# High-speed Software Data Plane via Vectorized Packet Processing

David Barach[1], Leonardo Linguaglossa[2], Damjan Marion[1], Pierre Pfister[1], Salvatore Pontarelli[2,3], Dario Rossi[2]
[1]Cisco Systems, [2]Telecom ParisTech, [3]CNIT - Consorzio Nazionale Interuniversitario per le Telecomunicazioni
{first.last}@cisco.com, {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. 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) the benefits brought by techniques such as batch processing that have become commonplace in high-speed networking stacks (such as 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 (called *vector*) before moving to the next node. This design choice enables several code optimizations that greatly improve the achievable processing throughput. This paper introduces the main VPP concepts and architecture, and experimentally evaluates the impact of design choices (such as batch packet processing) on performance.**

## I. Introduction

Software implementation of networking stacks offers a convenient paradigm for the deployment of new functionalities, and provides an effective way to escape from network ossification. Therefore, the past two decades have seen tremendous advances in software-based network elements, capable of advanced data plane functions in commodity servers. One of the seminal attempts to circumvent the lack of flexibility in network equipment is represented by the Click modular router [1]: its main idea is to move some of the network-related functionalities, up to then performed by specialized hardware, into software functions executed by general-purpose computers. 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 commodity server. The original Click approach 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, Click applications need to explicitly perform system calls and use the kernel as an intermediate step. This overhead is not negligible: in fact, due to recent improvements in transmission speed and network card capabilities, a general-purpose kernel stack is far too slow for processing packets at wire-speed over multiple interfaces [2]. As such, a tendency has emerged to implement high-speed network stacks via *kernel bypass* (referred to as KBstacks in what follows) and bringing the hardware abstraction directly to the user-space, with a number of efforts (cfr. Sec. II) targeting both the low-level building blocks for kernel bypass like netmap [4] and the Intel Data Plane Development Kit (DPDK), and the design of full-blown modular frameworks for packet processing [2, 5]. Interested readers can find an updated survey of fast software packet processing techniques in [6].

In this paper, we report on a framework for building high-speed data plane functionalities in software, namely Vector Packet Processor (VPP). VPP is a mature software stack already in use in rather diverse application domains, ranging from Virtual Switch in data-center to support virtual machines[1] and inter-container[2] networking, as well as Virtual Network Function (VNF) in different contexts such as 4G/5G[3] and security[4].

In a nutshell, VPP offers the flexibility of a modular router, retaining a programming model similar to that of Click. 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 reduce interrupt pressure (as done by netmap [4] or DPDK) or are non-systematic and pay the price of a high-level implementation (e.g., in FastClick [2] batch processing advantages are offset by the overhead of linked-lists).

The rest of the paper is organized as follows: in Sec. II we put VPP in the context of related work. We then introduce the main elements of the VPP architecture in Sec. III, and in Sec. IV we assess their benefits with an experimental approach. We finally discuss our findings in Sec. V.

## II. Background

Almost all the software frameworks for high-speed packet processing based on kernel bypass share some commonalities. By definition, KBstacks avoid the overhead associated with kernel-level system calls and additionally employ a plethora of techniques. For the sake of space, we only briefly define them here, and refer the reader to e.g., [2] or [15] for a more complete description. At hardware level, KBstacks leverage

[1]NetworkingVPP https://wiki.openstack.org/wiki/Networking-vpp
[2]Contiv http://contiv.github.io/
[3]Intel's GTP-U implementation https://ossna2017.sched.com/event/BEN4/improve-the-performance-of-gtp-u-and-kube-proxy-using-vpp-hongjun-ni-intel
[4]NetGate's pfSense (today called TNSR) https://www.netgate.com/

Network Interface Card (NIC) support of multiple RX/TX hardware queues such as *Receive-side Scaling (RSS)*. RSS queues are made accessible to userland software and used for hardware-based packet classification and especially to assist (per-flow) *Lock-free multi-threaded (LFMT)* processing. Additionally, KBstacks are Non Uniform Memory Access (NUMA) aware, and exploit memory locality when running on NUMA systems (like multi-processor systems based on Intel Xeon processors). Furthermore, to avoid costly memory copy operations, latest KBstacks use *Zero-copy* techniques via Direct Memory Access (DMA) to the memory region used by the NIC. Finally, costs associated with system calls or interrupts are mitigated in KBstacks by means of *I/O batching* (i.e., by using a separate zero-copy DMA buffer, and sending an interrupt once the whole buffer is full, as opposed to sending one interrupt per packet). For high-speed KBstacks, batching is commonly coupled with *poll-mode drivers*, where interrupts are replaced by periodic polling of the device.

KBstacks are supported by several low-level building blocks like netmap [4], DPDK[5] and PF_RING [7]. Most of them support high-speed I/O through zero-copy, kernel-bypass (at least to some extent, as for netmap [4]), batched I/O and multi-queuing. A more detailed comparison of features provided by a larger number of frameworks is available in [2], while an experimental comparison of DPDK, PF_RING and netmap (for relatively simple tasks) is described in [8]. Worth mentioning are also the eXpress Data Path (XDP) project[6], which embraces similar principles but using a kernel-level approach, and The Open Data Plane (ODP) Project[7], an open-source, cross-platform set of APIs running on several hardware platforms such as x86 servers or networking System-on-Chip (SoC) processors.

Another active research activity is represented by prototypes targeting very specific network functionalities such as IP routing [9], traffic classification [10], name-based forwarding [11]. In spite of the different goals, and the possible use of CPUs only [10], network processors [11] or GPUs [9], a number of commonalities arise. PacketShader [9] is a GPU accelerated software IP router. In terms of low-level functions, it provides kernel bypass and batched I/O, but not zero copy. MTclass [10] is a CPU-only traffic classification engine capable of working at line rate, by employing lock-free multi-threading; at low-level, MTclass uses PacketShader hence inheriting the aforementioned limitations. Prototype in [11] addresses high-speed functions related to Information-centric networking (ICN) architectures, such as name-based forwarding and caching. All these tools scale-up by leveraging RSS queues and lock-free multi-threading at 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 closer in scope to VPP are [1, 2, 5, 12]. VPP shares with Click [1] the goal of building a flexible and fully programmable software router. Whereas the original Click cannot be listed among KBstacks applications (as it requires a custom kernel, and runs in kernel-mode), a number of extensions [2, 5, 12] have brought KBstacks elements into Click, e.g. introducing support for HW multiqueue [5], batching [12], and high-speed processing [2]. Important differences among Click (and variants) and VPP arise in the scheduling of packets in the processing graph (cfr. Sec.III). Finally, the recently proposed MoonRoute [13] heavily uses a multi core scheduling where there are several fast path components distributed to different CPU cores and a single slow path component.

[5]http://dpdk.org

[6]https://www.iovisor.org/technology/xdp

[7]https://www.opendataplane.org/

## III. VPP Architecture

Initially proposed in [3], VPP was recently released as open source software, in the context of the Fast Data IO (FD.io) Linux Foundation project[8]. VPP is a user-space high-speed framework for packet processing, designed to take advantage of general-purpose CPU architectures. VPP can exploit the recent advances in the KBstacks low-level building blocks described above: 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-KBstacks interfaces such as `AF_PACKET` sockets or tap interfaces are also supported.

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 layer-2 and 3 functionalities, which follows from the fact that the VPP technology was included in Cisco router products since about a decade. Furthermore, the release of VPP as open source software fueled the development of novel features, in particular for layer-4 (and above) functionalities, which are currently in an active development phase, and that enable the use of VPP as a framework for data-center and NFV environments, as early introduced. For the sake of the example in the rest of the paper we focus on layer-2 and 3 functionalities.

The VPP main loop follows a "run-to-completion" model. First a batch of packets is polled using a KBstacks 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). A unique fieature of VPP is that its natively executes per-node batch processing to increase the processing troughput.

VPP consists of a set of low-level libraries for realizing custom packet processing applications as well as a set of high-level libraries implementing a specific processing task (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 its processing.

More specifically, VPP allows three sets of nodes: namely *process*, *input*, and *internal* (which can be terminating leaves, i.e, output nodes). Process nodes do not participate in the

[8]https://fd.io

packet forwarding graph, being simply software functions running on the main cores[9] and reacting to timers and 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*[10] in VPP's terminology) can be activated/deactivated on demand at runtime.

VPP architecture adopts all well-known KBstacks techniques discussed in Sec.II, to which it adds a design (Sec.III-A) and coding practices (Sec.III-B) 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.

## A. 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.IV-B). Input nodes produce a vector of work to process; then, the graph 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. Packets may follow different paths within the forwarding graph (i.e., vectors may be different from node to node). While it is outside the scope to provide a full account of all the available nodes, Fig.1 compactly depicts a subset of the full VPP graph (comprising 253 nodes and 1479 edges) and 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`). A drop decision may be taken at every step (`error-drop`).

**Advantages of vectorized processing:** In a classic "run-to-completion" [1, 2] approach, different functions of the graph are applied to the same packet. This generates a significant performance penalty, due to 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 sizable framework overhead tied to the selection of the next node to process in the 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 process the specific feature. This allows to prefetch the data for the $(i+1)$-th packet while the node processes the data of the $i$-th packet.

**Vector processing vs I/O Batching:** In some sense, VPP extends I/O batching to the upper layers of KBstacks 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 [14], FastClick and the pipelines of the DPDK Packet Framework 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 [2]), is 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. Instead, 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. 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-bits pointers to manage the list).

Ultimately, these low-level differences translate into quite diverse performance benefits. VPP's vectorized processing is lightweight and systematic: 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 [2], in "*the forwarding test case [. . . ] the batching couldn't improve the performance*").

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

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

## B. Other 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 $i, \ldots, i+N$ 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. IV verifies the efficiency of this technique, by measuring the IPC achievable enabling or disabling multi-loop for some example VPP nodes. This technique provides significant performance improvements for certain processing nodes (see Sec. IV for details) but it presents two limitations: (i) the multi-loop technique is applied writing C code that is explicitly parallel[11] (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 the Sec. IV the efficiency of this technique by measuring IPC with prefetching enabled vs disabled.

**Function flattening:** In VPP a majority of graph nodes 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).

## IV. Experimental results

This section describes the experimental setup (Sec.IV-A) we use to assess the impact of VPP architectural choices (vector size, Sec.IV-B) and coding practices (multi-loop and data prefetching Sec.IV-C). While this paper reports a limited set of experiments, an extended set of results is available in [15] for the interested reader.

### A. Setup

**Hardware:** Our hardware setup consists in a server with $2\times$ Intel Xeon Processor E52690, each with 12 physical cores running at 2.60 GHz in hyper-threading and 576KB (30MB) L1 (L3) cache. The server is equipped with $2\times$ 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). 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.

**Metrics:** As observed in [4], performance of switching/routing functionalities are dominated by per-packet operations: for a given bandwidth, minimum size packets require more CPU cycles to be processed, and therefore corresponds to the maximum packet dropping rate. In fact, most of the clock cycles are used to process the packet headers, while the payload processing requires very few clock cycles. Therefore, the processing time per byte is worst in case of minimum size packets. To stress the system, we thus measure the VPP packet-level processing rate $R$ for the 64B packets, corresponding to 14.88 Mpps. Of the two NUMA nodes, one is used as Traffic Generator and Sink (TGS), the other as the System Under Test (SUT): the TGS generates via DPDK synthetic traffic that is the input workload to the VPP SUT, which is sent back to the TGS that measures VPP processing rate.

**Scenarios:** To gather results representative of different network operations, we consider different input workloads (where the L2/L3 addresses are either static or vary in a round-robin or uniformly random fashion) and processing tasks (such as pure IO, Ethernet switching and IPv4 and IPv6 forwarding). Due to space limitations, we refer the reader to an extended technical report [15] for details and, in line with the push toward research reproducibility, we make all scripts available[12].

### B. Vector size

The VPP input node works in polling mode, processing a full batch of packets from the NIC: during this time frame, which depends on the duration of the packet processing tasks, further packets arrive at the NIC. The size of the actually processed vector cannot therefore be directly controlled, though it is possible to cap its maximum size. In the default VPP 17.04 branch, the `VLIB_FRAME_SIZE` is set to 256 by default, but it can be tuned at compile time (to a minimum of 4 packets, due to quad-loop operations in some nodes).

Fig.2 presents a set of metrics gathered from three main use cases, namely (i) *Cross-connect (XC)* case, (ii) a *IP longest-prefix-match* forwarding and (iii) a *mixed traffic (MIX)* in which the incoming traffic activates L2, IPv4 and IPv6 nodes. In the XC case, packets are purely moved from the SUT input to the SUT output port without processing. In the IP case, a longest-prefix matching lookup is performed from a FIB comprising over 130,000 entries to select the output interface (in this setup, the one towards the TGS). In the MIX case, 1/3 of the traffic is forwarded at L2, while the remaining 1/3 IPv4 and 1/3 IPv6 traffic is managed by the IPv4 and IPv6 nodes.

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

[12] http://newnet.telecom-paristech.fr/index.php/vpp-bench

The plots in Fig.2 depict a set of key performance indicators (y-axis) as a function of the vector size (x-axis). Experiments are repeated 10 times, so that in addition to the average of the metric of interest, we additionally report the minimum and maximum values[13] over the repetitions.

The per-core packet processing rate in Mpps is reported in Fig.2-(a): 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 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 (see [15] for a more detailed comparison with DPDK `l3fwd`). We also remark that, that while VPP is able to sustain the 14.88Mpps line rate for XC and IP on on a single core for different architectures [15], however we prefer to report *conservative results* in this paper: while quantitative results may vary, the observations we gather here are qualitatively valid to a larger extent.

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. Measuring the per-packet latency of this operational point with an input rate set to 99% of the lossless throughput (i.e., close to overload), we observe an average latency of $10.7\mu s, 16.7\mu s$ and $35.5\mu s$ for the XC, IP and MIX scenario respectively. Furthermore, 99-th and higher percentiles of the latency distribution are of the same order of magnitude of the mean [15], which testifies predictable performance.

Fig. 2-(b) shows the average number of instructions per packet with respect to the 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 in 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. 2-(c) 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. III-B. As we can see, 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. 2-(d) 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. The number of misses in the instruction cache is relatively small in modern CPU. In fact, we observe that the percentage of cache miss is between 0.3% (when the max vector size is 4 ) and 0.05% (when the max vector size is 256 or above) in our experiments. An L1 instruction-cache miss requires to access to 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.

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

### C. 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). Fig. 3 reports the number of cycles per packet obtained by enabling and disabling the quad-loop and pre-fetching options, for IPv4 and IPv6 processing. While the difference between IPv4 and IPv6 is ties to the data structures used for lookups (tries for v4 and [15] hash-tables for v6, cfr. [15]), the figure clearly shows that the quad-loop technique sizeably reduces the number of clock cycles both when prefetching is enabled (25% for IPv4) or disabled (16%).

Additionally, we stress that these gains are consistent across a wide range of input workloads: notably, the largest observed discrepancy for XC, IP and MIX scenarios considering random uniform, deterministic or round-robin spatial variation of input traffic tops to just 8% in the MIX case with very large FIBs (with 138K elements) and is otherwise negligible [15].

## V. Conclusion

This paper introduces Vector Packet Processing (VPP), a feature-rich high-performance software router, by illustrating its main novelty from an architectural viewpoint, as well as briefly assessing its performance. With respect to known techniques for high-speed software processing, VPP extends *batching* from pure I/O (which merely reduces interrupt pressure) to *complete graph processing* (which amortizes the software framework overhead). 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).

## 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).

## References

[1] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. Kaashoek. The Click Modular Router. Operating Systems Review, 34(5):217231, 1999.

[2] T. Barbette, C. Soldani, and L. Mathy. Fast userspace packet processing. In ACM/IEEE ANCS, 2015.

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

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

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

[6] D. Cerovi, V. Del Piccolo, A. Amamou, K. Haddadou, and G. Pujolle. Fast packet processing: A survey. IEEE Communications Surveys & Tutorials, 2018.

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

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

[9] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In ACM SIGCOMM, 2010.

[10] P. M. Santiago del Ro, 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.

[11] D. Perino, M. Varvello, L. Linguaglossa, R. P. Laufer, and R. Boislaigue. Caesar: a content router for high-speed forwarding on content names. In ACM/IEEE ANCS, 2014.

[12] J. Kim, S. Huh, K. Jang, K. Park, and S. Moon. The power of batching in the click modular router. In Asia-Pacific Workshop on Systems, 2012.

[13] S. Gallenmller, P. Emmerich, R. Schnberger, D. Raumer, and G. Carle. Building fast but flexible software routers. In Proceedings of the Symposium on Architectures for Networking and Communications Systems, pages 101102. IEEE Press, 2017.

[14] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen. Raising the bar for using gpus in software packet processing. In USENIX NSDI, 2015.

[15] L. Linguaglossa et al. High-speed Software Data Plane via Vectorized Packet Processing (Extended Version). In Tech.Rep. avail. at https://newnet.telecom-paristech.fr/ index.php/ vpp-bench/, 2017.

## Biographies

**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.

**Leonardo Linguaglossa** is a post-doctoral researcher at Tlcom ParisTech (France). He received his master degree in telecommunication engineering at University of Catania (Italy) in 2012. He pursued a Ph.D. in Computer Networks in 2016 through a joint doctoral program with Alcatel-Lucent Bell Labs (nowadays Nokia), INRIA and University Paris 7. Leonardo's research interests focus on architecture, design and prototyping of systems for high-speed software packet processing, future Internet architecture and SDN.

**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.

**Salvatore Pontarelli** received a master degree in electronic engineering at University of Bologna and the PhD degree in Microelectronics and Telecommunications from the University of Rome Tor Vergata. Currently, he works as Senior Researcher at CNIT (Italian National Inter-University Consortium for Telecommunications), in the research unit of University of Rome Tor Vergata. His research interests include hash based structures for networking applications, use of FPGA for high speed network monitoring and hardware design of software defined network devices.

**Dario Rossi** received his MSc and PhD degrees from Politecnico di Torino in 2001 and 2005 respectively, and his HDR degree from Universit Pierre et Marie Curie (UPMC) in 2010. He is currently a Professor at Telecom ParisTech and Ecole Polytechnique and is the holder of Cisco's Chair NewNet@Paris. He has coauthored 9 patents and over 150 papers in leading conferences and journals, that received 8 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.