Virtual Acceleration Engine

Michael J. Miller
CTO

Virtualization

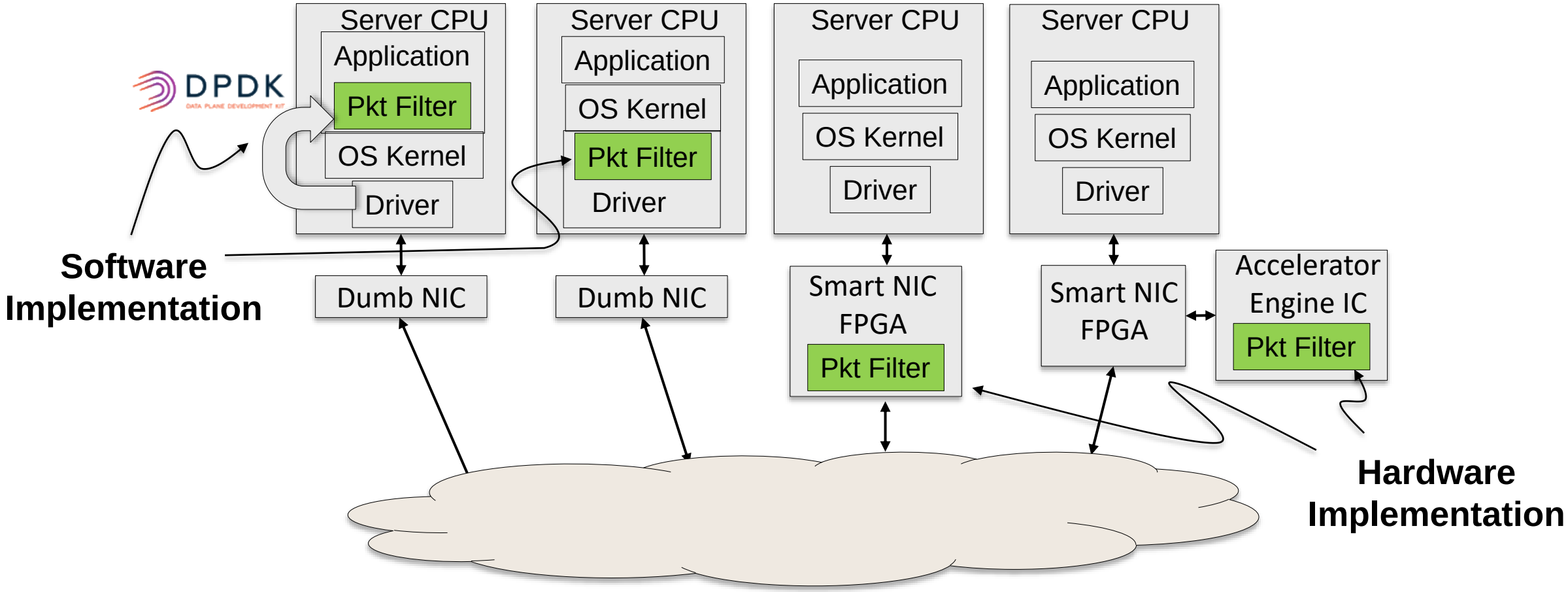
- **Virtualization is a replication of hardware resources via software**
- **Virtualized resources ease deployment and improve HW utilization in datacenters and across networks**
  - Compute, storage, networking, applications
- **First there was time sharing of CPUs & OS**
  - DEC, CDC, IBM, Unix, Tymeshare, VMWare, ...
- **Then storage**
  - EMC, NFS, iCloud, SharePoint, DropBox, Box, …
- **Next was networking**
  - VPN, VLAN, NFV, OpenFlow, SDN, OVS, …
- **Now applications**
  - Containers, Open Virtual Formats (OVF), …..
- **Flexibility is great, BUT it comes at a cost!**
  - Virtualizing can impact HW efficiency, throughput, latency and power
  - Opportunity for flexible hardware acceleration

Hardware Acceleration Addresses Bottlenecks, However…

- **Hardware accelerators speed up execution of well-defined tasks**
  - Overcome compute, memory latency and bandwidth challenges
  - Move the data and compute resources closer together
  - Heterogeneous solutions with GPU and FPGA in datacenters today
- **Today's challenges**
  - SW that is dependent on HW acceleration is limited in portability
  - CPU & DRAM perform poorly accessing random unstructured data
  - Getting the data to the HW adds overhead and latency
  - Hardware accelerators are often an after thought ➔ less optimal performance
- **Today's solutions**
  - High level: OpenCL on FPGAs and GPUs, CUDA on GPUs, etc.
  - Very narrow: TCP Offload Engines
- **What about accelerating other <u>embedded</u> tasks?**
  - Embedded data search (network addresses), packet classification, data analytics, anomaly detection, flow tracking analysis, security analysis, etc.

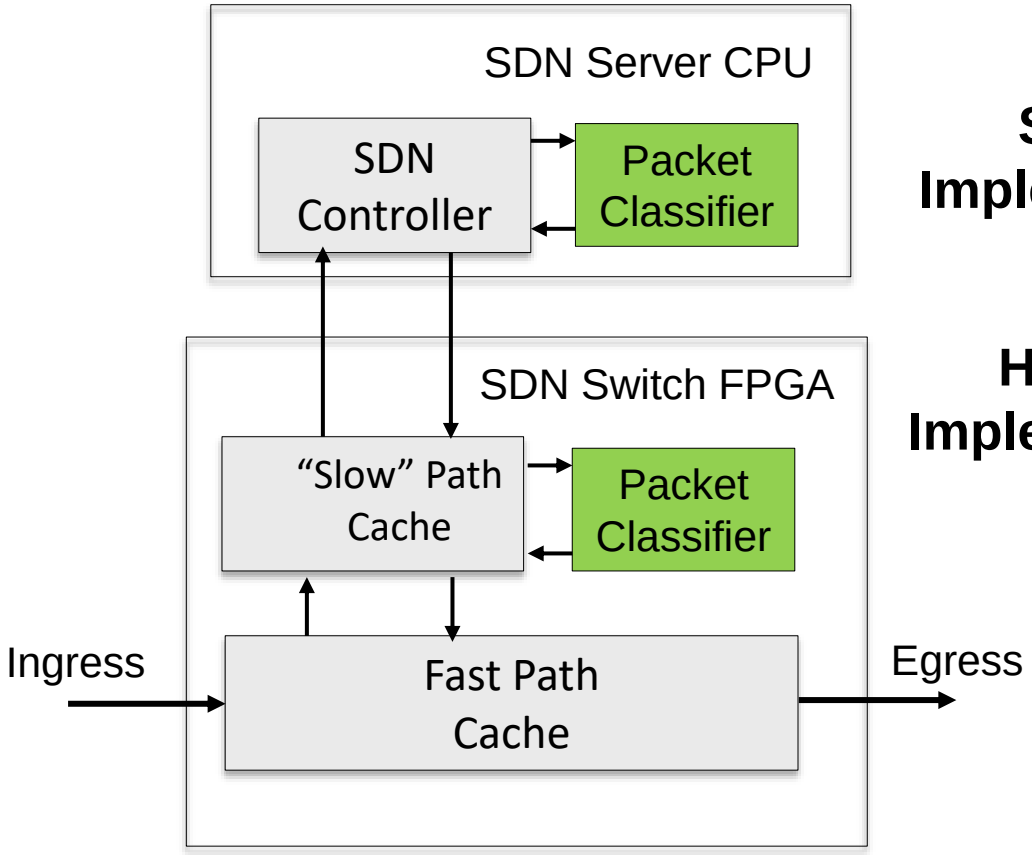
Embedded Packet Filtering Function in Various Network Interface Stacks

- **Same virtualized Packet Filter function everywhere**
  - Very desirable to have unified control software across all platforms
  - Provides for different cost, performance and capacity



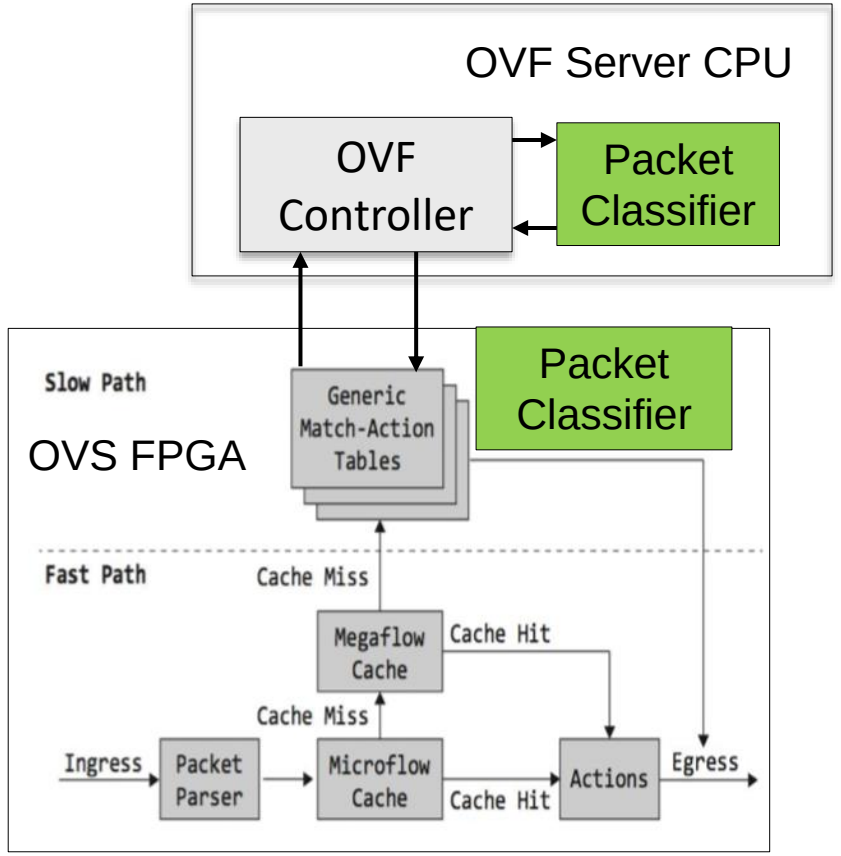
Embedded Packet Classifier Function in Hierarchical Switch Architectures

- **Same virtualized packet classifier function and API at each level**
  - Different latency, throughput and capacity
  - Minimizes total cost of development/ownership to manage all levels



Software Implementations

Hardware Implementations

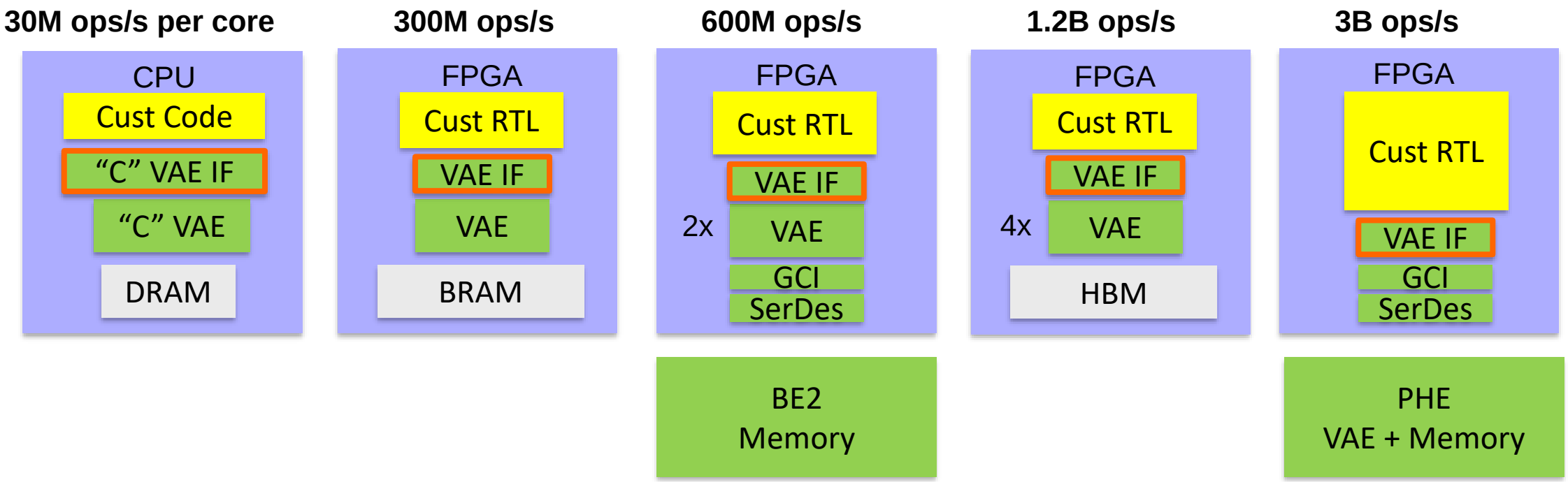


Definition: Virtual Accelerator Engine (VAE)

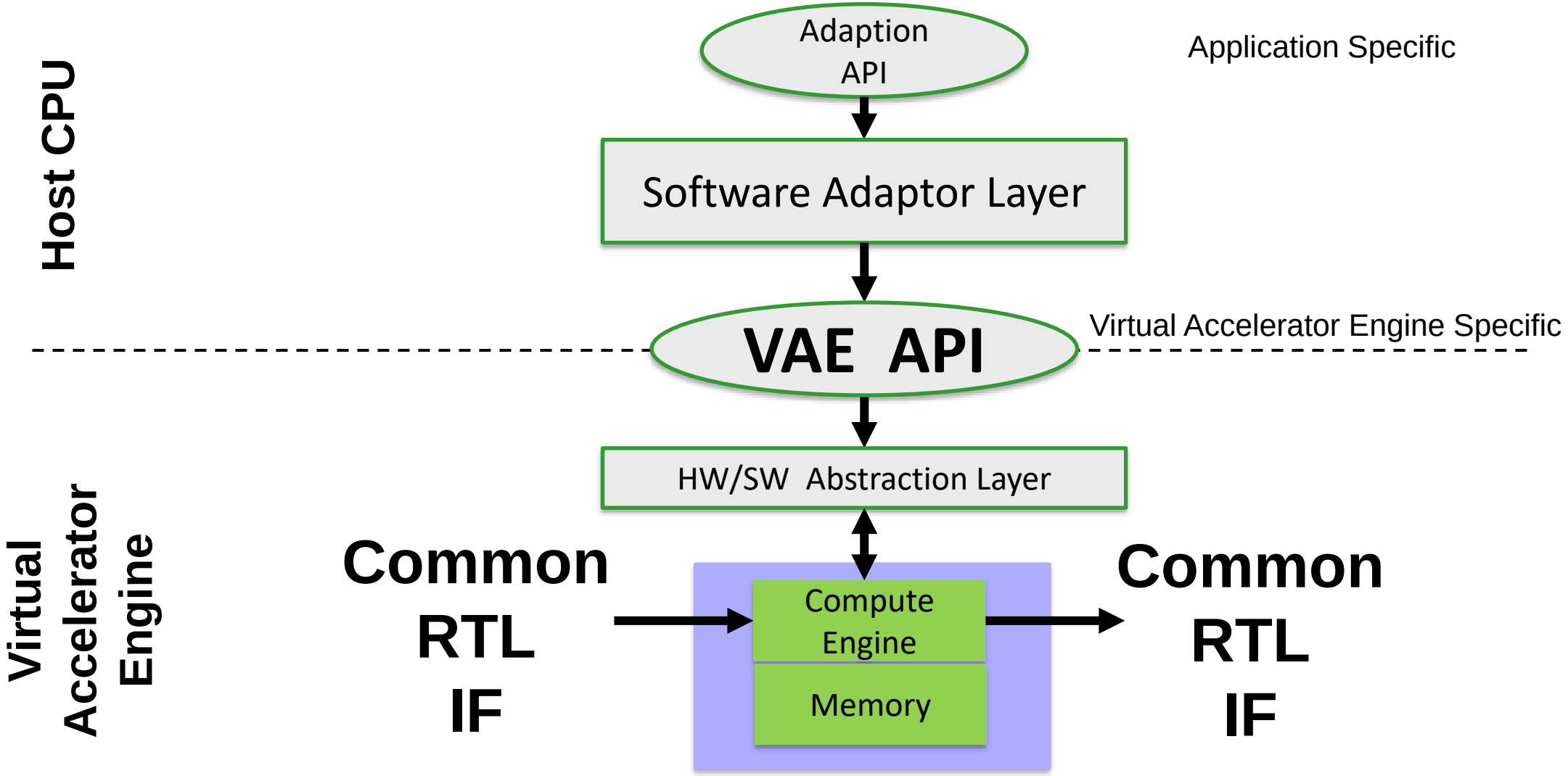
- **Supports trend towards “virtual everything”**
  - Open Virtual Switch, Virtual Machines, Network Virtual Function, SDN
- **Abstract Virtual Function**
  - Defined precisely at a functional level
  - Given the same inputs, the same results will be produced
  - Transparent to implementation
  - Maps well into hardware
  - Embeddable (makes use of natural boundaries)
- **Common API and RTL Module Interfaces**
  - API provided as a software library model
  - Same across all implementations of the VAE
  - Supports adaptation layers to existing higher level application code API
- **Scalable implementation**
  - Lowest level performance & highest capacity: software on CPU core
  - Highest performance with FPGA (or ASIC) plus MoSys Si
  - Can be ported to future or alternative hardware solutions

Virtual Accelerator Engine Scalability
Software Programmable, Hardware Performance

- **VAE example opportunities in:**
  - Matching, filtering, classification, sorting, searching, structured computation
- **Scalable across many platforms**
  - From “C” → FPGA IP core w/BRAM → FPGA w/BE2 → FPGA w/PHE
  - Same high level software interface across all platforms
  - Same RTL Module Interface for FPGA and ASICs

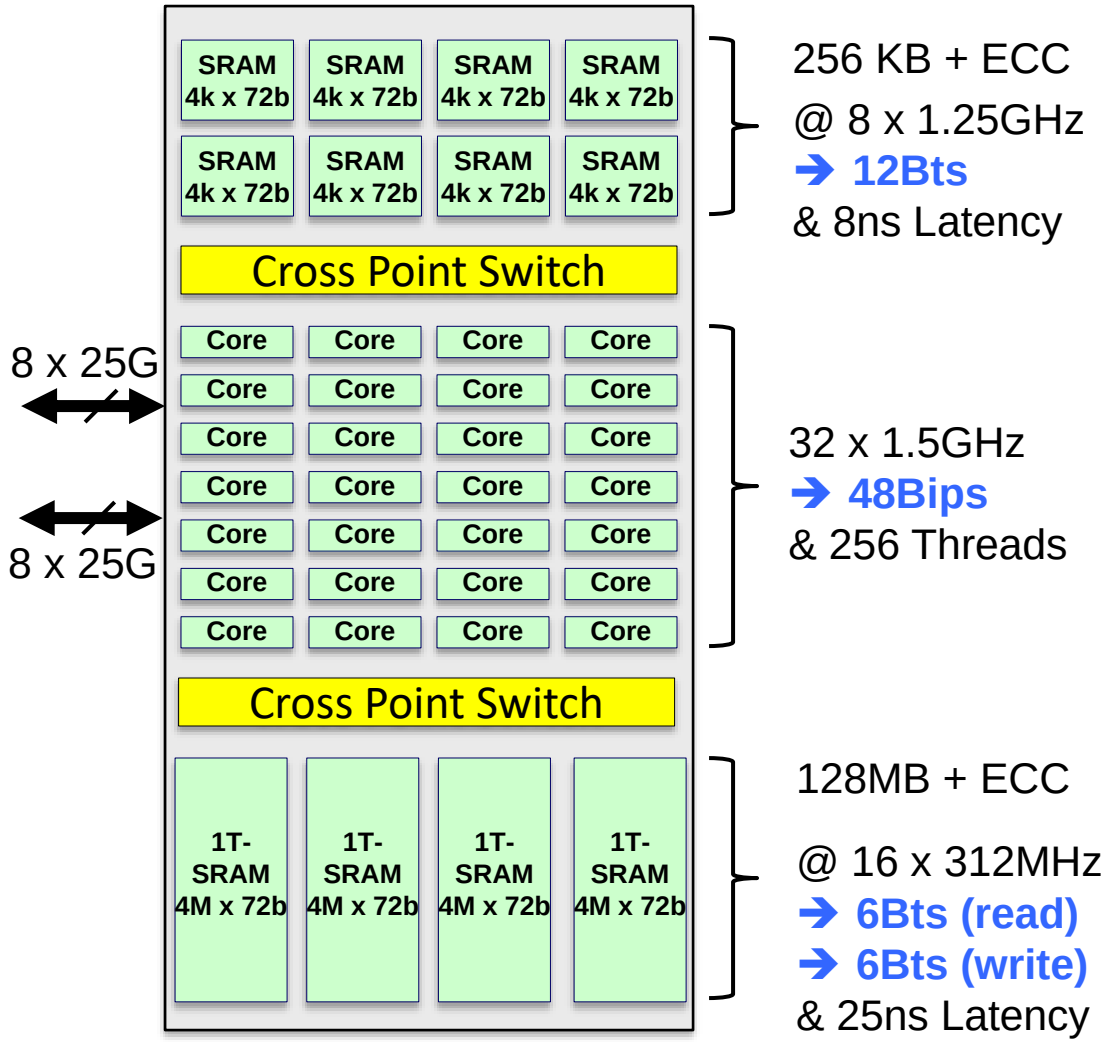


Control Stack for Virtual Accelerator Engine



MoSys Programmable HyperSpeed Engine (PHE) Device Accelerating Algorithms + Data Structures

- **Monolithic Silicon**
  - High density 1T-SRAM 128MB + ECC
  - 32 high speed RISC cores
- **Tightly coupled cores & memory**
  - Direct connect via cross point Switch
  - No cache or TLB ➔ no miss variability
- **Optimized Instruction Set**
  - Hash, Compressed Trie etc.
  - Packed bit fields
  - 24b x 24b Multiplier
- **High parallelism**
  - Up to 8 way thread cores
  - 8 way threaded SRAM
  - 16 way Thread 1T-SRAM
- **Low latency memory access**
  - 6ns to 25ns
  - Up to 4x faster than DRAM
  - 2 Level hierarchy possible

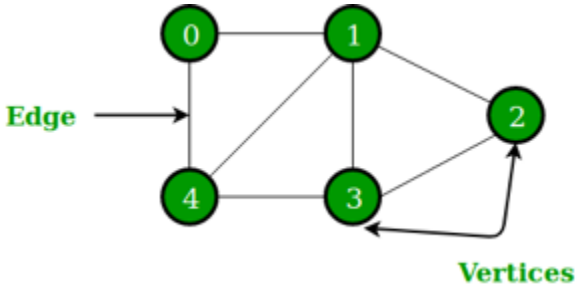


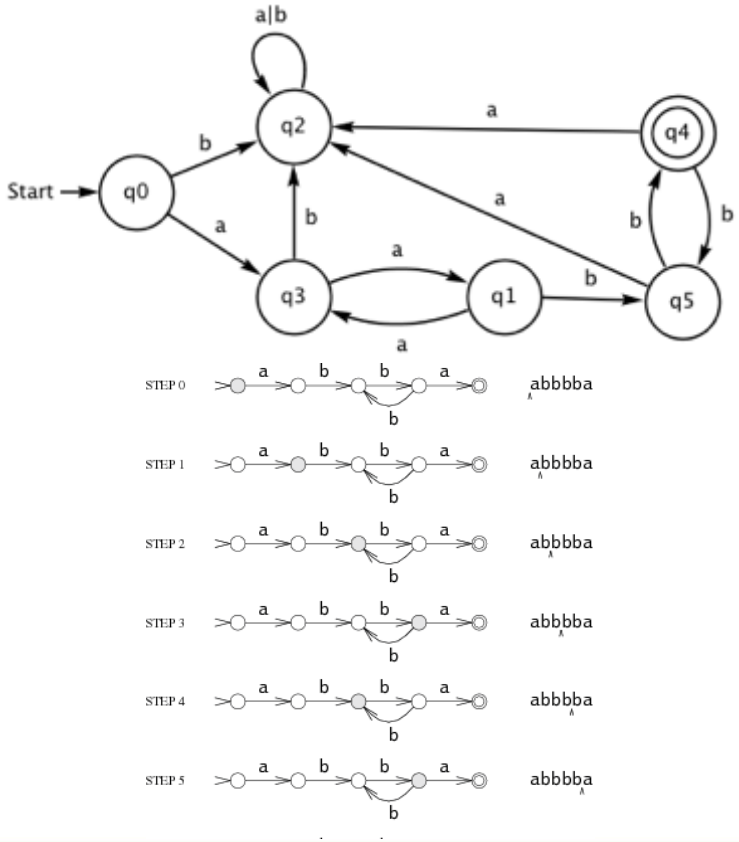
Benefits of VAE

- **Provides a wide range of performance/capacity: 100 to 1 possible**
  - Enables a wider range of product SKUs
  - Easier adoption of HW acceleration
  - Migration path for the future Si and Hardware
- **Reduced time to deployment of new features**
  - Software definable function without the cost of RTL design timelines
- **High level platform portability**
  - Higher layers of code are not limited to specific hardware
  - Use available HW VAE or software version of VAE
  - Enables “graceful fall back”
- **SW investment is protected**
  - Insulated from hardware implementation by common API
- **Programmers can take advantage of HW without knowing details**
  - Today’s software engineers are are focused on a higher level than firmware or RTL
  - Allows programmers to focus on the bigger picture

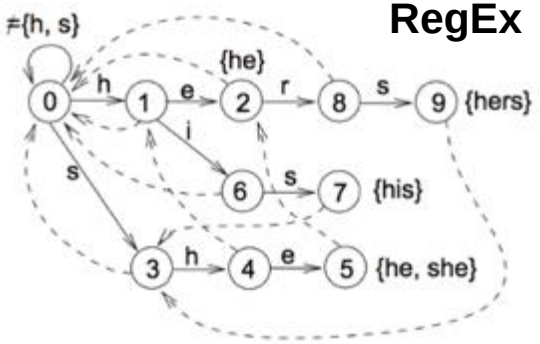
Virtualized Graph Memory Engine

Abstract Graphs Can Be Used For Many Problems



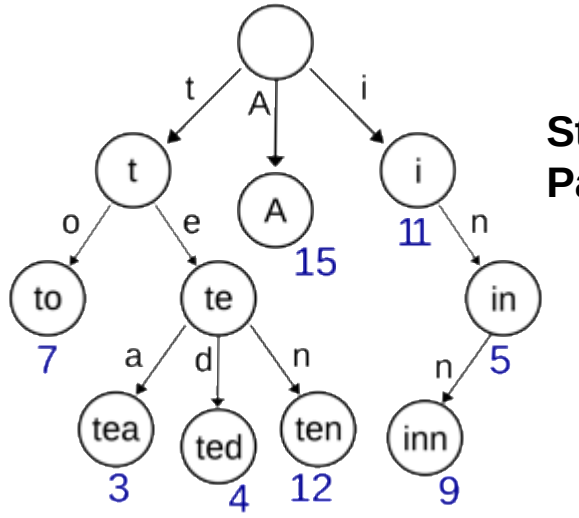


RegEx



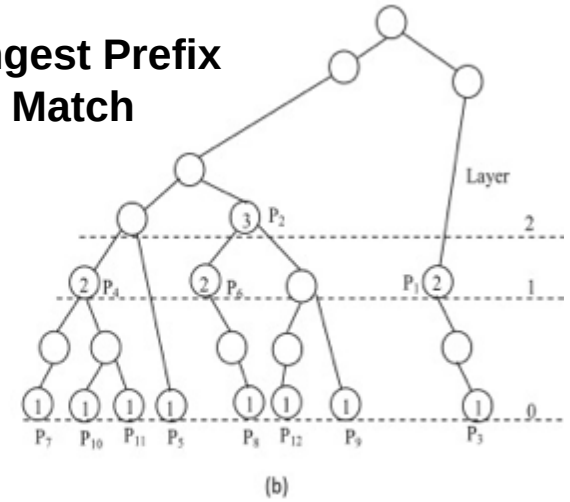
ID	Prefix	Nexthop
P_1	10*	NH_1
P_2	001*	NH_2
P_3	1011*	NH_3
P_4	0000*	NH_4
P_5	0001*	NH_5
P_6	0010*	NH_6
P_7	000000	NH_7
P_8	001011	NH_8
P_9	00111*	NH_9
P_{10}	000010	NH_{10}
P_{11}	000011	NH_{11}
P_{12}	001100	NH_{12}

(a)



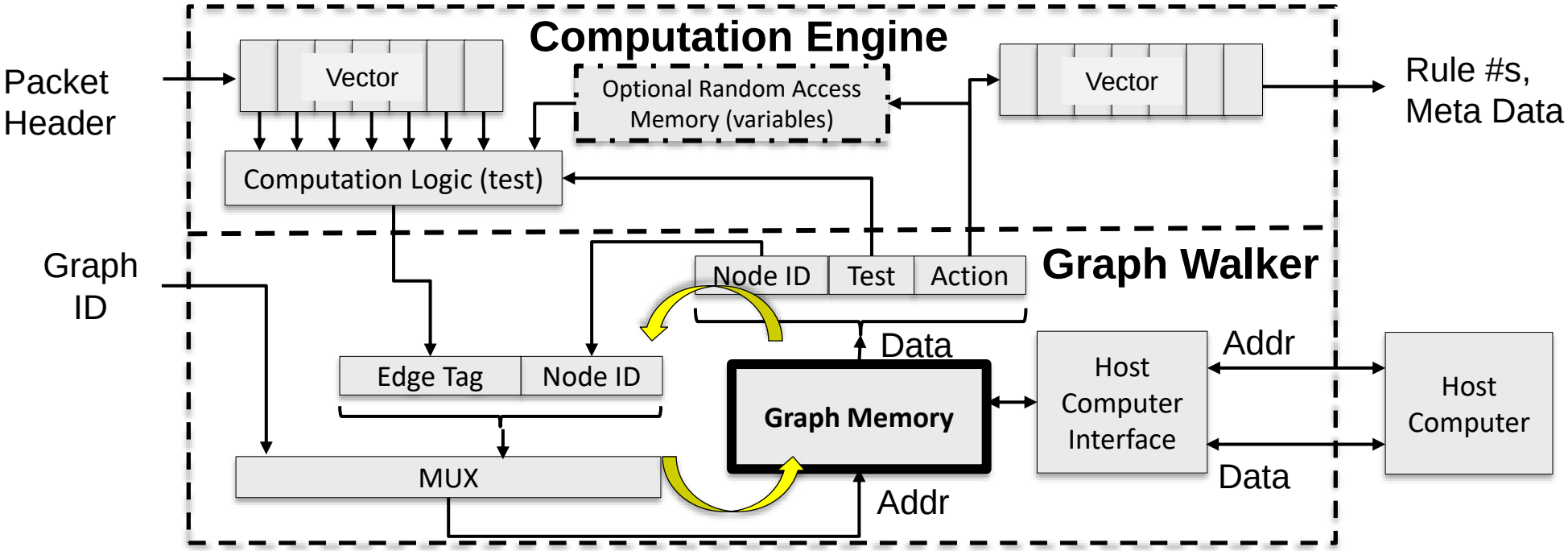
String Match Patricia Tree

Longest Prefix Match



Virtualized Graph Memory Engine Block Diagram

- **Graph Memory Engine is composed of:**
  - **Graph Walker** including the Graph Memory
  - **Computation Engine** which computes the next Edge as a function of input vector



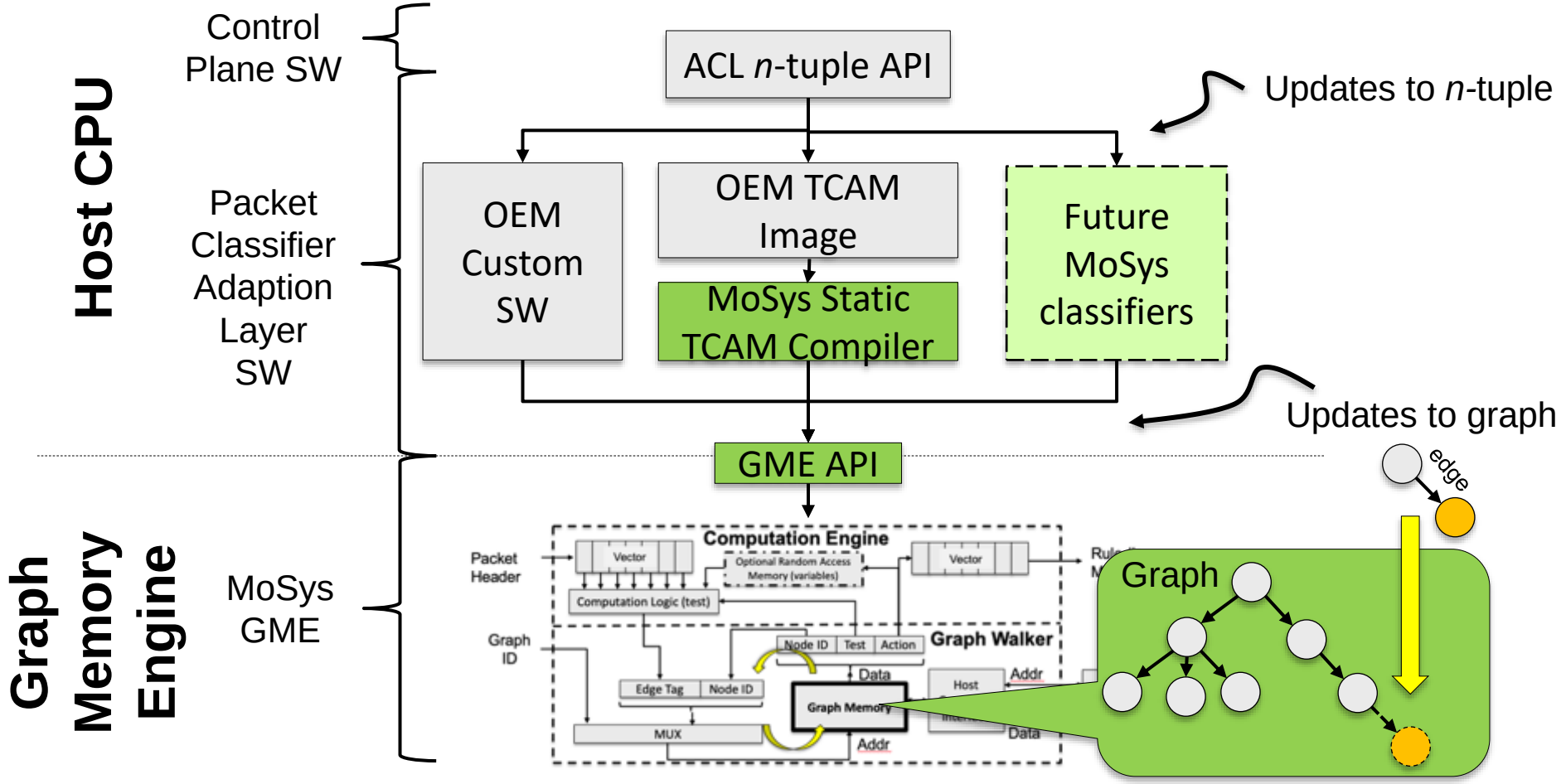
Graph Memory Engine API

- Edges connect two nodes (n, m) with a unique edge value for a given node n
- Multiple graphs can be stored in the memory

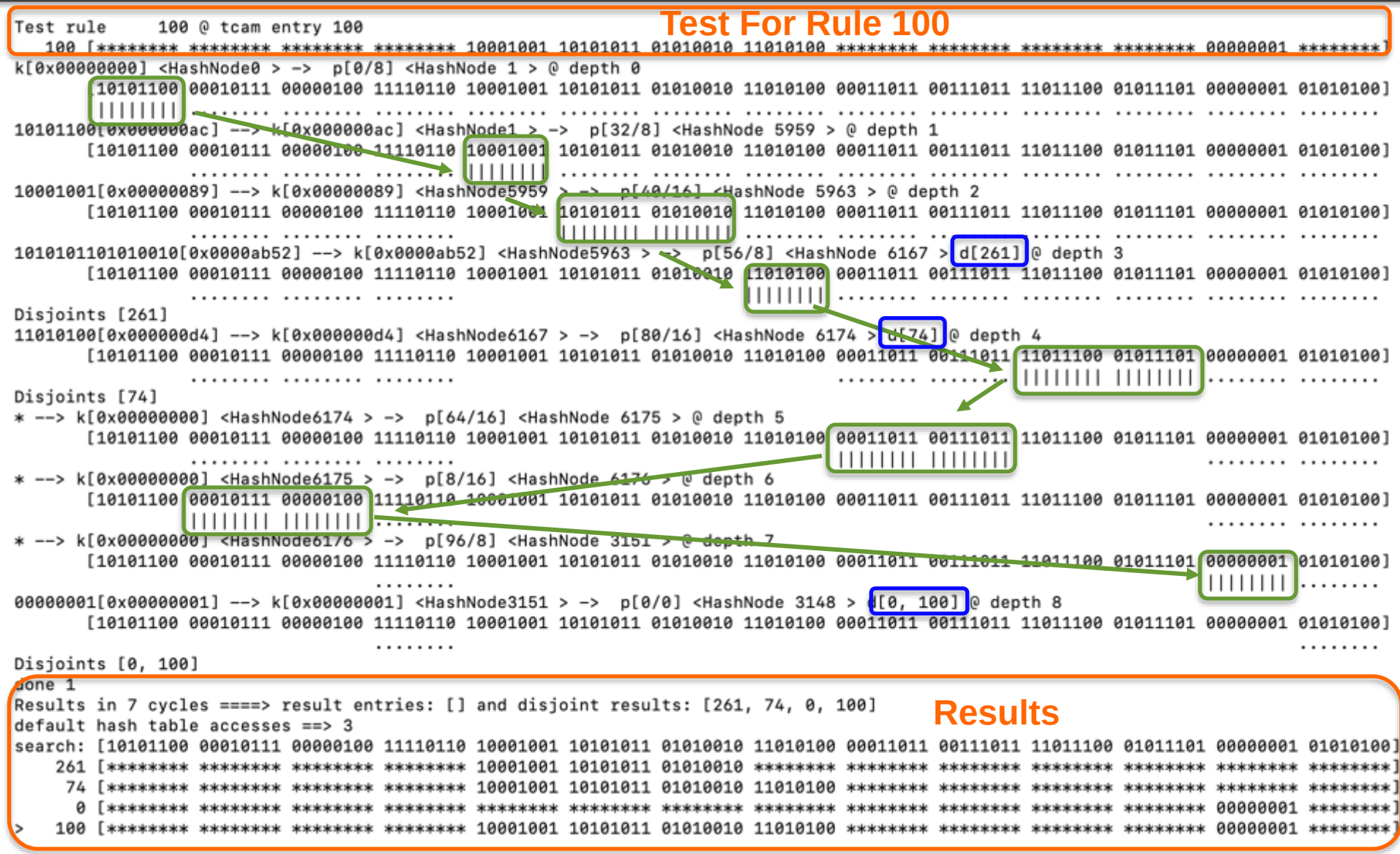
```
          add_node(n, comp_edge_operation)   // add node n with next action
          add_edge(n, m, edge_value)         // add an edge from n to m
          add_default_edge(n, m)             // add a default edge from n to m
          has_node(n)                        // test if node n exists
          has_edge(n, edge_value)            // test if there is an edge from n
          adjacent(n, m)                     // test if n and m are connected
  list  = neighbors(n)                       // list of adjacent edges to n
  list  = edge_values(n)                     // list of all edge values
  func  = comp_edge_function(n)              // returns compute edge function
action  = action(n, edge_value)              // returns action associated with edge
action  = default_action(n)                  // returns action for default edge
          delete_edge(n, edge_value)
          delete_node(n)
```

GME Packet Classifier Platform

- **MoSys Packet Classifier Platform utilizes an Adaption Layer**
  - Various Adaption Layers for building and maintaining search graphs
  - Allows for options in function, throughput, latency and capacity

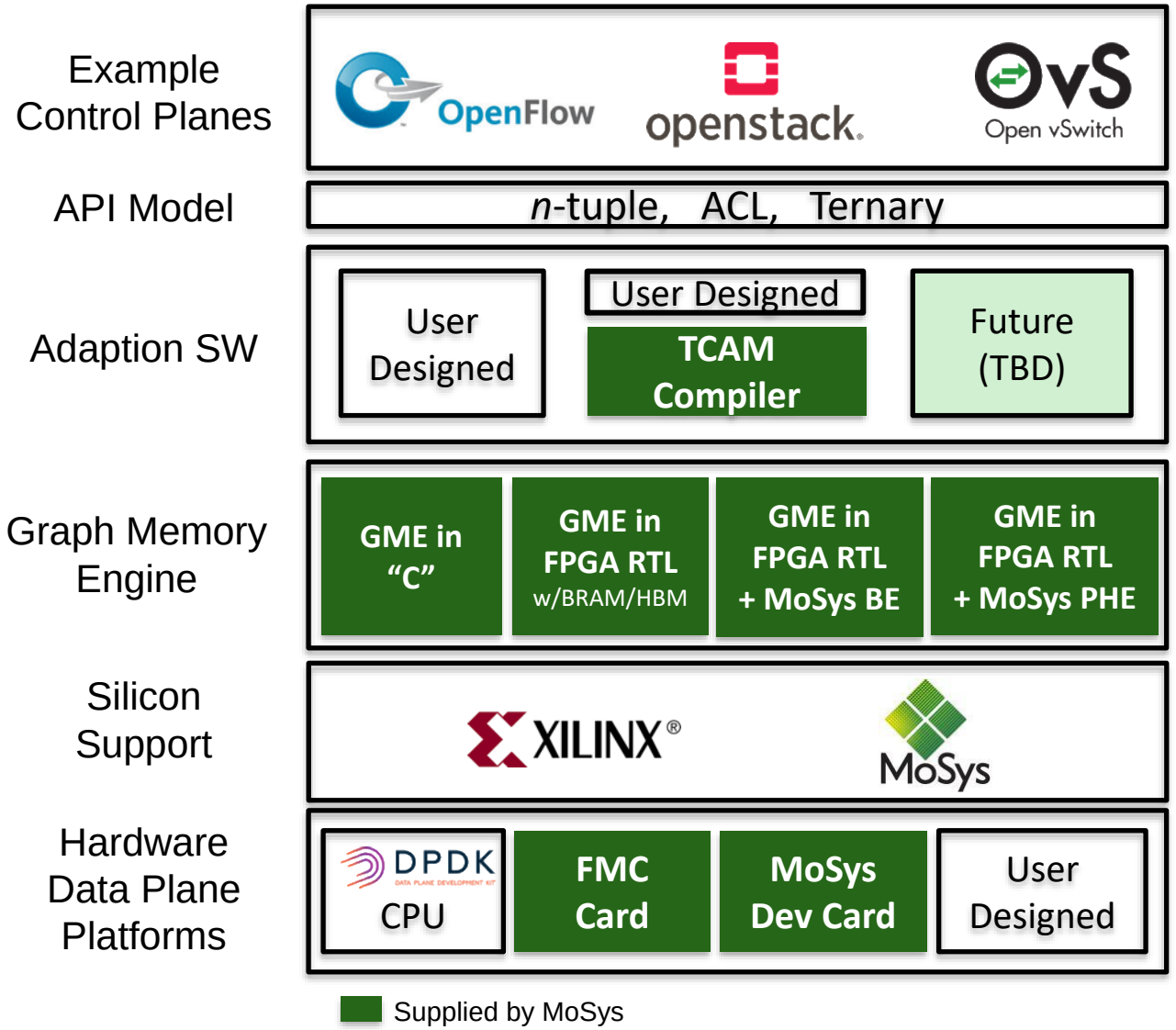


Example Multi-Field Match




```
Test rule     100 @ tcam entry 100
   100 [******** ******** ******** ******** 10001001 10101011 01010010 11010100 ******** ******** ******** ******** 00000001 ********]
k[0x00000000] <HashNode0 > ->  p[0/8] <HashNode 1 > @ depth 0
       [10101100 00010111 00000100 11110110 10001001 10101011 01010010 11010100 00011011 00111011 11011100 01011101 00000001 01010100]
        ||||||||
10101100[0x000000ac] --> k[0x000000ac] <HashNode1 > ->  p[32/8] <HashNode 5959 > @ depth 1
       [10101100 00010111 00000100 11110110 10001001 10101011 01010010 11010100 00011011 00111011 11011100 01011101 00000001 01010100]
                          ||||||||
10001001[0x00000089] --> k[0x00000089] <HashNode5959 > ->  p[40/16] <HashNode 5963 > @ depth 2
       [10101100 00010111 00000100 11110110 10001001 10101011 01010010 11010100 00011011 00111011 11011100 01011101 00000001 01010100]
                                    ||||||||
101010110101010[0x0000ab52] --> k[0x0000ab52] <HashNode5963 > ->  p[56/8] <HashNode 6167 > d[261] @ depth 3
       [10101100 00010111 00000100 11110110 10001001 10101011 01010010 11010100 00011011 00111011 11011100 01011101 00000001 01010100]
                                                ||||||||
Disjoints [261]
11010100[0x000000d4] --> k[0x000000d4] <HashNode6167 > ->  p[80/16] <HashNode 6174 > d[74] @ depth 4
       [10101100 00010111 00000100 11110110 10001001 10101011 01010010 11010100 00011011 00111011 11011100 01011101 00000001 01010100]
                                                                     ||||||||
Disjoints [74]
* --> k[0x00000000] <HashNode6174 > ->  p[64/16] <HashNode 6175 > @ depth 5
       [10101100 00010111 00000100 11110110 10001001 10101011 01010010 11010100 00011011 00111011 11011100 01011101 00000001 01010100]
                                                   ||||||||
* --> k[0x00000000] <HashNode6175 > ->  p[8/16] <HashNode 6176 > @ depth 6
       [10101100 00010111 00000100 11110110 10001001 10101011 01010010 11010100 00011011 00111011 11011100 01011101 00000001 01010100]
        ||||||||
* --> k[0x00000000] <HashNode6176 > ->  p[96/8] <HashNode 3151 > @ depth 7
       [10101100 00010111 00000100 11110110 10001001 10101011 01010010 11010100 00011011 00111011 11011100 01011101 00000001 01010100]
                                                                                                        ||||||||
00000001[0x00000001] --> k[0x00000001] <HashNode3151 > ->  p[0/0] <HashNode 3148 > d[0, 100] @ depth 8
       [10101100 00010111 00000100 11110110 10001001 10101011 01010010 11010100 00011011 00111011 11011100 01011101 00000001 01010100]
Disjoints [0, 100]
done 1
Results in 7 cycles ====> result entries: [] and disjoint results: [261, 74, 0, 100]
default hash table accesses ==> 3
search: [10101100 00010111 00000100 11110110 10001001 10101011 01010010 11010100 00011011 00111011 11011100 01011101 00000001 01010100]
    261 [******** ******** ******** ******** 10001001 10101011 01010010 ******** ******** ******** ******** ******** ******** ********]
     74 [******** ******** ******** ******** 10001001 10101011 01010010 11010100 ******** ******** ******** ******** ******** ********]
      0 [******** ******** ******** ******** ******** ******** ******** ******** ******** ******** ******** ******** 00000001 ********]
>   100 [******** ******** ******** ******** 10001001 10101011 01010010 11010100 ******** ******** ******** ******** 00000001 ********]
```

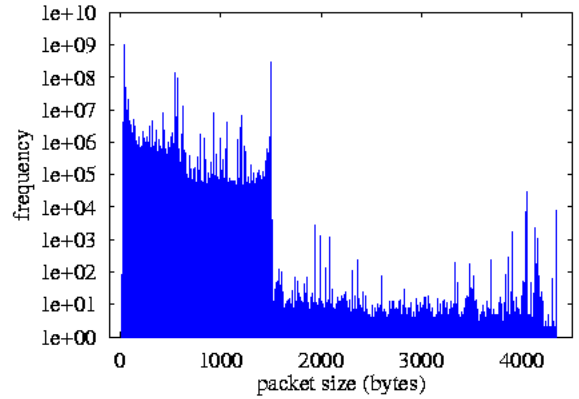
MoSys Packet Classification Platform

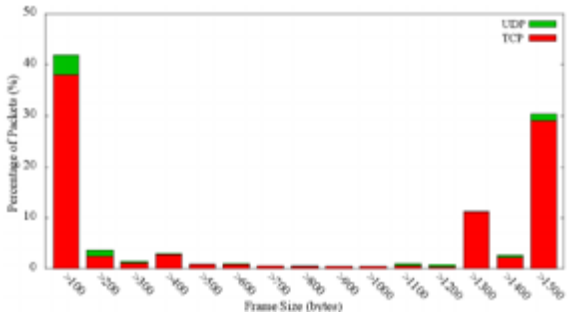


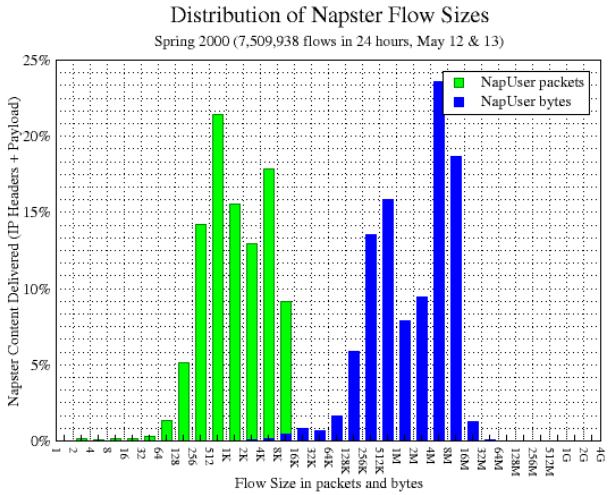
MoSys "Cheetah" Accelerator Dev Kit
2 x 100G Network Accelerator

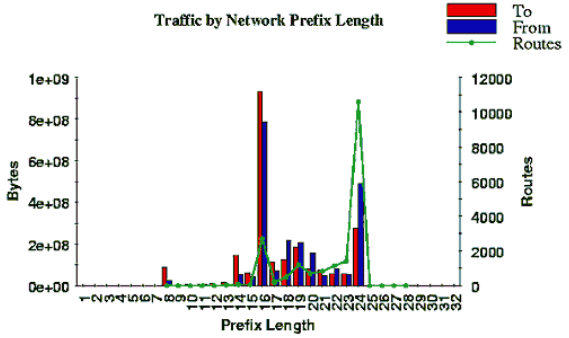
- **Clone of Xilinx VCU1525 Card**
  - XCVU9P Xilinx FPGA (other assembly options possible)
  - 4 x 16GB DDR4 DIMMS
  - 2 x QSFP28
- **MoSys PHE 1.5GHz**
  - 1 Gb of 1T-SRAM + 32 RISC cores

Future VAEs: Data Analytics?

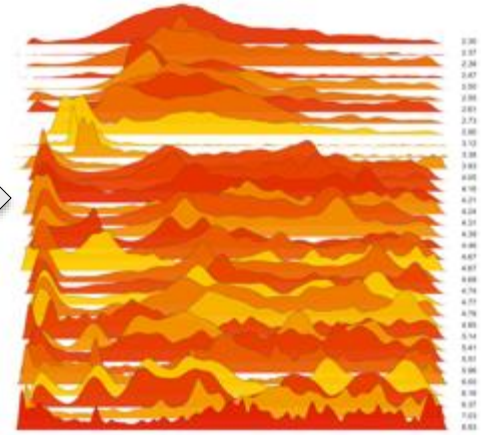








Profile Snap Shots Over Time



Shannon Entropy, ML Random Forest of Trees

Anomaly Alert

Up to 36B Features/s

Summary

- **Virtual Accelerator Engines provide:**
  - Software at hardware speeds
  - Wide range of performance/capacity: 100 to 1 possible
  - OEM platform portability
  - Preservation of software investments
  - Future proof to improvements in hardware acceleration
- **MoSys will be creating application specific platforms**
  - Using VAE technology for scalability
  - For applications that are dominated by random memory access and unstructured data
  - Require high throughput and low latency
- **Applicable for embedded functions:**
  - Searching, Filtering, Matching, Sorting, Security, Data analytics, Compute on Sparse or Random data
- **Equipment such as:**
  - Embedded Switching/Routing, Smart NICs, Security appliances, Application Severs, HPC, 5G edge, defense/aerospace, test/measurement equipment, datacenter acceleration

Thank You,

Questions?