Scaling Up Clustered Network Appliances with ScaleBricks

SIGCOMM 2015, a work by CMU group that developed Cuckoo switch

Overview

- A new switching architecture for clustered network appliances.
- Good scalability with this new switching architecture:
  - Throughput scaling.
  - FIB scaling.
  - Update scaling.
- Low latency compared with existing solutions (one hop routing.)
- Practical usage to improve the performance of LTE-to-Internet Gateway.

Motivation: Evolved Packet Core in LTE

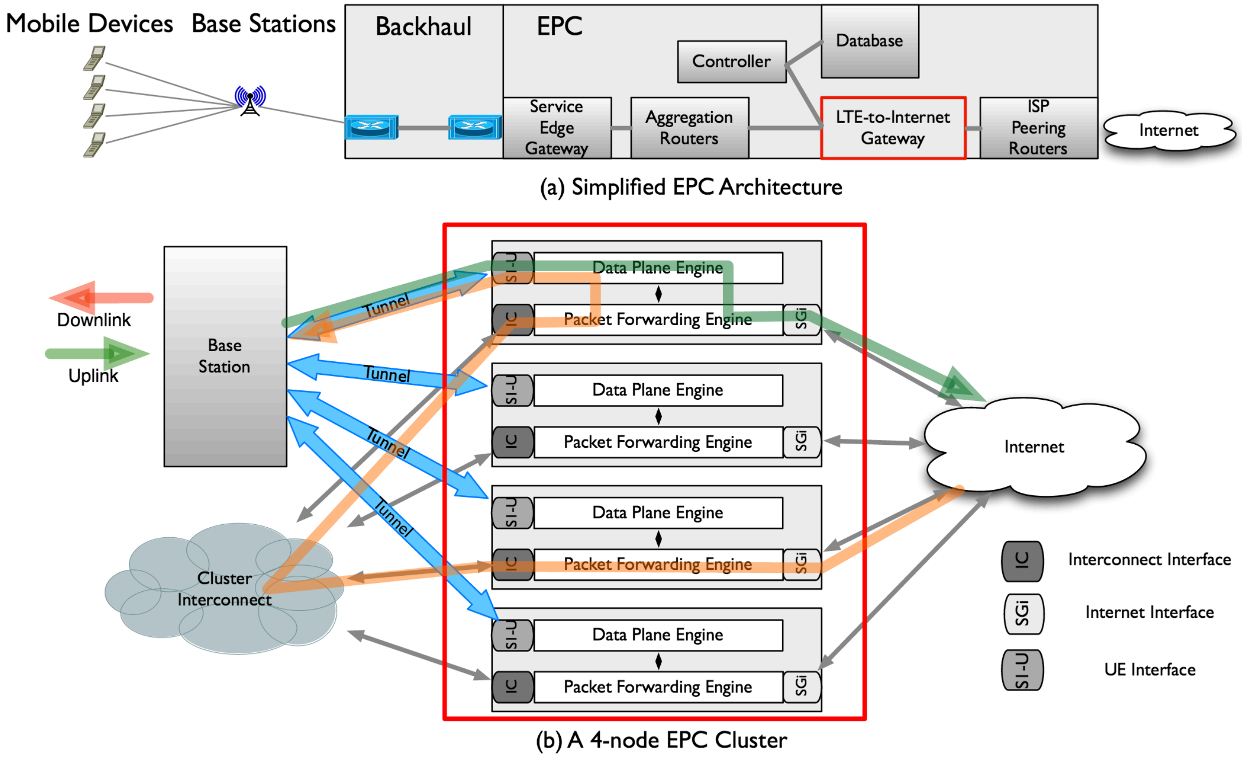


Figure 1: (a) Simplified Evolved Packet Core (EPC) architecture and (b) a 4-node EPC cluster

Forward Information Base (FIB)

- FIB is a mapping between a key and a value.
- In the LTE example, a FIB is a mapping from the **5-tutple flow identifiers** to **(handling node, TEID) pair.**

Design New Switching Layer for Clustered Network Appliances

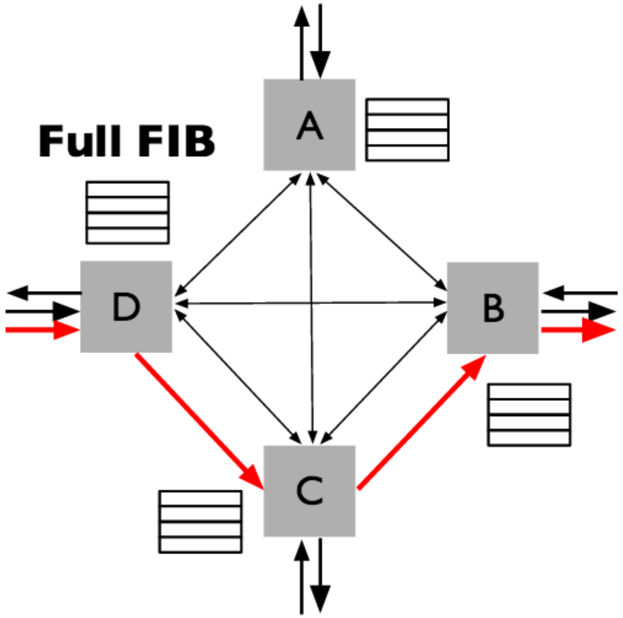
- Design goal:

  1. **Scalable FIB size**. The number of FIB size should scales with the number of network appliances. Full replication doesn't work.

  2. **Low latency switching**. The number of network appliances that a packet travels before reaching handling node should be as small as possible.

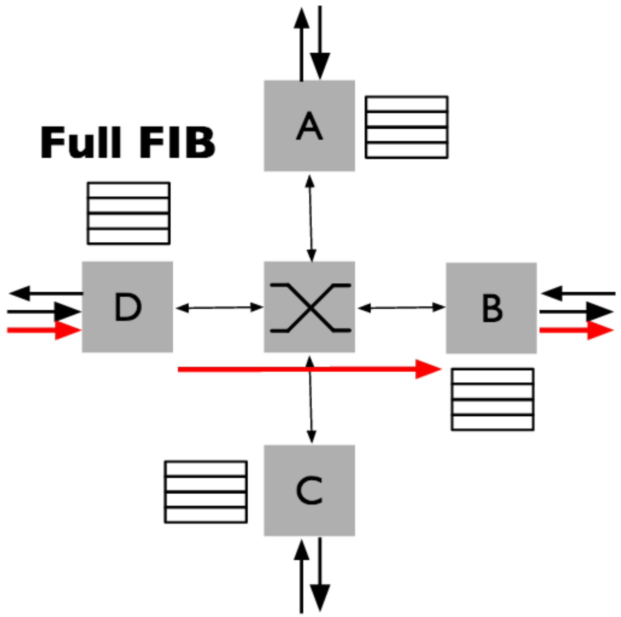
  3. **Small inter-connection bandwidth**. In order to support 40Gbps throughput, the inter-connection bandwidth should also be 40Gbps.

RouteBricks Violates Goal 1 2 3.



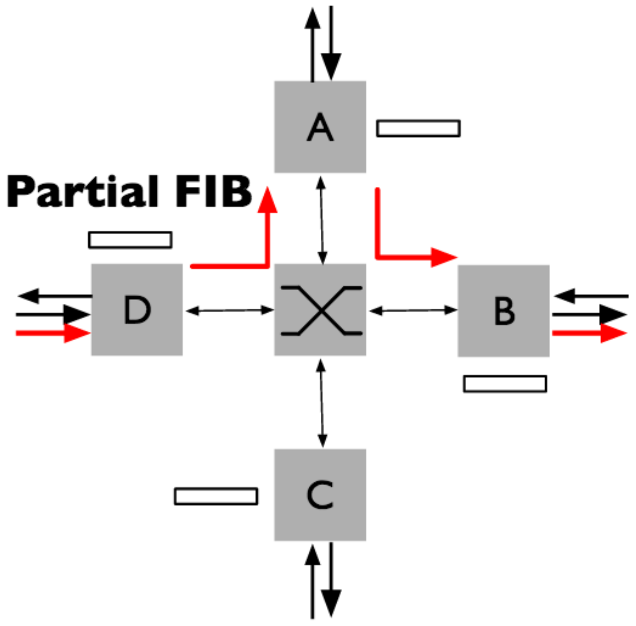
(a) RouteBricks

Full Replication + Commercial Switch Violate Goal 1



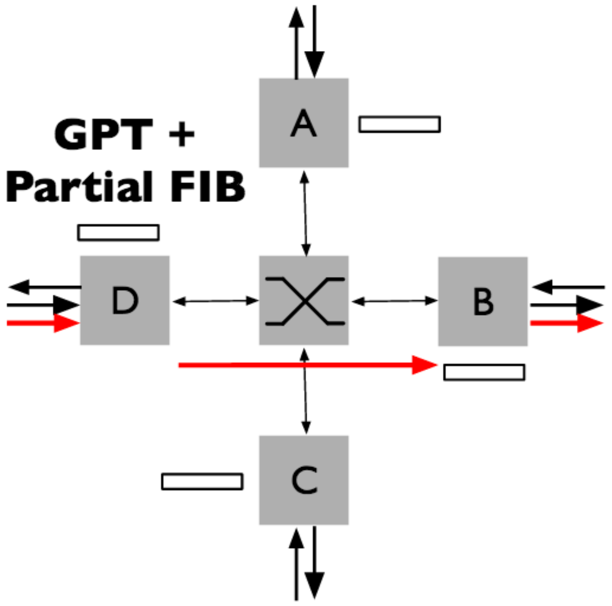
(b) Full Duplication

Consistent Hashing Violates Goal 2



(c) Hash Partitioning

ScaleBricks Satisfy All the Goal



(d) ScaleBricks

1. GPT stores flow key to handling node mapping.
2. GPT is replicated on every node.
3. GPT has a concise representation.

1. Paritial FIB stores flow key to other data mapping. (i.e. flow ket to TEID as in LTE network)
2. Partial FIB is partitioned across the cluster.
3. Each node only stores partial FIB that points to itself.

Why GPT Has a Concise Representation

- GPT stores flow key to handling node mapping.
- The number of handling node is small, in the range of 16 to 32.
- Using general purpose look up table is unnecessary.
- Treat it as a set partition problem to achieve concise representation.
- Partial FIB is stored using Cuckoo hashing developed by this group's previous work.

Binary Set Separation

- n key-value pairs (x_j, y_j). $j=0\ldots n-1$, y_j is either 0 or 1.
- Find out a hash function $H_i(x)$, so that $H_i(x_j)=y_j$, $j=0...n-1$.
- Practically, $H_i(x) = G_1(x) + i*G_2(x)$.
- Store hash function index i using variable-length encoding.
- This method requires **n bits on average** to store a function for binary set separation of **n keys**.

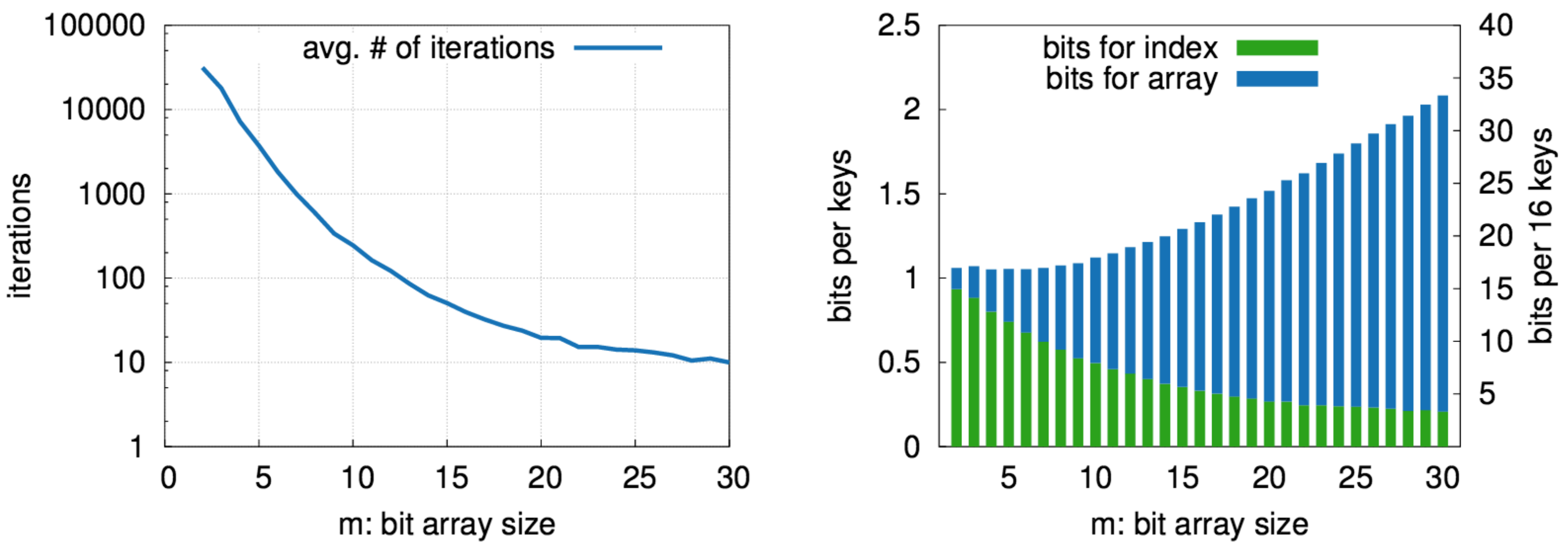
Trade Space for Fast Construction

- Construct hash function is slow. Need to test 2^n hash functions.
- Use m (2*m ?) more bits to speed up hash function construction.
- $H_i(x_j)$ points to a position in a bitarray:
  - Initially all bits in bitarrary are “not marked”.
  - If $H_i(x_j)$ points to a position that is “not marked”, assign y_j to that position.
  - If $H_i(x_j)$ points to a position that is “marked”, if bitarray[$H_i(x_j)$] == y_j, then continue testing with next j, otherwise reject $H_i(x_j)$ and test another hash function.

Trade Space for Fast Construction

- This optimization speeds up the hash function construction by 100x, while using 24 bits to encode hash function for 16 keys.
- To represent V > 2 subsets instead of only 2 subsets, $\log_2 V$ hash functions will be constructed, one for each bit.

Performance of Set Separation



(a) Avg. # of iters. to get one hash func. **(b) Space cost breakdown**

Figure 3: Space vs. time, as the function of bit array size ***m***

Scaling to Billions of Items

- All the keys are partitioned using a hash function into different groups, each group should contain 16 keys on average.
- Then use set separation techniques to construct GPT.
- But the group size generated by a hash function has large variance.
  - Hash function search time increases exponentially with the number of keys in a group.

2-level Hashing

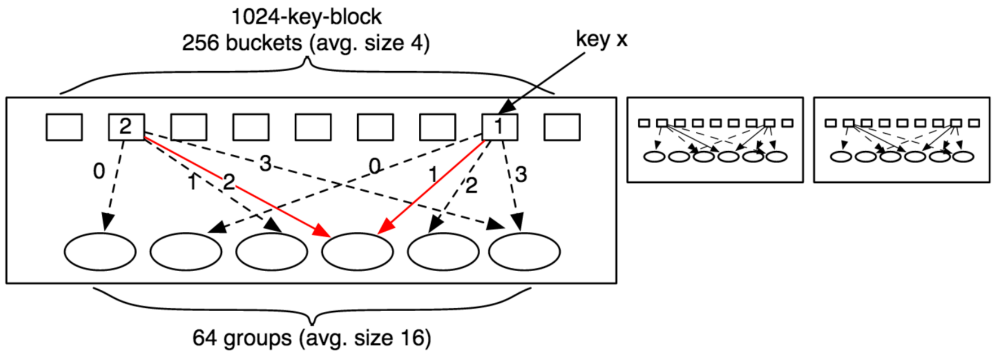


Figure 5: Illustration of two-level hashing

- First use hashing to map keys to buckets with small average size (4).
- Then assign 256 consecutive buckets to 64 groups, each group has a size of 16.
- Calculate the assignment using a greedy algorithm.
- The 2-level hashing greatly decrease the variance in the group size.

Scalable Update

- For initial construction, controller partitions keys using a hash function, so that consecutive 256 buckets will locate on the same node.
- Node only calculates its only set separation and then broadcasts its result to other nodes.
- When updating a key k, only node responsible for k recomputes the group that k belongs to, and then broadcasts the result to other nodes.

Implementation

- Implemented using Intel DPDK.
- Use memory prefetch to increase cache hit rate.
- Use multi-threading to accelerate the computing of hash function of a group.
- For the partial FIB, use Cuckoo hashing.

Algorithm

Algorithm 1: Batched SetSep lookup with prefetching

```
BatchedLookup(keys[1..n])
begin
    for i ← 1 to n do
        bucketID[i] ← keys[i]'s bucket ID
        prefetch(bucketIDToGroupID[bucketID[i]])
    for i ← 1 to n do
        groupID[i] ← bucketIDToGroupID[bucketID[i]]
        prefetch(groupInfoArray[groupID[i]])
    for i ← 1 to n do
        groupInfo ← groupInfoArray[groupID[i]]
        values[i] ← LookupSingleKey(groupInfo, keys[i])
    return values[1..n]
```

Evaluation: Hash Function Construction

Construction setting			Construction throughput	Fallback ratio	Total size	Bits/ key
	$x+y$ bits to store a hash function, x-bit hash function index and y-bit array					
16+8	1-bit value	1 thread	0.54 Mkeys/sec	0.00%	16.00 MB	2.00
8+16	1-bit value	1 thread	2.42 Mkeys/sec	1.15%	16.64 MB	2.08
16+16	1-bit value	1 thread	2.47 Mkeys/sec	0.00%	20.00 MB	2.50
			increasing the value size			
16+8	**2-bit value**	1 thread	0.24 Mkeys/sec	0.00%	28.00 MB	3.50
16+8	**3-bit value**	1 thread	0.18 Mkeys/sec	0.00%	40.00 MB	5.00
16+8	**4-bit value**	1 thread	0.14 Mkeys/sec	0.00%	52.00 MB	6.50
			using multiple threads to generate			
16+8	1-bit value	**2 threads**	0.93 Mkeys/sec	0.00%	16.00 MB	2.00
16+8	1-bit value	**4 threads**	1.56 Mkeys/sec	0.00%	16.00 MB	2.00
16+8	1-bit value	**8 threads**	2.28 Mkeys/sec	0.00%	16.00 MB	2.00
16+8	1-bit value	**16 threads**	2.97 Mkeys/sec	0.00%	16.00 MB	2.00

Table 1: Construction throughput of SetSep for 64 M keys with different settings

Evaluation: Look Up Throughput of GPT

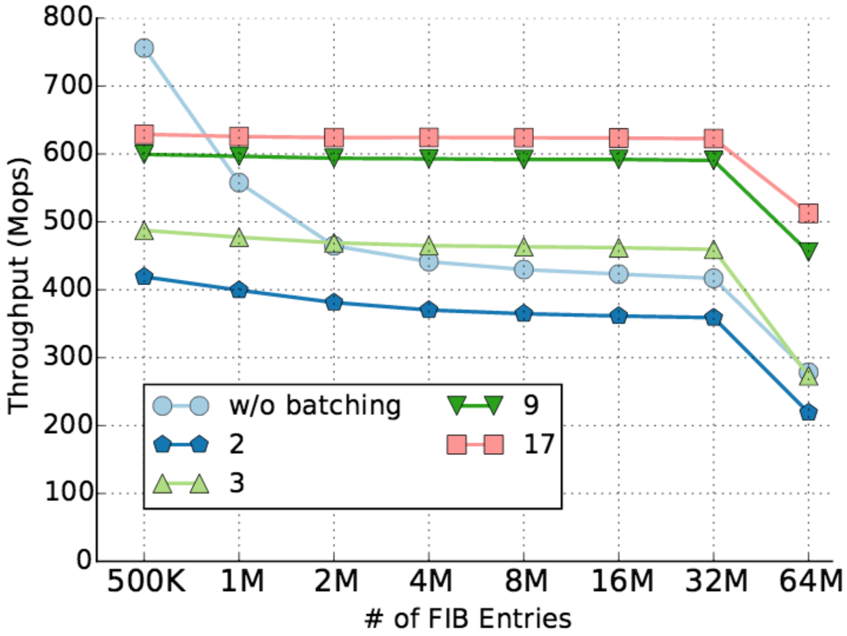


Figure 7: Local lookup throughput of SetSep (GPT)

Evaluation: Single Node Throughput

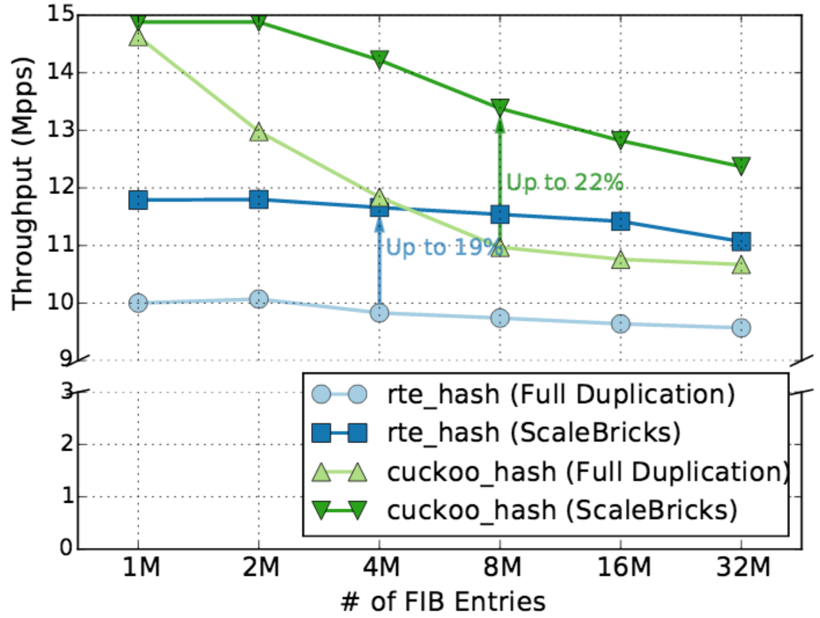


Figure 8: Single node packet forwarding throughput using 30 MiB L3 cache

Evaluation: Latency

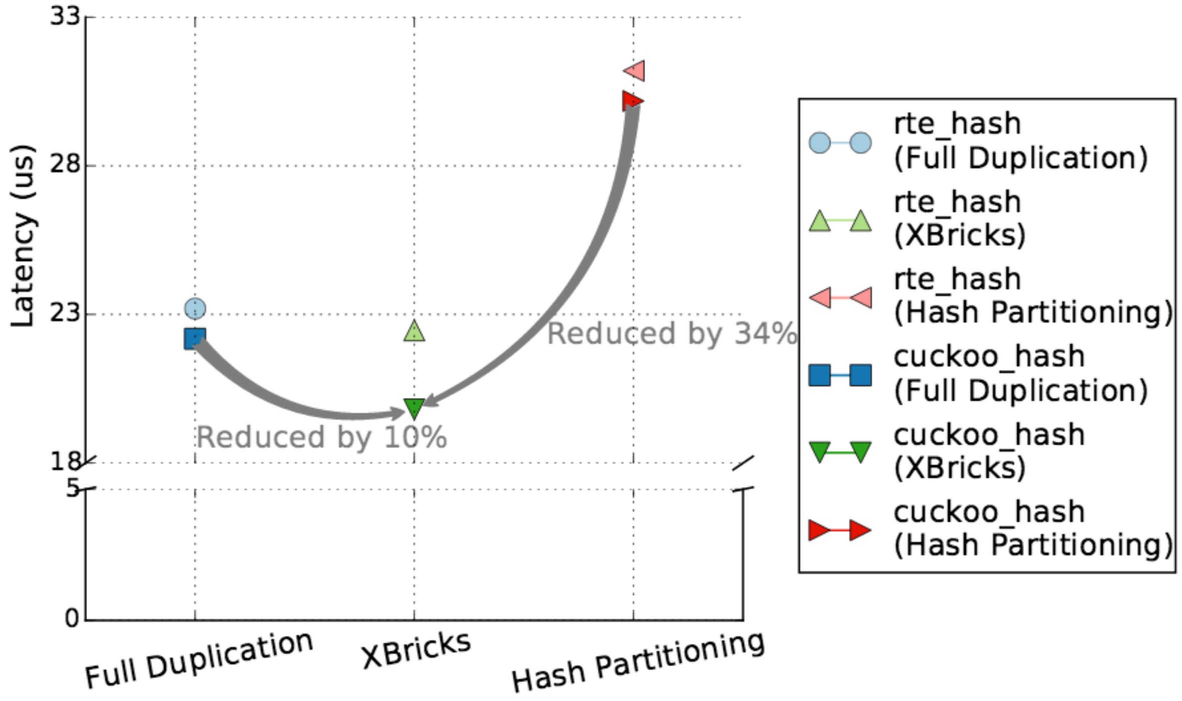


Figure 10: End-to-end latency of different approaches

Evaluation: FIB Scalability

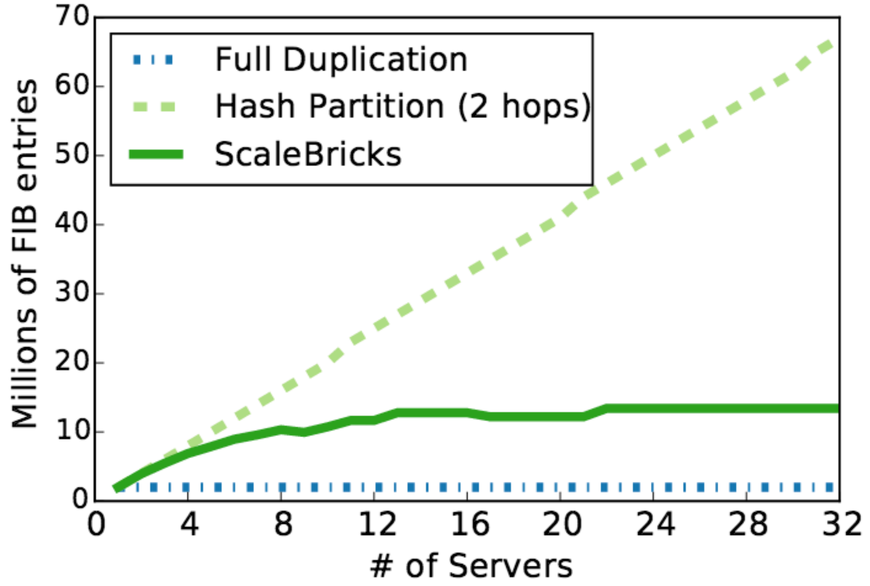


Figure 11: # of FIB entries with different # of servers