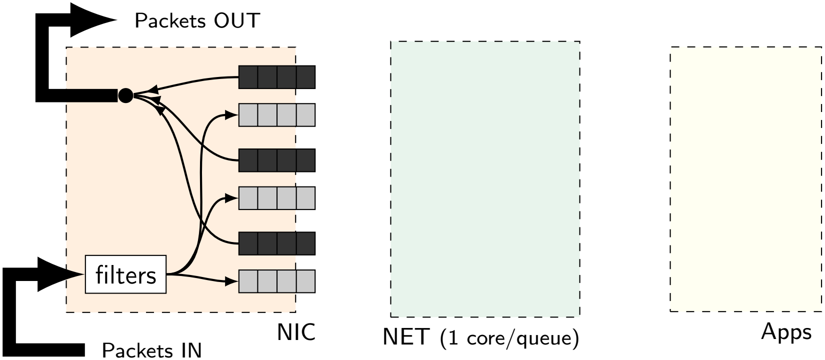
Intelligent NIC Queue Management in the Dragonet Network Stack

Kornilios Kourtis[2] Pravin Shinde[1] Antoine Kaufmann[3]
Timothy Roscoe[1]

[1] ETH Zurich [2] IBM Research [3] University of Washington

TRIOS, Oct. 4, 2015

NIC queues and network stacks



NIC queues and network stacks

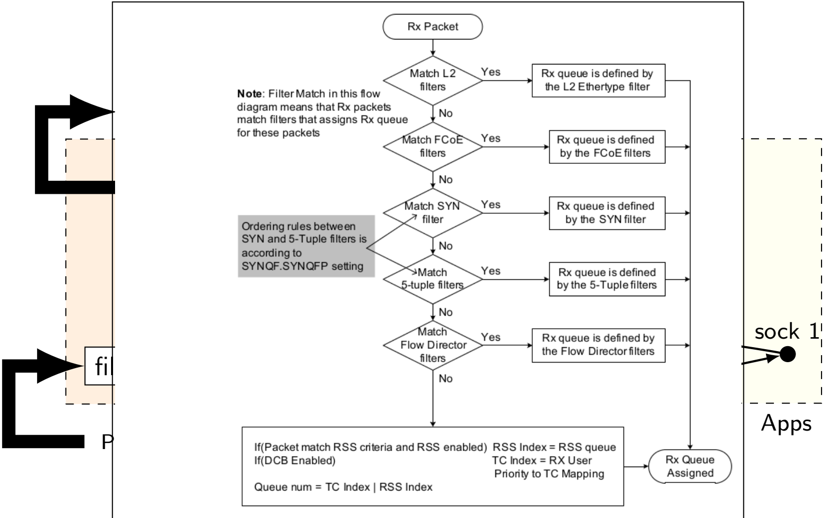
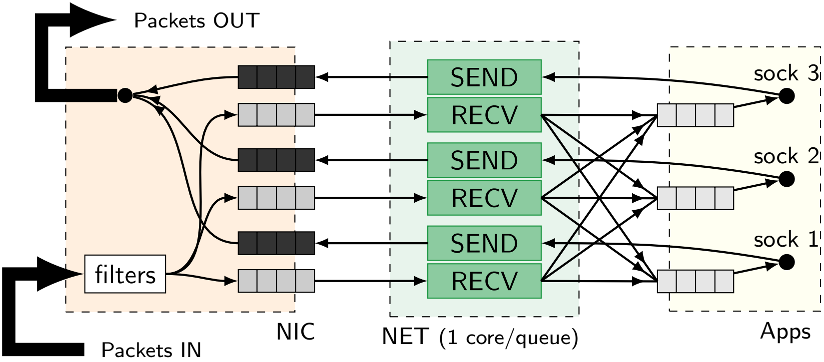
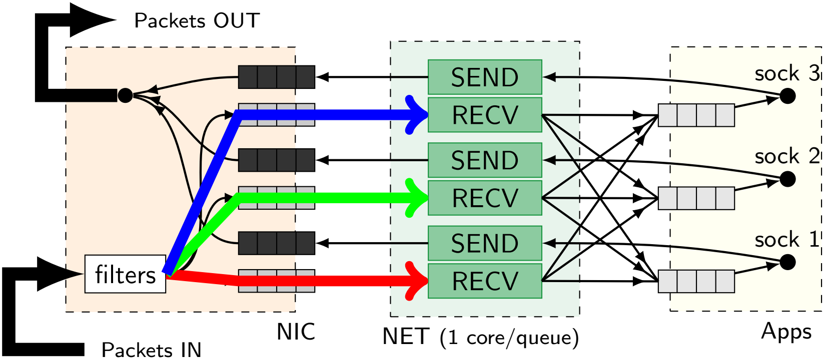


Figure 7-6 Rx Queuing Flow (Non-Virtualized)

NIC queues and network stacks

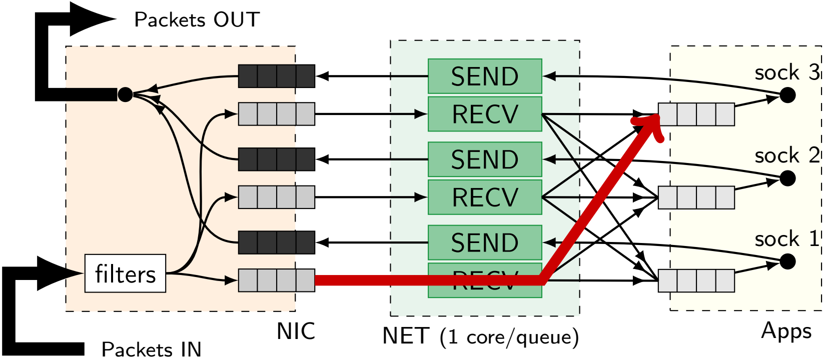


NIC queues and network stacks



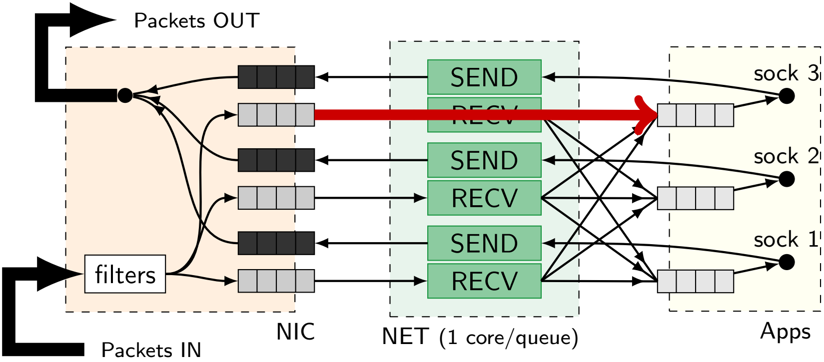
- 1st approach: **policy in the NIC**
- Receive Side Scaling (RSS)

NIC queues and network stacks



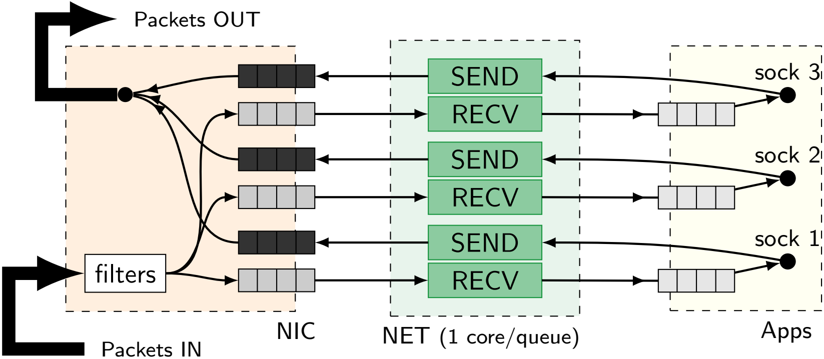
- 1st approach: **policy in the NIC**
- Receive Side Scaling (RSS)
  $\rightarrow$ poor locality

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])

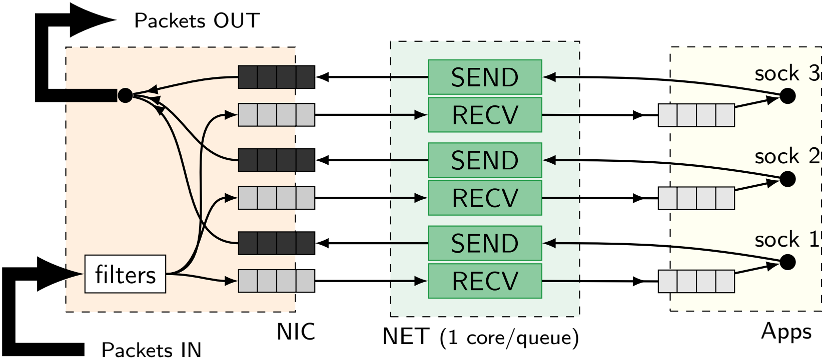
NIC queues and network stacks



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

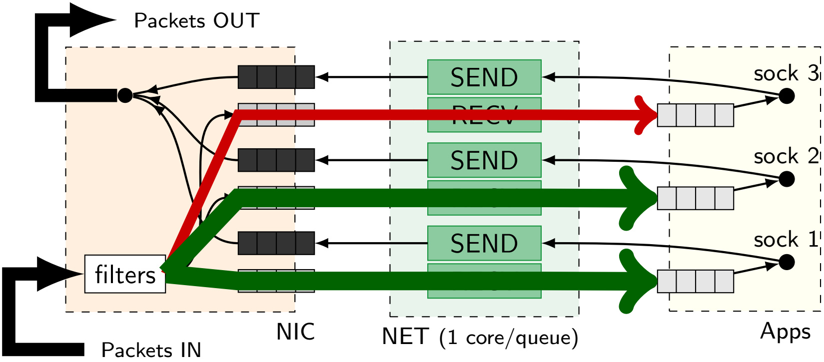
- remove OS from the data path, demultiplexing on NIC

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

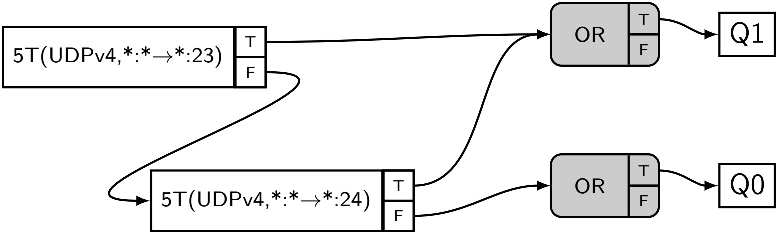
Talk outline

- 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:

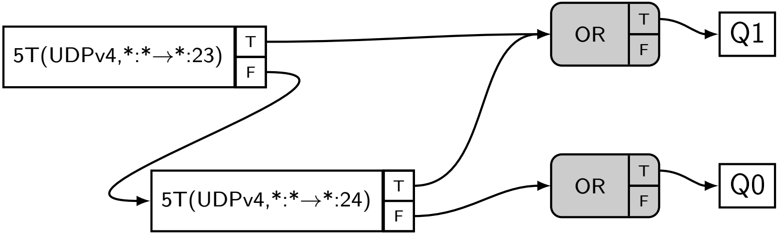
- single input
- ports, each with (possibly) multiple outputs
  - when computation is done, one port is activated
  - subsequently, nodes connected to that port are activated



NIC model

O-nodes:

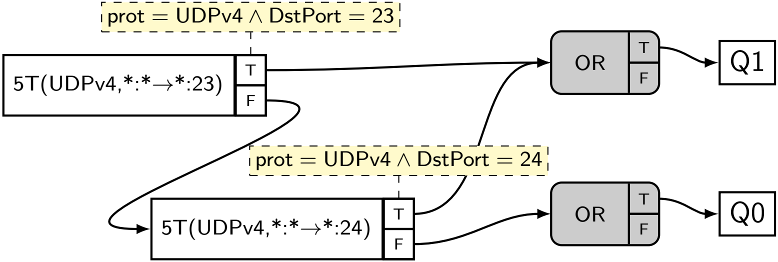
- multiple inputs: $\{T, F\} \times$ operands
- can be short-circuited



NIC model

Predicates:

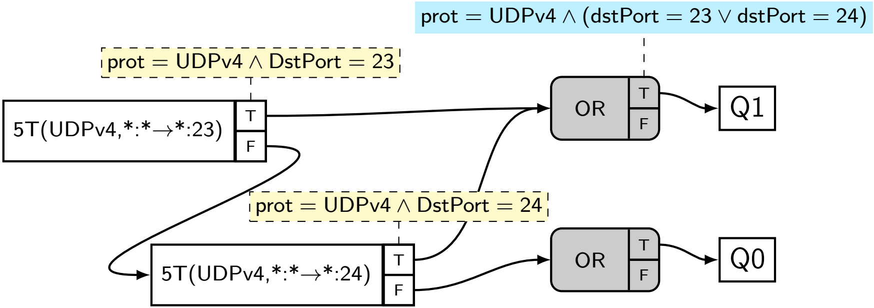
- boolean expressions about the packet
  (atoms of the form: $k = v$)
- each F-node port has a predicate



NIC model

Predicates:

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



Modeling NIC Configuration

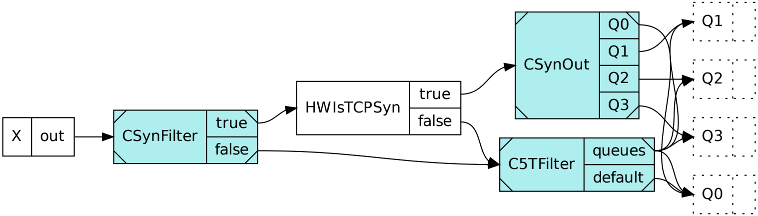
- modern NICs offer rich configuration options
- drastically modify behaviour of NIC

Configuration nodes (C-Nodes)

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

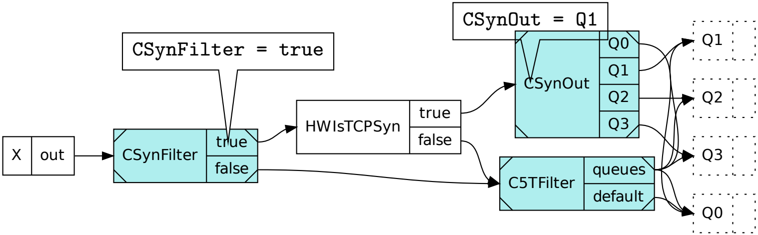
PRG configuration example

(i82599: SYN filter + 5-tuple filters)



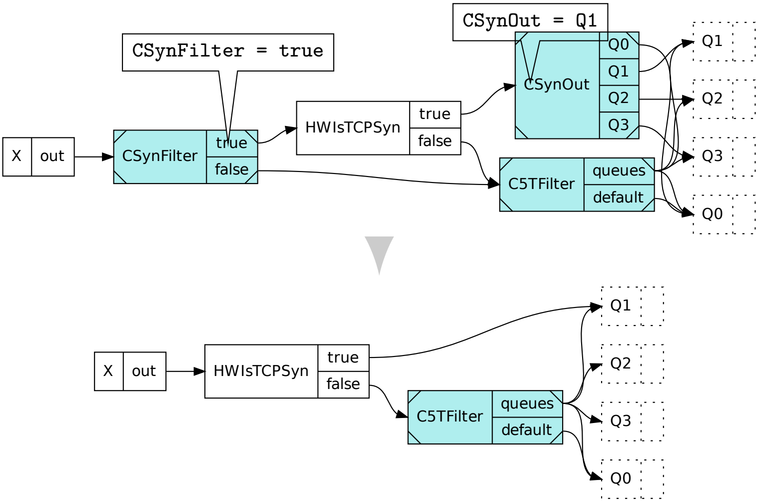
PRG configuration example

(i82599: SYN filter + 5-tuple filters)



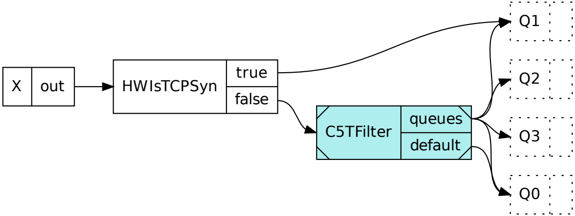
PRG configuration example

(i82599: SYN filter + 5-tuple filters)



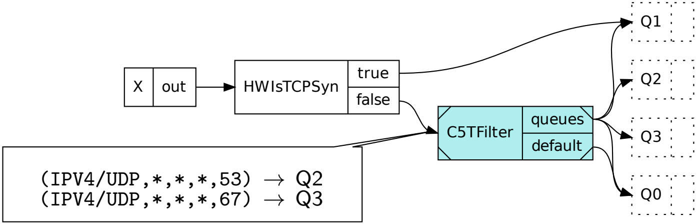
PRG configuration example (cont'd)

(i82599: SYN filter + 5-tuple filters)



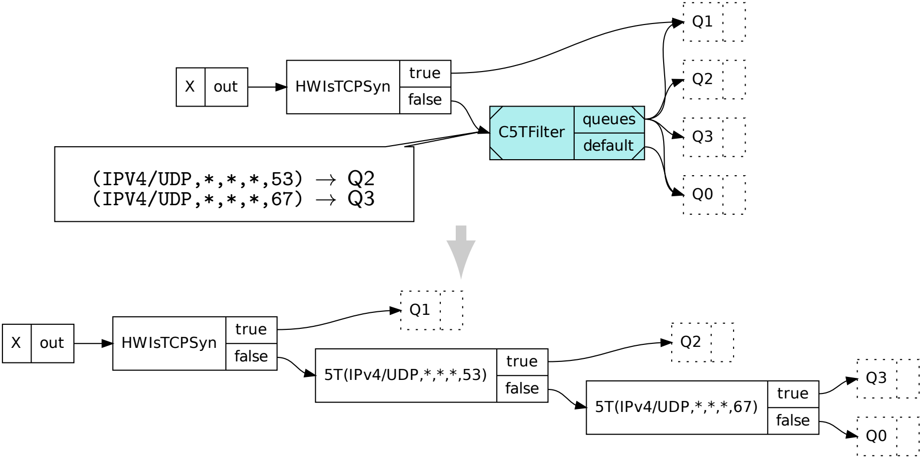
PRG configuration example (cont'd)

(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

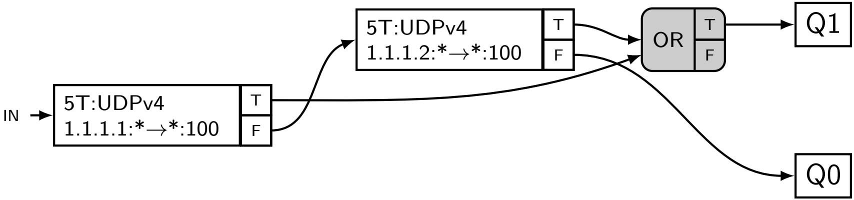
Load balancing:

- variance of number of flows per queue across queues

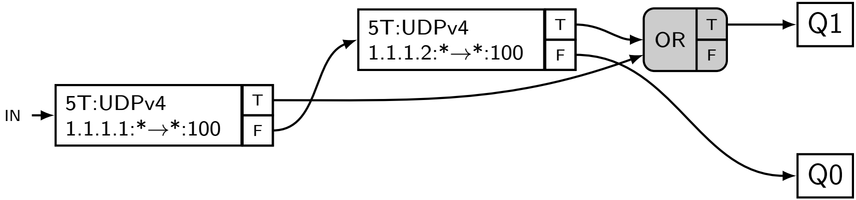
QoS/Performance isolation:

- 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 → queue mapping
(cost function input)



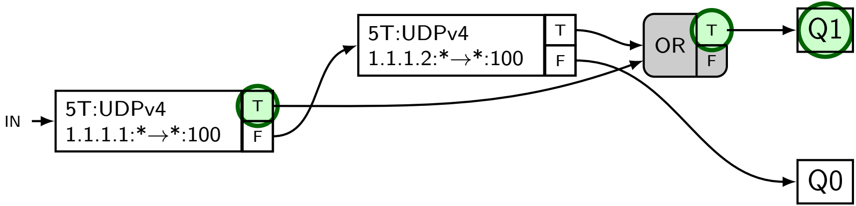
Computing flow → queue mapping
(cost function input)



Flow: UDPv4 / 1.1.1.1:9001 → 1.1.1.42:100
predicate:

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

Computing flow → queue mapping
(cost function input)



Flow: UDPv4 / 1.1.1.1:9001 → 1.1.1.42:100
predicate:

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

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 ← C_0                         // start with an empty configuration
F ← ∅                                  // flows already considered
foreach f in F_all do
    // CC_f: A set of configuration changes on f
    // that incrementally change c
    CC_f ← oracleGetConfChanges(c, f)
    F ← F + f                                      // Add f to F
    find cc ∈ CC_f that minimizes cost(qmap(PRG(c + cc), F))
    c ← 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 ← C_0                    // start with an empty configuration
F ← ∅                                 // flows already considered
foreach f in F_all do
    // CC_f: A set of configuration changes on f
    // that incrementally change c
    CC_f ← oracleGetConfChanges(c, f)
    F ← F + f                                        // Add f to F
    find cc ∈ CC_f that minimizes cost(qmap(PRG(c + cc), F))
    c ← 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
  $CC_f \leftarrow$ `oracleGetConfChanges(`c, f`)`
  $F \leftarrow F + f$ `// Add` f `to` F
  find $cc \in CC_f$ that minimizes `cost(qmap(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 ← C_0                          // start with an empty configuration
F ← ∅                            // flows already considered
foreach f in F_all do
    // CC_f: A set of configuration changes on f
    // that incrementally change c
    CC_f ← oracleGetConfChanges(c, f)
    F ← F + f                    // Add f to F
    find cc ∈ CC_f that minimizes cost(qmap(PRG(c + cc), F))
    c ← 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 `cost(qmap(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 ∈ CC_f that minimizes cost(qmap(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

- Haskel + C
- SolarFlare SFC9020 (OpenOnload)
- Intel i82599 (DPDK)

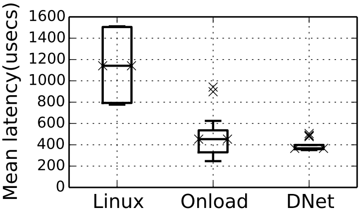
Setup

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

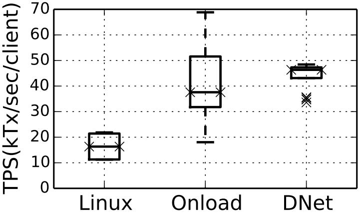
Experiment #1: basic comparison

- **goal**: to show that Dragonet has reasonable performance under the same conditions
- μbench: 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

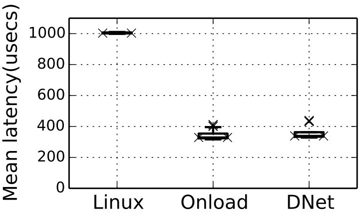
echo server performance on the SFC9020 NIC



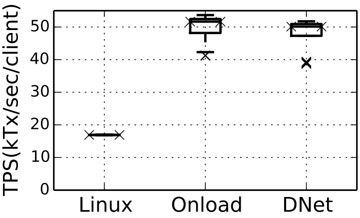
(a) Latency, 1024 bytes



(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 ×16 flows, 18 BE clients ×16 flows (320 flows in total) (*stable*)

- we show here results for the Intel i82599
  (similar* results for Solarflare SFC9020 are in the paper)

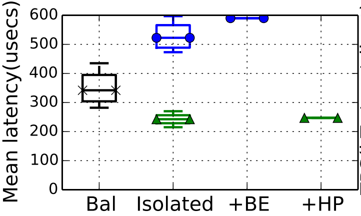
Experiment #2

Performance isolation/Qos

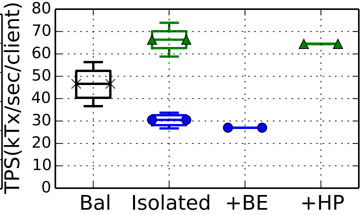
- UDP memcached, memaslap clients
- HP clients: 4 queues, BE clients: 6 queues
- 2 HP clients $\times$16 flows, 18 BE clients $\times$16 flows (320 flows in total) (*stable*)
- after 10secs, we start a new HP client that runs for 50secs
- after new HP is done, we start new BE client
- we show here results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

Performance Isolation

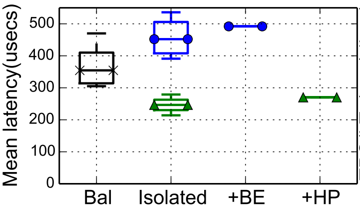
(Intel i82599)



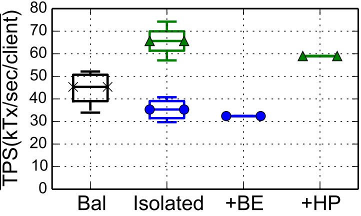
(e) Latency, 1024 bytes



(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 µs	23.7 µs
100	1.2 s	0.6 s	9 ms	94 ms	74 µs	117 µs
250	13 s	4 s	21 ms	219 ms	190 µs	277 µs
500	76 s	17 s	43 ms	484 ms	382 µs	548 µs

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: ETH Barrelfish team!

filter configuration for the Intel i82599 NIC

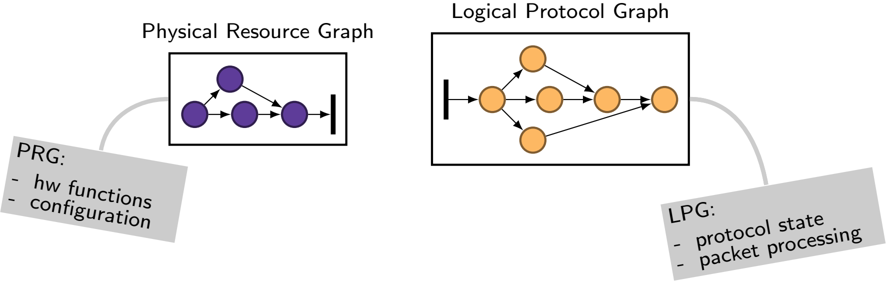
- *5-tuple filters*: 128 filters that match: `<protocol, source IP, destination IP, source port, destination port>`. Each field can be masked.
- *Flow director filters*: Similar to 5-tuple filters. Increased flexibility at the cost of additional memory and latency (stored in the receive-side buffer space and implemented as a hash with linked list chains).

filter configuration for the Intel i82599 NIC

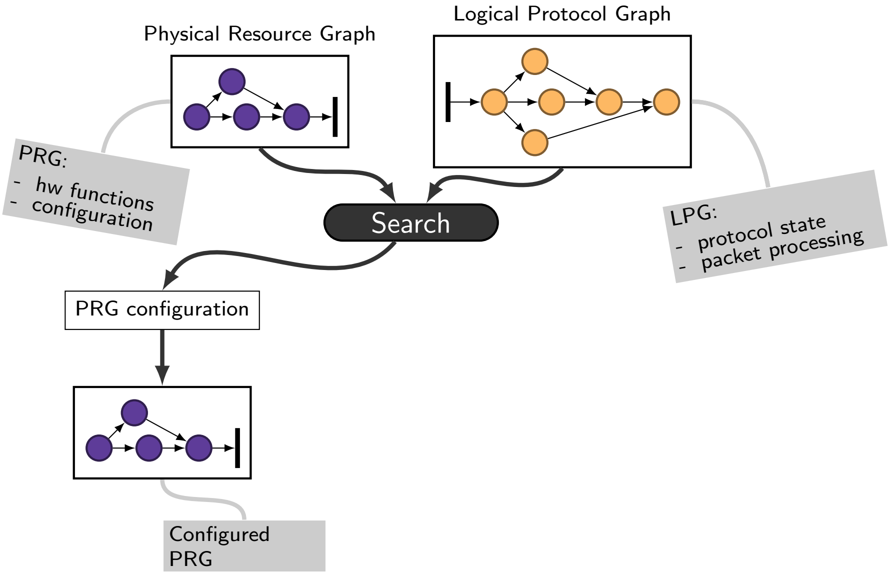
- *5-tuple filters*: 128 filters that match: `<protocol, source IP, destination IP, source port, destination port>`. Each field can be masked.
- *Flow director filters*: Similar to 5-tuple filters. Increased flexibility at the cost of additional memory and latency (stored in the receive-side buffer space and implemented as a hash with linked list chains).

  Flow director filters can operate in two modes: "perfect match", which supports 8192 filters and matches on fields, and "signature", which supports 32768 filters and the matches on a hashed-based signature of the fields. Flow-director filters support global fine-grained masking, enabling range matching.
- *Ethertype filters*: these filters match packets based on the Ethertype field (although they are not to be used for IP packets) and can be used for protocols such as Fibre Channel over Ethernet (FCoE).
- a *SYN filter* for directing TCP SYN packets to a given queue, for example to mitigate SYN-flood attacks.
- *FCoE redirection filters* for steering Fibre Channel over Ethernet packets based on FC protocol fields. Originator Exchange ID or Responder Exchange ID
- *MAC address filters* for steering packets into queue pools, typically assigned to virtual machines.
- Receive Side Scaling (RSS) where packet fields are used to generate a hash value used to index a 128-entry table with 4-bit values indicating the destination queue.

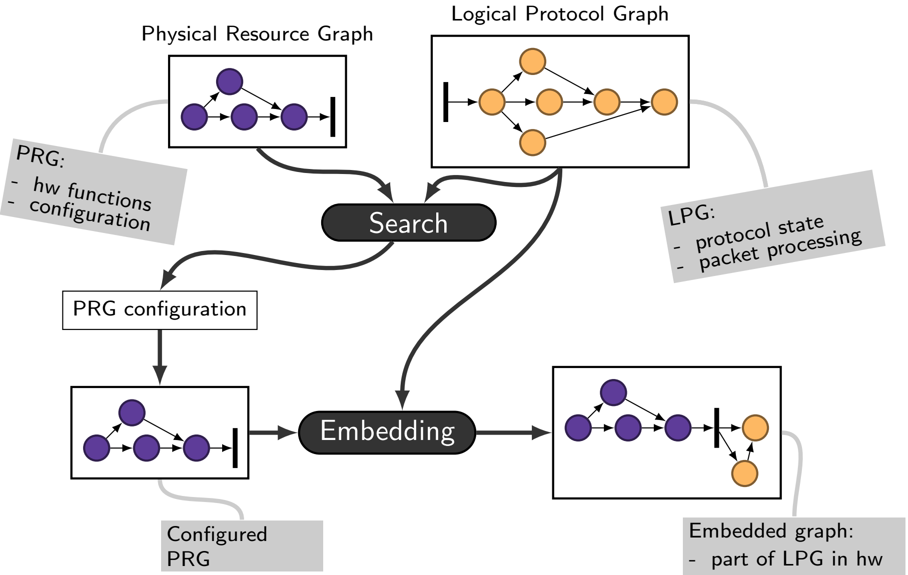
Dragonet in a nutshell



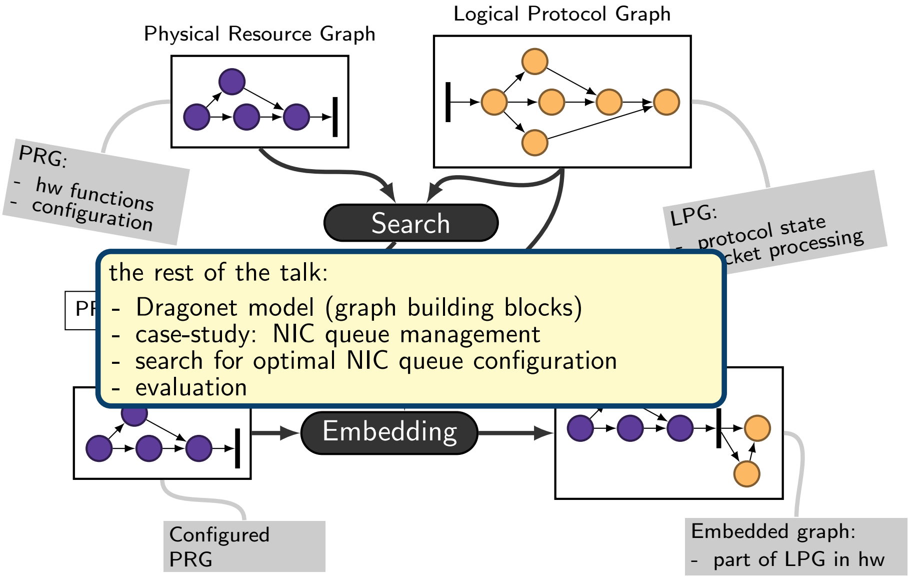
Dragonet in a nutshell



Dragonet in a nutshell



Dragonet in a nutshell



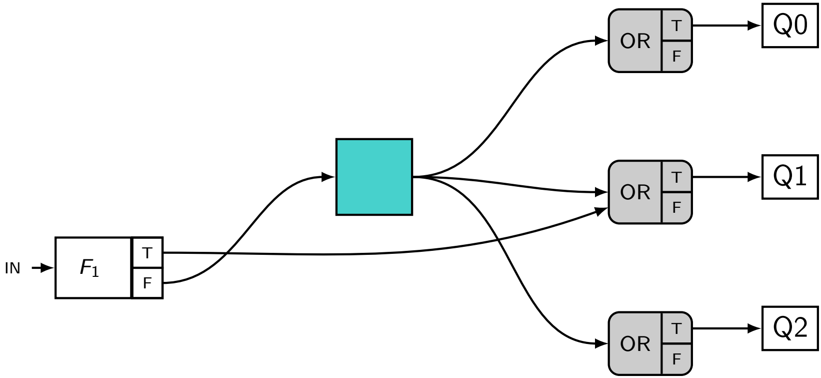
the rest of the talk:

- Dragonet model (graph building blocks)
- case-study: NIC queue management
- search for optimal NIC queue configuration
- evaluation

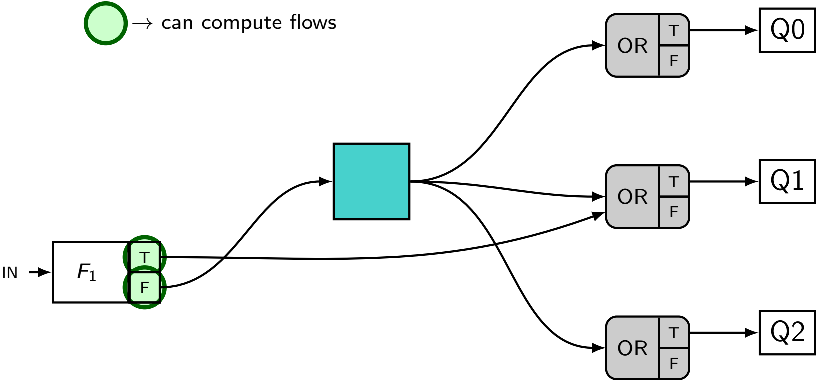
Performance isolation cost function

```
costFn isHp nHpQs queues fm
 | not hpOK = CostReject 1
 | not beOK = CostReject 1
 | length hpFs == 0 = CostVal balBe
 | length beFs == 0 = CostVal balHp
 | otherwise = CostVal $ balHp + balBe
 where
  hpQs = take nHpQs queues -- HP queues
  beQs = drop nHpQs queues -- BE queues
  -- partition flows to HP/BE
  (hpFs,beFs) = partition (isHp . fst) fm
  -- check if HP (BE) flows are assigned
  -- only to HP (BE) queues
  hpOK = and [q `elem` hpQs | (_,q)<-hpFs]
  beOK = and [q `elem` beQs | (_,q)<-beFs]
  -- compute costs of individual classes
  CostVal balHp = balanceCost_ hpQs hpFs
  CostVal balBe = balanceCost_ beQs beFs
```

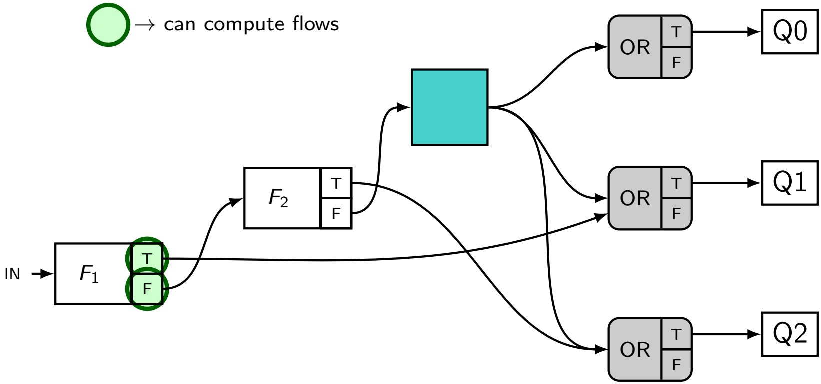
Incremental C-nodes



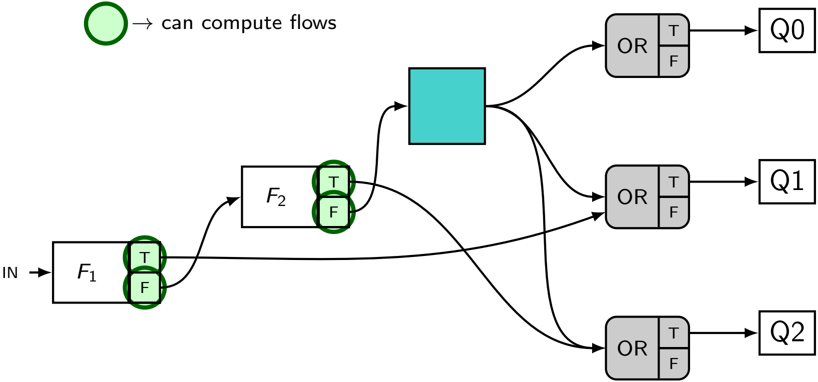
Incremental C-nodes



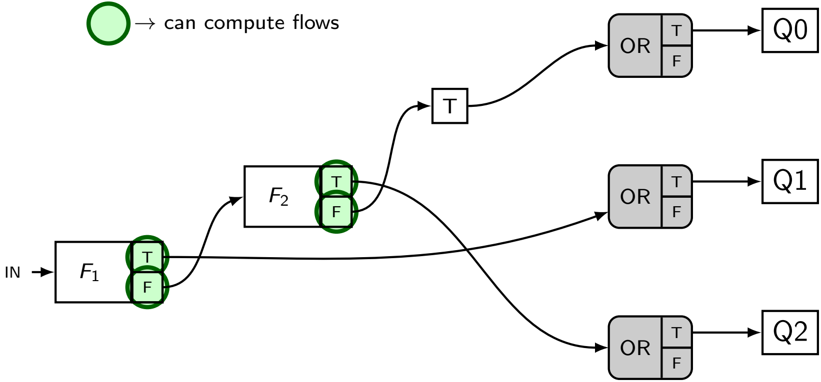
Incremental C-nodes



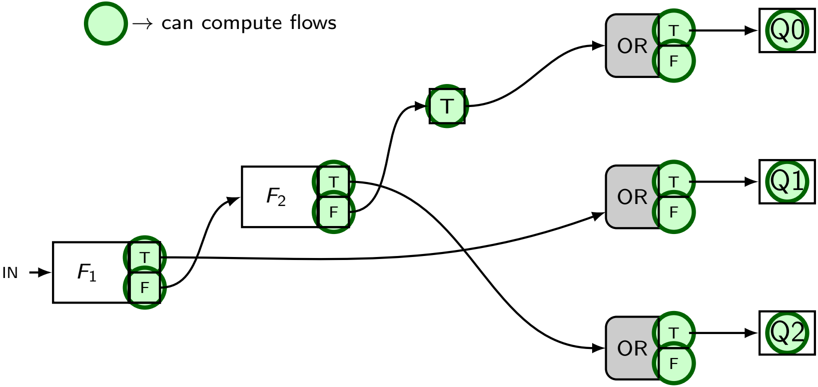
Incremental C-nodes



Incremental C-nodes



Incremental C-nodes

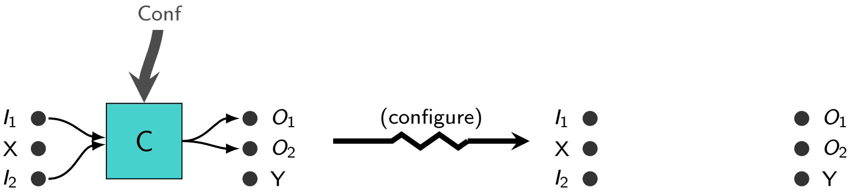


Adding/removing flows

- adding flows: another step in the greedy search
- we remove flows lazily:
  - each *cc* paired with a flow
  - remove the flow, but keep *cc* (do not change configuration)
  - oracle repurposes *cc*'s generated nodes

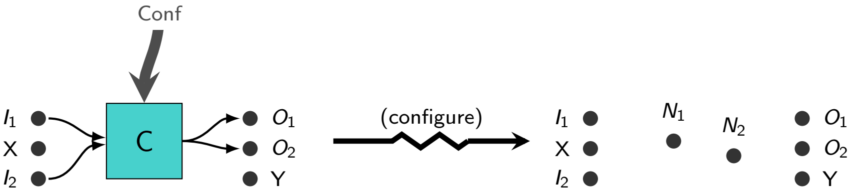
Configuration nodes

(C-nodes)

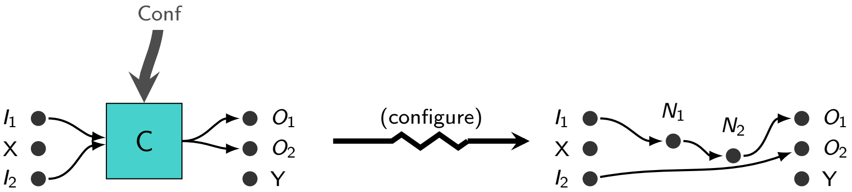


Configuration nodes

(C-nodes)

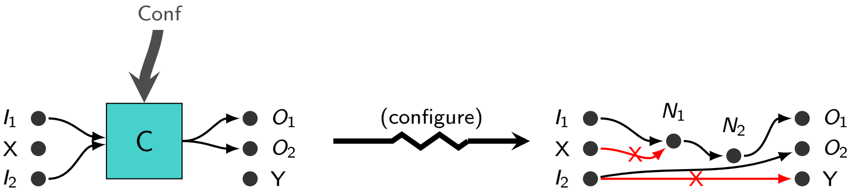


Configuration nodes
(C-nodes)



Configuration nodes

(C-nodes)

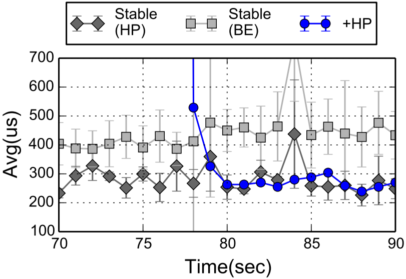


Managing NIC queues

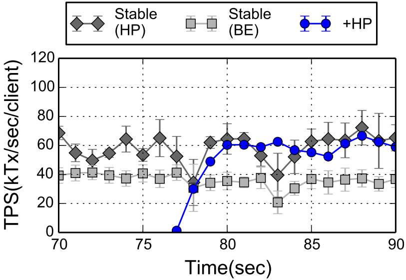
- hardware receive filters (packets $\rightarrow$ Rx queue)
  - Receive Side Scaling (RSS): hash-based load balancing
  - NIC-specific hardware filters (e.g., 5-tuples, TCP SYN packets)
- Linux support
  - RSS (does not consider application locality)
  - Accelerated Receive Flow Steering
    - aims to steer packets to core that application resides
    - maintains flow information
    - calls the NIC driver to steer flows
    - inlined in the protocol implementation
  - Application Targeted Receive
    - implemented in the driver
    - driver samples transmit packets
    - uses device-specific filters to steer packets

Adding an HP client when using 64 byte requests (i82599)

- we instruct clients to provide results every one second (minimum possible value).



(i) Latency



(j) Throughput