Design Considerations for a High-Performing Virtualized LTE Core Infrastructure

Arun Rajagopal

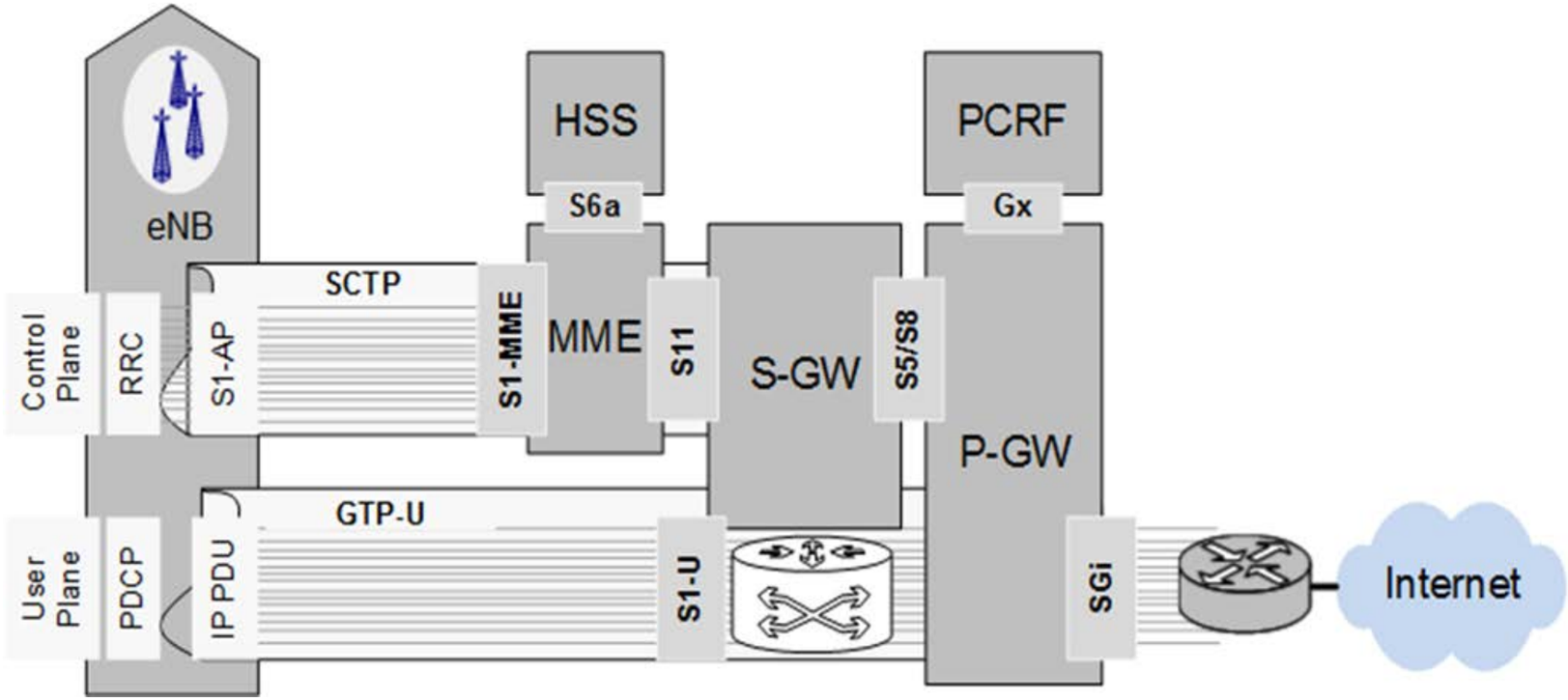
Sprint CTO Office

Sameh Gobriel

Intel Labs

#moveforward

Current EPC Network Infrastructure



#moveforward

Carrier Business Problems

Rigid capacity models lead to inefficient utilization of network resources

Capacity added when one dimension exhausts (e.g., signaling *vs.* bearer capacity on SBC)

Difficult to align service revenue with costs (e.g., low volume M2M)

No means to re-use stranded capacity on platforms

Long time-to-market intervals for new products/services

Long service development processes with limited service agility

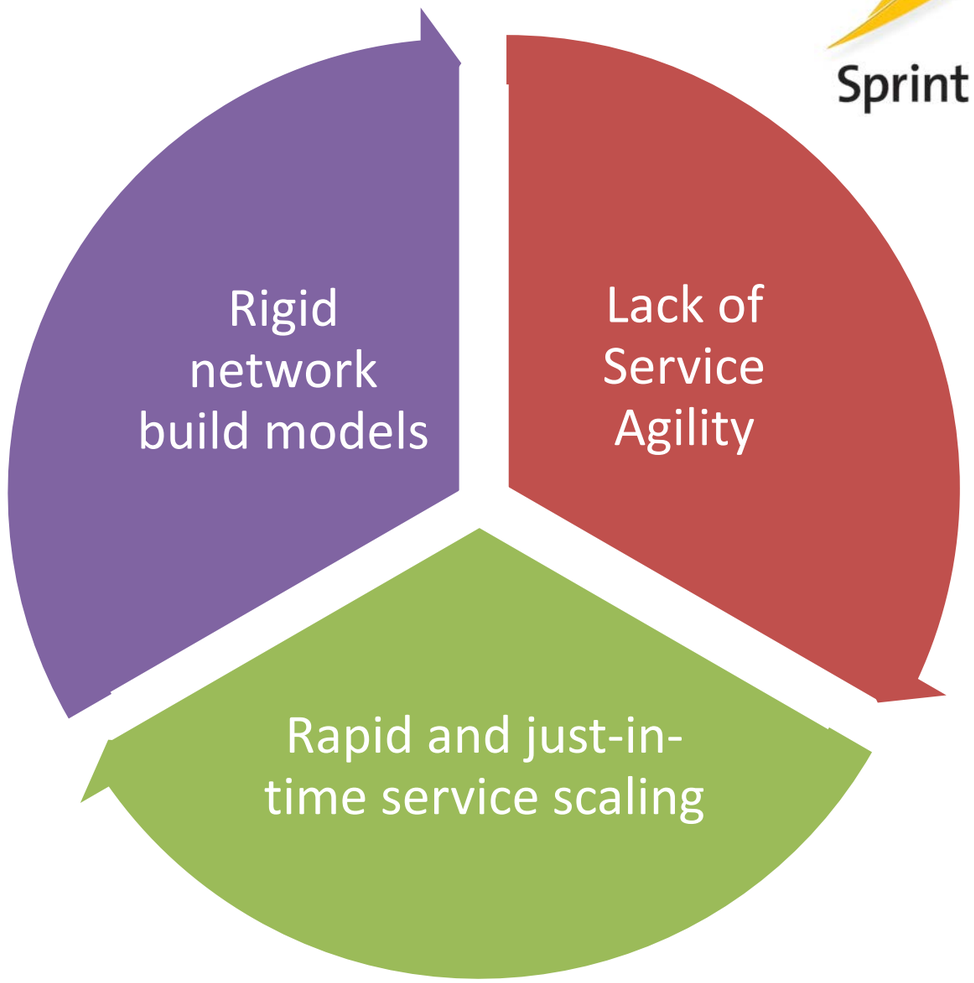
Limited fast fail opportunities and platform re-usability

Rapid service scaling is a challenge

Adding new capacity to existing services takes time

Managing scale by adding additional hardware and using load balancing mechanisms is complex

More nodes/elements to manage as the function scales



#moveforward

The Case for NFV

Simplifies Network Architecture

- Common hardware
- Independent scaling of components
- Standard and repeatable configurations

Lower Capex

Simplifies Network Operations

- Just-in-time allocation
- Automated deployment
- Automated capacity add
- Agile, high velocity service creation environment

Lower Opex

Creates New Revenue Opportunities

- Combine Mobility and call control with cloud technologies
- Monetize network based on service value

Higher Revenue

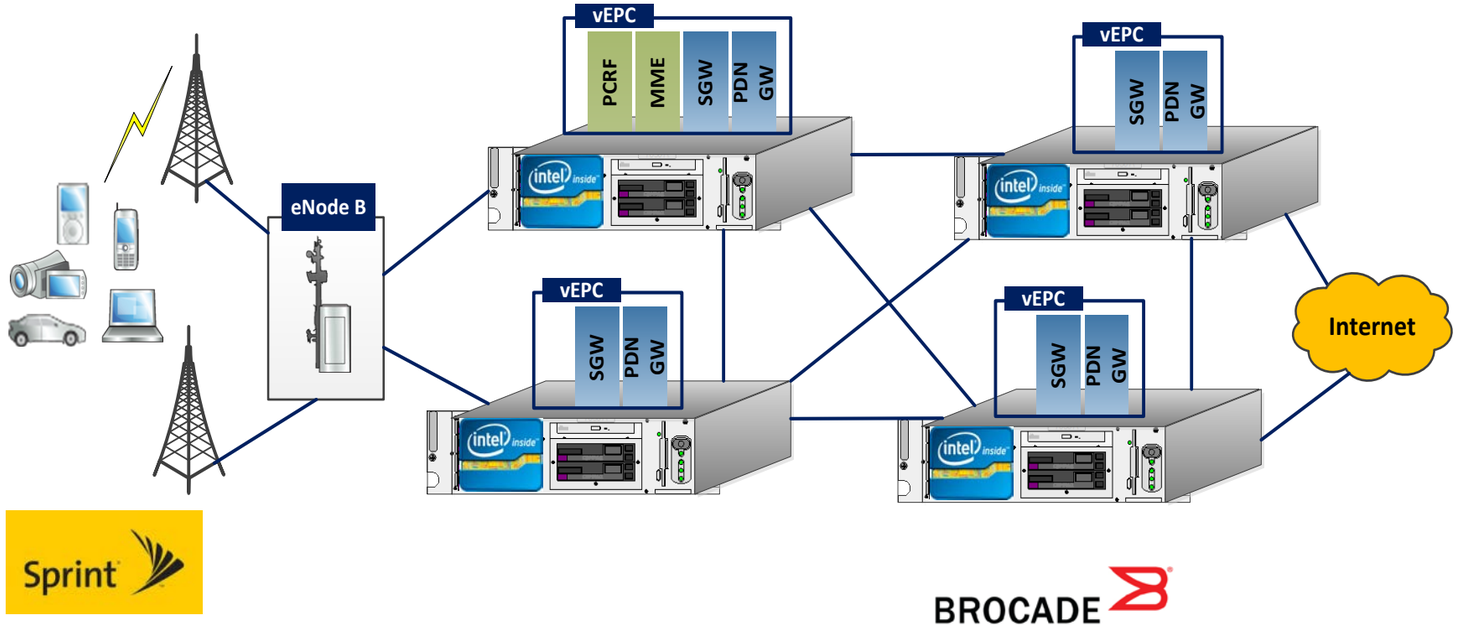
#moveforward

From Purpose Built ASICs to General Purpose IA

How to build a scalable EPC cluster on IA servers?

- Fully programmable control & data planes
- Incrementally scalable as needed by adding nodes to the cluster
- S/P GW ported as DPDK Apps on top of IA Cluster.
- Leverages multi-core/socket, DDIO, SSE instructions, ..etc.

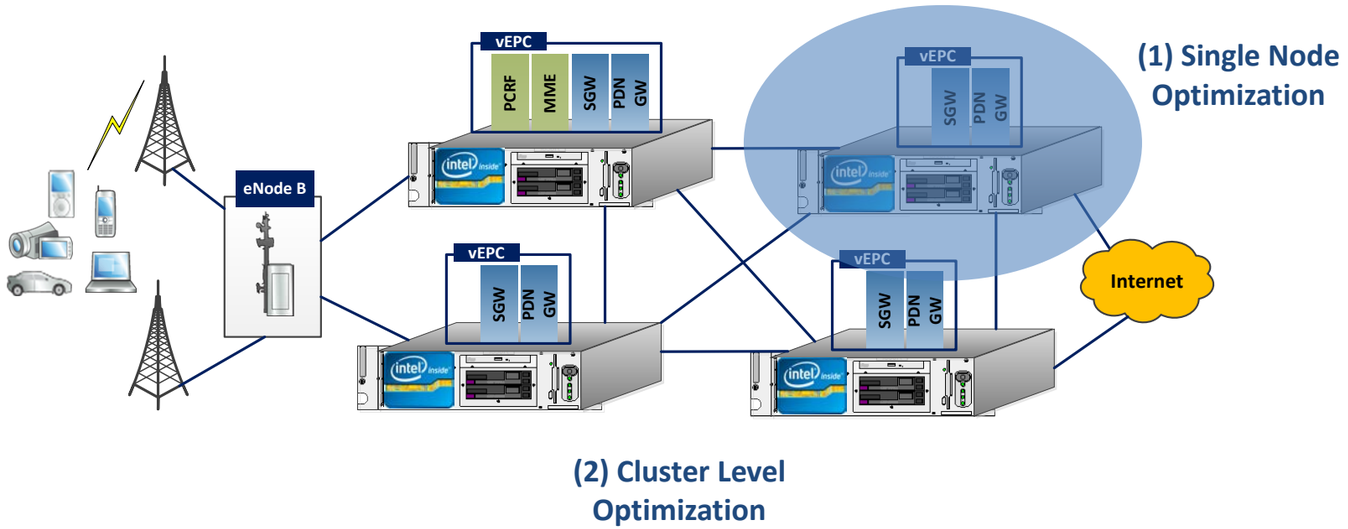
A first step towards a flexible network infrastructure



#moveforward

Flow Table Size and Packet Classification Bottleneck

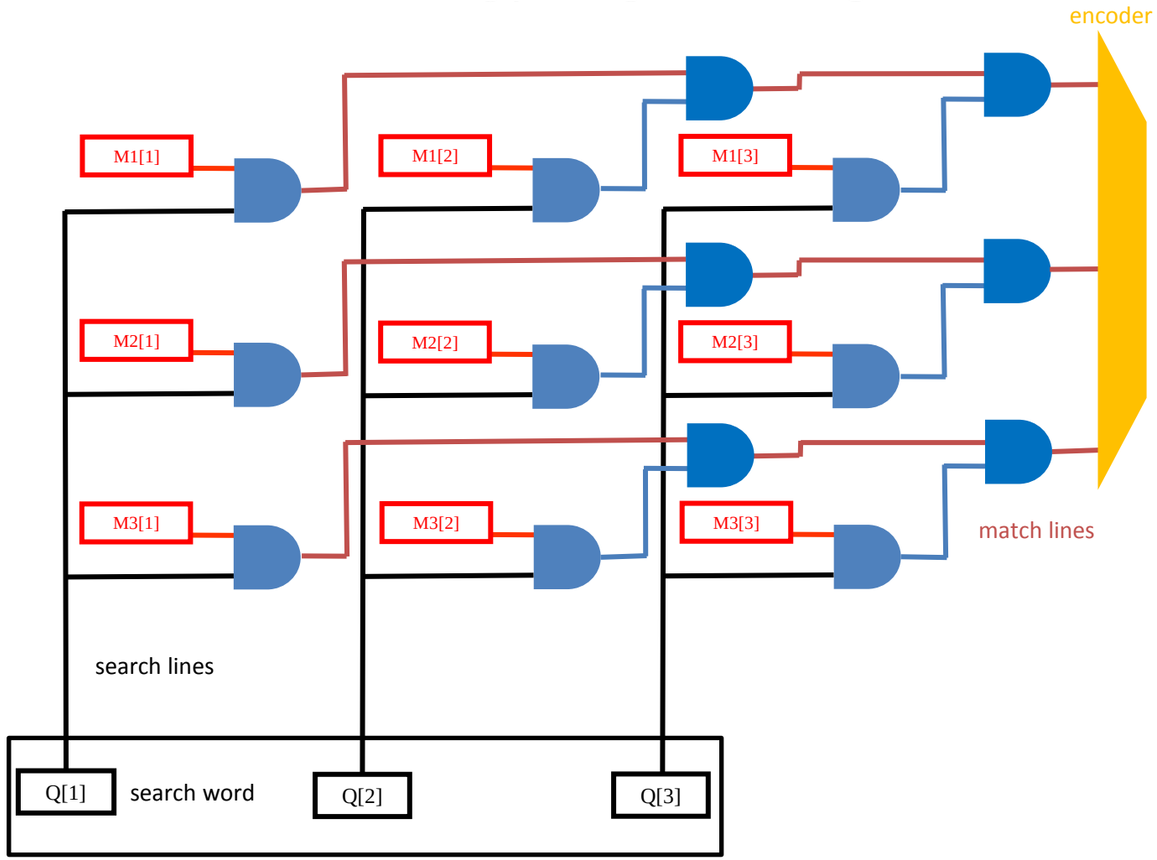
- EPC SGW session table size grow significantly (millions of entries) with the number of subscribers/bearers/flows.
- Flow lookup and Packet classification is common for many VNFs.
- Distributed flow table as a single entity to control/management plane.



#moveforward

Flow Lookup & Classification Bottleneck for NFV

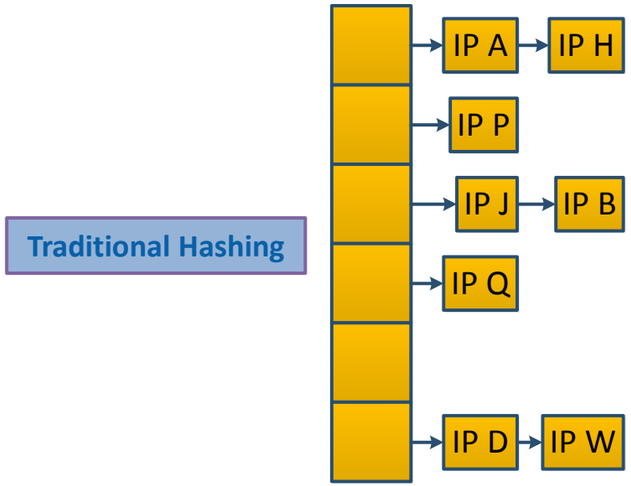
- Flow lookup and Classification a common operation for many network functions.
- NFV workload will typically have large flow table sizes



- ASICs, NPUs uses TCAM to address this bottleneck.
- TCAMs sizes are very limited

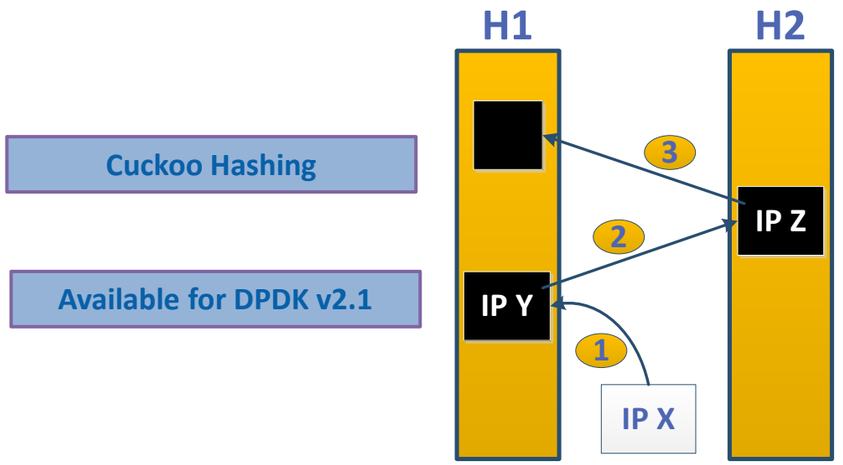
#moveforward

Flow Lookup & Classification Bottleneck for NFV



Traditional J-hash library:

- relies on a “sparse” hash table implementation
- Simple exact match implementation
- Significant performance degradation with increased table sizes.

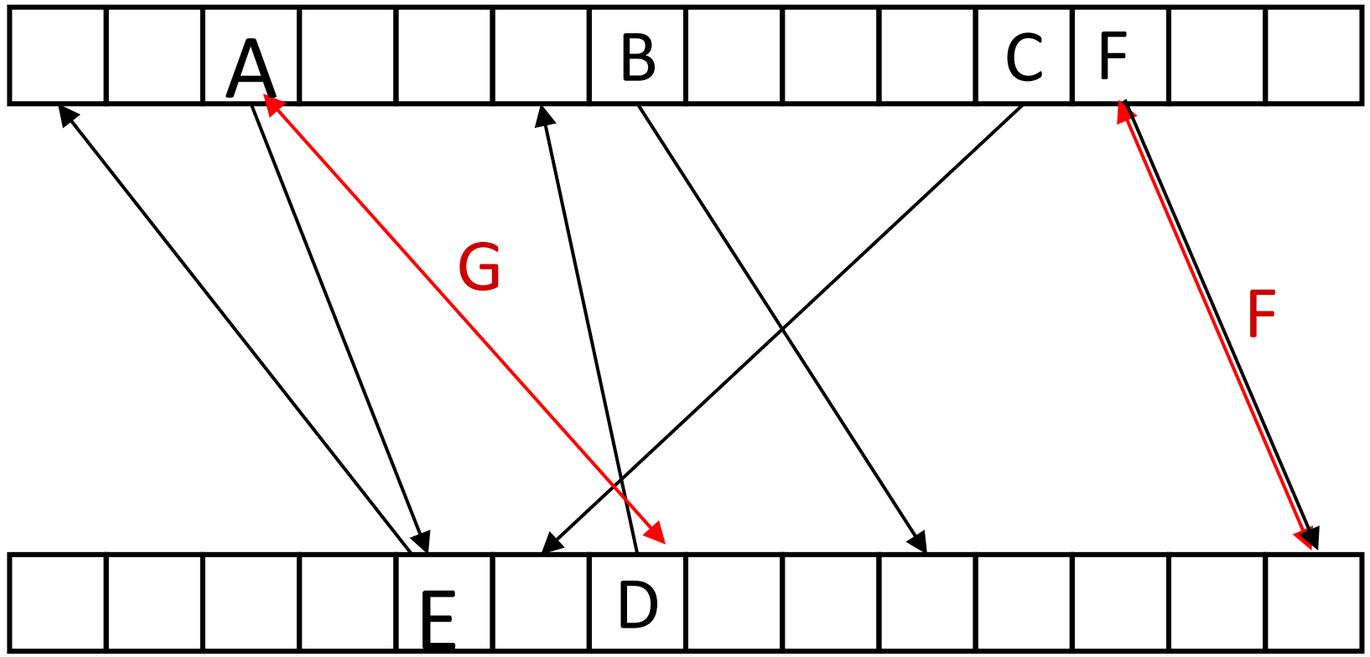


Cuckoo Hashing – Better Scalability:

- Denser tables fit in cache.
- Can scale to millions of entries.
- Significant throughput improvement

#moveforward

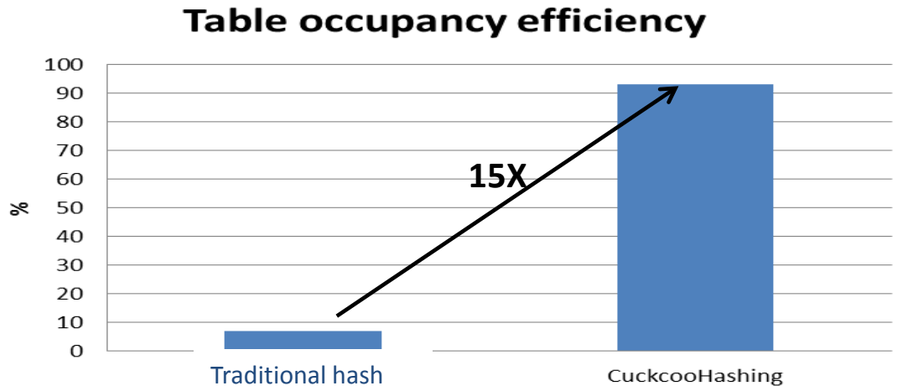
Cuckoo Hashing [Pagh ‘01]



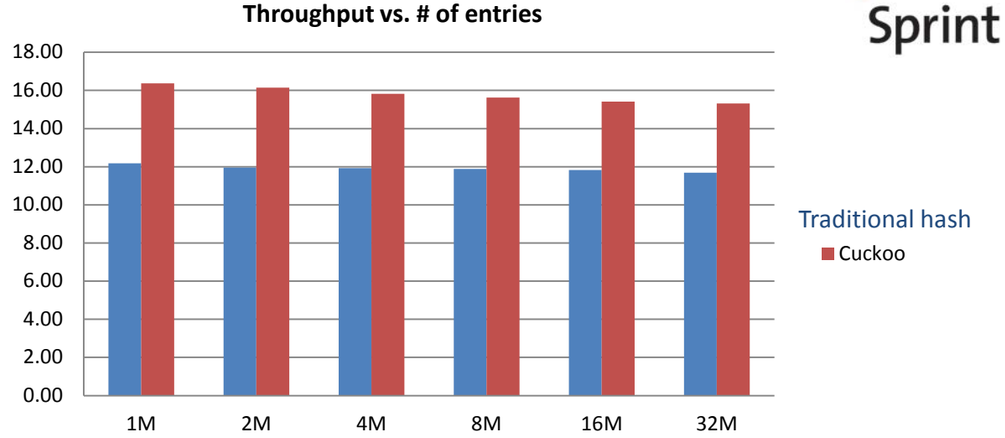
#moveforward

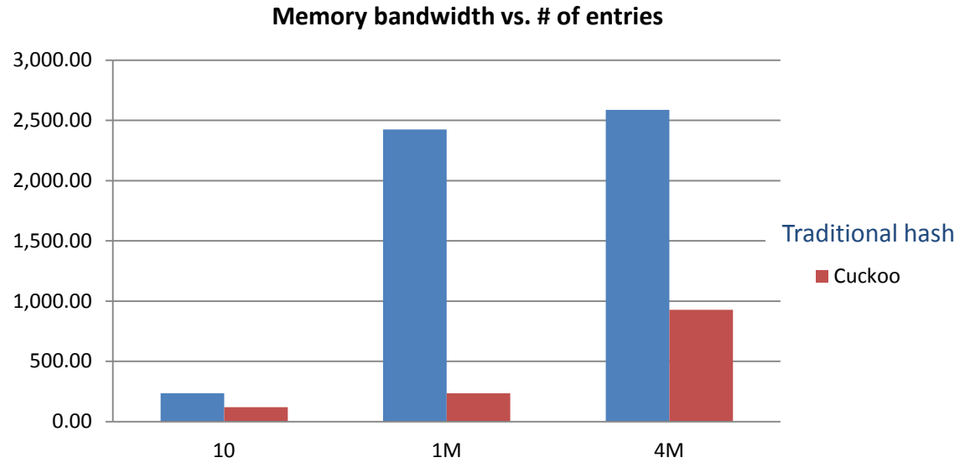
Performance benefits of CH w/ DPDK

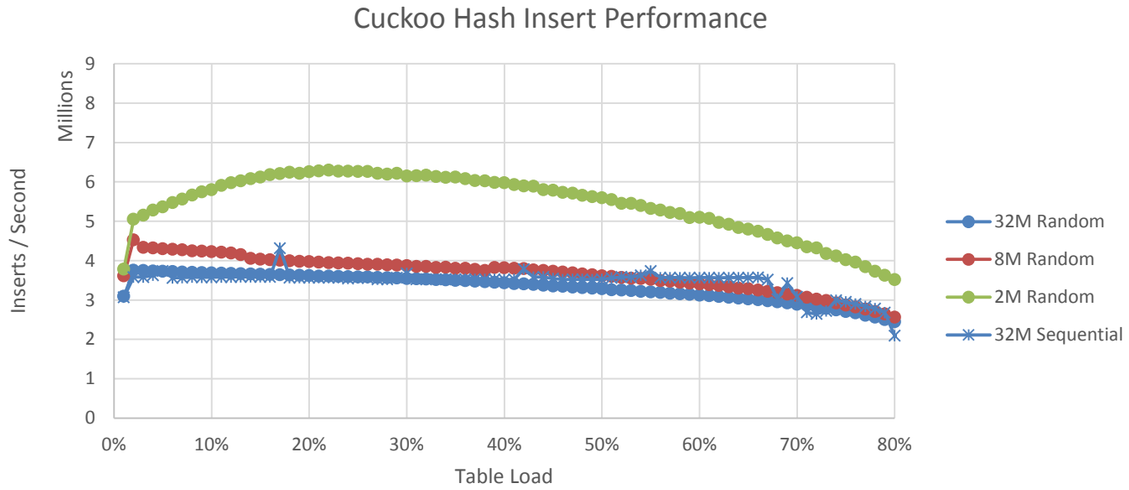
Improvement on table efficiency



~40% Throughput increase





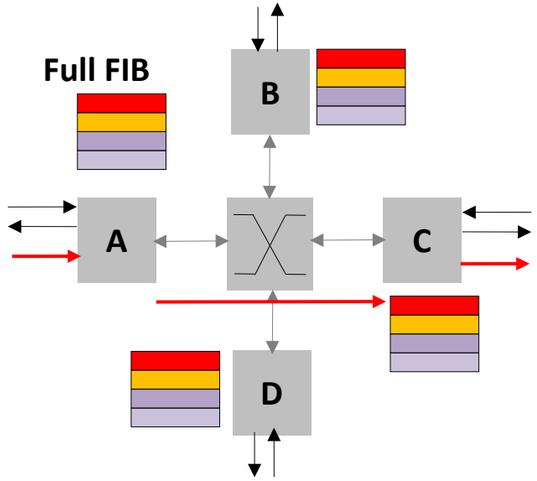


Memory bandwidth significantly reduced due to higher cache utilization

Single Core Table Insertions per second

#moveforward

Distributed software flow lookup

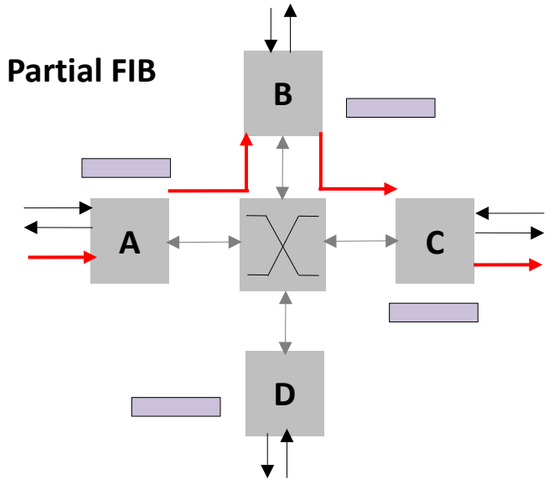


Full Duplication

Nodes store FULL copy of FIB

Pros: Design simplicity, scales throughput

Cons: FIB does not scale as FIB capacity does not increase with the number of nodes in the cluster

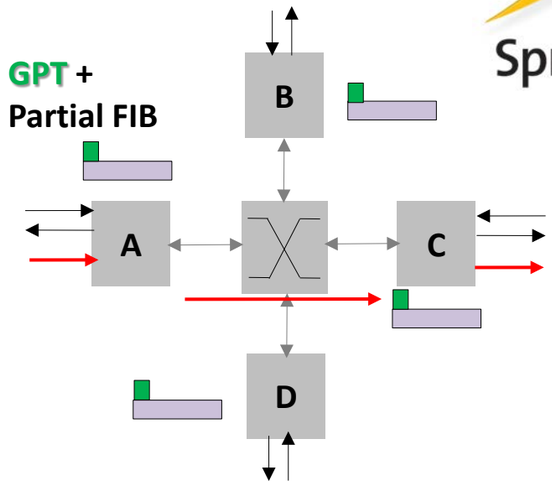


Hash Partitioning

Node stores ONLY a portion of the FIB based on the hash of the keys (destination address, flow identifier …)

Pros: Design simplicity, near linear scalability

Cons: Latency w/ extra hop, increased interconnect load and CPU load for IO bouncing, potential traffic hot spots (w/ elephant flows)



Scalable Switch Route Forward (S2RF)

Nodes keep globally-replicated but extremely compact, and fast, table (Global Partition Table) mapping keys to lookup nodes

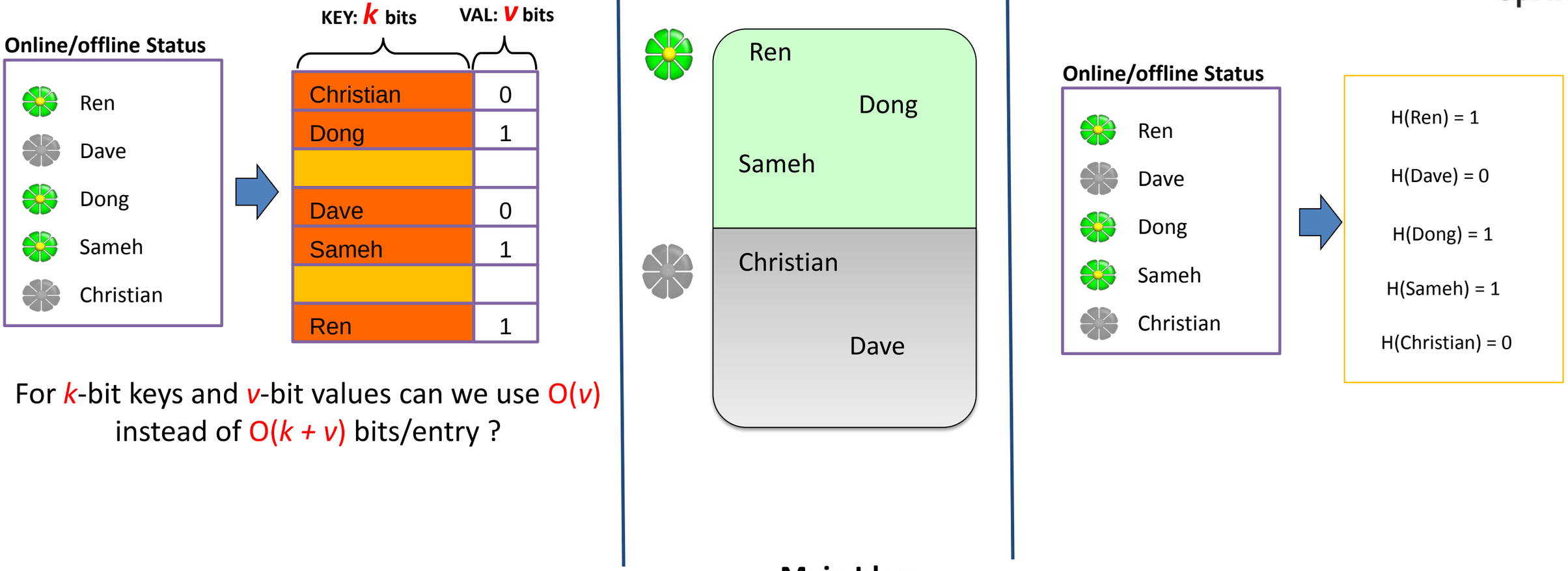
FIB partitioned so lookup node for packet is also its egress node

Pros: No extra latency and interconnect load, min resources required

“Improving Clustered Network Appliances with Xbricks”, Sigcomm ‘15

#moveforward

Lookup Table Space Optimization for GPT



For *k*-bit keys and *v*-bit values can we use O(*v*) instead of O(*k* + *v*) bits/entry ?

Main Idea:

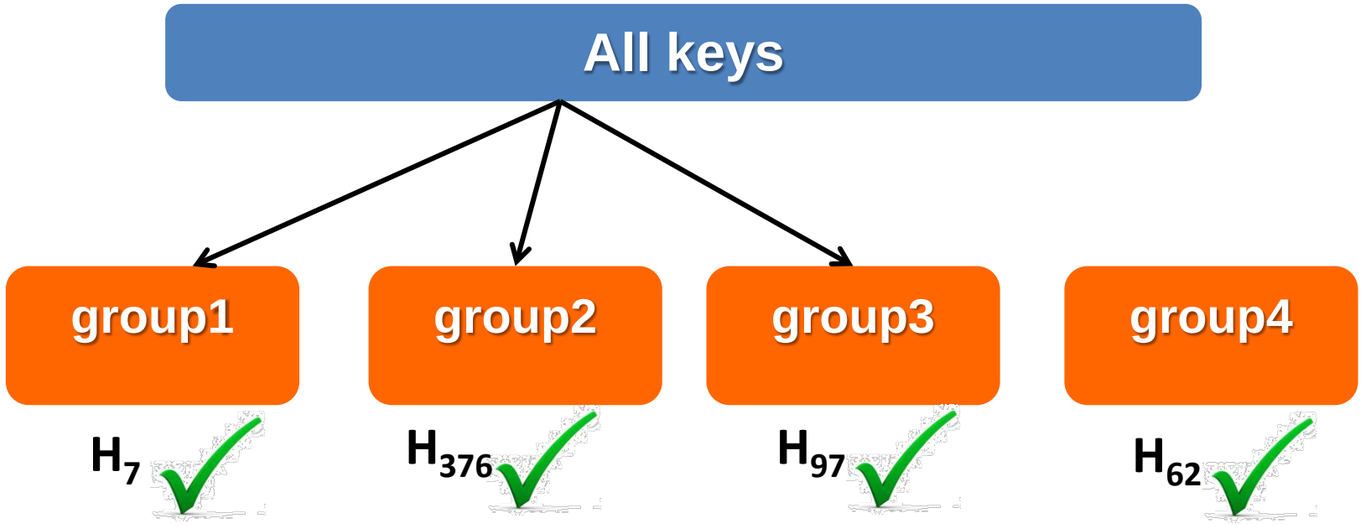
Throw away keys (from cache), Use perfect hashing to avoid collision

#moveforward

GPT: From One Group to Many Groups

	Target Value	$H_1(x)$	$H_2(x)$	...	$H_m(x)$
key1	0	0	1		0
key2	1	0	1		1
...					
key16	0	0	0		0

Store *m* for this group of keys



Store hash function index for each group of keys

#moveforward

S2RF Code Snippet

GPT Lookup

```
void NxtHopTableLookupMulti(tNxtHopTable *table, int numKeys, U32 *keyList, U8
*valueList) {
…
   for (i = 0; i < numKeys; i++)    {
      U32 h = CheapHash(keyList[i]);
      chunkIdList[i] = h % table->numChunks;
      binIdList[i] = (h / table->numChunks) %          NXTHOPTABLE_CHUNK_NUM_BINS;
      rte_prefetch0(&table->chunks[chunkIdList[i]].choiceList[binIdList[i]]); }

   for (i = 0; i < numKeys; i++)    {
      choiceList[i] = GetChoice(table, chunkIdList[i], binIdList[i]);
      groupIdList[i] = BinToGroup(binIdList[i], choiceList[i]);
      rte_prefetch0(&table->chunks[chunkIdList[i]].groups[groupIdList[i]]);  }

   for (i = 0; i < numKeys; i++)    {
      hashValA[i] = NXTHOPTABLE_HASHFUNCA(keyList[i]);
      hashValB[i] = NXTHOPTABLE_HASHFUNCB(keyList[i]);
      valueList[i] = 0;

      for (bit = 0; bit < NXTHOPTABLE_VALUE_SIZE_MAX; bit++)      {
         U16 hashFuncIdx;
         U16 lookupTbl;
         GetGroup(
            &table->chunks[chunkIdList[i]].groups[groupIdList[i]],
            NXTHOPTABLE_VALUE_SIZE_MAX - bit - 1,
            &hashFuncIdx, &lookupTbl)
         valueList[i] = LookupBit(table, hashFuncIdx, lookupTbl);
      }
```

Find group_id using cheap hash

Using hash index do lookup

GPT Update

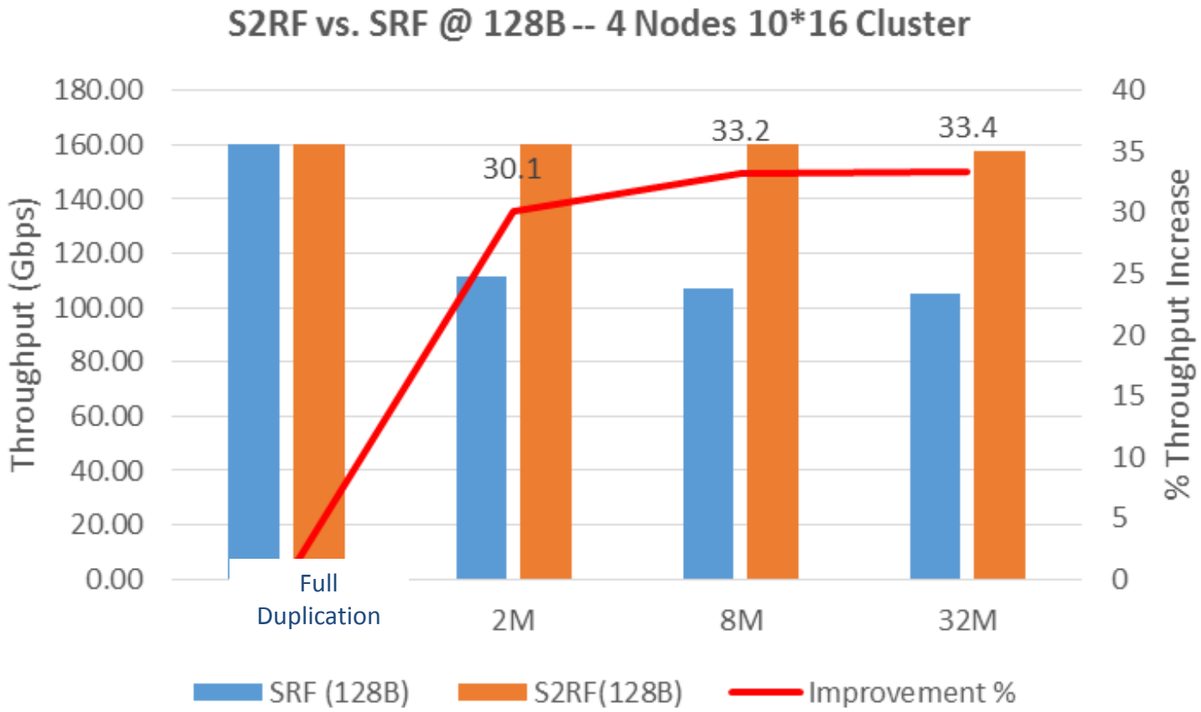
```
void NxtHopTableUpdate(tNxtHopTable *table, U32 key, U8 value)
{
 U32 h = CheapHash(key);
 U32 chunkId = h % table->numChunks;
 U32 binId = (h / table->numChunks) % NXTHOPTABLE_CHUNK_NUM_BINS;
 U8 choice_chunk = table->chunks[chunkId].choiceList[binId / 4];
  int i, offset = (binId & 0x3) * 2;
  U8 choice = (U8)((choice_chunk >> offset) & 0x3);
   U32 groupId = binPerm[choice][binId];
   for (i = 0; i < table->chunkRuleList[chunkId].groupSize[groupId]; ++i)
         if (table->chunkRuleList[chunkId].groupRuleList[groupId][i].ip == key)
          {
        table->chunkRuleList[chunkId].groupRuleList[groupId][i].Id = value;
        break;
            }
 int ret = SearchHash(table,  table->chunkRuleList[chunkId].groupSize[groupId],
table->chunkRuleList[chunkId].groupRuleList[groupId])}
```

Update → finding a new hash function that satisfies the new value

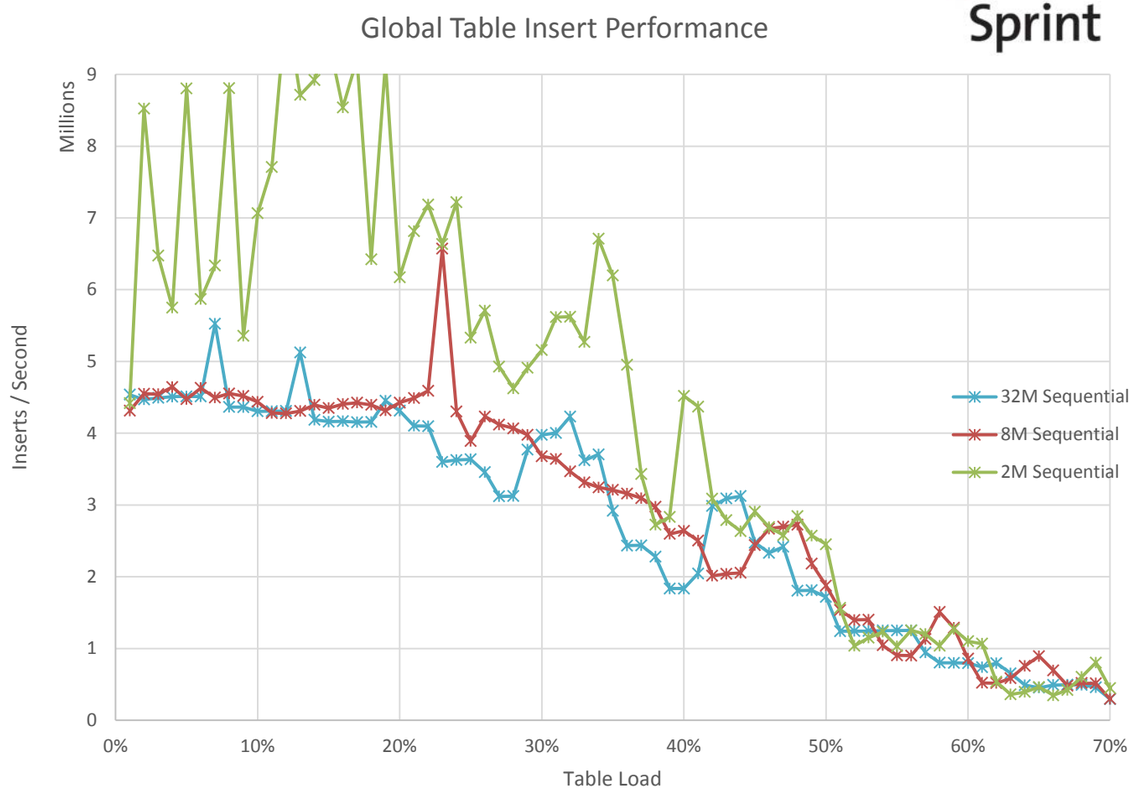
Lookup → hash computation knowing the group hash index

#moveforward

S2RF Performance Quantification



~35% Better Throughput



Single Core Table Insertions per second

Scales Linearly with number of cores

- SNB @ 2.2Ghz, 20 MB LLC
- 4-Node Cluster, 16*10 Gbps Niantic vector driver/DPDK
- IPv4 random traffic, i.e. 1/N on local node, ¾ on remote node

#moveforward

Best Practices for Efficient Packet processing

Avoiding serialization in the packet-processing pipeline, including serializing events such as locks, special instructions such as CLFLUSH, and large critical sections

Accessing data from the cache where possible by making use of prefetch instructions and observing best practices in design of the software pipeline

Designing data structures to be cache-aligned and avoiding occurrences of data being spread across two cache lines, partial writes, and contention between write and read operations

Maintaining affinity between software threads and hardware threads, as well as isolating software threads from one another with regard to scheduling relative to hardware threads

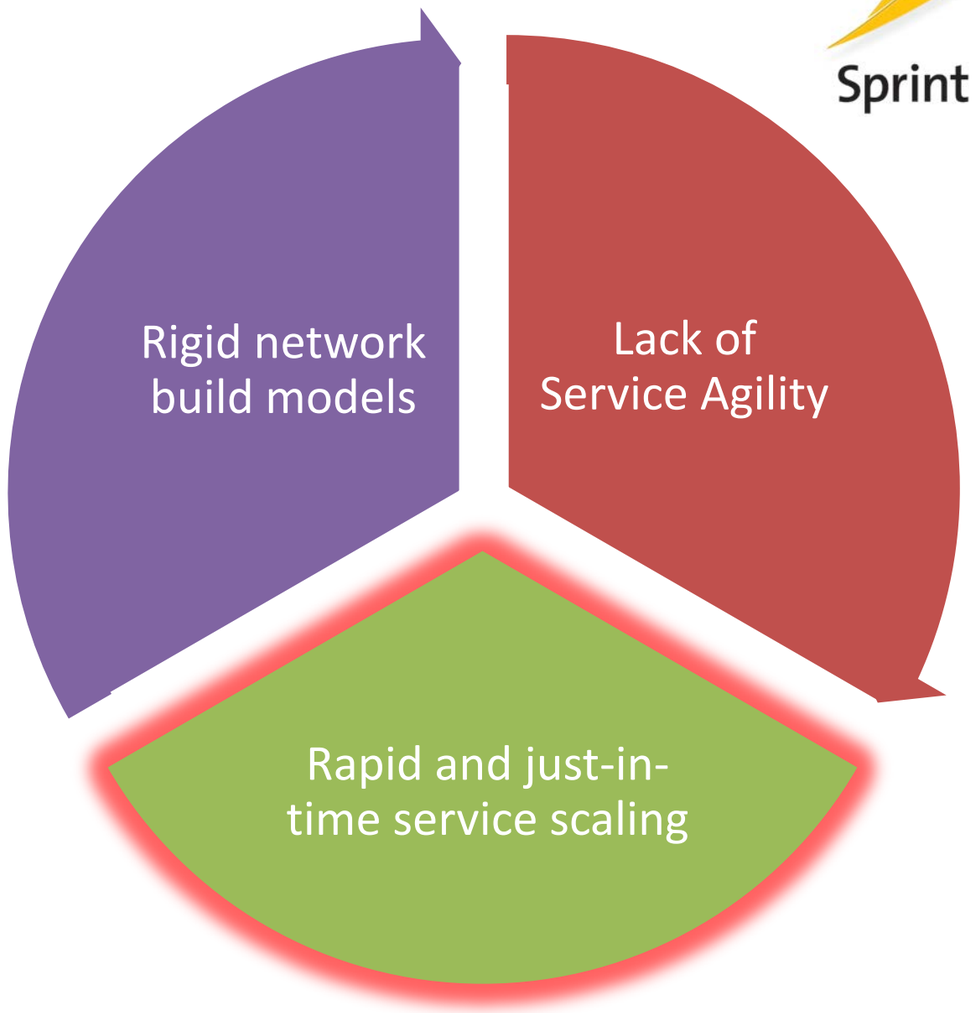
Breaking down user-plane functionality so that it can be implemented with a combination of RTC (Run to Completion) and pipeline methods

Use of pre-tuned open source optimized software components like DPDK

#moveforward

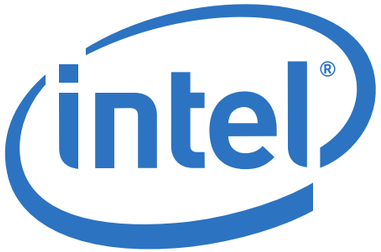
Summary

- Scalable Switch Route Forward (S2RF) helps address some of the scaling challenges in carrier networks
  - Scales linearly the number of ports and flow classification size with the number of nodes in a cluster
  - Uses DPDK and IA optimizations for efficient packet processing and I/O performance



#moveforward

Questions



#moveforward