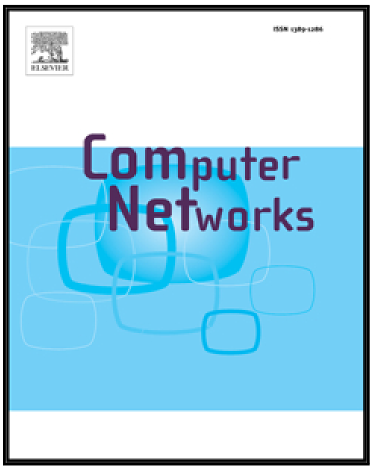


High-Speed Data Plane and Network Functions Virtualization by Vectorizing Packet Processing

Leonardo Linguaglossa, Dario Rossi, Salvatore Pontarelli,
Dave Barach, Damjan Marjon, Pierre Pfister

PII: S1389-1286(18)31295-7
DOI: https://doi.org/10.1016/j.comnet.2018.11.033
Reference: COMPNW 6662

To appear in: *Computer Networks*

Received date: 11 May 2018
Revised date: 19 October 2018
Accepted date: 29 November 2018

Please cite this article as: Leonardo Linguaglossa, Dario Rossi, Salvatore Pontarelli, Dave Barach, Damjan Marjon, Pierre Pfister, High-Speed Data Plane and Network Functions Virtualization by Vectorizing Packet Processing, *Computer Networks* (2018), doi: https://doi.org/10.1016/j.comnet.2018.11.033

High-Speed Data Plane and Network Functions Virtualization by Vectorizing Packet Processing

Leonardo Linguaglossa[1], Dario Rossi[1], Salvatore Pontarelli[2], Dave Barach[3], Damjan Marjon[3], Pierre Pfister[3]

[1] *Telecom ParisTech,* [2] *CNIT and University of Rome Tor Vergata,* [3] *Cisco Systems, Inc.*
first. last@telecom-paristech. fr

Abstract

In the last decade, a number of frameworks started to appear that implement, directly in user-space with kernel-bypass mode, high-speed software data plane functionalities on commodity hardware. This may be the key to replace specific hardware-based middleboxes with custom pieces of software, as advocated by the recent Network Function Virtualization (NFV) paradigm. Vector Packet Processor (VPP) is one of such frameworks, representing an interesting point in the design space in that it offers: (i) in user-space networking, (ii) the flexibility of a modular router (Click and variants) with (iii) high-speed performance (several millions of packets per second on a single CPU core), achieved through techniques such as batch processing that have become commonplace in high-speed networking stacks (e.g. netmap or DPDK). Similarly to Click, VPP lets users arrange functions as a processing graph, providing a full-blown stack of network functions. However, unlike Click where the whole tree is traversed for each packet, in VPP each traversed node processes all packets in the batch (or *vector*) before moving to the next node. This design choice enables several code optimizations that greatly improve the achievable throughput. This paper introduces the main VPP concepts and architecture, and experimentally evaluates the impact of its design choices (such as batch packet processing) on its performance.

Keywords: Software routers, High-speed networking, Vector Packet Processing, Kernel-bypass

1. Introduction

Software implementation of networking stacks offers a convenient paradigm for the deployment of new functionalities, and as such provides an effective way to escape from the network ossification. As a consequence, the past two decades have seen tremendous advances in software-based network elements, capable of advanced data plane functions in common off-the-shelf (COTS) hardware. This eventually evolved in new paradigms such as Network Function Virtualization (NFV), which proposes that classical middleboxes implementing regular network functions can be replaced by pieces of software executing virtual network functions (VNFs). One of the seminal attempts to circumvent the lack of flexibility in network equipment is represented by the Click modular router [30]: its main idea is to move some of the network-related functionalities, up to then performed by specialized hardware, into software functions to be run by general-purpose COTS equipment. To achieve this goal, Click offers a programming language to assemble software routers by creating and linking software functions, which can then be compiled and executed in a general-purpose operating system. While appealing, this approach is not without downsides: in particular, the original Click placed most of the high-speed functionalities as close as possible to the hardware, which were thus implemented as separate kernel modules. However, whereas a kernel module can directly access a hardware device, user-space applications need to explicitly perform system-calls and use the kernel as an intermediate step, thus adding overhead to the main task.

More generally, thanks to the recent improvements in transmission speed and network cards capabilities, a general-purpose kernel stack is far too slow for processing packets at wire-speed among multiple interfaces [11]. As such, a tendency has emerged to implement high-speed stacks *bypassing operating system kernels* (aka kernel-bypass networks, referred to as KBnets in what follows) and bringing the hardware abstraction directly to the user-space, with a number of efforts (cfr. Sec. 2) targeting (i) low-level building blocks for kernel bypass like netmap [38] and the Intel Data Plane Development Kit (DPDK), (ii) very specific functions [24, 39, 36, 33] or (iii) full-blown modular frameworks for packet processing [32, 28, 11, 37, 12].

In this manuscript, we describe Vector Packet Processor (VPP), a framework for building high-speed data plane functions in software. Though being released as open source only recently, specifically within the Linux Foundation project "Fast Data IO" (FD.io) [20], VPP is already a mature software stack in use in rather diverse application domains, ranging from Virtual Switch (SW) in data-center to support virtual machines [7] and inter-container networking [3], as well as virtual network function in different contexts such as 4G/5G [5] and security [6].

In a nutshell, VPP combines the flexibility of a modular router (retaining a programming model similar to that of Click) with high-speed performance. Additionally, it does so in a very effective way, by extending benefits brought by techniques such as batch processing to the whole packet processing path, increasing as much as possible the number of instructions per clock cycle (IPC) executed by the microprocessor. This is in contrast with existing batch processing techniques, that are merely used to either reduce interrupt pressure (e.g., as done by lower-building blocks such as [38, 4, 18]) or are non-systematic and pay the price of a high-level implementation (e.g., in FastClick [11] batch processing advantages are offset by the overhead of managing linked-lists).

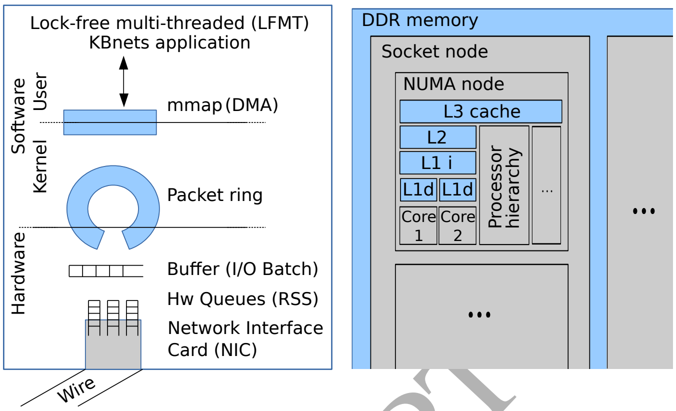


Figure 1: The architecture of a typical COTS software router with special attention the NIC (left) and the memory hierarchy (right).

Our previous analysis [10] shows that VPP can sustain a rate of up to 12 millions packets per second (Mpps) with our general-purpose CPU. We extend our performance evaluation, proving highly predictable performance (as shown by the constrained latency), a throughput scale-up (depending on whether we increase the CPU frequency, use Hyper-Threading or simply allocate more cores) and the possibility to adopt VPP for different use-cases (testified by the different scenarios evaluated).

In the rest of the article, we put VPP in the context of related kernel-bypass effort (Sec. 2). We then introduce the main architectural ingredients behind VPP (Sec. 3), and assess their benefits with an experimental approach, by describing both our methodology (Sec. 4) and our results (Sec. 5). We finally discuss our findings and report on open questions in Sec. 6. In line with the push toward research reproducibility, we make all scripts available in [9] alongside with instructions to replicate our work.

2. Background

This section overviews the state-of-the-art of software data plane: Sec. 2.1 introduces popular and new techniques and Sec. 2.2 maps them to existing KBnets frameworks, of which we provide a compact summary in Tab. 1.

2.1. KBnets-related techniques

By definition, KBnets avoid the overhead associated to kernel-level system calls: to achieve so, they employ a plethora of techniques, which we overview with respect to Fig. 1, that illustrates the general COTS architecture we consider in this work. We now describe in details the columns of Table 1, by following the packet reception path.

Lock-free Multi-threading (LFMT). Current tendency to multi-core COTS allows to embrace a *multi-thread* programming paradigm. Ideally, the *parallelism degree* of a network application, which represents the number of threads simultaneously running, is related to a speed-up in the performance of the system: the more threads available, the better the performance, up to a saturation point where increasing the number of threads does not affect the performance. At the same time, to achieve this ideal speed-up, it is imperative to avoid performance issues tied to the use of synchronization techniques (mutexes, semaphores, etc.) in order to achieve *lock-free operations*. Lock-free parallelism in KBnets is tightly coupled with the availability[1] of multiple hardware queues (discussed next), to let threads operate on independent traffic subsets.

I/O batching (IOB). Upon reception of a packet, the NIC writes it to one of its hardware queues. To avoid raising an interrupt as soon as a new packet is ready to be scheduled for processing, KBnets batch packets in a separate buffer and send an interrupt once the whole buffer is full: the NIC simply writes the buffer via Direct Memory Access (DMA) and appends a reference to its position to the *packet ring* (a circular buffer of memory accessible by both the network device and the user-space application). Overall, I/O batching amortizes the overhead due to the interrupt processing, and can speed-up the overall processing. It is worth pointing out that VPP extends batch processing from pure I/O (which reduces interrupt overhead) to complete graph processing (which ameliorates the efficiency of the CPU pipeline).

RSS Queues. Modern NICs support multiple RX/TX hardware queues, and Receive-side Scaling (RSS) [25] is the technique used to assign packets to a specific queue. RSS queues are generally accessible in userland and are typically used for hardware-based packet classification or to assist (per-flow) multi-thread processing. Depending on hardware capabilities, packets can be simply grouped in flows by means of a hash function over their 5-tuple (grouping both directions is also trivial [42]), but recent NICs support more involved matching (e.g., up to 4096-bit hash filtering, which the framework needs to make accessible in userland).

Zero-Copy (Z-C). When the Network Interface Card (NIC) has some packets available, it writes them to a reserved memory region, which is shared between the network interface and the operating system. Early KBnets required user-space applications to access this memory through system calls (i.e., a memory copy operation), whereas in most of the latest KBnets approaches the user-space application has Direct Memory Access to the memory region used by the NIC. Notice that zero-copy is sustainable only in case the packet consumer is faster than the packet arrival rate (occasional slowdown may need application-level buffers or drops). Typical Zero-Copy approaches leverage the fact that most of the network applications perform little to no operations to the packet payload, which can therefore be left unchanged, and only a subset of metadata (usually the L2/L3 addresses) is provided to the user-space application via the NIC driver.

Compute Batching (CB). When packets are retrieved from the NIC, the software can start the packet processing.

[1] Even when a single receive queue is available, a software scheduler (potentially the system bottleneck) can assign different packets to different threads, and then perform independent processing.

Table 1: State of the art in KBnet frameworks

Framework [Ref.]	LFMT	IOB	RSS	Z-C	CB	CC&L	LLP	Main Purpose
DPDK[4]	✓	✓	✓	✓				Low-level IO
netmap[38]	✓	✓	✓	✓				
PacketShader[24]	✓	✓	✓					Routing
MTclass[39]	✓	✓	✓					Classification
Augustus[29]	✓	✓	✓	✓				Name-based fwd
HCS[33]	✓	✓	✓	✓				Caching
Click[30]								Modularity
RouteBricks[32]	✓		✓					
DoubleClick[28]	✓	✓	✓	✓				
FastClick[11]	✓	✓	✓	✓	partial			
BESS/SoftNIC[23]	✓	✓	✓	✓	✓	✓		
VPP (this work)	✓	✓	✓	✓	✓	✓	✓	

This is typically performed on a per-packet basis, that is, all network functions in the processing path are applied to the same packet until the final forwarding decision. With compute batching, the notion of I/O batching is extended to the processing itself, meaning that the network functions are implemented to treat *natively* batches of packets rather than single packets. Similarly to the I/O batching, CB helps mitigating the overhead of function accesses (e.g. context switch, stack initialization) as well as providing additional computational benefits: when a batch of packets enters the function, the code of the function will be loaded and a miss in the L1 instruction cache will occur only for the first packet, while for all other packets the code to be run will be already in the cache; furthermore, the code might be written to exploit low-level parallelism (i.e. when instructions on subsequent packets are independent) in order to increase the number of instructions issued at the same clock cycle. CB can provide performance speed-up or not, depending on how systematic it is the implementation of CB techniques (cfr. Sec. 5.2).

Cache Coherence & Locality (CC&L). A major bottleneck for software architecture is nowadays represented by memory access [13]. At hardware level, current COTS architectures counter this by offering 3 levels of cache memories with a faster access time (for the sake of illustration, the cache hierarchy of an Intel Core i7 CPU [17] is reported in the right of Fig. 1). In general, L1 cache (divided into instruction L1-i and data L1-d) is accessed on per-core/processor basis, L2 caches are either per-core or shared among multiple cores, and L3 caches are shared within a NUMA node. The speed-up provided by the cache hierarchy is significant: access time of a L1 cache is about 1ns, whereas access time to a L2 (L3) cache is about 10ns (60ns) and in the order of 100ns for the main DDR memory. When the data is not present at a given level of the cache hierarchy, a cache miss occurs forcing access to the higher levels, slowing the overall application.

Low-level parallelism (LLP). Together with the userland parallelism, a lower level of parallelism can be achieved by exploiting the underlying CPU micro-architecture, which consists of a multiple stages pipeline (`instruction_fetch` or `load_store_register` are two examples of such stages), one or more arithmetical-logical units (ALU) and branch predictors to detect "if" conditions (which may cause pipeline invalidation) and maintain the pipeline fully running [26]. An efficient code leads to (i) an optimal utilization of the pipelines and (ii) a higher degree of parallelism (that is, executing multiple instructions per clock cycle). Furthermore, giving "hints" to the compiler (e.g. when the probability of some "if condition" is known to be very high) can also improve the throughput. As we shall see, the vectorized processing, coupled with particular coding practices, can exploit the underlying architecture, thus resulting in a better throughput for user-space packet processing. To the best of our knowledge, VPP is among (if not the) first approaches to leverage systematic low-level parallelism through its design and coding practices.

2.2. KBnets frameworks

We identify three branches of software frameworks for high-speed packet processing based on kernel bypass, depending on whether they target low-level building blocks [38, 4, 18], very specific functions [24, 39, 36, 33] or full-blown modular frameworks [32, 28, 11, 37, 12].

Low-level building blocks. This class of work has received quite a lot of attention, with valuable frameworks such as netmap [38], DPDK [4] and PF_RING [18]. In terms of features, most of them support high-speed I/O through kernel-bypass, zero-copy, I/O batching and multi-queuing, though subtle[2] differences may still arise among frameworks [11] and their performance [21]. A more detailed comparison of features available in a larger number of low-level frameworks is available at [11], whereas an experimental comparison of DPDK, PF_RING and netmap (for relatively simple tasks) is available at [21]. Worth mentioning are also the eXpress Data Path (XDP)[41], which embraces similar principles but in a kernel-level approach, and the Open Data Plane (ODP) project [8], an open-source cross-platform set of APIs running on several hardware platforms such as x86 servers or networking System-on-Chip processors.

Purpose-specific prototypes. Another class of work is represented by prototypes that are capable of a very specific and restrained set of capabilities such as IP routing [24], traffic classification [39], name-based forwarding [36] or transparent hierarchical caching [33]. In spite of the different goals, and the possible use of network processors [36] or GPUs [24], a number of commonalities arise. PacketShader [24] is a GPU accelerated software IP router. In terms of low-level functions it provides kernel-bypass and I/O batching, but not zero-copy. MTclass [39] is a CPU-only traffic classification engine capable of working at line rate, employing a multi-thread lock-free programming paradigm; at low-level, MTclass uses PacketShader hence inheriting the aforementioned limitations. Prototypes in [36, 29] and [33] address high-speed solutions of two specific functions related to Information-centric networking (ICN) architectures, namely name-based forwarding and caching ([36] employs a network processor whereas [29, 33] uses DPDK). In all these cases, multi-thread lock-free programming enabled by RSS queues is the key to scale-up operations in user-space. In contrast to these efforts, VPP aims for generality, feature richness and consistent performance irrespectively of the specific purpose.

Full-blown modular frameworks. Full-blown modular frameworks are closer in scope to VPP. Letting aside relevant but proprietary stacks [2], work such as [37, 12, 32, 28, 11] is worth citing. In more details, Arrakis [37] and IX [12] are complete environments for building network prototypes, including I/O stack and software processing model. Arrakis' main goal is to push further kernel-bypass

[2] A limitation of netmap that it does not allow to directly access the NIC's registers [19]

beyond network stack functionalities, whereas IX additionally separates some functions of the kernel (control plane) from network processing (data plane), and is as such well suited to building SDN applications.

Closest work to VPP is represented by Click [30], which shares the goal of building a flexible and fully programmable software router. Whereas the original Click cannot be listed among KBnets applications (as it requires a custom kernel, and runs in kernel-mode thus being not suited for high-speed processing), however a number of extensions have over the years brought elements of KBnets into Click. Especially, RouteBricks [32], DoubleClick [28], FastClick [11] all support the kernel version of Click, introducing support for HW multi-queue [32], batching [28], and high-speed processing [11], possibly obtained through a dedicated network processor [31]. Click has also inspired work such as the Berkeley Extensible Software Switch (BESS, formerly known as SoftNIC) [23] which presents a similar programming model and implements the most important software acceleration techniques. Important differences among Click (and variants) and VPP arise in the scheduling of packets in the processing graph, and are discussed further in Sec. 3.

The evolution of full-blown modular frameworks is the key to enable NFV composition and deployment. Most of the recent NFV frameworks are in fact built on top of one of the aforementioned tools: we cite, for instance, ClickNF [22], a modular stack for the composition of L2-L7 network functions built on FastClick; the E2 [34], a framework for creating and managing virtual functions built on top of BESS; NetBricks [35], a clean-slate approach that leverages pure DPDK and the Rust [1] programming language to provide high-speed NFV capabilities.

3. VPP Architecture

Initially proposed in [16], VPP technology was recently released as open-source software, in the context of the FD.io Linux foundation project [20]. In a nutshell, VPP is a framework for high-speed packet processing in user-space, designed to take advantage of general-purpose CPU architectures. In contrast with frameworks whose first aim is performance on a limited set of functionalities, VPP is feature-rich (it implements a full network stack, including functionalities at layer 2, 3 and above), and is designed to be easily customizable. As illustrated in Fig. 2, VPP aims at leveraging recent advances in the KBnets low-level building blocks early illustrated: as such, VPP runs on top of DPDK, netmap, etc. (and ODP, binding in progress) used as input/output nodes to the VPP processing. It is to be noted that non-KBnets interfaces such as `AF_PACKET` sockets or tap interfaces are also supported.

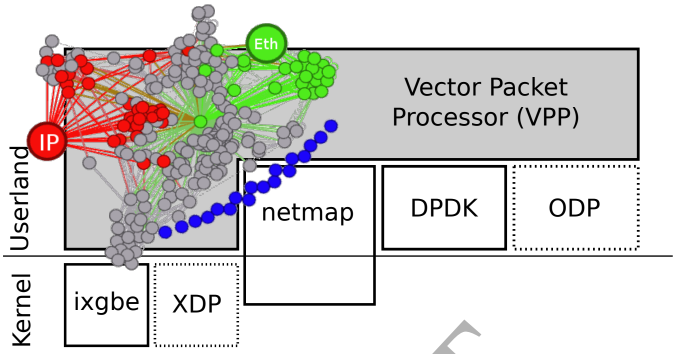


Figure 2: VPP scope and processing tree. The nodes we use later in the experiments (as well as their neighbors) are highlighted in red (IP) and green (Eth). Process nodes are depicted in blue.

At a glance. VPP's processing paradigm follows a "run-to-completion" model. First a batch of packets is polled using a KBnets interface (like DPDK), after which the full batch is processed. Poll-mode is quite common as it increases the processing throughput in high traffic conditions (but requires 100% CPU usage regardless of the traffic load conditions), whereas native compute batch is a novel ingredient.

VPP is written in C, and its sources comprise a set of low-level libraries for realizing custom packet processing applications as well as a set of high-level libraries implementing specific processing tasks (e.g. `l2-input`, `ip4-lookup`) representing the *main core* of the framework.

User-defined extensions, called *plugins*, may define additional functionalities or replace existing ones (e.g., `flowperpkt-plugin`, `dpdk-plugin`). The main core and plugins together form a *forwarding graph*, which describes the possible paths a packet can follow during processing.

In more details, VPP allows three sets of nodes: namely *process*, *input*, and *internal* (which can be terminating leaves, i.e., output nodes). Process nodes (blue nodes in Fig. 2) do not participate in the packet forwarding, being simply software functions running on the main core[3] and reacting to timers or user-defined events. Input nodes abstract a NIC interface, and manage the initial vector of packets. Internal nodes are traversed after an explicit call by an input node or another internal node. For some nodes, a set of fine-grained processing tasks (aka *features*[4] in VPP's terminology) can be activated/deactivated on demand at runtime.

VPP architecture adopts all well-known KBnets techniques discussed in Sec. 2, to which it adds a design (Sec. 3.1) and coding practices (Sec. 3.2) that are explicitly tailored to (i) minimize the data cache misses using data prefetching, (ii) minimize the instruction cache misses, (iii) increase the instructions per cycle that the CPU front-end can fetch, and that we describe in what follows.

3.1. Vectorized processing

The main novelty of VPP is to offer a systematic way to efficiently process packets in a "vectorized" fashion: instead of letting each packet traverse the whole forwarding graph, each node processes all packets in a batch, which provides sizable performance benefits (cfr. Sec. 5.2).

Input nodes produce a vector of work to process: the graph node dispatcher pushes the vector through the directed graph, subdividing it as needed, until the original vector has been completely processed. At that point, the process recurs. Notice that not all packets follow the same path in the forwarding graph (i.e., vectors may be different from node to node). While it is outside of the scope to provide a full account of all the available nodes [40], Fig. 3 compactly depicts a subset of the full VPP graph (comprising 253 nodes and 1479 edges) and illustrates the vectorized processing. We consider a case of a vector consisting in a mixture of traffic, and then focus on classical IPv4 processing for the sake of the example. Notice how the overall processing is decoupled in different components, each of them implemented by a separate node. VPP's workflow begins with a node devoted to packet reception (`dpdk-input`), and then the full vector is passed to the next node dealing with packet parsing (`l2-input`). Here the vector can be split in case of multiple protocols to process. After this step, we enter the IPv4 routing procedure (split in `ip4-input`, `ip4-lookup`, `ip4-rewrite`). The workflow finally ends in a forwarding decision (`l2-forward`) for the re-assembled vector. A drop decision may be taken at every step (`error-drop`).

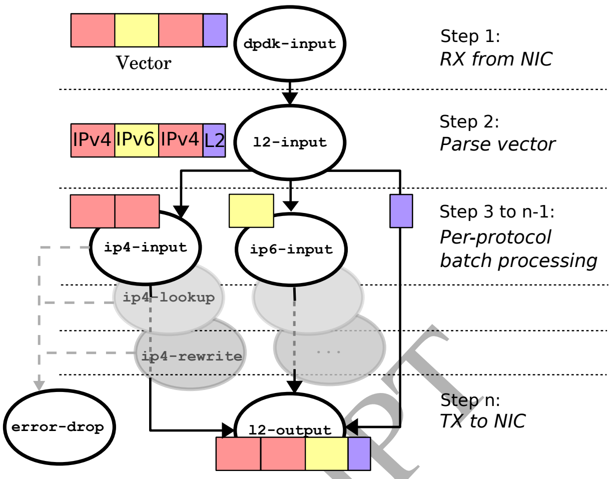


Figure 3: Illustration of the vectorized packet processing model.

Advantages of vectorized processing. In a classic "run-to-completion" [30, 11] approach, different functions of the graph are applied to the same packet, generating a significant performance penalty. This penalty is due to

[3] VPP features its own internal implementation of cooperative multitasking [14], which allows running of multiple process nodes on the main core.

[4] Which are not functions and thus do not incur function call overhead.

several factors. (i) The instruction cache miss rate increases when a different function has to be loaded and the instruction cache is already full. (ii) There is a non-negligible framework overhead tied to the selection of the next node to access in the forwarding graph and to its function call. (iii) It is difficult to define a prefetching strategy that can be applied to all nodes, since the next execution node is unknown and since each node may require to access to a different portion of the packet data.

VPP exploits a "per-node batch processing" to minimize these effects. In fact, since a node function is applied to all the packets in the batch, instruction misses can occur only for the first packet of the batch (for a reasonable codesize of a node). Moreover, the framework overhead is shared among all the packets of the batch, so the per-packet overhead becomes negligible when the batch size is of hundreds of packets. Finally, this processing enables an efficient data prefetching strategy. When the node is called, it is known which packet data (e.g. which headers) are necessary to perform the processing task. This allows to prefetch the data for the $(i+1)$-th packet while the node processes the data of the i-th packet.

Vector size. Notice that while the *maximum* amount of packets per vector can be controlled, the *actual* number of packets processed depends on the duration of the processing tasks, and on the number of new packets arrived during this time. Intuitively, in case of a sudden increase in the arrival rate, the next vector will be longer. However, the processing efficiency (measured in terms of clock cycles per packet) increases by amortizing fixed costs over a larger number of elements. Thus, when a batch is larger, it is expected that the processing time of each packet in the batch decreases, and so does the size of the subsequent input vector. In practice, this sort of feedback loop helps maintaining a stable equilibrium in the vector size.

Vector processing vs I/O Batching. In some sense, VPP extends I/O batching to the upper layers of KBnets processing. However, if the goal of batching I/O operations is to reduce the interrupt frequency, the goal of vectorized processing is to decrease the overall numbers of clock cycles needed to process a packet, amortizing the overhead of the framework over the batch. These two goals are complementary.

Vector processing vs Compute Batching. It is worth pointing out that tools such as G-opt [27], FastClick [11] and the pipelines of the DPDK Packet Framework [4] do offer some form of "Compute Batching", which however barely resemble to batching in VPP only from a very high-level view, as several fundamental differences arise on a closer look. In G-opt batching serves only the purpose of avoiding CPU stalls due to memory latency. The pipeline model of DPDK Packet Framework is used to share the processing among different CPUs and is not focused on improving the performance on a single core. Instead, FastClick "Compute Batching" (see Sec 5.7 in [11]) and the BESS [23] implementation, are close in spirit to VPP.

However, the functions implemented in any VPP node are designed to systematically process vectors of packets. This natively improves performance and allows code optimization (data prefetching, multi-loop, etc). In contrast, nodes in FastClick implicitly process packets individually, and only specific nodes have been augmented to *also* accept batched input. Indeed, per-vector processing is a fundamental primitive in VPP. Vectors are pre-allocated arrays residing in contiguous portions of memory, which are never freed, but efficiently managed in re-use lists. Additionally, vector elements are 32-bit integers that are mapped to 64-bit pointers to the DMA region holding the packet with an affine operation (i.e., multiplication and offset that are performed with a single `PMADDWD` x86 instruction). In FastClick, batches are constructed by using the *simple linked list* implementation available in Click, with significantly higher memory occupancy (inherently less cacheable) and higher overhead (adding further 64-bit pointers to manage the list).

Ultimately, these low-level differences translate into quite diverse performance benefits. VPP's vectorized processing is lightweight and systematic (as for the BESS compute batching): in turn, processing vectors of packets increase the throughput consistently, and our measurements confirm that the treatment of individual packets significantly speeds up. In contrast, opportunistic batching/splitting overhead in FastClick, coupled to linked list management yields limited achievable benefits in some cases and none in others (e.g., quoting [11], in "*the forwarding test case [...] the batching couldn't improve the performance*").

3.2. Low-level code optimization techniques

As mentioned before, batch processing in VPP enables additional techniques to exploit all lower-level hardware assistance in user-space processing.

Multi-loop. We refer to *multi-loop* as a coding practice where any function is written to explicitly handle N packets with identical processing in parallel: since computations on packets $i, \dots, i + N$ are typically independent of each other, very fine-grained parallel execution can be exploited letting CPU pipelines be continuously full. The CPU front-end can in fact execute in parallel several instructions applied to data coming from different packets in the same clock cycle. Sec. 5 verifies the efficiency of this technique, by measuring the IPC achievable enabling or disabling multi-loop for some VPP nodes. This technique provides significant performance improvements for certain processing nodes (see Sec. 5 for details) but it presents two limitations: (i) the multi-loop technique is applied writing C code that is explicitly parallel[5] (the programmer leverages C template code to write multi-loop functions); (ii) the multi-loop technique increases the number of IPC removing data dependency, but does not provide benefit when the performance bottleneck is due to the number of memory accesses.

Data prefetching. Once a node is called, it is possible to prefetch the data that the node will use for the $i + 1$-th packet while processing the i-th packet. Prefetching can be combined with multi-loop, i.e. prefetching data for packets from $i + 1$ to $i + N$ while processing packets from $i - N$ to i. This optimization does not work at the vector bounds: for the first N packets of the batch no prefetching is possible, while for the last N packets there is no further data to prefetch. However, since in the standard setting VPP uses a quad-loop ($N = 4$) or a dual-loop ($N = 2$), and the vector size is 256, this effect is negligible. We verify in Sec. 5 the efficiency of this technique by measuring IPC with prefetching enabled or disabled.

Branch-prediction. This practice is based on the assumption that most processing will follow a "Pareto law" in that the majority of packets will require very similar processing and thus follow the same path in the graph. VPP encourages a coding practice where programmers give the compiler some expert hints about the most likely case in an if-then-else branch: in case the prediction is true (which happens often due to the Pareto law), a pipeline reset is avoided saving clock cycles (possibly more than 10 in practice). If the prediction is false, then additional processing is needed (i.e., invalidate pipelined operations and revert to previous state), with however a low impact on the average case (since mispredictions are rare due to the Pareto law). While Branch Predictor (BP) is a very powerful component in the latest CPUs, and so the majority of compiler hints are unnecessary, it still can be relevant upon BP failures (e.g., when many branches are present in a small part of code).

Function flattening. In VPP a majority of graph nodes

[5] Automating the deployment of multi-loop functions is an on going work.

make use of *inline* functions. This avoids the cost of reshuffling registers to comply with the Application Binary Interface (ABI) calling convention and avoids stack operations. As a beneficial side effect, flattening likely yields to additional optimizations by the compiler (e.g., removing unused branches).

Direct Cache Access (DCA). As an extension of the zero-copy operation achieved through DMA, in case of Direct Data IO (DDIO) systems, it is possible to prefill packets directly in the L3 cache, so that with a careful buffer allocation strategy it is possible to achieve (or at least aim for) close-to-zero RAM memory utilization.

Multi-architecture support. VPP supports runtime selection of graph node function code optimized for specific CPU micro-architecture: e.g., at run-time the same binary can detect and execute code which utilizes AVX2 (e.g, on Haswell/Broadwell) and still work on systems without AVX2 support (e.g., Atom Sandy/Ivy Bridge).

4. Methodology

This section describes the experimental setup and the methodology that we use to assess the impact of VPP architectural choices as well as the scenarios that we evaluate. We start by providing a quick overview of our hardware and software setup in Sec. 4.1 and Sec. 4.2 respectively; we describe the evaluated scenarios in Sec. 4.3, and we detail our metrics in Sec. 4.4.

4.1. Hardware Setup

We reproduce our experimental environment as suggested in the RFC2544 by connecting one device under test (DUT) running VPP to one measurement device equipped with a packet transmitter and a packet receiver, referred to as Traffic generator and sink (TGS). Performance is evaluated at the endpoints of the measurement device. This setup is represented in Fig. 4.

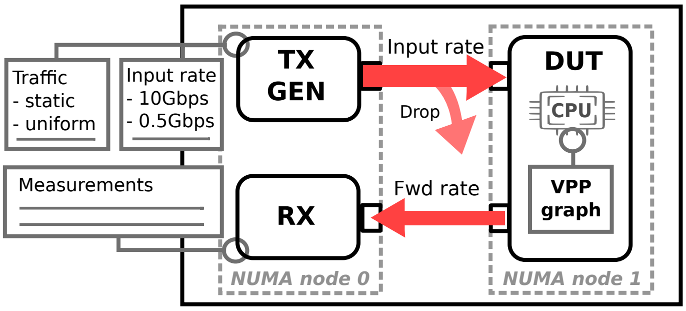


Figure 4: Hardware setup.

Our hardware consists of a server with 2× Intel Xeon Processor E52690, each with 12 physical cores running at 2.60 GHz in hyper-threading and 576KB (30MB) L1 (L3) cache. Each processor socket is attached to a NUMA node (95 GByte of RAM, for a total of 190 GB). The server is equipped with 2× Intel X520 dual-port 10Gbps NICs, that are directly connected with SFP+ interfaces. The server runs a vanilla Linux kernel 4.8.0-41 (Ubuntu 16.04.3).

In order to reproduce the scenario depicted in Fig. 4, we separate the DUT and the TGS assigning one line card to the DUT and another one to the TGS; within the TGS, we assign one port for the traffic generation (TX GEN) and the other one for the the measurements (RX). The TGS and the DUT are further isolated being physically located in two different NUMA nodes (each one attached to a different CPU socket).

Since we are interested into gathering insights on the performance gains tied to the different aspects of the whole VPP architecture, as opposed to gathering raw absolute performance data for a specific system, we study *per-core* performance – intuitively, provided a good lock-free multi-threaded application design, the overall system performance can be deduced by aggregating the performance of individual cores. The CPU core dedicated to the VPP forwarding can only run the VPP software and no other processes are ever scheduled to it[6].

[6] The system tuning procedures are described in details on the project wiki page at `https://wiki.fd.io/view/VPP/How_To_Optimize_Performance_(System_Tuning)`

4.2. Software setup

We describe in the following the software setup for both the DUT and the two components of the TGS.

DUT. VPP (version 17.04) starts with a static configuration that is maintained for the duration of a single experiment. The configuration, stored in a plain text file, describes the size of the memory to allocate, the number of cores to use as well as the number of RSS queues.

We provide isolation by instructing one VPP instance to allocate 4 GB of RAM from the NUMA node 1. This is done via the use of hugepages of size 1 GB. Such memory is used for both packet I/O and software data structures. VPP is assigned to a single core running in isolation mode (i.e. no other threads are ever scheduled to this core) and with TurboBoost[7] deactivated. Finally, both input and output interfaces are configured with a single RSS queue.

Notice that VPP is always running a full forwarding graph – which includes all the supported protocols, as well as nodes for managing the command-line interface (CLI), logging and management operations. Yet, in a more common setup, tasks such as CLI, logging and management would typically run in a separate core (the so-called *main core*, whereas operations related to the data-path would be executed in different cores (aka *workers*), bound to different RSS queues. Our results are obtained in the configuration where all tasks are executed in a single core, which is shown to provide conservative results (cfr. Sec. 5.7 where we consider in addition the model of a main core plus a number of workers).

TX GEN and RX. Both traffic generation and measurements processes are executed in the same server placed on NUMA node 0. Isolation is provided with the same technique already described for the DUT (separate cores in isolation with TurboBoost disabled).

Traffic generation and measurements are performed by the `DPDK-pktgen` standard traffic generator, running on two separate cores assigned to two separate RSS queues with an allocated RAM of 4 GB. This software provides a command-line interface and a programmable kit of functionalities for rate selection and packet crafting, which is sufficient for forwarding rate measurement. For latency measurements we use the MoonGen [19] traffic generator, still based on DPDK, which provides the API to obtain the observed latency via the use of Precision Time Protocol (PTP) packets.

Both traffic generators exports several NIC counters to the user space: the RX components is in fact executed within the same process dealing with traffic generation. Here the RX periodically checks the interface's packet counters for measuring the forwarding rate, or the difference between packet timestamps to measure latency.

4.3. Scenarios

To gather results representative of different network operations, we consider different input workloads and processing tasks. In particular, our scenarios should be representative of realistic use-cases where VPP can be used either as a pure software-router, or as a tool for deploying VNFs on COTS equipment. In line with the push toward research reproducibility, we release all scripts to configure the workloads and the VPP tasks available at [9].

Input workload. We vary the L2/L3 addresses generated by the TX GEN. The most simple case is represented by a static traffic, where packets with the same source/destination pair are continuously sent to the DUT. For our second scenario we perform a round-robin variation in the IP destination address by incrementing the least significant bits by one in the range [1 : 255]. In this scenario, the input traffic is deterministically predictable.

[7] This feature of modern CPUs consists in allowing a higher clock frequency w.r.t. the nominal one. However, this cannot be sustained for long time, and a cool-down period can occur where the clock frequency is kept at a lower rate, thus giving non-deterministic results.

Finally, we reproduce a scenario in which packets are randomly generated: in particular, the TX GEN generates packets by selecting the IP source/destination pair uniformly at random.

Processing task. The processing task is chosen at runtime by executing some scripts via the VPP CLI. We recreate three different scenarios, with an increasing level of complexity.

The least complex scenario is the cross-connect (XC), where packets are just forwarded from the input interface to the output port. This scenario is the closest to a pure I/O task, but it is important to underline that VPP still runs a full forwarding graph, where only a few nodes are accessed and packets follow always the same path (the nodes accessed are `dpdk-input, l2-input, l2-output` and the final node representing the output NIC, cfr. Fig. 3).

The second scenario is the classic IPv4 processing (IP). VPP is configured to read a routing table, taken from the RIPE database[8] which contains 130k entries with different prefix lengths. Whether the complexity is higher than the XC scenario, the VPP graph still contains a single path followed by all incoming traffic.

In order to add a bifurcation in the forwarding graph, we engineered a mixed scenario (MIX) in which VPP is configured to deal with packets from different protocols: in particular, we choose IPv4, IPv6 and L2 forwarding altogether, so that the forwarding graph contains at least three different paths. This scenario is closer to a realistic application of VPP in an NFV environment in which the software may execute different virtual network functions deployed on the same commodity hardware.

4.4. Metrics

We observed in Sec.3.1 that we can configure at compile time the maximum vector size, but the actual vector size depends on the processing task. This can be measured by logging the actual vector size under different input rate generated by the TX GEN. This is the first metric considered in our results.

When the TX rate is at its maximum (that is 10Gbps for our line cards) it is possible to measure the throughput of the system representing the maximum capability of the DUT+VPP to run under stress conditions. As observed in [38], I/O performance are dominated by per-packet operations: i.e., it is only slightly more expensive to send a 1.5KB packet than a 64B one. To stress the system, we consider the VPP packet-level processing rate $\tilde{R}$ for the smallest 64B packet size. This is measured by sending 10 Gbps traffic from the TX GEN and gathering the forwarding rate observed at the RX side after the VPP processing. In addition to the overall throughput, we can also measure other interesting variables related to the underlying CPU architecture: we thus consider (i) the number of CPU instructions per packet (IPP), that represents the "weight" of the SW framework; (ii) the number of instructions per clock cycle (IPC), that is related to the capability of the framework in instruction parallelization; (iii) miss-rate in the L1 Instruction cache, that shows the benefit of the compute batching technique.

We measure the latency observed in the VPP graph, defined as the difference between the departure time of a packet from the TX GEN and the arrival time at the RX. This kind of measurement is not defined in a lossy regime (or otherwise latency values would be undefined for such packets). In fact, the RFC2544 suggests to measure latency at the maximum rate at which none of the input packets are dropped, aka non-drop rate (NDR). Since the forwarding rate that we measure does not assure that zero packets are lost, we configure the TX GEN to send packets at 99% of the observed forwarding rate. We complete our latency measurements by providing the latency observed also at 50% and 10% of the forwarding rate.

We finally provide a sensitivity analysis, measuring the throughput obtained on different CPUs, with an increasing

[8] `https://www.ripe.net/`

number of cores and different thread placements.

5. Experimental results

In the following we present our experimental results. In particular, we evaluate the effect of the traffic load on the actual vector size in Sec. 5.1, for the default value of the maximum allowed vector size (the `VLIB_FRAME_SIZE`). We then evaluate the impact of the maximum vector size in Sec. 5.2; we analyze the per-node processing cost in Sec. 5.3, the benefits given by coding practices in Sec. 5.4, as well as of the impact of exogenous factors such as the input workload in Sec. 5.5. We measure the latency in Sec. 5.6, and we further evaluate VPP under different core allocation and CPU frequencies in Sec. 5.7. We conclude the section with a comparison w.r.t the DPDK `l3fwd` application in Sec. 5.8.

5.1. Vector size as a function of the rate

VPP controls the *maximum* number of packets to be processed by varying the maximum batch size, the `VLIB_FRAME_SIZE`, controlling the delay due to batching. We underline that batching is especially useful for high traffic rates, since it allows to increase the instruction per clock-cycle efficiency. Conversely, at low traffic rates the NIC queues seldom fills, so that the vectors polled from the NIC at time t typically have a size smaller than `VLIB_FRAME_SIZE`.

Fig. 5 reports actual batch sizes polled from the NIC for increasing input traffic rates. We observe that the vector size is generally small, and only as the input load approaches the processing capacity, the vector size grows to `VLIB_FRAME_SIZE`. Otherwise stated, vectors grow only when this is actually needed. This additionally confirms that the expected delay will be negligible in practical non-overloaded scenarios.

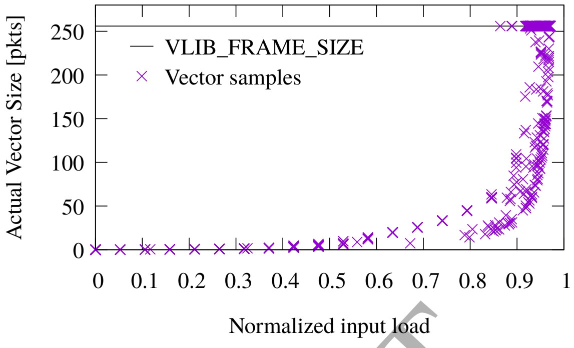


Figure 5: Actual vector size as a function of the traffic rate for `VLIB_FRAME_SIZE`=256 in realistic scenarios.

5.2. Impact of maximum vector size

While the previous experiment was performed with a fixed `VLIB_FRAME_SIZE` and a variable input rate, we now fix the input rate at 10 Gbps (14.88 Mpps of 64-byte packets) and we vary the `VLIB_FRAME_SIZE`. The maximum vector size can be changed at compile time (to a minimum of 4 packets, due to quad-loop operations in some nodes): intuitively, increasing the frame size increases the L1-instruction cache hit benefits, but also increases the average per-packet delay.

Fig. 6 presents a set of metrics gathered from three main use cases, namely (i) *Cross-connect (XC)* case, (ii) a *IP longest-prefix-match (IP)* forwarding and (iii) a *mixed traffic (MIX)* in which the incoming traffic activates L2, IPv4 and IPv6 nodes (cfr. Sec. 4.3). Such plots depict a set of key performance indicators (y-axis) as a function of the `VLIB_FRAME_SIZE` (x-axis). Experiments are repeated 10 times, so that in addition to the average of the metric of interest, we report as well the minimum and maximum values[9] over the repetitions.

The per-core packet processing rate in Mpps is reported in Fig. 6a: for all cases, the packet processing rate increases linearly with the vector size, up to a saturation point where increasing the size further does not bring noticeable benefits. When the vector size becomes too large (greater than 512 packets), a slight penalty arises, probably due to the overhead of managing a larger memory

[9] We prefer to report the full range since performance are tightly clustered around the mean and 95% confidence intervals are hardly distinguishable.

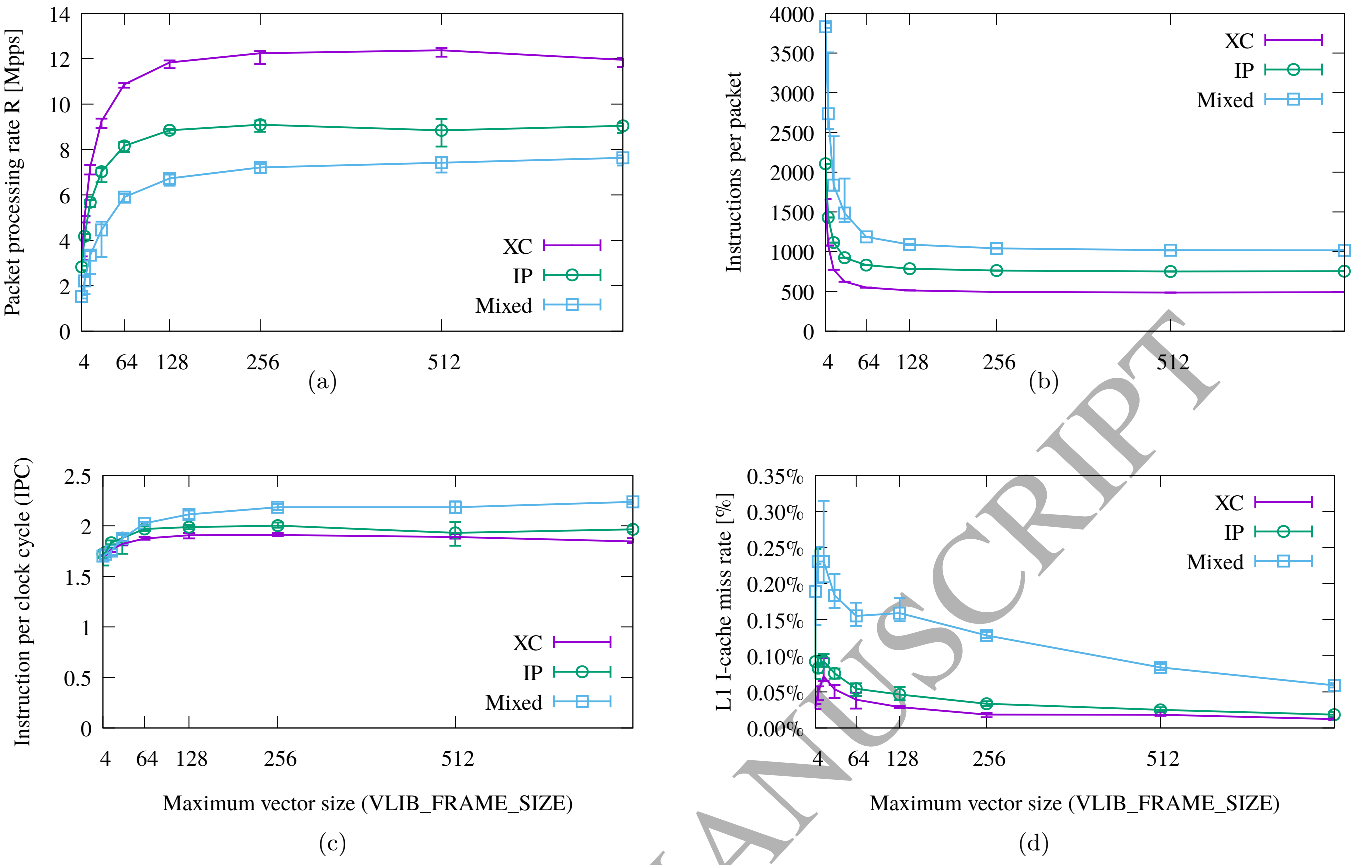


Figure 6: Packet processing performance as a function of the maximum vector size (10Gbps traffic rate on a single-core): (a) packet processing rate, (b) average number of instructions per packet, (c) average number of instructions per clock cycle and (d) miss rate of L1 i-cache.

portion. This holds across all types of processing function: interestingly, for the IP case, a single core is able to forward over 8 million packets per second, for a fairly large FIB. In a sense, the XC gap with the 14.88Mpps line rate can be considered as a measure of the VPP framework overhead (but see Sec. 5.7–5.8 for more details). Increasing the frame size also increases the average per-packet latency: for all the traffic cases, the knee in the curve at about 256 packets-per-vector corresponds to a sweet-spot with maximum throughput and bounded *maximum* delay (see latency measurements in Sec. 5.6).

Fig. 6b shows the average number of instructions per packet with respect to the maximum vector size. The decrease of the number of instructions relates to the amount of code that is executed just once per vector (e.g., call of the input DPDK function, scheduling of the node graph, etc.), and that represents the overhead of the processing framework. Even if this overhead is directly related to the VPP framework, we argue that any kind of software-based packet processing framework will experience similar overhead. Instead of optimizing the code to minimize this overhead, an easier and most efficient solution consists in *sharing this overhead over several packets*, as VPP does.

Fig. 6c reports the average number of instructions per clock cycle (IPC), which is related to the ability of VPP to optimize the code execution by leveraging multi-loop and prefetching strategies discussed in Sec. 3.2. We observe that the IPC significantly increases when the vector size grows up to 256. In particular, for the XC case the increase is around 10%, while for the IP forwarding case it is around 20%.

Finally, Fig. 6d shows the average L1 instruction-cache misses occurred for each packet: as expected the miss rate decreases when the vector size increases, thus avoiding stalling in the CPU pipeline and improving the processing capabilities. An L1 instruction-cache miss requires to access the L2 level cache, and each miss has a penalty of around 10 clock cycles: the reduction of i-cache miss rate corresponds to a saving of roughly 30-40 clock cycles for each processed packet.

5.3. Per-node breakdown

We now analyze Fig. 7, which shows the per-packet processing cost, measured in clock cycles, spent in each node for the XC and L2, IPv4 and IPv6 traffic. Nodes are grouped by colors, depending on their activity. Gray nodes represent the low-level I/O functionalities, mainly consisting in accessing the NIC through DPDK drivers. Then, we have the processing functions, which are related to L2 (green), IPv4 (red) and IPv6 (yellow) processing.

Notice that XC consists only of gray nodes, and the per-packet total cost is about 200 clock cycles per packet. For the L2, IPv4 and IPv6 cases some processing is introduced, but a big component of the processing cost is still related to the I/O. In fact, for L2 forwarding, only few tens of clock cycles are spent in the actual processing, while the rest is due to I/O.

For both IPv4 and IPv6 some gray nodes are not present: this is related to some code optimization, that can early detect the protocol of the packet and can anticipate some functions to be run directly in a processing node. For instance, both IPv4 and IPv6 lacks of `l2-input` and `l2-output` nodes, that are replaced by `ip4-input` and `ip4-output` (`ip6-` respectively).

Finally, we notice that in the IPv4 and IPv6 scenarios a major cost is represented by the lookup function. Here we can observe the implementation difference between the IPv4 mtrie and the IPv6 hashtable, for which we highlight a cost of about 40 clock cycles for IPv4 and about 70 cycles for IPv6 (an increment of almost 50% w.r.t. IPv4).

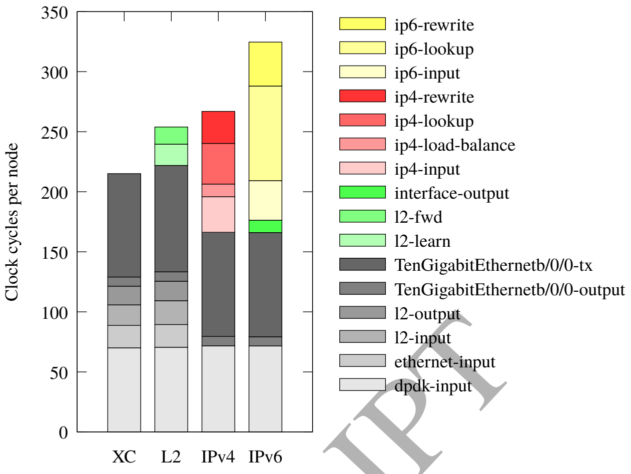


Figure 7: Per-node breakdown for different traffic scenarios

5.4. Multi-loop

Fixing `VLIB_FRAME_SIZE` to 256, we next consider the impact of multi-loop programming practice due to the instruction parallelization (improvement in terms of IPC) and prefetching (avoiding stalls due to data cache misses). We recall that the multi-loop technique aims at amortizing per-packet processing by parallelizing instructions over multiple independent packets, thus increasing the IPC that the CPU can issue. It works well for nodes that require significant processing (i.e. there are several instructions to execute). However, the multi-loop benefits are negligible in the XC case (the active nodes have few instructions, so that multi-loop does not actually increase the IPC, making this technique useless in this case). We consider for this experiment two different scenarios that require more processing, and we focus on longest-prefix matching on IPv4 and IPv6 traffic.

Fig. 8 reports the number of cycles per packet obtained by enabling and disabling the quad-loop and prefetching options, for IPv4 and IPv6 processing. Rather than reporting aggregate processing rates, we report the average packet processing duration expressed in CPU cycles.

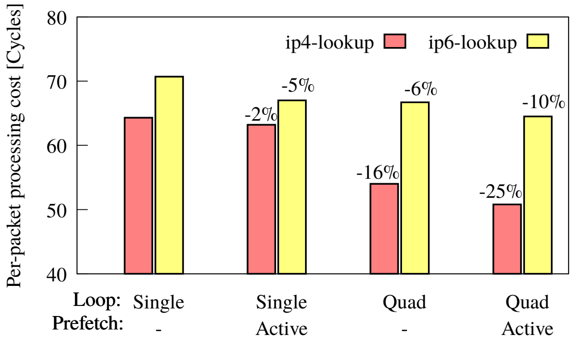


Figure 8: Impact of multi-loop and prefetching

While the difference between IPv4 and IPv6 is tied to the data structures used for lookups (tries for v4 and hashtables for v6), the figure clearly shows that the quad-loop technique sizably reduces the number of clock cycles both when prefetching is enabled (25% for IPv4) or disabled (16%).

5.5. Input workload

We next turn our attention to the impact of different input workloads on the processing rate. As before, we consider the XC, IP and MIX cases. In particular, we now consider a packet arrival processes with a different spatial variability for the source and destination IPs, specifically: (i) a *Static* scenario where traffic corresponds to a single source-destination pair; (ii) a *Round-robin* scenario where destination is simply incremented by one and (iii) a *Uniform* case where source/destination pair is extracted uniformly at random.

In the XC case, we don't expect any noticeable effect. Concerning the IP forwarding case, we instead expect the Static case to correspond to a best case since, in addition to the L1-i cache hit, the IP lookup can take advantage also of a L1-d cache hit. Similarly, IP lookups in the round-robin traffic case can leverage L1-d cache hits to some extent (it depends on the length of the FIB netmask: specifically, for a IP/x netmask the first miss is followed by $2^{32-x}-1$ hits) whereas the Uniform random case can be expected to be close[10] to the worst-case.

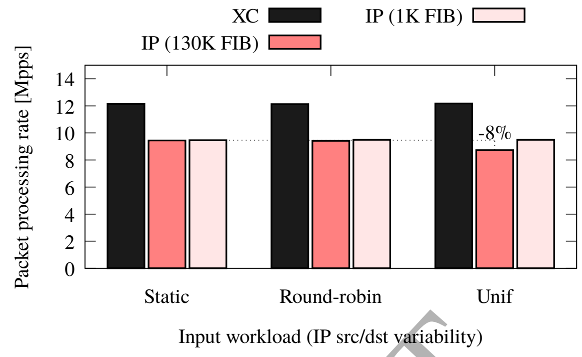


Figure 9: Variation of the packet processing rate as a function of the input workload process.

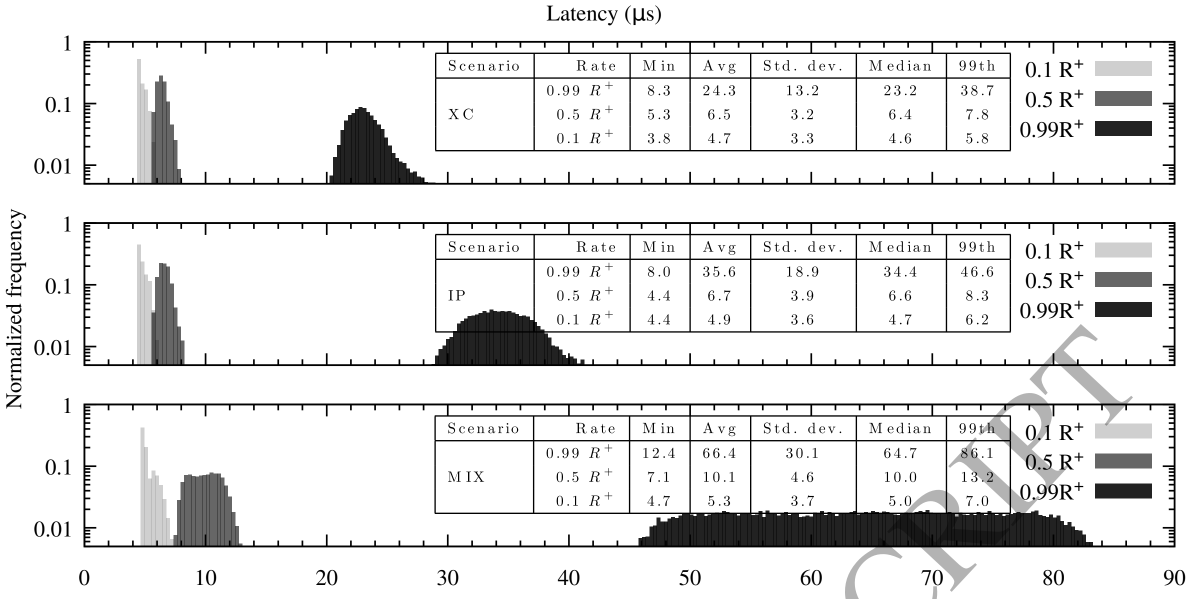
Results, shown in Fig. 9, confirm the expectation. Not only XC performance are hardly affected by stochastic properties of the spatial IP address distribution, but it is especially interesting to notice that also L3 performance are only minimally affected: the drop in IP packet forwarding rate from Static to Uniform case is limited to less than 10%, hinting to the fact that the L1-d cache misses implied by random traffic generation have a minor impact with respect to the L1-i cache hits obtained by processing packets in vectors. This confirms the soundness of VPP design, and the robustness of its potential gain.

5.6. Latency

The batch processing may impact the latency that packets experience in a VPP router, because of the forwarding operation occurring only after a full graph traversal for a given batch. Furthermore, we may expect that latency is also affected by the system load: whether a high input traffic rate corresponds in bigger vectors, a lower traffic may reduce the batch sizes thus reducing the overall latency.

As shown in Sec. 5.2, when the maximum vector size is fixed at 256 packets, VPP is able to deal with about 12 Mpps, which is lower than the line rate of 14.88, meaning

[10] In practice, L1-d cache hits can happen, even though with low probability, in the uniform random case; a truly adversarial worst-case could be obtained by using a linear shift register so to produce a sequence that minimizes the L1-d cache hit, with however a significant engineering effort for a minimum expected difference.



Scenario	Rate	Min	Avg	Std. dev.	Median	99th
XC	0.99 R^+	8.3	24.3	13.2	23.2	38.7
	0.5 R^+	5.3	6.5	3.2	6.4	7.8
	0.1 R^+	3.8	4.7	3.3	4.6	5.8

Scenario	Rate	Min	Avg	Std. dev.	Median	99th
IP	0.99 R^+	8.0	35.6	18.9	34.4	46.6
	0.5 R^+	4.4	6.7	3.9	6.6	8.3
	0.1 R^+	4.4	4.9	3.6	4.7	6.2

Scenario	Rate	Min	Avg	Std. dev.	Median	99th
MIX	0.99 R^+	12.4	66.4	30.1	64.7	86.1
	0.5 R^+	7.1	10.1	4.6	10.0	13.2
	0.1 R^+	4.7	5.3	3.7	5.0	7.0

Figure 10: Latency measured for three different scenarios (XC, IP, MIX) represented through a histogram. The measured values obtained at three different rate conditions (10%, 50% and 99% of the forwarding rate) are arranged in the x-axis, while the y-axis is the normalized frequency in log-scale of the occurrences. We report as well each scenario's minimum, average, standard deviation, median and 99th percentile.

that the system is in overload (that is, we have a drop rate of 2.88 Mpps). Latency computation would provide non-valid results in such a scenario: we tune the input rate to match the *Maximum Forwarding Rate (R^+)*, which is the maximum rate for which the DUT is able to forward packets with zero losses[11]. Thus, we measure the maximum forwarding rate as observed in Fig. 6a for a `VLIB_FRAME_SIZE` equals to 256, and we consider the 10%, 50% and 99% of such rate, referred to as 0.1, 0.5 and 0.99 R^+.

We report our measurements in Fig. 10, that shows the latency, measured in microseconds, for the XC, IP and MIX scenarios and for the three different input rates. Measured values are organized in a histogram w.r.t. normalized frequency (in log-scale). We first observe that at low input rates (0.1 $R^+$and 0.5 R^+) the average values are similar across the XC, IP and MIX cases, all falling in the range [4.7, 10.1]μs. The higher values for the average and standard deviation in the MIX scenario are compensated by the fact that the 99th percentile is bounded at 13.2μs. In other words, when the system is not overloaded, the measured latency is similar independently from the scenario that we consider.

Focusing on the 0.99 R^+case, we observe that, as expected, the XC scenario shows the lowest values of latency, being on average 24.3μs. In this scenario, as soon as packets are received at the input port, a vector is immediately created and forwarded to the output port (therefore, the graph traversal is minimal, and consists only to few nodes dealing mostly with I/O). Latency increases when VPP starts performing a regular graph traversal, as shown by Fig.10. The average latency is 35.6μs and 66.4 μs for the IP and the Mixed case respectively. The histogram shows that the latency histogram moves towards higher values of latency when the scenario changes from XC to IP and then to the MIX case. At the same time, the standard deviation increases more for the MIX case, meaning that values are less concentrated among the average, thus resulting in a more spread histogram. This increase is due

[11] Precisely calculating the non-drop rate requires a dichotomic search to find an the input rate with no packet losses, but we found it very difficult to accurately measure it with our software traffic generator, as both DPDK-pktgen and MoonGen experience non-deterministic drops in the order of some parts per millions, which are however sufficient to bias the NDR computation.

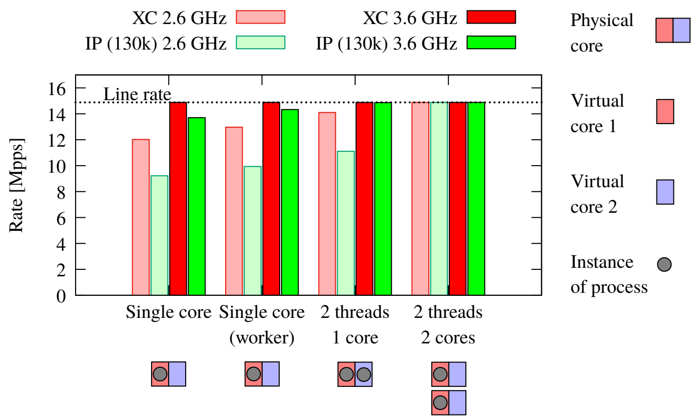


Figure 11: VPP performance for different CPUs (2.6GHz vs 3.6GHz) and core allocation strategies. We plot, left to right, results for a single core running management and forwarding graph; then a single core running just the forwarding graph (1 worker); we then place two workers on a single core and finally two workers on different cores.

to the presence of multiple paths in the forwarding graph: a packet in the first positions of the first sub-vector would experience a higher latency than the last packet of the last sub-vector since the final forwarding operation should wait for all the batch to be processed (and in the case a path split, the small batches have to be re-aggregated at the last node).

We finally highlight that even for the MIX scenario (our worst case) the 99th percentile is smaller than 90 μs and the standard deviation is 30.1 μs: this proves deterministic performance and a small worst case value, testifying that VPP is suitable for many applications in the context of high-speed networking in an NFV environment.

5.7. Impact of core allocation

Processing and forwarding rates are tightly coupled with characteristics of the HW architecture, such as the cache size, the CPU frequency, the thread placement etc. We show in Fig. 11 the achievable forwarding rates for a CPU running at 2.6 GHz (the one used in the manuscript) as well as for a second one running at a higher clock speed of 3.6 GHz. For both experiments we first focus on VPP running on a single core (i) that is our baseline. We then separate the main core from the workers (cfr. Sec. 4.2) and place (ii) one worker on a single core; (iii) two workers on a single core in HyperThreading; (iv) two workers on two separate cores.

The figure clearly shows that working at 3.6 GHz, VPP (XC) is able to sustain line rate in all scenarios, including the most conservative one. Similarly, the IPv4 forwarding performance are very close to the line rate already for a single worker (i)-(ii) and achieve the line rate already for two workers sharing a physical core (iii). Clearly, as IPv4 forwarding was already achieving line rate at 2.6GHz on two cores, this is still the case at 3.6GHz (iv). We point that this behavior also applies for the IPv6 case: with our 2.6 GHz CPU, the throughput of a simple IPv6 scenario ranges from 7.78 Mpps on a single core, to 8.10 Mpps when using a separate worker, to reach 9.19 Mpps when placing two workers on a single physical core and finally ending close to line rate, at 14.15 Mpps when allocating two separate cores.

These results confirm that VPP is able to sustain the 14.88Mpps line rate for XC and IP on on a single core for different architectures. However we preferred to report conservative results in this article (as all our experiments refer to the 2.6GHz single core case). Indeed, while quantitative results may vary (as hardware obsoletes by design), the observations we gather here are qualitatively valid to a larger extent.

5.8. Comparison with `l3fwd` DPDK application

We finally confront VPP with open-source software performing similar functions: we choose the default DPDK `l3fwd` application. At the same time, we point out that there is a *fundamental difference concerning the operations* performed by VPP and those performed by `l3fwd`: notably, VPP L3 operations include *all* necessary operations for a fully fledged IP router, which not only involve *lookup*, but also modification of the forwarded packets (i.e., TTL decrement, checksum update). Conversely, `l3fwd` is a specialized function, built on top of the DPDK library, which

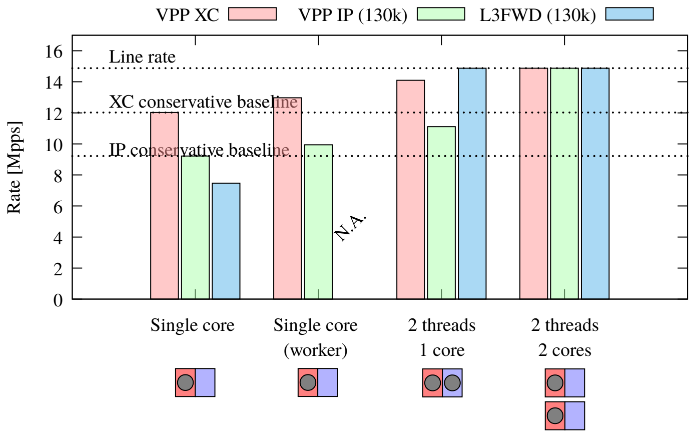


Figure 12: VPP vs DPDK `l3fwd` performance for different core allocation strategies

merely accesses the `mbuf` memory in read mode to perform the L3 lookup, but is not otherwise writing to the memory to modify the packet payload. Given the cost of memory accesses, this is an important difference to keep in mind when comparing VPP and `l3fwd` performance.

Furthermore, there is a second *fundamental difference concerning the system level architecture* of VPP and the `l3fwd`, which concerns in particular the DPDK and VPP programming model and notably impacts the core allocation. VPP runs as a router, and it is aware of all the interfaces it deals with: thus, in VPP a thread is bound (via the DPDK drivers) to a NIC RSS queue, typically allocated in a separate core. On the contrary, `l3fwd` uses the DPDK poll-mode driver that avoids lock contention allocating a different logical core (lcore) to each receive NIC[12]. Thus, the simplest DPDK `l3fwd` case is represented by two NICs, governed using two cores (in our scenario, one core for the RX on the first NIC, and one core for the TX on the second NIC).

With these differences in mind, we report in Fig.12 a comparison of XC and L3 lookup for VPP and `l3fwd` in the same experimental conditions of the paper, for 4 different configurations: (i) single core (ii) single core worker (iii) 2 threads 1 core (iv) 2 threads 2 cores.

Notice that (i) is the conservative scenario used throughout the paper for VPP: in this scenario, for which the DPDK `l3fwd` example is not designed for, VPP outperforms `l3fwd`. Notice then that in the typical case (ii) of a dedicated per-worker core, which is not supported by `l3fwd`, the performance of VPP ameliorate mainly because of the reduced interference with non-data-plane tasks (and despite the main core being almost idle). The `l3fwd` application is instead optimized for scenario (iii), where it indeed achieves line rate forwarding: VPP performance benefits of the increased number of cores, although the rate does not match that of `l3fwd`. This is due to two main reasons: first, the fully fledged VPP router also has to alter the forwarded packets (unlike the simpler `l3fwd` application); second, the high IPC observed in Sec. 5.2 (already higher than 2 on average) implies that VPP performance can ameliorate (but not double) when run over multiple logical cores in hyper-threading. Finally, (iv) both `l3fwd` and VPP achieve line rate IPv4 forwarding over fairly large FIBs when using 2 cores.

To summarize the results for different core allocations shown in Fig.12, we deduced that using a similar core allocation in VPP allows to reach line rate with only 2 cores at 2.6 GHz in both XC and IP experiments; at the same time, running the `l3fwd` example in a scenario in which all resources are used in a single core at 2.6 GHz (hence, locking mechanisms are required and `l3fwd` is not optimized for this) lowers the overall performance.

6. Conclusion

This paper introduces Vector Packet Processing (VPP), a feature-rich high-performance software router, both illustrating its main novelty from an architectural viewpoint, as well as extensively assessing its performance through experiments. Thanks to its flexibility and high performance (up to more than 12 Mpps on a single commodity CPU core), VPP may allow network managers to deploy NFV

[12] see section 8.1 of the DPDK manual `https://dpdk.org/doc/guides/prog_guide/poll_mode_drv.html`)

functions on COTS hardware with hardware-comparable performance. We now summarize our main findings.

Compute batching. With respect to the previously introduced techniques, it is worth pointing out that VPP extends *batching* from pure I/O, which merely reduces interrupt overhead, to *complete graph processing*, in a systematic fashion so that it amortizes the software framework overhead). Indeed, even though extension of batching to processing is attempted in FastClick [11], it is interesting to notice that its implementation is done at a much higher level (i.e., on top of the processing Click graph, only from dedicated elements with separate APIs, and implemented with linked lists) than in VPP, where batch processing becomes the sole low-level primitive. The low-level vectorized processing primitive offered by VPP is key to its performance, as it increases both data cache hit rate (with prefetching) and instruction cache hit rate (inherently, since the same node functions are repeated over packets in the batch), as well as increasing the processor IPC (using multi-loop).

Performance analysis. We evaluate the VPP performance, both in stress tests and realistic scenarios, demonstrating the soundness of VPP's architectural design. We show that the vectorized processing approach allows to experience high-speed packet processing while keeping low latency values. We show that the design is scalable, as line rate is achieved on commodity CPUs with a small number of cores. We explain the relationship of the maximum vs actual vector size as a function of the load, which is of uttermost importance for the framework gain. We also quantify the precise cycle-level breakdown of the function cost in each node of the VPP framework, to show the relative cost. We show the impact of varying workload (random/deterministic/round-robin spatial traffic) on VPP performance and we finally assess the predictability of packet processing latency across a spectrum of load and functions.

Comparison with `l3fwd`. We prove that the overhead of the framework is limited, by comparing VPP with the simple `l3fwd` DPDK application that, differently from VPP, implements only a set of layer 3 functionalities (the longest-prefix matching). We compare the performance of VPP and `l3fwd` on a set of multi-core configurations, showing that VPP can achieve a similar forwarding rate and, at the same time, a superset of L2/L3 and L4 functionalities.

Open points. VPP is a mature software router that provides a full-router functionalities, well-suited for NFV applications or any high-speed network application in commodity hardware. At the same time, VPP is not the only high-speed software packet forwarding framework available nowadays, several of which are overviewed in [15]: as such, interesting questions that remain now open are (i) a detailed quantitative performance comparison of such frameworks, as well as (ii) a more systematic study in regimes where the aggregate traffic flowing on a single box exceed the 100Gbps, where the PCIx express bus can become a bottleneck.

Acknowledgments

This work has been carried out at LINCS (`http://www.lincs.fr`) and benefited from support of NewNet@Paris, Cisco's Chair "Networks for the Future" at Telecom ParisTech (`https://newnet.telecom-paristech.fr`).

Bibliography

References

[1] Rust - system programming language. `https://www.rust-lang.org/en-US/`, (visited on 04-10-2018).

[2] 6WIND Products. `http://www.6wind.com/products/`, (visited on 10-09-2018).

[3] Contiv. `http://contiv.github.io/`, (visited on 10-09-2018).

[4] Data Plane Development Kit. `http://dpdk.org`, (visited on 10-09-2018).

[5] Intel's GTP-U implementation. `https://goo.gl/CHYTCn`, (visited on 10-09-2018).

[6] NetGate's pfSense. `https://www.netgate.com/solutions/pfsense/`, (visited on 10-09-2018).

[7] Networking VPP. https://wiki.openstack.org/wiki/Networking-vpp, (visited on 10-09-2018).

[8] The Open Data Plane (ODP) Project. https://www.opendataplane.org/, (visited on 10-09-2018).

[9] Vector packet processor benchmark. https://newnet.telecom-paristech.fr/index.php/vpp-bench, (visited on 10-09-2018).

[10] Dave Barach, Leonardo Linguaglossa, Damjan Marion, Pierre Pfister, Salvatore Pontarelli, and Dario Rossi. High-speed Software Data Plane via Vectorized Packet Processing. In *IEEE Communication Magazine (to appear)*, 2018.

[11] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *ACM/IEEE ANCS*, 2015.

[12] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *USENIX OSDI*, 2014.

[13] Peter A Boncz, Stefan Manegold, Martin L Kersten, et al. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, 1999.

[14] Joe Bradel. Non-preemptive multitasking. *The Computer Journal*, 30, 1988.

[15] D. Cerovic, V. Del Piccolo, A. Amamou, K. Haddadou, and G. Pujolle. Fast packet processing: A survey. *IEEE Communications Surveys Tutorials*, pages 1–1, 2018.

[16] D. Barach and E. Dresselhaus. Vectorized software packet forwarding, June 2011. US Patent 7,961,636.

[17] David Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf, (visited on 10-09-2018).

[18] Luca Deri et al. Improving passive packet capture: Beyond device polling. In *Proc. of SANE*, 2004.

[19] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: a scriptable high-speed packet generator. In *ACM IMC*, 2015.

[20] The Linux Foundation. Fast Data Project (FD.io). https://fd.io, (visited on 10-09-2018).

[21] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle. Comparison of frameworks for high-performance packet io. In *ACM/IEEE ANCS*, 2015.

[22] Massimo Gallo and Rafael Laufer. Clicknf: a modular stack for custom network functions. In *2018 USENIX Annual Technical Conference (ATC)*, 2018.

[23] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. Softnic: A software nic to augment hardware. *Tech.Rep.*, 2015.

[24] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-accelerated software router. In *ACM SIGCOMM*, 2010.

[25] Tom Herbert and Willem de Bruijn. Scaling in the linux networking stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt, 2011.

[26] Intel. Haswell Micro-architecture Reference Manual. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html, (visited on 10-09-2018).

[27] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G Andersen. Raising the bar for using gpus in software packet processing. In *USENIX NSDI*, 2015.

[28] Joongi Kim, Seonggu Huh, Keon Jang, KyoungSoo Park, and Sue Moon. The power of batching in the click modular router. In *Asia-Pacific Workshop on Systems*, 2012.

[29] Davide Kirchner, Raihana Ferdous, Renato Lo Cigno, Leonardo Maccari, Massimo Gallo, Diego Perino, and Lorenzo Saino. Augustus: a ccn router for programmable networks. In *ICN*, 2016.

[30] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and Frans Kaashoek. The Click Modular Router. *Operating Systems Review*, 34(5):217–231, 1999.

[31] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, and Peng Cheng. ClickNP. In *ACM SIGCOMM*, 2016.

[32] M. Dobrescu et al. Routebricks: exploiting parallelism to scale software routers. In *SIGOPS*, 2009.

[33] Rodrigo Mansilha, Lorenzo Saino, M Barcellos, M Gallo, E Leonardi, D Perino, and D. Rossi. Hierarchical Content Stores in High-speed ICN Routers: Emulation and Prototype Implementation. In *ACM ICN*, 2015.

[34] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A framework for nfv applications. In *SOSP*. ACM, 2015.

[35] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. Netbricks: Taking the v out of nfv. In *USENIX OSDI*, 2016.

[36] Diego Perino, Matteo Varvello, Leonardo Linguaglossa, Rafael P. Laufer, and Roger Boislaigue. Caesar: a content router for high-speed forwarding on content names. In *ACM/IEEE ANCS*, 2014.

[37] Simon Peter, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System as Control Plane. *ACM Transactions on Computer Systems*, 38(4):44–47, 2013.

[38] Luigi Rizzo. netmap: a novel framework for fast packet I/O. In *USENIX ATC*, 2012.

[39] Pedro M. Santiago del Río, D. Rossi, F. Gringoli, L. Nava,

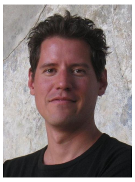
L. Salgarelli, and J. Aracil. Wire-speed statistical classification of network traffic on commodity hardware. In *ACM IMC*, 2012.

[40] The Linux Foundation. FD.io's VPP Features. `https://wiki.fd.io/view/VPP/Features`, (visited on 10-09-2018).

[41] The Linux Foundation - IO Visor. The eXpress Data Path (XDP) project. `https://www.iovisor.org/technology/xdp`, (visited on 10-09-2018).

[42] Shinae Woo and KyoungSoo Park. Scalable TCP session monitoring with symmetric RSS. In *KAIST Tech. Rep.*, 2012.

Leonardo Linguaglossa is a post-doctoral researcher at Telecom ParisTech. He received his MSc. in telecommunication engineering from University of Catania in 2012, and his PhD in Computer Science in 2016 through a joint program among Alcatel-Lucent Bell Labs, INRIA and University Paris 7. His research interests focus in the design and prototyping of systems for high-speed networks.



Dario Rossi is Professor at Telecom ParisTech and Ecole Polytechnique. He received his MSc and PhD degrees from Politecnico di Torino in 2001 and 2005 respectively. He has coauthored over 150 conference and journal papers, received several best paper awards, a Google Faculty Research Award (2015), and an IRTF Applied Network Research Prize (2016). He is a Senior Member of IEEE and ACM.

Salvatore Pontarelli is a CNIT Senior Researcher at University of Rome Tor Vergata. He received his MSc. in electronic engineering from University of Bologna in 2000 and his PhD from the University of Rome Tor Vergata in 2003. He participated in several national and EU funded research programs. His research interests include hash based structures for network applications, HW design of SDN devices and stateful programmable data planes.

David Barach is a Cisco Fellow specializing in networking data-plane codes. He is the inventor of the Vector Packet Processor code: before the recent open-source release, VPP principles were implemented in most of the high-speed Cisco routers.

Damjan Marion is a Principal Engineer of the Chief Technology and Architecture Office (CTAO) at Cisco Systems. He is a regular commiter of many open-source projects, among which the Fast Data I/O project (FD.io) and of the VPP source code relase in particular.

Pierre Pfister is a Software Engineer at Cisco. He received a MSc from the Ecole Polytechnique in 2012. He is an active participant and author at IETF (homenet, 6man, bier and hackathons) and co-developed the reference implementation of HNCP on OpenWrt platform. He is now commiter to FD.io and active contributor to the VPP project.