Hash Table Design and Optimization for Software Virtual Switches

PRESENTER: **REN WANG**

YIPENG WANG, SAMEH GOBRIEL, REN WANG, CHARLIE TAI, CRISTIAN DUMITRESCU

INTEL LABS

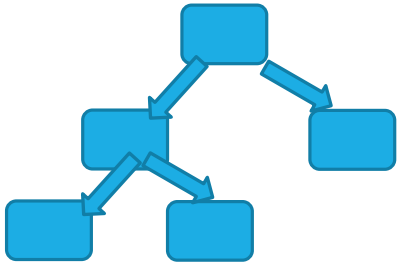
OUTLINE

- Background and motivation
- Survey and understanding
- Analysis

Background

- We found the most common data structure used in virtual switch is hash table.
  - wildcarding match (tuple space search): routing table, ACL
  - exact match: con-track table, flow cache, etc.

- Comparing to tree based data structure, hash table based data structure has certain advantages:
  - More parallelism: no pointer chasing
  - Faster rule updates



Background

- Hash table lookup is also one of the most time consuming stage during packet processing:
  - E.g. Open vSwitch (100k rules, 20 subtables)

	IO	Preprocessing	Rule lookup	others
Execution percentage	~8%	~5%	~78%	~9%

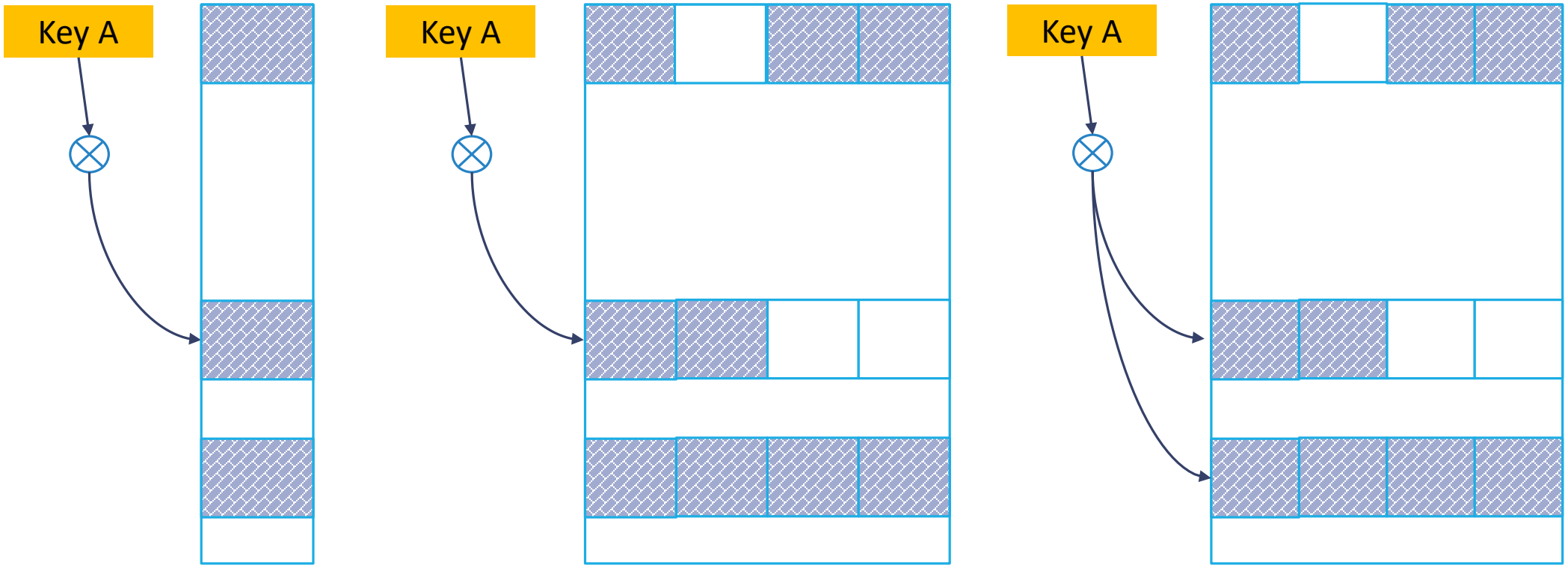
- A major source of hash table lookup overhead is memory access latency.

Motivation

- Hash table is a simple data structure, but there are many different design and implementations.
- Understanding of hash table performance and how to design an efficient hash table structure is the key to a good software switch.
- A general guideline to hash table designs will benefit future vswitch development.

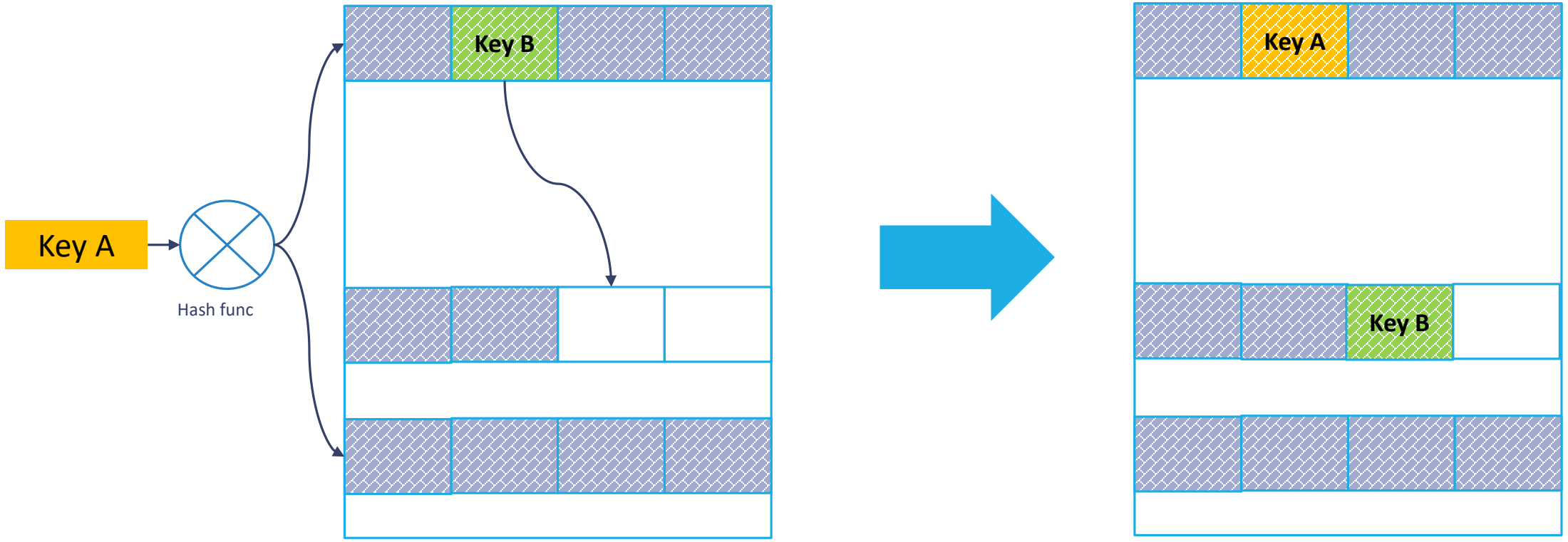
Basic hash table structure

- The evolution of hash table algorithms: single array -> bucket-based -> n-hash



Cuckoo hashing

- Cuckoo hash algorithm: existing keys can be displaced to alternative bucket



Survey

- We also studied into various open source virtual switch applications to learn their implementations.

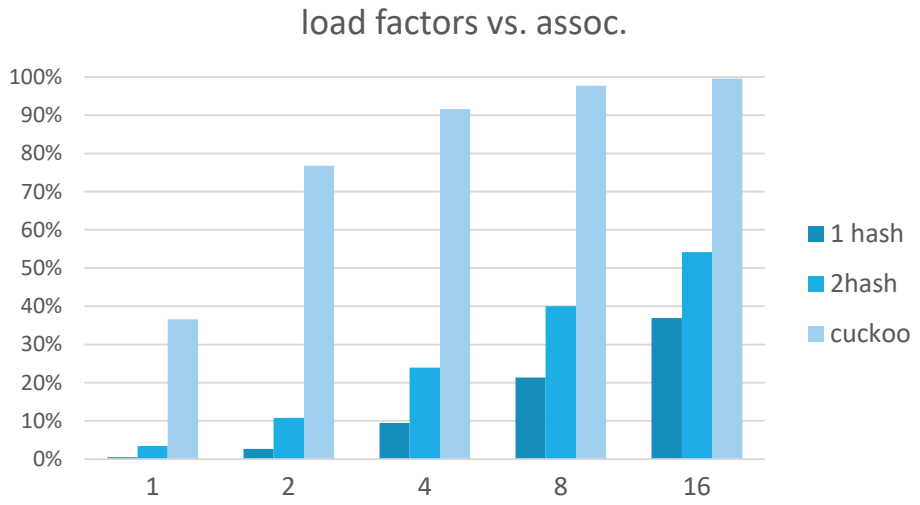
- Three major purposes these applications use hash table for:
  - Routing table/ACL – tuple space search
  - Connect tracking table – exact match
  - Flow cache – exact/signature match with replacement policy

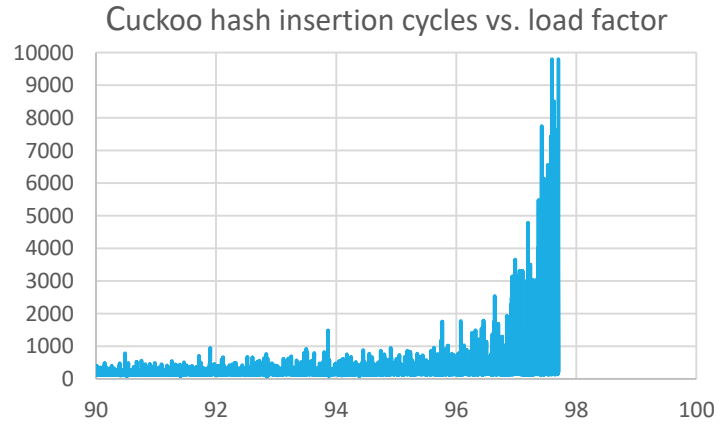
Observations

- Set-associative table and cuckoo hash are widely used.
- Bucket size is usually 4-8 entries
  - Cache alignment
  - Vectorization
- Capacity guarantee is needed in telcom use cases
  - Linked list based hash table as extended table
- Software techniques to improve performance:
  - Software pipelining
  - Batching
- Read write concurrency
  - Optimistic locking
  - Intel TSX

Analysis
- Table organization and data structure

- Number of keys per bucket
  - More entries in a bucket can directly improve the table utilizations.

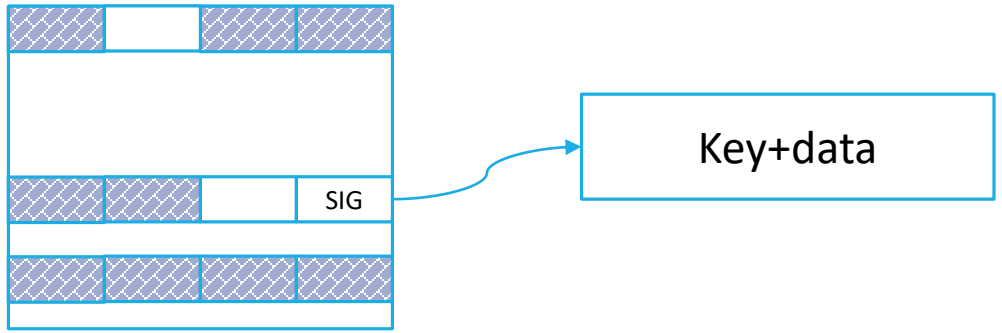




  - Conclusion: when table utilization is important, cuckoo hash should be used. Multiple hash function and multiple ways per bucket also help a lot.

Analysis
- Separate key storage and cache alignment

- Hash tables could store key-data pair in a separate memory location, and only keep signatures and index in the table.
  - Pros: signature and index are easily to be cache aligned, benefit cache miss case.
  - Cons: requires another memory jump when hit.

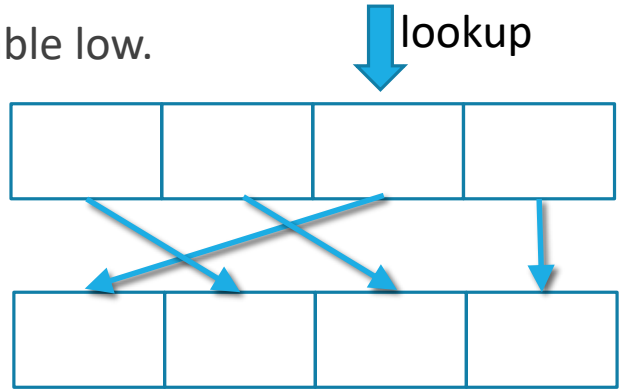


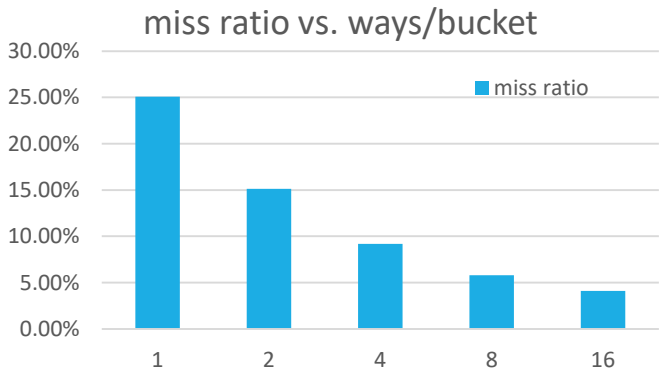
- Out tests show that with optimized DPDK hash tables, storing keys in or outside the table does not show major difference with 16 or 32-byte key size.

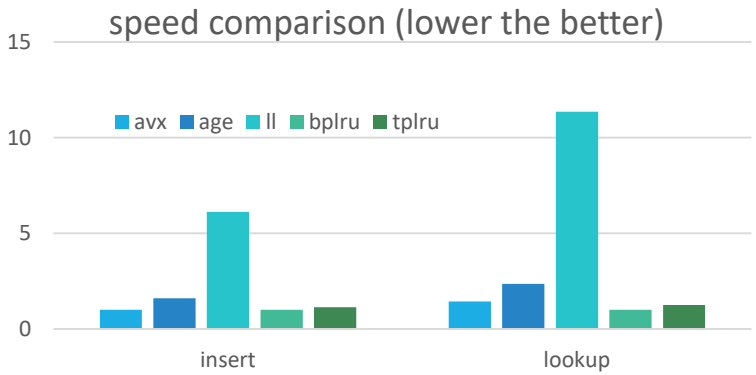
- However, cache alignment will improve hash table lookup speed by 6.5-16.7% in our DPDK based performance test.

Analysis
- Hash table based cache

- When use hash table for flow cache we need to consider cache miss ratio.
  - 4-8 ways per bucket can already keep the miss ratio to be reasonable low.
- We propose a new AVX-based LRU implementation.
  - Use Intel AVX instruction to permute the bucket.



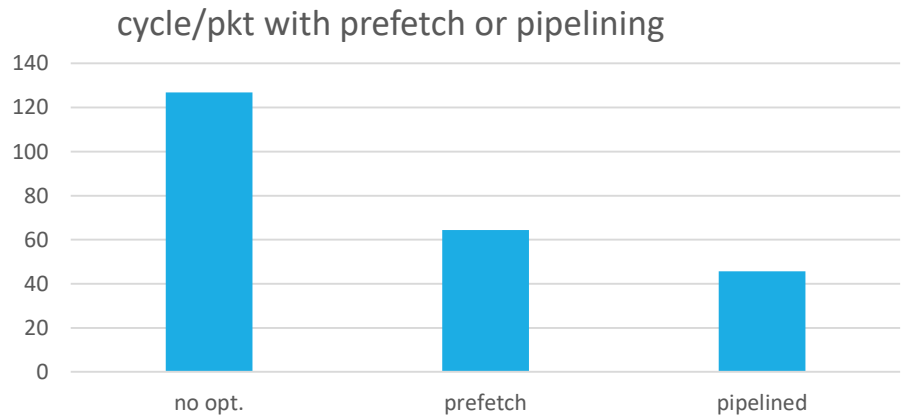





```
void adjust_location(int location, bucket* bucket){
    __m256i array = avx_load(bucket)
    __m256i permute_pattern = avx_load(permute_index[location])
    __m256i permuted_array = avx_permute(array, permute_pattern)
    avx_store (bucket, permuted_array)
}
```

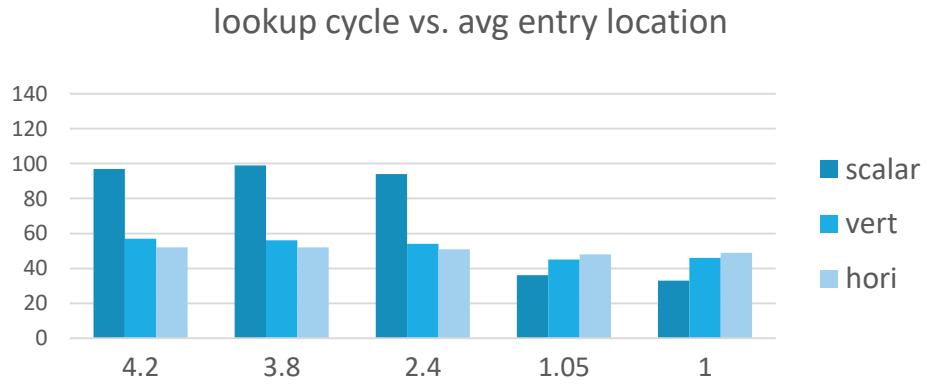
Analysis
- Software pipelining and batching

- Batching can enable us to prefetch hash table bucket for different lookup keys.
- Together with batching, software pipelining can further improve performance.
- Software pipelining + batching easily improve performance by 2X in our test case.



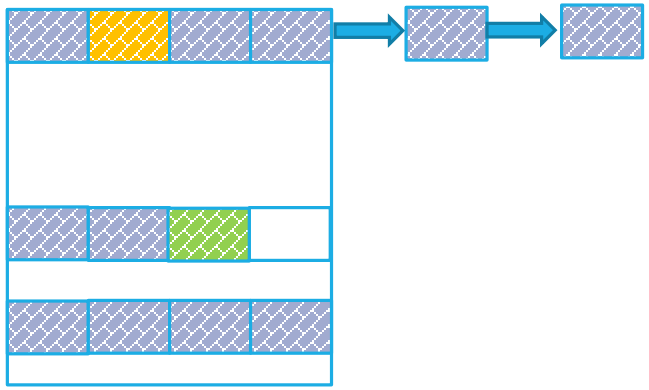
Analysis- Vectorization

- Besides using Intel AVX instruction for LRU operation, we can also use AVX instruction to perform signature comparison.
- We compare three mechanisms:
  - No vectorization.
  - Horizontal vectorization: compare one key's signature to all signatures in a bucket.
  - Vertical vecotrization: compare all key's signatures in a batch to different entries across different buckets.
- Observation:
  - Vertical or scalar better for low table utilization.
  - Horizontal better for high table utilization.
  - An adaptive method could benefit.



Future directions of Hash Table Design

- Cuckoo hash + extended linked list design
  - Linked list based hash table provides capacity guarantee.
  - Cuckoo hash table provides high table utilization and constant table lookup time.
  - The combination of both to achieve both capacity guarantee and better utilization.



- Adaptive vrouter
  - From the study, we found no single data structure could fit all use cases.
  - Runtime decision based on traffic patterns could benefit.
  - During runtime, a “learning” (e.g., trial and rank) phase to try various hash table data structures.

Conclusion

- We investigated multiple hash table algorithms and implementations in popular virtual switches.
- We analyzed various hash table designs and provide guide lines for different use cases.
- We proposed Intel AVX based LRU cache implementation and adaptive signature comparison.
- We proposed future directions on hash table design in virtual switches.