Scaling Up Clustered Network Appliances with ScaleBricks

DongZhou
Bin Fan
Hyeontaek Lim
David G. Andersen
Michael Kaminsky

Focus

- Throughput Scaling

  scale with number of servers

- FIB Scaling

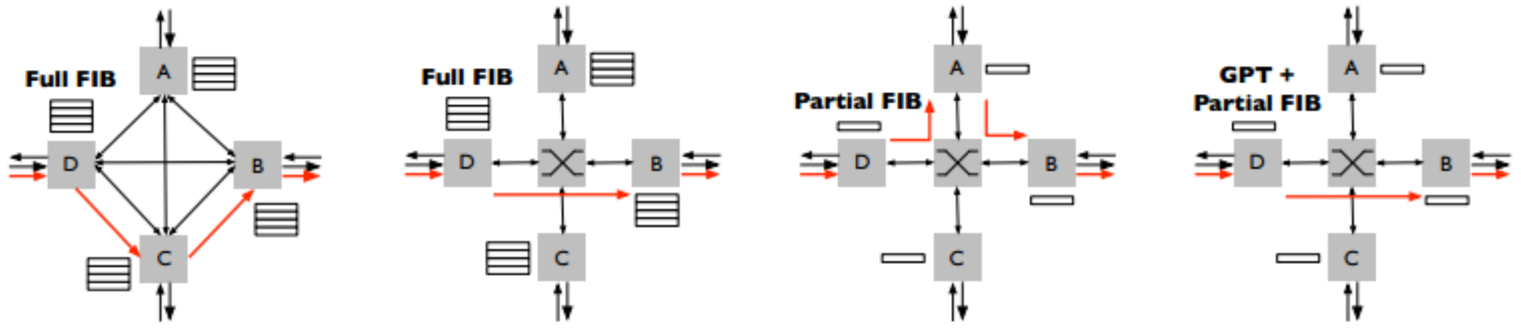
  total size of forwarding table(the number of supported keys)

- Update Scaling

  update rate of the FIB

FIB (comprised Entries for Forwarding or Processing)

Cluster Architecture



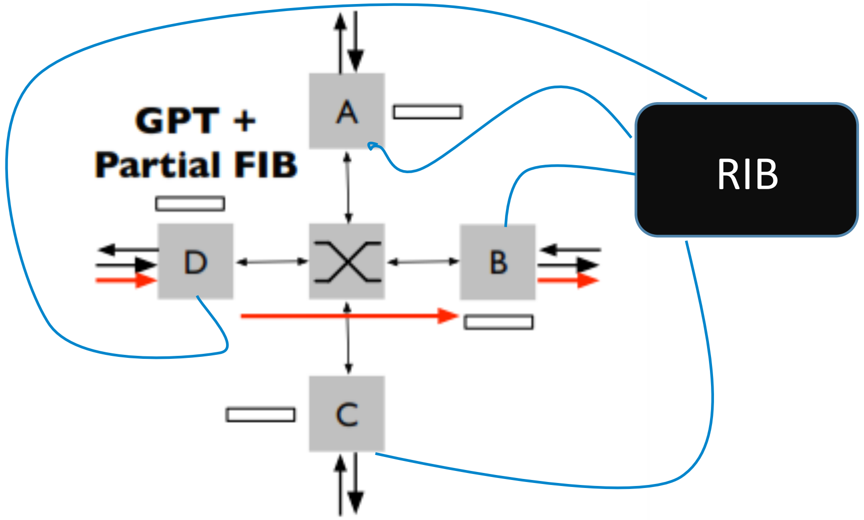
- FIB size (full or slice)
- Intermidate (Server or Switch)
- Internal Bindwidth

ScaleBricks' Design

Maintian entire routing information in **RIB**

Distribute RIB across the cluster

Generate FIB & GPT



- **Partial FIB**
  - Each handling node stores FIB entries that point to it
  - Based on prior work (leveraging CuckHashing)
- **Global Partition Table(GPT)**
  - Used for forward packet to handling node
  - Replicated to every ingress node
  - **Must be compact**
- **RIB partition and updates**

Global Partition Table

Attributes of switch-based “middle box” cluster

- Total number of nodes is typically modest
- They can handle one-side error

Key Question: How to map millions of input key to nodes?
correct
Basic idea in high level:

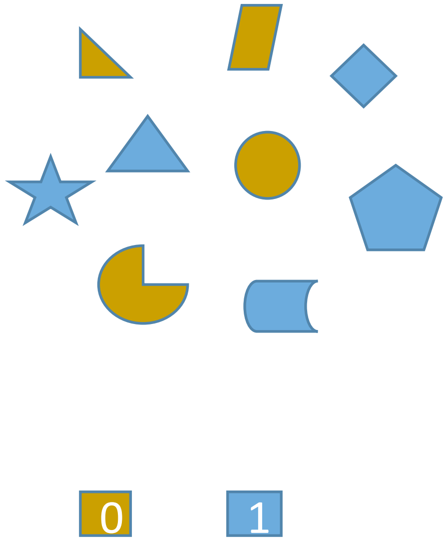
- Maintain a Hash function **families**
- Use **brute force** to find the suitable hash function
- Store the **indices** rather than key/value

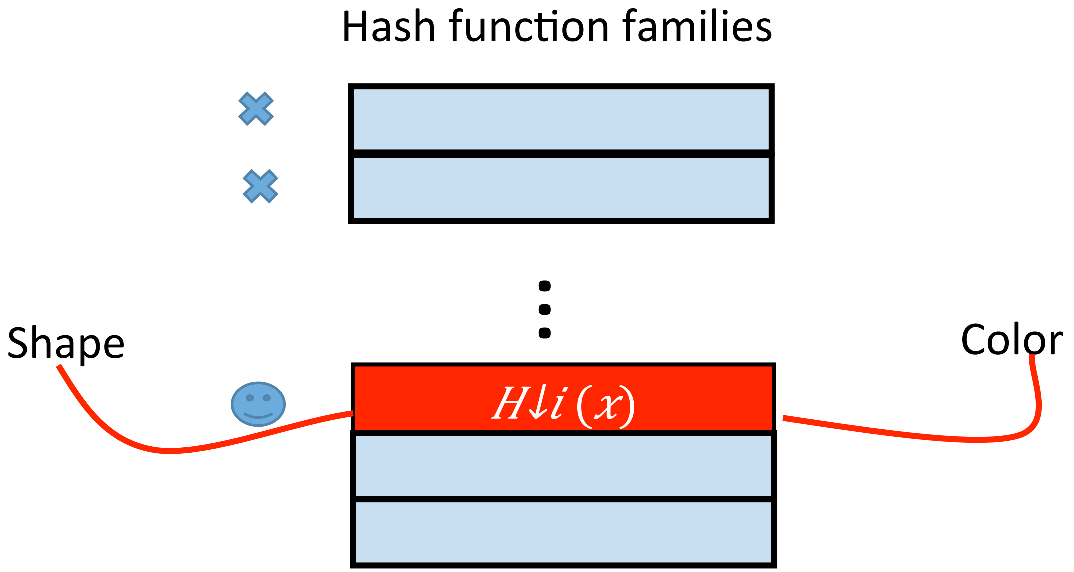
Group & Set Separation(**SetSep**)

Global Partition Table

How to divide a group of n keys into two disjoint subsets when n is small?

Binary Separation





Store the index i if all match

If no function succeeds for $i<l$, a fallback mechanism is triggered.

Global Partition Table

Why SetSep Save Space?

Optimistically assume hash function produce fully random hash values:

Probability all n keys are properly mapped is $p=(1/2)^{n}$

The number of tested functions is a random variable with Geometric distribution with

Entropy: $-(1-p)\log_2 (1-p)-p\log_2 p/p \approx -\log_2 p=n$

Storing a function for binary set separation requires 1 bit per key

16bits for a group of 16 keys

Global Partition Table

Generate the hash function family

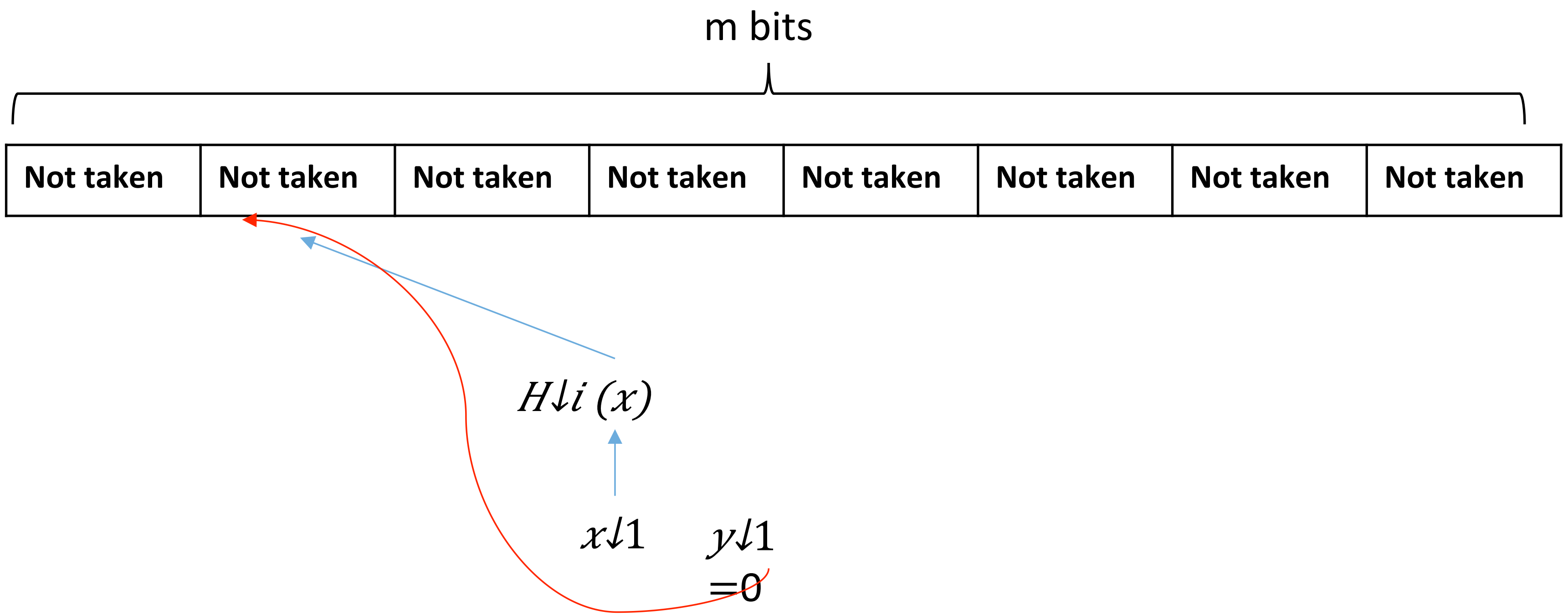
- $H_i(x) = G_1(x) + i \cdot G_2(x)$
- In Practice, only the most significant bit are used
- Construct fast but theoretically weak(lack sufficient independence) Empirically feasible

Horrible iterations for finding a hash function(2^n), how to **speed up** construction ?

Global Partition Table

Trading Space for faster Construction

- Adds an array of m bits(m>2)
- intuitive thinking: more buckets , fewer collisions and increase odds of success.



Global Partition Table

Trading Space for faster Construction

m bits

Not taken	Taken : 0	Not taken	Not taken	Not taken	Not taken	Not taken	Not taken

$H_i(x)$

x_2 $y_2 = 1$

Global Partition Table

Trading Space for faster Construction

Oops! Conflicts.

m bits

Not taken	**Taken : 0**	**Not taken**	**Not taken**	**Not taken**	**Taken : 1**	**Not taken**	**Not taken**

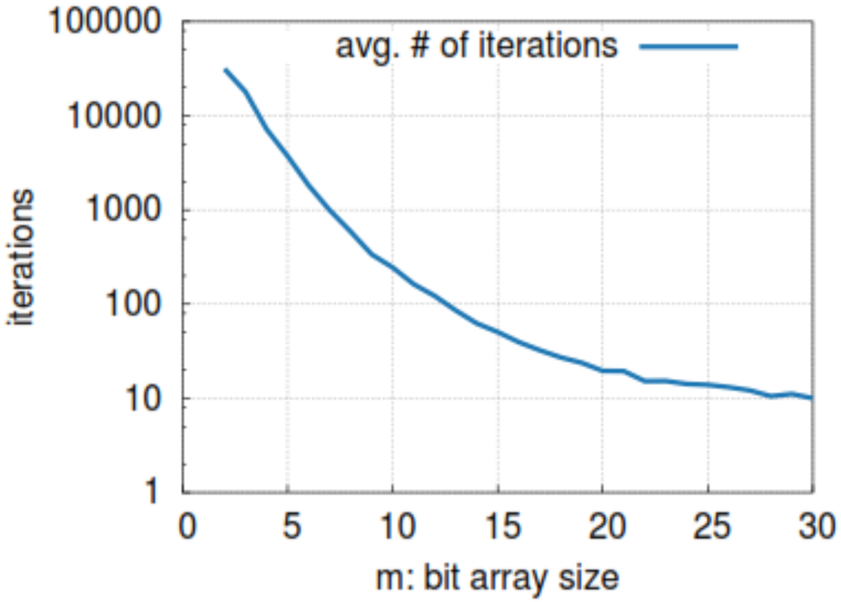
$H_i(x)$

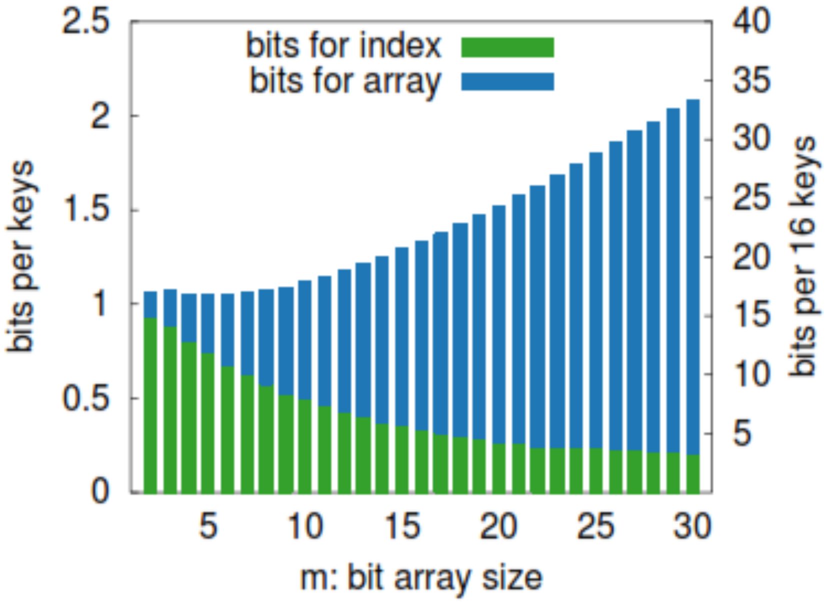
$x_3 \quad y_3 = 0$

Global Partition Table

Representing the **SetSep**

- Fixed 24-bit representation per group
- 16 bits represent hash index and m=8





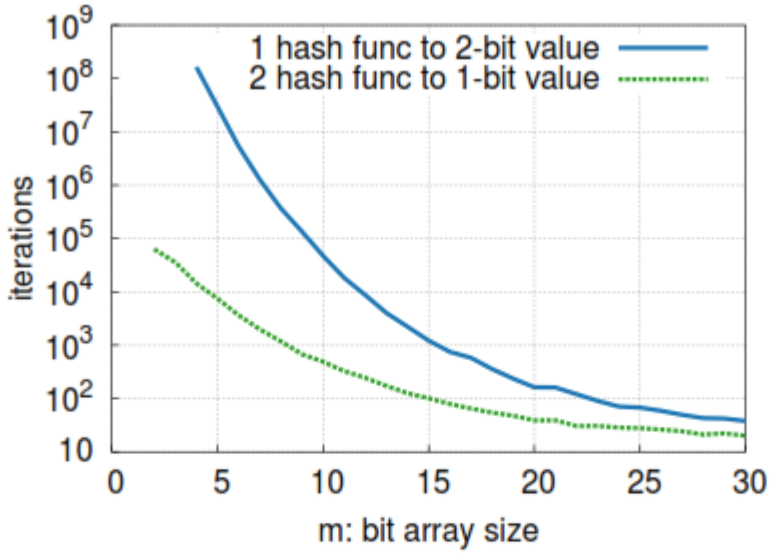
Global Partition Table

Representing the Non-Boolean values

- j-th hash function is responsible for generating j-th bit of final mapping value.
- Mapping value: {0,1,2,3}

“foo” maps to 1, “bar” maps to 2

Then hash function 1 maps “foo”, “bar” to 0,1 respectively; hash function 2 maps “foo”, “bar” to 1,0 respectively.

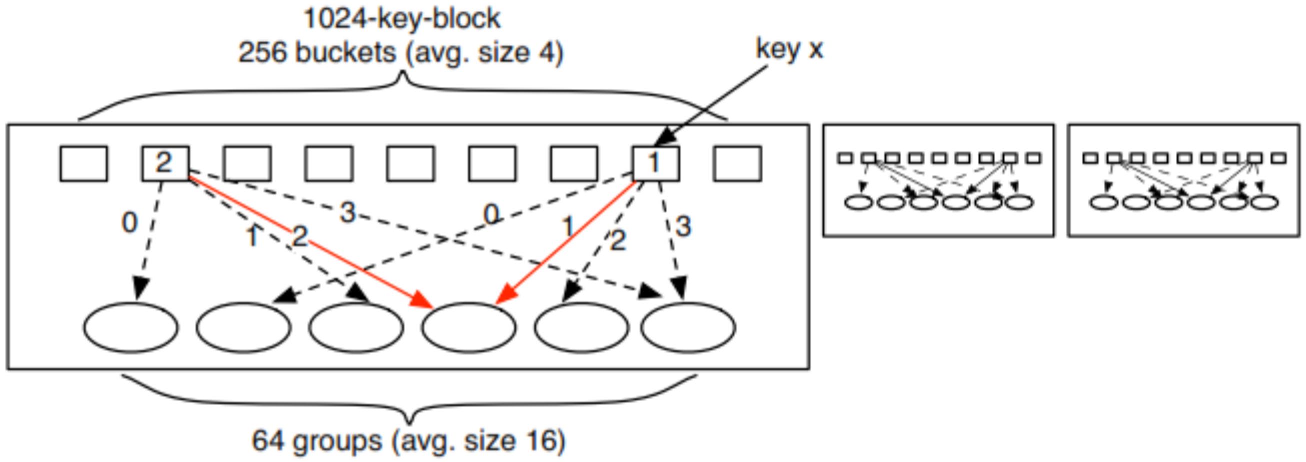


Global Partition Table

Group(how to map a key to a group)

- Low variance in group size(strong hash function/ sort and assign both failed)
- Mapping should add little space

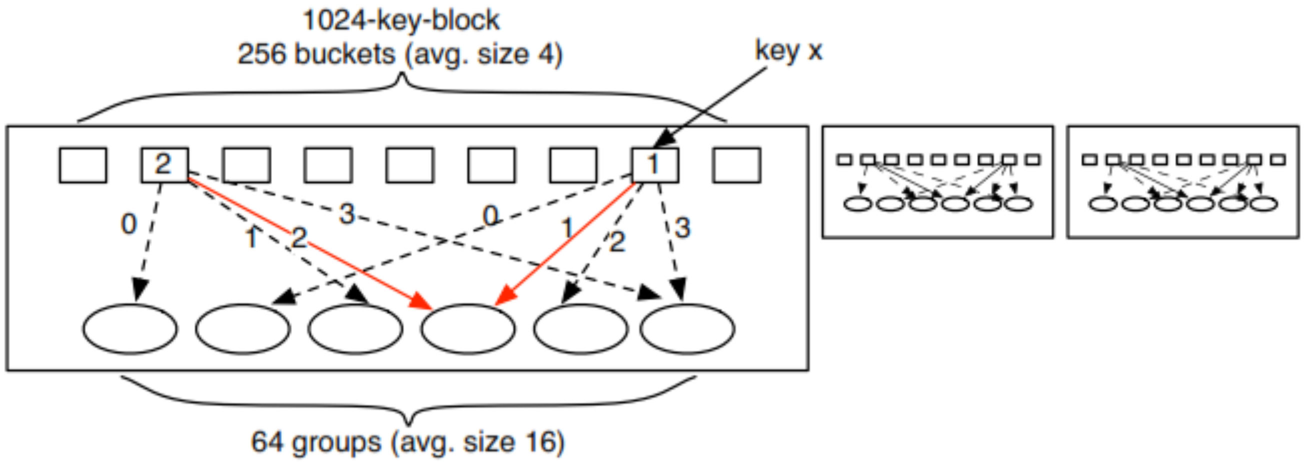
Two level hashing



Global Partition Table

Two level hashing

- Long range of small buckets shows less variance
- Map 1024-key block to 64 groups of average size 16
- Pre-assigned “Candidate groups” & additional storage for choice
- Keys assignment is an NP-hard variant of knapsack problem



Imlementation & Optimization

Global Partition Table using Setsep

- Intel DPDK
- Batched look-ups and prefetch
- Hardware Accelerate Construction

  SIMD or GPU may help

Partial FIB using Cuckoo Hashing

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

Micro-Benchmark

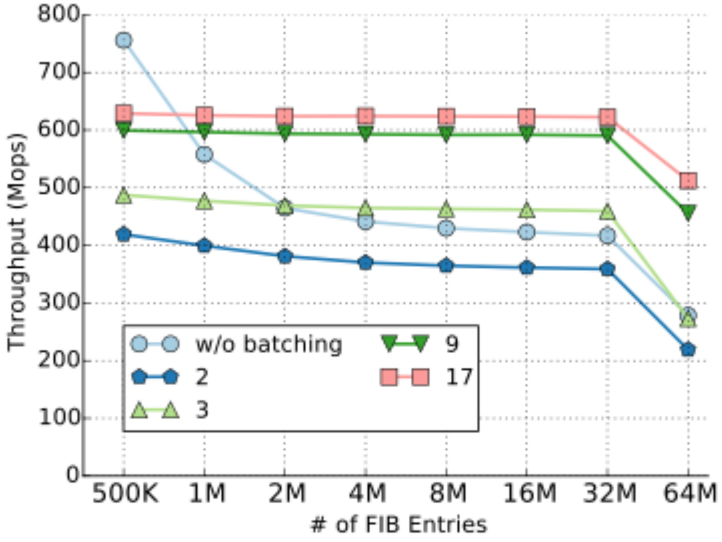
- Construction Throughput

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

Evaluation

Micro-Benchmark

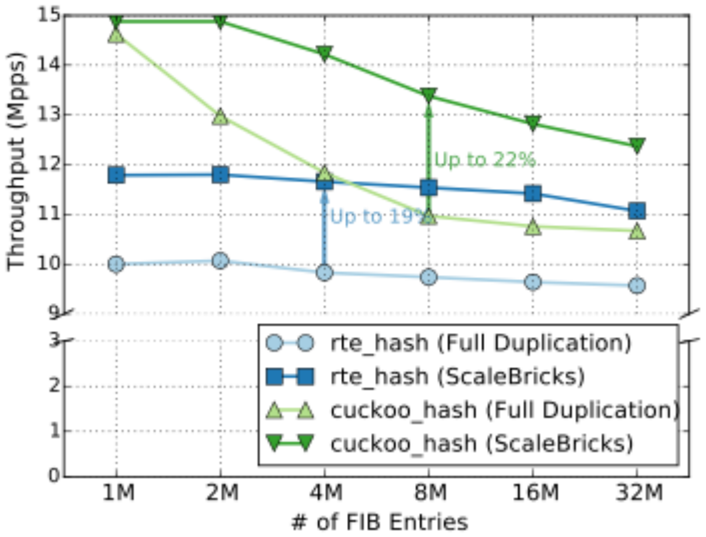
- Local Lookup Throughput

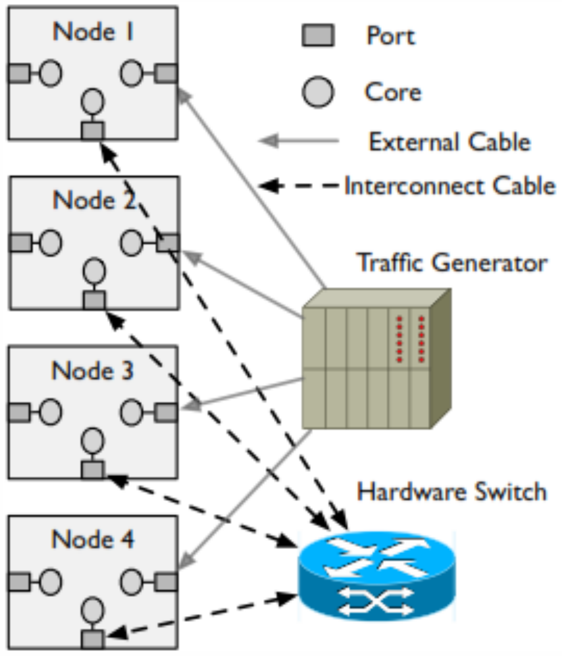


Evaluation

Macro-Benchmark

- rte_hash & extended cuckoo hash table
- Single node throuput /Full duplication & ScaleBricks
- Improve the through put and core utilization.

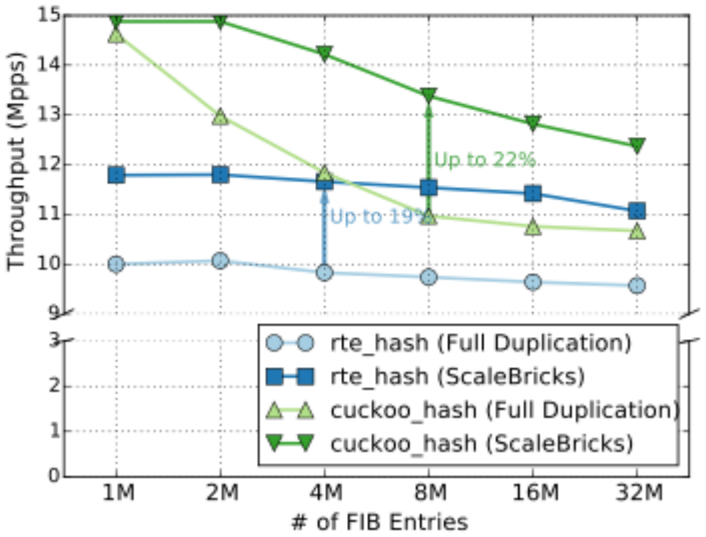


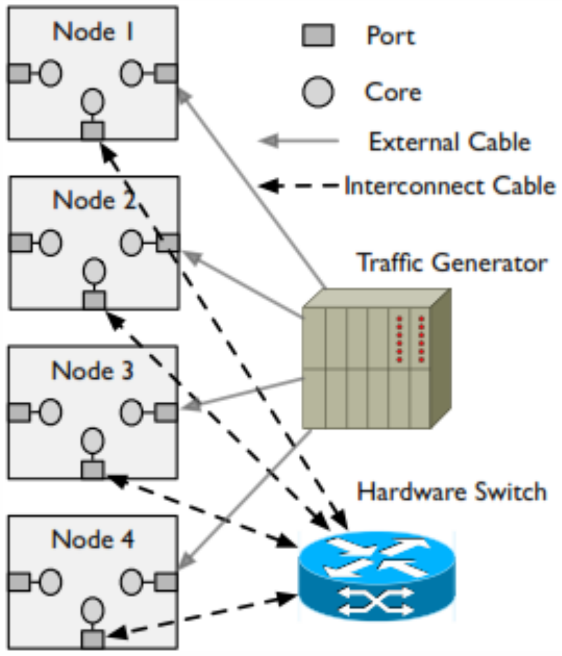


Evaluation

Scalabilityrte_hash & extended cuckoo hash table

- Single node throuput /Full duplication & ScaleBricks
- Improve the through put and core utilization.





Evaluation

Macro-Benchmark

- Share the CPU cache with other application/ launch thread to consume cache
- Latency
- Update rate

Single core handle 60K updates/sec, 4-node cluster for aggregated rate of 240K updates/sec.