The Case for a Flexible Low-Level Backend for Software Data Planes

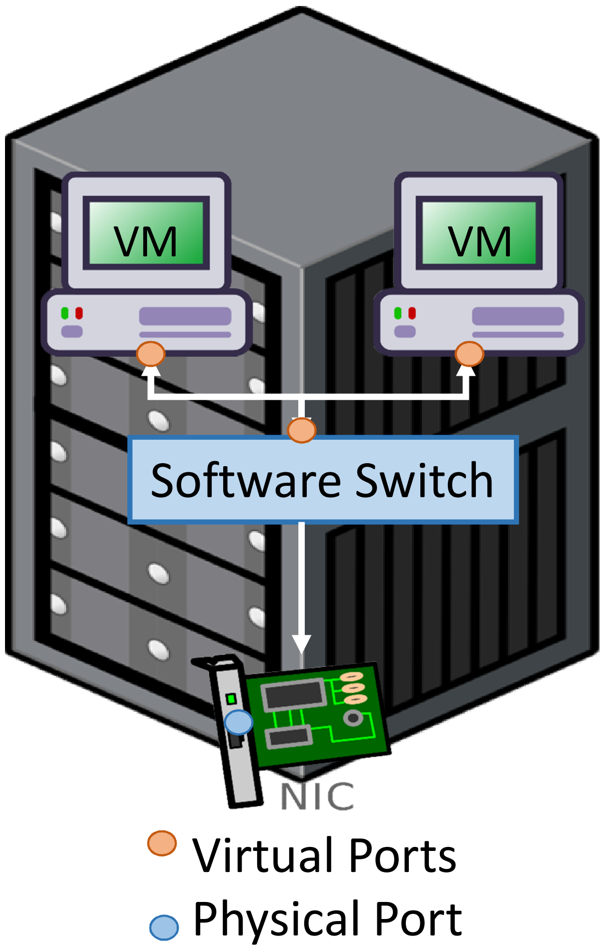
Sean Choi[1], **Xiang Long**[2], Muhammad Shahbaz[3],

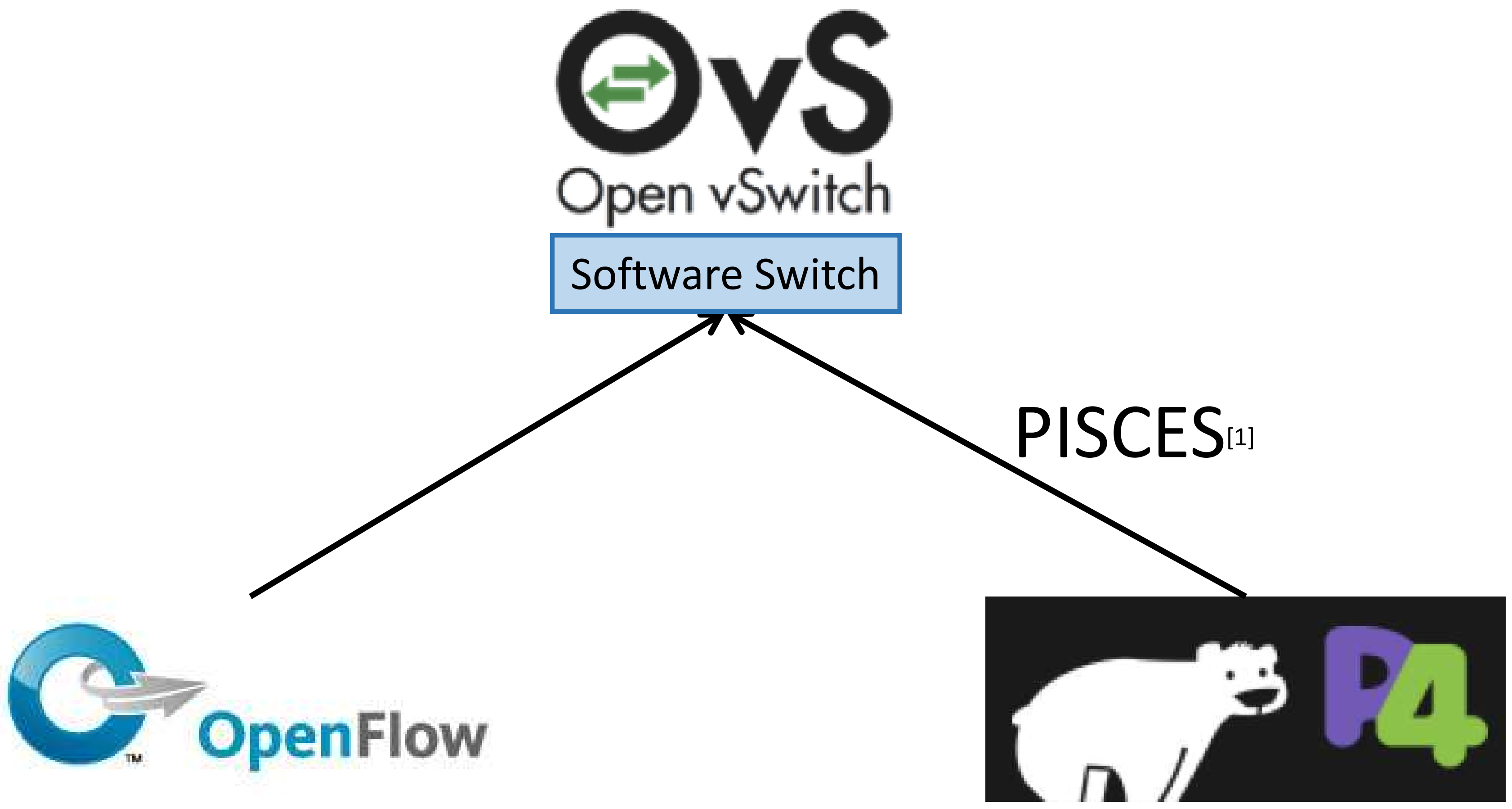
Skip Booth[4], Andy Keep[4], John Marshall[4], Changhoon Kim[5]

1 2 3 4 5

Why software data planes?

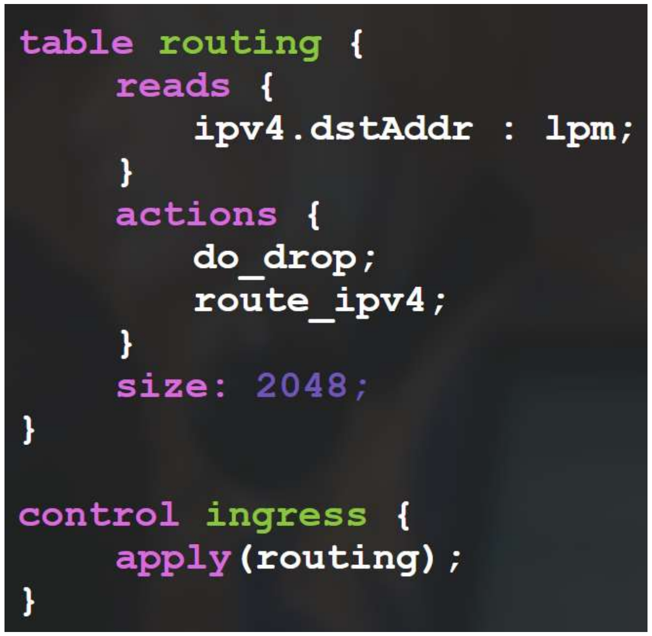
- VM hypervisors
- Cost savings with commodity general purpose processing units – where desired throughput < ~100 Gbps
- Prototyping protocol design
- Prototyping hardware DP architecture





[1] **PISCES.** *ACM SIGCOMM* 2016.

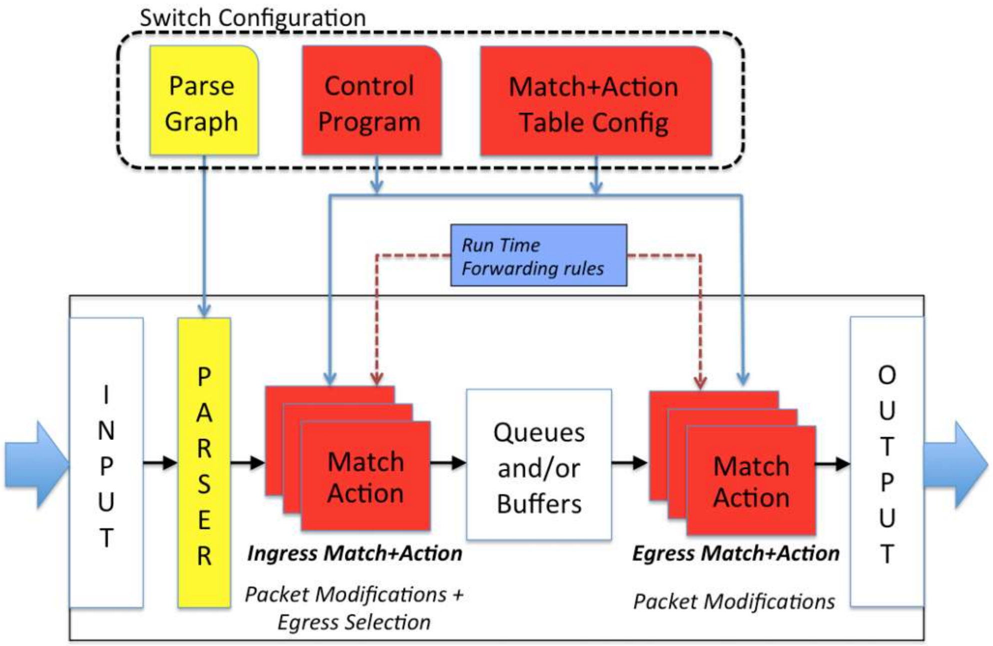
Software switch DSLs



```
table routing {
    reads {
        ipv4.dstAddr : lpm;
    }
    actions {
        do_drop;
        route_ipv4;
    }
    size: 2048;
}

control ingress {
    apply(routing);
}
```

High-level, close to protocol



Abstract forwarding model

Nice for programmers…

- Familiar and logical model in mind when programming,
  e.g. match/action pipelines
- Can specify packet data without worrying about implementation
- Portable code across platforms
- …

Not so nice for compilers

- Abstract forwarding model not designed for e.g. CPU-based architectures
- Limited in expressiveness
- Insulated from underlying low-level APIs
- Result: Difficult to realize full performance potential of underlying hardware

Hypothesis

If software switches exposed more low-level characteristics to the data plane compiler

improvements are possible in performance and features

Our contribution

- Identify a software switch that can be programmed at low-level w.r.t to the hardware architecture
- Create compiler targeting that switch to allow it to support high-level data plane programs
- Compare performance

Target Switch: Vector Packet Processor (VPP)

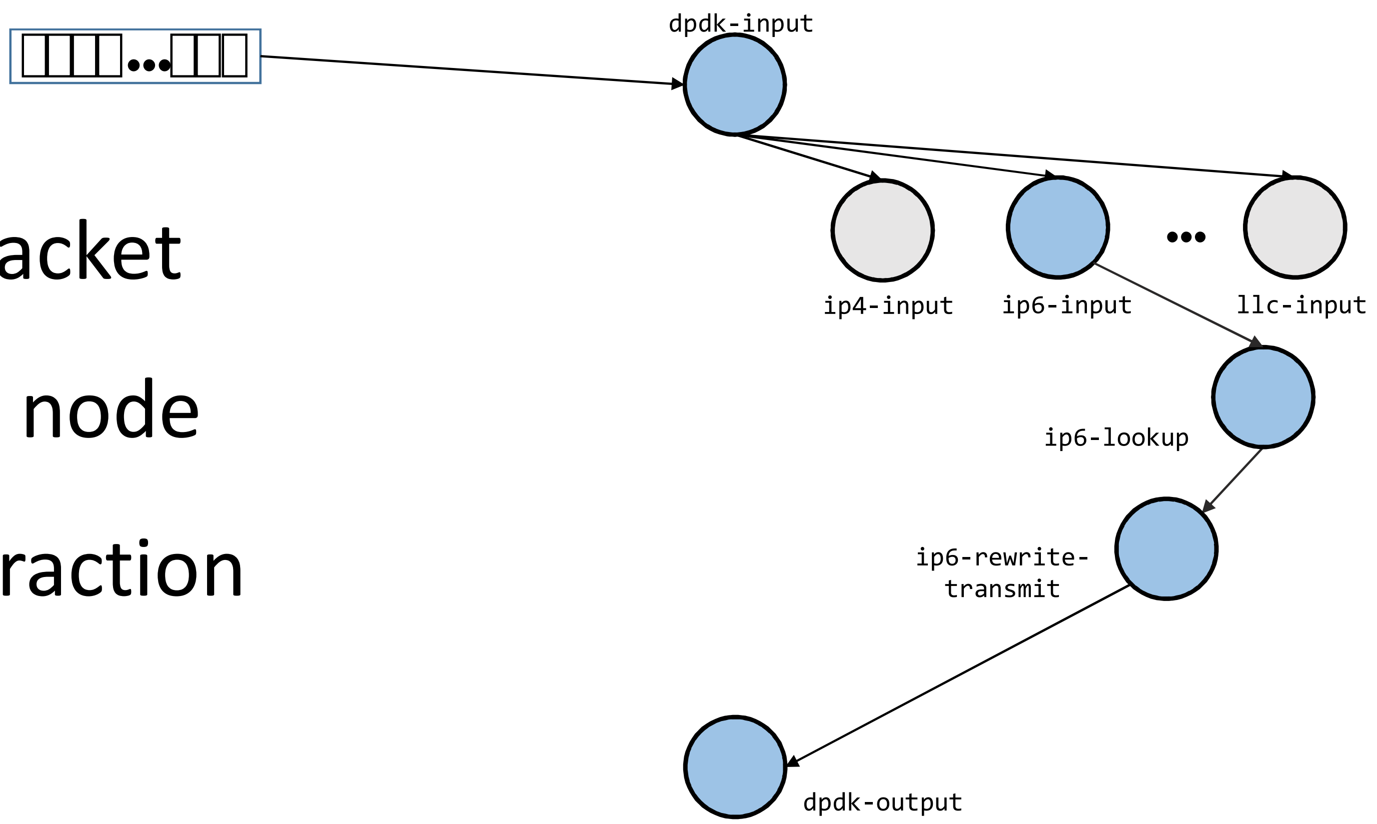
- Open sourced by Cisco
- Can be programmed at low-level

The Universal Dataplane

- Part of the FD.io project

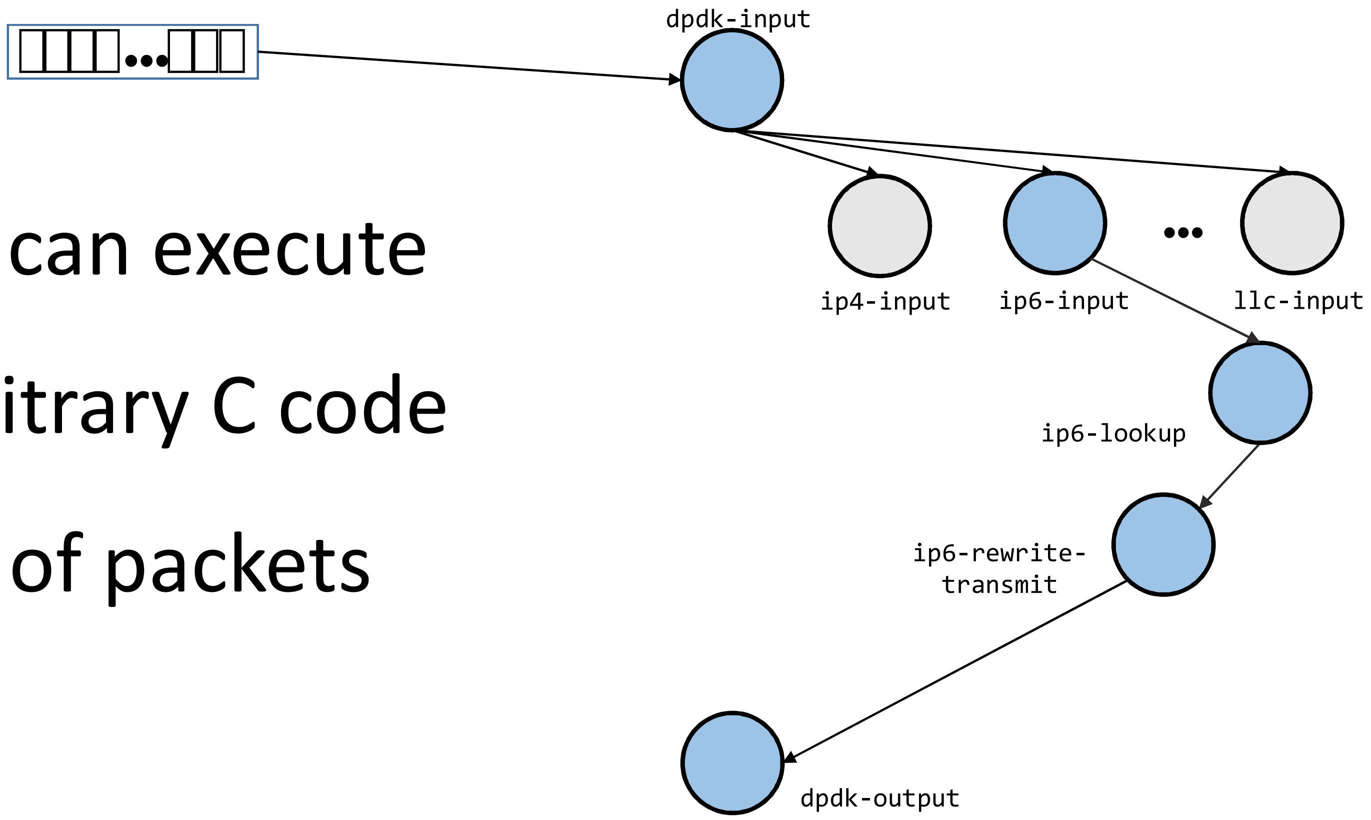
Vector Packet Processing (VPP) Platform

- Modular packet processing node graph abstraction



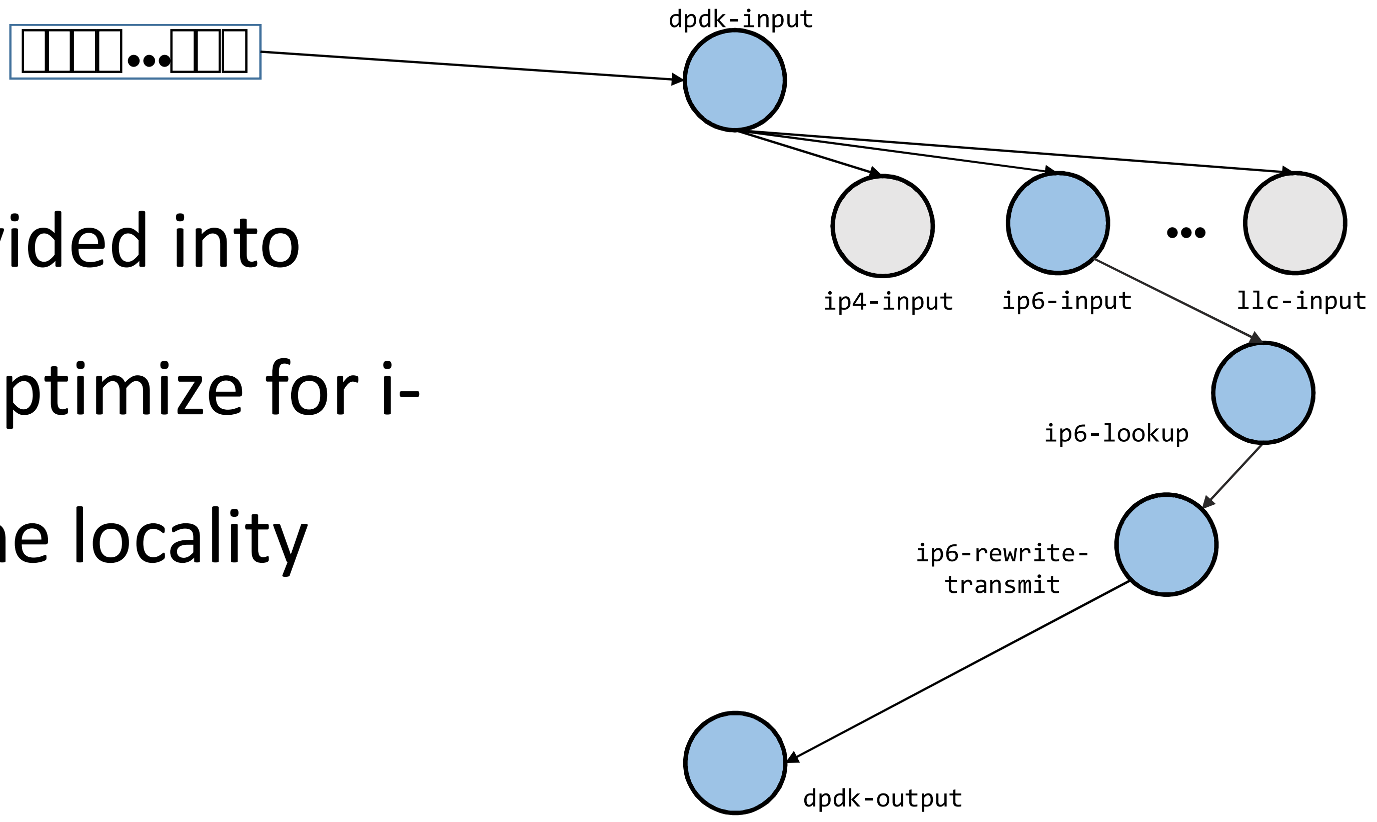
Vector Packet Processing (VPP) Platform

- Each node can execute almost arbitrary C code on vectors of packets

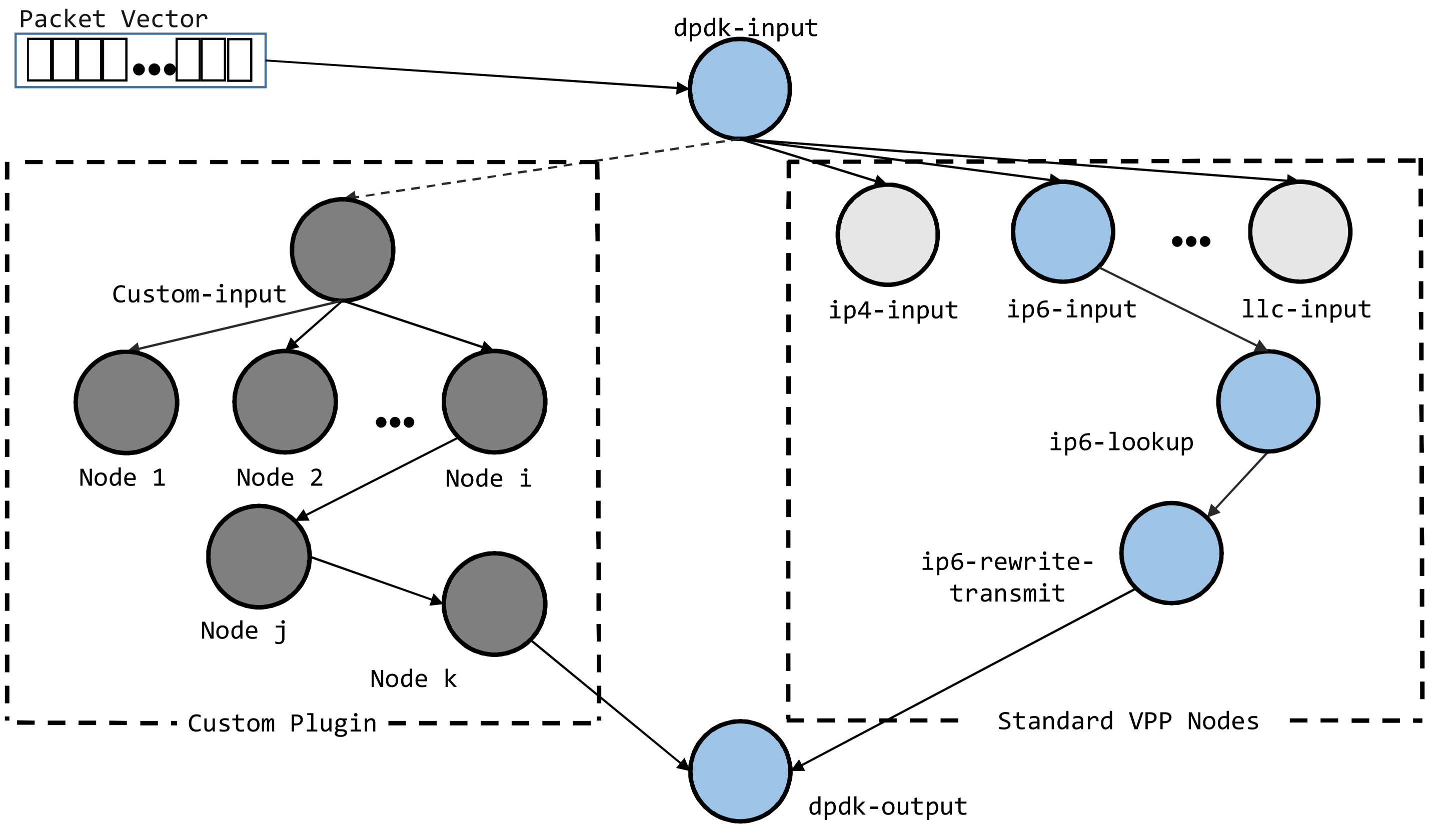


Vector Packet Processing (VPP) Platform

- Code is divided into nodes to optimize for i- and d-cache locality



Vector Packet Processing (VPP) Platform



- Extensible packet processing through first-class plugins

Vector Packet Processing (VPP) Platform

- Proven performance[1]
  - Multiple MPPS from a single x86_64 core

    1 core: 9 MPPS ipv4 in+out forwarding

    2 cores: 13.4 MPPS ipv4 in+out forwarding

    4 cores: 20.0 MPPS ipv4 in+out forwarding
  - > 100Gbps full-duplex on a single physical host
  - Outperforms Open vSwitch in various scenarios

[1] `https://wiki.fd.io/view/VPP/What_is_VPP%3F`

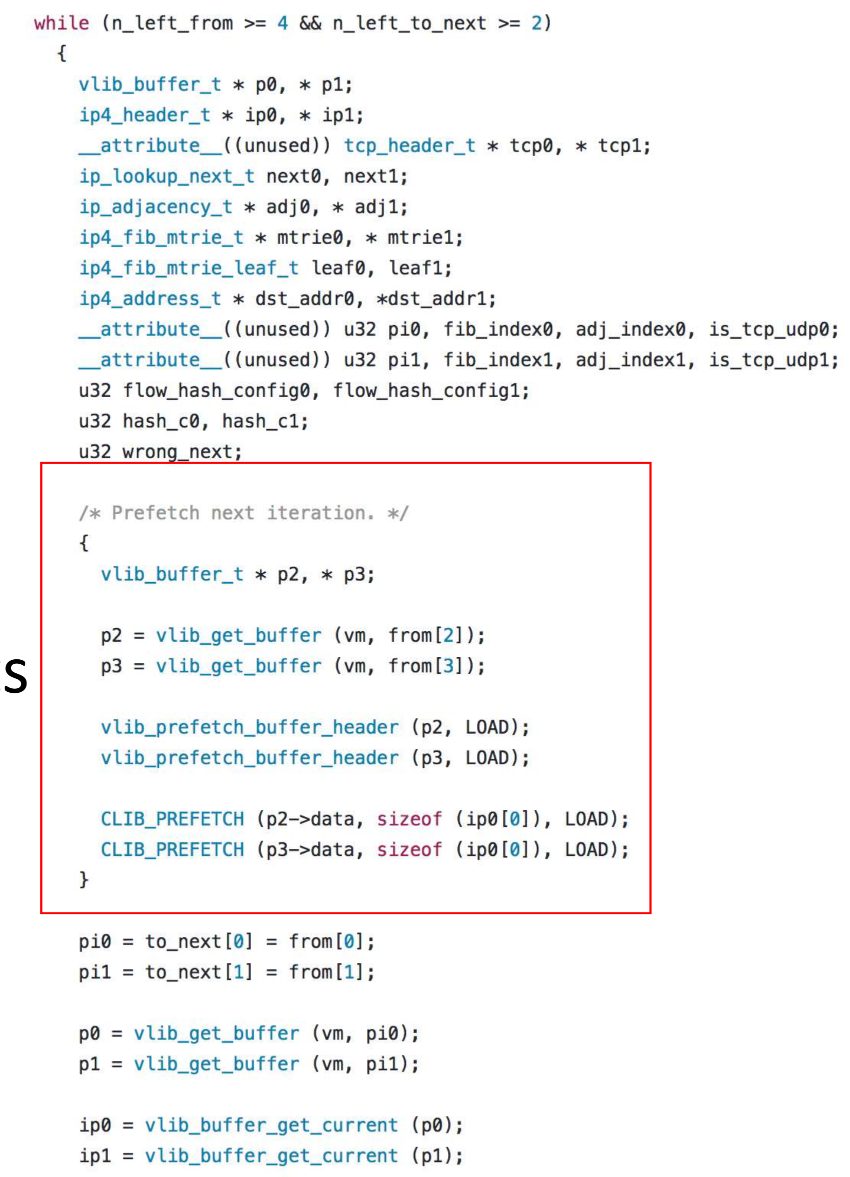
Vector Packet Processing (VPP) Platform

- Disadvantage: large burden on the programmer
- Requires knowledge from different fields: protocols, operating systems, processor architecture, C compiler optimization….
- Some Magic Required for good performance

Some Magic Required

Manually fetch two packets

Consequence of being low-level



```
while (n_left_from >= 4 && n_left_to_next >= 2)
  {
    vlib_buffer_t * p0, * p1;
    ip4_header_t * ip0, * ip1;
    __attribute__((unused)) tcp_header_t * tcp0, * tcp1;
    ip_lookup_next_t next0, next1;
    ip_adjacency_t * adj0, * adj1;
    ip4_fib_mtrie_t * mtrie0, * mtrie1;
    ip4_fib_mtrie_leaf_t leaf0, leaf1;
    ip4_address_t * dst_addr0, *dst_addr1;
    __attribute__((unused)) u32 pi0, fib_index0, adj_index0, is_tcp_udp0;
    __attribute__((unused)) u32 pi1, fib_index1, adj_index1, is_tcp_udp1;
    u32 flow_hash_config0, flow_hash_config1;
    u32 hash_c0, hash_c1;
    u32 wrong_next;

    /* Prefetch next iteration. */
    {
      vlib_buffer_t * p2, * p3;

      p2 = vlib_get_buffer (vm, from[2]);
      p3 = vlib_get_buffer (vm, from[3]);

      vlib_prefetch_buffer_header (p2, LOAD);
      vlib_prefetch_buffer_header (p3, LOAD);

      CLIB_PREFETCH (p2->data, sizeof (ip0[0]), LOAD);
      CLIB_PREFETCH (p3->data, sizeof (ip0[0]), LOAD);
    }

    pi0 = to_next[0] = from[0];
    pi1 = to_next[1] = from[1];

    p0 = vlib_get_buffer (vm, pi0);
    p1 = vlib_get_buffer (vm, pi1);

    ip0 = vlib_buffer_get_current (p0);
    ip1 = vlib_buffer_get_current (p1);
```

Ease of programmability sacrificed for performance at low-level

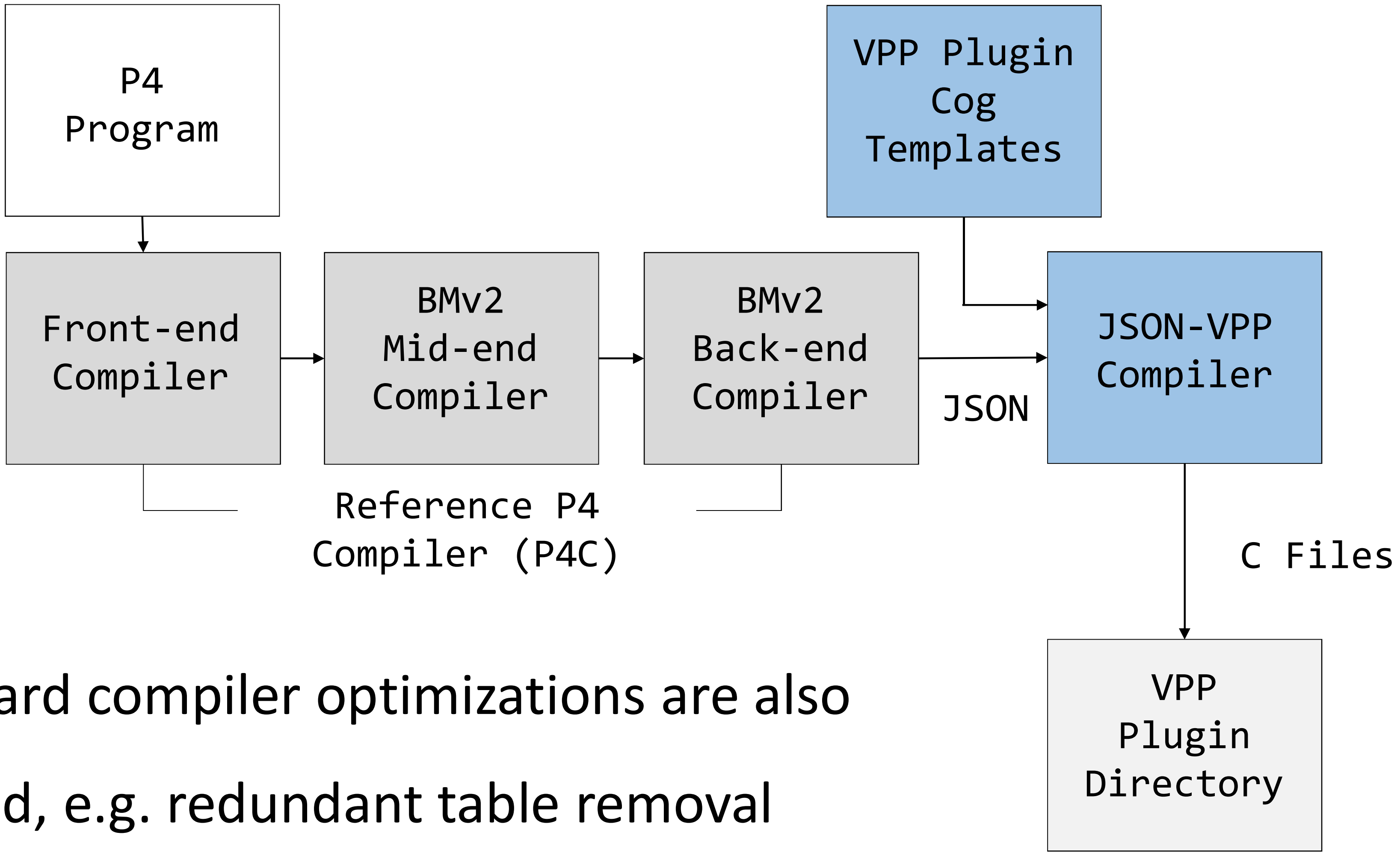
Can a high-level DSL compiler help?

+

The Universal Dataplane

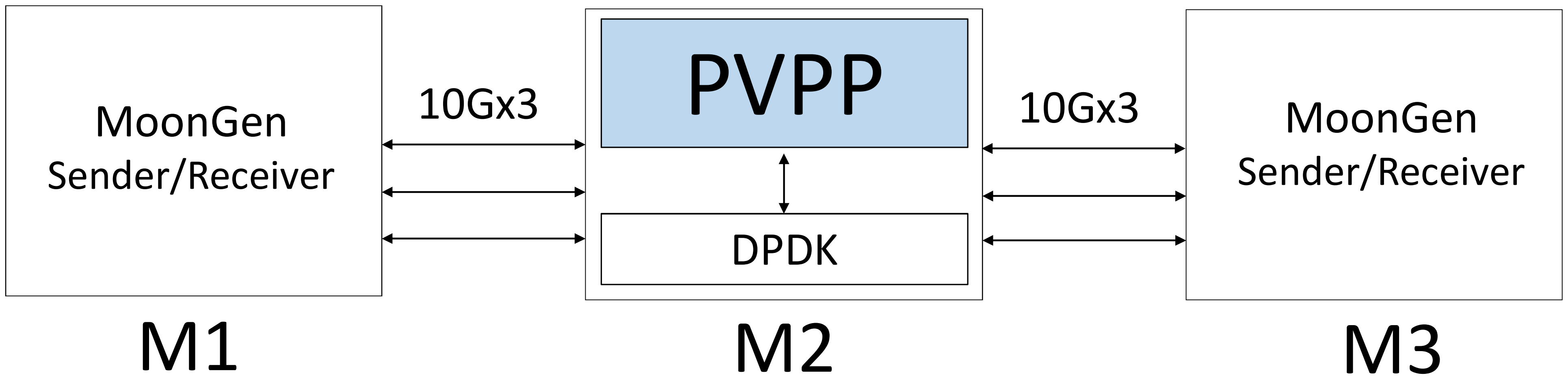
Programmable Vector Packet Processor (PVPP)

PVPP structure



Standard compiler optimizations are also applied, e.g. redundant table removal

Experimental Setup



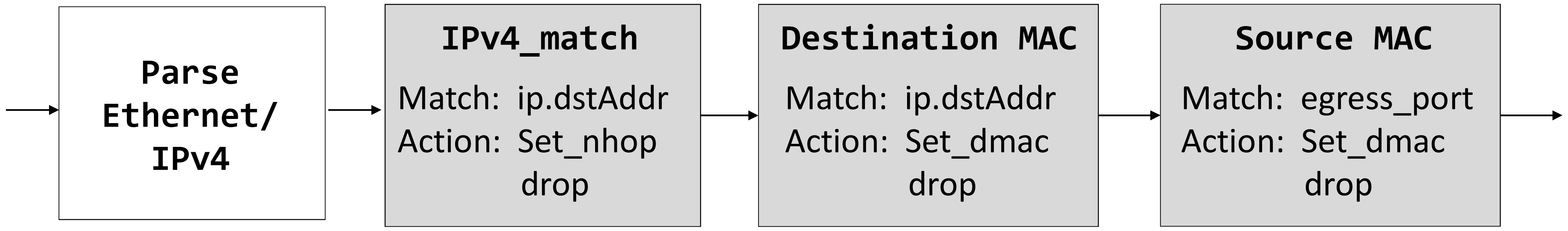
CPU: Intel Xeon E5-2640 v3 2.6GHz

Memory: 32GB RDIMM, 2133 MT/s, Dual Rank

NICs: Intel X710 DP/QP DA SFP+ Cards

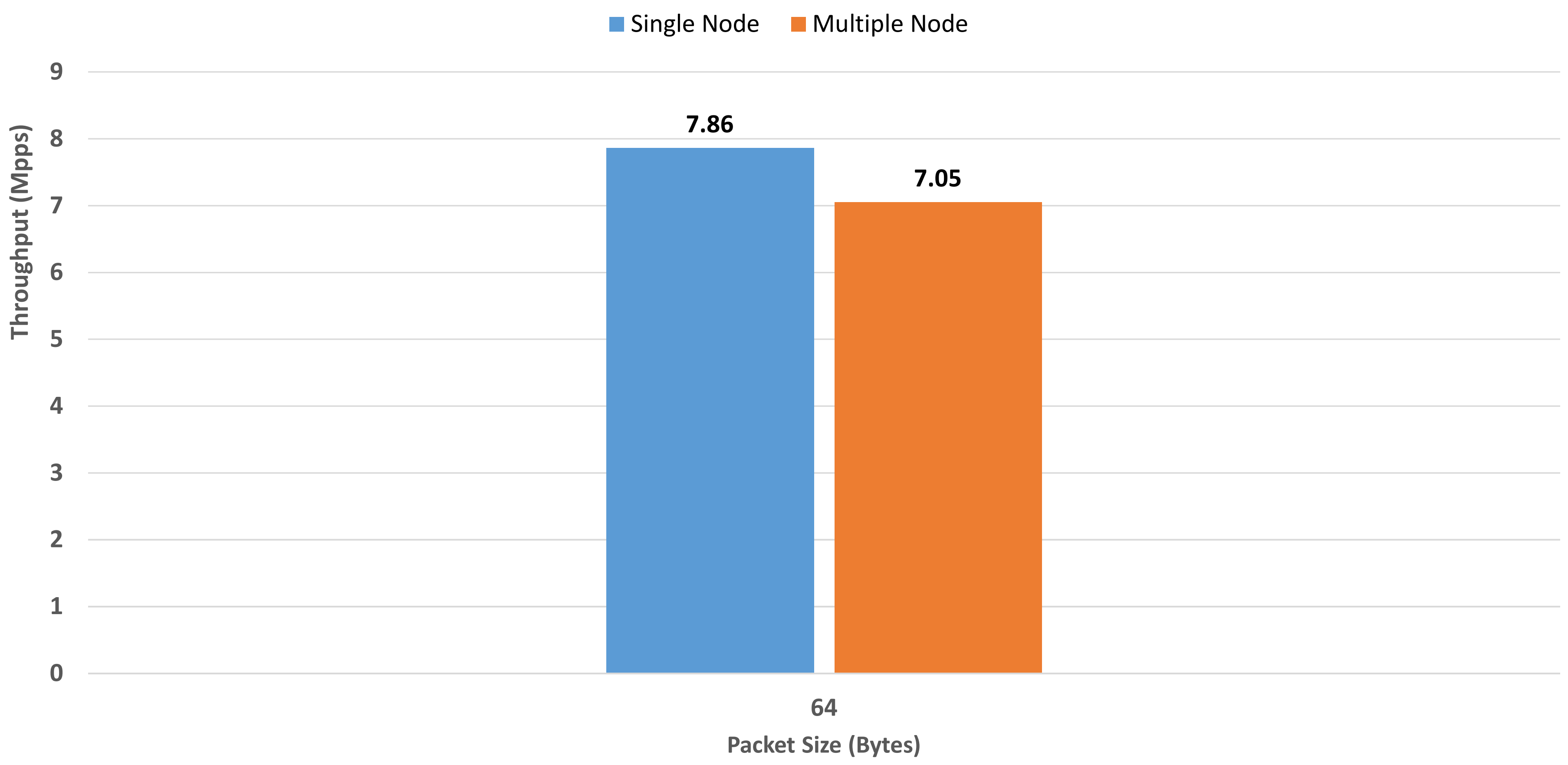
HDD: 1TB 7.2K RPM NLSAS 6Gbps

Benchmark Application

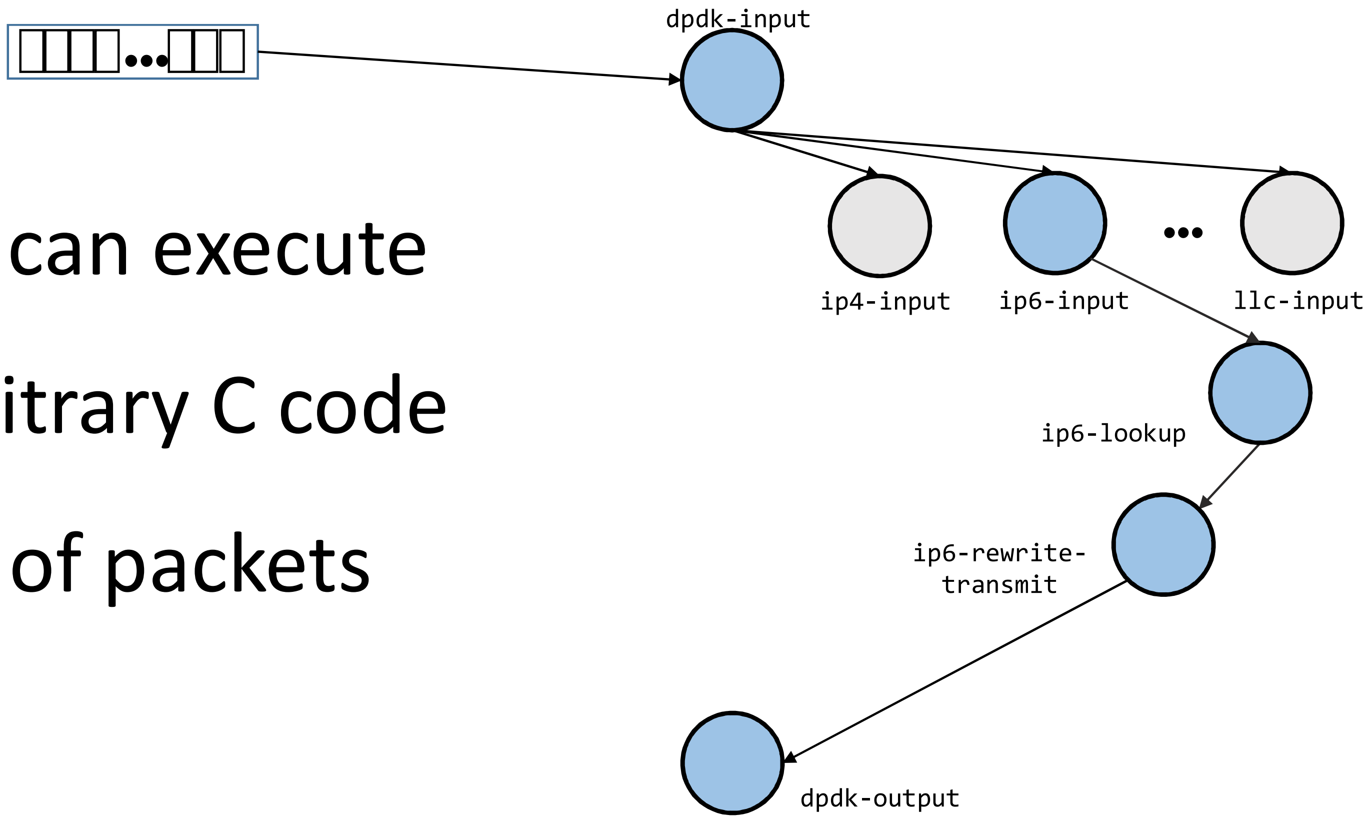


Baseline Performance

64 byte packets, single 10G port



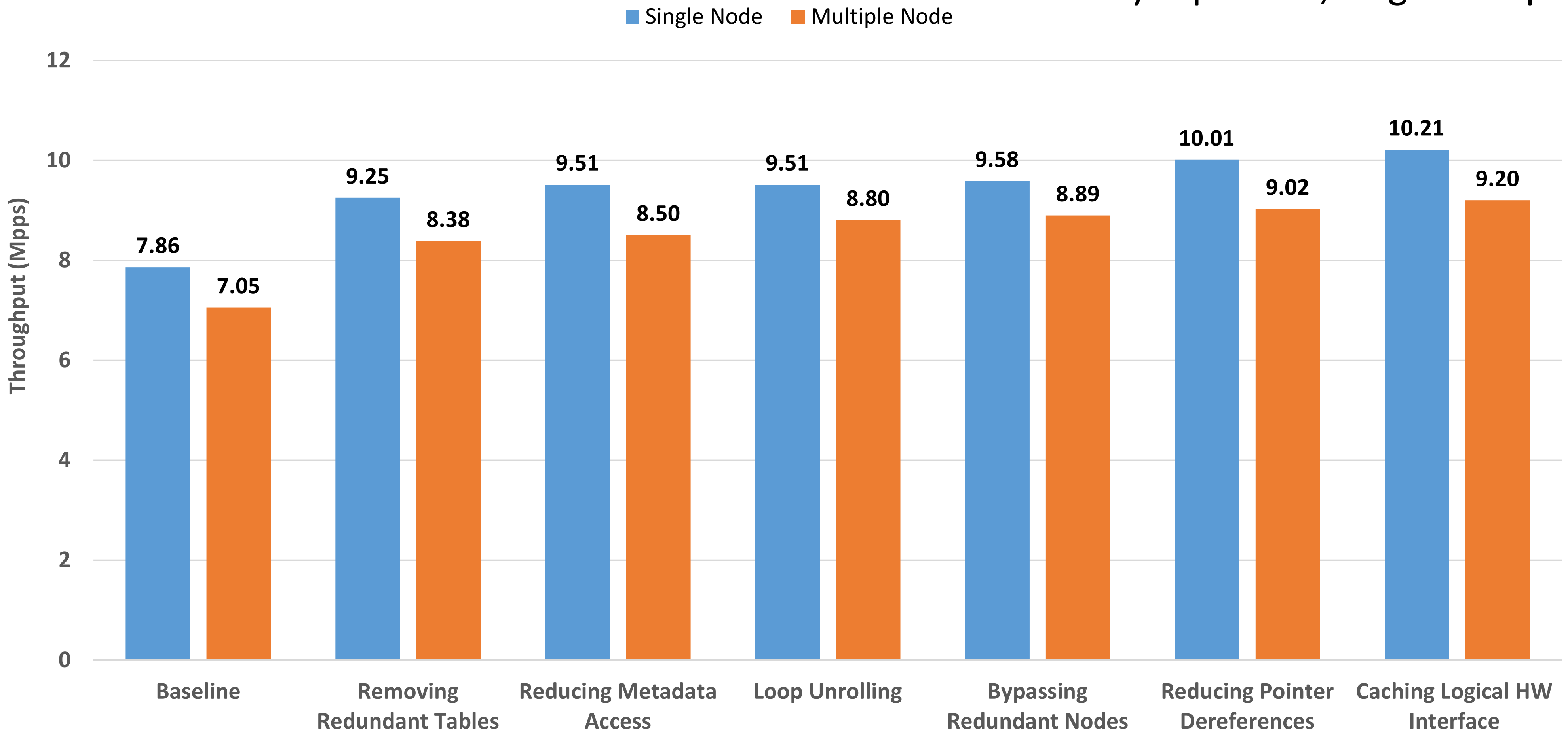
Vector Packet Processing (VPP) Platform



- Each node can execute almost arbitrary C code on vectors of packets

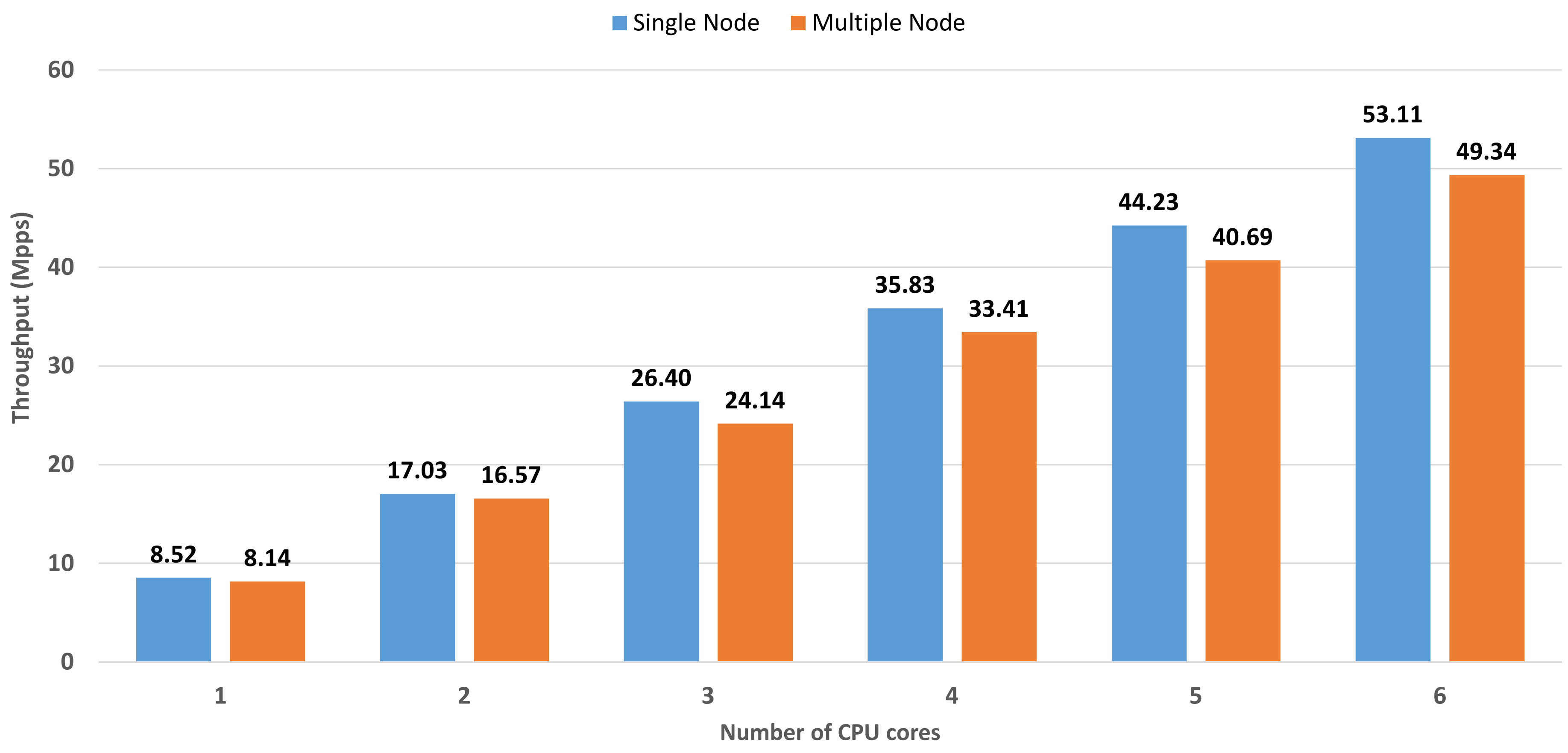
Optimized Performance

64 byte packets, single 10G port

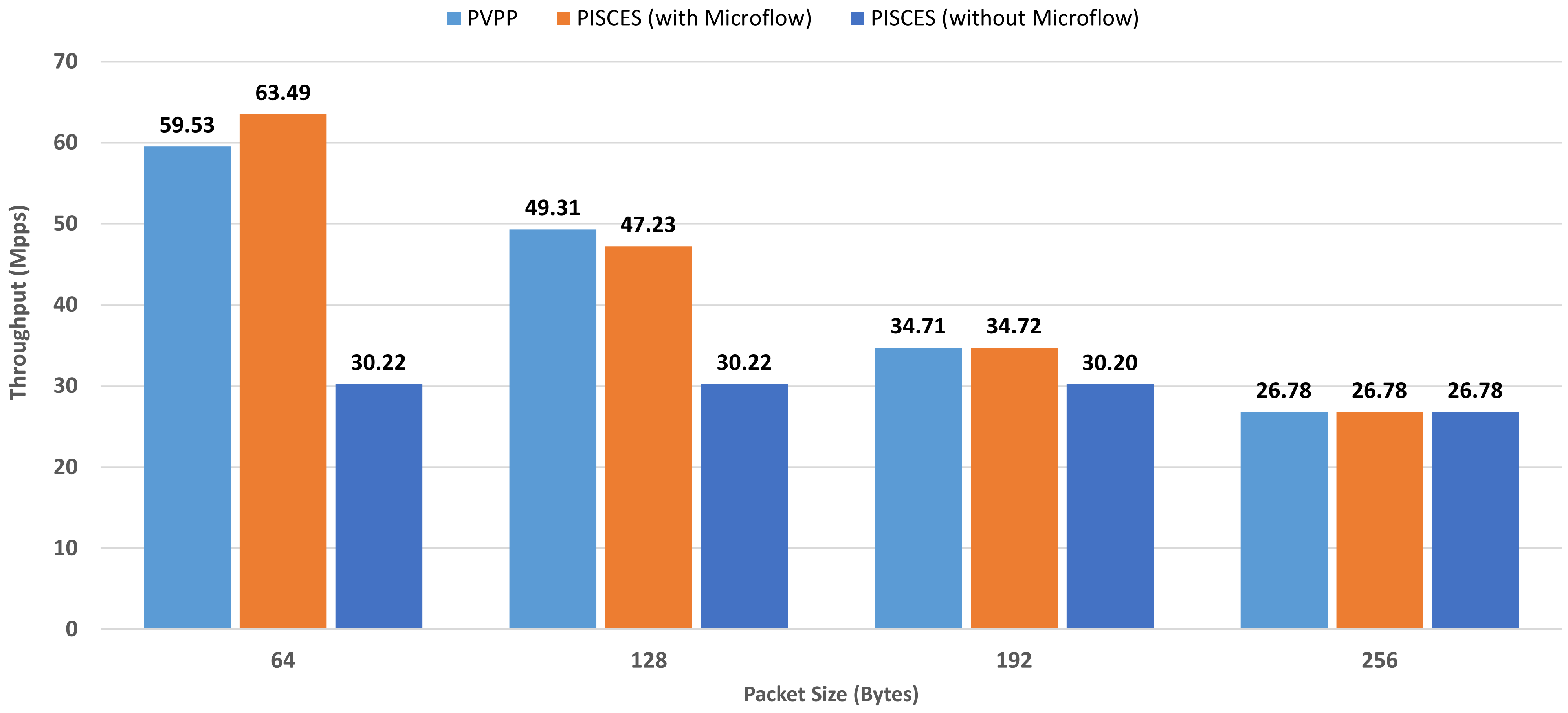


Scalability

64 byte packets across 3 x 10G ports



Performance Comparison



Future work

- Microbenchmarking VPP to inform VPP-specific optimizations
- P4 compiler annotations for low-level constructs
- Explore when multi-node compilation is beneficial for PVPP
- Demonstrate use cases where OVS microflow cache is defeated – to show PVPP is just as programmable without resorting to separated fast/slow path

Summary

- High-level DSLs are great for programmers of software switches, but lack expressivity for optimizations.
- Low-level software switches such as VPP are performant but hard to program.
- We propose that best of both is possible with PVPP.
- Comparable to state-of-art performance achieved but still work in progress.