Benchmarking of VPP

Arijit Pramanik

RnD Project

Abstract

Vector Packet Processing Technology, currently released under FD.io is a framework for high-speed packet processing in **user-space**. VPP is capable of implementing a full network stack, including L2, L3 functionalities. Currently, it runs on top of DPDK used as input/output nodes in addition to VPP packet processing.

1 Introduction

VPP follows a **batch-wise packet processing paradigm**. *Scalar packet processing* involves each packet traversing the whole forwarding graph, but in VPP, each node processes all the packets in a batch, which provides great performance benefits. Not all packets follow the same path in a forwarding graph and hence, the vectors will be different form node to node.

The input nodes produce a vector of packets to process, the graph node dispatcher pushes the vector through the different processing nodes in the directed graph, subdividing and redistributing the packets as required, until the original vector has been completely processed. 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 a specific processing task. The main core and user-defined plugins, which define additional functionalities, form a *forwarding graph*, illustrated in Fig. 1 describing the possible paths a packet can follow during its processing. A generalized processing path is shown in Fig. 2

The three main goals of VPP are :

- minimize the instruction cache misses
- minimize the data cache misses using data prefetching
- increase the instructions per cycle that the CPU can fetch

VPP consists of a set of nodes, also illustrated in 1 namely :

- *Input Nodes* : produce data from a network driver for consumption. These are present at the start of the graph, they are responsible for generating packets from the NIC and push them into rest of the graph, i.e. abstract the NIC and manage an initial vector of packets. e.g. `dpdk-input` in Fig. 1 polls packets from the NIC and pushes them to forwarding graph.

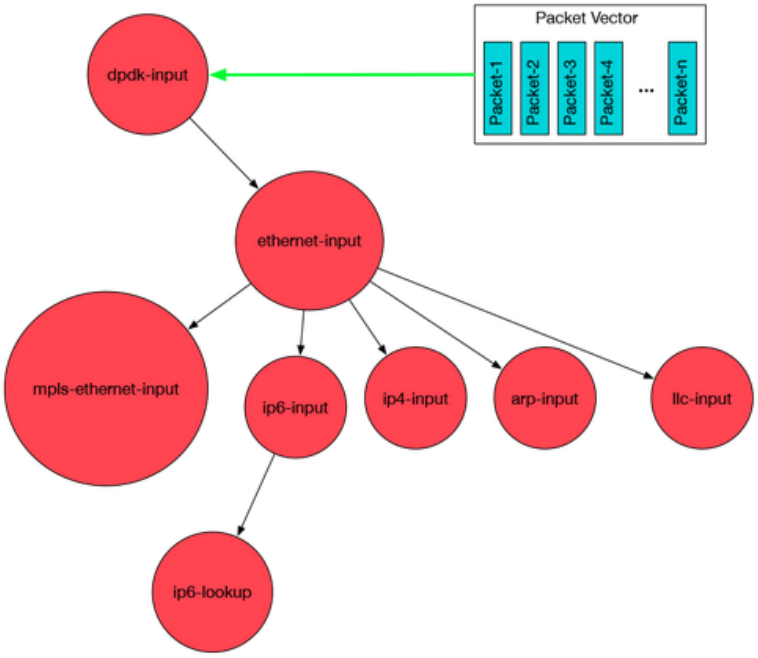


Figure 1: VPP Forwarding Graph

- *Internal nodes* : These are traversed after an explicit call by an input node or another internal node. These get their packets from another source and are responsible for processing incoming data in the graph. e.g. `ip4-lookup, ip6-lookup` in Fig. 1, which are responsible for looking up the routing path corresponding to dest. IP address is called by another internal node, `ip4-input`.

- *Process nodes* : This exposes a thread-like behavior, in which the node's callback routine can be suspended and reanimated based on events or a timer. This is useful for sending periodic notifications or polling some data that's managed by another node. e.g. modifying forwarding tables. So long as the table-builder leaves the forwarding tables in a valid state, one can suspend the table builder to avoid dropping packets as a result of control-plane activity.

As illustrated in 2, the graph node dispatcher pushes the work vector through the directed graph, subdividing it as needed, until the original work vector has been completely processed. At that point, the process recurs. A vector of packets briefly encounter the following nodes, as shown in Fig. 2. This shows an incoming vector of different L2, L3(IPv4, IPv6) packets, polled by the input node, `dpdk-input`. The vector is pushed to the next input node, which parses their headers, and passes them to the respective internal nodes, like `ip4-input` for IPv4 packets. Now, a vector of IPv4 packets are formed post redistribution for processing. The processed packets are then assembled after processing for

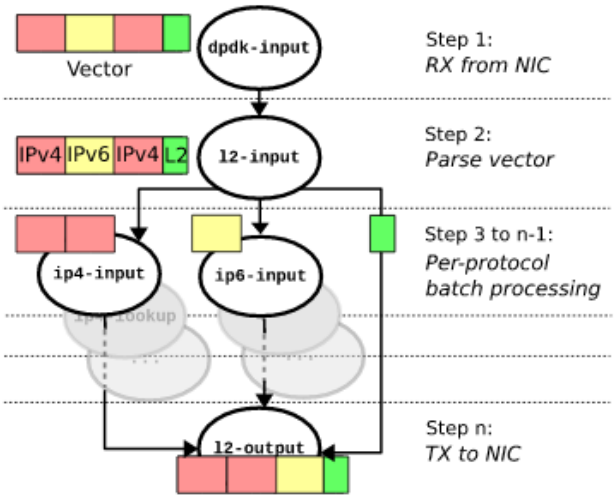


Figure 2: Packet Processing Path

output to NIC.

2 Working

First, a batch of packets is polled from the DPDK interface, after which the entire 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), which explains the full CPU utilization of 100% by each of the worker polling threads, witnessed via `htop` command.

VPP can work in two modes, as per 12:

- *Single-thread* : In this case, only one main thread handles both packet processing and other management functions (Command-Line Interface (CLI), API, stats). This is the default setup.
- *Multi-thread with worker threads* : In this mode, the main threads handles management functions(debug CLI, API, stats collection) and one or more worker threads handle packet processing from input to output of the packet. Each worker thread polls input queues on subset of interfaces. With **RSS (Receive Side Scaling)** enabled, multiple threads can service one physical interface (RSS function on NIC distributes traffic between different queues which are serviced by different worker threads through hash functions over 5-tuple{IP address, L4 protocol and ports}).

The instruction cache gets warm with the instructions from a single graph node, and consequently loads the next nodes in order to process the entire vector of packets from input to output interface. *Sufficiently long packet processing paths can cause significant i-cache misses.* On the other hand, the data cache gets warm with first few packets from the vector in a multi-loop setup, prefetching the next set of packets into the d-cache, while processing the current

set. These caches work using the '*Least Recently Used*' policy. A step-by-step demonstration can be found in 10.

2.1 Optimizations

- *Multi-loop implementation* : N packets are simultaneously processed in parallel, since computations on packets $i, i+1, ., i+N-1$, are typically independent of one another. This allows CPU pipelines to be persistently full and delays due to cache misses are amortized among these N packets instead of one. VPP uses a quad loop implementation ($N = 4$)

- *Data prefetching* : Once the node is invoked, it is posible to prefetch into the data cache, the $i+1^{th}$ packet while procesing the i^{th} packet. It can be combined with the above multi-loop implementation, to prefetch packets from $i+1$ to $i+N$, while processing packets $i-N$ through i. There is no prefetching for the first batch of N packets, while there is nothing to prefetch for the last batch. However, these effects are negligible over a large vector size, as in VPP where the default value is 256.

- *Cisco Express Forwarding* [3]: VPP maintains a *Forwarding Information Base* and the *adjacency table*. This is used to make IP destination *prefix-based switching decisions*. The FIB maintains next-hop address information based on the information in the IP routing table. Because there is a one-to-one correlation between FIB entries and routing table entries, the FIB contains all known routes and eliminates the need for route cache maintenance. Any change in routing table is updated in the FIB. **The FIB is specific to an interface and maintained separately for each interface.**
  Instead of carrying the plain next hop's IP address from the routing table over into the FIB, each entry in the FIB that represents a destination prefix can instead contain a pointer toward the particular entry in the adjacency table that stores the appropriate rewrite information: Layer 2 frame header and egress interface indication. The adjacency table maintains Layer 2 next-hop addresses for all FIB entries.

  The FIB contains all known destination prefixes from the routing table, plus additional specific entries, organized as *mtrie* or a *multiway prefix tree*, detailed in 4. Generally, to identify the outbound interface, we lookup the dest. IP from the trie. Traditionally, tries are used only for routing, and distinguish only between leaf and node elements. The basic building block of all Mtrie plus nodes is an oppointer including an address(preferably, that of the next node) and an opcode. The opcode dictates the action the switch has to take on the packet label to select the next oppointer node. If it points to an 8-bit termination leaf, the lookup is terminated. High speeds are achieved by the multi-pipelined threads of the MTrie Plus engine.

3 Goal

We are trying to develop an overall understanding of the working of VPP and evaluate the overall performance of its *batch-packet-processing heuristic*. In ad-

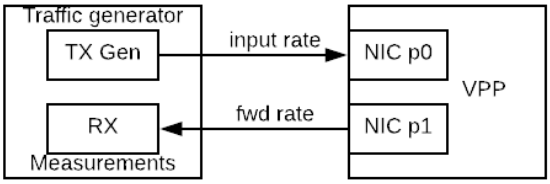


Figure 3: Experimental Setup

dition, we are trying to analyze its latency, throughput, cache performance, performance scale-up with increasing cores, amidst other statistics. This will help us efficiently allocate resources to the switch for different use cases, and maximize its performance under different scenarios.

4 Evaluation

4.1 Setup

We connect two machines, one acting as **server** running VPP and another acting as the **Traffic generator and sink** (packet generator and receiver) as shown above in Fig. 3. Both of the are equipped with dual-port NICs capable of handling *10 Gbps* traffic and the corresponding ports of the two NICs on either machine are connected via LAN cables. One port on client NIC is assigned to *traffic generation* and the other one for *RX measurements*.

4.2 Traffic Load generation

We generate different kinds of L3 traffic using the MoonGen packet generator. The first one simply involves sending packets **sequentially** in a given IP range (round-robin fashion). The *gaps* between the destination IP addresses are varied. Second, we generate packets from a **Gaussian** distribution with *mean* as the midpoint of the IP address range, with varying *variance*.

This involves **IPv4 processing**, where all traffic follows a single path in the VPP graph.

4.3 Setup parameters

VPP version	18.04 stable
DPDK version	18.02.1 stable
CPU	Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
Logical cores	48
NIC	10-Gigabit X540-AT2
Packet Generator	MoonGen
Traffic load	10 Gbps
Packets	UDP
Packet size	128 bytes
L1 data cache	32 KB
L1 instruction cache	32 KB
LLC (L3)	30720 KB

4.4 Vector Size/No. of packets in a vector

We can configure the *maximum vector size* at compile time, by modifying `VLIB_FRAME_SIZE`, in `vpp/src/vlib/node.h` but the actual vector size depends on the processing task. This can be measured by looking up `$vppctl show run`. The miminimum specified size needs to be 4, since VPP works on a quad-loop implementation, hence fetches 4 packets to process at once.

A good indicator of CPU load is ‘`average vectors/node`‘, which shows how many packets are processed in a batch, as mentioned in 5. A bigger number means VPP is more busy but also more efficient. As we see below in Fig. 4, if VPP is not loaded it will likely poll so fast that it will just get one or few packets from the RX queue. And as load goes up, VPP will have more work, so it will poll less frequently, and that will result in more packets waiting in RX queue. The decrease in number of packets in a batch is *not linear* with the increase in worker threads, because *it is dependent on polling rate of the thread*(which will increase now, thread being less loaded) and traffic load(which is fixed at 10 Gbps here). A similar analysis has been done in 8.

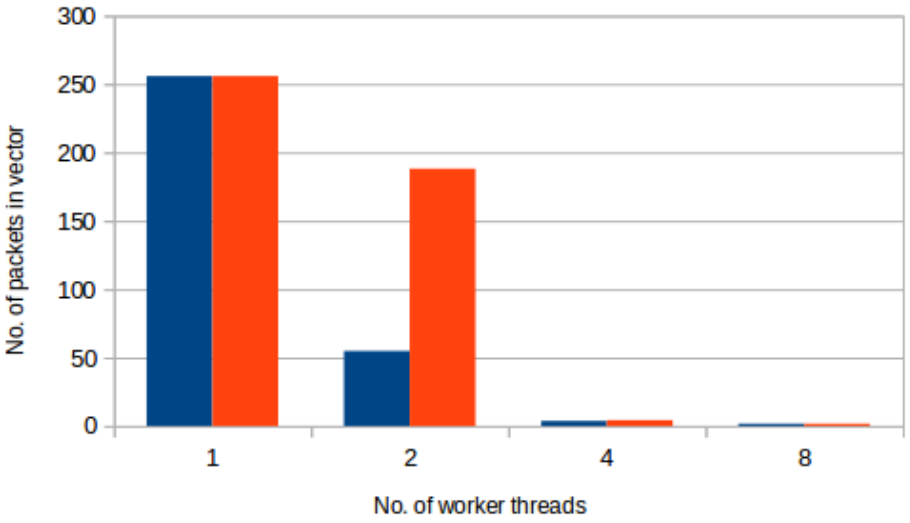


Figure 4: No. of packets in Vector Vs. No. of workers/cores
BLUE : Uniform, RED : Gaussian
No. of flows : 256

We see in Fig. 4 that in case of 2 worker threads, Gaussian traffic load incurs a *higher batch size* owing to the fact that most packets belong to a narrow dest. IP range compared to uniform flow of packets. Hence, a *smaller set of same instructions* need to be executed for a larger batch, leading to a larger aggregation ensuring efficient execution, in terms of higher cache hit rates, etc.

We now try and plot the vector size as a function of the number of entries in the routing table in Fig. 5. We see that the vector size for uniform traffic load is almost the same for different routing table sizes, and we see corresponding larger vector sizes for the Gaussian traffic load, as explained above.

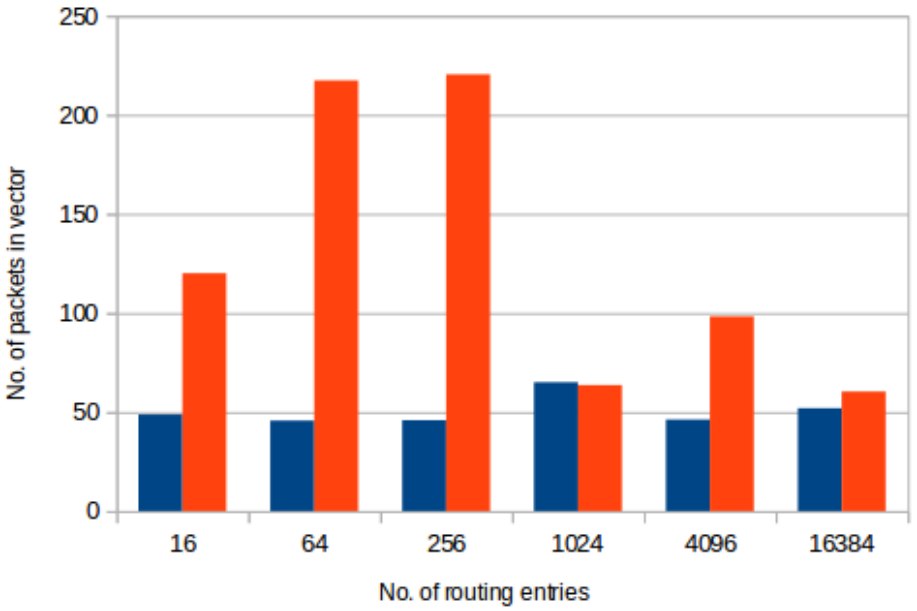


Figure 5: No. of packets in Vector Vs. Number of routing entries
BLUE : Uniform, RED : Gaussian
No. of workers : 2

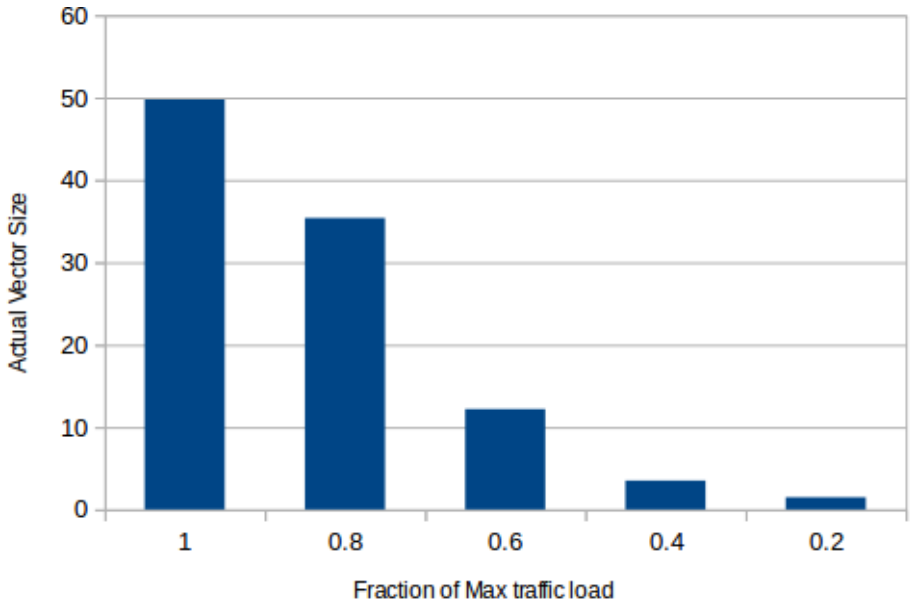


Figure 6: No. of packets in Vector Vs. Fraction of Max Traffic load (10Gbps)
No. of workers : 2, Flow : Uniform

As seen in Fig. 6, the input traffic load when decreased, i.e., lesser no. of

packets sent to the switch per unit time, the threads being less loaded, poll in lesser packets into a vector. More packets result in more efficient execution, so number of clock cycles/packet will go down. When "vector size" goes up close to `VLIB_FRAME_SIZE`, we observe RX queue tail drops.

4.5 Instructions per Packet/Cycles Per Packet

The number of cycles per instruction helps probe into the instruction parallelisation capabilities of the framework, along with per-packet processing metric like instructions per packet. A combination of these two parameters give us the number of clocks per packet.

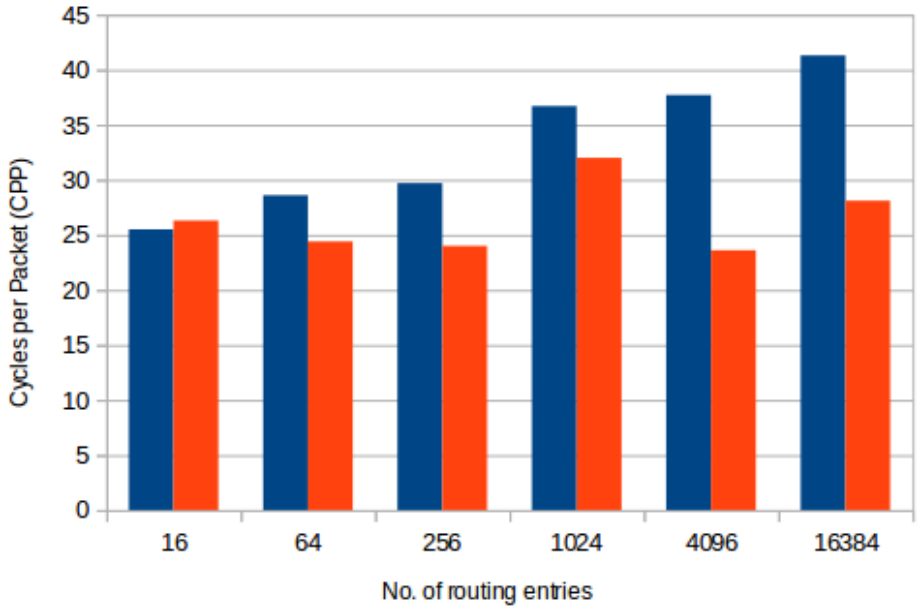


Figure 7: Cycles Per Packet Vs. No. of entries in routing table
BLUE : Uniform, RED : Gaussian
No. of workers : 2, Max Vector Size : 256

Taking inspiration for 7 and 11, which claims VPP's success in reduced CPP, we now focus on the **ip4-lookup** node in the forwarding graph responsible for performing the lookup for the packet according to its dest. IP address. We plot the cycles for packet for different number of entries in the routing table in Fig. 7.

We see that the number of cycles per packet increases as we increase the number of routing entries. This is consistent with the fact that as the number of routing entries increase, to lookup the corresponding route for dest. IP prefix takes longer time, to search in the routing table, in the case of *uniform traffic flow*. However, for the *Gaussian traffic load*, we see that the cycles per packet is almost the same, at roughly 26.

4.6 Number of Entries/Routing table size

We forward packets at the maximum throughput rate of 10 Gbps and record the forwarding rate observed at the RX side after the VPP processing. We do this as a function of the number of routing rules added to VPP. The operation is more expensive for larger packets though VPP only parses through and processes the packet headers, because the entire packet has to be prefetched and loaded onto the data cache for processing.

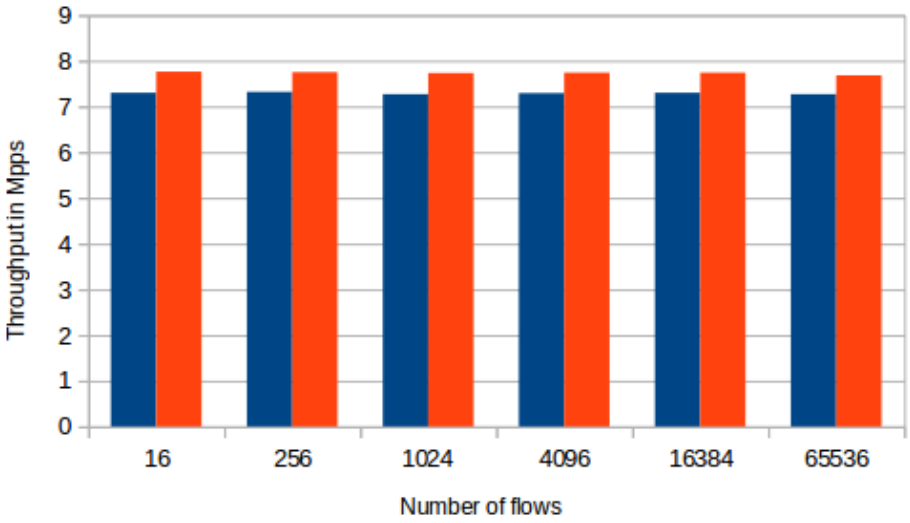
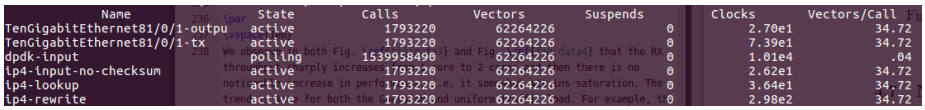


Figure 8: RX Throughput in **Mpps** Vs No. of routing entries;
No. of workers : 2, Max Vector Size : 256
BLUE : Uniform, RED : Gaussian

We observe almost no change in the RX throughput as a function of the number of routing rules, in both Fig. 8 and 11 owing to the fact that VPP is able to store the routing instructions for the vectors of packets, in a node that fits into the i-cache, and the data structure employed for such storing of rules is space-efficient so as to avoid significant i-cache misses. This agrees with VPP's claim in 6.

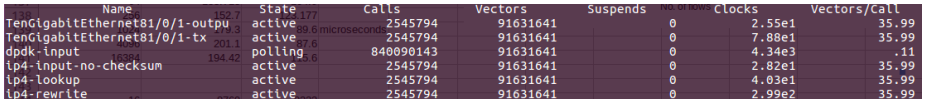
For the Gaussian traffic load, RX throughput is greater than Uniform owing to the fact that there are relatively packets from a smaller IP range in the former case. Hence, the i-cache can sustain with the same set of routing instructions for those packets, without significant misses, implying lesser processing time.

As depicted in Fig. 7, the cycles per packet increase for increase in size of the routing table, without any decline in throughput as witnessed in Fig. 8 and 11. However, it is only for the node `ip4-lookup`. The increase in CPP for this node is compensated by a decrease for other nodes in the processing path, notably `dpdk-input`(which polls packets to form vectors) and other output nodes. This is shown as part of experiments in Fig. 9 and 10



Name	State	Calls	Vectors	Suspends	Clocks	Vectors/Call
TenGigabitEthernet81/0/1-outpu	active	1793220	62264226	0	2.70e1	34.72
TenGigabitEthernet81/0/1-tx	active	1793220	62264226	0	7.39e1	34.72
dpdk-input	polling	1539958490	62264226	0	1.01e4	.04
ip4-input-no-checksum	active	1793220	62264226	0	2.62e1	34.72
ip4-lookup	active	1793220	62264226	0	3.64e1	34.72
ip4-rewrite	active	1793220	62264226	0	2.98e2	34.72

Figure 9: Runtime statistics
No. of workers : 2, Max Vector Size : 256, Routing table size : **4096**



Name	State	Calls	Vectors	Suspends	Clocks	Vectors/Call
TenGigabitEthernet81/0/1-outpu	active	2545794	91631641	0	2.55e1	35.99
TenGigabitEthernet81/0/1-tx	active	2545794	91631641	0	7.88e1	35.99
dpdk-input	polling	840090143	91631641	0	4.34e3	.11
ip4-input-no-checksum	active	2545794	91631641	0	2.82e1	35.99
ip4-lookup	active	2545794	91631641	0	4.03e1	35.99
ip4-rewrite	active	2545794	91631641	0	2.99e2	35.99

Figure 10: Runtime statistics
No. of workers : 2, Max Vector Size : 256, Routing table size : **16384**

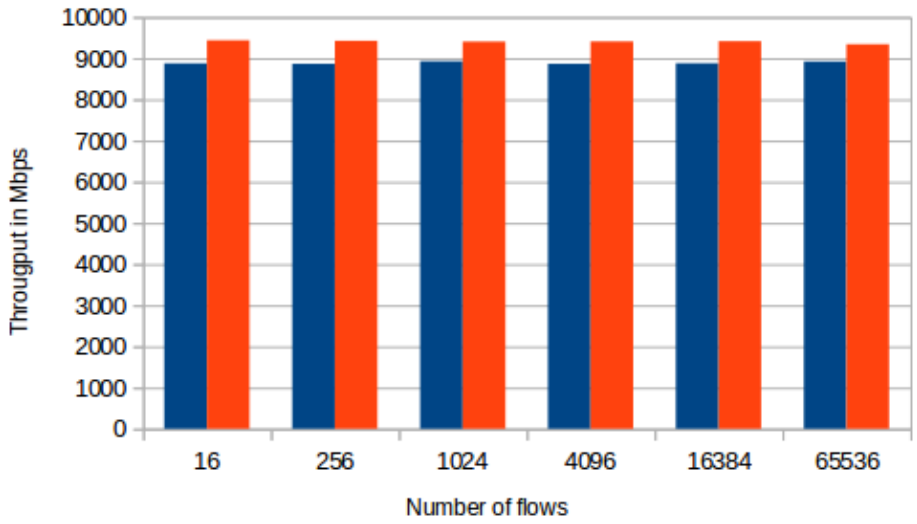


Figure 11: RX Throughput in **Mbps** Vs No. of routing entries;
No. of workers : 2, Max Vector Size : 256
BLUE : Uniform, RED : Gaussian

4.7 No. of cores

We record the performance scaling up of VPP with the increase in the number of cores, i.e., worker threads, where each worker thread polls and collects packets separately at RX.

We observe in both Fig. 12 and Fig. 13 that the RX throughput sharply increases from 1 core to 2 cores and then there is no noticeable increase in performance, i.e. it somewhat attains saturation. The trend is same for both the Gaussian and uniform traffic load. For example, the RX throughputs for Uniform flow are 7.27 Mpps and 7.33 Mpps for $4096, 256$ entries in the routing table respectively, showing barely any difference.

We also notice the same trend when we increase the *RX-queues*, since VPP can now make use of *Receiver Side Scaling* to handle packets faster, leading to enhanced RX throughput rates.

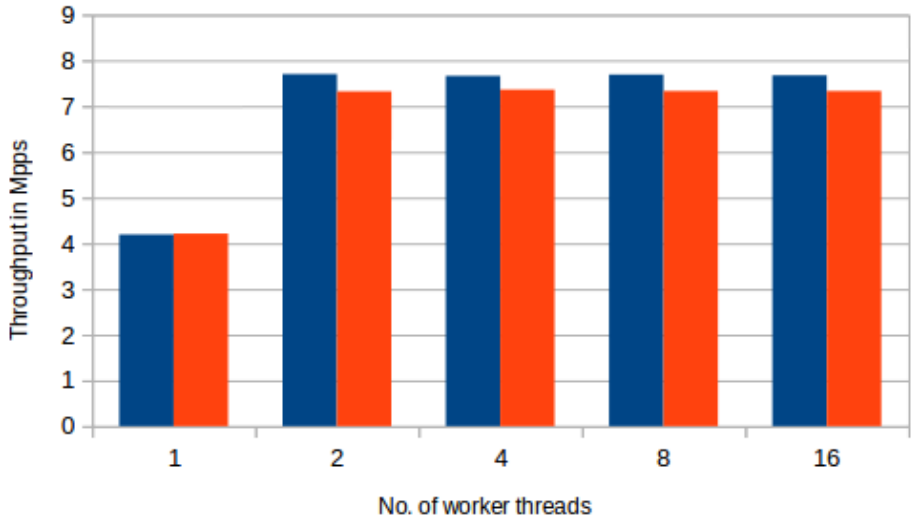


Figure 12: RX throughput(Mpps) Vs No. of cores/worker threads
BLUE : Gaussian, RED : Uniform
No. of flows : 256, Max Vector Size : 256

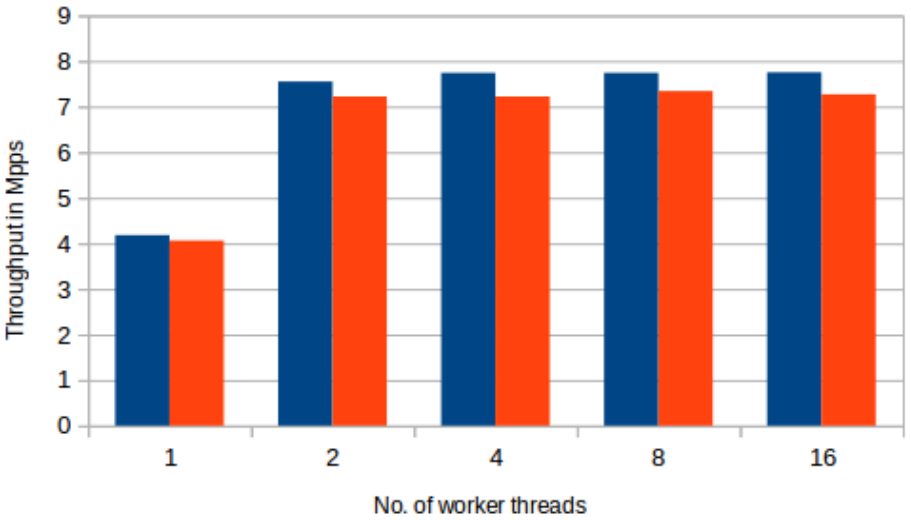


Figure 13: RX throughput(Mpps) Vs No. of cores/worker threads
BLUE : Gaussian, RED : Uniform
No. of flows : 4096, Max Vector Size : 256

4.8 Cache Performance

We also compute the miss rates in the L1 instruction cache and the LLC cache to measure the benefits of vector processing.

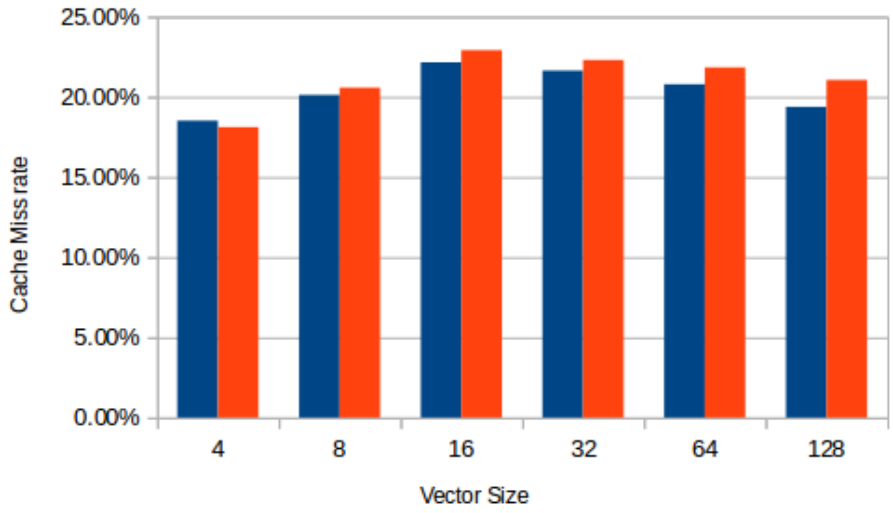


Figure 14: LLC(L3) Cache miss rate vs. Vector Size
BLUE : Uniform, RED : Gaussian

For an increasing vector size, the data cache miss rate may vary, LLC miss rate reflects changes in both the i-cache miss rate and d-cache miss rate. However, as long as the vector size is lesser than 256, d-cache misses are rare since L1-d cache is of 32 KB and so no. of 128 byte packets that fit would be $\frac{32\times1024}{128} = 256$ packets.

The trend in Fig. 14 is because it has to frequently load a different set of routing instructions for different vectors, which change frequently when the vector size is small. This overflows the i-cache in a smaller amount of time and leads to more i-cache misses per unit time. We accordingly observe an anticipated gentle fall in the LLC miss rate. However, we also observe a very small unexpected rise in the LLC miss rate with increasing vector size from 4 till 16 in Fig. 14. Overall, we observe no significant changes in cache miss rate. However, this can be misleading if L2, L3 cache miss rates are significant compared to L1 cache misses and the i-cache is large enough to hold several instruction nodes for different vectors one after the other.

Now, we generate traffic with different leading prefixes, so that VPP can't make use of the fast lookup using mtries. We generate traffic sequentially with little spatial similarity between consecutive dest. IP addresses. Now, we try to probe the i-cache miss rate as a function of the number of flows/rules. In Fig, 15, we plot the explicit i-cache miss rate as a function of different number of flow rules, keeping the max vector size as 64, so that in all cases, the actual number of packets in a vector is forced to be 64.

We see that the i-cache miss rate increases and rises sharply from 16 to

256 flows as expected, and peaks at 256, but however, goes down again from 512 routing entries till 4096. Since as we keep on adding more rules, the dest. IP addresses become spatially closer and at some point, for more rules, there are no longer additional tries required for storing routing addresses, and hence there is no significant cache miss rate. However for lesser no. of rules till 256, we observe the L1 i-cache miss rate to be increasing so as to fetch on tries corresponding to the spatially distant routing prefixes of different packets, which explains significant L1 i-cache misses.

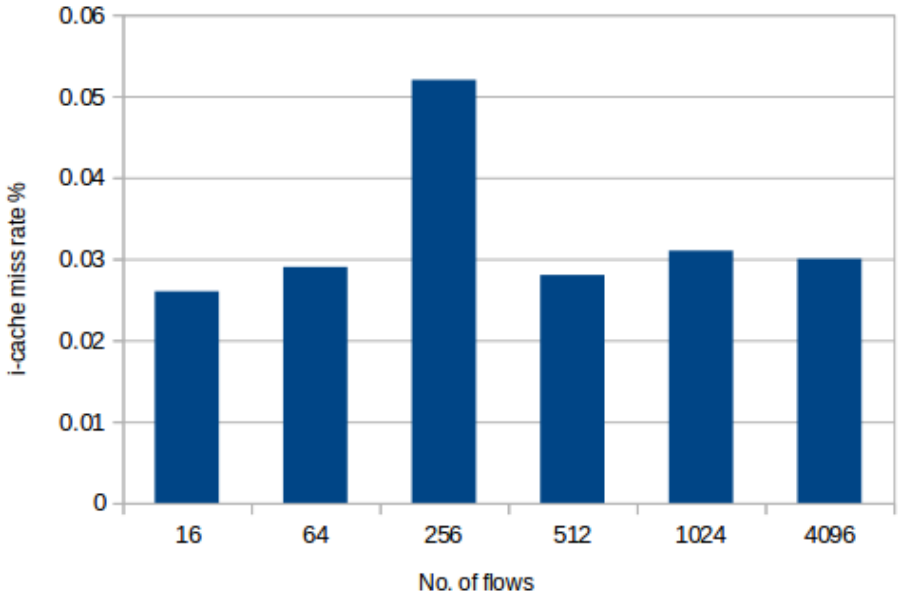


Figure 15: L1 i-cache miss rate Vs. No. of flows
Actual Vector size : 64

Now, we try to keep the routing table size fixed to 256 entries, with the load being generated as explained above, but with varying vector sizes. Since data cache misses are negligible, we expect the trend in Fig. 14 to be somewhat similar to Fig. 16. This is not completely true, since LLC cache miss rate reflects L2 and L3 cache misses as well, so 14 will not be completely representative of 16 as we see above. The i-cache miss rate should be decreasing since smaller vectors are processed fast and so different instructions will be loaded in lesser time, leading to more overflows and consequent i-cache misses in unit time. This might be due to the fact that the i-cache can fit in several instruction nodes for processing vectors one after the other without overflowing and apart from `ip4-lookup`, there are other nodes, which might have a significant role in packet processing, and are dependent upon the size of the vector, whose miss rates are significant.

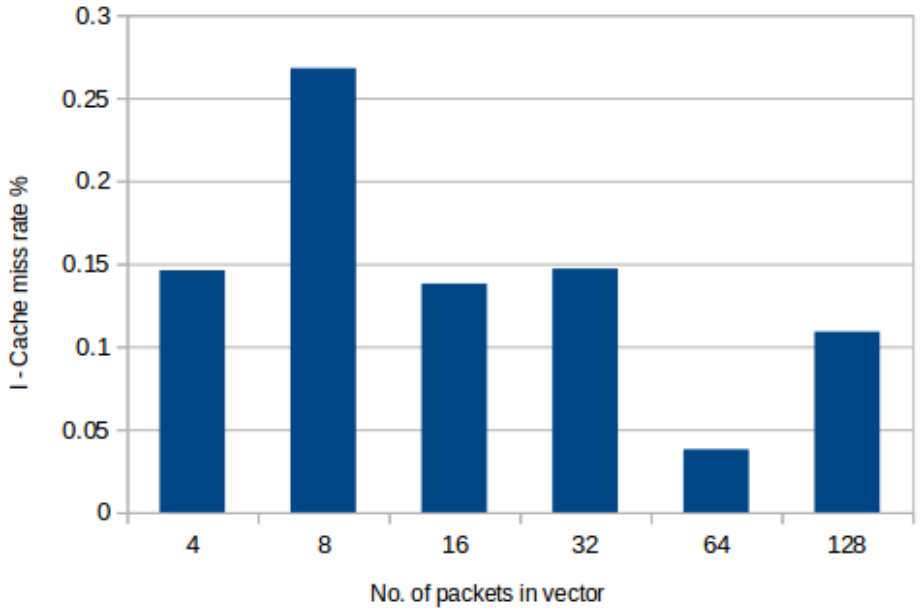


Figure 16: L1 i-cache miss rate Vs. Vector Size
Routing table size : 256, Workers : 2

4.9 Latency

The latency of individual packets are higher in comparison to scalar packet-processing heuristics, as a packet will be emitted to the output interface only after the entire vector of packets has been processed. Here, we plot the latencies of packets, corresponding to increasing number of routing entries.

As we see in Fig. 17, the overall latencies per packet of the uniform traffic load generation is more than that of the Gaussian traffic load, owing to the fact that the latter contains dest. IP addresses from a smaller range, and requires lesser pre-processing time, since the same instruction `ip4-lookup` node can be reused(if in i-cache) for routing the packets to dest. IP addresses, whereas for the uniform traffic load, a set of routing instructions for each vector has to be fetched in most of the cases, since the routing addresses(dest. IP address) vary over a large range.

The uniform traffic load shows an overall increasing trend, barring the peak at 1024 routing entries, since we expect that the per-packet processing rate increases, since each packet has a distinct IP address, and hence for each vector of packets, a different set of instructions correspond to routing of such packets, which has to fetched while searching amidst a larger set of rules when routing table increases in size. For the Gaussian case, the latencies are almost the same, barring the dip at 4096 entries, since the same instruction node can be used for many vectors of packets, which can be found in the i-cache more often, not having to lookup among the increasing number of rules in the routing table.

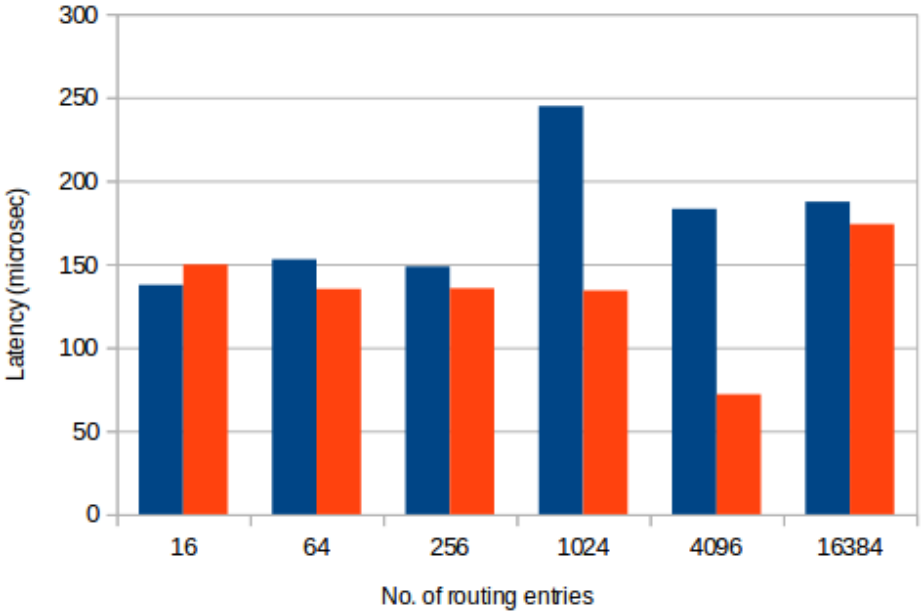


Figure 17: Latency (per packet) Vs. No. of routing entries
BLUE : Uniform, RED : Gaussian
Max Vector size : 256, Workers : 2

4.10 Drawbacks

- VPP crashes frequently when trying to add more than 16384 flow rules (IP entries) and ARP entries.
- We do not have any explicit control on the Cache parameters or the Actual vector size.
- MoonGen has an inherent **rate control**, and hence it tries to send no more packets to the switch than can be handled in case of smaller packet sizes like 64 bytes. This is mentioned in Rate Control in MoonGen.

5 Conclusion

We have tried to explore the working and various optimizations of VPP and have developed a fair understanding of the same. We see that through batch processing in a graph, VPP amortizes the processing overhead as we see that it achieves near line rate with very insignificant decline in throughput, with increase in number of routing table entries, that too only with *2 workers/2 cores*. The performance also *scales quite well with increase in the number of cores*, besides being capable of *dividing workload among multiple RX queues using RSS*.

The key to its performance being 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)**. We see that its efficient data structures for forwarding enables high-speed packet processing upto a large routing table, and the performance is scalable with more resources, like no. of cores and RX queues.

VPP also adjusts the vector size accordingly so as to provide the best performance, and hence we see different scenarios, where vectors are formed and redistributed according to polling rate and type of load and packet-processing. The Cycles Per Packet is also minimized overall, by amortizing extra costs over a larger number of packets. This combined with relatively low latencies is witnessed(given that packets are processed in batches, overall latency increases for first few packets in a vector since they are transmitted only after processing the entire vector). This makes VPP a good choice for high-speed switching.

We have not been able to accurately evaluate the cache performance, since we were not quite sure about the heurisitic of loading of nodes in i-cache for processing of packets. We realized that the default i-cache is too big, 32 KB, given the optimizations that VPP uses for storing even very large routing tables; and in order to significantly overflow the i-cache, we needed a thorough mixture of a large number of different packets, so as to load every node corresponding to a processing a variety of packets. We needed a lot of experiments to narrow down upon this fact. In all experiments in cache, we have noted that VPP incurs a significantly lower cache miss rate, even when the routing table size is quite large.

6 References

1. Blog : Plugins and Nodes in VPP
2. VPP/Software Architecture
3. Cisco Express Forwarding
4. Patent on Mtrie packet processing
5. Documentation for VPP
6. Overview of VPP
7. Introductory White Paper on VPP
8. Survey of VPP By Cisco
9. Performance Challenges in Software Networking
10. Performance and Working of VPP : FD.io
11. Benchmarking and Analysis of Software Network Data Planes
12. Using VPP in a Multi-threaded model