Software Acceleration

... for Network Applications and multi-core architectures

nicola@pfq.io

nicola.bonelli@cnit.it
Agenda

- BEBA project
- High Performance Network Programming
- Multithreading on multi-core
- PFQ framework
- OFSS/BEBA acceleration
What’s the BEBA point?

OpenFlow / SDN
- Control-plane
- Data-plane
- DUMB!
- Forwarding rules

BEBA / SDN
- Control-plane
- Data-plane
- SMART!
- BEBA switch
- BEBA switch still decides how switches shall «behave»
- Forwarding behavior:
  - Forwarding rules AND how they should change or adapt to «events»
- Smart switches can dynamically update flow tables at wire speed
Why is BEBA needed?

- Dramatic (control) latency reduction!
  - Enable state transition at **data plane level**
  - Transition ~ usec (packet rate) vs. transition ~ msec (control rate)

- Implications
  - Reliability (fast fault management)
  - Resilience (e.g. controller unreachable)
  - Signaling offload

- Control plane simplified (keep local what is local)
CNIT role

- Implementation of software BEBA prototype
- Why not OVS?
  - Rigid architecture designed for speed
    - Cache and megaflow cache
  - OpenFlow abstraction is *not* built-in in the data-plane
    - OF manager is supposed to program the underlying engine
    - Communication kernel <-> user space is very slow!

- OFSS (OpenFlow Soft Switch)
  - Simple switch architecture that embodies OpenFlow spec. In the data-plane
  - Linux Single-threaded, run in user-space, C language
  - PF_PACKET socket (no TPACKET variant)!
  - Terribly slow! ~150,000 pps!
Network programming
Network programming

- Commodity hardware running open-source software
  - Linux Os
- 10/40/100 NICs with multiple interrupt lines (MSI-X)
- Modern multi-core architectures
  - Multi-core CPU, multiple packages NUMA, cluster of servers etc.
- Multi-core is a central element?
  - Speed of network links in increasing...
  - CPU frequency has nearly reached a hard limit
- Non-trivial network apps can spend lots of cpu cycles
  - 1000/10000 cpu cycles per packet is very common
    - On a 3Ghz CPU, 3 Mpps is the most
What does performance mean?

- The number of event/sec!
  - packets/sec processed, forwarded etc.

- Application vs socket performance
  - The packet rate a socket can sustain is not a valid metric for applications!

- CPU load is a valid metric?
  - A middlebox is not a general purpose PC
  - High perf. sockets do active polling!

- Misconception: a CPU overloaded is bad!
- A low CPU (not overloaded) is in general a bad sign!

- A reliable index is in any case required!
The high-performance gourmet recipe...

- Fast packet I/O (DPDK, PF_RING, Netamp, PFQ...)
- Concurrency?
  - Many applications are still implemented as single process/thread
- Lock-less programming (no mutexes or spinlock)
  - Lock-free/wait-free algorithm, amortized atomic operations (CAS and all.)
  - Memory Model (C11,C++11)?
- Polling vs latency
- Memory
  - Zero dynamic memory allocations (0-malloc)
  - Reduced true sharing, zero false-sharing of cache-line
  - NUMA and memory-aware applications? Hugepages?
- Affinity
  - Interrupt affinity of multi-queues NICs?
  - Process/thread pinning, cpu isolation and real-time scheduling...
Other network driver optimizations

- Intel Direct Cache Access (DCA)
  - Controversial? (Napi <-> rx k-thread)
- Intel MSI-X (RSS)
  - Interrupt affinity
- Busy polling? (since Linux kernel 3.11 - 2013)
  - After 4 years only 6 drivers implement it:
    - Chelsio, Myricom, Cisco/enic, Emulex/benet
    - Intel/ixbge, Broadcom/bnxt
- Batch transmission
  - xmit_more
Memory: static is better

- Dynamic memory allocations is the first enemy!
  - 200 - 500 CPU clock required (alloc+free)!
  - Not acceptable to pay this fee on per-packet basis...

- JEmalloc, TCmalloc are specifically tailored for multi-core...
  - Useful to detect if the bottleneck is due to dynamic memory allocation

- Re-design the application to get rid of dynamic memory allocations

- Dynamic allocation vs life-time
  - Think carefully about life-time of memory chunks

Either you can avoid it or will have a bad time implementing a better allocator!
Memory: it’s about data

● You should care about memory latency...
  ○ Read from a central memory can cost 100 - 200 CPU cycles!

● Cache miss is very common! Why?

● A packet forwarder is the most simple network application!
  ○ It is stateless
  ○ It does not copy bytes (fast memory mapped sockets)
  ○ It is probably *not* affected by cache misses
  ○ Cannot be used as metric!

● Real network applications
  ○ Cache-misses on per-packet basis!
  ○ Waste cpu cycles retrieving the state associated to a certain packet!
  ○ The faster is the network, the higher is the number of the states to retrieve
    ■ E.g. number of TCP flows
  ○ Depending on the algorithm many memory access per-packet are required
Memory: it’s about data structure

- Dispose data in a smart way
  - the most frequently used data should be arranged in the same cache line (64 bytes)...
- Vector data structure is good for few elements
  - Slots are contiguous in memory
    - e.g. vector of int: 1 read of a cache line moves 16 integers into cache
  - In modern architectures some operations can be performed in parallel
    - If there’s no data-dependency
- Hash tables perform better than associate containers
  - ... when using a fast hash function
- Probabilistic data structure can used to alleviate the costs of retrieving the state
  - Bloom filters, counting bloom filters
  - Sketches, Hyperloglog counters (network anomalies)
Multi-threading

Because applications/algorithms are slow...

- Multi-threading/process is the only the viable option to exploit modern multi-core architectures
  - You can have also processes that share memory like threads (clone system call)
- Your best bet is that your algorithm scales linearly with the n. of cores
  - Sometime is possible, sometime is not!
- Network applications usually *do* allow parallel processing
- State-aware traffic split
  - State per-flow, user, network, link, vlan, tunnel, GTP teid etc.
  - To exploit the whole number of cores and let each one process packets in isolation...
Multi-threading: counters

- Instrumenting an application with counters is easy
  - ... but if the application is multi-thread is not that easy!
    - Basic counters, maximum, minimum, sliding-window average etc.
    - Resettable vs non-resettable counters
    - Coherent vs non-coherent counters

- Mutex and spinlock are too expensive!
  - Data-race, multiple readed with at least a writer...

- Atomic RMW operations are expensive as well!
  - nearly 3 times faster than using a spinlock (it is not enough)
  - With high contention we have to avoid them!
Multi-threading: sparse/per-core counters

- Distributed counters are the way to go

- Each component is arranged in a separated cache line
  - This avoids false sharing

- You can either:
  - Accept benign race conditions
    - You have to deal with architecture (e.g. linux kernel use local_t in place of atomic_t)
  - Use *relaxed* atomic operations (read and write instead of atomic increment)

- The thread interested in a counter:
  - Reads *(atomically and relaxed)* all the components of the *sparse-counter* and make the sum
Counters: performance comparison

1. Basic counter protected by a mutex
2. Basic counter protected by a spinlock
   ○ Ticket spinlock to avoid thread/core starvation
3. Atomic (relaxed) counter
4. Dense atomic counter (contiguous components)
   ○ 1 component per core
5. Per-core sparse atomic counter (no false sharing)
   ○ 1 component per-core aligned to the cache-line size
Counters: performance comparison

Xeon(R) CPU E5-1660 sandy-bridge
@ 3.00GHz

- Max. speed 180M.increment/sec
- Atomic and dense atomic counters perform similarly!
  - **FALSE SHARING!**
- Only sparse without false sharing scale linearly with n. of cores!
Counter performance
SPSC queue

- SPSC is the most simple lock-less queue implementation
- Intel architecture allows a trivial implementation:
  - compiler usually aligns integers (unless forced not to do)
  - hardware do not reorder writes to different address of memories

1. It is possible to implement SPSC@Intel without atomics...
2. A compiler barrier is required if the functions are declared inline!!!
3. It was the only queue that could be implemented without assembly prior C11/C++11!

-> Drawback: it is not portable!
-> Basic implementation is slow!
SPSC queue: why is it slow?

- push/pop operations require to read **both indices**
  - push has to detect whether the queue is full
  - pop has to detect whether the queue is empty

  -> cache misses cause a continuous ping/pong between two cores!

- Alignment to cache-line is of any help?

- To improve the performance...
  - We can add a cache in the code to reduce the ping/pong
  - We can add an additional batch policy
    - Further reduce cache synchronization
    - Reduce cache missing for the consumer!
SPSC: performance comparison

1. SPSC simple (Intel ver)
2. SPSC (producer/consumer index in different cache-line)
3. SPSC (portable, relaxed atomic with memory model)
   ○ acquire/release semantic
4. SPSC with soft-cache
   ○ Hiligy reduce cache misses
5. SPSC + soft-cache and internal batch
   ○ No modification to the client code is required
   ○ Transparent batching between producer and consumer
SPSC: performance comparisons
PFQ framework

PFQ Network Programming Framework (pfq.IO)

- Capture, processing, distribution and transmission of network traffic
- Linux compliant (kernel 3.2 up to 4.9)
- High-performance
  - 14.8Mpps on 10G Intel 2 core (Xeon@3Ghz)
- Programmable with different languages
  - C/C++11-14/Haskell
  - Early stage processing with pfq-lang

“Network Traffic Processing with PFQ” JSAC-SI-MT/IEEE journal Special Issue on Measuring and Troubleshooting the Internet
PFQ architecture

- **Kernel module**
  - prefetch + batch queues (SPSC)
  - socket queue Rx/Tx (MPDB)

- **User-space libraries**
  - C/C++11-14
  - Haskell bindings
  - Libpcap (fanout)

- **User-space Q-lang (eDSL)**
  - C++11-14
  - Haskell DSL
    - RebindableSyntax
  - Experimental Qlang compiler
PFQ features...

- Run on top of standard network drivers (**pfq-omatic**)
  - Drivers delivered by manufacturers like intel ixbge, ixb, i40e
  - In-kernel tree drivers
  - Binary patching?

- Pool of socket buffers (skb memory allocator)
  - Faster than Linux memcache (2 or 3 times)

- Take advantage of copies
  - Intel DCA technology
  - Hugepages (2M and 1G supported)
  - Batch transmission (xmit_more and pktgen)

- Lock-free architecture
  - No-mutexes or spinlock in the fast data-path
PFQ features...

- Communications via
  - DB-MPSC queue (socket queue)
  - SPSC with cache

- Atomic-free or amortized atomic operations!
  - Batch in MPSC and SPSC queues

- No false-sharing
  - Per-core data structures padded to the cache-line size.
  - ... and use the minimal amount of sharing

- Sparse counters
  - Distributed components to avoid false sharing

- Aggressive pre-fetching
  - Prefetch of packet headers
  - Qbuff in place of SkBuff (fit into a single cache line)
Is PFQ fast?

Packet capture and packet steering figures
pfq-lang

- Functional language as eDSL (AST)
  - syntax borrowed from Haskell
- Compositions of effectful functions (monads)
  - function :: Arg1 -> Arg2 -> ... -> QBuff -> Action QBuff
- The Action context is responsible to:
  - Forwarding, filtering, steering, logging, sharing state across functions
  - Meters and counters
- Experimental compiler
  - Grammar parser, IR as JSON/binary
- Runtime functional engine
  - Implementation exists within PFQ kernel module (BPF on steroid)

*With steering enif-lang open the doors to parallel processing!*
pfq-lang functions overview: 1/3

- **Predicates :: Arg1 -> Arg2 -> ... -> Qbuff -> 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)
  then pass
  else drop
  ```
  when (has_addr "192.168.0.0/24") do
  forward "eth1"

- **Combinators :: (Qbuff -> Bool) -> (Qbuff -> Bool) -> (Qbuff Bool)**
  - `.||., .&&., .^^., not...

    ```
    when (is_tcp .||. is_udp)...
    ```

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

  ```
  filter (ip_ttl < 5) when (get_state == 10) kernel
  ```
pfq-lang functions overview: 2/3

- Comparators:
  - `<`, `<=`, `==`, `/=`, `>`, `>=`, `<=`, `==`, `/=`, `>`, `>=`
  - `any_bit :: (Qbuff -> Word) -> Word -> Bool`
  - `all_bit :: (Qbuff -> Word) -> Word -> Bool`

  
  if (any\_bit tcp\_flags (SYN.|.ACK))...

- Filters (effectful functions): {pass, drop}
  - `ip, udp, tcp, icmp, rtp, rtcp, sip, voip, gtpv1, gtpv2, no\_frag... :: (Arg...-> Qbuff -> Action Qbuff)`
  - `l3_proto, l4_proto :: Int16 -> Qbuff -> Action Qbuff`
  - `vlan_id_filter :: [Int] -> Qbuff -> Action Qbuff`
  - `port, src_port, dst_port :: Int16 -> Qbuff -> Action Qbuff`
  - `addr, src_addr, dst_addr :: CIDR -> Qbuff -> Action Qbuff`
pfq-lang functions overview: 3/3

- Steering :: Arg1 -> Arg2 ... -> Qbuff -> Action Qbuff
  - 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 -> Qbuff -> Action Qbuff
    - `steer_local_ip "192.168.1.0/24"
  - steer_flow, steer_field, steer_field_symm, double_steer_field
  - etc..
pfq-lang syntax

- do notation desugaring:

```plaintext
dns = port 53
main = do dns
   log_packet
   forward "eth1"

main = dns --> log_packet --> forward "eth1"
```

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

```plaintext
main = steer_flow -- doesn’t work

main = shift $ steer_flow

main = shift $ shift $ do
   addr "192.168.0.0/24"
   kernel
```
OpenFlow Soft-Switch
OFSS Architecture

- Two processes (`ofprotocol` and `ofdatapath`)
- `ofdatapath` is a single process
  - does not exploit multi-core arch.
- Built in AF_PACKET sockets
  - receive/transmit packets
  - manage queuing disciplines, stats, etc.
OFSS and BEBA extension

**State table**

- match key | state
- ...
- ...
- ...
- *
  - DEFAULT

**XFSM table**

- match fields
  - headers | state
  - ...
  - ...
  - ...

- actions
  - ...
  - ...
  - ...

**Packet Flow**

- pkt headers -> Key extractor (lookup-scope) -> State table
  - pkt headers + state
  - Key extractor (update-scope) -> pkt headers + next_state
  - SET_STATE next-state

- CONTROLLER
  - flow-mod
  - state-mod
OFSS rejuvenation

- Native support of PF_PACKET replaced by libpcap
  - Moving to libpcap fanout to support multi-core even with standard sockets

- Packet handler replaced by FLAT_PACKET
  - A single chunk of memory instead of 3 scattered buffers
    - Save 2 malloc/free
  - Handler with and without memory ownership of the payload
    - Packets are consumed in the queue of the socket whenever is possible (save 1 malloc/free)
  - Handler without memory ownership of itself (save 1 malloc/free)
  - Cache line friendly

- Core affinity
  - ofdatapath can be pinned to a certain core
OFSS rejuvenation

- **Hmap** is a generic associative container
  - Key: integer that identifies fields of network protocols
  - Value: is a generic field of any possible protocol (variadic in the size)

- **Hmap** has been extended to support:
  - A set of small nodes preallocated in the memory of Hmap (size <= 8 bytes)
    - Save 5 malloc/free on average (on per-packet basis)
  - Bin reservation to not rehash in the average case (save 1 malloc/free and rehash)

- **Configurable batch with a work-budget**:
  - The forwarder thread pools every port processing packet until the max work-budget is reached
OFSS Architecture

● PFQ under the hood that:
  ○ dispatches packets to multiple instances of ofdatapath

● Ofprotocol local controller
  ○ Supposed to handle multiple instance of ofdatapath
  ○ Supposed to load computations into PFQ (according to the BEBA program in execution)
OFSS performance gain (1 core)

- PFQ 32%
- Zero-malloc 29%
- Batch processing 20%
- Hashtable improv. ~ 9%
- Zero copy ~ 8%
OFSS performance gain

- Comparison with OFSS native
- Forwarding speedup factor:
  - \((\text{Mpps BEBA})/(\text{Mpps OFSS})\)

<table>
<thead>
<tr>
<th></th>
<th>1 core</th>
<th>2 core</th>
<th>3 core</th>
<th>4 core</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>28 x</td>
<td>55 x</td>
<td>76 x</td>
<td>90 x</td>
</tr>
<tr>
<td>128</td>
<td>25 x</td>
<td>48 x</td>
<td>55 x</td>
<td>55 x</td>
</tr>
<tr>
<td>256</td>
<td>20 x</td>
<td>29 x</td>
<td>29 x</td>
<td>29 x</td>
</tr>
</tbody>
</table>
OFSS performance gain

- Comparison with OFSS native
- Forwarding speedup factor:
  - \((\text{Mpps BEBA})/(\text{Mpps OFSS})\)

<table>
<thead>
<tr>
<th></th>
<th>1 core</th>
<th>2 core</th>
<th>3 core</th>
<th>4 core</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>4.3 Mpps</td>
<td>8.52 Mpps</td>
<td>11.70 Mpps</td>
<td>13.8 Mpps</td>
</tr>
<tr>
<td>128</td>
<td>3.9 Mpps</td>
<td>7.4 Mpps</td>
<td>8.4 Mpps</td>
<td>8.4 Mpps</td>
</tr>
<tr>
<td>256</td>
<td>3.1 Mpps</td>
<td>4.52 Mpps</td>
<td>4.52 Mpps</td>
<td>4.52 Mpps</td>
</tr>
</tbody>
</table>
"NO! Try not! DO or DO NOT, There is no try."
Bonus contents
Libpcap+fanout

- Pcap library extension aims at implement a generic fanout for different sockets!
- An implementation for TPACKET3 and PFQ is available!
- A new API is implemented:
  ```c
  int pcap_fanout(pcap_t *p, int fanout_id, const char *fanout)
  ```
- Legacy applications do not require to be changed!
- Groups of sockets (fanout id) and the fanout algorithm can be specified:
  - in a configuration file (e.g. pcap.conf)
    - A declarative grammar is shared across different sockets!
  - With environment variables: PCAP_FANOUT, PCAP_GROUP etc.
## Fanout algorithms

<table>
<thead>
<tr>
<th>TPACKET v3</th>
<th>PFQ</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash</td>
<td>steer_p2p</td>
<td>Symm. hash (ipv4)</td>
</tr>
<tr>
<td>lb</td>
<td>steer_rrobin</td>
<td>Isomorphic</td>
</tr>
<tr>
<td>cpu</td>
<td>-</td>
<td>Cpu id (of the receiving core)</td>
</tr>
<tr>
<td>rnd</td>
<td>-</td>
<td>Random (useful?)</td>
</tr>
<tr>
<td>rollover</td>
<td>-</td>
<td>Useful?</td>
</tr>
<tr>
<td>qm</td>
<td>steer_rrs</td>
<td>Recorded queue vs RSS hash</td>
</tr>
</tbody>
</table>
Pcap fanout: TPACKET3 vs PFQ

- Fanout performance is not affected by the no. of sockets (rather by the no. of CPU involved!)
Parallelism at large
Enif-lang in a nutshell...

- Instantiate an engine
  - A master thread/process with PFQ API
  - Declared in a pcap config. file
  - By means of run-enif application
- Define the endpoints
  - Threads, processes or NICs...
    - Ingress-points bind to a group
    - Egress-points join a group to receive traffic
  - Applications via PFQ/pcap sockets
    - No modifications to applications are required
- Load the enif-lang program...
  - Dynamic update (hot-swappable)
Memory order (wikipedia)

I wish all architectures had total-memory order...

<table>
<thead>
<tr>
<th>Type</th>
<th>Alpha</th>
<th>ARMv7</th>
<th>PA-RISC</th>
<th>POWER</th>
<th>SPARC RMO</th>
<th>SPARC PSO</th>
<th>SPARC TSO</th>
<th>x86</th>
<th>x86 oostore</th>
<th>AMD64</th>
<th>IA-64</th>
<th>z/Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads reordered after loads</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Loads reordered after stores</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Stores reordered after stores</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Stores reordered after loads</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atomic reordered with loads</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Atomic reordered with stores</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependent loads reordered</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incoherent instruction cache pipeline</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>