** | **small forwarding tables**

parallel search with TCAM

model port status → flip in a single bit

support arbitrary FRR mechanisms, *i.e.*, arbitrary FRR input sequences

Cat by dDara from the Noun Project

PURR: a primitive for reconfigurable FRR

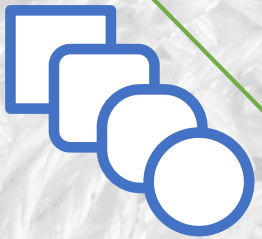
A **building block** for implementing arbitrary FRR mechanisms

- intriguing connection to (algorithmic) "**string theory**"

high throughput

low forwarding latency

efficient reroute



flexibility

small forwarding tables

Cat by dDara from the Noun Project

PURR: Encoding FRR in the packet metadata

Input

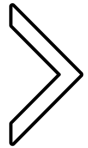
$FRR_1 = 1\ 2\ 3\ 4$

packet metadata
FRR = 1

port 1 2 3 4 is active

port status
1 1 1 1

P4 register



match	action
port status	fwd
1 * * *	1
* 1 * *	2
* * 1 *	3
* * * 1	4

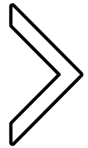
TCAM wildcard match memory

PURR: Encoding FRR in the packet metadata

Input

$FRR_1 = 1\ 2\ 3\ 4$

packet metadata	port status
FRR = 1	1 1 1 1



match	action
port status	fwd
1 * * *	1
* 1 * *	2
* * 1 *	3
* * * 1	4

TCAM wildcard match memory

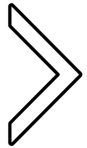
PURR: Encoding FRR in the packet metadata

Input

$FRR_1 = 1\ 2\ 3\ 4$

packet metadata	port status
FRR = 1	0 1 1 1

port 1 fails



match	action
port status	fwd
1 * * *	1
* 1 * *	2
* * 1 *	3
* * * 1	4

TCAM wildcard match memory

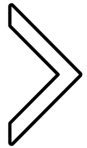
PURR: Encoding FRR in the packet metadata

Input

$FRR_1 = 1\ 2\ 3\ 4$

packet metadata	port status
FRR = 1	**0 0** 1 1

port 2 fails



match	action
port status	fwd
1 * * *	1
* **1** * *	2
* * 1 *	3
* * * 1	4

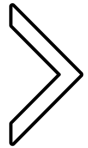
TCAM wildcard match memory

PURR: One tempting option: “Duplication” TCAM

Input

FRR_1 = 1 2 3 4 FRR_2 = 2 3 4 1

packet metadata	port status
FRR = 2	0 0 1 1



match		action
FRR	port status	fwd
1	1 * * *	1
1	* 1 * *	2
1	* * 1 *	3
1	* * * 1	4
2	* 1 * *	2
2	* * 1 *	3
2	* * * 1	4
2	1 * * *	1

similar to FRR recirculation: high memory overhead

PURR: Encoding FRR in the packet metadata

Input

$FRR_1 = 1\ 2\ 3\ 4$ $FRR_2 = 2\ 3\ 4\ 1$

packet metadata	port status
FRR = 1	1 1 1 1

frr_ports
$b_1b_2b_3b_4b_5$

Encoding FRR input:

- add a packet metadata field *frr_ports*
- map bits to the switch ports
- set bit to 1 to include a port
- set bit to 0 to skip a port

PURR: Encoding FRR in the packet metadata

Input

FRR_1 = 1 2 3 4 FRR_2 = 2 3 4 1

packet metadata	port status
FRR = 1	1 1 1 1

match	action write frr_ports
FRR = 1	1 1 1 1 0
FRR = 2	0 1 1 1 1

bit-to-port mapping
1 2 3 4 1

match		action
frr_ports	port status	fwd
1 * * * *	1 * * *	1
* 1 * * *	* 1 * *	2
* * 1 * *	* * 1 *	3
* * * 1 *	* * * 1	4
* * * * 1	1 * * *	1

PURR: Encoding FRR in the packet metadata

Input

FRR_1 = 1 2 3 4 FRR_2 = 2 3 4 1

packet metadata	port status
FRR = 1	1 1 1 1

match	action write
	frr_ports
FRR = 1	1 1 1 1 0
FRR = 2	0 1 1 1 1

bit-to-port mapping
1 2 3 4 1

match		action
frr_ports	port status	fwd
1 * * * *	1 * * *	1
* 1 * * *	* 1 * *	2
* * 1 * *	* * 1 *	3
* * * 1 *	* * * 1	4
* * * * 1	1 * * *	1

PURR: Encoding FRR in the packet metadata

Input

$FRR_1 = 1\ 2\ 3\ 4$ $FRR_2 = 2\ 3\ 4\ 1$

packet metadata	port status
FRR = 1	0 1 1 1

match	action write
	frr_ports
FRR = 1	1 1 1 1 0
FRR = 2	0 1 1 1 1

bit-to-port mapping
1 2 3 4 1

match		action
frr_ports	port status	fwd
1 * * * *	1 * * *	1
* 1 * * *	* 1 * *	2
* * 1 * *	* * 1 *	3
* * * 1 *	* * * 1	4
* * * * 1	1 * * *	1

PURR: re-cycling TCAM entries

Input

FRR_1 = 1 2 3 4 FRR_2 = 2 3 4 1

packet metadata	port status
FRR = 2	1 1 1 1

match	action write
	frr_ports
FRR = 1	1 1 1 1 0
FRR = 2	0 1 1 1 1

bit-to-port mapping
1 2 3 4 1

match		action
frr_ports	port status	fwd
1 * * * *	1 * * *	1
* 1 * * *	* 1 * *	2
* * 1 * *	* * 1 *	3
* * * 1 *	* * * 1	4
* * * * 1	1 * * *	1

The key problem: How to compute the bit-to-port mapping that minimizes memory occupancy?

FRR_1 = 2 3 1 0 FRR_2 = 0 2 1 3 FRR_3 = 3 0 2 1 FRR_4 = 1 0 2 3

bit-to-port mapping = 2 3 1 0 2 1 3

FRR_1 = 2 3 1 0

FRR_2 = 0 2 1 3

FRR_3 = 3 0 2 1

FRR_4 = 1 0 2 3

Shortest Common Supersequence (**SCS**) problem without repetitions

- SCS without repetitions is **computationally hard** (based on [2])
- Dynamic Programming (**DPSCS**) computes optimum in exponential time

 [2]T. Jiang, M. Li. On the approximation of shortest common supersequences and longest common subsequences. In Journal on Computing.

Fast-Greedy: a heuristic for solving this specific SCS

idea: remove the most frequent left-most element among the longest sequences

See the paper for:

- multi-table optimization

F_1=2 3 1 0	F_1=3 1 0	F_1=1 0	F_1=1 0
F_2=2 0 1 3	F_2=0 1 3	F_2=0 1 3	F_2=1 3
F_3=2 3 0 1	F_3=3 0 1	F_3=0 1	F_3=1
F_4=3 1 2 0	F_4=3 1 2 0	F_4=1 2 0	F_4=1 2 0
remove **2**	remove **3**	remove **0**	remove **1**

F_1=0	F_1=0	F_1= ---
F_2=3	F_2=3	F_2=3
F_3= ---	F_3= ---	F_3= ---
F_4=2 0	F_4=0	F_4= ---
remove **2**	remove **0**	remove **3**

SCS = **2 3 0 1 2 0 3**

Implementation feasibility

P4-based implementations:

- implemented different FRR mechanisms in P4 using PURR (e.g., F10 [1], arborescences [2], BFS, DFS, rotor router [3])
- compiled on Tofino

FPGA-based implementation:

- implemented PURR on the NetFPGA-SUME platform

[1] *V. Liu et al. "F10: A Fault-Tolerant Engineered Network" in NSDI 2013*
[2] *M. Chiesa et al. "On the Resiliency of Randomized Routing Against Multiple Edge Failures" in Transactions on Networking 2016*
[3] *Borokhovich et al ""Graph exploration algorithms" in HotSDN 2013*

Evaluation: How does the FRR implementation impact memory and performance?

Two subquestions:

1. How much memory does PURR save?
2. How does performance in a datacenter vary depending on how one implements a FRR primitive?

See the paper for:

- multi-table optimization
- random vs tree-based FRR sequences
- FPGA chip occupancy
- low-size FRR sequences

How much memory does PURR save? The “circular sequences” case

Input:

- switch with k ports
- 10 circular set of FRR sequences

- 10 Top-of-Rack switches in a datacenter with F10 FRR [nsdi-13]
- 10 destinations with the ″k arc-disjoint″ FRR mechanism [ton-16]

“Duplication TCAM” FRR:

- k^2 number of TCAM entries

With PURR encoding:

- *k-1* number of TCAM entries

[nsdi-13] *V. Liu et al. "F10: A Fault-Tolerant Engineered Network" in NSDI 2013*
[ton-16] *M. Chiesa et al. "On the Resiliency of Randomized Routing Against Multiple Edge Failures" in Transactions on Networking 2016*

How much memory does PURR save? The “circular sequences” case

Input:

- switch with k ports
- 10 circular set of FRR sequences

“Duplication TCAM” FRR:

- k^2 number of TCAM entries

With PURR encoding:

- *k-1* number of TCAM entries

For k = 24

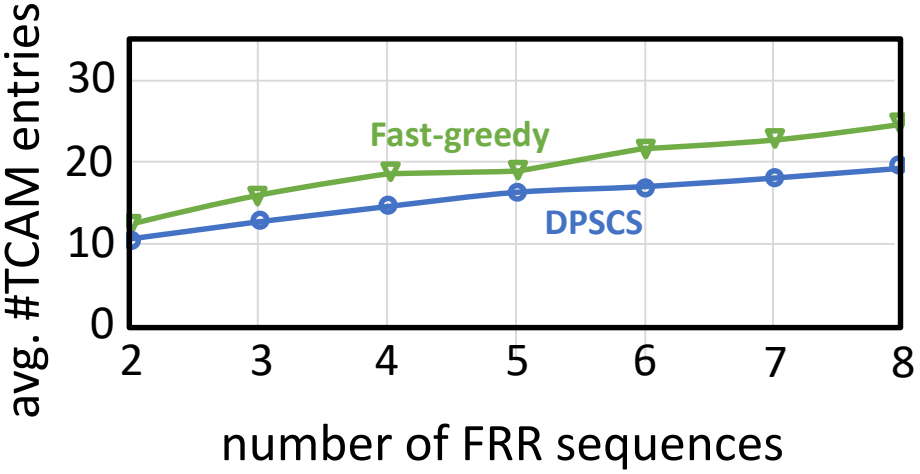
- **92%** less TCAM entries
- 470 instead of 5.760

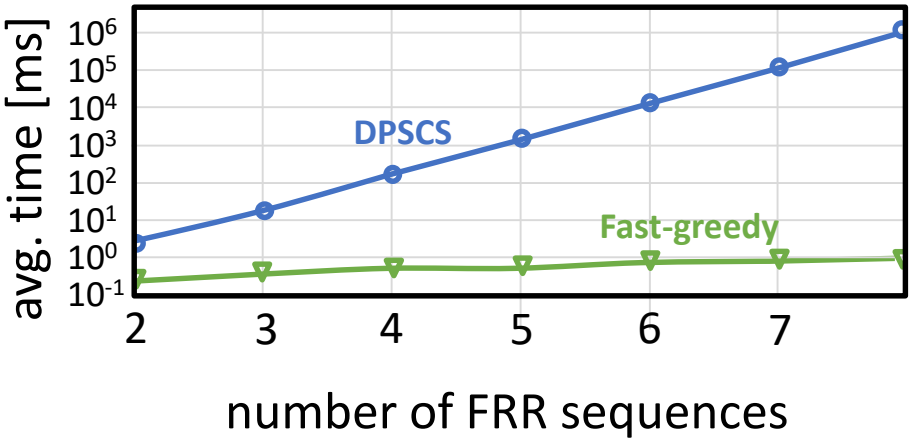
For k = 48

- **96%** less TCAM entries
- 950 instead of 23.040

[nsdi-13] *V. Liu et al. "F10: A Fault-Tolerant Engineered Network" in NSDI 2013*
[ton-16] *M. Chiesa et al. "On the Resiliency of Randomized Routing Against Multiple Edge Failures" in Transactions on Networking 2016*

How much memory does PURR save?
Fast-greedy performs close to the optimum

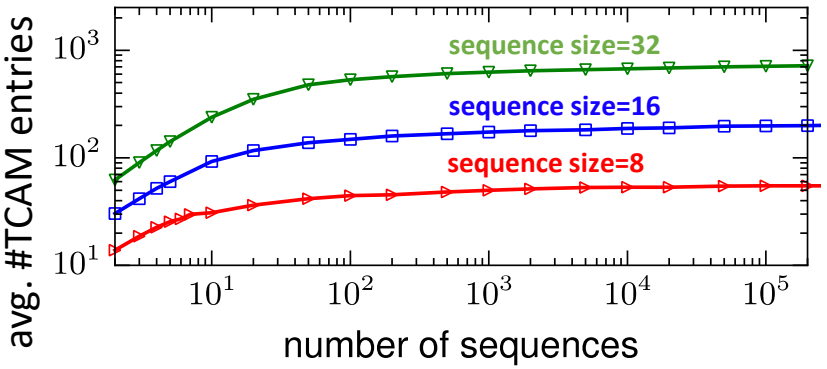


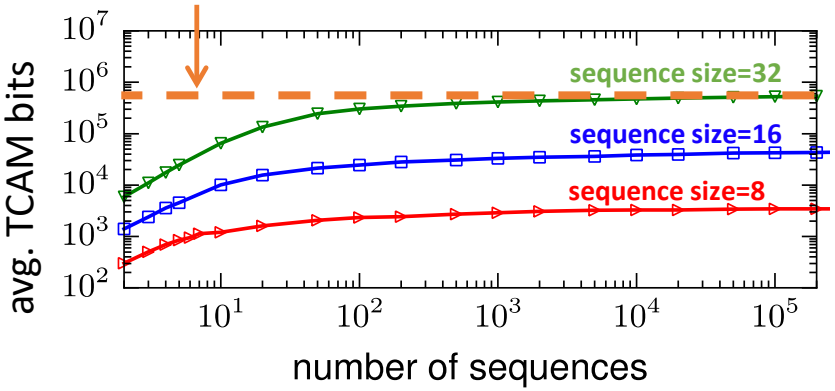


Input: randomly generated set of FRR sequences of length 7

How much memory
Fast-greedy scales t

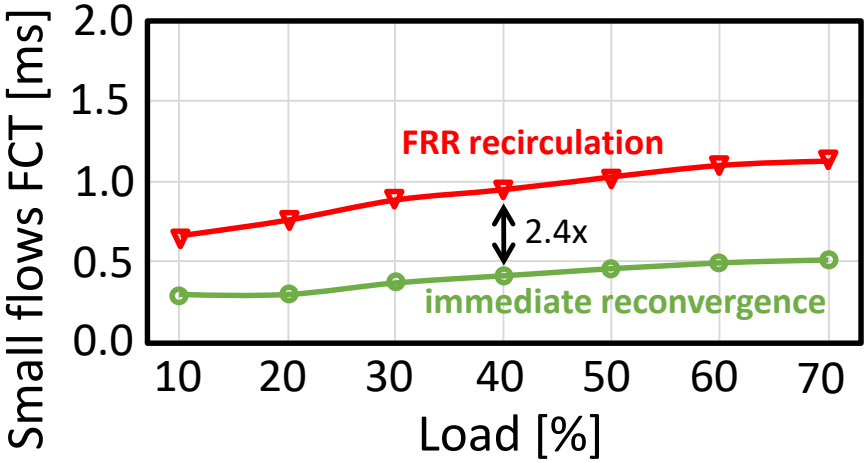
32 factorial possible FRR sequences
The “duplication” TCAM or recirculation FRR approaches would not scale

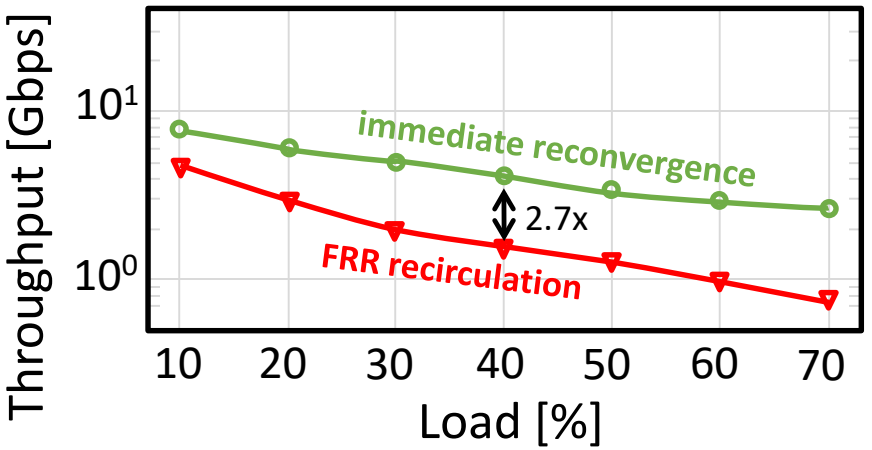




Input: randomly generated set of FRR sequences

How does FCT vary depending on the FRR primitive? PURR improves both FCT and throughput



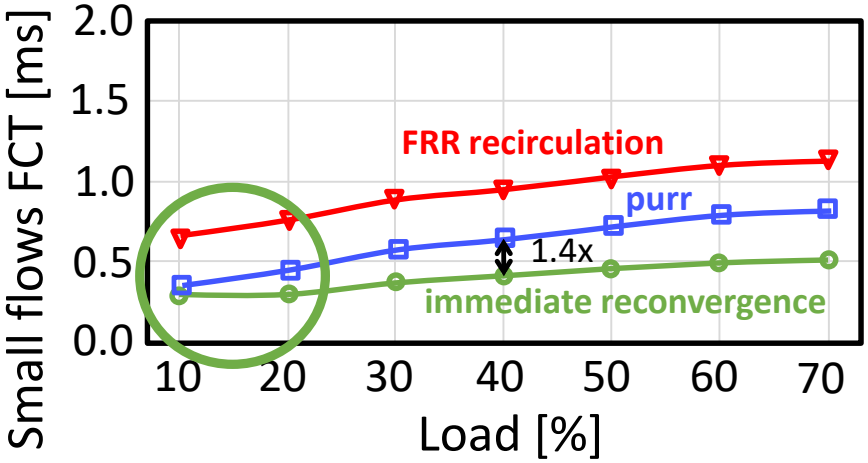


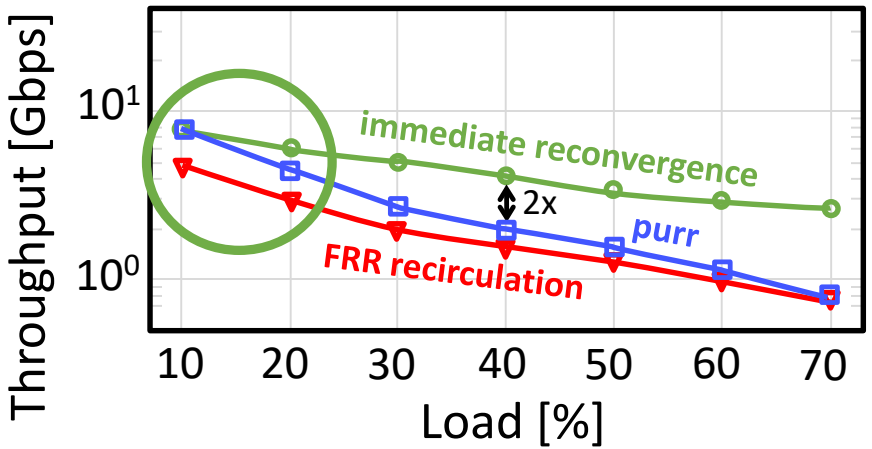
NS-3 simulations

Topology: 32-server Clos network, 10Gbps links

Workload: data-mining **Transport:** DCTCP **One link failure** at 0.5s

How does FCT vary depending on the FRR primitive? PURR improves both FCT and throughput





NS-3 simulations

Topology: 32-server Clos network, 10Gbps links

Workload: data-mining **Transport:** DCTCP **One link failure** at 0.5s

Conclusions: Keep calm and enjoy programmability

Fast Reroute is a critical functionality in today's network

- requires high throughput, low latency, fast reactiveness, small forwarding tables

P4 does not define an FRR built-in primitive

- pipeline compilers and control-plane must program the P4 pipeline

PURR: We propose a lightweight TCAM-based FRR primitive

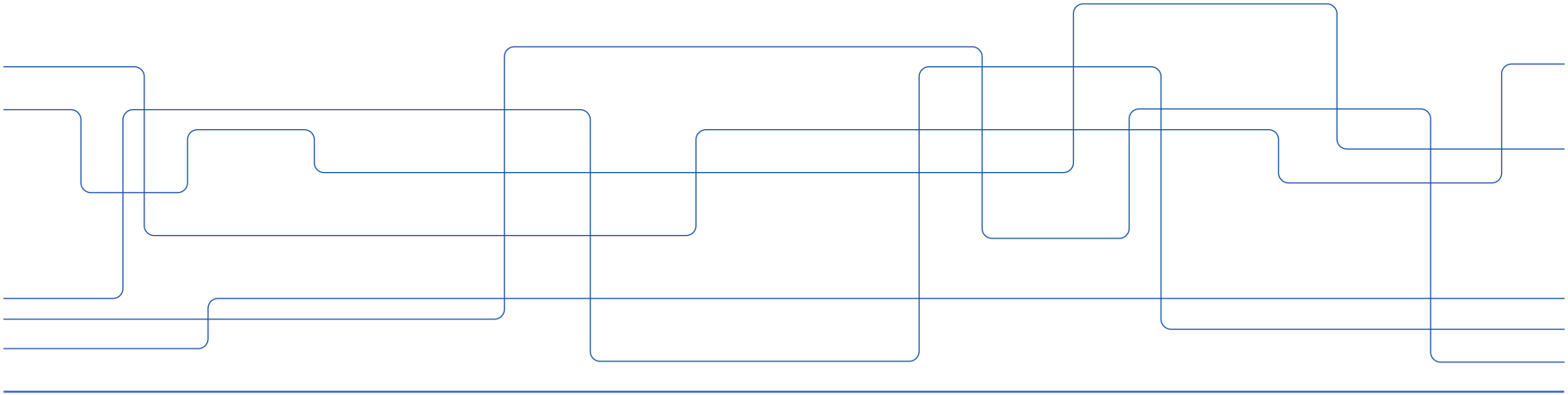
- an intriguing connection to algorithmic string theory
- no FRR-tailored hardware support
- improve performance by a factor of ~2x w.r.t. FRR recirculation

reusable

Thank you!

Marco Chiesa
KTH Royal Institute of Technology
Code: bitbucket.org/marchiesa/purr

Backup slides



Smaller sequences on switches with high-density ports

Input:

- a **64-port** programmable switch
- all possible **three ports FRR sequences**

“Duplication TCAM” approach:

- #TCAM entries = 64*63*62*3 = 750K
- TCAM memory = 750K * (20 + 3) = 17.2Mb

FRR recirculation:

- #TCAM entries = 64*63*62 = 250K
- TCAM memory = 250K * (17+3) = 5Mb

PURR encoding approach:

- #TCAM entries < (64*3) = 192
- TCAM memory < 192 * (64*4) = 50Kb

more than 99% memory reduction!

Three naïve approaches to implementing FRR

	FRR recirculation	FRR sequential	FRR parallel duplication
throughput	low	high	high
latency	low	low	high
reactiveness	low	low	high
memory	high	high	high
flexibility	high	high	high

Second approach: sequential search

Input

FRR_1 = 1 2 3 4 FRR_2 = 2 3 4 1 FRR_3 = 3 4 1 2 FRR_4 = 4 1 2 3

match	action
FRR = 1	fwd(1)

match	action
FRR = 1	fwd(2)

match	action
FRR = 1	fwd(3)

match	action
FRR = 1	fwd(4)

Second approach: sequential search

Input

FRR_1 = 1 2 3 4 **FRR_2 = 2 3 4 1** FRR_3 = 3 4 1 2 FRR_4 = 4 1 2 3

match	action
FRR = 1	fwd(1)
FRR = 2	**fwd(2)**

match	action
FRR = 1	fwd(2)
FRR = 2	**fwd(3)**

match	action
FRR = 1	fwd(3)
FRR = 2	**fwd(4)**

match	action
FRR = 1	fwd(4)
FRR = 2	**fwd(1)**

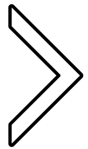
Second approach: sequential search

Input

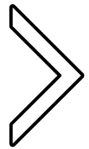
FRR_1 = 1 2 3 4 FRR_2 = 2 3 4 1 **FRR_3 = 3 4 1 2** FRR_4 = 4 1 2 3

match	action
FRR = 1	fwd(1)
FRR = 2	fwd(2)
FRR = 3	**fwd(3)**

match	action
FRR = 1	fwd(2)
FRR = 2	fwd(3)
FRR = 3	**fwd(4)**



match	action
FRR = 1	fwd(3)
FRR = 2	fwd(4)
FRR = 3	**fwd(1)**



match	action
FRR = 1	fwd(4)
FRR = 2	fwd(1)
FRR = 3	**fwd(2)**

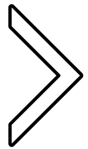
Second approach: sequential search

Input

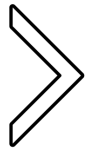
FRR_1 = 1 2 3 4 FRR_2 = 2 3 4 1 FRR_3 = 3 4 1 2 **FRR_4 = 4 1 2 3**

match	action
FRR = 1	fwd(1)
FRR = 2	fwd(2)
FRR = 3	fwd(3)
FRR = 4	**fwd(4)**

match	action
FRR = 1	fwd(2)
FRR = 2	fwd(3)
FRR = 3	fwd(4)
FRR = 4	**fwd(1)**



match	action
FRR = 1	fwd(3)
FRR = 2	fwd(4)
FRR = 3	fwd(1)
FRR = 4	**fwd(2)**



match	action
FRR = 1	fwd(4)
FRR = 2	fwd(1)
FRR = 3	fwd(2)
FRR = 4	**fwd(3)**

A sequential search wastes hardware resources

Input

FRR_1 = 1 2 3 4 **FRR_2 = 2 3 4 1** FRR_3 = 3 4 1 2 FRR_4 = 4 1 2 3

match	action
FRR = 1	fwd(1)
FRR = 2	~~fwd(2)~~
FRR = 3	~~fwd(3)~~
FRR = 4	fwd(4)

port 2 fails

match	action
FRR = 1	~~fwd(2)~~
FRR = 2	~~fwd(3)~~
FRR = 3	fwd(4)
FRR = 4	fwd(1)

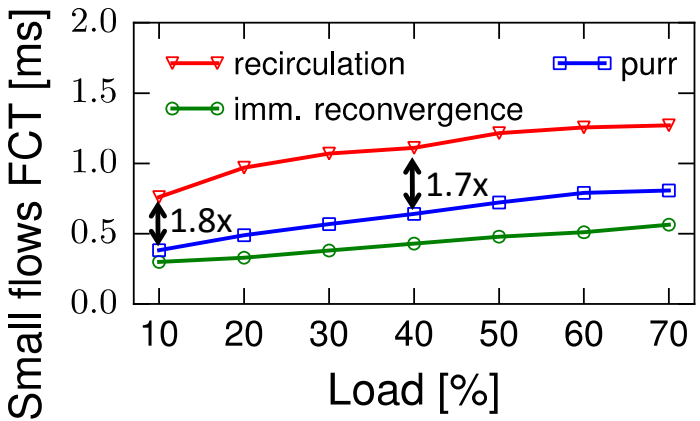
port 3 fails

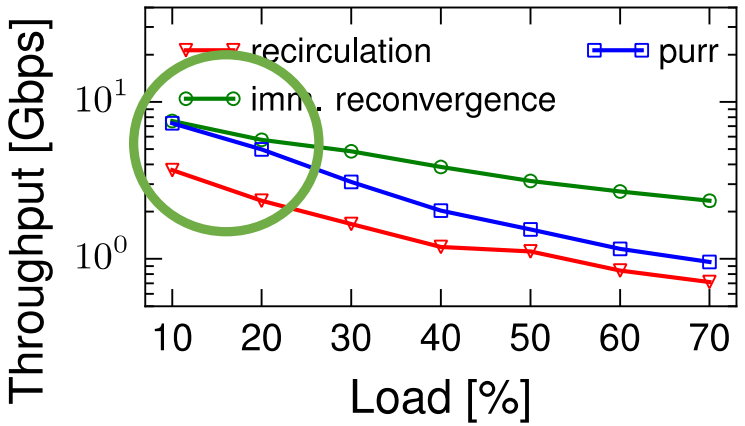
match	action
FRR = 1	~~fwd(3)~~
FRR = 2	fwd(4)
FRR = 3	fwd(1)
FRR = 4	~~fwd(2)~~

match	action
FRR = 1	fwd(4)
FRR = 2	fwd(1)
FRR = 3	~~fwd(2)~~
FRR = 4	~~fwd(3)~~

- increased latency
- waste of resources at each stage
- many updates to the forwarding table

PURR improves both FCT and throughput
Two link failures, higher gains



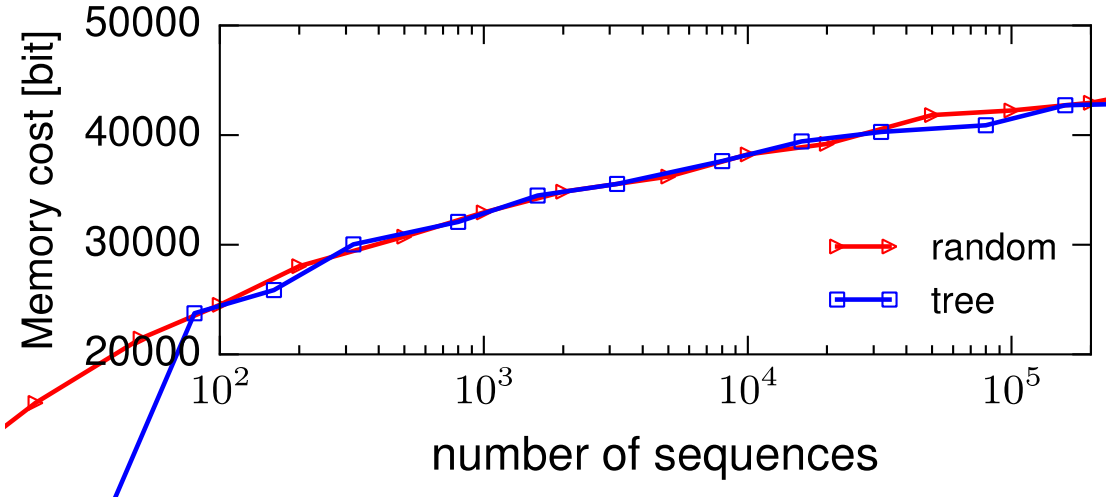


NS-3 simulations

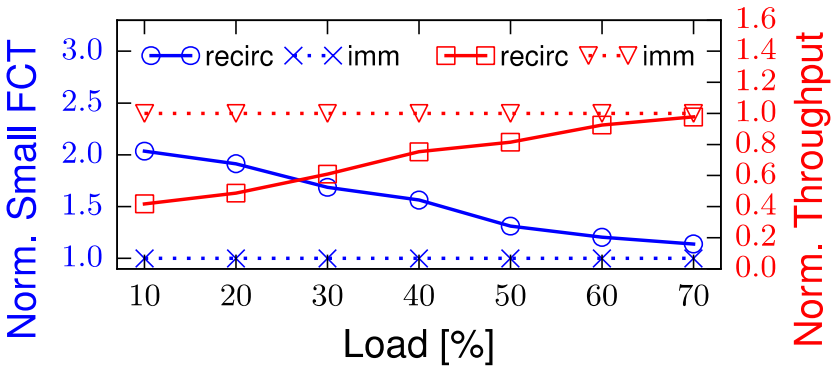
Topology: 32-server Clos network, 10Gbps links, 10μs link latency

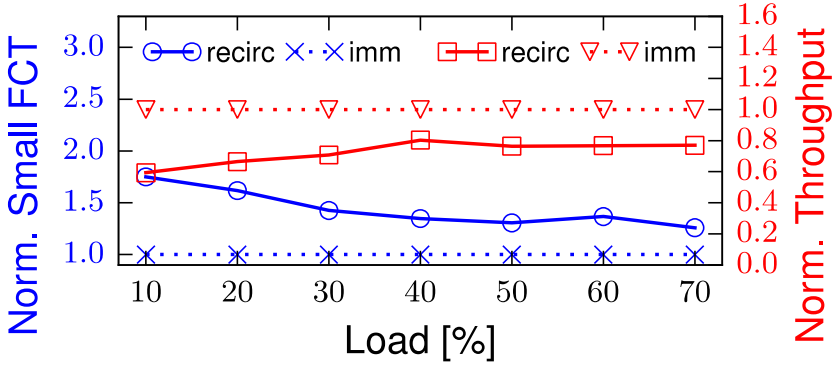
Workload: data-mining **Transport:** DCTCP

Evaluation: Random vs tree-based sequences



The web search workload





NS-3 simulations

Topology: 32-server Clos network, 10Gbps links, 10μs link latency

Workload: web-search **Transport:** DCTCP