DPDK
DATA PLANE DEVELOPMENT KIT

DPDK Summit

VPP overview

Shwetha Bhandari
Developer@Cisco

Scalar Packet Processing

- A fancy name for processing one packet at a time
- Traditional, straightforward implementation scheme
- Interrupt, a calls b calls c … return return return
- Issues:
  - thrashing the I-cache (when code path length exceeds the primary I-cache size)
  - Dependent read latency (packet headers, forwarding tables, stack, other data structures)
  - Each packet incurs an identical set of I-cache and D-Cache misses

Packet Processing Budget

14 Mpps on 3.5 GHz CPU = 250 cycles per packet

Memory Read/Write latency

	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

Introducing VPP: the *vector packet processor*

Introducing VPP

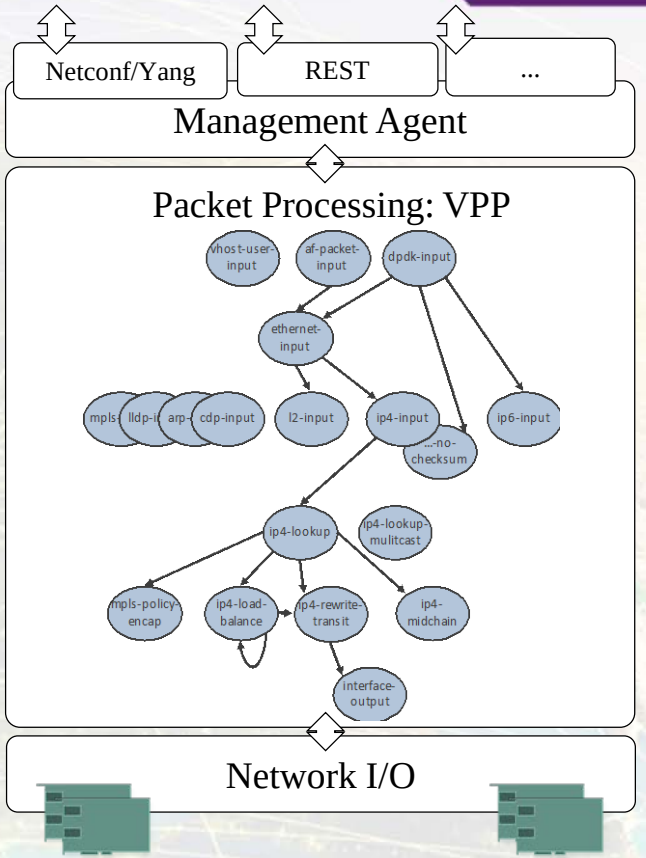
Accelerating the dataplane since 2002

Fast, Scalable and consistent

- 14+ Mpps per core
- Tested to 1TB
- Scalable FIB: supporting millions of entries
- 0 packet drops, ~15µs latency

Optimized

- **DPDK** for fast I/O
- **ISA:** SSE, AVX, AVX2, NEON ..
- **IPC:** Batching, no mode switching, no context switches, non-blocking
- **Multi-core:** Cache and memory efficient



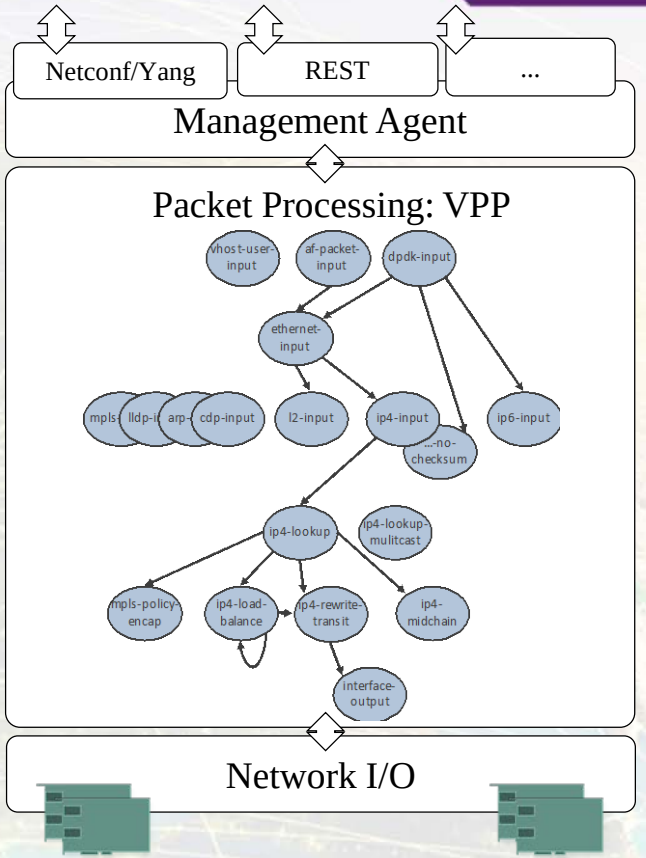
Introducing VPP

Extensible and Flexible modular design

- Implement as a directed graph of nodes
- Extensible with plugins, plugins are equal citizens.
- Configurable via CP and CLI

Developer friendly

- Deep introspection with counters and tracing facilities.
- Runtime counters with IPC and errors information.
- Pipeline tracing facilities, life-of-a-packet.
- Developed using standard toolchains.



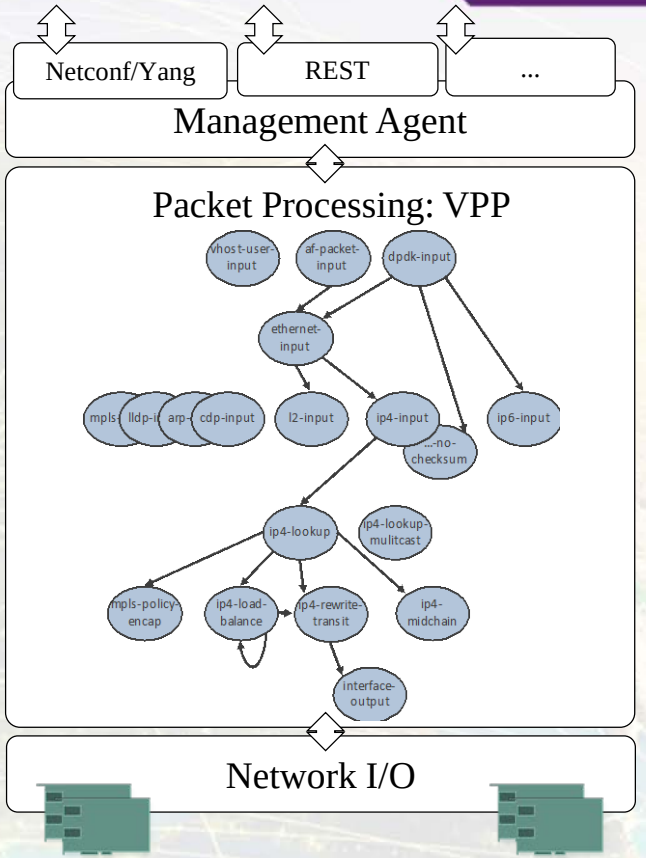
Introducing VPP

Fully featured

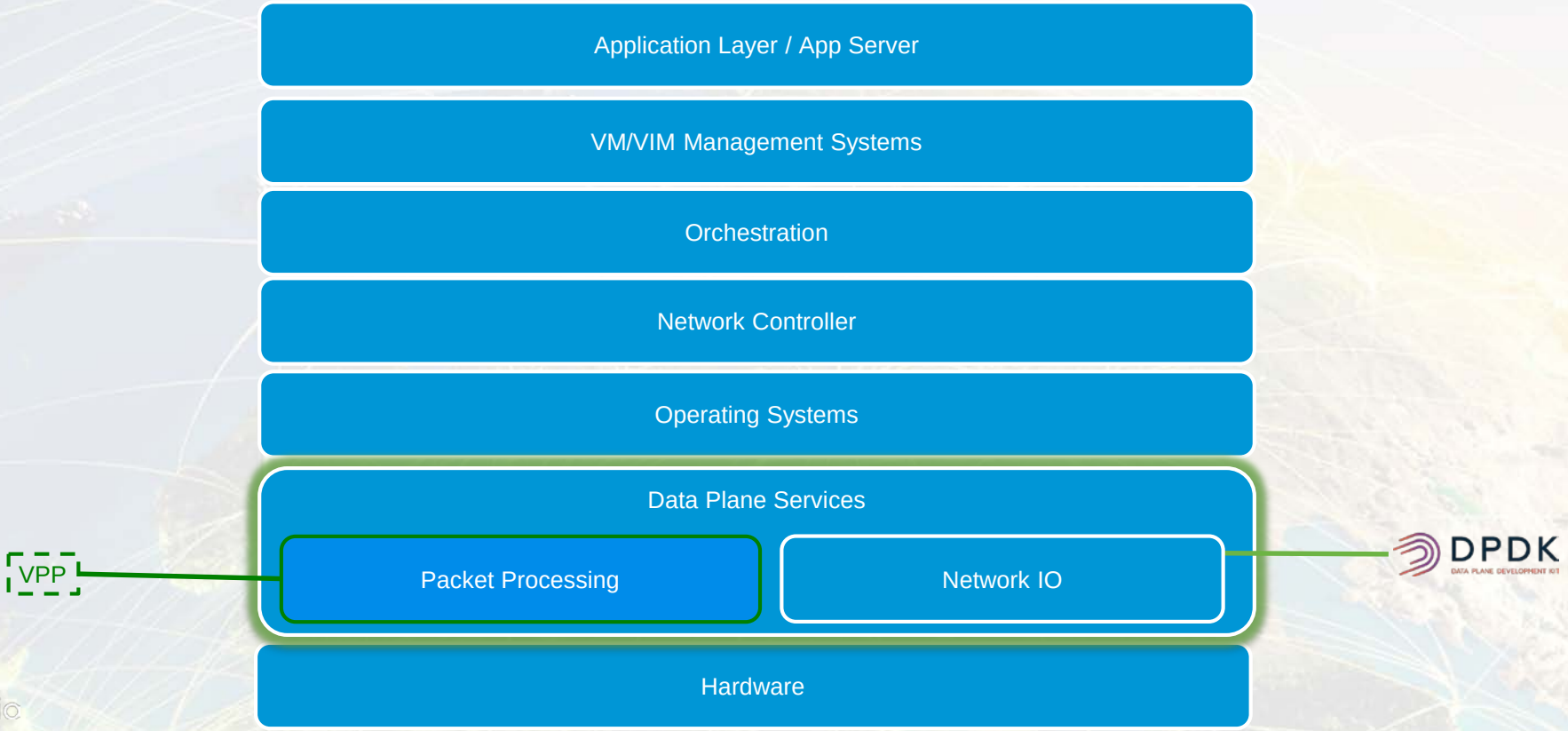
- **L2:** VLan, Q-in-Q, Bridge Domains, LLDP ...
- **L3:** IPv4, GRE, VXLAN, DHCP, IPSEC …
- **L3:** IPv6, Discovery, Segment Routing …
- **CP:** CLI, IKEv2 …

Integrated

- Language bindings
- Open Stack/ODL (Netconf/Yang)
- Kubernetes/Flanel (Python API)
- OSV Packaging



VPP in the Overall Stack

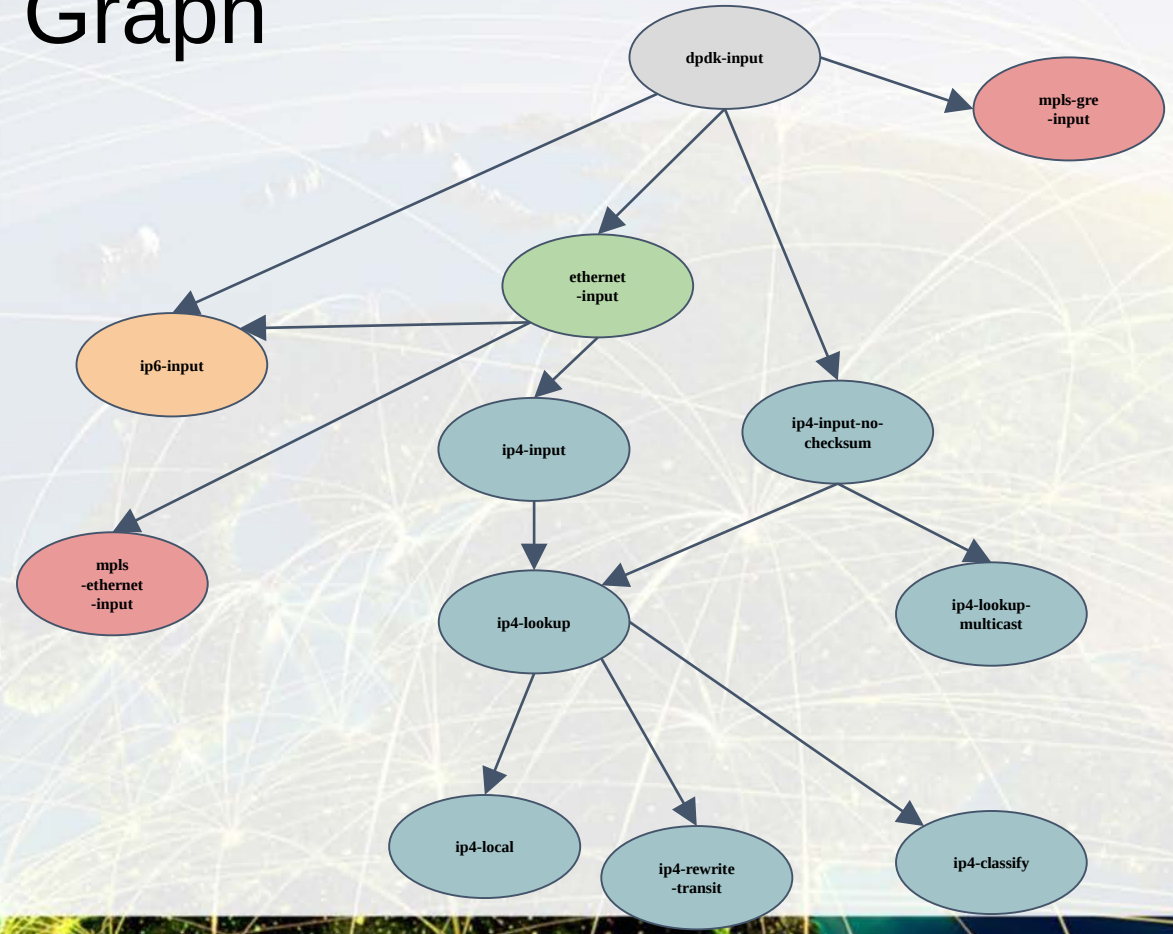


VPP: *Dipping into internals..*

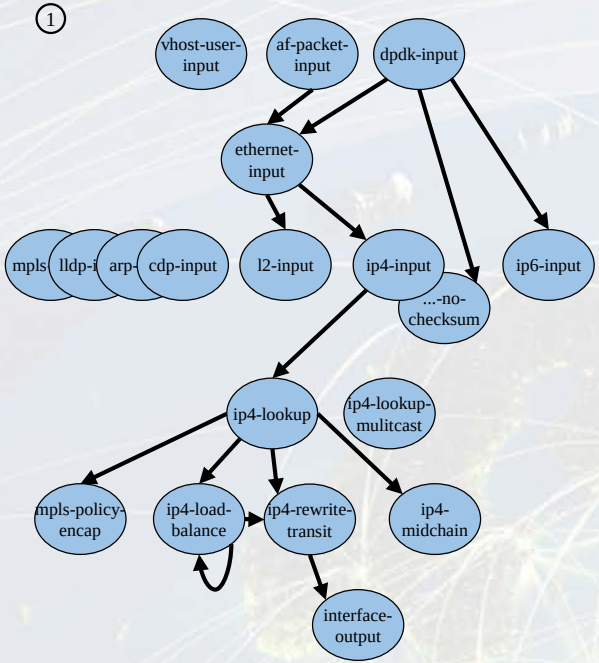
VPP Graph Scheduler

- Always process as many packets as possible
- As vector size increases, processing cost per packet decreases
- Amortize I-cache misses
- Native support for interrupt and polling modes
- Node types:
  - Internal
  - Process
  - Input

Sample Graph

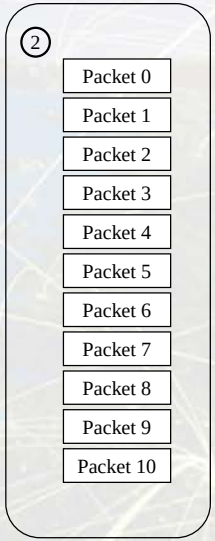


How does it work?



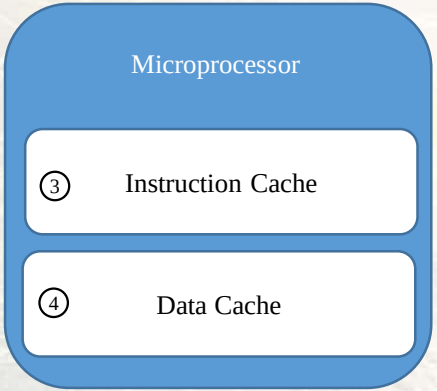
Packet processing is decomposed into a directed graph node …

* approx. 173 nodes in default deployment



… packets moved through graph nodes in vector …

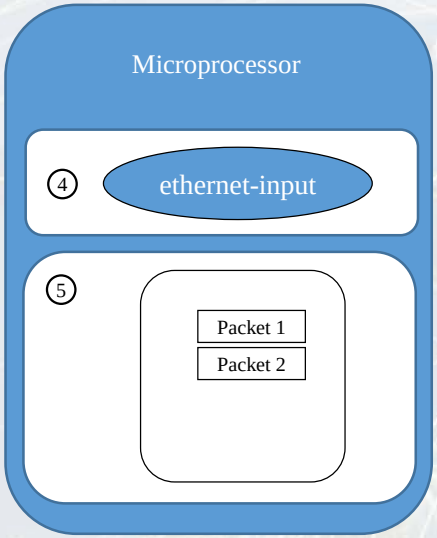
… graph nodes are optimized to fit inside the instruction cache …



… packets are pre-fetched, into the data cache …

How does it work?

… instruction cache is warm with the instructions from a single graph node …



… data cache is warm with a small number of packets ..

⑥

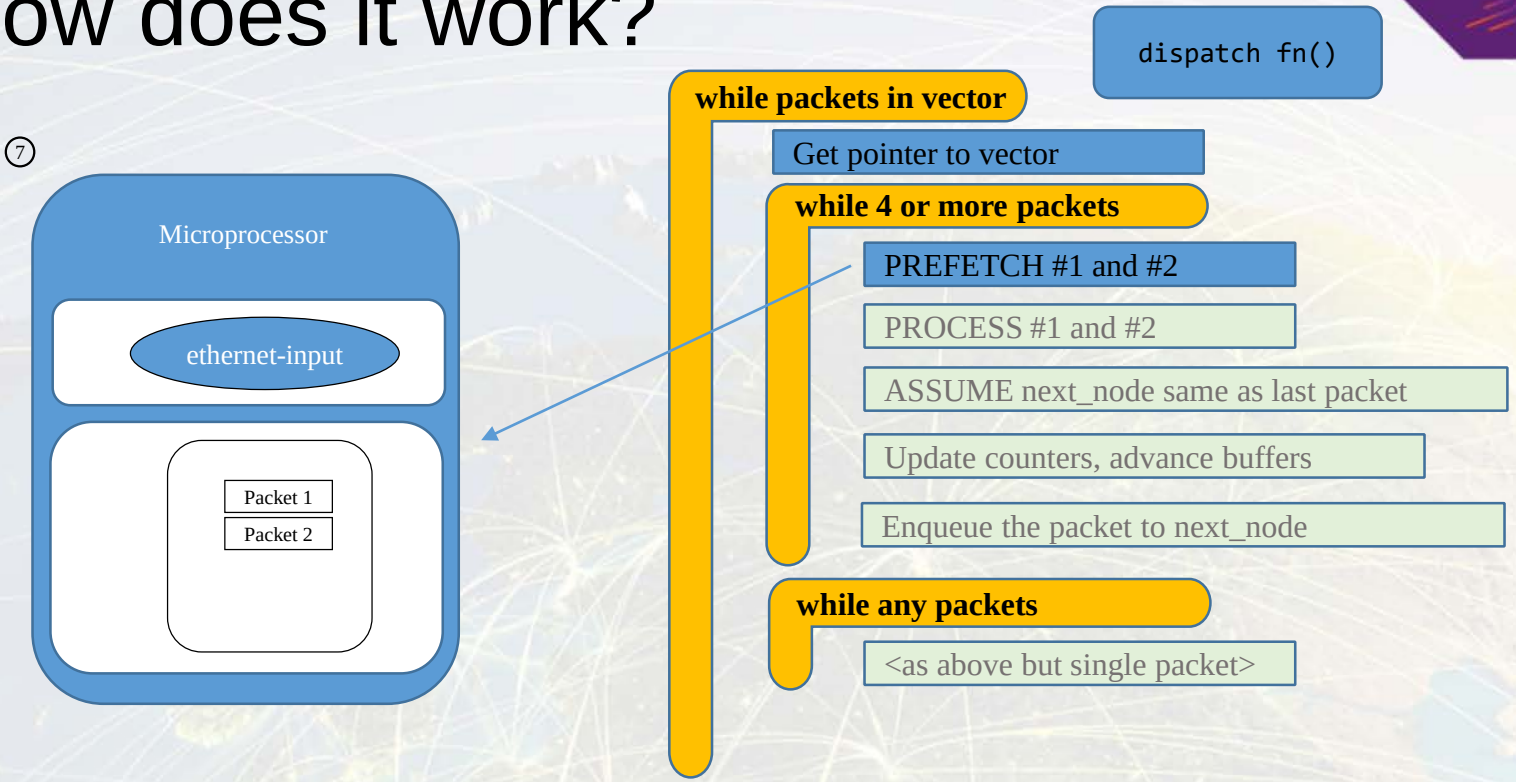
dispatch fn()

while packets in vector

- Get pointer to vector
- **while 4 or more packets**
  - PREFETCH #3 and #4
  - PROCESS #1 and #2
  - ASSUME next_node same as last packet
  - Update counters, advance buffers
  - Enqueue the packet to next_node
- **while any packets**
  - <as above but single packet>

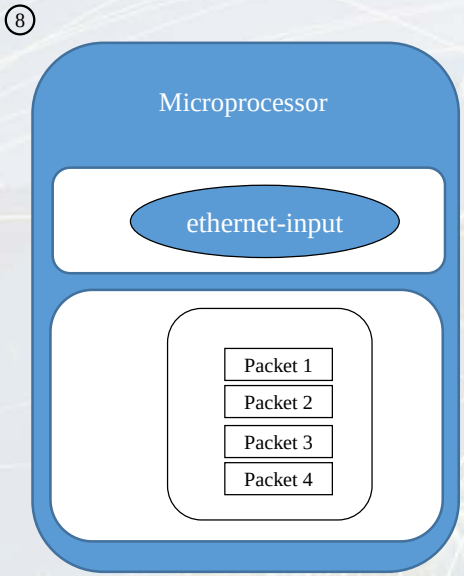
… packets are processed in groups of four,
any remaining packets are processed on by one …

How does it work?



… prefetch packets #1 and #2 …

How does it work?



dispatch fn()

while packets in vector

- Get pointer to vector
- **while 4 or more packets**
  - PREFETCH #3 and #4
  - PROCESS #1 and #2
  - ASSUME next_node same as last packet
  - Update counters, advance buffers
  - Enqueue the packet to next_node
- **while any packets**
  - <as above but single packet>

… process packet #3 and #4 …
… update counters, enqueue packets to the next node …

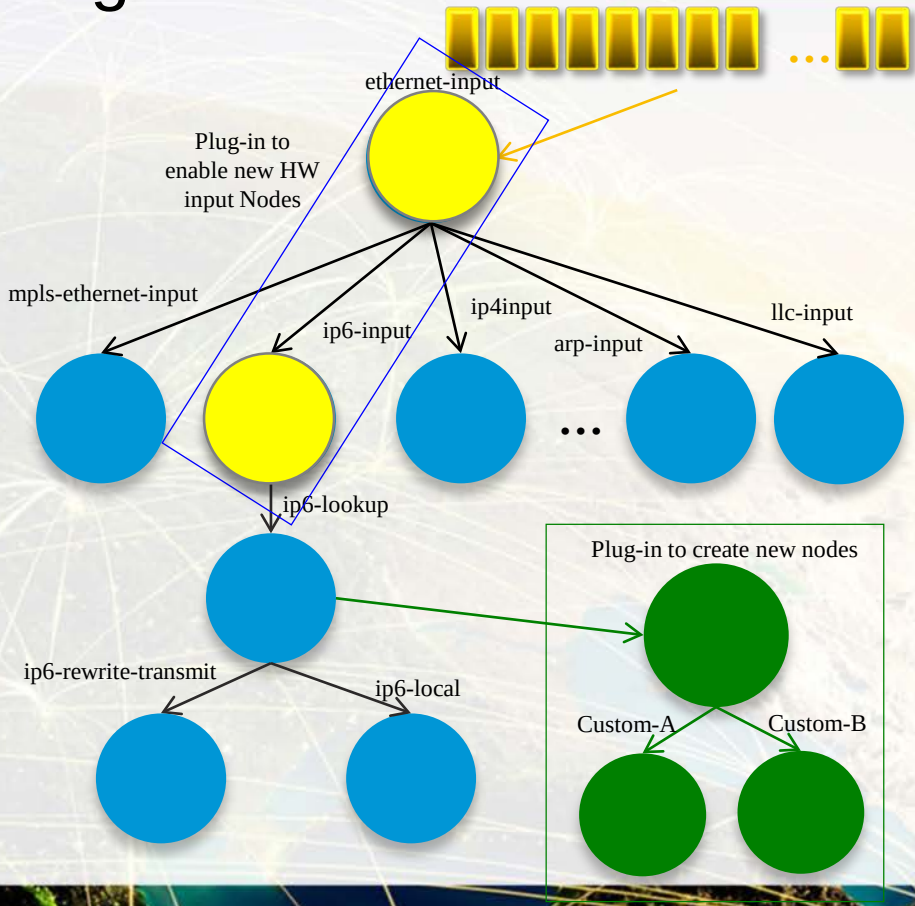
Modularity Enabling Flexible Plugins

Plugins can:

- Introduce new graph nodes
- Rearrange packet processing graph
- Can be built independently of VPP source tree
- Can be added at runtime (drop into plugin directory)
- All in user space

Enabling:

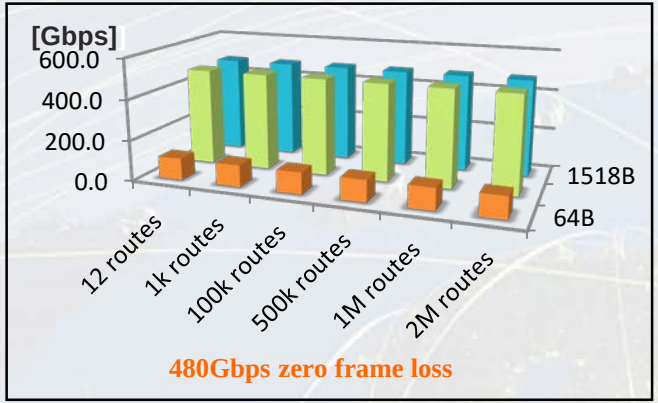
- Ability to take advantage of diverse hardware when present
- Support for multiple processor architectures (x86, ARM, PPC)
- Few dependencies on the OS (clib) allowing easier ports to other Oses/Env



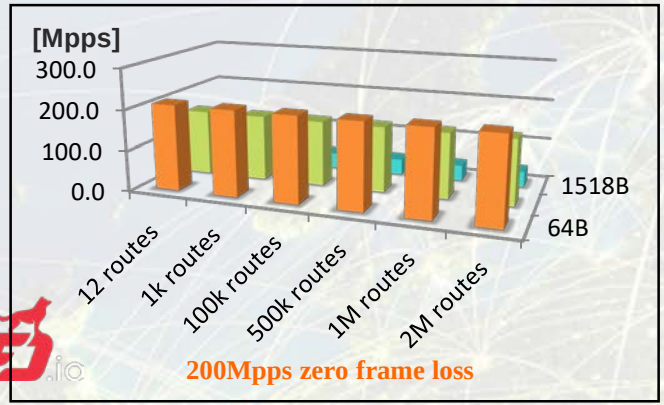
VPP: *performance*

VPP Performance at Scale

IPv6, 24 of 72 cores

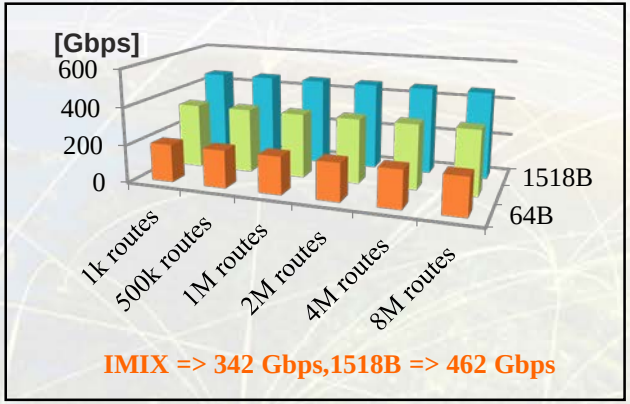


480Gbps zero frame loss

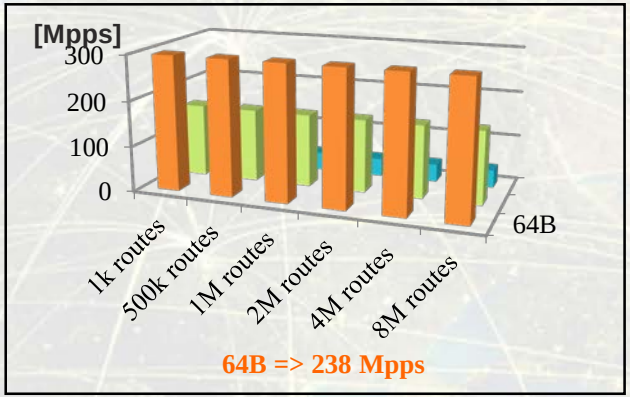


200Mpps zero frame loss

IPv4+ 2k Whitelist, 36 of 72 cores



IMIX => 342 Gbps,1518B => 462 Gbps



64B => 238 Mpps

Zero-packet-loss Throughput for 12 port 40GE

Hardware: Cisco UCS C460 M4
Intel® C610 series chipset
4 x Intel® Xeon® Processor E7-8890 v3 (18 cores, 2.5GHz, 45MB Cache)
2133 MHz, 512 GB Total
9 x 2p40GE Intel XL710 18 x 40GE = 720GE !!

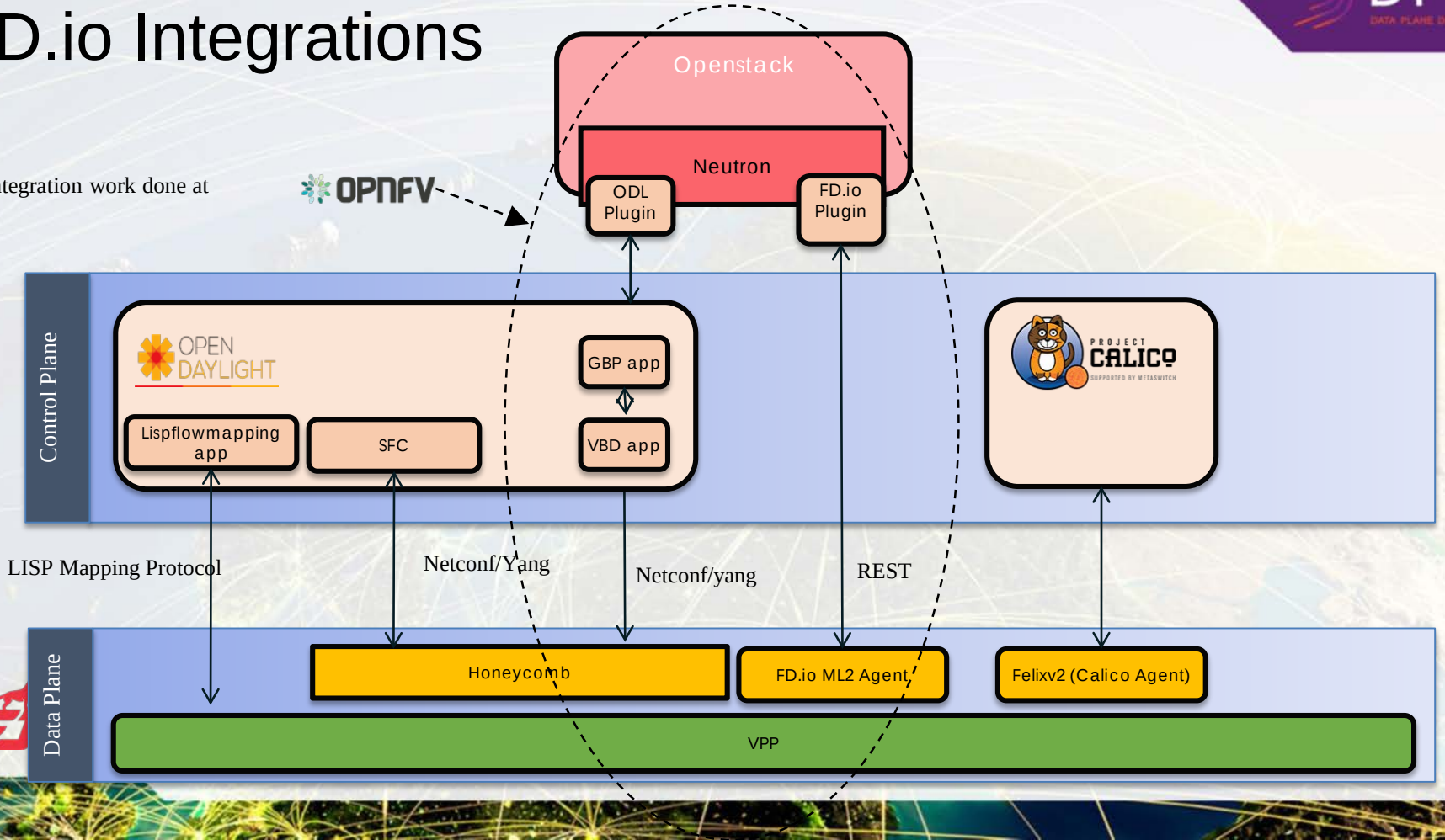
Latency
18 x 7.7trillion packets soak test
Average latency: <23 usec
Min Latency: 7…10 usec
Max Latency: 3.5 ms

Headroom
Average vector size ~24-27
Max vector size 255
Headroom for much more throughput/features
NIC/PCI bus is the limit not vpp

VPP: *integrations*

FD.io Integrations

Integration work done at OPNFV



Summary

- VPP is a fast, scalable and low latency network stack in user space.
- VPP is trace-able, debug-able and fully featured layer 2, 3 ,4 implementation.
- VPP is easy to integrate with your data-centre environment for both NFV and Cloud use cases.
- VPP is always growing, innovating and getting faster.
- VPP is a fast growing community of fellow travellers.

ML: **vpp-dev@lists.fd.io**

Wiki: **wiki.fd.io/view/VPP**

Join us in FD.io & VPP - *fellow travellers are always welcome.*
Please reuse and contribute!

Contributors…

Universitat Politècnica de Catalunya (UPC)

THANK YOU