FD.io - Vector Packet Processing

- One Terabit Software Router

on Intel® Xeon® Scalable Processor Family Server

Table of Contents

Introduction

What is VPP? Is it a software router? A virtual switch? A virtual network function? Or, something else? In fact it is all of the above and a whole lot more. It is a modularized and extensible software framework for building bespoke network data plane applications. And equally importantly, VPP code is written for modern CPU compute platforms (x86_64, ARMv8, PowerPC, to name a few), with a great deal of care and focus given to optimizing the software-hardware interface for real-time, network I/O operations and packet processing.

VPP takes advantage of CPU optimizations such as Vector Instructions (e.g. Intel® SSE, AVX) and direct interactions between I/O and CPU cache (e.g. Intel® DDIO), to deliver best in class packet processing performance. The result is a minimal number of CPU core instructions and clock cycles spent per packet - enabling Terabit performance using the latest Intel® Xeon® Scalable Processor (processor details found at the end of this paper).

The Most Efficient on the Planet! - VPP is the most efficient software packet processing engine on the planet. Imagine running it in your private cloud, or in many public clouds all around the planet. Flipping fast networking software running in the cloud and ruling the Internet services. Imagine that… Well, you actually can do a bit more than just imagine, you can actually use it, extend it, and integrate it into your Internet service life. You can build a physical or virtual switch, router, a load balancer, a NETFLOW probe, an IPv4 as a service VNF... Pretty much anything that manipulates packets or implements protocols. The world is your oyster.

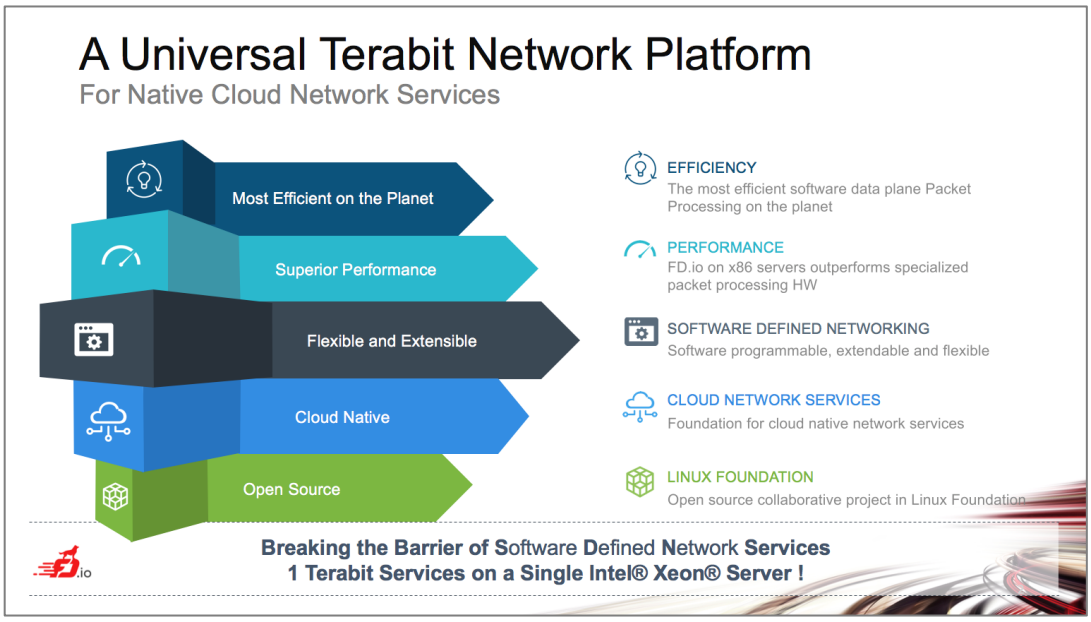

Figure 1. A Universal Terabit Software Network Platform.

Culture shift

Traditionally routers were built with a tightly coupled data plane and control plane. Back in the 80s and 90s the data plane was running in software on commodity CPUs with proprietary software. As the needs and desires for more speeds and feeds grew, the data plane had to be implemented in ASICs and FPGAs with custom memories and TCAMs. While these were still programmable in a sense, they certainly weren't programmable by anyone but a small handful of people who developed the hardware platform. The data plane was often layered, where features not handled by the hardware data plane were punted to a software only data path running on a more general CPU. The performance difference between the two were typically an order or two of magnitude. Software being slower, of course!

In the 2000s there was a shift towards a larger split between control plane and the data plane, where the data plane would have a fixed API, like Openflow. That still didn't make the data plane very flexible, but at least the data plane was more or less decoupled from the control plane.

The latest developments are one of extreme modularity. Hardware is becoming much more programmable and we see the arrival of P4, a domain specific language for the purpose of programming the network data plane. We are also seeing complete decoupling of the control plane, both in function and in locality.

This decoupling requires a shift in thinking of how a network and network functions are managed. Take the example of the "IPv4 as a service" VNF, a simple function that takes an IPv4 packet from the Internet, does a table lookup and encapsulates the packet in an IPv6 header and sends it out to an end-user. Does it make sense that this simple network service should have a full CLI, support SNMP, and every other control plane function we are familiar with for traditional routers? Probably not. The model of managing it should probably be closer to how you manage and operate an application daemon, e.g. BIND or Apache httpd.

It has *always* been possible to do pure software forwarding and to build a software router. Some of us remember from back in the late 80s, that this was what the universities did to serve their main Internet connection of a whopping 64 kbps line. What has changed over the last few years is that you can now get real performance from software routing running on commodity hardware.

While you cannot expect a router with tens or hundreds of 100GbE ports and multi-terabit performance, we anticipate FD.io VPP running on an industry standard 2RU server, breaking the 1Tbps throughput boundary for the networking data plane. This is made possible by the new Intel® Xeon® Processor Scalable family, bringing architectural improvements and PCIe bandwidth increase while decreasing overall cycles per packet. VPP is here to take advantage of this increased network I/O as it has been starving for more I/O fan-out with current commodity servers for a while, no VPP code change required. Read on to find out more...

Architecture

30,000 feet view

VPP is a data plane, a very efficient and flexible one. It consists of a set of forwarding nodes arranged in a directed graph and a supporting framework. The framework has all the basic data structures, timers, drivers (and interfaces to driver kits like DPDK), a scheduler which allocates the CPU time between the graph nodes, performance and debugging tools, like counters and built-in packet trace. The latter allows you to capture the paths taken by the packets within the graph with high timestamp granularity, giving full insight into the processing on a per-packet level.

VPP has a plugin architecture. Plugins are first class citizens, and are treated just like modules embedded directly in VPP. Plugins are typically forwarding nodes serving a particular function, but can also be a driver, or another CLI or API binding. Plugins can insert themselves at various points in the graph, depending on their function and enable fast and flexible development of new and bespoke network or service functions leveraging existing VPP nodes and graph as a stable packet processing platform. One can focus on the new functionality and leverage for free what already works, and works very well.

The input node polls (or is interrupt driven) an interface's RX queue for a burst of packets. It assembles those packets into a vector or a frame per next node, e.g. it sorts all IPv4 packets and passes those to the ip4-input node, the Ipv6 packets into the ip6-input node and so on. When the ip6-input node is scheduled, it takes its frame of packets and processes them in a tight dual loop (or quad-loop) with prefetching to the CPU cache to achieve optimal performance. This makes more efficient use of the CPU cache by reducing misses, and scales efficiently for larger CPU caches. The ip6-input node pushes the various packets onto another set of next-nodes, e.g. error-drop if validation checks failed, or most typically ip6-lookup. The frame of packets move like a train through the system until they hit the interface-output node and are shipped onto the wire.

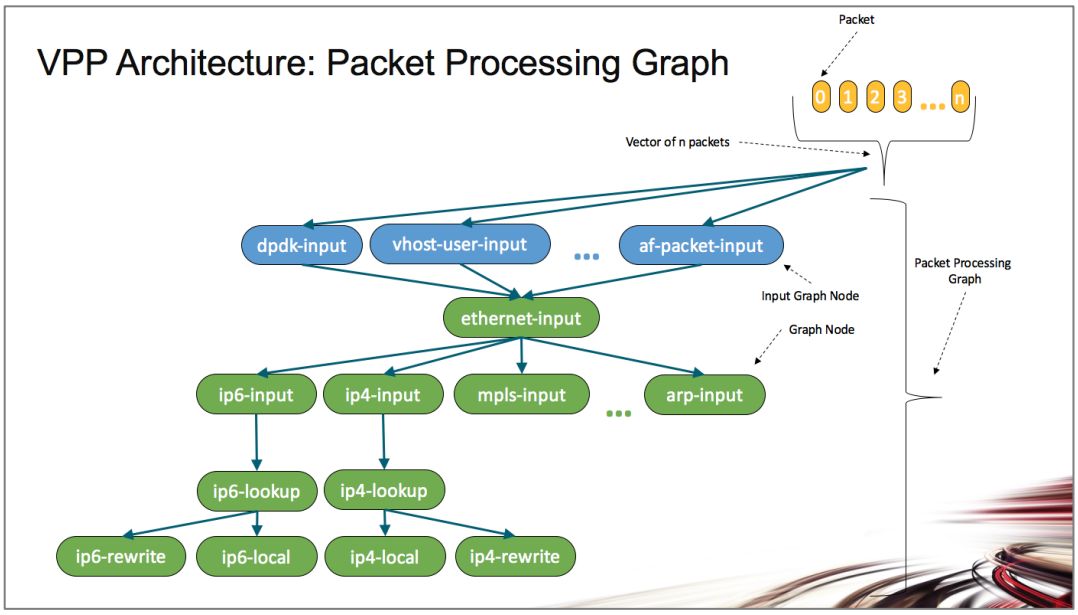

Figure 2. VPP Architecture: Packet Processing Graph of Nodes.

Processing a vector at a time per network function has several benefits.

- From a software engineering perspective, each node is independent and autonomous.

- From a performance perspective, the primary benefit is derived from optimizing the use of the CPU's instruction cache (i-cache). The first packet heats up the cache, and the rest of the packets in the frame (or vector) are processed "for free". Here, VPP takes full advantage of CPU’s super scalar architecture, enabling packet memory loads and packet processing to be interleaved for a more efficient processing pipeline. Compare that with a scalar packet processing approach, where one packet would be processed from input to output. The instruction cache would overflow, and you would get a significant amount of i-cache misses per packet.

- Similarly, VPP is optimized to take advantage of CPU’s speculative execution. Secondary performance benefits are gained from speculatively re-using the forwarding objects (like adjacencies and IP lookup tables) between packets, as well as pre-emptively loading data into the CPUs local data cache (d-cache) for future. This efficient use of the compute-hardware allows VPP to exploit fine-grained parallelism.

The graph processing nature of VPP represents a loosely coupled, highly cohesive software architecture. Each graph node uses a “frame” as the atomic unit of input and output. This provides the loose-coupling. The architecture is cohesive as common functions are grouped into each graph node based on desired functionality.

Nodes in the graph are fungible and when combined with VPP’s ability to support the loading of “plugin” graph nodes at runtime, means new and interesting features can be prototyped and rapidly developed without forking and compiling a bespoke version of the source.

Developer's delight

VPP is written completely in C. It runs in user space and is very self-contained. If you have tried to develop networked applications using BSD sockets and interacting with the Linux kernel, you will find writing network functions in VPP a blast. There is no awkward API to go through to access various headers and parts of a packet. "Here's a pointer to the start of the IP header, go and have fun", that's our attitude.

VPP provides a plethora of data structures to help you; Hashes, Pools, Dynamic arrays, LPM, Timer wheels. There is a lot of templates and patterns to base new plugins or forwarding nodes on. The data structures generally deals with indexes, so the developer is spared against maintaining pointers.

Fundamental to VPP's software design is the notion of composability. Take a relatively simple idea of a vector. Optimise it, harden it, provide useful functions for memory management and usability then build on it. Vectors with Bitmaps become Pool structures. This collection of utility infrastructure components (in *src/vppinfra*) combined with the VPP application harness for node utilities, threading support etc (*src/vlib*) form the foundation of the VPP networking data plane application (*src/vnet*).

Potentially a developer could leverage (*src/vppinfra*) and (*src/vlib*) for completely new and interesting use-cases outside of networking, which is the point… VPP rests on an architectural foundation of software composability.

It is a run to completion system, and it generally avoids locking or any complex mechanisms for thread synchronisation.

VPP can run and be tested on a laptop in isolation without any requirement for external interfaces.

Control plane and data plane separation

Drawing the line between what goes in the control plane and what belongs to the data plane isn't always easy. Do protocols used for address resolution belong in the data plane for example? In VPP they do, but that's an engineering trade-off.

Architecturally we want the VPP data plane to be as generic as possible and be purely driven by an API. A lot of the complexity in control planes come from the amount of "glue" required to couple the various components together. Imagine a simple function like DHCPv6 PD (RFC3633). A DHCPv6 client must initiate PD on an interface, whenever the DHCP process acquires a set of IPv6 prefixes, these have to be subnetted and addresses assigned to downstream interfaces, routing tables have to updated, and perhaps other component notified (e.g. a routing protocol). All this complexity and tight coupling between components is something we want to avoid in the data plane. These functions can still be tightly integrated into

the system, e.g. through a plugin, where the "glue" code can be written in a more high level language like Lua or Python, and the various components are programmed through their respective APIs.

The VPP control plane interface

VPP offers a high performance binary API using shared memory, a CLI and support for rudimentary configuration files. The binary API is written in C and supports language bindings for Python, Java, Lua and C, with development work underway on GO and C++.

The C API can do upwards of 500Kmsgs/s while Python is an order of magnitude slower.

Integration with the host stack

When a host application wishes to **communicate**, it typically uses POSIX sockets to do so. The fact that this is ultimately implemented in a TCP/UDP/IP/Ethernet stack is a **networking** implementation choice. VPP recently introduced a shared memory FIFO mechanism that allows applications on the same VPP instance to **communicate** directly without the **networking** overhead. When communication is required from the FIFO to outside of VPP, a TCP/UDP graph node implements a high-performance, user-space, host networking stack.

Network researchers and innovators, due to VPP's architecture, are free to experiment with alternate inter-host communication mechanisms such as RDMA and RINA to interoperate with the shared-memory communication library. In fact, the congestion-control mechanism of VPP's TCP host-stack is designed to be modular, such that alternate implementations such as BBR can easily be integrated and tested.

In short, VPP is as much a platform for communication/networking innovation, as it is a highly performant production ready data plane.

VPP Use cases

Virtual router or virtual switch

VPP can be used as a fully featured high performance multi-layer virtual router and virtual switch.

VPP can be deployed as an IPv4/IPv6 software router, delivering consistent throughput performance with large-scale lookup tables. VPP supports concurrent L2 switching, IP routing and a rich set of modern overlay tunneling encapsulations - GRE, VXLAN, L2TPv3, LISP-GPE. Combined with software and hardware based crypto support (IPSec, SSL), VPP's cocktail of functionality, performance and scalability enables building large scale dynamic network overlay

designs for VPNs between sites and/or private/public Data Centers. And it can be run from the cloud - what more would you want?

VPP is also crammed with many stateless and stateful security features - access lists, security groups and stateful ACLs. To facilitate dynamic Service Function Chaining, VPP supports IETF compliant SFC/NSH, as well as Segment Routing for IPv6. All enabling efficient and scalable creation of bespoke cloud network services. To check VPP applicability and suitability to your current or future use cases, see the VPP functional breakdown listed later in this document.

A sample end-to-end Software Defined Cloud Network Service use case design, a cloud VPN, leveraging FD.io VPP software is depicted below. It places VPP vRouter network functions in Data Center servers, making them part of an encrypted overlay network (e.g. LISP-GPE, VXLAN) and connects remote sites and any cloud based network and application services.

Orchestrating VPP is simple - VPP instances can run in Containers or VMs, so you can deploy the orchestration stack of your choice, as VPP is already integrated with a number of the stacks including OpenStack (networking-vpp), Kubernetes, and Docker.

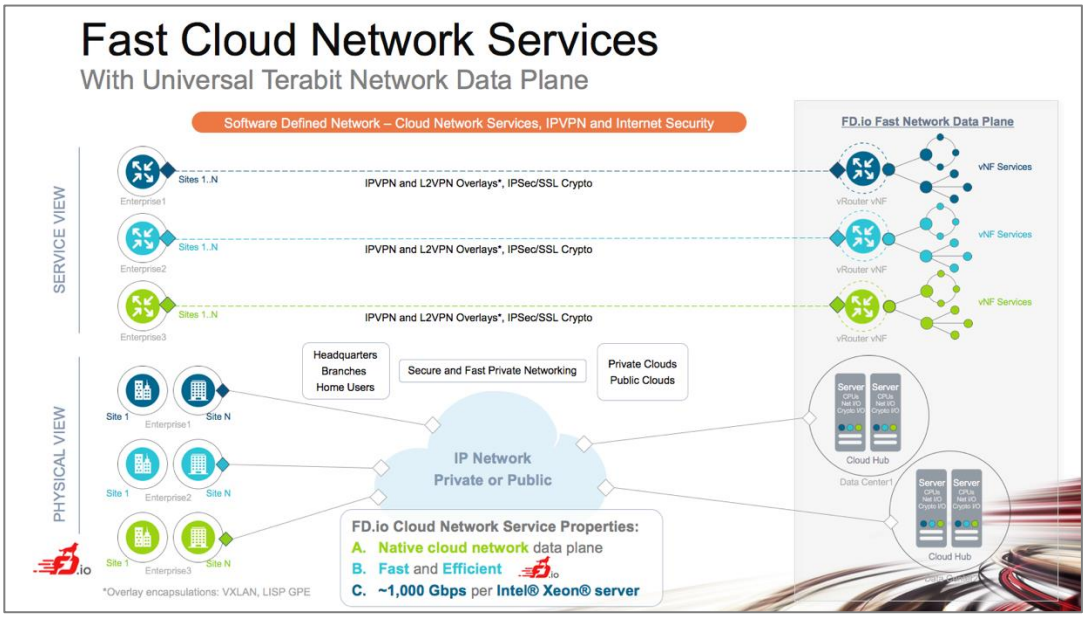

Figure 3. Fast Cloud Network Services – FD.io VPP Use Case Examples.

Replacing a hardware router or switch

VPP is a fully functional router, albeit with the expectation of an external control plane. It is possible to run VPP as a CPE or branch router or whatever location where a 1Tbps router is sufficient (quite a few, one would imagine). With configuration files or a small script, or with a

more fully fledged control plane, all the functions required to act as an IPv4 CPE (DHCP, NAT, Security groups) are already implemented. The main restriction is the network I/O of servers and not VPP. Commodity servers typically do not have a switching fabric, and you would be limited to the number of available network I/O and PCIe slots for networking devices.

Network functions

VPP supports a set of Virtual Network Functions (VNFs), be that one of the multitude of "IPv4 as a service" mechanisms (MAP-E, MAP-T, LW46, SIIT-DC), or the reverse, IPv6 as a service (6RD), Load-balancing, IPFIX/Netflow probe, NAT44 / CGN...

VPP can be used in a VNF targeting Virtual Machine (VM), container, or bare metal deployments. When used in a VM, the VPP-based VNF would typically connect over a vHost-user interface, while in a container the new high speed shared memory interface, MemIf, can provide the connectivity.

Multiple network functions can be implemented as nodes in a single VPP instance, or one can have a VPP container per function and chain them together with SFC / NSH, which are also supported.

Of course the lowest overhead and highest performance is achieved by running VPP directly on bare metal. Taking one of the IPv4 as a service mechanisms as an example, these implementations can easily forward in the excess of 15-20 Gbps (IMIX), 12-17 Mpps (64B) per core on Intel® Xeon® processors (core frequency dependent of course) and can fill all available bandwidth in a system (hundreds of Gbps).

New VNFs can be easily built as plugins. These can be incorporated in a more traditional router model, where packets pass through a FIB lookup, or one could make them a very simple "VNF on a stick", where frames are received on an interface, the frames are passed through some packet processing function and then dumped on the TX interface.

VPP's plugin architecture lets the operator add new network functions to a VPP instance running on bare metal. For network centric infrastructure functions (e.g. network functions generating hundreds of gigabits of traffic per second) it makes sense to avoid the additional overhead and complexity of operating and isolating network functions in VMs or containers.

Integrations

FastDataStacks

Any NFV solution stack is only as good as its foundation: the networking infrastructure. Key foundational assets for a NFV infrastructure are:

- *The virtual forwarder*: The virtual forwarder needs to be a feature-rich, high performance, highly scale virtual switch-router. It needs to leverage hardware accelerators when available and run in user space. In addition, it should be modular and easily extensible.
- *Forwarder diversity*: A solution stack should support a variety of forwarders, hardware forwarders (physical switches and routers) as well as software forwarders. This way virtual and physical forwarding domains can be seamlessly glued together.
- *Policy driven connectivity*: Business policies should determine the network level connectivity, rather than the other way around. Historically this has often been the other way around which quite often resulted in operational challenges.

In order to meet these desired qualities of an NFV infrastructure, the OPNFV FastDataStacks project was started in spring 2016, shortly after the FD.io Linux Foundation collaborative project was launched. FastDataStacks set out to compose a variety of scenarios using FD.io as a foundation to create an NFV solution that is both *fast and flexible*. OPNFV runs NFV for real – which also means that a virtual forwarder has to supply multi-million packets per second forwarding capability – even, and especially, when integrated into a full stack solution. Simple software switches which are often found in cloud deployments with forwarding rates in the tens of thousands of packets per second don't offer appropriate performance for NFV deployments.

FastDataStacks scenarios are created with components from a set of open source projects. While performing the integration, FastDataStacks had to integrate and significantly evolve the functionality of different upstream components used, and evolve the automated installation and testing tools in OPNFV. FastDataStacks is a great example of OPNFV's modus operandi: create, compose, deploy, test, iterate.

The key ingredient for all these scenarios is the data-plane forwarding and control infrastructure supplied by FD.io, i.e. VPP and Honeycomb along with OpenStack as the VM manager. In addition, OpenDaylight as a network controller plays a key role in many of the scenarios built by FastDataStacks.

The picture showcases typical key components in FastDataStacks:

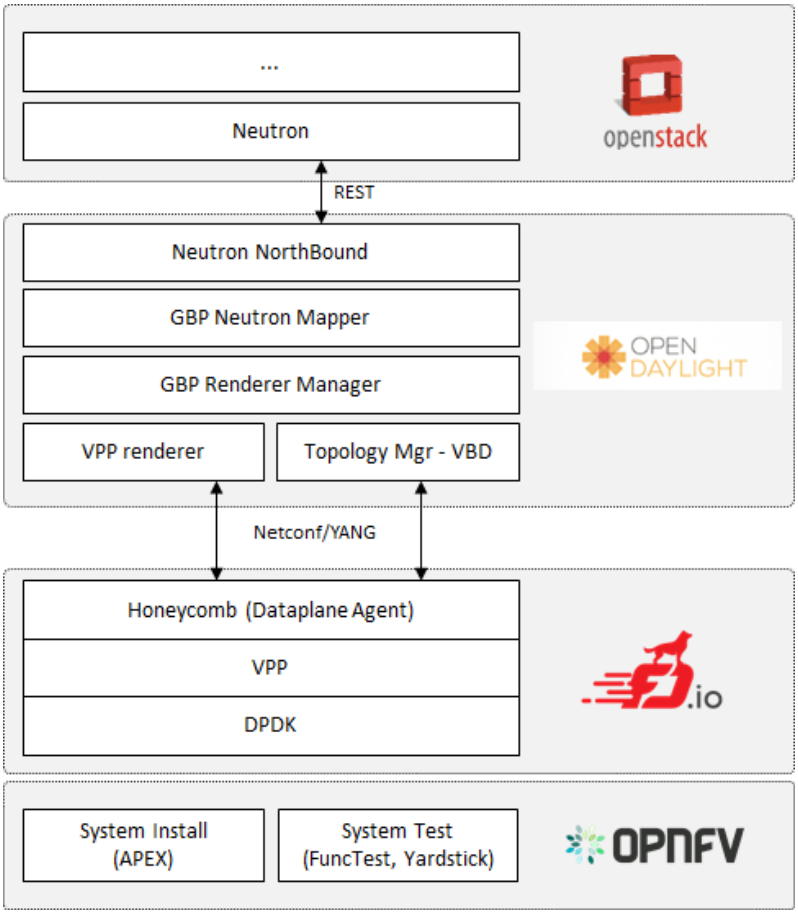

Figure 4. OPNFV FastDataStacks Components.

OPNFV Scenarios Built by FastDataStacks

FastDataStacks builds a series of scenarios with a variety of features and functions, following the different needs of the market. These include for example:

- *OpenStack – VPP*
- *OpenStack – OpenDaylight (Layer2) – Honeycomb – VPP*
- *OpenStack – OpenDaylight (Layer3) – Honeycomb – VPP*

FastDataStacks scenarios are released once complete and fully tested. The first FastDataStacks scenario, termed “OpenStack – OpenDaylight (Layer2) – Honeycomb – VPP”, became available as part of the OPNFV Colorado 1.0 release in September 2016. With the OPNFV Danube release in 2017, it is for the first time that FastDataStacks introduces a scenario which leverages FD.io/VPP for all forwarding – within a tenant network, between tenant networks, as well as between tenants and external networks. VPP is used for Layer-2 and Layer-3 networking, including the support for security groups, etc. making forwarding by legacy solutions such as the Linux Kernel or OVS superfluous.

All FastDataStacks scenarios continue to be enhanced as new capabilities mature in the upstream projects. Features like service function chaining (SFC), or IPv6, Gluon, etc. will naturally find their way into FastDataStacks. See also: FastDataStacks OPNFV project wiki and OPNFV FastDataStacks - Whitepaper

OpenStack: networking-vpp

In addition to the model for driving VPP via the OpenStack Neutron Modular Layer 2 (ML2) Driver for OpenDaylight, VPP supports a native integration with OpenStack and delivers required functions to operate a cloud environments as well as an NFV Infrastructure. It enables fast communications between VMs, fast forwarding within a tenant network and fast routing between tenants or external networks. This integration is done through a Neutron driver called *networking-vpp*.

Networking-vpp supports all key features required to run a production cloud:

- flat/vlan networks
- scalable overlay using LISP-GPE and layer 2 population
- Security Groups, Port Security, Address pairs, RBAC
- fast inter VM communication with vhost-user interfaces
- Layer 3 agent: Floating IP, SNAT, IPv6

While many Neutron drivers suffer from complex code and architecture making them complex to troubleshoot and operate, *networking-vpp* relies on few design principles:

- Scalability: all states are maintained in a scalable key-value store cluster (*etcd*)
- Simplicity: All communications are REST based, code is very compact
- Availability: All communication are asynchronous, system tolerates machine failure

The picture below illustrates the resulting architecture:

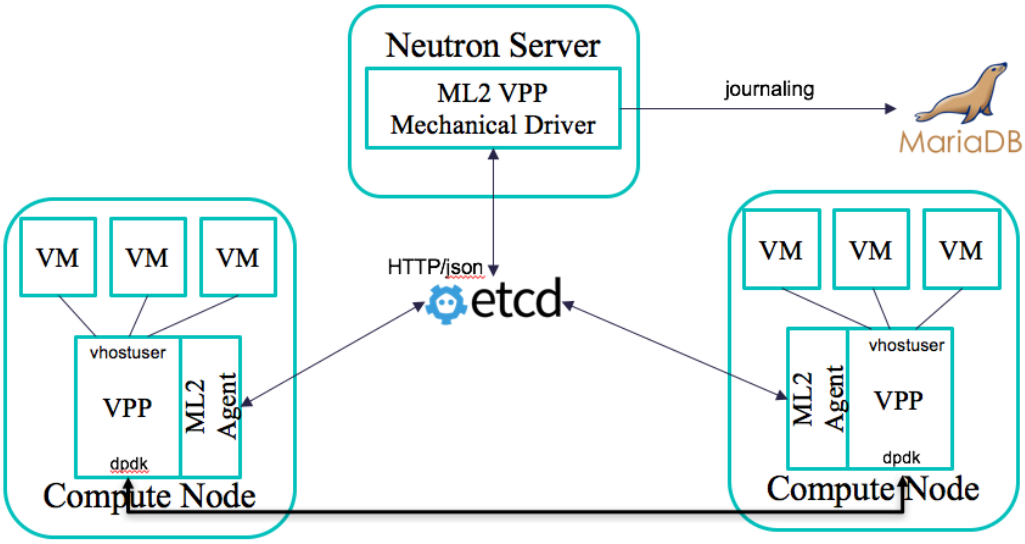

Figure 5. OpenStack Networking-vpp Architecture.

Main components in this architecture are:

- The mechanism driver which is in charge of implementing OpenStack Neutron ML2 API. All Neutron commands are put in a journaling system in order to support potential machine or communication failure. Any Create/Update/Delete (CRUD) operation results in a communication with the key value store
- Etcd stores agent states as well as OpenStack “desired” states.
- VPP Agents communicate with VPP using native Python API. Agents are running in a separate memory space from VPP.

All communications between these three components are HTTP/JSON based. Etcd can enforce Role Based Access Control to isolate key-value operations per compute node and increase system security and resiliency.

Networking-vpp takes advantage of the OpenStack CI for testing. In addition to that, more advanced testing scenario (including High Availability scenario) are executed in the frame of the OPNFV FastDataStack project.
It supports OpenStack distributions installer such as DevStack or Redhat TripleO / APEX.

Community update

After the project was launched just over a year ago it has been very active. The FD.io project has the following supporting members: Cisco, Intel, Ericsson, 6WIND, Huawei, AT&T, Comcast, Cavium Networks, Red Hat, ZTE, Inocybe, Metaswitch and Netgate.

Apart from the main VPP project, there are several adjacent projects under the FD.io umbrella: Honeycomb (ODL integration), CSIT (Integration testing and performance), NSH SFC, ONE (Overlay Network Engine), VPP Sandbox, TLDK, Package management, TRex, ICN (Information centric networking).

The VPP project has had over 2000 commits from over 50 contributors over the last year and has done 5 releases. With a time-based release model, releasing about every 3 months.

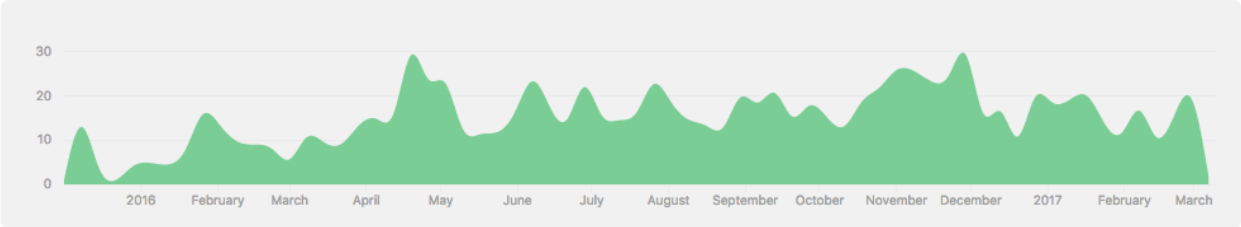

Figure 6. FD.io VPP Commit Distribution over Time.

Current functionality

Table 1. FD.io VPP Functionality.

Features	Details
Platforms	x86, ARM, Power
Interfaces	DPDK, Netmap, AF_PACKET, Memif, TAP, vhost-user
Linux distributions	Ubuntu 14.04/16.04, CentOS 7, OpenSUSE
API	Bindings for C, Python, Lua, Java
FIB	Hierarchical FIB, VRFs,
IPv4	ARP, ARP proxy, VRFs, ECMP, Source RPF
IPv6	ND, ND proxy, VRFs, ECMP, Source RPF
DHCP	DHCPv4 client/server/relay, DHCPv6 relay, Option 82 / Remote-id support
BFD	IPv4, IPv6, BFD echo
NSH	SFC SFF's & NSH Proxy
Security groups	L2, L3, IPv6 extension headers, Stateful ACLs
Tunnelling	GRE/VXLAN/VXLAN-GPE/LISP-GPE/NSH/IPSEC
MPLS	MPLS VPNs, Tunnels
Segment Routing	SRv6 Traffic Engineering SRv6 LocalSIDs functions to support L3VPN and L2VPN use-cases Framework to expand SRv6 LocalSID functions with VPP plugins
LISP	xTR, RTR, multi-homing, multi-tenancy, L2 and NSH over LISP-GPE.
Host stack	UDP, TCP
NAT	NAPT, 1:1 mode, Multi-tenant, Deterministic CGN, Hairpinning, CPE mode, IPFIX
Bridging	VLAN Support, Single/Double tag, L2 forwarding w/EFP Bridging Split-horizon group support/EFP Filtering Bridge domains, ARP termination, IRB - BVI Support Flooding, Input ACLs, MAC Learning Interface cross-connect L2 GRE over IPSec tunnels VTR – push/pop/Translate (1:1,1:2, 2:1, 2:2)

	Netflow collection
Monitoring	Netflow/IPFIX, SPAN, Counters, Ganglia, Lawful Intercept
Netflow/IPFIX	Recording of L2, L3 and L4 information from L2 or L3 paths
IPv4 as a service	MAP-E, MAP-T, LW46, SIIT-DC BR
QoS	Policing 1R2C, 2R3C, color-blind, color-aware.
IOAM (in-band OAM)	In-band OAM for IPv6, NSH, VXLAN-GPE transport IOAM Telemetry export infra (raw IPFIX) SRv6 and IOAM co-existence IOAM proxy mode / caching (M-Anycast server solution) IOAM probe and responder (for fast failure detection/isolation)
DPDK	17.02
IPsec	IKEv2, AES-GCM, CBC-SHA1

Performance

VPP superior data plane performance is one of its main claims to fame. But claims only get you thus far, so let's look at some data points. A summary table lists key performance and resource efficiency metrics for selected data plane functionality:

- Compute resources used: CPU core frequency, threads per core.
- VPP non-drop rate throughput: 64B packets per second, IMIX gigabits per second
- Efficiency indicators: clock cycles per packet, throughput speedup with multi-core multi-threading configurations.

In a nutshell - take 3.2 GHz Intel® Xeon® E5-2667 v4 (Broadwell) and you can drive Ethernet line at over 50 Gbps (IMIX) - with JUST ONE CPU CORE! You have an older computer with a tad slower 2.3 GHz Intel® Xeon® E5-2698 v3 (Haswell), you still can drive IPv4 routing at 36 Gbps per core - not too shabby!

Table below contains a sample listing of FD.io VPP data plane performance for different network functions, measured on different compute machines and reported by different labs involved in FD.io collaborative project.

Performance is listed for the following VPP data plane network functions:

- **IPv4 Routing:** IPv4 routing with 1M /32 routes, IPv4 header validations, IPv4 lookup per packet, L2 Ethernet header rewrite per packet.

- **L2 Xconnect:** L2 point-to-point switching with all Ethernet frame validations, no VLAN tag manipulations.
- **L2 MAC Switching:** L2 multi-point switching with 100k MAC addresses, MAC learning and flooding.
- **IPv6 Routing:** IPv6 routing with 0.5M /128 routes, IPv6 header validations, IPv6 lookup per packet, L2 Ethernet header rewrite per packet.
- **IPv4 IPSec AES-GCM:** IPv6 routing with IPSec AES-GCM cipher, using Intel QAT IPSec hardware acceleration card.
- **CGNAT:** carrier grade network address translation for IPv4 subscribers, 1k users, 15 ports each.

Table 2. FD.io VPP Data Plane Forwarding Performance – Benchmarking Data.

VPP Data Plane Network Function[1]	Core Freq [GHz][2]	Threads per Core [N=1..2][3]	Cycles per Packet[4]	Pkt Thput per Core 64B [Mpps][5]	BW per Core IMIX [Gbps][6]	Multi-Core Thput Speedup[7]	Test Results Source[8]
L2 Xconnect	2.2	2	115	**19.1**	**57**	Linear	Intel labs
L2 Xconnect	3.2	2	118	**27.1**	**81**	Linear	Intel labs
IPv4 Routing	2.2	2	180	**12.2**	**36**	Linear	Intel labs
IPv4 Routing	3.2	2	180	**17.8**	**53**	Linear	Cisco labs, Intel labs
IPv4 Routing	2.3	2	188	**12.0**	**36**	Linear	Cisco labs

[1] All tests performed with VPP 17.04 release, source code: https://git.fd.io/vpp/tree/?h=stable/1704.
[2] Compute machines used with the following CPUs and associated core frequency (TurboBoost disabled): Intel® XEON® E5-2699v4 **2.2GHz**, Intel® XEON® E5-2667v4 **3.2 GHz**, Intel® XEON® E5-2698v3 **2.3 GHz** (Cisco labs), Intel® XEON® E5-2699v3 **2.3 GHz** (FD.io CSIT).
[3] Tests done with either HyperThreading Enabled (with **2** threads per physical core) or with Hyperthreading Disabled (**1** thread per physical core).
[4] Cycles per Packet (CPP) calculated using formula: CPP = (core_frequency / packet_non_drop_thput_rate); zero packet loss.
[5] Packet non-drop throughput rate (zero packet loss) per physical CPU core for 64B Ethernet frame size, reported in Millions packets/sec [Mpps].
[6] Bandwidth throughput (zero packet loss) per physical CPU core for IMIX packet sequence, reported in Gigabits/sec [Gbps].
[7] Multi-Core throughput speedup – measured or expected increase of throughput as a function of adding physical CPU cores within the same NUMA node.
[8] Listed results following labs benchmarking VPP: Intel labs, Cisco labs, Intel & Cisco labs executing same tests independently, FD.io CSIT labs numbers from CSIT rls1704 report, https://docs.fd.io/csit/rls1701/report/.

L2 MAC Switching	2.2	2	212	**10.4**	**31**	Linear	Intel labs
L2 MAC Switching	3.2	2	215	**14.8**	**44**	Linear	Intel labs
L2 MAC Switching	2.3	2	221	**10.4**	**31**	Linear	Cisco labs
IPv6 Routing	2.3	2	243	**9.5**	**28**	Linear	Cisco labs
IPv4 Routing	2.3	1	242	**9.5**	**28**	Linear	FD.io CSIT
L2 MAC Switching	2.3	1	228	**10.1**	**30**	Linear	FD.io CSIT
IPv6 Routing	2.3	1	307	**7.5**	**22**	Linear	FD.io CSIT
CGNAT44	2.3	1	359	**6.9**	**20**	Expected Linear	FD.io CSIT
IPv4 IPSec AES-GCM	2.3	1	920	**2.5**	**7**	Linear up to HW limit	FD.io CSIT

We all like numbers, but how do these apply to the actual capacity sizing. With VPP it's actually fairly straightforward - due to VPP data plane multi-threading design being not-locking, it provides a linear speedup once VPP threads run across multiple cores, as per table. In other words if more throughput capacity is needed use more cores, and multiply the per core throughput. This of course assumes no other bottlenecks in the compute systems - and for physical interface to interface (NIC to NIC) scenarios this works just fine, as indicated on the graphs below for IPv4 routing and L2 MAC switching. Results shown are from the same setup as listed in the table.

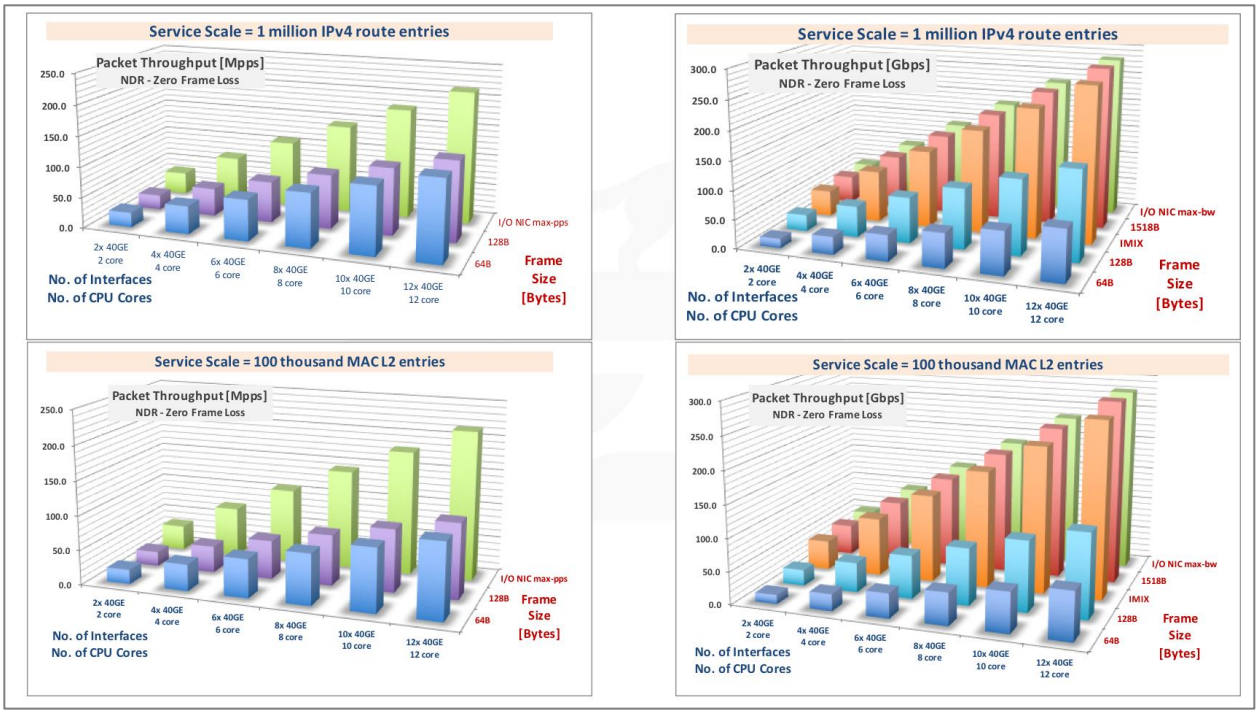

Figure 7. FD.io VPP Data Plane Forwarding Performance – Speedup with Multi-core Multi-threading.

Now, imagine the network I/O bandwidth available per processor socket increasing, from 160 Gbps available on the current Intel® Xeon® processors, to 280 Gbps offered by the new Intel® Xeon® Scalable processors. Imagine a 2RU server with four Intel® Xeon® Scalable processors, that's 4x 280 Gbps in, and 4x 280 Gbps out. That's enough I/O for FD.io VPP to process packets at the rates of One Terabit per Second in, and One Terabit Per Second out. Welcome to a One Terabit Software Router.

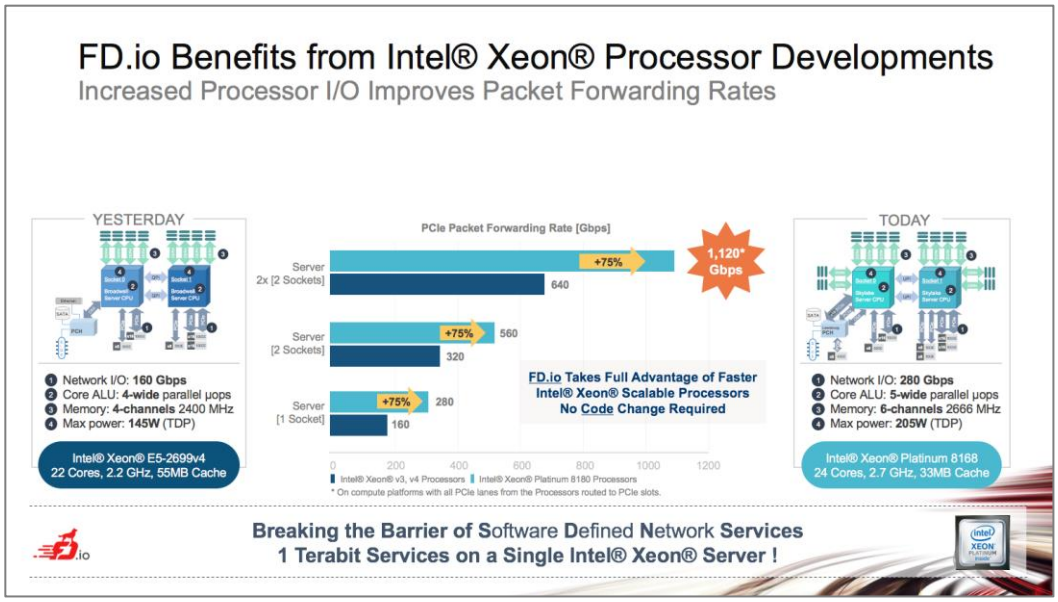

Figure 8. Increased Processor I/O Improves Packet Forwarding Rate per Socket.

When taken for a ride in Cisco labs with Intel® Reference Board “Neon-City”, FD.io VPP One Terabit Software Router performed well. The setup with total of four Intel® Xeon® Platinum 8168 processors running FD.io VPP loaded with half a million of IPv4 /32 routes, was forwarding at the aggregate rate of 948 Gbps with 512B packets, and zero packet loss. Not all cores were used, and throughput was limited only by PCIe I/O slot layout on tested compute machines.

A new dawn of high-speed cloud networking...

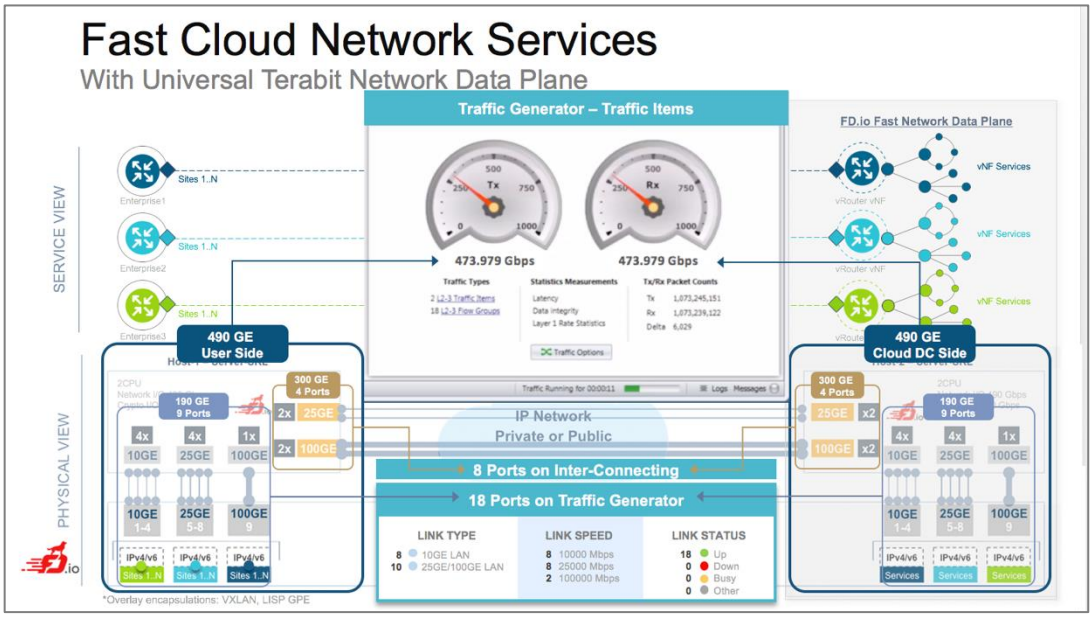

Figure 9. Benchmarking FD.io VPP and Intel® Xeon® Scalable Processors - Setup.

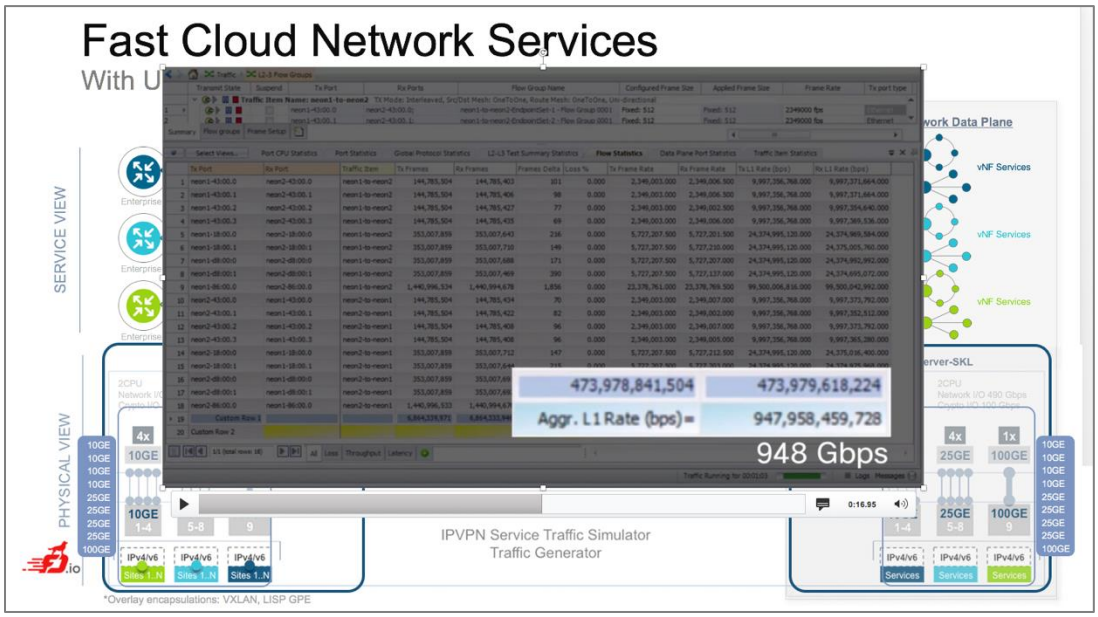

Figure 10. Benchmarking FD.io VPP and Intel® Xeon® Scalable Processors – Bandwidth Rate.

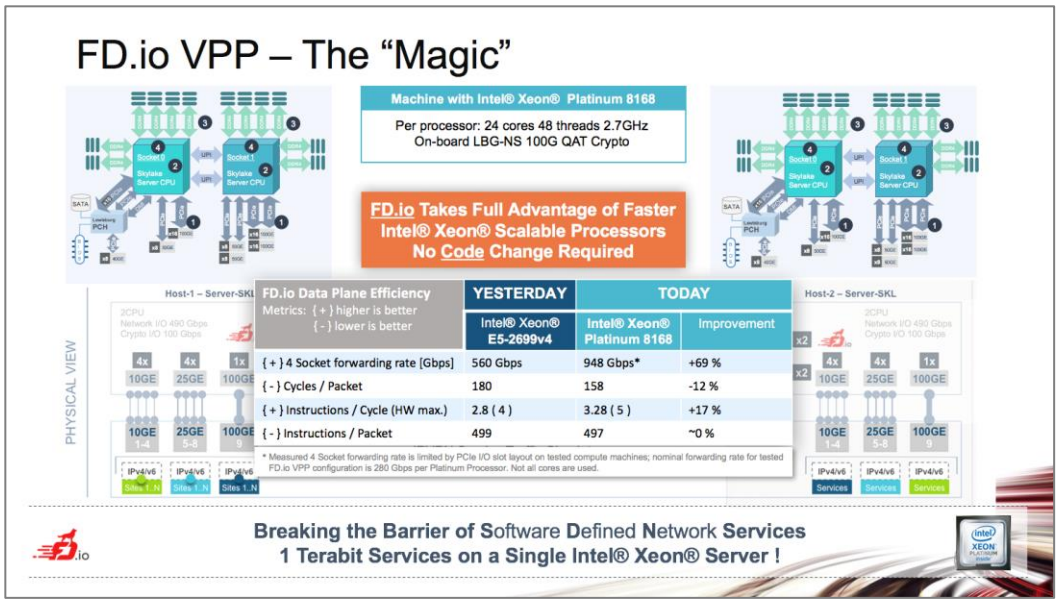

Figure 11. Benchmarking FD.io VPP and Intel® Xeon® Scalable Processors – Efficiency Improvements.

Note that the measured performance was shy of the expected 4x 280 Gb/s rate only due to the lack of PCIe slots on the compute machines used, see specification below.

The specifics for this benchmark: Two of Intel® Reference Boards "Neon-City", each with 2x Intel® Xeon® Platinum 8168 Processors, Intel® C620 PCH, 384GB DDR4 2400 MT/s RDIMMs, 3x FM10420 Intel® Ethernet cards, 3x XXV710 Intel® Ethernet Controllers, Ubuntu 17.04, VPP 17.04.2.

Intel® Xeon® Scalable Processor Family

Intel® Xeon® Scalable processors are the 5th generation Intel® Xeon® processors with new Intel® Mesh Architecture. This processor offers significant performance combined with a rich feature set and cutting-edge technology to address a variety of workloads for optimum performance. Some relevant examples of these workloads include Networking, Communication, Storage, Cloud, and Enterprise.

Intel® Xeon® Scalable processors offer up to 28 cores and 50% more memory channels than the preceding generation, and new AVX-512 instruction which, when coupled with DPDK and network applications like FD.io VPP, offer significant performance improvements. They also offer new and improved crypto and compression acceleration integration (in PCH), integrated Intel® Ethernet with up to 4x 10 GbE interfaces, and reduced total platform investment by converging application, control, and data plane workloads on one platform.

Additional features include High Availability, Carrier Class reliability and long supply life, data security, reduced latency, Enhanced Platform Awareness for optimum resource management for NFV, high performance pattern recognition and high performance I/O.

Summary

FD.io VPP is a software network data plane, a very efficient and flexible one. It consists of a set of forwarding nodes arranged in a directed graph and a supporting software framework. The framework has all the basic data structures, timers, drivers (and interfaces to driver kits like DPDK), a scheduler which allocates the CPU time between the graph nodes, performance and debugging tools, like counters and built-in packet trace. The latter allows you to capture the paths taken by the packets within the graph with high timestamp granularity, giving full insight into the processing on a per-packet level.

FD.io VPP takes full advantage of the latest processor optimizations such as Vector Instructions (e.g. Intel® SSE, AVX) and direct interactions between I/O and CPU cache (e.g. Intel® DDIO), to deliver best in class packet processing performance.

While you cannot expect a router with tens or hundreds of 100GbE ports and multi-terabit performance, we anticipate FD.io VPP running on an industry standard 2RU server built with new Intel® Xeon® Scalable processors, breaking the 1Tbps throughput boundary for the networking data plane.

References

- FD.io
- CSIT rls17.04 performance report
- Need a bigger helping the Internet
- VPP community wiki
- GIT repo