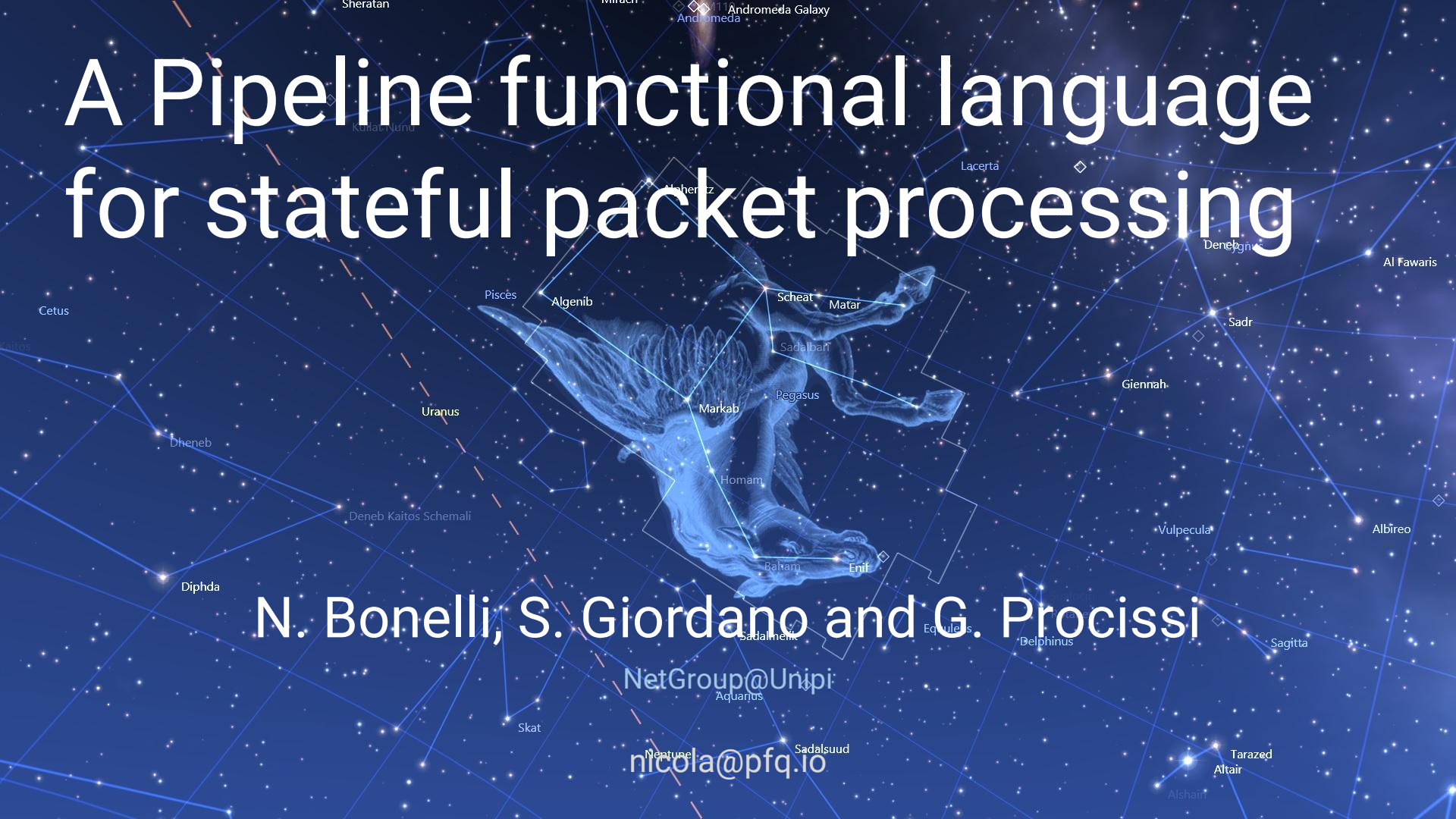


A Pipeline functional language for stateful packet processing

N. Bonelli, S. Giordano and G. Procissi

NetGroup@Unipi

nicola@pfq.io

Context of Network Programming

- Commodity hardware running open-source software (Linux)
- 10/40/100 G NICs + multiple queues + multicore/NUMA arch.
  - Ever increasing speed of the network links (uni -core is no longer an option)
  - Network applications can spend lots of per-packet cpu-cycles

- What makes network apps so special (with respect to generic apps)?
  - Processing: number of packet/sec (extremely high, e.g. VoIP traffic)
  - Consistency (state management)
    - high rate, read/update the state, possibly on-per packet basis
    - synchronization (among cores)

Network programming challenges

- Requirement
  - Scale linearly with the number of cores/threads
- Avoid common programming idioms
  - Mutex and spinlock
- Want be fast? Be lock-free
  - Lock-free/wait-free algorithms
  - Amortized atomic operations (CAS etc.)
  - Memory models (C11, C++11-14) ?
- Memory and data
  - Zero dynamic memory allocations (0-malloc)
  - Minimize sharing and avoid false-sharing of cache-line
  - NUMA and memory-aware applications?
  - HugePages (TLB cache)

Specialized languages (DSL)

- Few specialized network programming languages have emerged:
  - Erlang (Ericsson)
  - P4 + Domino and Banzai (for programmable switches)
  - BPF -> eBPF (in-kernel advanced programming)
  - pfq-lang (PFQ in-kernel functional network programming)
- General purposes languages are still commonly used for network applications:
  - C/C++ (most of the linux app)
  - LUA (SnabbSwitch)
- What about a specialized language for SDN/NFV?

Enif-lang

- High level domain-specific language for SDN/NFV
  - Designed to program middleware boxes
    - Switches, network filters, traffic shapers, load balancers, orchestrate NIDS, DPI..
  - Compact, expressive, and fast (line-rate)

- Drawn on the following principles:
  - Strongly, statically typed (no undefined-behavior)
  - Purely functional (packet immutability)
    - Copy-on-write
  - Concurrent
    - allow multiple applications to works on the same source/NIC/packets
  - Stateful
    - custom flow keys to handle a generic state

Enif-lang

- Enif-lang runs in user-space (subset of Haskell)
  - Abstract engine makes it independent from the different I/O (sockets)
    - **Experimental impl. runs on PFQ**
    - -> DPDK, Netmap, TPACKET3
- Constructs and semantic similar to those of Haskell
- Flexibility
  - Embeddable into Haskell applications
  - Haskell and C/C++ libraries can be mixed in
  - Easily extensible for parsing of new protocols
- Concurrent
  - Contention and parallelism handled by the enif run-time

Enif-lang is functional

- Compositions of pure and effectful functions (Action monad)

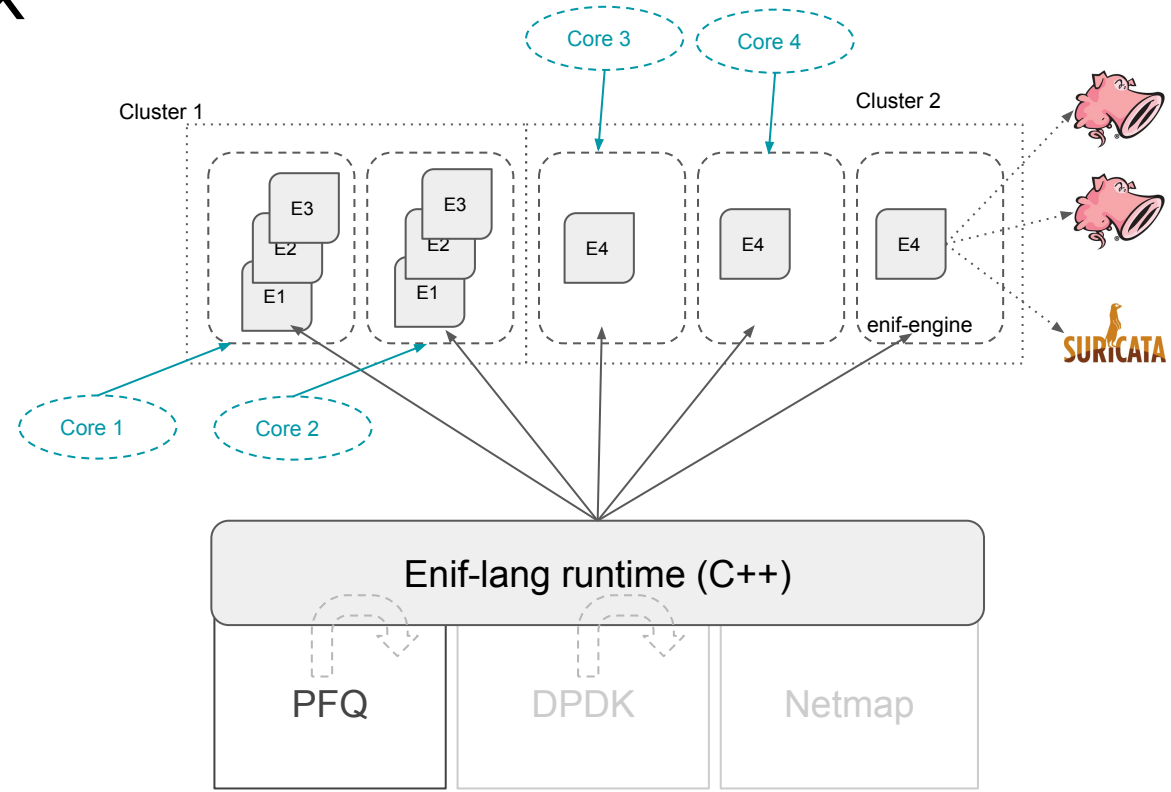
```
function :: Arg1 -> Arg2 -> ... -> Packet -> Action Packet
```

- Action are used for:
  - Forwarding, filtering, steering, logging, sharing state across functions
  - State handling associated with generic “flows”
- Meters and counters (locals and globals)

```
udpCounter = counter 0

enif_main :: Packet -> Action ()
enif_main pkt =
     when (isUDP pkt) $ modifyCounter udpCounter (+1)
```

Enif-lang stack

- Enif-lang app are compiled as .so
- Enif-lang app shares mem queues of packets with the runtime
- Foreign app receive packets via tun interfaces
- Real I/O is performed by the runtime



Enif-lang state handle strategy...

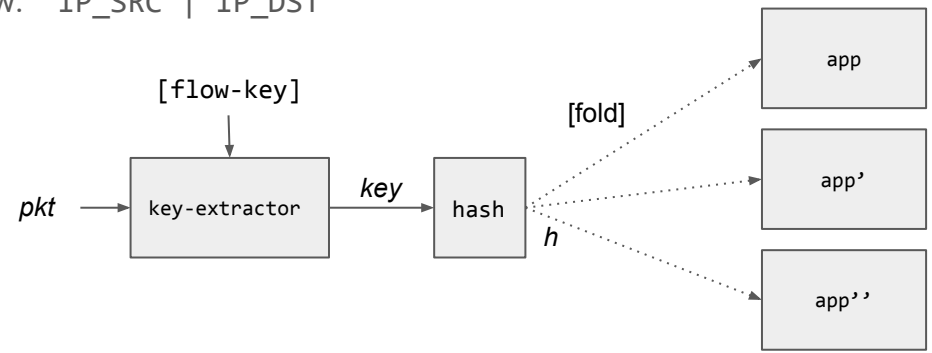
- Stateful processing
  - Data sharing among cores at high rate is hard due to:
    - Synchronization
    - Cache coherence among cores
    - CAS and STM techniques are no better
- Sharing Hash tables
  - Per-bin locking of basic HT is not the a solution...
    - Very difficult to rehash (increase the size of the table)
    - Difficult to implement in plane memory (e.g. HugePages)
  - More efficient (cache friendly) Cuckoo or RobinHood hashing
    - Are shareable without losing efficiency?

Enif-lang state handle strategy...

- Enif-lang runtime takes advantage of packet distributions to scale with the number of cores…
- Is it possible with any kind of application?
  - Stateless applications or applications with constant states is trivial… (Round Robin is fine!)
    - E.g. routing app with static tables
  - Tolerating out-of-order of packets?
    - Improbable to create misorder…
      - Distance of packets that belong to a certain flow is huge (40Gbps)
- Stateful applications are not trivial…
  - Packet distribution must be done with state consistency guarantees!

State consistency guarantees

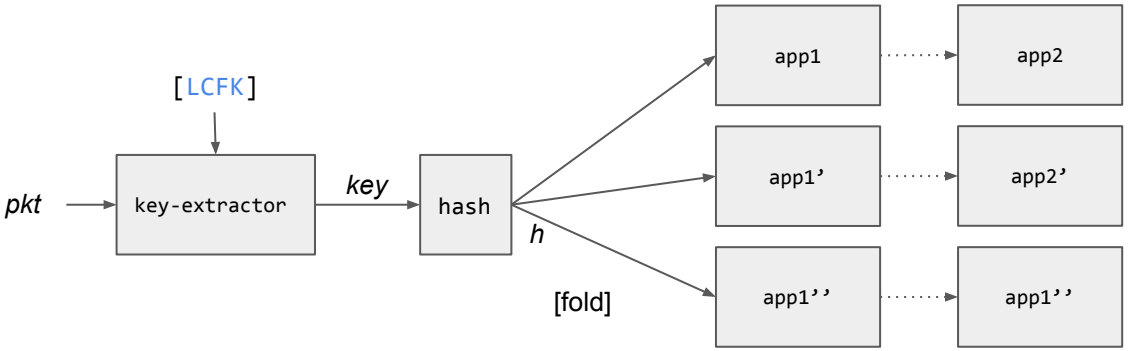
- What is a state for a network app?
  - Anything associated with a certain kind of flow…
  - Packets that belong to a certain flow “share” a generic state
    - to avoid contention we have to properly partition the traffic
- A flow is identified by a mask:
  - E.g. 5-tuple: `IP_SRC | IP_DST | PORT_SRC | PORT_DST | PROTOCOL`
  - Mega-flow: `IP_SRC | IP_DST`



Multiple applications...

- App1: 5-tuple -> state `[ IP_SRC | IP_DST | PORT_SRC | PORT_DST | PROTO ]`
- App2: 3-tuple -> state `[ IP_SRC | IP_DST |    -     |     -    | PROTO ]`

=> Largest Common Flow Key (bitwise and) **`[ IP_SRC | IP_DST |    -     |     -    | PROTO ]`**



Multiple applications...

- App1: 5-tuple -> state
- App2: 3-tuple -> state
- App3: ports -> state

=> **LCFK (bitwise and?)**

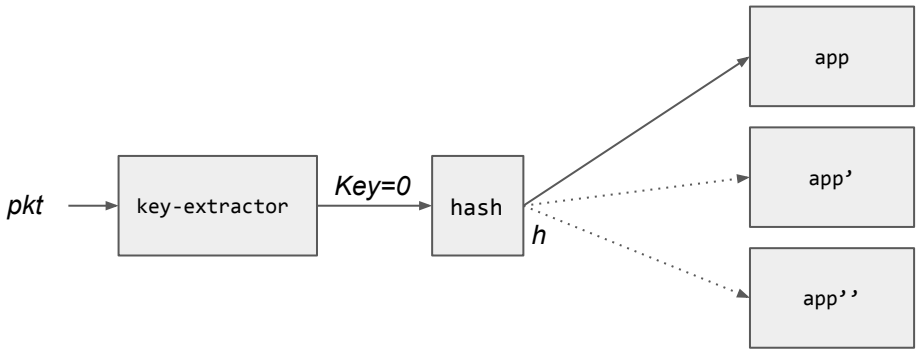
	IP_SRC	IP_DST	PORT_SRC	PORT_DST	PROTO
App1	IP_SRC	IP_DST	PORT_SRC	PORT_DST	PROTO
App2	IP_SRC	IP_DST	-	-	PROTO
App3	-	-	PORT_SRC	PORT_DST	-
LCFK	-	-	-	-	-

Multiple applications...

- App1: 5-tuple -> state
- App2: 3-tuple -> state
- App3: ports -> state

=> **LCFK (bitwise and?)**

	IP_SRC	IP_DST	PORT_SRC	PORT_DST	PROTO
App1	IP_SRC	IP_DST	PORT_SRC	PORT_DST	PROTO
App2	IP_SRC	IP_DST	-	-	PROTO
App3	-	-	PORT_SRC	PORT_DST	-
LCFK	-	-	-	-	-

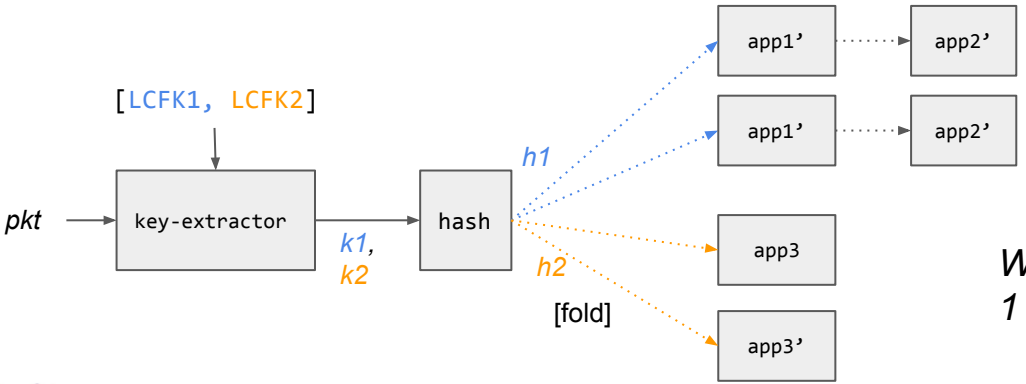


Multiple applications...

- App1: 5-tuple -> state
- App2: 3-tuple -> state
- App3: ports -> state

=> **LCFK1 (app1*app2)**
=> **LCFK2 (app3)**

	IP_SRC	IP_DST	PORT_SRC	PORT_DST	PROTO
App1	IP_SRC	IP_DST	-	-	PROTO
App2	-	-	PORT_SRC	PORT_DST	-
LCFK1	**IP_SRC**	**IP_DST**	**-**	**-**	**PROTO**
LCFK2	**-**	**-**	**PORT_SRC**	**PORT_DST**	**-**



We pay an extra cost of
1 copy of packet per cluster

Enif-lang Hello Word:
“Flow tracker”

Flow tracker...

- [pfq-gen] => [enif-lang/pfq]
  - xeon@3Ghz
  - PFQ 6.0
  - Intel 10G cards (ixgbe RSS=2)
  - p2p link (chopper)

- HashTable
  - Cuckoo hashing
    - Hashtables package
  - +RTS -H1G (heap size)
  - Standard pages 4k pages
    - -> move to HugePages!

```
flowMap :: HashTable FlowKey5 Int
flowMap = newHashTable 1000000


enif_run :: Packet -> Action IO ()
enif_run pkt = do
       let key = mkFlowKey5 pkt
       e <- lookup flowMap key
       case e of
              Nothing -> insert flowMap key 1
              Just x  -> insert flowMap key (x+1)
```

Flow tracker...

- [pfq-gen] => [enif-lang/pfq]
  - xeon@3Ghz
  - PFQ 6.0
  - Intel 10G cards (ixgbe RSS=2)
  - p2p link (chopper)

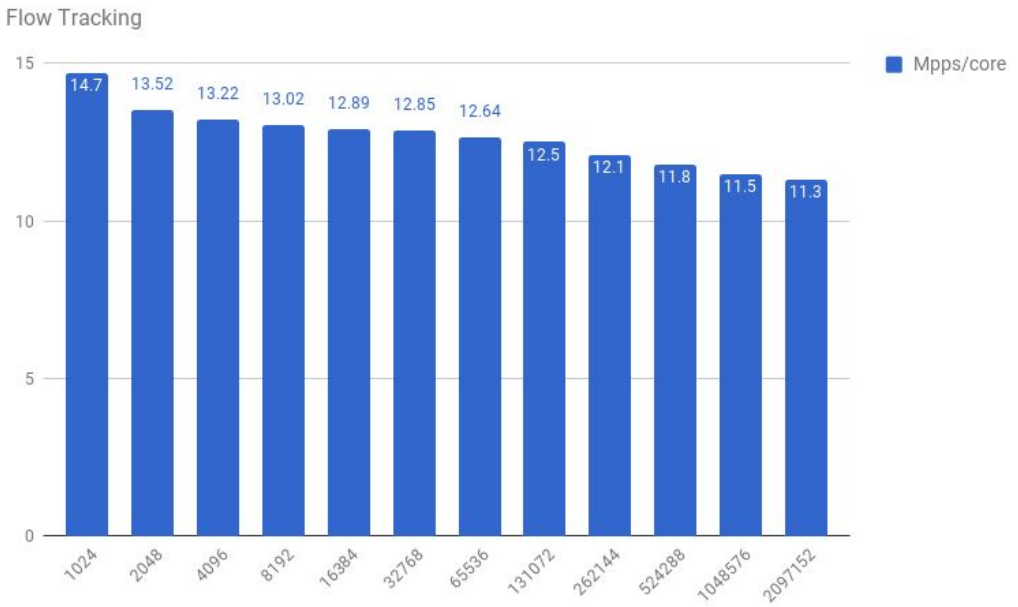
- HashTable
  - Cuckoo hashing
    - Hashtables package
  - +RTS -H1G (heap size)
  - Standard pages 4k pages
    - -> move to HugePages!

```
flowMap :: HashTable FlowKey5 Int
flowMap = newHashTable 1000000

-- defined in Language.Enif.Flow
mkFlowKey5 :: Packet -> FlowKey5
mkFlowKey5 pkt = FlowKey5 byteSwap32 (ipSaddr pkt)
                          byteSwap32 (ipDaddr pkt)
                          byteSwap16 (srcPort pkt)
                          byteSwap16 (dstPort pkt)
                          protocol pkt


enif_run :: Packet -> Action IO ()
enif_run pkt = do
       let key = mkFlowKey5 pkt
       e <- lookup flowMap key
       case e of
              Nothing -> insert flowMap key 1
              Just x  -> insert flowMap key (x+1)
```

Use-case: flow tracker performance (xeon@3Ghz)...

- `[pfq-gen] => [enif-lang/pfq]`
  - xeon@3Ghz
  - PFQ 6.0
  - Intel 10G cards (ixgbe RSS=2)
  - p2p link (chopper)

- `HashTable`
  - Cuckoo hashing
    - Hashtables package
  - +RTS -H1G (heap size)
  - Standard pages 4k pages
    - -> move to HugePages!



Open questions

Open questions

- Best algorithm for app clustering by LCFM?
- K-means or probabilistic clustering?
  - Distance: number of common bits of the flow mask (manhattan)
- Smarter distances... what take into account?
  - Entropy of the flow mask?
    - The more you cluster apps
      - -> the smaller results the common flow mask...
      - -> the less the entropy you get…
      - Unfair clustering
  - Manhattan distance where each single bit is weighted by an entropy factor?
- Learning mode to estimate the entropy of a LCFK?

Open questions

- Best hashing algorithm? Toepliz (RSS)?
  - No funnels
- Faster fold operation
  - We don’t really need modulo operation...
  - Most significant bits of multiplication operation is good enough?
  - Lookup Tables (ala Toepliz)
- How to take into account the processing cost of each single app?
  - Const of per-packet processing (can be measured by the run-time)

Bonus content

Enif-lang functions overview: 1/4

- Predicates :: Arg1 -> Arg2 -> … -> Packet -> Bool (used in conditional expressions):
  - is_ip, is_udp, is_tcp, is_icmp, has_addr CIDR, is_rtp, is_rtcp, is_sip, is_gtp...

```
if (not is_tcp pkt)
      then pass pkt
      else drop pkt
```

```
when (has_addr “192.168.0.0/24” pkt) do
      forward pkt “eth1”
```

- Combinators (action):
  - `inv :: (Packet -> Action IO Packet) -> (Packet -> Action IO Packet)`
  - `par :: (Packet -> Action IO Packet) -> (Packet -> Action IO Packet) -> (Packet -> Action IO Packet)`
    - `par tcp udp >=> kernel`

Enif-lang functions overview: 2/4

- Properties :: Arg1 -> Arg2 -> ... -> Packet -> Word:
  - ip_tos, ip_tot_len, ip_id, ip_frag, ip_ttl, get_mark, get_state, tcp_source, tcp_dest...

```
filter (ip_ttl pkt < 5) pkt      when (get_state pkt == 10) kernel
```

- Comparators:
  - `any_bit :: (Packet -> Word) -> Word -> Packet -> Bool`
  - `all_bit :: (Packet -> Word) -> Word -> Packet -> Bool`

```
if (any_bit tcp_flags (SYN|ACK))...
```

Enif-lang functions overview: 3/4

- Filters (effectful functions): {pass, drop}
  - ip, udp, tcp, icmp, rtp, rtcp, sip, voip, gtpv1, gtpv2, no_frag... :: (Arg..-> Packet -> Action IO Packet)
  - l3_proto, l4_proto :: Int16 -> Packet -> Action IO Packet

```
l3_proto 0x842 >=> log_msg "Wake-on-LAN!"
```

  - vlan_id_filter :: [Int] -> Packet -> Action IO Packet

```
flan_id_filt [1,2,3] >=> kernel
```

  - port, src_port, dst_port :: Int16 -> Packet -> Action IO Packet

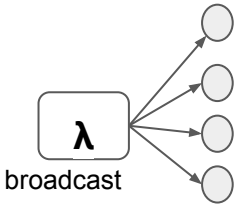
```
port 80 >=> log_msg "http packet!”
```

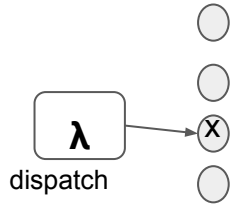
  - addr, src_addr, dst_addr :: CIDR -> Packet -> Action IO Packet

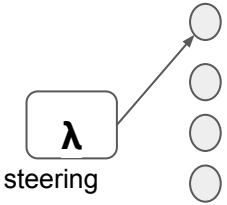
```
src_addr "192.168.0.0/24" >=> log_packet
```

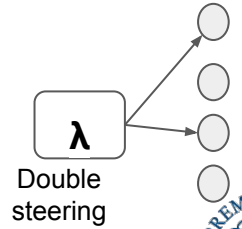
Enif-lang steering

- Provides a bunch effectful functions for steering
  - Fine-grained control of packet dispatching across end-points
- Steering the traffic
  - To fully exploit the hardware of multi-core architectures
  - To let core work independently from each other
    - Reduce or avoid the inter-core communications
    - Find a trade-off between correctness and performance
- Different kind of (weighed) steering
  - Pass (unit), Drop (filtering)
  - Broadcast (deterministic)
  - Dispatch CLASS (deterministic)
  - Steer HASH (randomized)
  - DoubleSteer Hash1 Hash2 (randomized)









Enif-lang functions overview: 4/4

- Steering :: Arg1 -> Arg2 … -> Packet -> Action IO Packet
  - broadcast, drop, kernel, forward (e.g. forward “eth1”)
  - steer_rrobin (useful only for stateless operations)
  - steer_link, steer_local_link (GW)
    - `steer_local_link "4c:60:de:86:55:46"`
  - double_steer_mac
  - steer_vlan
  - steer_p2p, double_steer_ip, steer_local_ip :: CIDR -> Packet -> Action IO Packet
    - `steer_local_ip "192.168.1.0/24"`
  - steer_flow, steer_field, steer_field_symm, double_steer_filed
  - etc..

Enif-lang syntax

- do notation desugaring:

```
dns pkt = port 53 pkt
enif_main pkt = do dns pkt
     log_packet pkt
     forward “eth1”
```

```
enif_main = dns >=> log_packet >=> forward “eth1”
```

- shifting computations (tunnels)
  - Support for IPIP, IPv6IP, (IPGRE, MPLS? future work)

```
enif_main = steer_flow -- doesn’t work
```

```
enif_main pkt = shift $ steer_flow pkt
```

```
enif_main pkt = shift $ shift $ do
                addr “192.168.0.0/24” pkt
                kernel pkt
```

Enif-lang syntax

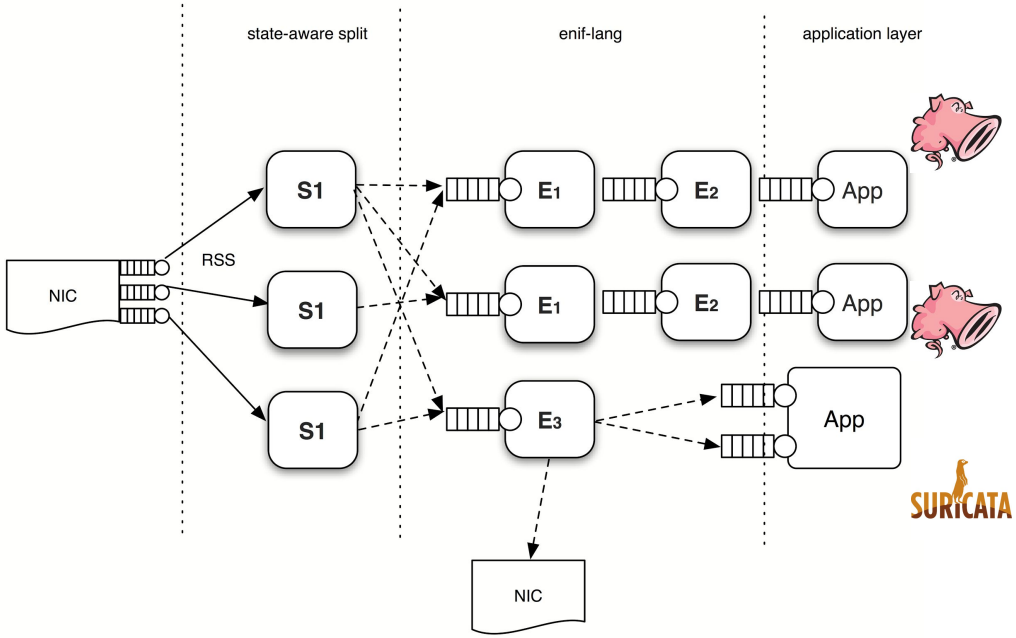
- High-order functions
  - Functions that can takes functions or effectful functions as argument
  - `conditional, when, unless, inv, par, filter` etc.
  - `if-then-else` is a built-in support borrowed from Haskell

- Context (high-order functions)
  - All functions that operates on src or dst fields can be restricted
  - `src, dst, local, remote` create restricted contexts:

```
src $ do
      addr “192.168.0.0/24”  pkt
      Kernel pkt
```

```
local “192.168.0.1/24” $ do
      if (has_port 80)
            then steer_p2p pkt
            else drop pkt
```

Enif-Lang architecture



Enif-lang internal notes (Haskell)

- State Monad is pretty slow!
  - transformers or mtl -> 4.2 Mpps!
- ST Monad has good performance
  - However it does not provide a state (like State Monad)
  - The state must be provided via mutable STRef
- Because we need IO monad…
  - Data.IORef for mutable internal state is preferred (IORef and STRef perform equal)
  - LANGUAGE ImplicitParams (hidden state)
- Waiting for GHC 8.2 for unboxed sum types!
  - Performance issue with polymorphic Maybe a

Enif-lang as executable eDLS...

1. Classic way
    - Grammar (BNF) -> Parser -> AST (Abstract Syntax Tree)
    - Compiler -> IR (possibly to plain C) -> binary
2. eDSL on host language for AST
    - Haskell is commonly used as host language
        - Compiler/RT interpreter/or an hybrid engine (ala pfq-lang)
            - E.g. AST of pfq-lang: args and ptrs to functions without RT optimizations
3. eDSL as a host library <- Enif-lang
    - Subset of the host language (Haskell)
    - No parser, no compiler -> GHC does all (including optimizations)
    - FFI (foreign call from/to C)