BROCADE

DPDK performance
Lessons learned in vRouter

Stephen Hemminger
stephen@networkplumber.org
@networkplumber

Agenda

- DPDK Overview
- Performance Goals
- Historical viewpoint
- Design choices
- Resources
- Lessons learned

DPDK Overview

- Environment Abstraction Layer
- Poll Mode Drivers
- Optimized algorithms
- Sample applications
- Tests

Environment Abstraction Layer

- DPDK on Linux
  - Dedicated threads
  - Pinned memory
  - PCI bus access

Poll Mode Drivers

- IXGBE – Intel 10G
- IGB/E1000 – Intel 1G
- Virtio – KVM
- VMXNET3 – Vmware
- I40E – Intel 40G
- Broadcom/Qlogic – Bnx2x
- Mellanox
- ...

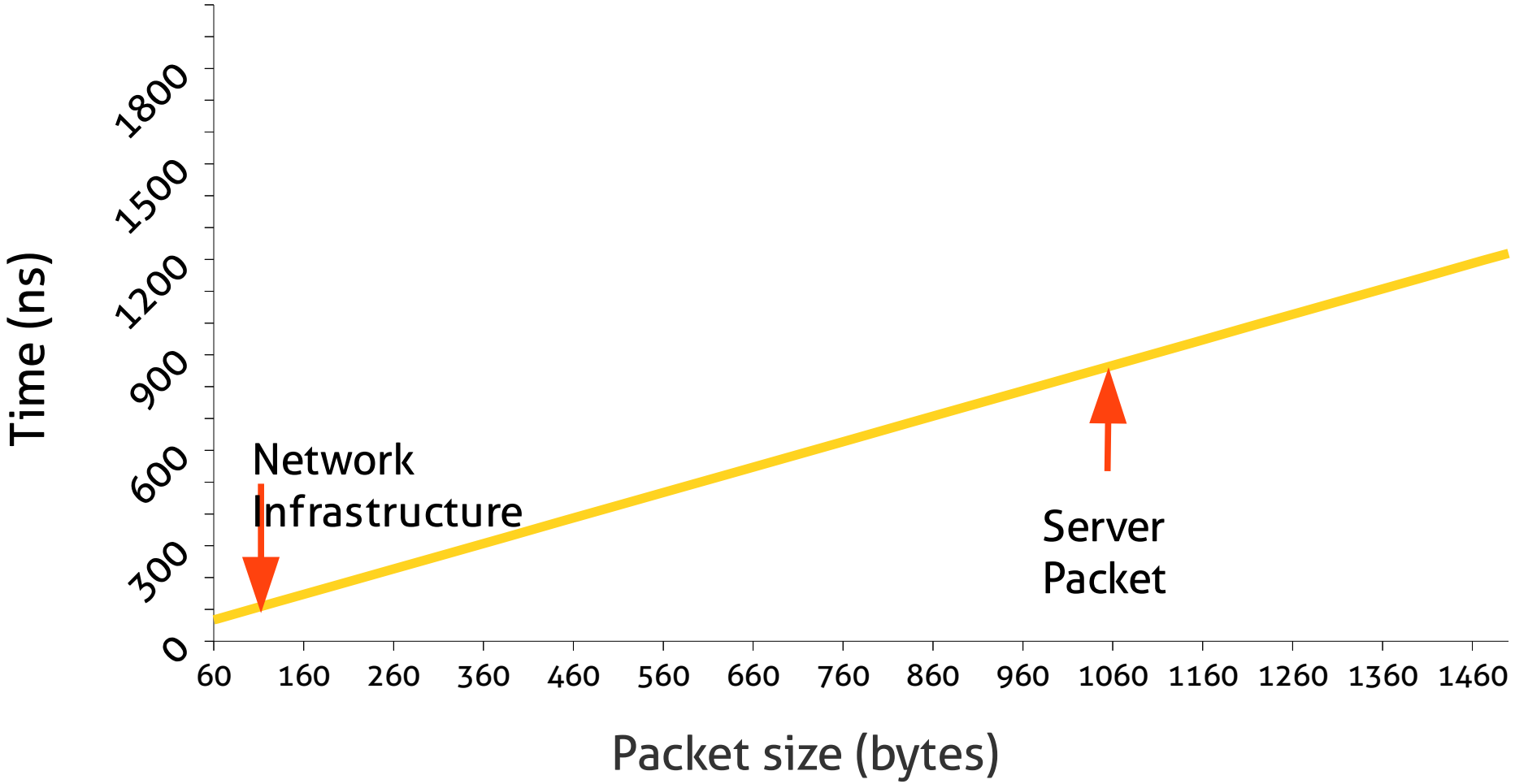
Optimized Algorithms

- Longest Prefix Match
- Hash – variable and 4 byte
- Match (ACL)
- IP fragmentation
- Memory – mbuf, pool, malloc
- Ring
- QoS
- Timer

Demo Applications

- Examples
  - L2fwd → Bridge/Switch
  - L3fwd → Router
  - L3fwd-acl → Firewall
  - Load_balancer
  - qos_sched → Quality Of Service

Packet time vs size



Time Budget

- Packet
  - 67.2ns = 201 cycles @ 3Ghz
- Cache
  - L3 = 8 ns
  - L2 = 4.3
- Atomic operations
  - Lock = 8.25 ns
  - Lock/Unlock = 16.1

Some basics ...

	Sandy Bridge Ivy Bridge	Haswell	Skylake
L1 data access (cycles)	4	4	4
L1 Peak Bandwidth (bytes/cycle)	2x16	2x32 load 1x32 store	2x32 load 1x32 store
L2 data Access (cycles)	12	11	12
L2 peak bandwidth (bytes/cycle)	1x32	64	64
Shared L3 Access (cycles)	26-31	34	44
L3 peak bandwidth (bytes/cycle)	32	-	32
Data hit in L2 or L1D Dcache of another core	43 – clean hit 60 – modified hit		

- BUT memory is ~70+ ns away (i.e. 2.0 GHz = 140+ cycles)

Source: Intel® 64 and IA-32 Architectures: Optimization Reference Manual

DPDK
DATA PLANE DEVELOPMENT KIT

The CPU Core

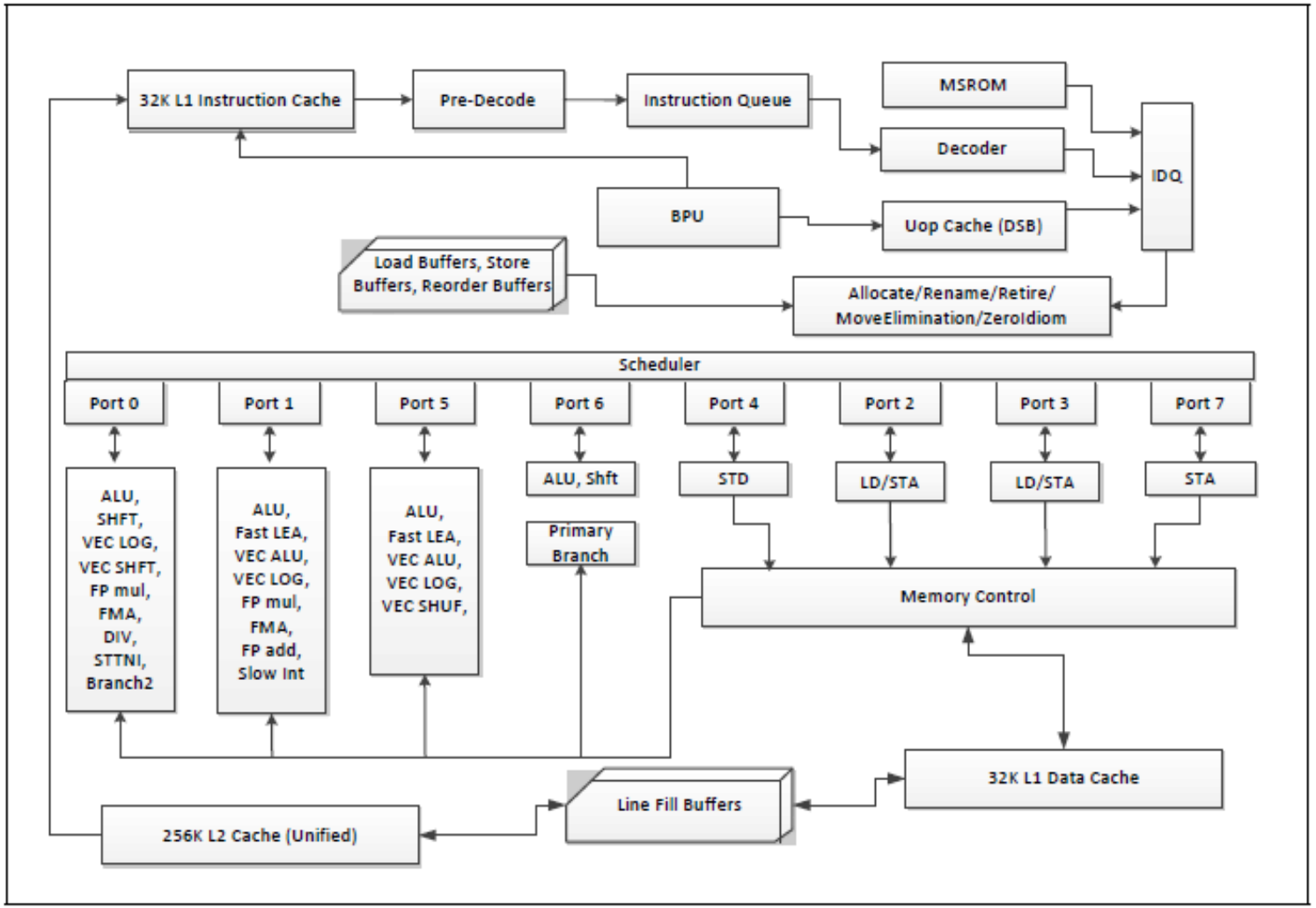


Figure 2-2. CPU Core Pipeline Functionality of the Haswell Microarchitecture

Intel® 64 and IA-32 Architectures: Optimization Reference Manual

DPDK
DATA PLANE DEVELOPMENT KIT

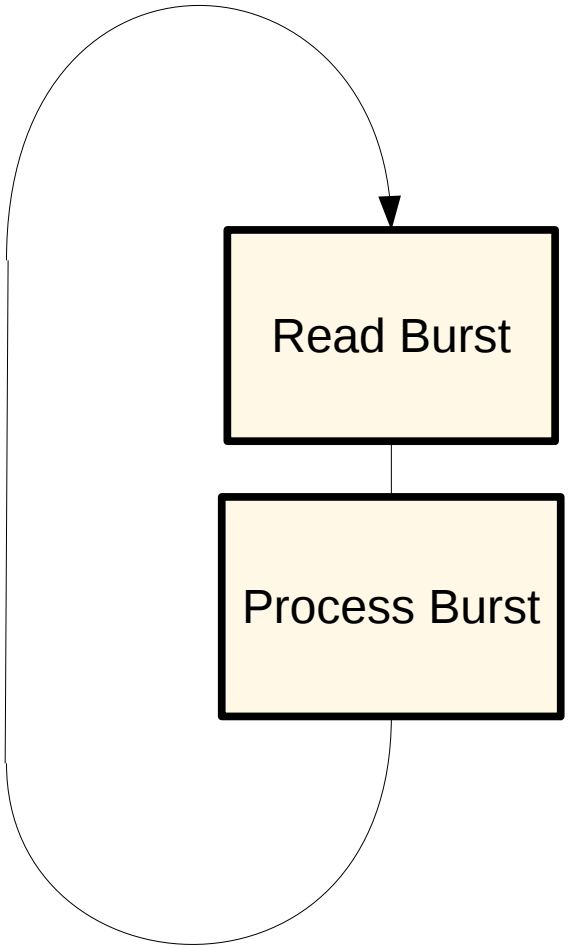
Architecture choices

- Legacy
  - Existing proprietary code
- BSD clone
  - Reuse permissive licensed code
- Buy
- Build
  - Incremental development

Mutual Exclusion

- Locking
  - Reader/Writer lock is expensive
  - Read lock more overhead than spin lock
- Userspace RCU
  - Don't modify, create and destroy
  - Impacts thread model

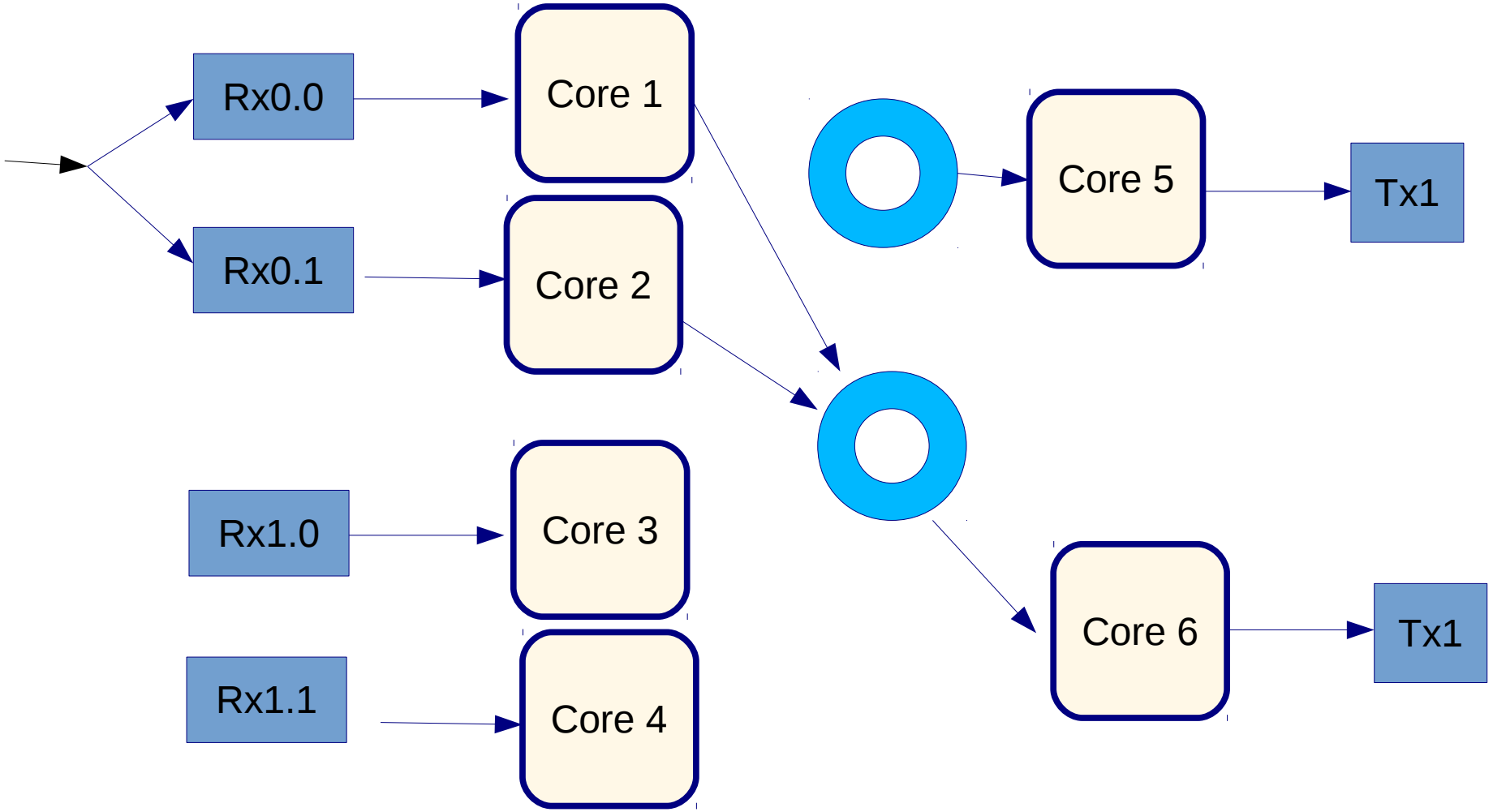
Forwarding thread



Statistics:
- Received Packets
- Transmit Packets

- Iterations
- Packets processed

Split thread model



Internal Instrumentation

Core	Interface	RX Rate	TX Rate	Idle
1	p1p1	14.9M		0
2	p1p1	0		250
3	p33p1	0		250
4	p33p1	1		250
5	p1p1		0	250
6	p33p1		11.9M	1

Memory Layout

- Cache killers
  - Linked lists
  - Poor memory layout
  - Global statistics
  - Atomic
- Use carefully
  - Prefetching
  - Inlining

Perf – active thread

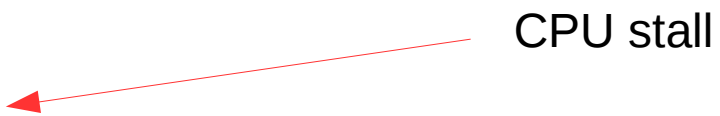
```
Samples: 16K of event 'cycles', Event count (approx.): 11763536471
 14.93%  dataplane  [.] ip_input
 10.04%  dataplane  [.] ixgbe_xmit_pkts
  7.69%  dataplane  [.] ixgbe_recv_pkts
  7.05%  dataplane  [.] T.240
  6.82%  dataplane  [.] fw_action_in
  6.61%  dataplane  [.] fifo_enqueue
  6.44%  dataplane  [.] flow_action_fw
  6.35%  dataplane  [.] fw_action_out
  3.92%  dataplane  [.] ip_hash
  3.69%  dataplane  [.] cds_lfht_lookup
  2.45%  dataplane  [.] send_packet
  2.45%  dataplane  [.] bit_reverse_ulong
```

Speed killer's

- I/O
- VM exit's
- System call's
- PCI access
- HPET
- TSC
- Floating Point
- Cache miss
- CPU pipeline stall

TSC counter

```
while(1)
   cur_tsc = rte_rdtsc();
   diff_tsc = cur_tsc – prev_tsc;

   if (unlikely(diff_tsc > drain_tsc)) {
      for (portid = 0; portid < RTE_MAX_ETHPORTS;
portid++) {

         send_burst(qconf,
              qconf->tx_mbufs[portid].len,
              portid);
```



Heisenburg: observing performance slows it down

Idle sleep

- 100% Poll → 100% CPU
  - CPU power limits
  - No Turbo boost
  - PCI bus overhead
- Small sleep's
  - 0 - 250us
  - Based on activity

fw_action_in

Memset overhead

```
        |   struct ip_fw_args fw_args = {
        |                     .m = m,
        |                     .client = client,
        |                     .oif = NULL };
  1.54  |1d:    movzbl %sil,%esi
  0.34  |       mov    %rsp,%rdi
  0.04  |       mov    $0x13,%ecx
  0.16  |       xor    %eax,%eax
 57.66  |       rep    stos %rax,%es:(%rdi)
  4.68  |       mov    %esi,0x90(%rsp)
 20.45  |       mov    %r9,(%rsp)
```

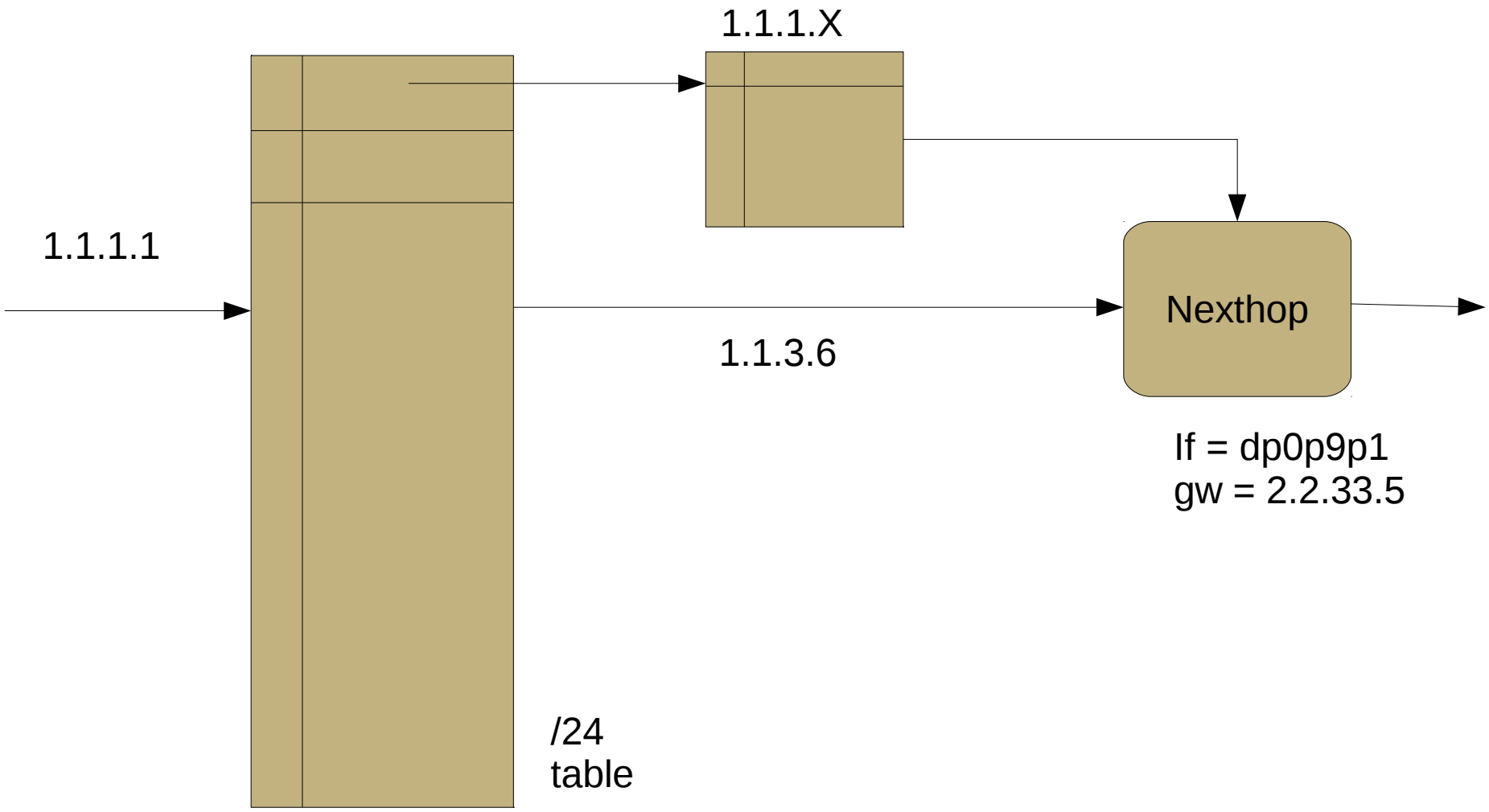
DPDK

BROCADE

Why is QoS slow?

```
static inline void
rte_sched_port_time_resync(struct rte_sched_port *port)
{
     uint64_t cycles = rte_get_tsc_cycles();
     uint64_t cycles_diff = cycles - port->time_cpu_cycles;
     double bytes_diff = ((double) cycles_diff) /
                         port->cycles_per_byte;

     /* Advance port time */
     port->time_cpu_cycles = cycles;
     port->time_cpu_bytes += (uint64_t) bytes_diff;
```

Longest Prefix Match



LPM issues

- Prefix → 8 bit next hop
- Missing barriers
- Rule update
- Fixed size /8 table

Conclusion

- DPDK can be used to build fast router
  - 12M pps per core
- Lots of ways to go slow
  - Fewer ways to go fast

Q & A

Thank you

Stephen Hemminger
stephen@networkplumber.org
@networkplumber

PCI passthrough

- I/O TLB size
  - Hypervisor uses IOMMU to map guest
  - IOMMU has small TLB cache
  - Guest I/O exceeds TLB
- Solution
  - 1G hugepage on host KVM
  - Put Guest in huge pages
  - Only on KVM – requires manual configuration

DPDK Issues

- Static configuration
  - Features
  - CPU architecture
  - Table sizes
- Machine specific initialization
  - # of Cores, Memory Channels
- Poor device model
  - Works for Intel E1000 like devices

Slowpath

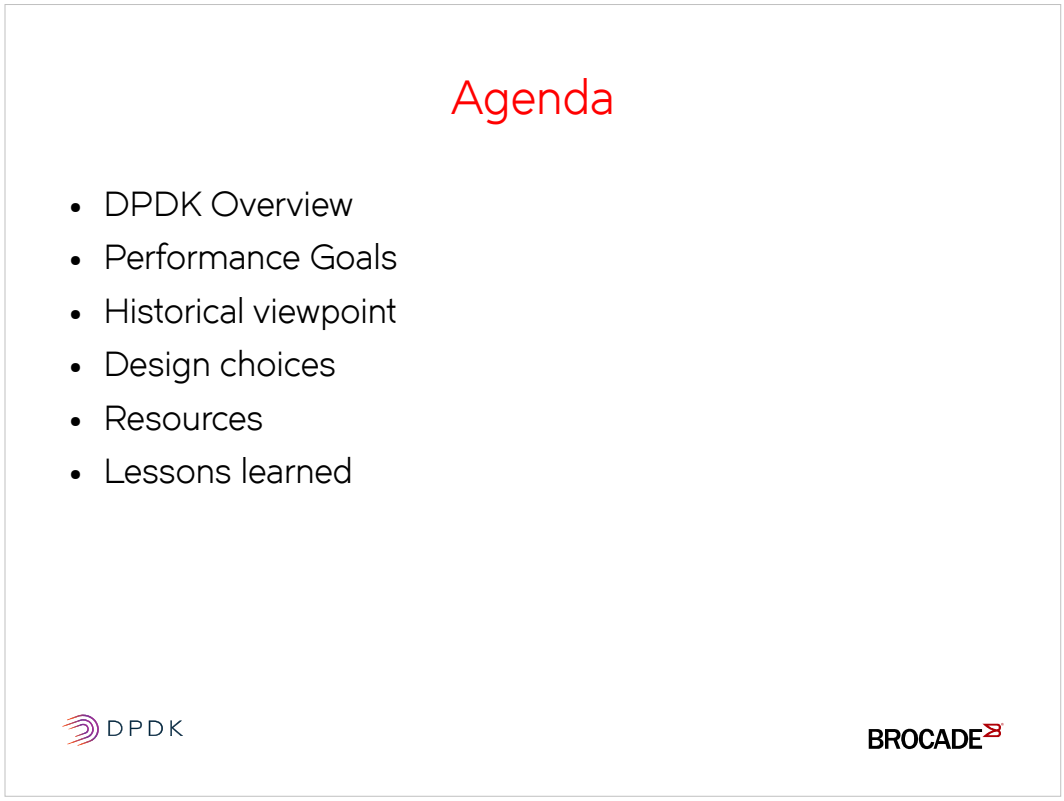
- Packets placed in DPDK rte_ring
  - Wakeup via eventfd
- Shadow thread
  - Poll's for event or kernel packets
- Packet's received
  - Sent to kernel via TAP device
- Local packets
  - injected into Tx Thread

Hello

My name is Stephen Hemminger and I am the chief Architect of the Brocade virtual Router product. You can follow me on twitter at @networkplumber.

In 2012, Intel released the Dataplane Development Kit and Vyatta was one of the first companies to develop an accelerated software router using it. This became known as the virtual Router (vRouter) after Brocade acquired Vyatta 3 years ago

This talk describes that journey.

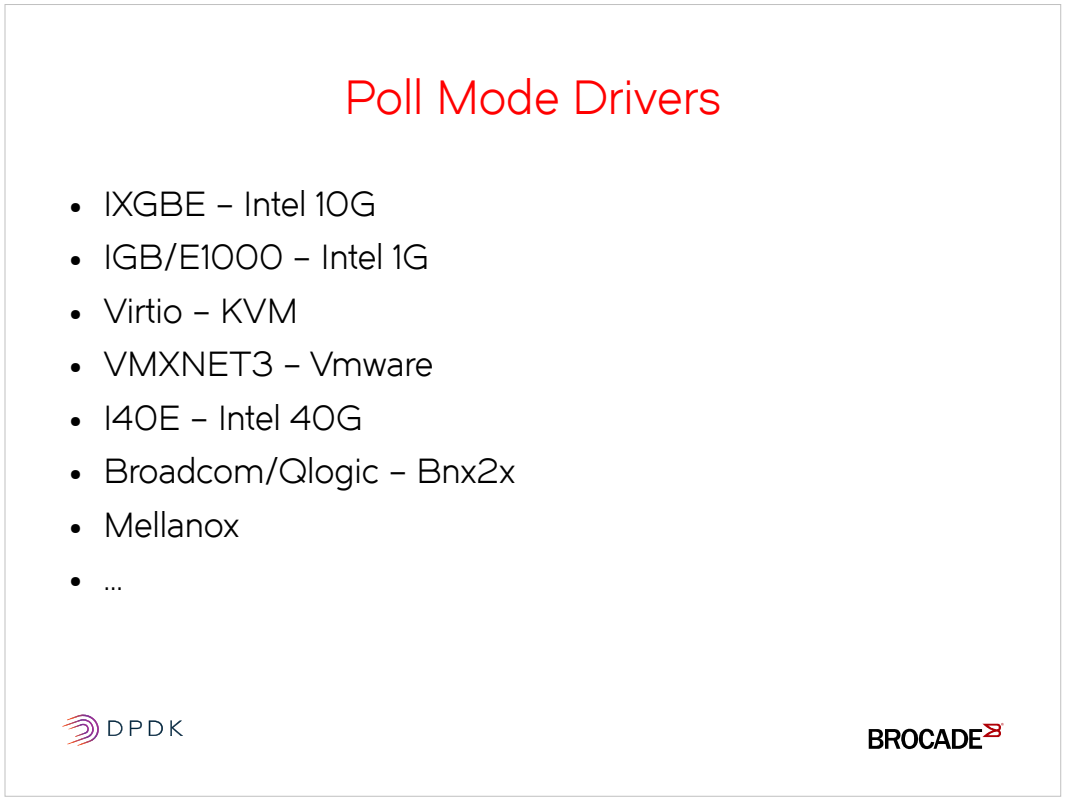


I will begin with overview of what is the DPDK.
Then talk about what the performance goals we were trying to achieve.
In order to provide some context, first we need to take a look inside modern hardware. I know you thought this was a software talk (joke).
This leads to several key design decisions.
Finally, the good part some of the lessons learned in the process.

The Intel DPDK contains 5 main areas

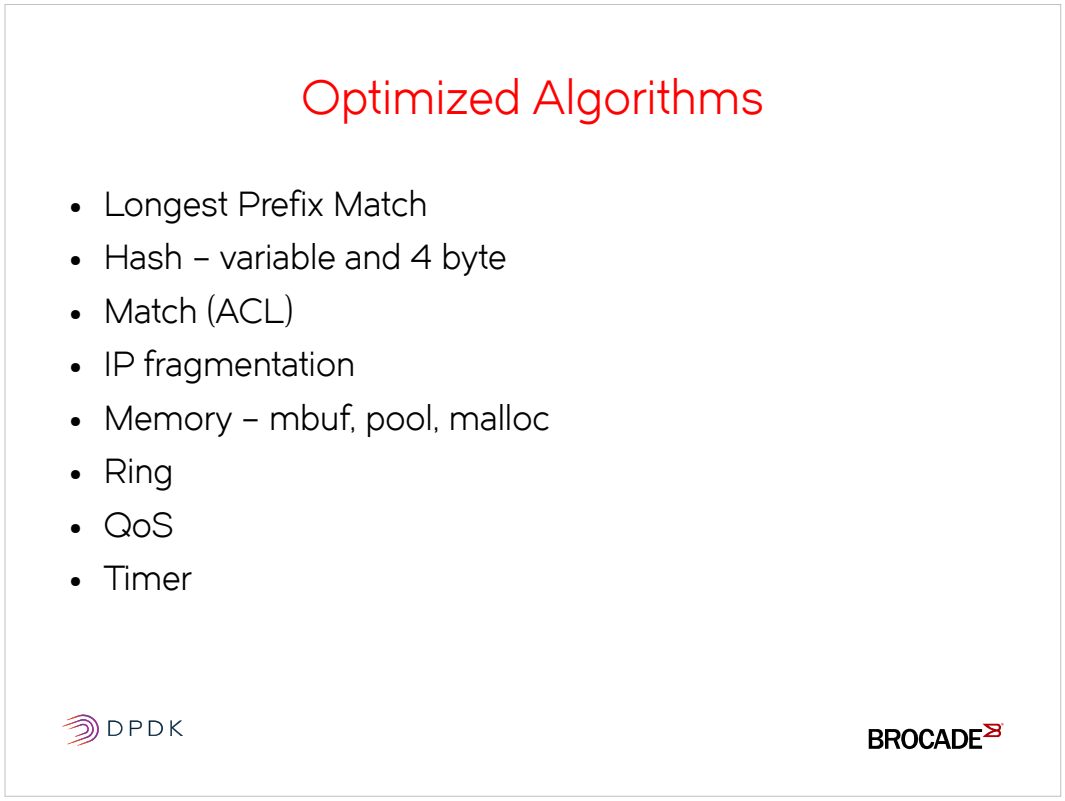
The DPDK is almost a stripped down mini-operating system itself. In many ways writing DPDK applications is like writing kernel drivers.

Originally DPDK was built to run on both Linux and bare metal. The bare metal support is mostly gone now because it is much easier to development in a full OS environment and the Linux scheduler is now able to achieve near baremetal performance for dedicated real time processes.



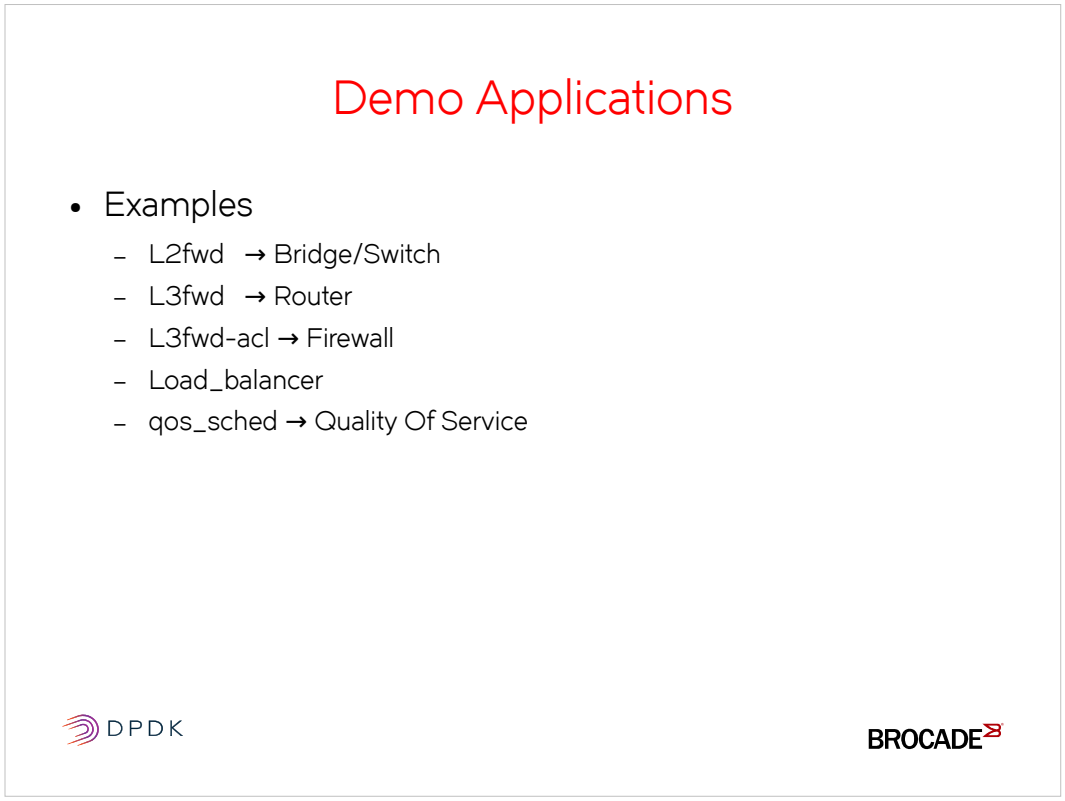
Ixgbe was the starting point of DPDK development. Intel, 6Wind, and Brocade all developed Virtio and VMXnet3 drivers in parallel; the project had not started to collabrate yet.

Intel and other vendors have gone to provide more physical and virtual devices.



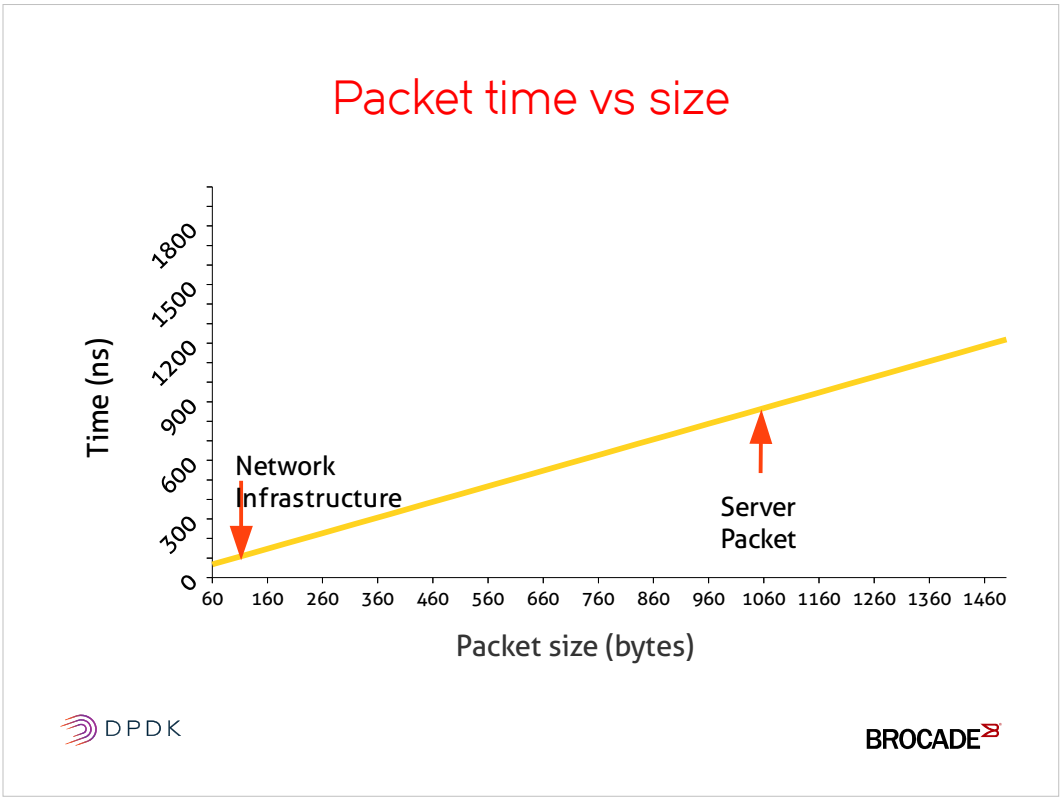
The normal glibc routines are often too general and have performance killers (like having to be thread safe).

The core of the DPDK is really the lockless ring and memory mangement.



What got us excited was seeing the performance potential in the demo applications. These are like the old technical notes that came with hardware.

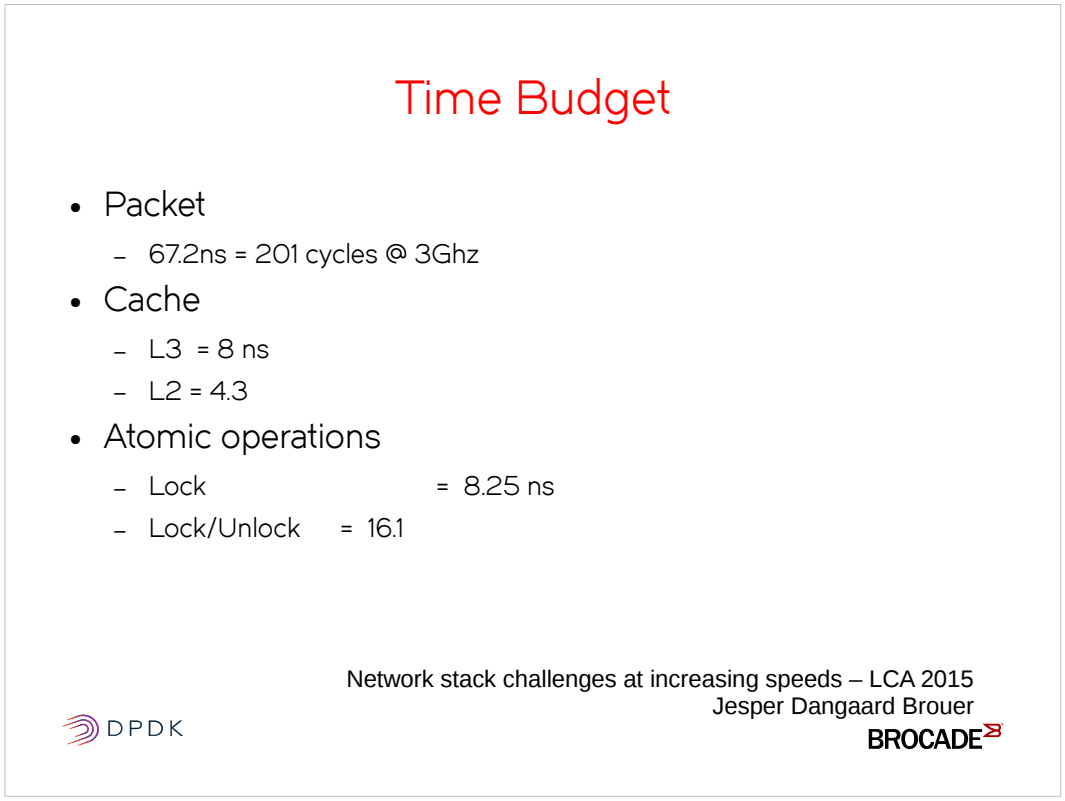
But it is important to realize that these are so stripped down that they do not match real world.



Linux on most hardware can process 1M packets per second per CPU core. This has gotten better in last few years but we wanted to do an order of magnitude better to be competive in Software Defined Networking.

A typical server packet is 1000 bytes. But Network operators look at small packet performance.

The goal of the vRouter was to process smallest size packets on a 10G bit interface with 1 CPU.



At 10G bit/sec, there is 67.2ns to process a packet

A cache miss is 8 nanoseconds

A lock operation (uncontended) is 8.25 ns

Some basics ...

	Sandy Bridge Ivy Bridge	Haswell	Skylake
L1 data access (cycles)	4	4	4
L1 Peak Bandwidth (bytes/cycle)	2x16	2x32 load 1x32 store	2x32 load 1x32 store
L2 data Access (cycles)	12	11	12
L2 peak bandwidth (bytes/cycle)	1x32	64	64
Shared L3 Access (cycles)	26-31	34	44
L3 peak bandwidth (bytes/cycle)	32	-	32
Data hit in L2 or L1D Dcache of another core	43 – clean hit 60 – modified hit		

- BUT memory is ~70+ ns away (i.e. 2.0 GHz = 140+ cycles)

Source: Intel® 64 and IA-32 Architectures: Optimization Reference Manual

DPDK
DATA PLANE DEVELOPMENT KIT

This slide is from Intel, it shows in more detail the overhead for each type of cache miss.

The bottom line is that even one full cache miss to memory means not being able to meet the performance goal!

The CPU Core

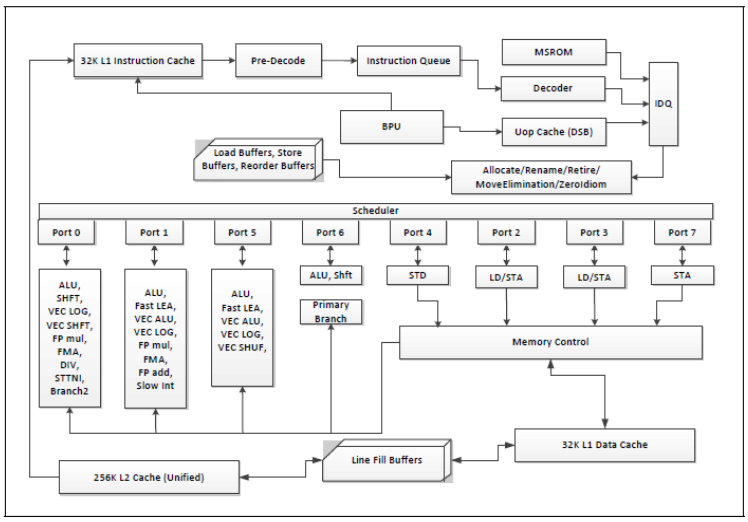


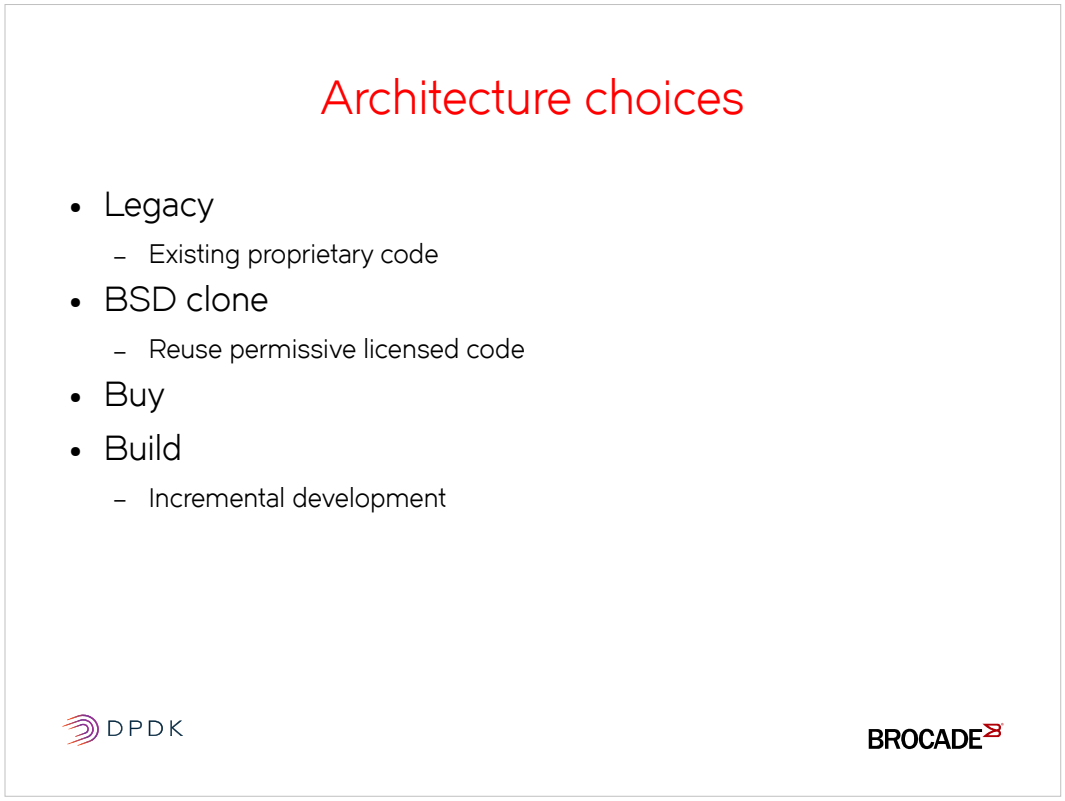
Figure 2-2. CPU Core Pipeline Functionality of the Haswell Microarchitecture

Intel® 64 and IA-32 Architectures: Optimization Reference Manual

DPDK
DATA PLANE DEVELOPMENT KIT

CPU's are no longer just a single flow. Modern CPU's have multiple execution units. The goal of high performance software is to keep all these execution units busy.

You can measure this with perf, and it is often surprising. Most programs are lucky to keep 2 pipeline's busy.



Now that we know the building blocks, it was time to chose the architecture. We had four choices. Legacy vendors like Cisco and Juniper have very rich software stacks and can use those but they are not designed to work in this environment.

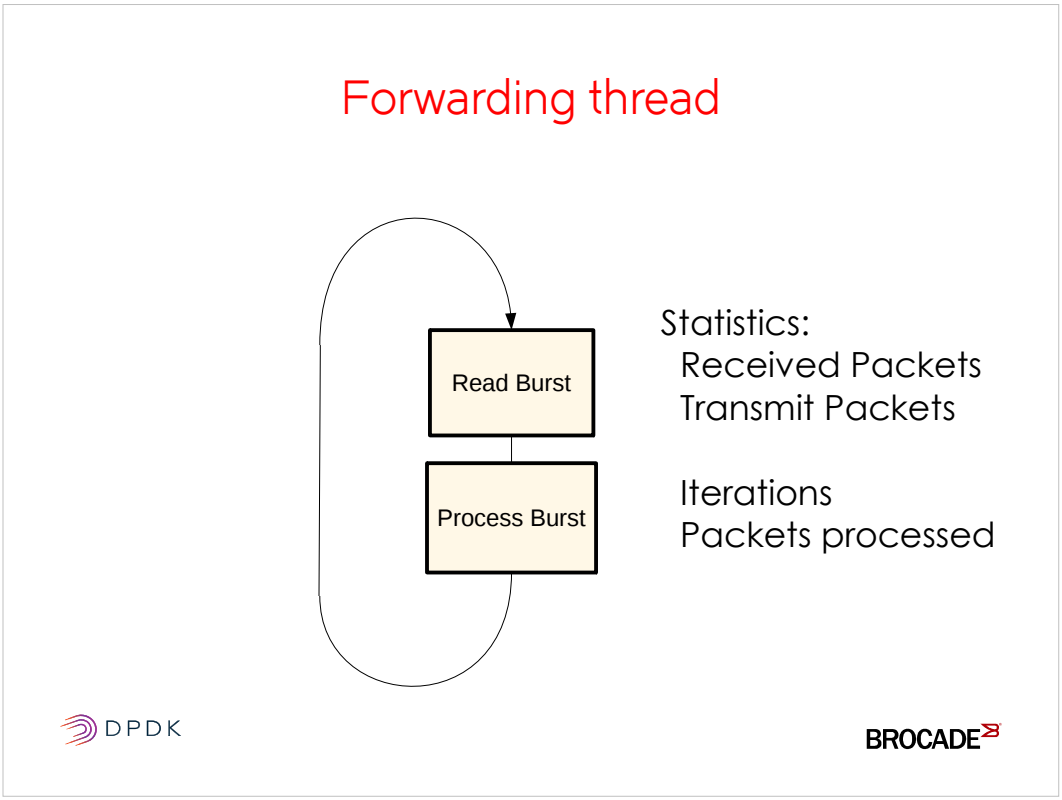
We also considered just using FreeBSD. But the SMP design was more primitive than Linux and would not do what we wanted either.

Several vendors offered proprietary stacks but they were too expensive, especially for a startup like Vyatta.

So we decided to build it from scratch using available permissive software.

Traditional SMP locking is safe but expensive. Every spin lock requires a locked operation on the bus. Remember we wanted to keep all those execution units busy! Reader/write locking is even more expensive. For the uncontended case a read lock takes more overhead than a spin lock.

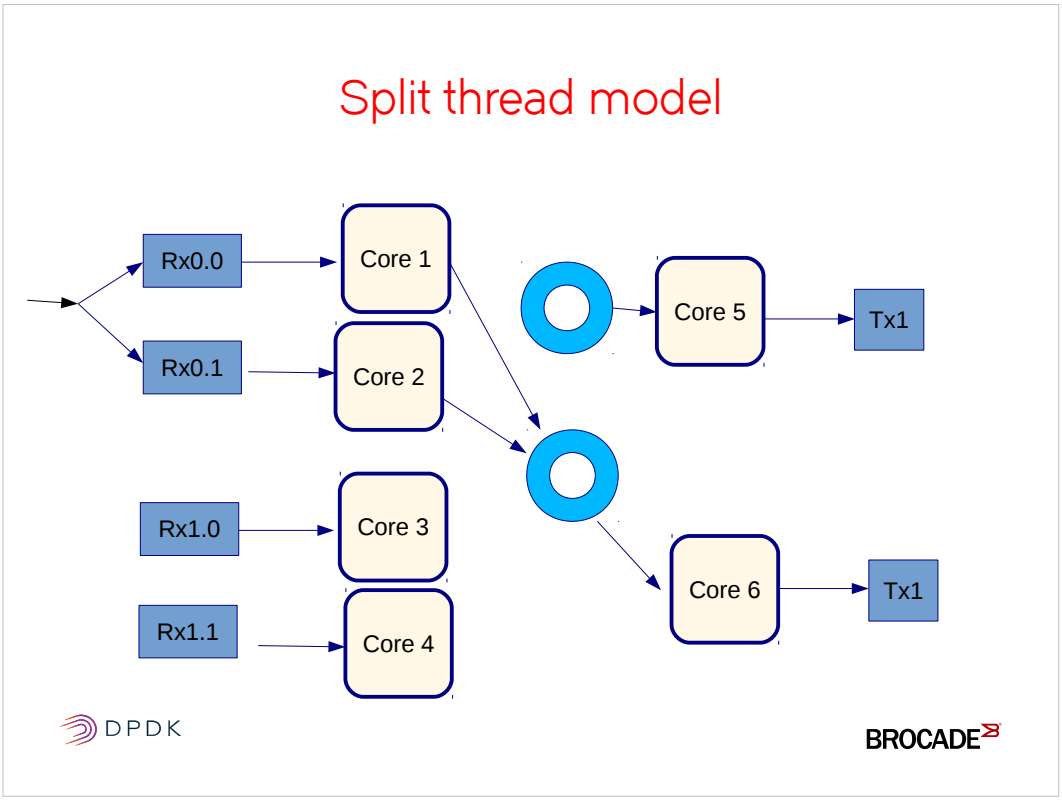
Instead we use the LGPL Userspace RCU library when ever possible. RCU is not part of the DPDK but goes well with the thread design of DPDK applications.



The thread design of most DPDK applications is what is known as “Run to completion” or “Hungry puppies”.

Each thread polls from one or more sources than processes that burst of packets.

There is a natural “grace period” for Read-Copy-Update to work.



The vRouter works like the Intel demo applications.

Packets are divided in hardware into multiple receive queues. This queues are polled by dedicated CPU cores that process the packets and put them into a ring between threads.

The transmit CPU's read from the ring and feed the transmit queue in the hardware.

Internal Instrumentation

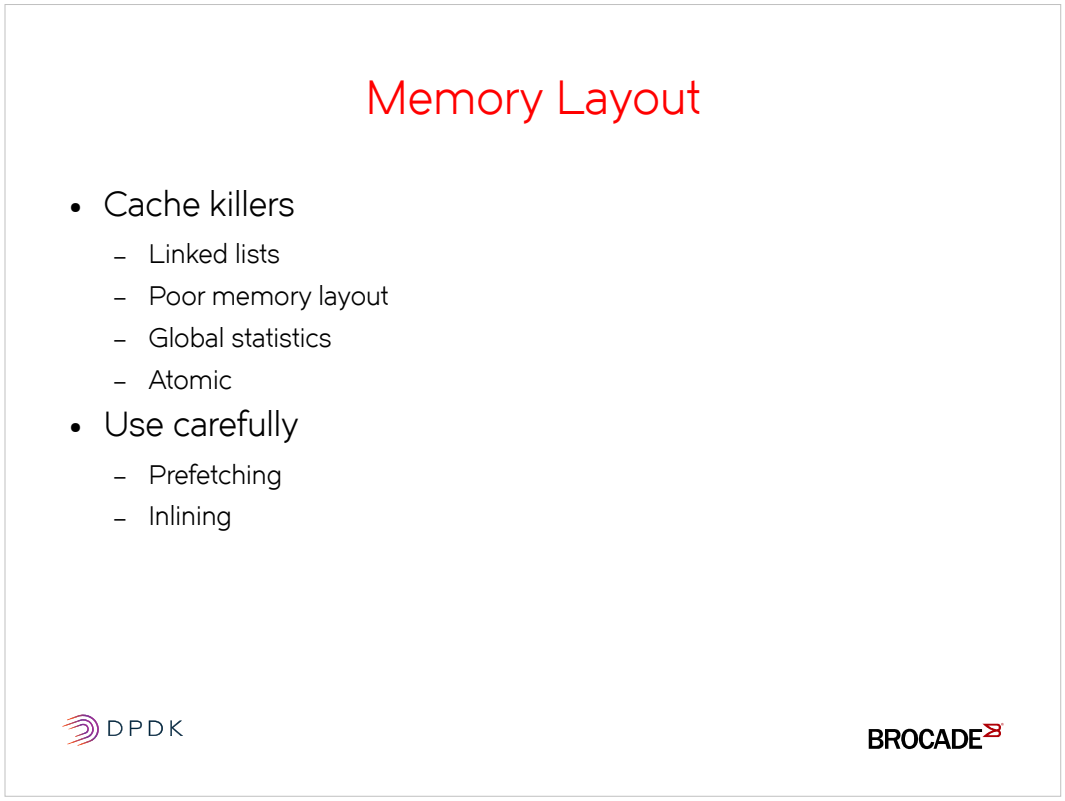
Core	Interface	RX Rate	TX Rate	Idle
1	p1p1	14.9M		0
2	p1p1	0		250
3	p33p1	0		250
4	p33p1	1		250
5	p1p1		0	250
6	p33p1		11.9M	1

DPDK

BROCADE

For development, we added internal measurements of the number of packets processed per second by each core. There is also a more detailed model of the load on each interface.

Using this we can see the receive rate of one cpu, and the transmit out the other. Note: on this system the transmit is on a slower slot and can not achieve line rate.



In order to achieve this performance it is important to think about using cache effectively. That means no cache hostile algorithms like linked lists. Also being very careful where elements are layed out in data structures.

Perf – active thread

```
Samples: 16K of event 'cycles', Event count (approx.): 11763536471
 14.93%  dataplane  [.] ip_input
 10.04%  dataplane  [.] ixgbe_xmit_pkts
  7.69%  dataplane  [.] ixgbe_recv_pkts
  7.05%  dataplane  [.] T.240
  6.82%  dataplane  [.] fw_action_in
  6.61%  dataplane  [.] fifo_enqueue
  6.44%  dataplane  [.] flow_action_fw
  6.35%  dataplane  [.] fw_action_out
  3.92%  dataplane  [.] ip_hash
  3.69%  dataplane  [.] cds_lfht_lookup
  2.45%  dataplane  [.] send_packet
  2.45%  dataplane  [.] bit_reverse_ulong
```

BROCADE

During development we also made extensive use of the Linux “perf” tool. This is an early example of the detail seen.

The actual data changes quite dynamically. Small changes in cache layout can have a big effect.

Often the code that is targeted as hot is not at fault, only getting blamed for a cache miss.

Speed killer's

- I/O
- VM exit's
- System call's
- PCI access
- HPET
- TSC
- Floating Point
- Cache miss
- CPU pipeline stall

DPDK

BROCADE

We have seen all of these.

TSC counter

```
while(1)
   cur_tsc = rte_rdtsc();
   diff_tsc = cur_tsc – prev_tsc;

   if (unlikely(diff_tsc > drain_tsc)) {
     for (portid = 0; portid < RTE_MAX_ETHPORTS;
portid++) {

        send_burst(qconf,
             qconf->tx_mbufs[portid].len,
             portid);
```

CPU stall

Heisenburg: observing performance slows it down

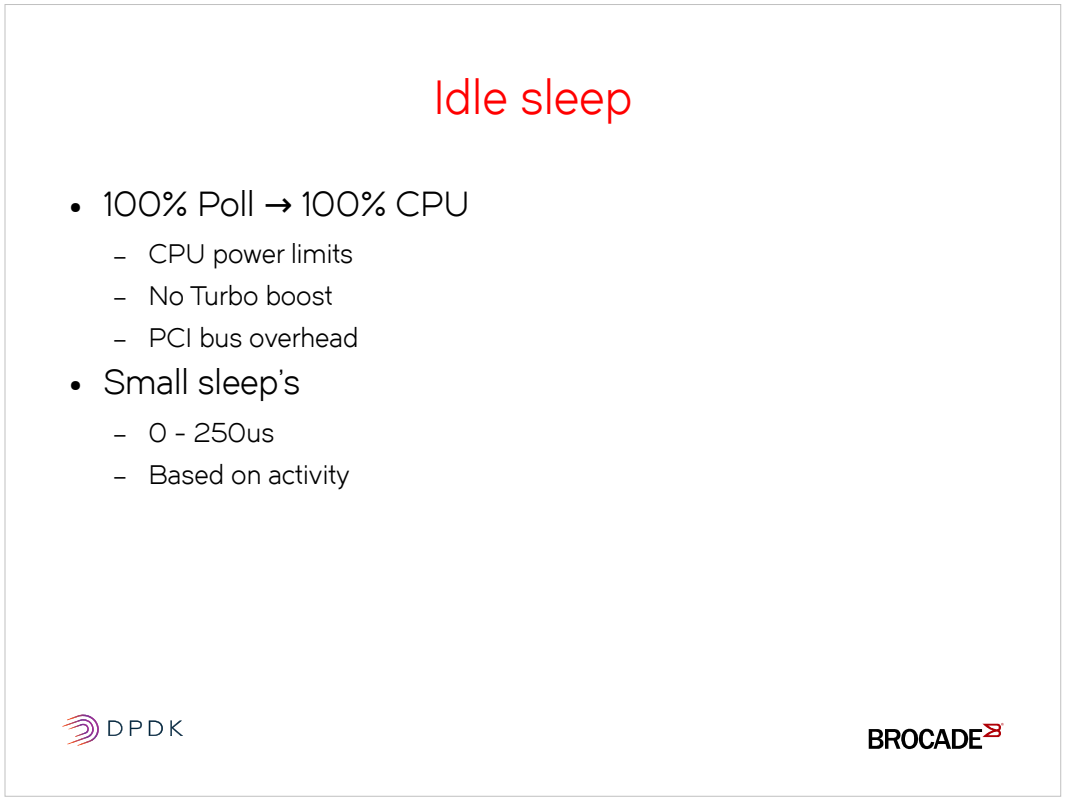
DPDK

BROCADE

This is an example from one of the Intel demo applications.

The operation to read the timestamp count register blocks the CPU, stalling multiple execution units until after the value is read.

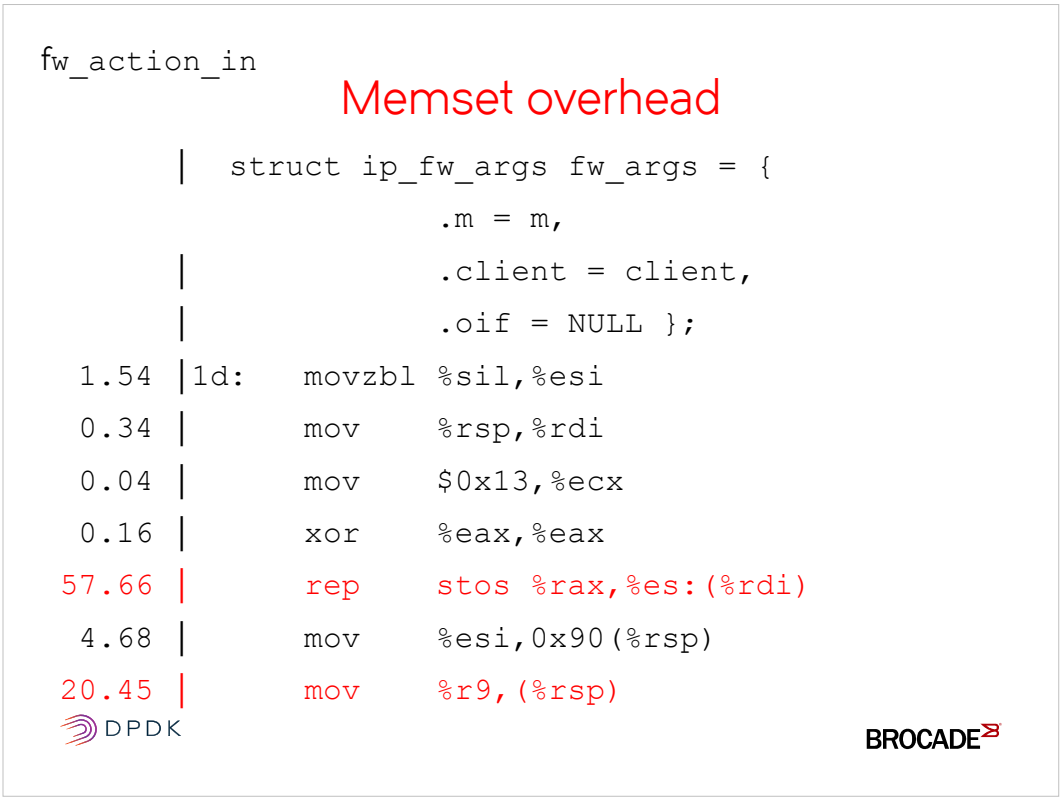
In doing these kind of things, the act of measuring the performance can slow it down.



Most of the Intel sample applications work by polling the CPU 100% of the time. This provides the lowest latency but often has worse performance.

Using 100% of the CPU means using 100% of the possible power budged of the CPU, and causes more PCI bus transactions.

To avoid this we used the example in the l3fwd power management application to sleep for small intervals when idle. And are also careful not to poll unused hardware ports.



Early in the development cycle, we used a lot of FreeBSD code. This code a coding style of creating internal data structures then passing that to other routines. The creation of these structures caused an implicit memory set. The memory set code in gcc would generate these repeat string instructions.

The repeat string instruction is a loop inside the CPU, and it keeps only one execution unit busy and the area on the stack can be cache stale.

The solution was to replace the FreeBSD code with better code.

Why is QoS slow?

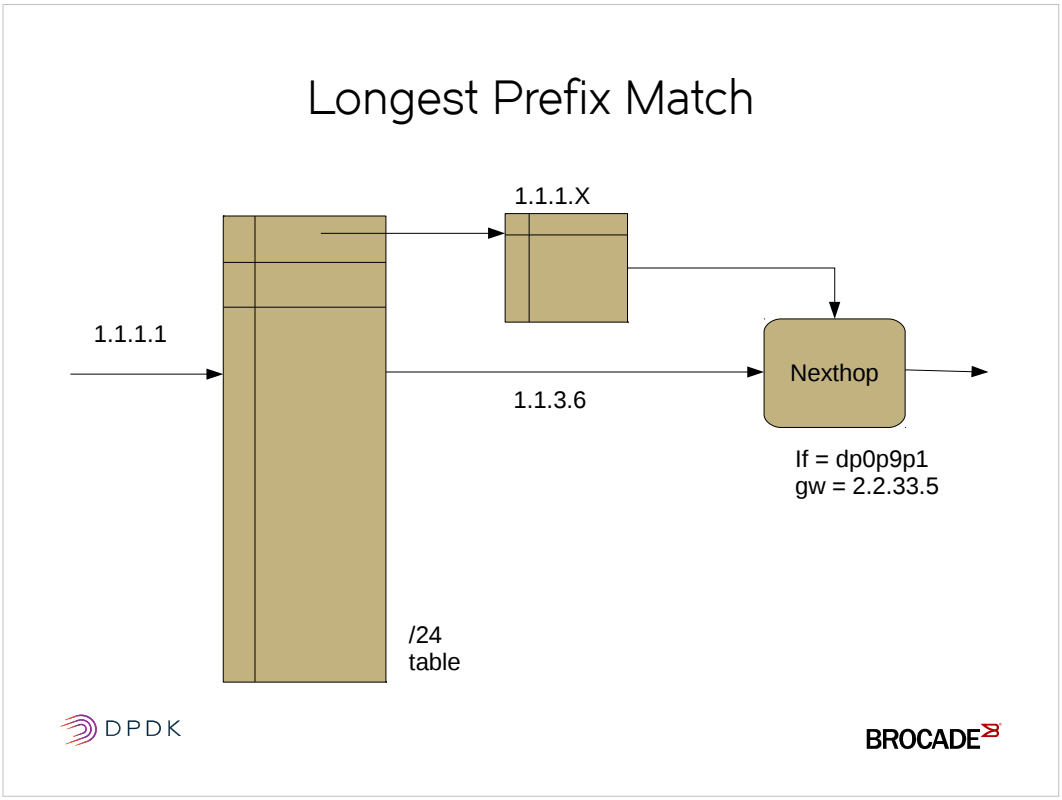
```
static inline void
rte_sched_port_time_resync(struct rte_sched_port *port)
{
      uint64_t cycles = rte_get_tsc_cycles();
      uint64_t cycles_diff = cycles - port->time_cpu_cycles;
      double bytes_diff = ((double) cycles_diff) /
                          port->cycles_per_byte;

      /* Advance port time */
      port->time_cpu_cycles = cycles;
      port->time_cpu_bytes += (uint64_t) bytes_diff;
```

BROCADE

Intel provide a rich QoS library which enabled hierarchical Quality Of Service. During testing it was discovered that enabling QoS was causing up to 20% drop in performance.

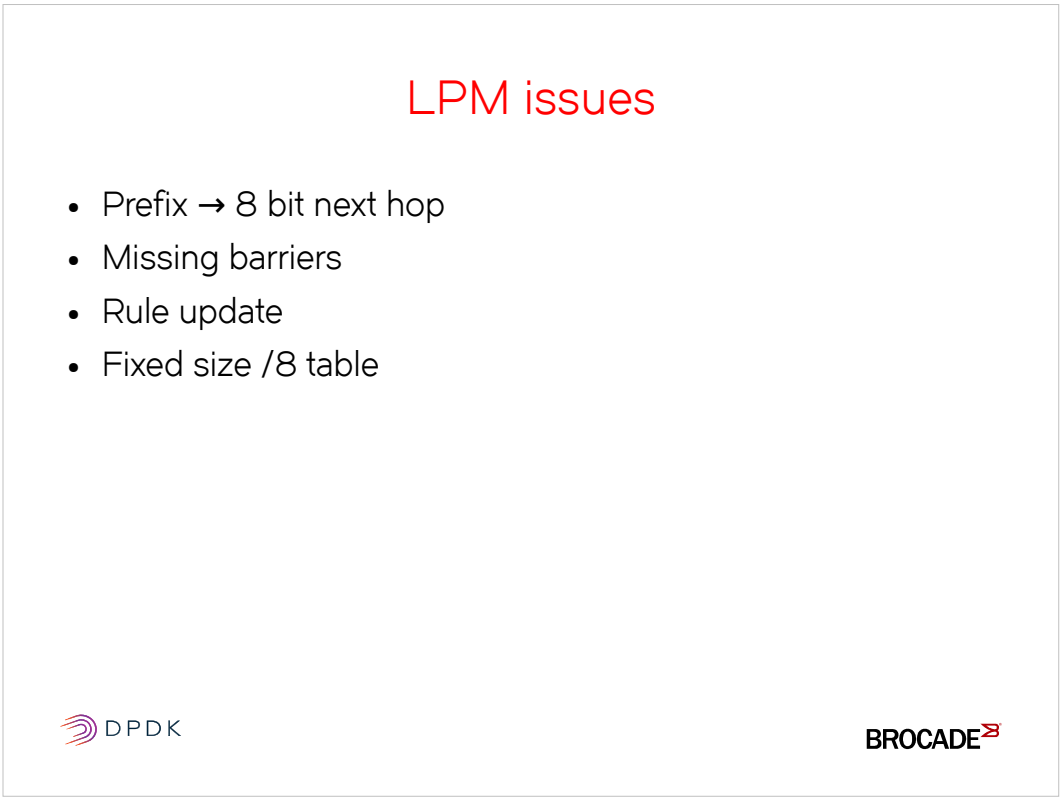
Using perf it was determined the problem was here. Can you see the problem?

The issues is that the code is doing a 64 bit floating point divide which one of the is the slowest instructions on the Intel Archiecture. The resolution was to convert this to a scaled integer multiply and the problem vanished.



One of the key algorithms in a router is Longest Prefix Match. This is the operation that looks up a destination address and returns the next hop gateway address and interface.

The DPDK provides the skeleton of a library for LPM. It uses a very large table to map 24 bits of the address to either a target or a sub-table. This is very similar to how routing lookup is done in hardware.



The DPDK LPM did not meet our needs because it would not scale. It was limited to 8 bits (256) entries for next hop; was missing key compiler barrires.

Also, it would not scale in testing to handling a millions of rules as is typically seen on backbone router.

Our solution was to extend the code to have wider entries and use a red-black tree for rule management.

We are working with the community to fold this back, but there are obstacles because it will be a major change to the existing Application Binary Interface.

Conclusion

- DPDK can be used to build fast router
  - 12M pps per core
- Lots of ways to go slow
  - Fewer ways to go fast

Q & A

Thank you

Stephen Hemminger
stephen@networkplumber.org
@networkplumber

DPDK

BROCADE

PCI passthrough

- I/O TLB size
  - Hypervisor uses IOMMU to map guest
  - IOMMU has small TLB cache
  - Guest I/O exceeds TLB
- Solution
  - 1G hugepage on host KVM
  - Put Guest in huge pages
  - Only on KVM – requires manual configuration

DPDK Issues

- Static configuration
  - Features
  - CPU architecture
  - Table sizes
- Machine specific initialization
  - # of Cores, Memory Channels
- Poor device model
  - Works for Intel E1000 like devices

Slowpath

- Packets placed in DPDK rte_ring
  - Wakeup via eventfd
- Shadow thread
  - Poll's for event or kernel packets
- Packet's received
  - Sent to kernel via TAP device
- Local packets
  - injected into Tx Thread