

# Intelligent NIC Queue Management in the Dragonet Network Stack

Kornilios Kourtis

`kou@zurich.ibm.com`

Currently at: IBM Research, Zurich.

This work done while at: Systems Group, ETH Zurich.

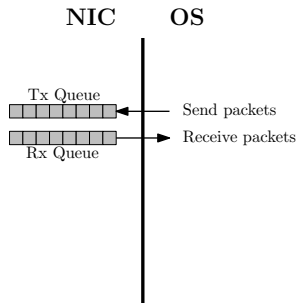
Computing Systems Research Day, NTUA, Jan. 8, 2016

# Dragonet

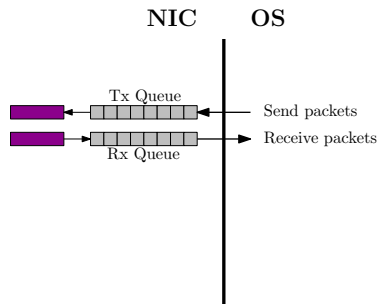


- ▶ Cores are not getting faster, the network is
- ▶ Network Interface Cards (NICs) offer rich functionality (cf. other specialized h/w such as GPUs, FPGAs, etc.)
- ▶ Dragonet is a host network stack that aims at a principled approach for **utilizing and managing NIC hardware**

# NIC hardware



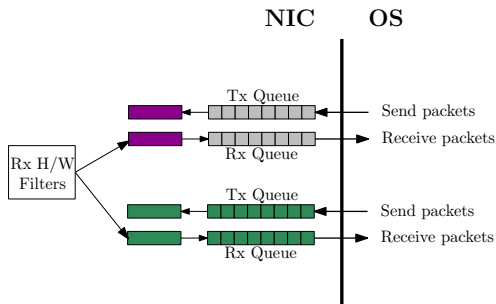
# NIC hardware



## Offload functions:

e.g.: send and receive checksum offload, segmentation offload, full protocol (TCP offload engines, IPSec)

# NIC hardware



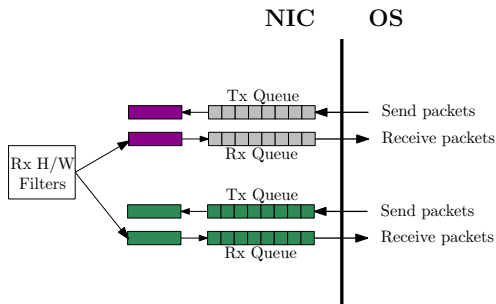
## Offload functions:

e.g.: send and receive checksum offload, segmentation offload, full protocol (TCP offload engines, IPSec)

## Multiple queues:

network stack can use multiple cores, virtualization, OS dataplanes, QoS. NIC hardware filters for steering packets into queues

# NIC hardware



## Offload functions:

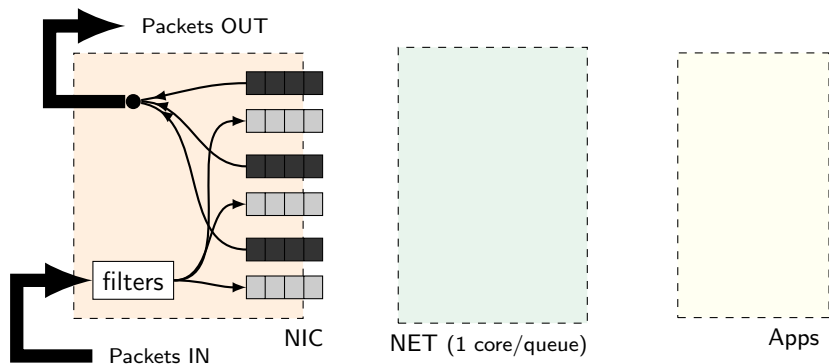
e.g.: send and receive checksum offload, segmentation offload, full protocol (TCP offload engines, IPSec)

## Multiple queues:

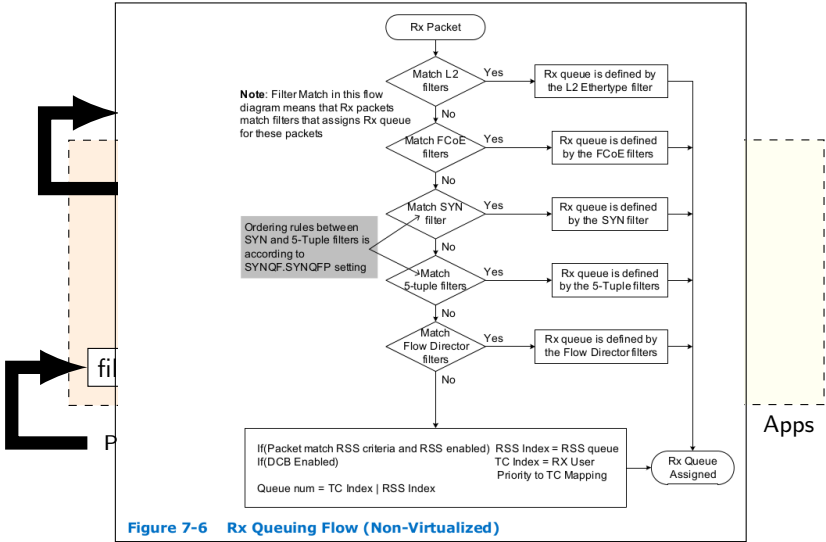
network stack can use multiple cores, virtualization, OS dataplanes, QoS. NIC hardware filters for steering packets into queues

this talk: **dealing with multiple queues**

# NIC queues and network stacks

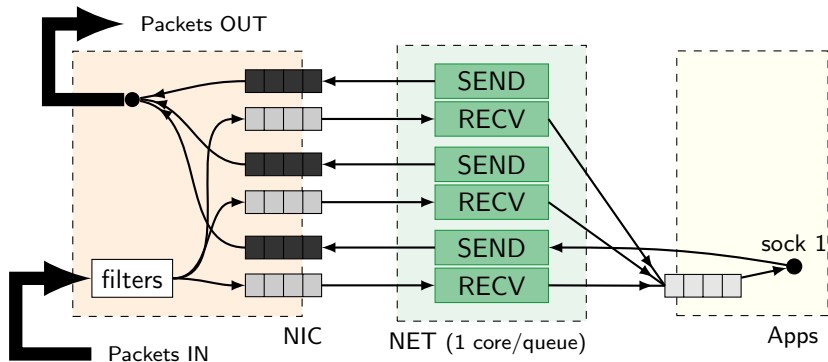


# NIC queues and network stacks

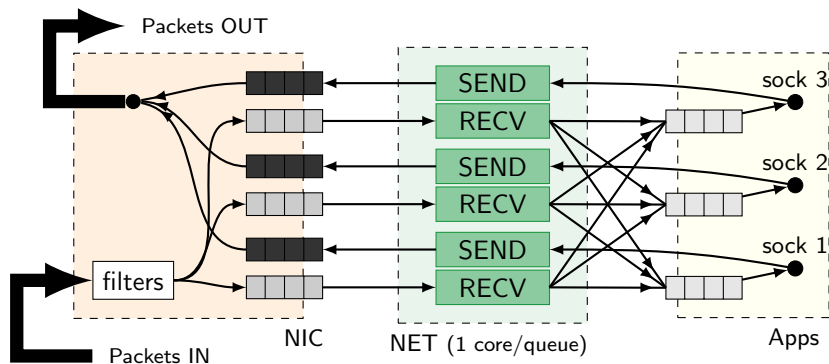




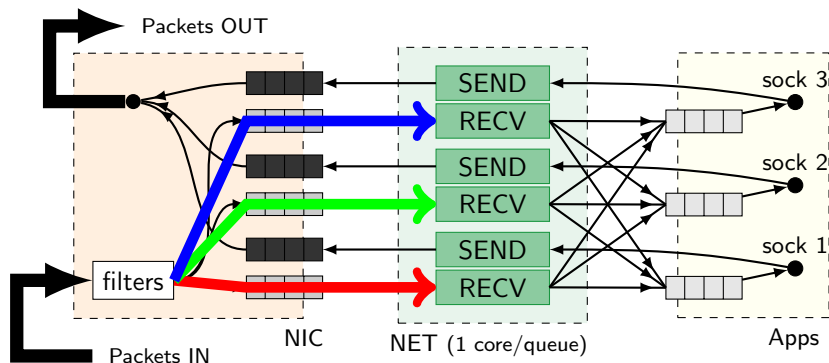
# NIC queues and network stacks



# NIC queues and network stacks

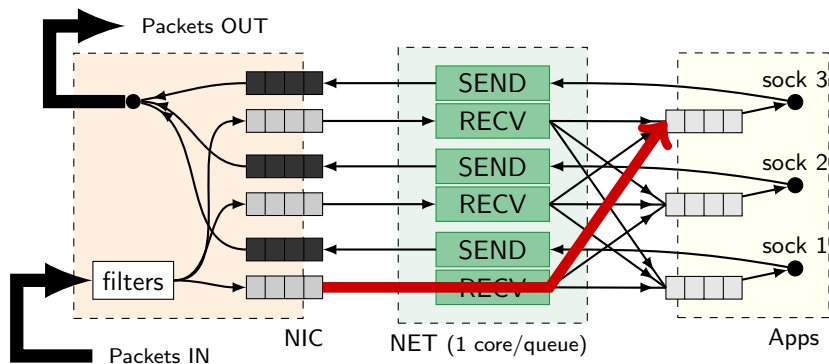


# NIC queues and network stacks



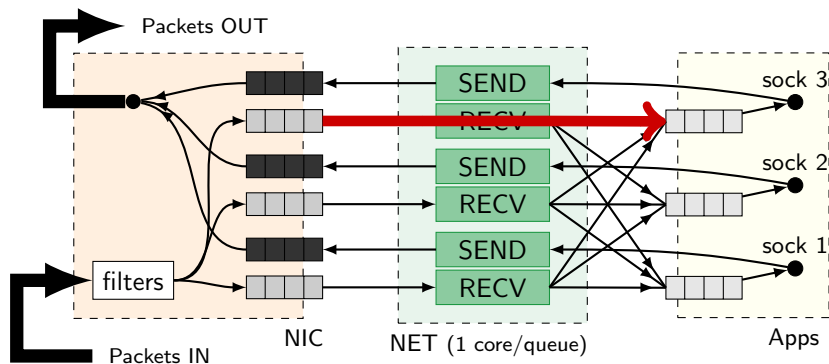
- ▶ 1st approach: **policy in the NIC**
- ▶ Receive Side Scaling (RSS)

# NIC queues and network stacks



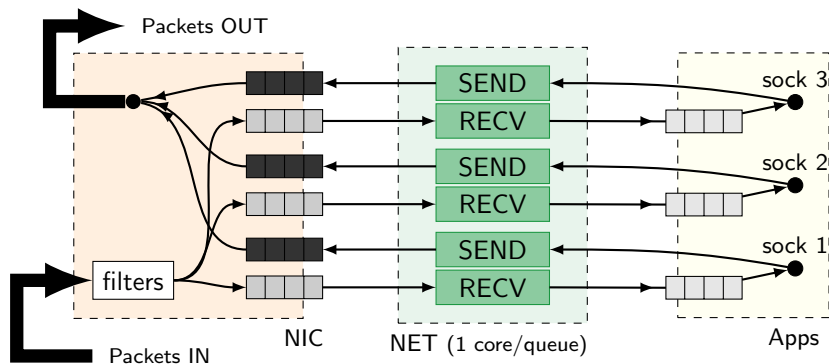
- ▶ 1st approach: **policy in the NIC**
- ▶ Receive Side Scaling (RSS)  
→ poor locality

## NIC queues and network stacks



NIC should steer packet in the core the application resides  
(aRFS in Linux, ATR in i82599 driver, Affinity-Accept [Eurosys12])

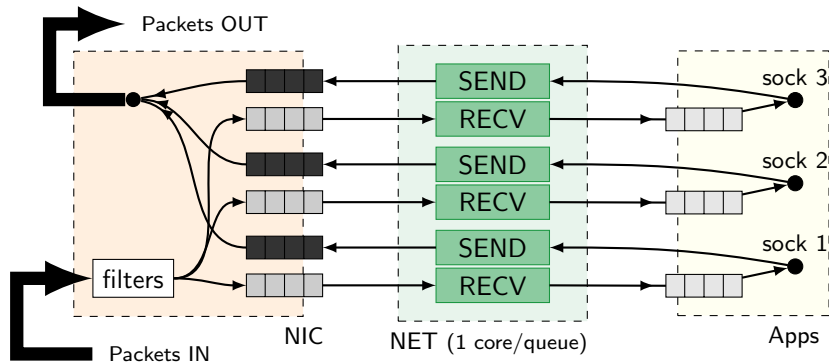
## NIC queues and network stacks



Data-plane OSeS: Arrakis [OSDI14a], IX [OSDI14b]:

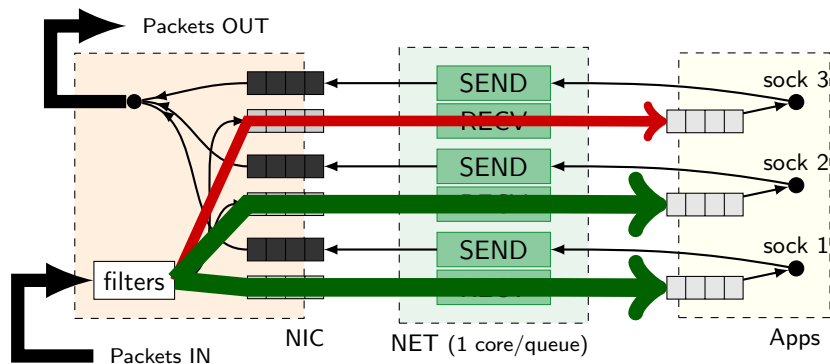
- ▶ remove OS from the data path, demultiplexing on NIC

## NIC queues and network stacks



- ▶ how do you configure the NIC?
- ▶ what happens if you run out of filters? or queues?

## NIC queues and network stacks



- ▶ how do you configure the NIC?
- ▶ what happens if you run out of filters? or queues?
- ▶ what if you want a different policy (e.g., QoS)?



## **Dragonet offers an alternative:**

- ▶ NIC queue policy in the OS (not in the NIC or the driver)
- ▶ NIC model that strives to fully capture the NIC capabilities
- ▶ NIC-agnostic policies expressed as cost functions

## Dragonet offers an alternative:

- ▶ NIC queue policy in the OS (not in the NIC or the driver)
- ▶ NIC model that strives to fully capture the NIC capabilities
- ▶ NIC-agnostic policies expressed as cost functions

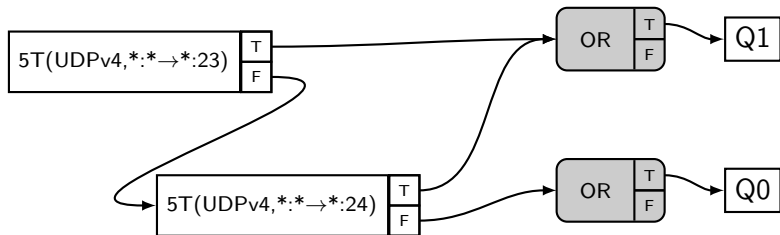
## Talk outline

- ▶ Dragonet models the NIC as a dataflow graph (called the Physical Resource Graph: **PRG**)
- ▶ Using the model to manage queues in Dragonet
- ▶ Evaluation

# NIC model

## F-nodes:

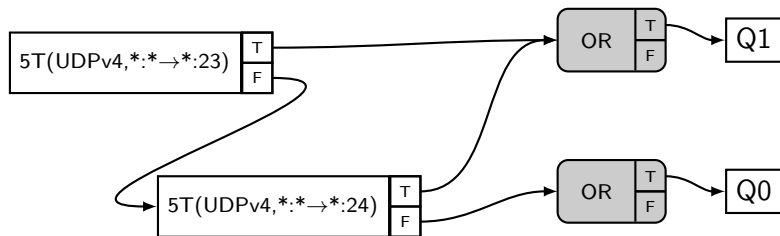
- ▶ single input
- ▶ ports, each with (possibly) multiple outputs
  - ▶ when computation is done, one port is activated
  - ▶ subsequently, nodes connected to that port are activated



# NIC model

## O-nodes:

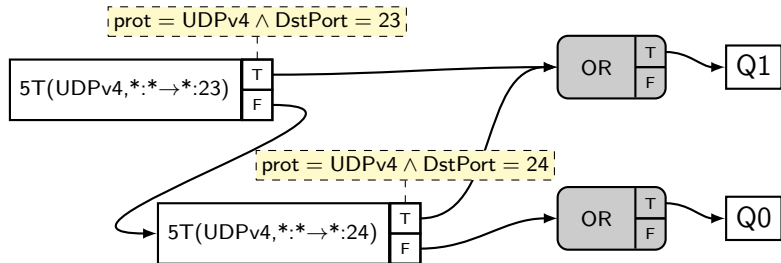
- ▶ multiple inputs:  $\{T, F\} \times$  operands
- ▶ can be short-circuited



# NIC model

## Predicates:

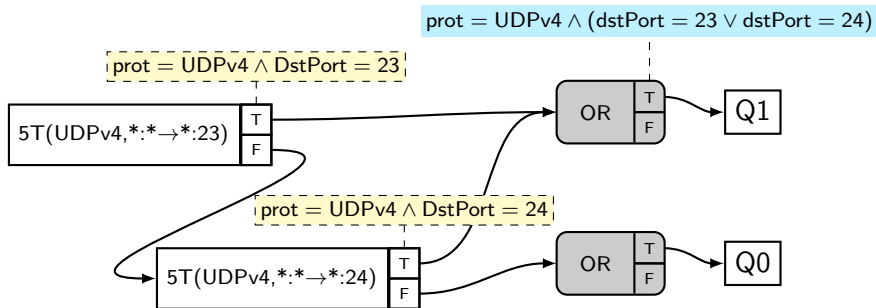
- ▶ boolean expressions about the packet  
(atoms of the form:  $k = v$ )
- ▶ each F-node port has a predicate



# NIC model

## Predicates:

- ▶ boolean expressions about the packet  
(atoms of the form:  $k = v$ )
- ▶ each F-node port has a predicate



# Modeling NIC Configuration

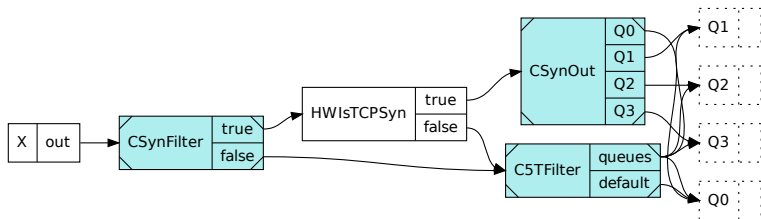
- ▶ modern NICs offer rich configuration options
- ▶ drastically modify behaviour of NIC

## **Configuration nodes (C-Nodes)**

- ▶ apply configuration value:
  - ▶ remove C-node and its edges
  - ▶ add a subgraph based on configuration value

# PRG configuration example

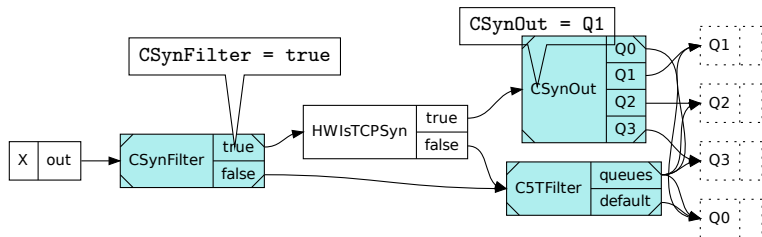
(i82599: SYN filter + 5-tuple filters)





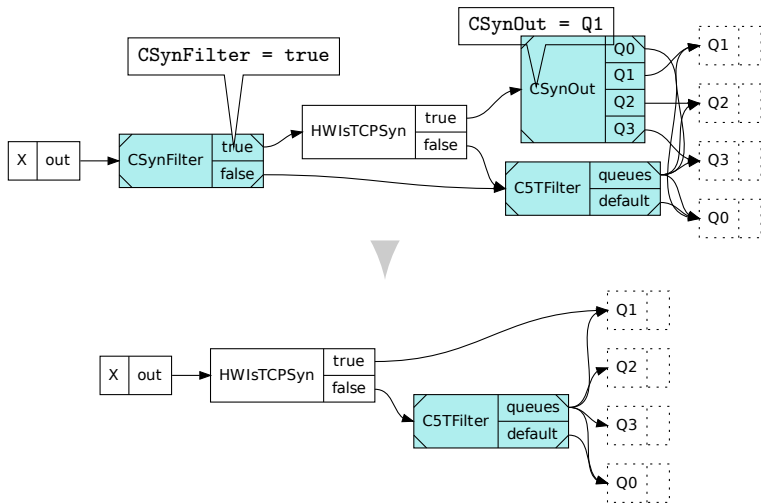
# PRG configuration example

(i82599: SYN filter + 5-tuple filters)



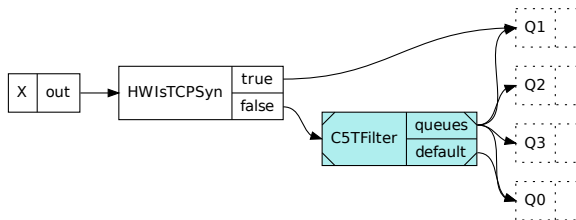
# PRG configuration example

(i82599: SYN filter + 5-tuple filters)



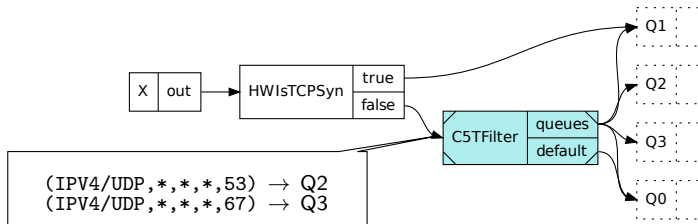
# PRG configuration example (cont'd)

(i82599: SYN filter + 5-tuple filters)



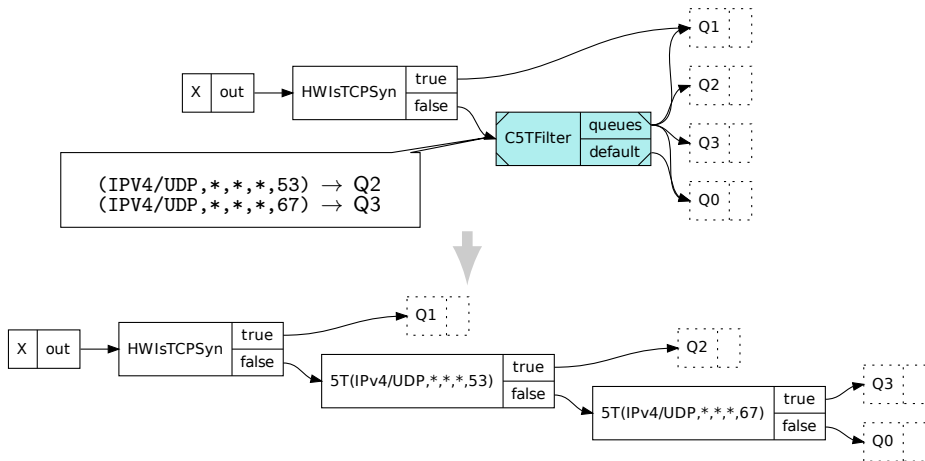
# PRG configuration example (cont'd)

(i82599: SYN filter + 5-tuple filters)



# PRG configuration example (cont'd)

(i82599: SYN filter + 5-tuple filters)



# Managing queues

Dragonet provides:

- ▶ NIC model (including configuration)
- ▶ boolean logic for reasoning

Example Policies:

1. balancing flows across queues (and subsequently cores)
2. providing performance isolation for high-priority flows

# Managing queues

Dragonet provides:

- ▶ NIC model (including configuration)
- ▶ boolean logic for reasoning

Policies are expressed as cost functions:

- ▶ input: How *flows* are mapped into *queues* ( $f \rightarrow q$ )
- ▶ output: cost

Example Policies:

1. balancing flows across queues (and subsequently cores)
2. providing performance isolation for high-priority flows

# Specifying policies with cost functions

## **Load balancing:**

- ▶ variance of number of flows per queue across queues



# Specifying policies with cost functions

## Load balancing:

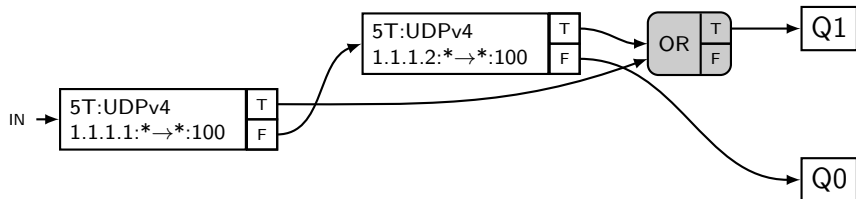
- ▶ variance of number of flows per queue across queues

## QoS/Performance isolation:

- ▶ high-priority (HP) flows, best-effort (BE) flows
- ▶ HP flows get  $N$  queues, rest to BE flows
- ▶ Each class provides its own cost function for its flows (e.g., balancing)
- ▶ reject all configurations that assign flows to queues of a different class
- ▶ accepted configurations cost:  $c = c_{BE} + c_{HP}$
- ▶ 20 lines of Haskell code

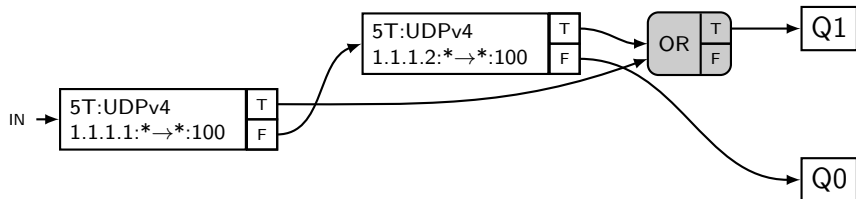
# Computing flow $\rightarrow$ queue mapping

(cost function input)



# Computing flow $\rightarrow$ queue mapping

(cost function input)



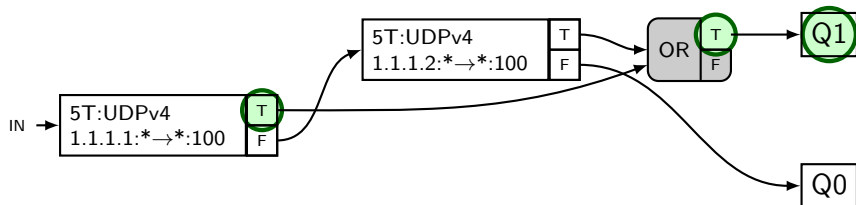
Flow: **UDPv4 / 1.1.1.1:9001  $\rightarrow$  1.1.1.42:100**

predicate:

$$\begin{aligned} & EtherType = IPv4 \quad \wedge \quad IpProt = UDP \\ \wedge \quad srcIp & = 1.1.1.1 \quad \wedge \quad srcPort = 9001 \\ \wedge \quad dstIp & = 1.1.1.42 \quad \wedge \quad dstPort = 100 \end{aligned}$$

# Computing flow $\rightarrow$ queue mapping

(cost function input)



Flow: **UDPv4** / 1.1.1.1:9001  $\rightarrow$  1.1.1.42:100

predicate:

$$\begin{aligned} & EtherType = IPv4 \quad \wedge \quad IpProt = UDP \\ \wedge \quad srcIp & = 1.1.1.1 \quad \wedge \quad srcPort = 9001 \\ \wedge \quad dstIp & = 1.1.1.42 \quad \wedge \quad dstPort = 100 \end{aligned}$$

# Searching the configuration space

$$c_o(\text{PRG}, F_{all}) = \arg \min_{c \in C} \text{cost}(\text{qmap}(\text{PRG}(c), F_{all}))$$

Performance concerns:

- ▶ full search space is too big

Improving performance:

- ▶ reduce space (e.g., NIC-specific heuristics)
- ▶ incremental computations (flows added, removed)

## Greedy search algorithm

**Input** : The set of active flows  $F_{all}$

**Input** : A cost function  $cost$

**Output** : A configuration  $c$

$c \leftarrow C_0$  // start with an empty configuration

$F \leftarrow \emptyset$  // flows already considered

**foreach**  $f$  in  $F_{all}$  **do**

    //  $CC_f$ : A set of configuration changes on  $f$

    // that incrementally change  $c$

$CC_f \leftarrow \text{oracleGetConfChanges}(c, f)$

$F \leftarrow F + f$  // Add  $f$  to  $F$

    find  $cc \in CC_f$  that minimizes  $cost(\text{qmap}(\text{PRG}(c + cc), F))$

$c \leftarrow c + cc$  // Apply change to configuration

# Greedy search algorithm

**Input** : The set of active flows  $F_{all}$

**Input** : A cost function  $cost$

**Output** : A configuration  $c$

$c \leftarrow C_0$  // start with an empty configuration

$F \leftarrow \emptyset$  // flows already considered

**foreach**  $f$  in  $F_{all}$  **do**

    //  $CC_f$ : A set of configuration changes on  $f$

    // that incrementally change  $c$

$CC_f \leftarrow \text{oracleGetConfChanges}(c, f)$

$F \leftarrow F + f$  // Add  $f$  to  $F$

    find  $cc \in CC_f$  that minimizes  $cost(\text{qmap}(\text{PRG}(c + cc), F))$

$c \leftarrow c + cc$  // Apply change to configuration

- ➔ generate configurations from flows
- ➔ oracle: NIC-specific configuration generation

## Greedy search algorithm

**Input** : The set of active flows  $F_{all}$

**Input** : A cost function  $cost$

**Output** : A configuration  $c$

$c \leftarrow C_0$  // start with an empty configuration

$F \leftarrow \emptyset$  // flows already considered

**foreach**  $f$  in  $F_{all}$  **do**

    //  $CC_f$ : A set of configuration changes on  $f$

    // that incrementally change  $c$

$CC_f \leftarrow \text{oracleGetConfChanges}(c, f)$

$F \leftarrow F + f$  // Add  $f$  to  $F$

    find  $cc \in CC_f$  that minimizes  $cost(\text{qmap}(\text{PRG}(c + cc), F))$

$c \leftarrow c + cc$  // Apply change to configuration

- generate configurations from flows
- oracle: NIC-specific configuration generation
- can be used incrementally, as flows arrive



## Greedy search algorithm

**Input** : The set of active flows  $F_{all}$

**Input** : A cost function  $cost$

**Output** : A configuration  $c$

$c \leftarrow C_0$  // start with an empty configuration

$F \leftarrow \emptyset$  // flows already considered

**foreach**  $f$  in  $F_{all}$  **do**

    //  $CC_f$ : A set of configuration changes on  $f$

    // that incrementally change  $c$

$CC_f \leftarrow \text{oracleGetConfChanges}(c, f)$

$F \leftarrow F + f$  // Add  $f$  to  $F$

    find  $cc \in CC_f$  that minimizes  $cost(\text{qmap}(\text{PRG}(c + cc), F))$

$c \leftarrow c + cc$  // Apply change to configuration

- generate configurations from flows
- oracle: NIC-specific configuration generation
- can be used incrementally, as flows arrive

# Efficient flow-to-queue map computation

**foreach**  $f$  in  $F_{all}$  **do**

...

find  $cc \in CC_f$  that minimizes  $\text{cost}(\text{qmap}(\text{PRG}(c + cc), F))$

...

**naive:**

- ▶ compute configuration ( $C$ ) from configuration changes ( $[cc]$ )
- ▶ apply  $C$  to PRG
- ▶ compute map

# Efficient flow-to-queue map computation

**foreach**  $f$  in  $F_{all}$  **do**

...

find  $cc \in CC_f$  that minimizes  $\text{cost}(\text{qmap}(\text{PRG}(c + cc), F))$

...

**incremental:**

- ▶ maintain a *partially configured* PRG
- ▶ compute flow-to-port mappings for each node
- ▶ Applying a  $cc$  adds new nodes
- ▶ propagate mappings

# Evaluation

# Implementation + Experimental setup

## Implementation

- ▶ Haskell + C
- ▶ SolarFlare SFC9020 (OpenOnload)
- ▶ Intel i82599 (DPDK)

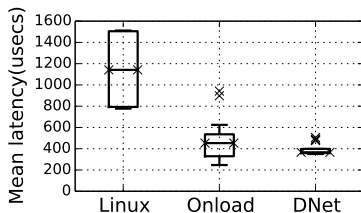
## Setup

- ▶ 10 client machines for load generation
- ▶ 1 server with 20 cores
  - ▶ 10 cores to Dragonet, 10 cores to application,
  - ▶ 10 queues.

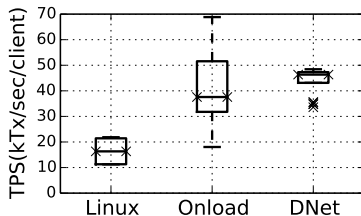
## Experiment #1: basic comparison

- ▶ **goal:** to show that Dragonet has reasonable performance under the same conditions
- ▶  $\mu$ bench: UDP echo server
- ▶ 20 netperf clients, 16 packets in-flight
- ▶ Solarflare SFC9020 (vs: Linux stack, OpenOnload user-level stack)
- ▶ Dragonet: load balancing cost function, other: RSS

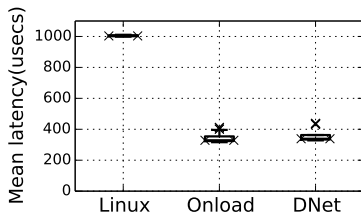
# echo server performance on the SFC9020 NIC



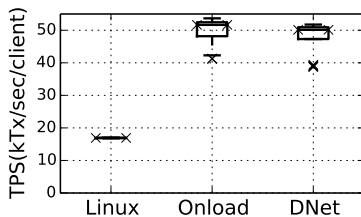
(a) Latency, 1024 bytes



(b) Throughput, 1024 bytes



(c) Latency, 64 bytes



(d) Throughput, 64 bytes

# Experiment #2

## Performance isolation/Qos

- ▶ UDP memcached, memaslap clients
  - ▶ HP clients: 4 queues, BE clients: 6 queues
  - ▶ 2 HP clients  $\times$  16 flows, 18 BE clients  $\times$  16 flows (320 flows in total)  
(*stable*)
- 
- ▶ we show here results for the Intel i82599  
(similar\* results for Solarflare SFC9020 are in the paper)



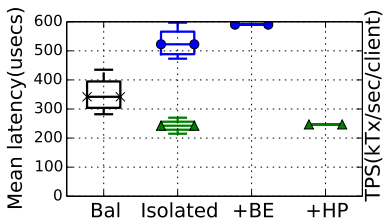
# Experiment #2

## Performance isolation/Qos

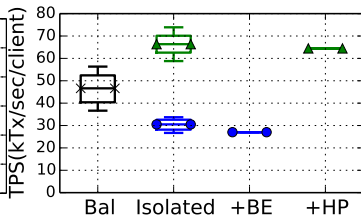
- ▶ UDP memcached, memaslap clients
- ▶ HP clients: 4 queues, BE clients: 6 queues
- ▶ 2 HP clients  $\times$  16 flows, 18 BE clients  $\times$  16 flows (320 flows in total)  
(*stable*)
- ▶ after 10secs, we start a new HP client that runs for 50secs
- ▶ after new HP is done, we start new BE client
  
- ▶ we show here results for the Intel i82599  
(similar\* results for Solarflare SFC9020 are in the paper)

# Performance Isolation

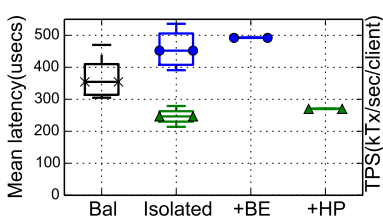
(Intel i82599)



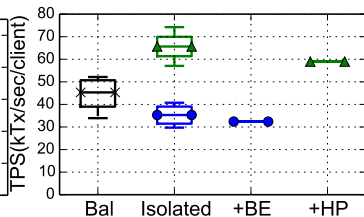
(e) Latency, 1024 bytes



(f) Throughput, 1024 bytes



(g) Latency, 64 bytes



(h) Throughput, 64 bytes

# Search cost

(10 queues, i82599 PRG)

	Naive	Incremental				
flows	full	full	+1 fl.	+10 fl.	-1 fl.	-10 fl.
10	11 ms	17 ms	2 ms	22 ms	9 $\mu$ s	23.7 $\mu$ s
100	1.2 s	0.6 s	9 ms	94 ms	74 $\mu$ s	117 $\mu$ s
250	13 s	4 s	21 ms	219 ms	190 $\mu$ s	277 $\mu$ s
500	76 s	17 s	43 ms	484 ms	382 $\mu$ s	548 $\mu$ s

## Conclusion

- ▶ Dragonet offers a systematic approach to managing queues
- ▶ Models NIC using a dataflow graph
- ▶ Expresses policy via cost-functions
- ▶ Incremental computations for improving performance
- ▶ Code available at <http://git.barrelfish.org/?p=dragonet>

# Conclusion

- ▶ Dragonet offers a systematic approach to managing queues
- ▶ Models NIC using a dataflow graph
- ▶ Expresses policy via cost-functions
- ▶ Incremental computations for improving performance
- ▶ Code available at <http://git.barrelfish.org/?p=dragonet>

**Thank you!**

- ▶ Acknowledgements: Pravin Shinde, Antoine Kaufmann, Timothy Roscoe, and the rest of the ETH Barrelfish team.