# High-speed Software Data Plane via Vectorized 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

**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 before moving to the next node. This design choice enables several code optimizations that greatly improve the achievable processing troughput. In this report we introduce the main VPP concepts and architecture, and presents several experiments that show the impact of design choices (such as batch packet processing) on VPP performances.

## Contents

# 1 Introduction

Sofware 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.
One of the seminal attempts to circumvent the lack of flexibility in network equipment is represented by the Click modular router [23]: 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 approach placed most of the high-speed functionalities as close as possile to the hardware, which were thus implemented as a separate kernel module. However, whereas a kernel module can directly access an hardware device, user-space applications need to explicitly perform system-calls and use the kernel as an intermediate step, thus adding overhead to the execution of the model.

More generally, recent improvements in transmission speed and network cards cababilities, have led to a situation where a general-purpose kernel stack is far too slow for processing packets at wire-speed among multiple interfaces [10]. As such, a tendency has emerged to implement high-speed stacks *bypassing operating system kernels* (KBnets) and bringing the hardware abstraction directly to the user-space, with a number of efforts (cfr. Sec2) targeting either low-level building blocks for kernel bypass like netmap [28] and the Intel Data Plane Development Kit (DPDK), very specific functions [19, 29, 26, 25] or full-blown modular frameworks for packet processing [16, 21, 10, 27, 11].

In this technical report, we describe a new framework for building high-speed data plane functionalities in software, namely Vector Packet Processor (VPP), that has recently been released as the Linux Foundation project "Fast Data IO"

(FD.io) [6]. In a nutshell, VPP combines 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 constrast with existing batch-processing techniques, that are merely used to either reduce interrupt pressure (e.g., by lower-building blocks such as [28, 5, 15]) or are non-systematic and pay the price of a high-level implementation (e.g., batch processing advantages are offset by the overhead of linked-lists in FastClick [10]).

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 (Sec.4). We next discuss our findings and report on future work (Sec.5). Finally, we provide brief instructions to replicate our work (App.A) and let our scripts available at [1]).

# 2 Background

This section overviews the state-of-the art 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 in a top-down fashion with respect to Fig. 1 that illustrates the general COTS architecture we consider in this work.

**Lock-free Multi-threading (LFMT).** Processing applications in user-space mandates to leverage the current tendency to multi-core COTS, which equates to embracing 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 speedup, it is imperative to avoid performance issues tied to the use of mutex and so to achieve *lock-free operation*. Lock-free parallelism in KBnets is is tightly coupled with the availability[1] of hardware queues (discussed next), to let threads operate on independent traffic subsets.

[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.] | CC&L | CB | LFMT | IOB | Z-C | RSS | Main Purpose |
|---|---|---|---|---|---|---|---|
| DPDK[5] | | | ✓ | ✓ | ✓ | ✓ | Low-level IO |
| netmap[28] | | | ✓ | ✓ | ✓ | ✓ | |
| PacketShader[19] | | | ✓ | ✓ | | ✓ | Routing |
| MTclass[29] | | | ✓ | ✓ | | ✓ | Classification |
| Augustus[22] | | | ✓ | ✓ | ✓ | ✓ | Name-based fwd |
| HCS[25] | | | ✓ | ✓ | ✓ | ✓ | Caching |
| Click[23] | | | | | | | Modularity |
| RouteBricks[16] | | | ✓ | | | ✓ | |
| DoubleClick[21] | | | ✓ | ✓ | ✓ | ✓ | |
| FastClick[10] | | ✓ * | ✓ | ✓ | ✓ | ✓ | |
| VPP (this work) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |

**Low-level parallelism.** 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 [7]. 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. 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 systematically leverage systematic low-level parallelism through its design and coding practices.

**Cache Coherence & Locality (CC&L).** A major bottleneck for software architecture is nowadays represented by memory access [12]. 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 [2] is reported in the right of Fig. 1). In general, L1 cache (divided into shared 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

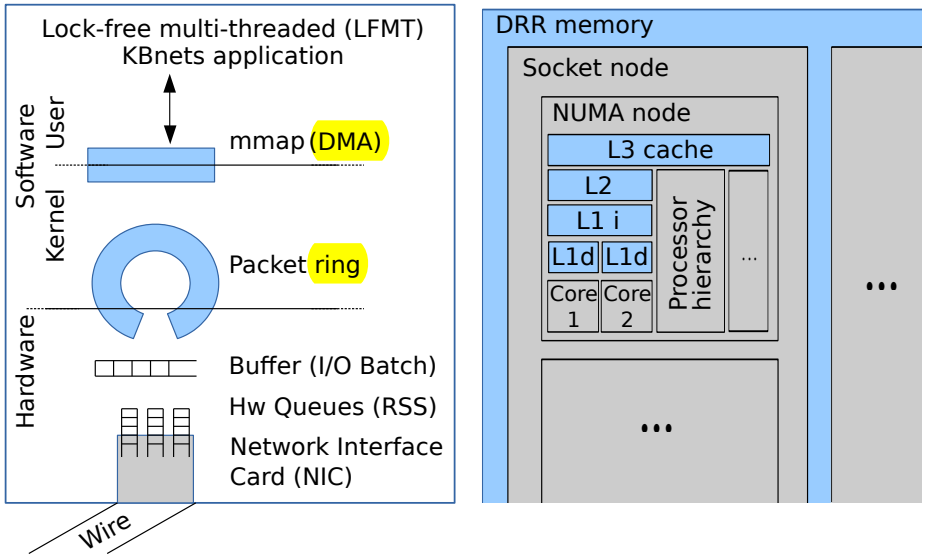


Figure 1: The architecture of a typical COTS software router with special attention the NIC (left) and the memory hierarchy (right).

of the cache hierarchy, a cache miss occurs forcing access to the higher levels, slowing the overall application.

**Zero-Copy (DMA).** When the Network Interface Card (NIC) has some packets available, it writes it to a reserved memory region, which is shared between the network interface and the operating system. Early KBnets approached 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 (DMA) to the memory region used by the NIC. Notice that zero-copy is sustainable only in case the packet consumer (i.e., the application) is faster than the packet arrival rate (occasional slowdown may need application-level buffer or drops).

**RSS Queues.** Modern NICs support multiple rx/tx hardware queues, and Receive-side Scaling (RSS) [20] 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-threaded 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 [31]), but recent NIC support more involved mathching (e.g., up to 4096-bit hash filtering, which framework needs to make accessible in userland).

**I/O batching.** 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 DMA and append 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 batched processing from pure I/O (which reduces interrupt overhead) to complete graph processing (to exploit for cache coherence and locality).

## 2.2 KBnets frameworks

We can individuate three branches of KBnet frameworks, depeding on whether they target lower-level building blocks [28, 5, 15], very specific functions [19, 29, 26, 25] or full-blown modular frameworks[16, 21, 10, 27, 11].

**Low-level building blocks.** This class of work has received quite a lot of attention, with valuable frameworks such as Netmap [28], DPDK [5] and PF_RING [15]. In terms of features, most of them support high-speed I/O through zero-copy, kernel-bypass, batched I/O and multi-queuing, though subtle[2] differences may still arise still among frameworks [10] and their performance [18]. A more detailed comparison of features available in a larger number of low-level frameworks is available at [10], whereas an experimental comparison of DPDK, PF_RING and netmap (for relatively simple tasks) is available at [18]. Worth mentioning are also eXpress Data Path (XDP)[8], which embraces similar principles but in a kernel-level approach, and Open Data Plane (ODP)[9] aimed at providing interoperability among proprietary optimized vendor-specific hardware blocks and low-level software libraries. As we shall see, VPP can leverage several of these useful low-level building blocks.

**Purpose-specific prototype.** Another class of work is represented by full-blown prototype that are capable of a very specific and restrained set of capabilities such as IP routing [19], traffic classification [29], name-based forwarding [26] or transparent hierarchical caching [25]. In spite of the different goals, and the possible use of network processors [26] or GPUs [19], a number of commonality arise. PacketShader [19] is a GPU accelerated software IP router. In terms of low-level function it provides kernel bypass and batched I/O, but not zero copy. MTclass [29] 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 [26, 22] and [25] address high-speed solution of two specific functions related to Information-centric network (ICN) architectures, namely name-based forwarding and caching ([26] employs a network-processor

[2] A limitation of netmap that it does not allow to directly access the NIC's registers [17]

whereas [22, 25] use DPDK). In all these cases, multi-thread lock-free programming enabled by RSS queues is the key to scale up operations in user-space. Differently from these work VPP aims for generality, feature richness and consistent performance irrespectively of the specific purpose.

**Full-blown modular frameworks.** Full-blown modular framework are closer in scope to VPP. Letting aside relevant but proprietary stacks [3], work such as [27, 11, 16, 21, 10] is worth citing. In more details, Arrakis [27] and IX [11] are complete environments for building network prototype, including I/O stack and software processing model. Arrakis main goal is to push further kernel bypass 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 [23], which shares the goal of building a flexible and fully programmable software router. Whereas the original Click cannot be listed among KBnet applications (as it requires a custom kernel, and runs in kernel-mode and it is therefore not suited for high-speed processing), however a number of extensions have over the years brought elements of KBnets into Click. Especially, RouteBricks [16], DoubleClick [21], FastClick [10] all support the kernel version of Click, introducing support for HW multiqueue [16], batching [21], and high-speed processing [10], possibly obtained through a dedicated network processor [24]. Interesting differences among Click and VPP are discussed further in Sec.3.

# 3 VPP Architecture

**Context.** Initially proposed in [14], VPP technology was recently released as open source software, in the context of the FD.io linux foundation project [6].

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 (with however a significant performance loss, except maybe with the ongoing kernel-level XDP module).

**At a glance.** VPP is written in C, and its sources are organized in two main groups of components: 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) as *plugins*. VPP main code

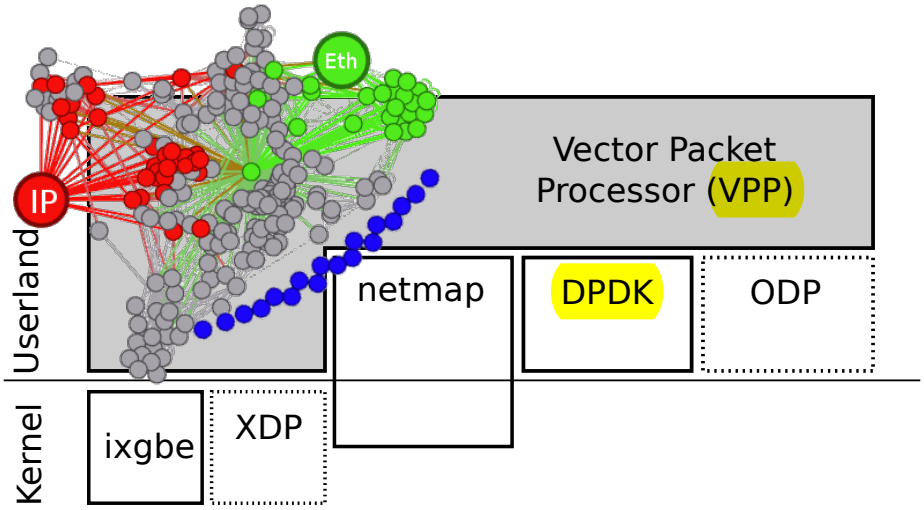


Figure 2: VPP scope and processing tree. The nodes we use later in the experiments (as well as their neighbors) are highlighted in red (IP) and green (Eth). Process nodes are depicted in blue.

and plugins altogether form a *forwarding graph*, which describes the possible paths a packet can follow during its processing. While it is outside of the scope to provide a full account of all the available nodes[4], for the sake of completeness Fig.2 depicts the VPP graph (comprising 253 nodes and 1479 edges), highlighting the nodes we will be using in the experimental section.

In more details, VPP allows three set of nodes: namely *input*, *process*, and *internal* (which can happen to be terminating leafs, i.e, output nodes). For some nodes, a set of finer-grained processing tasks (aka *features*[3] in VPP lingo) can be activated/deactivated on demand at runtime. Process nodes (blue nodes in Fig.2) are not participating in the packet forwarding graph, they are simply software functions running on the main core and reacting on timers and events[4].

Finally, VPP adopts all well-known KBnets techniques early illustrated. Additionally, VPP is NUMA aware (it prefers buffers on the local NUMA node) as well as Cache-aware: its design (Sec.3.1) and coding practices (Sec.3.2) are explicitly tailored to exploit cache coherence and locality.

## 3.1 Vectorized processing

The main novelty of VPP is to offer a systematic way to efficiently process packets in a "vectorized" fashion, which as we shall see in Sec.4.2, provides sizeable performance benefits. Notably, in VPP packets are batched in a vector:

[3]These are not functions and do not bring the overhead associated to function calls.

[4]VPP features its own internal implementation of cooperative multitasking[13], which allows running of multiple process nodes on the main core.

instead of letting each packet traverse the whole graph, in VPP each node processes all packets in its vector. In a nutshell, 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).

**Advantages of vectorized processing.** In a classic "run-to-completion"[23, 10] approach, different functions of the graph are applied to the same packet, generating a significative performance penalty. This penalty is due to several factors. (i) The instruction cache miss rate increases when a different function have to be loaded and the instruction cache is already full. (ii) There is a significative 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 pre-fetching 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 feature 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 ito 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 size.** The underlying assumption of vector processing is that subsequent packets will require similar processing: the first packet in the vector processed by a given node warms up the node dispatch function in the L1 I-cache, whereas subsequent packets profit of the L1-I hit. 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 in 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, L1 I-cache hit increases processing efficiency by amortizing fixed costs over a larger number of elements. In turn, the per-packet processing time decreases, and so the size of the subsequent input vector. Overall, this 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 the KBnets processing. However, if the goal of batching I/O operations is to reduce the interrupt frequency by the batch size, the goal of vectorized processing is to instead leverage coherence and locality

of the L1 instruction (L1-I) cache. These two goals are complementary and interoperable.

**Vector processing vs Compute Batching.** It is worth pointing out that FastClick does offer "Compute Batching" (see Sec 5.7 in [10]), that is however only similar from a very high-level view, while several fundamental differences arise at a closer look. First, API of any VPP nodes are designed to systematically process vectors of packets. This not only allows very efficient APIs, but also leaves room for optimization (e.g., see multi-loop). In contrast, nodes in FastClick implicitly process packets, and only specific nodes have been augmented to *also* accept batched input.

This difference has profound implications. Indeed, per-vector processing is a fundamental primitive in VPP: at low level, vectors are pre-allocated arrays residing in contiguous portions of memory. Vectors are never freed, but efficiently managed in re-use lists by VPP. Additionally, vectors elements are 32-bit integers that are mapped to a pointer 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 one cycle). In FastClick instead, batches are constructed by using the *simple linked list* implementation available in Click, with a higher memory occupancy (inherently less cacheable) and a higher overhead (adding further 64-bits pointers to manage the list).

Ultimately, these low-level differences translate into quite diverse performance benefits. In VPP, vectorized processing is lighweight and systematic: in turn, processing vectors of packets benefits of hits in the L1 I-cache, which we observe to significantly speed up the treatment of individual packets. In contrast, under FastClick the opportunistic batching/splitting overhead, coupled to linked list management yields to limited achievable benefits in some cases (and none in others, e.g., quoting[10], in "*the forwarding test case [...] the batching couldn't improve the performance*") – which we argue to be tied to the loss of L1 instruction cache hits.

## 3.2 Further optimization

VPP uses additional tricks to exploit all lower-level hardware assistance in user-space processing, as follows.

**Multi-loop.** We refer to *multi-loop* as a coding practice where any function is written to explicitly handle $N$ subsequent packets with identical treatment in parallel: since computations on packets $i, \ldots, i+N$ are typically independent of each other, very fined-grained parallel execution can be exploited in such case, (i) letting CPU pipelines to be continuously full. Additionally, under multi-loop (ii) a fetching delay caused by a cache miss is shared by $N$ packets instead of a single one.

In practice, most of the CPU pipeline benefits are expected already for small $N = 2$ (dual-loop) and, due to registers pressure, only very simple functions perform better when $N = 4$ (quad-loop). Finally, under the assumption that

not only temporally close packets need the same processing (L1-i hit) but that the same treatment is to be applied to packets that are spatially close in the memory buffer (L1-d hit when prefetched), it may be adviseable to proactively *prefetch* the next $N$ packets for multi-loop treatment.

**Branch-prediction.** This practice is based on the assumption that most treatment will follow a "Pareto law" that we loosely refer here as the fact that the majority of packets w ill require very similar processing and thus follow the same path in the tree. VPP thus 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 Pareto law), a pipeline reset is avoided saving cycles (possibly more than 10 in practice). In case 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 misprediction are rarely due to Pareto law). While Branch Predictor (BP) is a very powerful component in 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 small part of code).

**Function flattening.** To avoid expensive functions calls, VPP favors *flattening*, i.e., majority of graph nodes make use of inline functions, to avoid the cost of reshuffling registers to comply with ABI calling convention and to avoid 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 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 Experimental results

This section describes the experimental setup (Sec.4.1) we use to assess the impact of VPP architectural choices (vector size, Sec.4.2) and coding practices (multi-loop, Sec.4.3), as well as of exogeneous factors (input workload, Sec.4.4).

## 4.1 Setup

We start by providing a quick overview of our hardware and software setup, delegating to the Appendix a more detailed descritpion, as well as instructions to reproduce our experiments that we make available at [1].

**Hardware.** Our hardware setup comprises a COTS equipment with 2× Intel Xeon Processor E52690, each with 12 physical cores running at 2.60 GHz in hyper-threading and 576KB (30M) L1 (L3) cache. The server is equipped with 2× Intel X520 NICs, capable of dual-port 10Gbps full duplex link, that are directly connected with SFP+ interfaces. Since we are interested into gathering insights on the performance gains tied to the different aspects of the whole VPP architecture, as opposite to gathering the raw absolute performance of a specific system, we limitedly study *per-core* performance – intuitively, provided a good LFMT application design, the overall system perfomance can be gathered by aggregating performance of individual cores.

**Metrics.** As observed in [28], I/O performance are dominated by per-packet operations: i.e., it is slightly more expensive to send a 1.5KB packet than a 64B one. To stress the system, we thus measure the VPP packet-level processing rate $\tilde{R}$ for the smallest 64B packet size. 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 who can measure the VPP processing rate.

**Scenarios.** To gather results representative of different network operations, we consider different input workloads (where the L2/L3 address are either static or vary in a round-robin vs uniformly random fashion) and processing tasks (such as pure IO, Ethernet switching and IP forwarding). In line with the push toward research reproducibility, we make all scripts available at [1].

## 4.2 Vector size

VPP input node works in polling mode, processing a full batch of packets from the NIC: during this timeframe, which depends on the duration of the packet processing tasks itself, further packets arrive at the NIC. As such, while the size of the actually processed vector cannot be directly controlled, however it is possible to cap its maximum size. In the default VPP 17.2 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): intuitively, increasing the frame size increases the L1-instruction cache hit benefits, but also increases the average per-packet delay.

Fig.3 depicts the per-core packet processing rate in Mpps on the y-axis, as a function of the vector size on the x-axis. Two cases are shown in the picture, namely (i) *Cross-connect (XC)* case and (ii) a *IP longest-prefix-match*

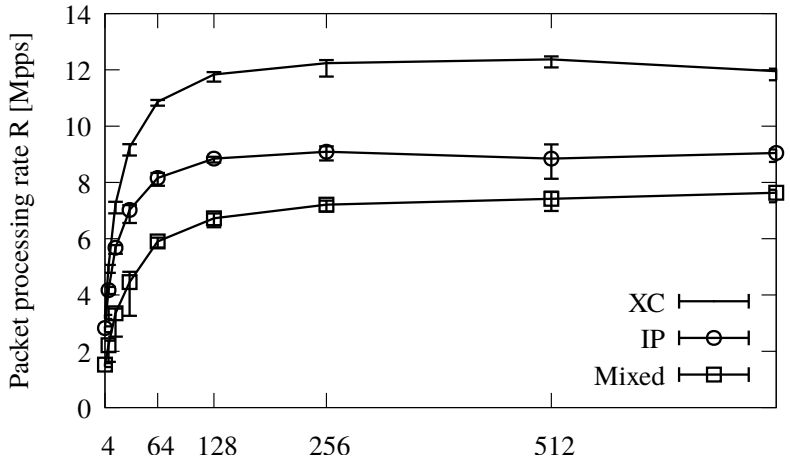


Figure 3: Packet processing rate as a function of the maximum vector size (10Gbps traffic rate on a single-core)

forwarding. In the XC case, packets are purely moved from the input to the output port (toward the TGS) without processing. In the IP case, a longest-prefix matching lookup is performed from a FIB comprising $2^{17}$ IP/17 ranges (i.e., about 130K entries covering the whole IPv4 space) to select the output interface (in this setup, the one toward the TGS).

It can be seen that from the XC case that packet processing rate increases linearly with the vector size, up to a saturation point where increasing the size further does not bring noticeable benefits – rather, a slight penalty arises when the vector becomes too large, as the overhead of managing a larger memory portion. A similar qualitative behavior is obtained for the other processing tasks: interestingly, for the IP case, a single core is able to forward over 10 millions packets per second, for a fairly large FIB.

For both cases, the knee in the curve at about 256 packets-per-vector corresponds to the sweet-spot of the aforementioned tradeoff, which maximizes the effect of the L1-I instruction cache hit, bounding the *maximum* delay due to vectorized processing to about $13\mu s$ (64B packets) – $300\mu s$ (1.5KB packets).

## 4.3 Multi-loop

Fixing `VLIB_FRAME_SIZE` to 256, we next consider the impact of multi-loop programming practice. Since multi-loop aims at amortizing per-packet processing by parallelizing instructions over multiple independent packets, it does not apply to the cross-connect XC case. For this scenario, we therefore consider two different processing tasks, namely (i) exact-match (Eth switching) vs (ii) longest prefix match (IP forwarding).

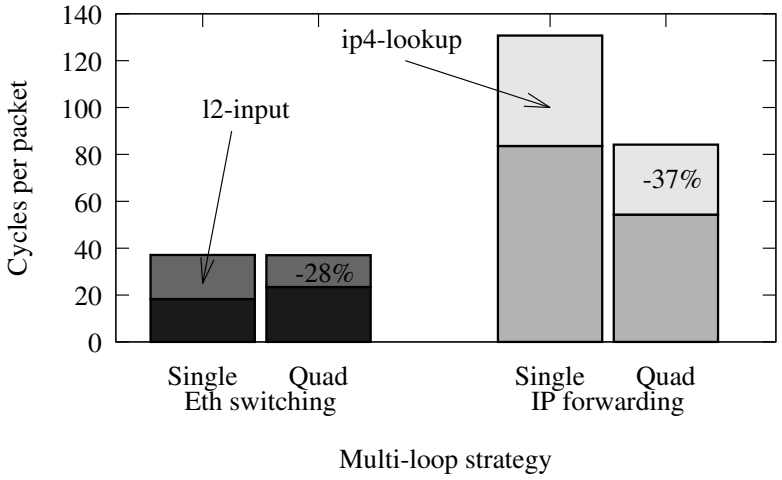


Figure 4: Impact of multi-loop programming practice on the per-packet processing cost

We instrument the code to selectively enable only specific multi-loop operations (for the sake of brevity, we only consider quad-loop implementation), that we contrast with simple single-loop ones in Fig.4. Rather than reporting aggregate processing rates, we report the average packet processing duration expressed in CPU cycles. The picture not only reports the per-packet processing cost for the whole IP forwarding vs Eth switching operations, but also highlights the biggest contributor node (`ip4-lookup` and `l2-input` respectively) in the chain, annotating the relative reduction. It can be seen that gains are sizeable, which especially holds true for IP (both the specific `ip4-lookup` node, as well as the overall chain decrease by about 37%).

## 4.4 Input workload

Finally, we next turn our attention to the impact of different input workloads on the processing rate.

As before, we consider the cross-connect XC case, and the IP forwarding 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

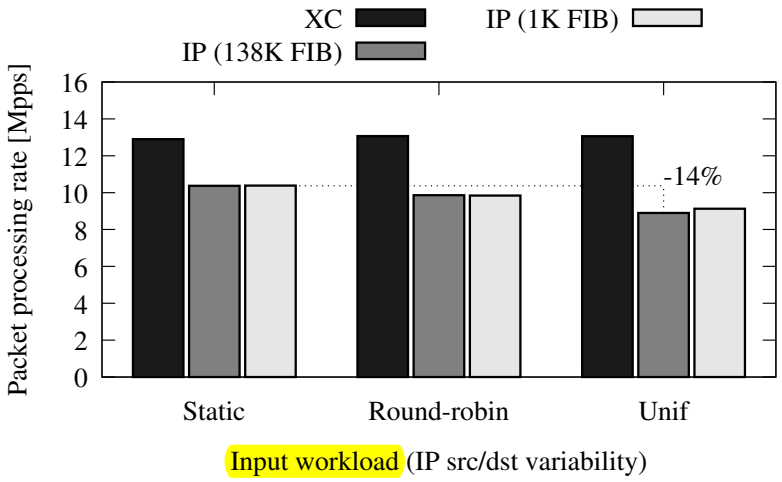


Figure 5: Variation of the packet processing rate as a function of the input workload process.

leverage L1 D-cache hits to some extent (i.e., which 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[5] to the worst-case.

Results, shown in Fig.5, 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 15%, 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.

## 4.5 Latency

The batch processing may impact the latency that packets experience in a VPP router, due to the fact that the forwarding operation occurs only after a full graph traversal. Table 2 shows the latency, measured in microseconds, for our different scenarios (cross-connect, IP only and mixed traffic). We report the minimum, the maximum and the average values, as well as the three quartiles and the standard deviation.

[5]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.

Table 2: Latency for different scenarios (in $\mu s$)

| Scenario | Rate | Min | Avg | Std. dev. | Median | 99th |
|---|---|---|---|---|---|---|
| XC | 0.99 NDR | 4.4 | 10.7 | 7.7 | 10.2 | 19.0 |
| | 0.5 NDR | 4.4 | 7.1 | 4.3 | 6.7 | 10.7 |
| | 0.1 NDR | 4.4 | 4.8 | 4.6 | 4.7 | 5.6 |
| IP | 0.99 NDR | 4.5 | 16.6 | 14.4 | 12.7 | 31.9 |
| | 0.5 NDR | 4.5 | 6.4 | 4.0 | 6.3 | 7.8 |
| | 0.1 NDR | 4.4 | 4.8 | 3.6 | 4.7 | 7.0 |
| Mixed | 0.99 NDR | 22.4 | 35.5 | 14.8 | 35.0 | 48.8 |
| | 0.5 NDR | 6.8 | 9.7 | 6.2 | 9.6 | 12.9 |
| | 0.1 NDR | 4.7 | 5.3 | 4.5 | 5.0 | 7.1 |

We observe that, as expected, the cross-connect scenario shows the lowest values of latency, being on average $10.8\mu s$. The average, the minimum and all the quartiles are close to each other, and the standard deviation is of 6.1 $\mu s$ (the only exception is the maximum, which is an outlier); the histogram ressembles a normal distribution centered in the average value. 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 is not traversed).

Latency increases when VPP starts activating the graph traversal, as shown by Table 2. The average latency is $258.0\mu s$ and 258.8 $\mu s$ for the IP and the Mixed case respectively. Interestingly, there are no huge differences in the statistics of these two scenarios. We observe only a greater value for the average and the maximum in the Mixed case. We claim that this can explained by the fact that once the forwarding graph is traversed, the presence of additional nodes only slightly impact the overall processing. For the IP and Mixed scenario, values are not normally distributed around the average: in fact, all quartiles are not located close to the average, and, although not shown in the Table, the 99th percentile is close to the maximum value. The histogram is similar to a uniform distribution between the minimum and the maximum value. The variation in the measurements is related to the fact that the first packets to be received observe a greater latency than the latest, since they stayed in a vector and traversed the graph for longer.

# 5 Conclusion

This paper introduces Vector Packet Processing (VPP), both illustrating its main novelty from an architectural viewpoint, as well as briefly assessing its performance. 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 furthermore systematic fashion). Indeed, even though extension of batching to processing is attempted in FastClick [10], 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. This lower-level implementation is crucial as, to the best of our knowledge, VPP is among (if not the) first KBnet approach to systematically leverage hits of the L1 I-cache, with a design that reinforces as much as possible the cache temporal coherence and spatial locality. We finally introduce further aspects worth studing as future work.

**Architectural improvements.** While this work shows the benefits of vectorized packet processing, we believe that there should be room for improvement by leveraging orthogonal optimizations in the packet aquisition design. Indeed, batching relieves interrupt ovehead, yet polling consumes CPU cycles that could be useful for other operations. However, the typical task of the input poll loop is to move DMA pointers, a non-CPU intensive operation. Hence it should be feasible to wake up quasi-periodically with either a $\mu$sleep vs `SCHED_DEADLINE` techniques, setting the periodic timer low enough to still guarantee loss-free operation[30]. Under this angle, an interesting tradeoff arises: POSIX $\mu$sleep is a user-space technique guaranteed to sleep at least $x \geq \delta$, wherea `SCHED_DEADLINE` is a kernel-level scheduling policy that would guarantee the input threads to be woken at least once in a $x \leq \delta$ interval. Assessing the potential benefits of these techniques for VPP, and for KBnets in general, is an interesting research question requiring further work.

**Comparison to other frameworks.** Admittedly, this work only scratches the surface of VPP performance evaluation. This is not only since numerous network functions are available in VPP, but also and especially because a comparison with other frameworks is still missing. Whereas one could be tempted to directly compare raw performance across studies, however we stress that such comparison has to be carefully conducted, as performance may vary due to numerous factors (e.g., clock speed, L1 cache size, input workload, etc.) that are difficult to extrapolate and compare across different studies. For instance, the setup in FastClick [10], the closest work to VPP that thus makes an obvious candidate for comparison, differs in the CPU (i7 vs Xeon in our work), clock speed (3.4Ghz vs 2.6Ghz), memory speed (1.6GHz vs 1.3GHz) and cache size (12MB vs 30MB). Additionally, even in case that the same HW setup would be used, it is to be noted that the specific implementation of the same SW function varies, so that is unclear to what extent the measured performance would relate to the design of the whole architecture (e.g., vector processing) as opposite to the implementation of a specific function in that architecture (e.g., L3 lookup). Similarly, aspects such as the input workload may further contribute to blur the picture. Reproducibility is needed to advance in this goal, for which we share all the relevant informations at [1].

## 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] https://newnet.telecom-paristech.fr/index.php/vpp-bench.

[2] https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

[3] http://www.6wind.com/products/.

[4] https://wiki.fd.io/view/VPP/Features.

[5] Data plane development kit. http://dpdk.org.

[6] Fast Data Project (FD.io). https://fd.io.

[7] Intel Haswell Micro-architecture Reference Manual.

[8] The eXpress Data Path (XDP) project. https://www.iovisor.org/technology/xdp.

[9] The Open Data Plane (ODP) Project. https://www.opendataplane.org/.

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

[11] 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.

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

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

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

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

[16] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *SIGOPS*, 2009.

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

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

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

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

[21] 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*. ACM, 2012.

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

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

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

[25] 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.

[26] 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.

[27] Simon Peter, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System as Control Plane. *;Login:*, 38(4):44–47, 2013.

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

[29] 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.

[30] Martino Trevisan, Alessandro Finamore, Marco Mellia, Maurizio Munafo, and Dario Rossi. Traffic Analysis with Off-the-Shelf Hardware: Challenges and Lessons Learned. *IEEE Communication Magazine*, 2017.

[31] Shinae Woo and KyoungSoo Park. Scalable tcp session monitoring with symmetric rss. In *KAIST Tech. Rep.*, 2012.

# A Experimental setup

In this section we discuss details of the testbed used for our experiments.

## A.1 Testbed details

Our equipment consists of two Intel X520 line cards, each one with two SFP+ 10Gbps interfaces, for an overall sustainable rate of 40Gbps. We reproduce an experimental environment as suggested in the RFC2544 by connecting one device under test (DUT) to one measurment device equipped with a packet transmitter and a packet receiver. Performance is evaluated at the endpoints of the measurement device.

In order to reproduce this scenario, we consider one line card running the vpp software as the device under test, while the second line card includes a DPDK traffic generator (TG) and the scripting software to take the measurements. The port 0 (1 respectively) of the TG is connected to the port 0 (1 respectively) of the DUT.

**TG.** The TG transmits packets on the port 1, and receives the processed packets on the port 0; obviously, the configuration is mirrored in the DUT. We can then evaluate the rate at which VPP operates on the DUT by observing the differences between TX and RX on the TG side.

The traffic generator runs on 8 cores on the socket 0. It may transmit three typology of traffic: a static traffic, implemented as a single pattern of the IP destinations; a round-robin traffic, where the destination address varys in the least significant bits, and finally a uniformly random traffic, where requests are distributed over the all 32bit IPv4 space.

**DUT.** We put our DUT under stress condition: it runs on a single core and with a single thread, handling both I/O and packet processing, for packets of the smallest size (64B).

Each experiment consists in sending a traffic type on the port 1 for 60 seconds. The DUT then, controlled by the VPP software, can process packets through the VPP forwarding graph, and send them to the output interface, towards the TG. After 60 seconds, we can mesure:

- The forwarding rate at minimum size
- The loss rate corresponding to NDR (non-drop rate) and PDR (partial-drop rate)

We analyze the differences on these measures. The fwd rate describes the capability of the DUT under stress conditions: it represents the maximum rate at which VPP+DUT may operate. In this experiment the TG sends 64b packets at 10Gbps and we may observe losses.

The NDR represents the steady-state for traffic measurements, and it is defined as the rate at which the DUT may operate without losses. The PDR is

a measurement which sits between the NDR and the FWD rate, and it represents the maximum rate at which the DUT may work with controlled losses. NDR and PDR are found with a dichotomic procedure reacting on the measured loss rate in a 60 second experiment. In this technical report we restrict our attention to the forwarding rate for minimum size packets.

## A.2 Script code and results

We now point the reader to the workflow and scripts used to *reproduce* our experiments. Notice that if you have a different hardware (i.e., CPU, CPU speed, L1 cache size, DRAM speed, etc.) you will be able to *repeat* our experiments, although your results' mileage may vary.

The scripts are available at GitHub, under the vpp-bench project [1]. We refer the reader to the GitHub page for up-to-date description and details.