Parrot Hashing: Fast and Low-Memory Table Lookups for Network Applications With One CRC-8

Yi Liu, *Graduate Student Member, IEEE*, Shouqian Shi, Ruilin Zhou, *Graduate Student Member, IEEE*, Yuhang Gan, *Graduate Student Member, IEEE*, and Chen Qian, *Senior Member, IEEE*

***Abstract*—Key-value lookup functions have been widely applied to network applications, including FIBs, load balancers, and content distributions. Two key performance requirements of a lookup algorithm are high throughput and small memory cost. One limitation of existing fast network lookup algorithms is that they require multiple independent and uniform hash functions, which cost high computation time and might not be available on existing hardware network devices. Recently developed learned model hashing (LMH) proposes to use a linear machine learning model to replace hash functions to avoid hash computation, but they are not optimized for memory cost. We propose a novel network lookup method called Parrot hashing, which uses a learned model to distribute keys into different buckets and applies a simple perfect hashing method to resolve the collisions of the keys in a bucket. Parrot can be implemented with only one CRC-8, which is available on all network devices. We implement Parrot in three prototypes: a software program on end hosts, a software switch, and a FIB running on a hardware programmable switch. The experimental results show that Parrot achieves the highest lookup throughput on all three prototypes, compared to existing methods. Its memory cost is also significantly lower than that of LMH.**

***Index Terms*—Key-value store, perfect hashing, learned hashing, programmable data plane.**

I. Introduction

TABLE lookups serve as fundamental functions and design blocks of numerous network protocols and algorithms from the data link layer to the application layer. Most network lookups can be generalized as searching a key and getting a value that corresponds to the search key, such as FIBs [1], [2], [3], load balancers [4], [5], [6], [7], CDNs [8], and key-value stores [9], [10], [11].

All of these applications share two major performance requirements. First, the lookup function should be fast to support high network throughput or the line rate. Second, the memory cost should be minimized because these functions are located in the fast memory (e.g., TCAM) of network devices which are expensive and power-hungry. As a result, existing designs for network lookup functions focus on optimizing these performance metrics [12], [13], [14], [15], [16].

Received 12 September 2024; revised 20 May 2025; accepted 27 June 2025; approved by IEEE Transactions on Networking Editor J. Wu. This work was supported by NSF Grant 2322919, Grant 2420632, Grant 2426031, and Grant 2426940. *(Corresponding author: Chen Qian.)*

The authors are with the Department of Computer Science and Engineering, University of California, Santa Cruz, CA 95066 USA (e-mail: cqian12@ucsc.edu).

Digital Object Identifier 10.1109/TON.2025.3587475

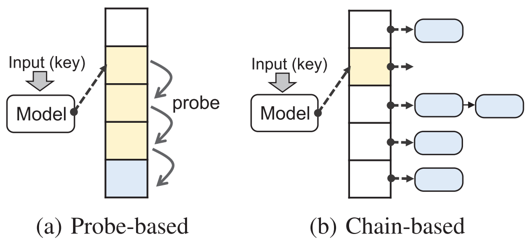


Fig. 1. LMH-based hash tables.

All of these designs rely on multiple independent universal hash functions, which introduces several inevitable problems including high computation time [2] and lack of sufficient independent hash functions on certain hardware devices [17]. For example, Broadcom switches support RTAG7 [18] and the Cisco Nexus 5500 Series [19] supports CRC-8, which do not satisfy the hash independence and uniformity requirements of the above designs [17]. Being a popular data structure, hash tables have been used as indices in various database systems as well as other computing applications. Resolving hashing collisions is the fundamental problem in the design of hash tables, because collisions might cause lower performance and/or errors for hash table lookups. Classic collision resolution methods include probing, chaining, and cuckoo hashing [20].

In database research, the learned model hashing (LMH) [21], [22], [23] was recently proposed to use a machine learning model to replace traditional hash functions for secondary indices. The idea of LMH is to train a model that approximates the cumulative distribution function (CDF) of all keys and predicts the position of a lookup key in a sorted array. Let each position of the array represent a bucket that can store the value corresponding to a lookup key. The array can then be considered a hash table where LMH replaces hash functions to calculate the position of a key. Since a trained model cannot distribute all keys evenly, collisions still happen. Similar to traditional hash tables, probe-based (Fig. 1(a)) and chain-based (Fig. 1(b)) collision resolution can be used. However, we realize that most studies about LMH [21], [22], [23], including a recent benchmark study [23], pay little attention to the memory cost.

We are curious about the feasibility of applying LMH for network lookups because it has no requirements for hash functions and provides high throughput. To this purpose, we conduct extensive empirical studies to evaluate LMH tables

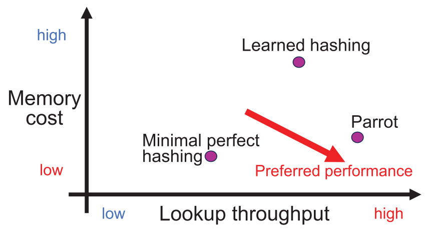


Fig. 2. Visualized comparison of hash tables.

and recent network lookup algorithms [16] on a public benchmark [23], [24]. Our key observations are as follows. LMH tables achieve higher lookup throughput because the learned models in the LMH tables compute faster than hash functions for distributing keys into buckets. However, recently-proposed *Minimal Perfect Hashing* (MPH) based network lookup [14], [16] costs much smaller memory because they do not need to store keys. Another advantage of LMH is that it supports range queries.

Based on the above observations, we motivate this research to find a solution that can achieve the best of two worlds: high throughput, no hash function requirements from LMH, and low memory from MPH. Our key innovation is to use a learned model to distribute keys into different buckets. When collisions happen in a bucket, we use perfect hashing to resolve all collisions into different slots – because perfect hashing can be easily implemented with a simple mapping function such as CRC-8 when the number of keys is small. Based on this idea, we present Parrot hashing, **a scalable, fast, memory-efficient, and dynamic lookup algorithm that has no requirement on the hash function support from network devices.**

We implement Parrot hashing in three prototypes: 1) a software program running on end servers and compatible with an existing benchmark for secondary indices [23], [24], which can be used for application-layer network lookups such as CDN and distributed storage; 2) a packet forwarding prototype implemented on hosts by the Intel Data Plane Development Kit (DPDK) [25] and 100GbE NICs, which can be considered an example of software network functions such as software switches and load balancers; 3) a packet forwarding prototype running on a Tofino programmable switch, which is an example for programmable network hardware. We believe the performance results of these prototypes are sufficient to demonstrate the advantages of Parrot in many network applications in various layers. The experimental results show that **Parrot hashing achieves the highest lookup throughput on all three prototypes**, compared to existing methods including LMH and MPH solutions. Parrot hashing costs higher memory than a recent MPH-based network lookup method [16], but smaller than other solutions. Unlike these methods, Parrot has no requirement on hash functions. A visualized performance comparison is shown in Fig. 2, where the right-bottom corner is the optimal design choice.

Our contributions in this paper are summarized as follows:

1) We conducted comprehensive empirical studies to compare LMH and MPH-based lookups, obtained important observations, and analyzed the feasibility of applying LMH for network lookups.

TABLE I

Recommended Hash Table Choice

Design preference	Hash table choice
Prioritize lookup throughput and small memory	Parrot hashing
Prioritize small memory, allow lower throughput	Ludo hashing
Prioritize short construction time	Learned hashing

2) Based on the new observations, we design Parrot hashing, a scalable, fast, memory-efficient, and dynamic lookup algorithm that can be implemented with only one CRC-8 function. By relaxing the restrictions of hash functions, Parrot hashing is more widely applicable than previous network lookup methods [12], [14], [15], [16]. Parrot hashing also supports range lookups.
3) We implement Parrot hashing in three prototypes and evaluate it with publicly available datasets [26] extensively. The results show that Parrot hashing provides higher throughput in all prototypes compared to existing methods.
4) Based on the evaluation results, we make recommendations for choosing hash tables as shown in Table I. When the application prioritizes high throughput and small memory, Parrot hashing is ideal. When the application wants to minimize memory while allowing lower throughput, MPH such as Ludo can be used. When the index receives many queries of non-existing keys, which might cause lower performance on MPH and Parrot hashing base databases, LMH or traditional hash tables can be chosen.

The rest of the paper is organized as follows. We present our observations from experimental comparisons and analysis of LMH and MPH in Section II. Section III presents the design and analysis of Parrot hashing. We evaluate Parrot hashing with extensive experiments in Section IV. Section V explains the related work, and we conclude our paper in Section VII.

II. LMH Vs. MPH: Experimental Studies and Observations

A. Learned Model Hashing (LMH)

LMH [21], [23] aims to reduce the collision rate of traditional hash functions by approximating the CDF of all keys and mapping keys into different positions in order. Learned models [21], [22], [27] leverage the order of keys to output associated positions so that all keys can be assigned to buckets in a consecutive array.

As learned models cannot map all keys to the distinct positions in the array, collisions still happen. LMH addresses the collisions – keys that are mapped to the same bucket – with similar designs as traditional hashing tables. The two representative collision resolution methods are probing and chaining similar to classic hash tables. One noticeable advantage of LMH is that model inference in many real datasets is much faster than hash function computation [23].

For all collision resolution methods mentioned above, keys must be stored in the table to recognize which bucket stores which key. We realized that most studies about LMH [21], [22], [23] pay little attention to the memory cost, and found

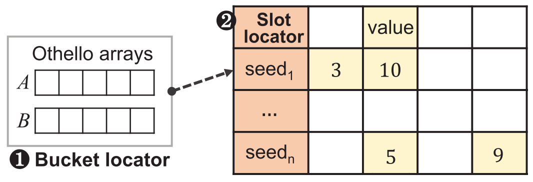


Fig. 3. Ludo hashing structure.

that the memory footprint is not optimized. However, LMH can be a potential solution to avoid the dependence on complex hash computations of existing network lookup solutions because it uses model inference instead of hash functions.

B. Minimal Perfect Hashing (MPH)

MPH is another line of research to reduce hash collisions [14], [16], [28], [29], [30], [31]. An MPH method constructs a function that maps n keys to different buckets *without collisions*. MPH achieves significant memory efficiency for network lookups because the table does not need to store keys to help the search – only values are required to be stored in each bucket. Note that storing keys may cost more space than storing the values in many network applications. For example, a MAC address (key) is 32-bit long and the return port index from FIB (value) can be as short as 8 bits (for 256-port switches); a 5-tuple (key) includes 104 bits while the server index returned by a load balancer (value) can be less than 20 bits.

To allow key dynamics, including key insertions and deletions, MPH may use $(1+\delta)n$ positions for n keys. The main idea of recent dynamic MPH for network lookups [14], [16] is first to divide the set of keys into groups, each of which includes a small number of keys. Then within each group, a hash function is found to map the keys in the group to different positions without collisions. The most recent solution of dynamic MPH is called Ludo hashing [16]. As shown in Fig. 3, Ludo hashing [16] first uses a data structure called Othello [15], a dynamic implementation of Bloomier filter [32], as the *bucket locator* to distribute keys into different buckets, each of which includes 4 slots. Then, in each bucket i Ludo uses brute force to find a hash seed s_i such that the hash function with s_i can map the 4 keys in the bucket to 4 different slots without collision. Hence, keys do not need to be stored in the table for collision resolution. The space cost of Ludo is $3.76 + 1.05l$ bits per key, where l is the length of the record value [16]. Ludo requires three independent hash computations, two for the bucket locator (Othello) and one for the slot locator. Ludo can be implemented for FIB and application-layer content lookups [16].

C. Experiments and Observations

We wonder whether LMH can be used for network lookup algorithms by understanding the tradeoffs between LMH and MPH. However, we realize that the recent benchmark study [23] that compares LMH with classic hash methods did not include the recent solutions of MPH. In addition, the benchmark study [23] did not show the memory cost comparison.

To this purpose, we have conducted experimental studies to compare LMH with recent MPH methods and analyze their tradeoffs. We show the results of the representative methods: chain-based LMH table with the RMI model [21] (whose throughput is higher than other LMH tables [23]) and Ludo. We use the same benchmark program as recent studies of LMH [23] do, as well as 20 million records from each of the public datasets WIKI and BOOK [26]. These datasets are application-layer names and we will use datasets including network addresses and 5-tuples in the performance evaluation sections of this paper. We run experiments on a server equipped with Intel(R) Xeon(R) CPU E5-2687W v4 at 3.0GHz and 32GB DDR4 memory. For each set of experiments, we conduct 1 million uniformly random lookup operations. The results of using application datasets and network addresses are similar.

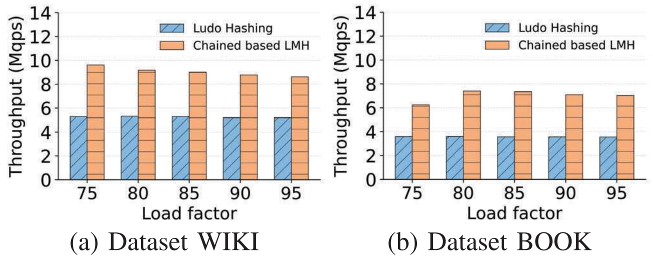


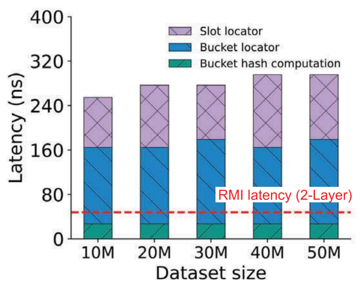
Fig. 4. Lookup throughput comparison.

As depicted in Fig. 4, we find that chain-based LMH tables always achieve higher throughput compared to Ludo by up to 2x in both datasets WIKI and book. We further analyze the throughput results by showing the time breakdown for querying a key in Fig. 5(a). With the dataset size varying from 10M to 50M, the accessing and computing in the bucket locator always take the most part (62%) of the latency. We also draw a red line to indicate the time used by a two-layer model RMI [21], the learned model used in LMH tables, to output the predicted bucket position of the queried key. The bucket locator of Ludo takes $3\times$ latency compared with RMI computation, due to more hash computation and random memory access for the bucket locator. We realize the root reason for LMH outperforming Ludo in throughput is that the learned models (RMI) distribute keys to buckets at a much faster speed – although the constructed bucket locator of Ludo can map exactly four keys to each bucket while RMI might cause random collisions.

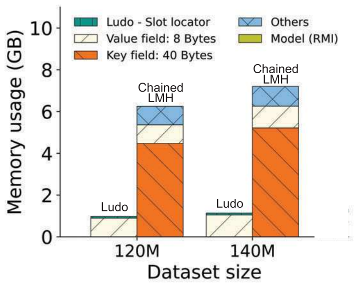
Thus, we have our first observation. **1) The bucket locator is the primary throughput bottleneck in MPH schemes.**

We then compare the memory cost of LMH-RMI based hash table and Ludo and give a breakdown of memory usage as shown in Fig. 5(b), where the key size is padded to be 40 Bytes with the dataset [33] size as 120 million and 140 million, respectively. In practical workloads [34], 40B is a typical average key length. We find that the majority of the memory cost comes from storing the keys, which almost makes up 80% of the usage. On the other hand, the memory cost of RMI only takes 1.608KB ($\approx 8^{-7}$ of all memory) because lots of keys can be indexed just with the slope and intercept of the approximated linear function. We obtain the second observation: **2) Avoiding storing full keys is the main reason for the memory efficiency advantage of Ludo hashing.**

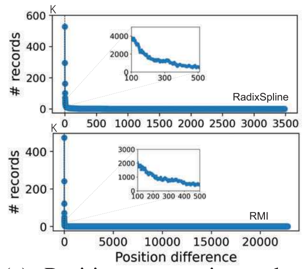
We further analyze the collisions when the learned models (e.g., RMI, RadixSpline [27]) map keys to buckets.



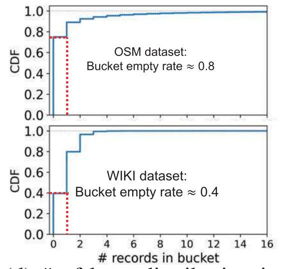
(a) Latency breakdown.



(b) Memory breakdown.



(c) Position errors in probe-based LMH.



(d) # of keys distribution in chain-based LMH.

Fig. 5. Observations: (a) Latency breakdown of data lookup in Ludo hashing w./ the varied dataset size. (b) Memory usage breakdown in probe-based LMH with the varied key lengths. (c) Predicted distance error distribution in probe-based LMH. (d) The CDF of the number of records distribution in all buckets of the chain-based LMH.

Fig. 5(c) shows the distribution of the position errors for 20M sorted keys in probe-based LMH with RadixSpline and RMI, respectively. We observe that there are only no more than 600K keys that could be found without probing. From the figure zoomed out, there are around 4000 keys with position errors up to 100. Unlike the chain-based hash table that can resolve collisions within the bucket, probe-based LMH tables accumulate prediction errors in consecutive buckets. The probing latency is often pulled back when querying a key, in which a large gap lies between the predicted and actual positions.

Fig. 5(d) shows the CDF of the number of keys in each bucket of chain-based LMH tables, when we set the load factor as 0.85 with RadixSpline for the OSM and WIKI datasets, respectively. For the WIKI dataset, which has been proven as a learned-friendly dataset with higher linearity, there are almost 40% of the buckets are empty. However, for the OSM dataset, there is a total of 80% empty buckets in the table. Also, over one-third of non-empty buckets have at least four records. The probing in the bucket along the collided chain also drags down the throughput of locating a key. We will show more throughput details in Section IV.

Our third observation is: **3) The "last-mile" search of LMH tables to fix the model errors is a major challenge of LMH that causes longer latency for probing or accessing linked buckets and higher memory cost because keys must be stored to resolve collisions.**

D. Idea of Achieving the Best of Two Worlds

By analyzing the advantages and disadvantages of both LMH-based and MPH-based tables, we realize LMH is strong at the initial key distribution method because the learned model is fast to compute and small in memory – and it does not rely on complex hash functions – while MPH is powerful for the "last-mile" search of resolving collisions of a small number of keys, which can use a constructed function to map keys to different slots without collision and avoiding storing the keys.

Based on these facts, we conducted research for a new network lookup design that uses LMH for initial key distribution to buckets and MPH for collision resolution within a bucket of keys. This design uses the powerful steps from both methods and can potentially achieve high throughput, low memory, and range queries, without relying on specific hash functions.

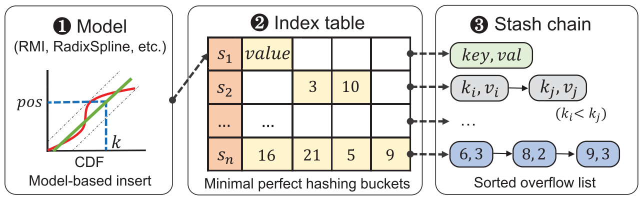


Fig. 6. Parrot hashing overview.

III. Design of Parrot Hashing

This section presents the new lookup algorithm called Parrot hashing and its corresponding implementation on programmable data planes.

A. Overview

Similar to other hash functions, Parrot hashing provides the following index function: For each *record* $< k, v >$ including the *key* k and *value* v (in practice, k is a network identifier and v is usually the index of forward port on hardware switches, the index of network function actions, or the index of a physical machine), if k is queried, Parrot hashing will return v.

Parrot hashing adopts an architecture design of three modules, as shown in Fig. 6. Module ❶ provides a pluggable interface for linear regression models used in LMH (e.g., RadixSpline [27], RMI [21]). Module ❷ is called the index table, which consists of a series of buckets, and each bucket contains four slots for holding the values of the keys and a seed. It organizes and places the values in the slots of each bucket based on the idea of Ludo, without storing keys. The keys are mapped into buckets indicated by the linear regression models according to the model-based insert policy [35]. If more than four records are mapped into one bucket, we will use the stash chain (Module ❸) to hold the records in the linked stash node. Note that the keys of records are kept in order between the buckets based on the properties of the learned models: For two consecutive buckets B_x and B_{x+1}, the biggest key k_1 of bucket B_x is guaranteed to be smaller than the smallest key k_2 of bucket B_{x+1}. In addition, record keys in the stash chain are also ordered by the keys. The values in the same bucket of the index table might not be ordered.

B. Index Table

The index table is the main structure to index most of the records. A bucket is a basic unit for locating records with the learned models' results. It includes four slots to store at most 4 record values and one seed (8 bits) used to realize the MPH for the four records in this bucket. No keys are stored in the index table. The number of buckets is set to $n/(4\epsilon)$, where ϵ is the *load factor* of the table and *n* is the total number of records, as we have four slots initialized in each bucket.

The load factor ϵ of a hash table refers to the ratio between the number of stored key-value pairs and the total number of available slots in the table. It reflects how full the table is. For example, if each bucket contains 4 slots, then the total number of buckets required to store *n* key-value pairs is $n/4\epsilon$. A lower load factor reduces the probability of collisions but increases memory usage, while a higher load factor improves space efficiency at the potential cost of performance.

Consider the first four keys of records inserted to a bucket are $\{k_0, k_1, k_2, k_3\}$. Parrot hashing uses brute force to select a specific seed s from 0 to 255, which can make a function H map the four keys into four distinct slots without collision. H can be a simple mapping function such as a CRC-8. We will describe how to implement a seeded CRC-8 for Parrot in Sec. III-G. In general, we use brute force to find s such that:

$$H(k_i, s) \neq H(k_j, s), \text{where } \; 0 \leq i, j \leq 3 \; and \; 0 \leq H(.) \leq 3$$

If the number of records is less than four, finding a seed to separate the records will take less time. Then, when we access a record value associated with key *k* in the bucket, we compute the hash $H(k, s)$ to locate the slot.

Compared with probing and chaining, realizing the slot locator with this MPH way is fast, and we will show the results in Section IV. Furthermore, even if some buckets overflow due to the uneven distribution of the learned model, index table can save the space to store the first four record keys, which is also a large portion of memory usage (>35%).

MPH could be realized with small overheads in the case of a small number of records [15], [16], [36]. The core idea of Parrot hashing is to distribute keys using LMH into different buckets with order-preserving and fast computation and then use MPH within each bucket to achieve low memory cost and high speed to locate the value.

C. Stash Chain

The learned model in module ❶ cannot map all keys to the index table with an even distribution of each bucket having exactly 4 keys. So we leverage the stash chain to hold the records overflowing each bucket. We have one pointer in each bucket of the index table to link to the head of a stash node, if more than four keys are mapped to the bucket. For a bucket with more than 4 keys, all other records are stored in the stash chain.

In the stash chain, each stash node contains one or more ordered records as shown in Fig. 6. Each stash chain stores the full keys of the records to realize order-preserving key insertion and reduce the time spent on queries along the chain.

D. Operations of Parrot Hashing

Lookup of keys. As shown in Fig. 6, there are one seed, four KV records in each bucket of the index table, and a pointer pointing to the first stash node. For a queried *key*, we first locate the bucket it exists based on the result of the learned model. We then access a slot indicated by the result of the hash function with the bucket's seed based on the rule of MPH. The value *val* we get from the slot "might be" the associated value for the queried key, because the target record might be stored in the index table or the stash chain. Before returning the data back, the lookup algorithm will check if there is an attached stash chain to this bucket.

If there is no overflow record in this bucket or the first key in the stash node is larger than *key*, we return the value *val* we get from the index table because *key* is absolutely not in the stash chain – recall that records in the stash chain should be ordered by their keys, hence all remaining keys must be even larger than *key*. Otherwise, the algorithm should search the stash chain until it finds the exact record sharing the same key with the queried *key*. Then, the algorithm returns the value found in the stash chain. This design helps Parrot hashing to achieve shorter query latency by avoiding searching the stash chain if *key* is smaller than the key in the first stash node of this bucket.

Algorithm 1 Parrot hashing Lookup

```
Index_Bucket:
    values array, seed, and stash_chain_ptr
Model model
procedure LOOKUP(key)
    bucket index ← model(key)
    slot index ← perfect hash (key and seed)
    ▷Slot hashing locator with seed
    $val$ ← values (bucket index, slot index)
    For each < $k$,$v$ > pair in stash node
      If $k$ is equal to key
      return $v$
        ▷Found the targeted key in stash chain
      else if $k$ is larger than key
        return val
      endif
end procedure
```

As shown in Algorithm I, there are one seed, four pairs of key-value in each bucket of the index table, and a pointer pointing to the first stash node (lines 4-7). For a queried *key*, we first locate the bucket it exists based on the result of the learned model (line 11). We then access a slot by the result of the hash function with the bucket's seed based on the rule of MPH (line 12). The value *val* we get from the slot "might be" the associated value for the queried key, because the corresponding record might be stored in the index table or the stash chain. The lookup algorithm should check if there is an attached stash chain to this bucket.

If there is no overflow record in this bucket or the first key in the stash node is larger than *key*, we return the value *val* we get from the index table because *key* is absolutely not in the stash chain – recall that records in the stash chain should be ordered by their keys, hence all remaining keys must be even larger than *key*. Otherwise, the algorithm should search the stash chain until it finds the exact record sharing the same

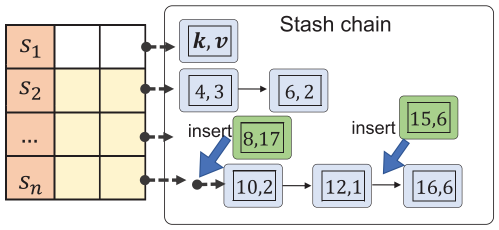


Fig. 7. Insert keys into stash chain.

key with the queried *key*. Then, the algorithm returns the value found in the stash chain (lines 14-19). This design helps Parrot hashing to achieve shorter query latency by avoiding searching the stash chain if *key* is smaller than the key in the first stash node of this bucket.

Insertion/update of keys. Similar to key lookups, the learned model will first compute the corresponding bucket when inserting a record with the key *key*. During the construction of the Parrot hashing table with bulk load, all records are inserted in order, and the first four records in the index table are the smallest in their buckets. The algorithm inserts the record value into the index table and generates a new seed to make the hash function distribute these records' keys into distinct slots without collision. If the four slots of the bucket are full, the inserting record should be appended and linked into the stash chain in order. Note that all keys are kept during bulk loading and will be discarded after that. For the individual record insertion after bulk loading, the inserting record key and value will be directly inserted into the stash chain with the order preserved.

As shown in Fig. 7, provided there is one record per stash node, the record $\{key, val\}$ =$\{$15,6$\}$ is mapped to the bottom bucket by the learned model. Then, the algorithm will visit the stash chain to find a position that maintains the order of the keys. Eventually, it should be inserted between the records $\{$12,1$\}$ and $\{$16,6$\}$. If the inserted key is smaller than the key in the first stash node, we will insert it as the first stash node. For example, the record $\{$8,17$\}$ will be inserted into the first position of the stash chain. Even if there is an old value for key 8 in the index table, it will not influence the lookup result because the newly inserted value will be stored with the full key in the stash chain.

When the exact key cannot be found in the stash chain during a data update operation, we treat the update as a data insertion. If a stash node's record key matches the update key, the record value is directly updated.

As the number of keys increases, memory usage rises due to the additional data insertions and updates. Once the memory usage in the current Parrot hashing table reaches a threshold – we set 1.2 times of the original table size – a reconstruction will be triggered. During reconstruction, data insertion will be temporarily halted, and the corresponding bucket will be locked. However, data lookups that map to unlocked buckets can continue using the old hash table. These data requests will be cached in an operation list and processed once the rebuilding is complete.

Range lookups. Although not our main focus in this work, range lookups are among the advantages brought by learned models. Range lookups in network lookups can be used for application content search. One might think that range lookups can be used to implement IP prefix matching, but we have not explored this direction and will leave it for future work. Parrot supports two kinds of range lookups. One is $scan(low\_key, high\_key)$, and Parrot will return the values whose key is between low_key and $high\_key$. The other one is $scan(low\_key, len)$, and a value list of length *len* starting from the low_key would be returned.

The starting key low_key will be located based on the lookup algorithm, and then the following *len* records or records that are smaller than $high\_key$ should be retrieved. If the starting key is found in the index table, then all four records in the bucket and records in the stash chain should be returned because the order is not maintained within the index table due to the MPH property. Note that some records in the index table bucket might not be within the range of the query, so when the data items are retrieved from the database, those items should be excluded in the final results of the range query. If the starting key is found in the stash chain of a bucket, then the records that are in the stash chain and before the starting key should not be returned because the order in each stash chain is preserved. After the algorithm traverses a bucket and has not reached the high bound or enough records, it should move to the next bucket because the keys in consecutive buckets are order-preserving.

Deletion of keys. The deletion operation in Parrot hashing can be implemented as records lookup and mark the data for this record as invalid. If the record is found in the index table, we remove the value in it. If it is found in the stash chain, we can remove the stash node (if empty) and release the space.

E. Analysis

In this section, we provide the theoretical analysis of the time complexity of lookup and insertion operations, as well as the theoretical estimation for the memory cost of Parrot hashing.

Expected records in each bucket. Provided we have m buckets in the index table with the load factor as ϵ, and there are n records in a dataset to be inserted. We have $m = n/(4\epsilon)$ where ϵ is the load factor and each bucket contains four slots. Assume all records keys are $k_0, k_1, .., k_n$, and the learned model will output the assigned bucket sequence as $b_0, b_2, .., b_{n-1}$, where $b_i \in [0, m)$ for all i. The gaps between each consecutive record's bucket sequences are expressed as g_i, and the bucket number can be expressed as $b_i = \sum_{u=0}^{i} g_i$. It is reasonable to assume g_i's distribution is i.i.d [23] with the probability density function (PDF) as $f_G(z)$, then the PDF function of b_i can be expressed as the multiple convolution of $f_B^{*i}(b) = f_G(z_0) * f_G(z_1) * \cdots * f_G(z_i)$ based on renewal process.

Provided $N(b) = \sum_{b_i<b} b_i$ refers to the number of record keys that appear in the first b buckets.

The probability for a bucket b to have exact t records is:

$$P(b,t) = P(N(b) - N(b-1) = t) = \sum_{i=0}^{n-t} P(N(b-1) = i)P(N(b-(b-1)) = t) \quad (1)$$

The probability $P(N(b-(b-1))=t)$ can derived from the PDF of bi as $f_B^{*t}(1)-f_B^{*(t+1)}(1)$. Thus, the expected number of record in bucket b is $\sum_{t=5}^{\infty} P(b,t)$.

Lookup complexity. In Parrot hashing, we adopt a two-layer linear regression model as the learned index (e.g., an RMI). As a result, locating the bucket for a given key requires at most two linear computations involving slopes and intercepts. The final step—referred to as the "last mile" search—occurs within the identified bucket and depends on the number of colliding records.

In the best case, each bucket contains no more than four records, and the minimal perfect hash (MPH) function distributes keys into distinct slots without collisions. In this case, we can directly compute the slot position using the associated hash seed, yielding a best-case lookup time of $O(1)$.

In the average case, the two-layer RMI model involves only two linear computations, each of constant cost. The average expected number of records in a bucket is given by $l = \frac{1}{n}\sum_{b=b_0}^{b_{n-1}}\sum_{t=5}^{\infty} P(b,t)$, where $P(b,t)$ denotes the probability that bucket b contains t records, as shown in Formula 1. Consequently, the overall expected time complexity for a lookup operation is $O(1+l)$, where l is the average number of records in a bucket.

Insertion complexity. For the in-place insertion shown in Fig. 7(b), we first need to find the position to insert the new records in the stash chain and allocate space to link the new records. The best case is there is no stash node in this bucket, and we can insert it with a new stash node linked to the bucket, and the complexity is $O(1)$. In an average case, the time complexity for inserting a record would be $O(l)$, where l is the average number of records in a bucket.

Memory usage. Assume we have four slots in each bucket before allocating extra space for stash nodes. Each bucket consists of a seed (8-bit), four slots for record values, and a pointer referring to the stash chain (64-bit).

Provided the length of the record key and value are l_k and l_v, respectively. Each bucket memory usage would be $8+64+4\cdot l_v = 72+4\cdot l_v$ bits. Thus, the memory usage for index table is $(72+4\cdot l_v)\cdot n/(4\cdot\epsilon) = n\cdot(18+l_v)/\epsilon$ and each record costs $(18+l_v)/\epsilon$ bits in the best case. The memory cost in the stash chain depends on the number of colliding records in each bucket. We get the probability that there are exact t records in one bucket as $P(b,t)$ in Formula 1, for any buckets with records number larger than four $(t \geq 4)$, at least $(t-4)$ stash nodes will be allocated to have collided records. Each stash node contains a KV record and a pointer linking the next node, so its memory cost is (l_k+l_v+64) bits. In addition, based on the expected number of records in Formula 1, the expected number of stash nodes in bucket b is $\sum_{t=5}^{\infty} P(b,t)$. Then, the memory cost of the stash nodes in all buckets is:

$$\sum_{b=0}^{m}\sum_{t=5}^{\infty}((l_k+l_v+64)\cdot P(b,t)) \tag{2}$$

F. Concurrency Design for Network Lookups

We design Parrot hashing as a dynamic key-value lookup engine for the "single writer, multiple reader" model. To ensure efficient lookup operations during concurrent access, we require all modifications to the same key to appear atomic to the lookup threads. Additionally, we enforce atomicity for sequential writes within a single bucket to maintain consistency for the lookup threads.

We implement the optimistic locking scheme introduced in MemC3 [37] for concurrency control. We reduce the size of the version counters from being proportional to the number of buckets to a constant 8192. Each counter is composed of 8 bits. At first, all version counters are set to zero. When a key insert operation starts, the hashed counter increases by one to an odd number. The following key operations sharing the same lock counter are delayed until the counter value becomes even. After completing a key operation, we will recheck the counter to detect any modifications that occurred between the initial and final counter reads. If the counter has changed, we will redo the key operation again.

G. Implementation With Crc-8

The index table requires the MPH functions to be random enough to find a seed to distribute four different keys into distinct slots without collision. Ludo hashing [16] leverages Murmurhash as the slot locator in their design. However, the main challenge is that multiple networking hardware only supports CRC-8 checksum or RTAG7 functions like Broadcom switches [38] and Cisco Nexus 5500 [19].

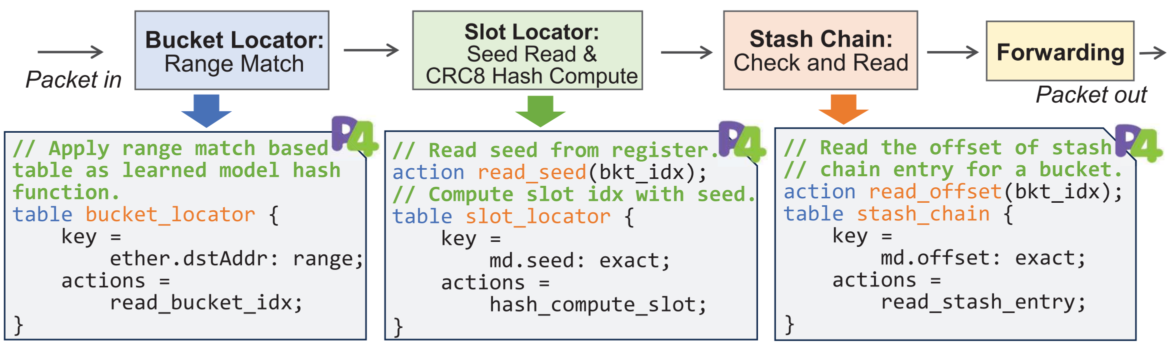
In our initial effort, we use CRC-8 function [39] with polynomial as 0×31 to distribute the keys in the index table by brute-force searching an 8-bit seed. It turns out we have over 15% buckets that cannot be distributed without collision with CRC-8. Provided the CRC-8 result of a given key is $c_7c_6..c_0$, we then propose to add the c_7c_6 and c_4c_3, and then use the least two bits as the slot index. This shuffle makes the CRC-8 more random and keeps the failure buckets rate within 2.7% in the experiments. All the keys in the failed buckets will be appended to the fallback table for additional checking. We can summarize our modified CRC-8 computation for MPH in a bucket as follows.

$$\begin{aligned}\text{perfect\_hash}_{\text{bucket}} = (&\text{crc8}(\text{key}, 0\times 31) \gg 3+\\ &\text{crc8}(\text{key}, 0\times 31) \gg 6)\ \&\ 0\times 03\end{aligned} \tag{3}$$

H. Implementation on Programmable Data Plane

We describe the data plane design for implementing Parrot hashing on a programmable switch. The Parrot hashing algorithm is portable and can be implemented on programmable switch ASICs or other network hardware. We implement Parrot hashing in P4, and the evaluation results in Section IV-C demonstrate that Parrot hashing can run on a Tofino switch at line rate.

The computation of learned models with match-action table. In Parrot hashing, learned models are used to map keys into a bucket, and its calculation consists of linear computations with slope and intercept. The input key is the extracted destination Ethernet address. The main challenge is that the multiplication between key and slope is not fully supported in Tofino 1 because float multiplication requires more ALUs than it provides.


Packet in
Bucket Locator: Range Match
Slot Locator: Seed Read & CRC8 Hash Compute
Stash Chain: Check and Read
Forwarding
Packet out

```
// Apply range match based
table as learned model hash
function.
table bucket_locator {
    key =
        ether.dstAddr: range;
    actions =
        read_bucket_idx;
}
```

```
// Read seed from register.
action read_seed(bkt_idx);
// Compute slot idx with seed.
table slot_locator {
    key =
        md.seed: exact;
    actions =
        hash_compute_slot;
}
```

```
// Read the offset of stash
// chain entry for a bucket.
action read_offset(bkt_idx);
table stash_chain {
    key =
        md.offset: exact;
    actions =
        read_stash_entry;
}
```

Fig. 8. The main modules of Parrot hashing in Tofino pipeline.

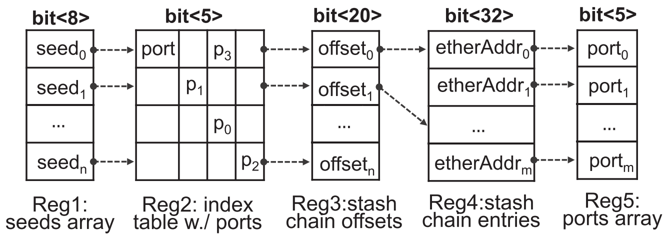


Fig. 9. The registers defined in the Tofino pipeline.

We observed that the learned models map all keys to buckets in order, and we can get to know the key ranges in each bucket after the construction of the index table before discarding the keys. Thus, we leverage the range match in the match-action table by setting the key ranges in each bucket. Then, we can bypass the float computation of learned models by directly mapping keys into buckets. The pseudocode is shown in Fig. 8.

In Fig. 9, we show the registers utilized within the Tofino pipeline. Upon a packet's arrival, the destination address is extracted, initiating a range match to determine the bucket index in the first stage. Subsequently, the seed associated with the bucket is retrieved from register *Reg1*. The port value is obtained from the index table *Reg2* by computing the CRC-8 with the seed. Register *Reg3* tracks the starting index of stash nodes packed in *Reg4*. If no stash node is present in the bucket, the offset value is set to `0xFFFFF`, indicating that subsequent register reads can be skipped. If the offset indicates at least one stash node exists, the low 32-bit Ethernet address stored in *Reg4* is read and compared against the target port destination address in *Reg5*. Due to no support for loop logic and the limited number of stages provided in the Tofino pipeline, at most two stash node entries can be read. The keys from terminated stash chains are added to the overflowed list.

IV. Evaluation

In this section, we present the experimental results to demonstrate the performance of Parrot hashing and compare it to other methods. We conduct experiments on a software benchmark program to illustrate that Parrot hashing can achieve high performance in lookup latency compared to other state-of-the-art solutions on end servers. In addition, we implement Parrot-based FIBs as software switches running on hosts using Intel DPDK. Furthermore, we evaluate a Parrot-based FIB prototype on an Intel Tofino programmable switch, based on the design mentioned in Section III-H. The throughput results of packet forwarding confirm the line rate of our Parrot hashing and demonstrate the feasibility of only using CRC-8 for building network lookups.

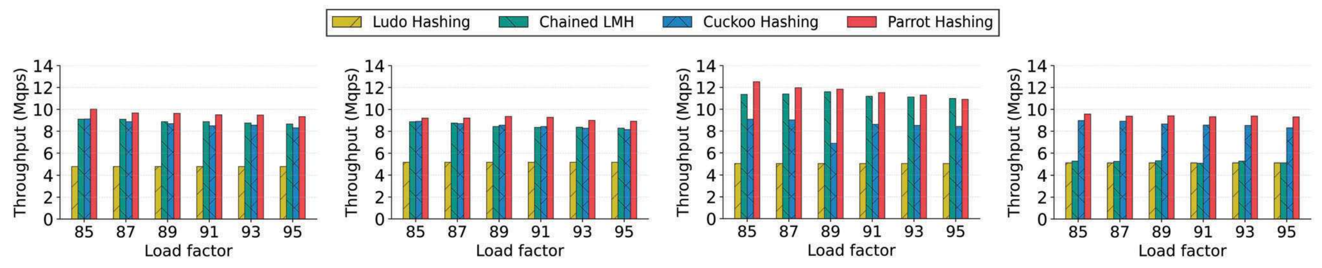
A. Algorithm Evaluation

We evaluate Parrot hashing in a hashing benchmark program with practical datasets and compare it against the chained-based LMH table and other state-of-the-art hash table schemes.

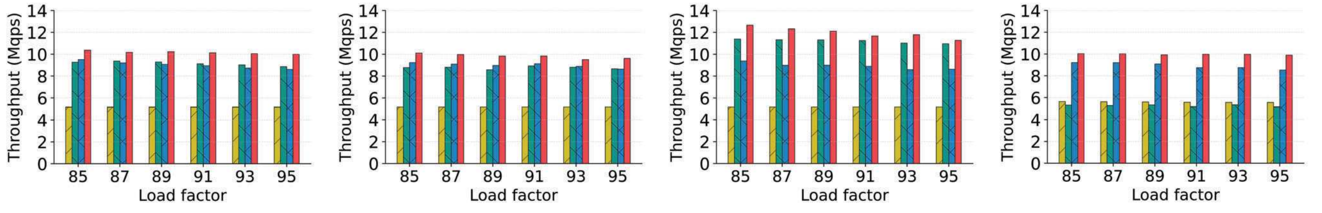
Setup. The experiments are run on a workstation with Intel Xeon CPU E5-2687W v4 at 3.0GHz, 32GB 2400MHz DDR4 memory, and 32MB LLC. The secondary storage device equipped with the workstation is an SK Hynix SC311 SATA SSD with a capacity of 1T. We use Ubuntu 18.04 LTS with Linux Kernel 4.15. As a common setting in the existing SOSD benchmark program [24] that has been used to benchmark LMH and classic hashing methods, we use a single thread unless otherwise stated.

Comparing methods. We compare Parrot hashing with the state-of-the-art solutions of both LMH and MPH that include the following methods. 1) Chained-based LMH table. They have been demonstrated to achieve the highest throughput among existing hash tables in many practical datasets [24]. The models we used for training the CDF of keys are two layers RMI [21] models We use a chain-based LMH table with a bucket capacity of 1, which is the default setting of it [23]. 2) Ludo hashing [16]. It is a recent MPH index solution with a lookup in $O(1)$ time and the smallest space cost among dynamic perfect hashing. We use the source code of Ludo hashing provided by the author on a public webpage [40]. 3) Cuckoo hashing [20]. It is a state-of-the-art hashing technique that addresses hash collisions by placing each key in one of multiple possible locations. Cuckoo has been widely applied to network lookups [6], [12], [25]. We use the (2,4)-Cuckoo hash table, meaning each bucket includes 4 slots and there are two alternative buckets.

Datasets. They are (1) WIKI (application-layer identifiers), time stamps of users' updates from Wikipedia. (2) BOOK (application-layer identifiers), which are keys for indexing the popularity of books on Amazon. (3) MAC (link-layer identifiers), randomly generated MAC addresses (48-bit), and each of them is associated with a port (8-bit). (4) 5-TUPLE (transport-layer identifiers), randomly-generated 5-tuples for TCP connections. WIKI and BOOK are publicly available datasets [26] used by many existing evaluations [23], [35], [41]. We generate the returned values in 16 bits for datasets



(a) Dataset: MAC; Dist: Uniform. (b) Dataset: TUPLE; Dist: Uniform. (c) Dataset: WIKI; Dist: Uniform. (d) Dataset: BOOK; Dist: Uniform.



(e) Dataset: MAC; Dist: Zipfian. (f) Dataset: TUPLE; Dist: Zipfian. (g) Dataset: WIKI; Dist: Zipfian. (h) Dataset: BOOK; Dist: Zipfian.

Fig. 10. Lookup throughput with uniform and zipfian workloads.

WIKI, BOOK, and 5-TUPLE. As indicated by the past work [35], [41], the key distribution of dataset WIKI shows better linearity, and there will be fewer collisions in each bucket computed by the learned models compared with that of dataset BOOK.

Configuration. In our default experiment setting, we first load 80M distinct keys from the original datasets after removing duplicates. Each query set contains 10 million keys generated from the loaded keys in uniform or Zipfian distribution. We run three query sets for each data point and report the average.

1) Lookup Throughput on End Systems: We evaluate the lookup throughput by varying the load factor of these hash tables. Fig. 10 shows the performance comparison with load factors from 0.85 to 0.95 for the four datasets. Figs. 10(a-d) are results on the query sets sampled in a uniform distribution, and Figs. 10(e-h) are those with a Zipfian distribution.

We find that Parrot hashing can always achieve the highest throughput compared with other baselines. In uniform workload, Parrot hashing outperforms Ludo hashing, Cuckoo hashing, and chained LMH by 2.08×, 1.10×, and 1.09× on the lookup throughput of dataset MAC with a load factor of 0.85, respectively. For WIKI lookups, Parrot hashing outperforms these three by 2.38×, 1.37×, and 1.10×, respectively. The results in the Zipfian workload show similar patterns. The experimental results demonstrate that Parrot hashing provides the highest lookup throughput across different datasets.

We observe Parrot hashing and Chained LMH show their own highest lookup throughput in the dataset WIKI because the key distributions of the dataset WIKI show better linearity, and there are fewer key collisions after keys are mapped to buckets. Parrot hashing can locate the first four keys in the bucket with one hash computation, but Chained LMH has to traverse keys and check them individually. Additionally, Ludo hashing and Cuckoo hashing maintain stable lookup throughput on their own with varied datasets and load factors because the computation and memory accessing required are the same.

2) Memory Usage: In this section, we show the memory cost of the compared lookup schemes.

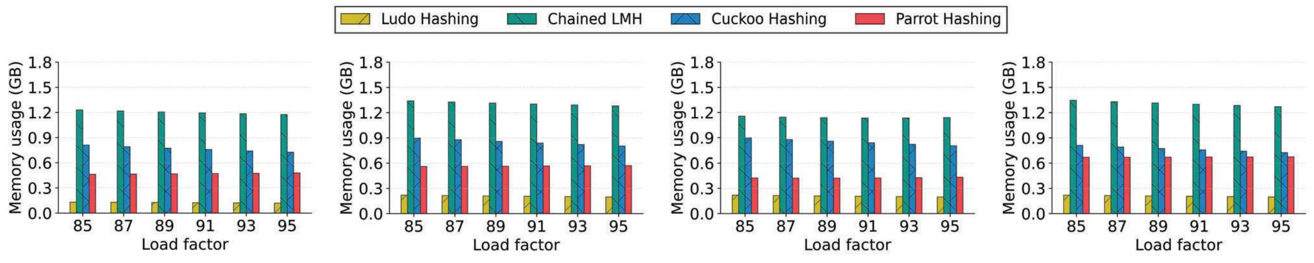
Fig. 11 show the memory cost of these methods by varying the load factor from 0.85 to 0.95, and each of these tables includes 80M items. Note that the memory usage of Chained LMH and Parrot hashing include all the learned models. Ludo hashing always achieves the lowest memory overhead for all datasets since it leverages MPH to manipulate the keys' placement in the bucket and slot to avoid extra memory costs.

As shown in Fig. 11(c), for the dataset WIKI, the memory usage of Parrot hashing is 1.94×, 0.36×, and 0.47× of that in Ludo hashing, Cuckoo hashing, and Chained LMH, respectively, when the load factor is 0.85. Compared with Chained LMH, the memory usage benefit comes from the space for storing the keys of the first four items in each bucket. Regarding the dataset BOOK, the memory usage of Parrot hashing is a bit larger than that of dataset WIKI. The reason is that the learned models map keys into buckets with more collisions in the dataset BOOK, and the amount of stash chains are used in Parrot hashing. Thus, the Parrot hashing consumes large memory resources with pointers and stash nodes.

In addition, we set the load factor to 0.95 and evaluated the memory cost by varying the number of records from 16M to 96M. As illustrated in Fig. 12(a-d), we observed that the memory usage of all tables increases linearly with the number of records. Comparatively, Ludo hashing incurs the lowest memory overhead among all schemes, while Chained LMH costs the most. Parrot hashing outperforms Cuckoo hashing and Chained LMH in all datasets by not storing full keys of the first four items in each bucket.

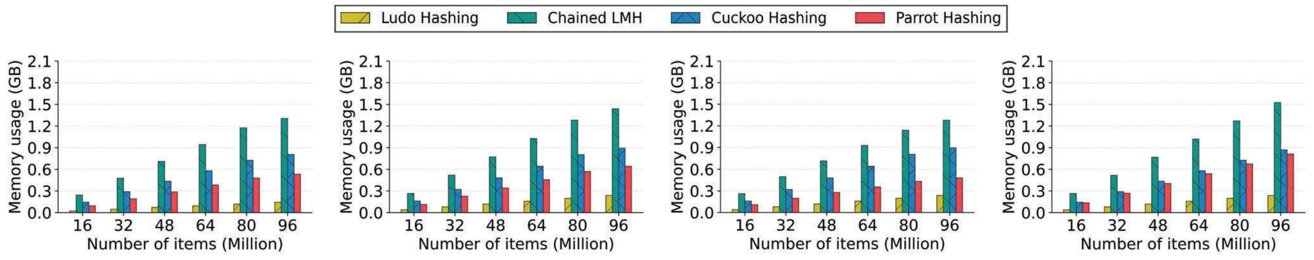
3) Range Query Performance: Both LMH and Parrot hashing support range queries to hash tables. Fig. 13 shows the evaluation results of range queries under two different datasets. We measure the time consumed from starting with a range query request to ending with a returned list of values from the index.

In chained LMH, all records are sorted and placed in buckets and link nodes because all keys are inserted into the bucket



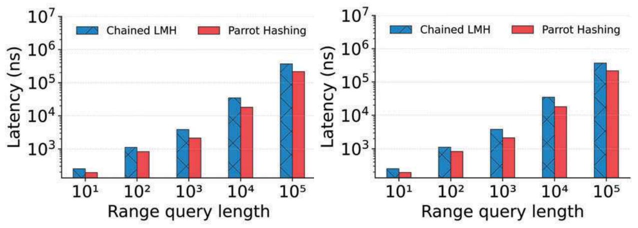
(a) Dataset: MAC. (b) Dataset: 5-TUPLE. (c) Dataset: WIKI. (d) Dataset: BOOK.

Fig. 11. Memory usage with the varied load factors.



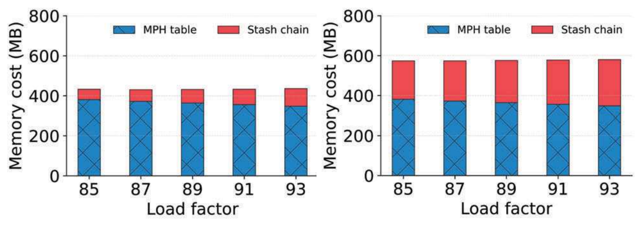
(a) Dataset: MAC. (b) Dataset: 5-TUPLE. (c) Dataset: WIKI. (d) Dataset: BOOK.

Fig. 12. Memory usage with the varied number of records.



(a) Dataset: WIKI. (b) Dataset: BOOK.

Fig. 13. Range query latency on dataset WIKI and BOOK.



(a) Dataset: WIKI. (b) Dataset: MAC.

Fig. 14. Memory cost breakdown for Parrot hashing.

TABLE II
CONSTRUCTION TIME

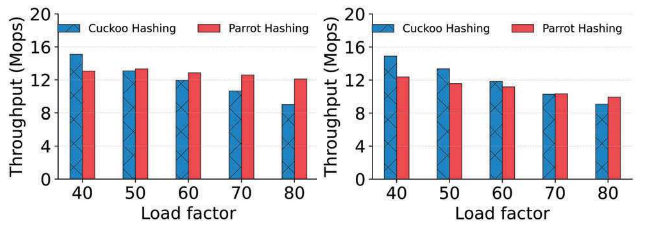
Datasets	Ludo Hashing	Chained LMH	Cuckoo Hashing	Parrot Hashing
WIKI	240.9s	4.9s	16.8s	9.8s
BOOK	293.8s	5.6s	16.2s	9.5s

indicated by the model's output. When there is range query $\{k_s, len\}$, we go directly to the bucket where k_s is stored and then read the following *len* records out in the table. Similarly, all keys are sorted between buckets in Parrot hashing. Thus, the range queried keys could be retrieved after we locate the starting key k_s. The only difference is the record key order is not maintained in the index table, so there are at most four or eight extra records that would be read out as indicated in III-D.

4) Construction Time: We evaluate the construction time in this section. Each table is constructed with 64 million records bulk loaded, and the load factor is set as 0.85. The construction time results are shown in Table II. Chained LMH tables can complete the construction in around 5 seconds, and Parrot hashing spends a bit more than that ($6 \sim 10$s), which is still very fast. On the other hand, Cuckoo Hashing takes over 15 seconds to complete the construction by spending time on making the hash table to accommodate all items without using extra space. Ludo hashing takes a much longer time (>100s) to construct their tables because both the bucket locator and slot locator will be reconstructed multiple times to find the proper hash functions to distribute keys into buckets and slots without collisions, as well as constructing an acyclic bipartite that can accommodate different records in its included Bloomier filter as the bucket locator.

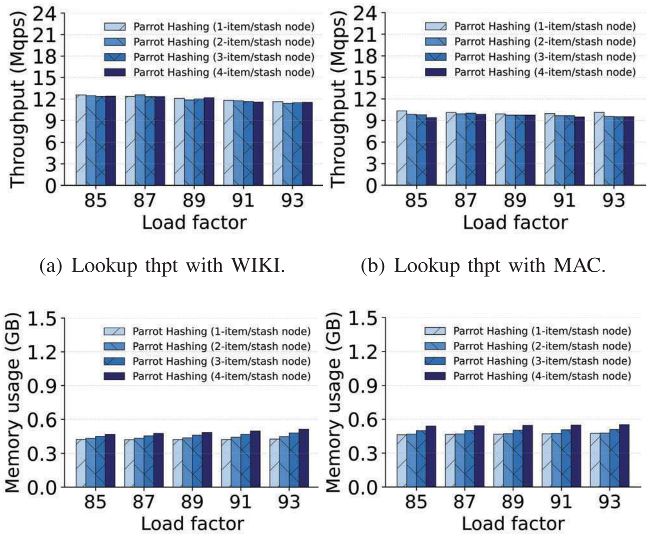
5) Memory Cost Breakdown for Parrot Hashing: In this section, we break down the memory cost of Parrot Hashing using two different datasets, highlighting the space usage of the MPH table and the stash chain. As shown in Fig. 14, for the WIKI dataset, which has a more linear key distribution, the RMI model is able to distribute keys more evenly across buckets. As a result, the stash chain only accounts for 18% of the MPH table size. In contrast, the MAC dataset has a non-linear key CDF, leading to more collisions and a higher stash chain overhead for collision resolution.

6) Evaluation With a Range of Load Factors: In this section, we evaluate Parrot hashing against state-of-the-art Cuckoo hashing across a range of load factors from 0.4 to 0.8. As shown in Fig. 15, the throughput of Cuckoo hashing decreases as the load factor increases. In contrast, Parrot Hashing maintains higher performance under high load factors, outperforming Cuckoo hashing when the load factor exceeds 0.7 in both datasets.



(a) Dataset: WIKI. (b) Dataset: MAC.

Fig. 15. Comparison with Cuckoo Hashing across a range of load factors.



(a) Lookup thpt with WIKI. (b) Lookup thpt with MAC.

(c) Memory usage with WIKI. (d) Memory usage with MAC.

Fig. 16. Influence of number of items in each stash node.

7) *Varying the Number of Items per Stash Node:* In this set of experiments, we vary the number of records in each stash node of Parrot hashing from 1 to 4 and insert 80M items at the beginning of the benchmark. For the dataset WIKI, we can see the lookup throughput is getting down when the load factor increases from Fig. 16(a), and the scheme with different numbers of items per stash node is close. Fig. 16(c) shows the memory cost for the dataset WIKI, and with the 4-item per stash node setting costing the most memory (slightly higher than others). For the dataset MAC, as shown in Fig. 16(b), all variants have similar throughput with different load factors and the 1-item oer stash node setting can always achieve the highest throughput. As shown in Fig. 16(d), the setting with one record per stash node costs the lost memory, which is a bit higher than the WIKI results shown in Fig. 16(c). As we mentioned before, learned models can distribute keys from the dataset WIKI more evenly and with fewer collisions, compared to that of the dataset MAC. If we set the number of items in the stash node to 4, we have to allocate the memory for at least 4 items, even if there is an overflow item. Thus, memory usage is larger than in the setting where each stash node only contains one item.

B. Software Switch Prototype With Dpdk

In this section, we show the results of the software switch prototype implemented by DPDK.

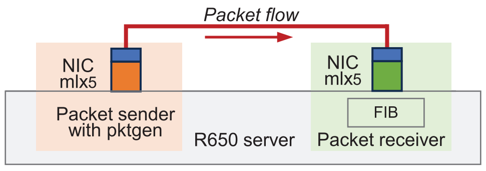
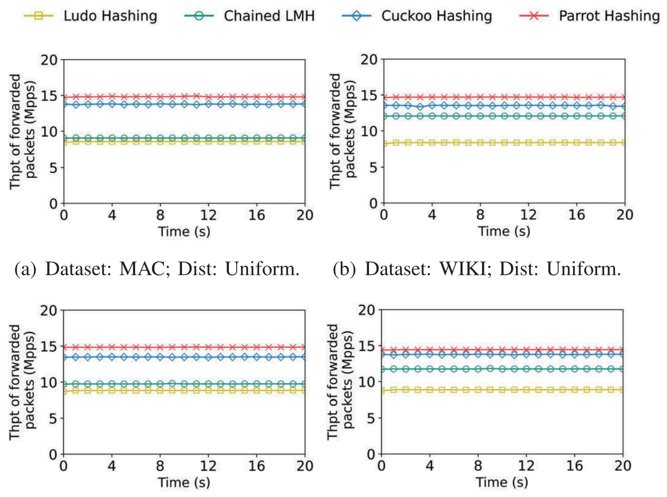


Fig. 17. Testbed for evaluating FIB via DPDK.



(a) Dataset: MAC; Dist: Uniform. (b) Dataset: WIKI; Dist: Uniform.

(c) Dataset: MAC; Dist: Zipfian. (d) Dataset: WIKI; Dist: Zipfian.

Fig. 18. Lookup and forwarding throughput with DPDK.

Setup. We use a workstation configured with an Intel Xeon Silver 4314 CPU@2.40GHz, 160GB 2133MHz DDR4 memory, and 48MB LLC as a testbed. We also have two Mellonax ConnectX-5 NICs installed in two RISE interfaces of the server, respectively.

As illustrated in Fig. 17, the left NIC serves as the packet sender, utilizing `pktgen` to transmit packets from a.pcap file generated by different dataset workloads. The FIB units act as receivers on separate cores, bound to the other NIC. These FIBs are implemented using chained LMH, Ludo hashing, Cuckoo hashing, and Parrot hashing. Upon receiving a packet, the receiver extracts its Ethernet address and performs a lookup in the corresponding FIB to determine the outgoing port for forwarding.

For each baseline, we load 64M distinct records from the WIKI and MAC datasets to construct the FIBs. We then sample 10K keys from these keys in uniform and Zipfian distributions and pack them into a.pcap file to feed the packet sender. Fig. 18 illustrates the throughput performance of FIB lookup and forwarding of all approaches. Across all scenarios, Parrot hashing consistently achieves the highest packet forwarding throughput. For instance, with the dataset WIKI, Parrot hashing outperforms Cuckoo hashing, Ludo hashing, and chained LMH by 1.08×, 1.21×, and 1.76× as depicted in Fig. 18(b), respectively. The throughput performance of chained LMH notably declines with the dataset MAC, and we attribute it to increased record collisions in buckets compared to the dataset WIKI.

C. FIB Prototype on a Hardware Switch

We implement and deploy a Parrot-based FIB on a Wedge 100BF-32X switch with programmable Tofino 1, using

TABLE III
COMPARISON OF EXISTING DATA LOOKUP STRUCTURE COULD USED FOR FIB

Method	Main structure	Lookup latency	Complex hash function required?	Range query	Memory usage
Ludo hashing [16]	Hash table	High	✓	✗	Lowest
Cuckoo hashing [20]	Hash table	Middle	✓	✗	Large
Probe-based LMH [21], [23]	Learned Hash table	High ($\geq$ 1K ns)	✗	✓	Middle
Chained-based LMH [21], [23]	Learned Hash table	Middle	✗	✓	Large
Parrot hashing (this work)	**Learned Hash table**	**Low**	✗	✓	**Low**

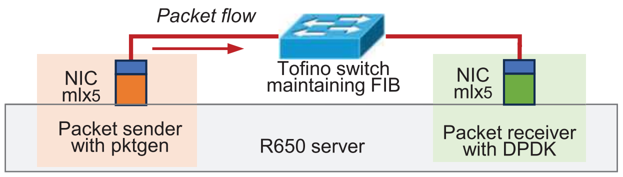
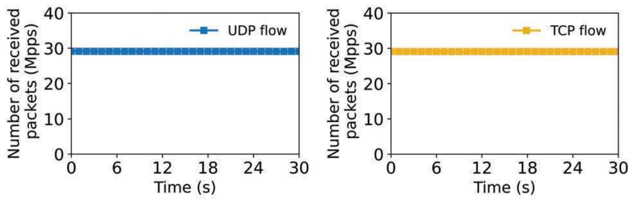

Packet flow
Tofino switch maintaining FIB
NIC mlx5
NIC mlx5
Packet sender with pktgen
R650 server
Packet receiver with DPDK

Fig. 19. Tofino programmable switch testbed.



(a) Packets forwarding throughput with UDP flow. (b) Packets forwarding throughput with TCP flow.

Fig. 20. Packets forwarding throughput with Tofino switch.

P4studio version 9.6.0 for experimentation. The two Mellanox NICs are connected to two 40GB ports on the switch. In the setup shown in Fig. 19, the left NIC saturates the bandwidth (40 Gbps) by sending packets at a rate of 32.3 million packets per second, each with a size of 155 bytes. The switch extracts the destination Ethernet address of each packet and looks up the outgoing port in the FIB. The right NIC keeps polling the NIC to receive the forwarded packets.

The Parrot hashing implementation on Tofino comprises approximately 800 lines of P4_16 code for data plane programming. Note that we use only CRC-8 hash functions in our FIB implementation, as detailed in Section III-H. In the performance evaluation shown in Fig. 20, we conduct tests with both UDP and TCP flows, achieving a throughput of 29.09 million packets per second for each. These results demonstrate that Parrot hashing can achieve a packet forwarding rate of 33.59 Gbits per second, indicating line-rate lookup throughput for FIB in the testbed.

V. RELATED WORK

Forwarding information bases (FIBs) FIBs are a critical component in modern networking [13], [20], [42], [43], [44], serving as the cornerstone for efficient and scalable packet forwarding in network devices, enabling routers and switches to quickly determine the egress interface for incoming packets based on their destination addresses. Network lookup algorithms are key design components for network functions in different layers and examples include classic IP and MAC table lookups and other key-value lookups such as load balancers (5-tuple to virtual IP) [5], [45], content-centric routing (content ID to forwarding port) [46], [47], [48], mobile host search (host ID to forwarding port) [14], [49], and in-network storage (content ID to destination) [9], [50]. This work focuses on a general case of in-network lookups: searching for a key and getting a returned value that matches the key.

Learned index schemes. Learned index schemes realize a fast indexing approach for a large-volume database with model-based mapping. In the existing LMH-based hash tables [21], [22], [23], collisions of records are resolved with its associated hash table schemes. Besides chained-based and probe-based LMH, shown in Table. III, cuckoo-based LMH [23] leverages the shadow bucket computation in Cuckoo hashing to increase the load factor of the whole table. However, it does not support range queries. Other than LMH-based hash tables, learned model based tree index schemes [35], [41], [51], [52], [53] have also been proposed, which include multiple levels of memory accesses. None of the existing LMH methods includes efforts to minimize the memory cost. Parrot is compatible with all existing learned index models [21], [22], [23], [54].

Learned indices in database and storage applications. Due to the high performance and lightweight model used in the learned index, there have been recent works [24], [53], [55], [56], [57] applying it to storage and database scenarios. To make the learned index fit non-volatile memory that can achieve a slightly lower speed compared to expensive DRAM, APEX [51] caches part of the metadata in DRAM (like bitmap for overflow array in PM) to reduce the number of PMs accessing. PLIN [58] trades local keys' order in leaves for less NVM block accessing meanwhile keeping global keys' order to support range query. Bourbon [55] proposes to apply the learned index to each layer in the LSM-tree KV store to reduce the latency on binary search, especially since immutable SSTables have no in-place updates to impact prediction accuracy negatively. Rolex [59] and XStore [60] propose to use the learned index as a cache to speed the one-sided RDMA-based KV store in disaggregated memory. FILM [57] adapts the layout of the learned index to manipulate its LRU policy to fit anti-caching architecture in a larger-than-memory database. Furthermore, XIndex [56] and FINEdex [53] designed partially locked schemes for updating data to achieve higher consistency, with a small part of data requiring to be locked in a multi-thread case.

VI. DISCUSSION

In this section, we highlight two potential application domains where our proposed Parrot hashing algorithm could have a significant impact: in-network caching and distributed learning workloads.

In-network caching of intermediate results. In many distributed learning pipelines, workers repeatedly generate intermediate results (e.g., shards of preprocessed data, partial aggregates, or key-value caches in large language models). Caching these results in the network can significantly reduce redundant computation. Parrot hashing can underpin a high-performance in-network cache. For instance, a programmable switch or NIC can maintain a Parrot-indexed table of frequently accessed intermediate key-value pairs. For each packet querying an intermediate result, the device computes the key's CRC-8, uses the learned model to identify the appropriate bucket, and retrieves the entry via the minimal perfect hash. Since Parrot hash table is compact and contains limited empty slots, it efficiently utilizes the limited on-switch SRAM.

Applicability to distributed model parameter synchronization. Modern distributed training frameworks, such as parameter server and all-reduce architectures, represent model parameters as key-value pairs (e.g., weight or embedding ID-value mappings) that must be shared efficiently across nodes. Parrot hashing can function as an ultra-fast in-network or NIC-based index for these parameter stores. By using a lightweight learned model to partition keys and a minimal perfect hash within each partition, Parrot enables on-chip one-byte CRC-8 hashing to drive fast lookups with minimal memory overhead.

VII. Conclusion

We conduct a comprehensive study on existing LMH and MPH lookup algorithms and analyze their feasibility to network lookup applications that require high throughput and small memory. Our observations show that LMH can achieve fast computation using linear models instead of hash functions while MPH can achieve low memory cost but rely on multiple independent and complex hash functions. We propose Parrot hashing, a novel lookup method that achieves both high throughput and small memory, by using only one CRC-8. We implement Parrot on three prototypes running on end systems, software switches, and a programmable hardware switch. The evaluation results show that Parrot achieves the highest packet processing throughput among existing solutions on these prototypes. We believe Parrot can be widely applied to various network lookup applications in different layers.

Acknowledgment

The authors would like to thank the anonymous reviewers for their comments.

References

[1] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with OpenSketch," in *Proc. USENIX Symp. Networked Syst. Design Implement.*, 2013, pp. 29–42.

[2] Z. Liu et al., "NitroSketch: Robust and general sketch-based monitoring in software switches," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2019, pp. 334–350.

[3] T. Yang et al., "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 561–575.

[4] P. Patel et al., "Ananta: Cloud scale load balancing," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, 2013.

[5] D. E. Eisenbud et al., "Maglev: A fast and reliable software network load balancer," in *Proc. USENIX NSDI*, Mar. 2016, pp. 523–535.

[6] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 15–28.

[7] S. Shi et al., "Concury: A fast and light-weight software cloud load balancer," in *Proc. ACM SoCC*, 2020, pp. 179–192.

[8] B. M. Maggs and R. K. Sitaraman, "Algorithmic nuggets in content delivery," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 3, pp. 52–66, Jul. 2015.

[9] Z. Liu et al., "DistCache: Provable load balancing for large-scale storage systems with distributed caching," in *Proc. USENIX FAST*, Feb. 2019, pp. 143–157.

[10] Z. Zhu, Y. Zhao, and Z. Liu, "In-memory key-value store live migration with netmigrate," in *Proc. 22nd USENIX Conf. File Storage Technol.*, 2024, pp. 209–224.

[11] Y. Liu, S. Shi, M. Xie, H. Litz, and C. Qian, "Smash: Flexible, fast, and resource-efficient placement and lookup of distributed storage," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 51, no. 1, pp. 1–22, Jun. 2023.

[12] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, high performance Ethernet forwarding with CuckooSwitch," in *Proc. 9th ACM Conf. Emerg. Netw. Exp. Technol.*, Dec. 2013, pp. 97–108.

[13] M. Yu, A. Fabrikant, and J. Rexford, "BUFFALO: Bloom filter forwarding architecture for large organizations," in *Proc. 5th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2009, pp. 313–324.

[14] D. Zhou et al., "Scaling up clustered network appliances with ScaleBricks," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 241–254.

[15] Y. Yu, D. Belazzougui, C. Qian, and Q. Zhang, "Memory-efficient and ultra-fast network lookup and forwarding using Othello hashing," *IEEE/ACM Trans. Netw.*, vol. 26, no. 3, pp. 1151–1164, Jun. 2018.

[16] S. Shi and C. Qian, "Ludo hashing: Compact, fast, and dynamic key-value lookups for practical network systems," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 2, pp. 1–32, 2020.

[17] Y. Xu et al., "Hashing design in modern networks: Challenges and mitigation techniques," in *Proc. USENIX ATC*, 2022, pp. 805–818.

[18] *Broadcom Bcm56070 Switch Programming Guide*. Accessed: Jul. 2025. [Online]. Available: https://docs.broadcom.com/doc/56070- PG2-PUB

[19] *Data Center Access Design With Cisco Nexus 5000 Series Switches and 2000 Series Fabric Extenders and Virtual Port Channels*. Accessed: Jul. 2025. [Online]. Available: https://itnetworkingpros.files.wordpress.com/2014/04/c07-572829-01.pdf

[20] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004.

[21] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, May 2018, pp. 489–504.

[22] P. Ferragina and G. Vinciguerra, "The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds," *Proc. VLDB Endowment*, vol. 13, no. 8, pp. 1162–1175, Apr. 2020.

[23] I. Sabek, K. Vaidya, D. Horn, A. Kipf, M. Mitzenmacher, and T. Kraska, "Can learned models replace hash functions?," *Proc. VLDB Endowment*, vol. 16, no. 3, pp. 532–545, Nov. 2022.

[24] A. Kipf et al., "SOSD: A benchmark for learned indexes," 2019, *arXiv:1911.13014*.

[25] *Intel DPDK: Data Plane Development Kit*. Accessed: Jul. 2025. [Online]. Available: https://www.dpdk.org

[26] R. Marcus, A. Kipf, and A. van Renen, "SOSD: A benchmark for learned indexes," 2019, *arXiv:1911.13014*.

[27] A. Kipf et al., "RadixSpline: A single-pass learned index," in *Proc. 3rd Int. Workshop Exploiting Artif. Intell. Techn. Data Manage.*, Jun. 2020, pp. 1–5.

[28] B. S. Majewski, "A family of perfect hashing methods," *Comput. J.*, vol. 39, no. 6, pp. 547–554, Jun. 1996.

[29] D. Belazzougui, F. C. Botelho, and M. Dietzfelbinger, "Hash, displace, and compress," in *Proc. Algorithms-ESA*, Jan. 2009, pp. 682–693.

[30] E. Esposito, T. M. Graf, and S. Vigna, "Recsplit: Minimal perfect hashing via recursive splitting," in *Proc. 22nd Workshop Algorithm Eng. Exp. (ALENEX)*, 2019.

[31] M. Genuzio, G. Ottaviano, and S. Vigna, "Fast scalable construction of (Minimal perfect Hash) functions," in *Proc. Int. Symp. Experim. Algorithms*, Jan. 2016, pp. 339–352.

[32] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier filter: An efficient data structure for static support lookup tables," in *Proc. Annu. ACM-SIAM Symp. Discrete Algorithms*, 2004, pp. 30–39.

[33] *YCSB Benchmark*. Accessed: Jul. 2025. [Online]. Available: https://github.com/brianfrankcooper/ycsb

[34] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. 12th ACM SIGMETRICS/PERFORMANCE Joint Int. Conf. Meas. Modeling Comput. Syst.*, Jun. 2012, pp. 53–64.

[35] J. Ding et al., "ALEX: An updatable adaptive learned index," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2020, pp. 969–984.

[36] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna, "Monotone minimal perfect hashing: Searching a sorted table with accesses," in *Proc. 20th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2009, pp. 785–794.

[37] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing," in *Proc. USENIX NSDI*, vol. 2013, Apr. 2013, pp. 371–384.

[38] (2020). *Broadcom Corporation. Bcm56070 Switch Programming Guide*. [Online]. Available: https://docs.broadcom.com/doc/56070-pg2-pub

[39] *CRC8-MAXIM*. Accessed: Jul. 2025. [Online]. Available: https://github.com/frankboesing/fastcrc

[40] *Implementation of Ludo Hashing in C++*. Accessed: Jul. 2025. [Online]. Available: https://github.com/QianLabUCSC/Ludo

[41] C. Wongkham, B. Lu, C. Liu, Z. Zhong, E. Lo, and T. Wang, "Are updatable learned indexes ready?," 2022, *arXiv:2207.02900*.

[42] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using Bloom filters," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2003, pp. 201–212.

[43] H. Lim, K. Lim, N. Lee, and K.-H. Park, "On adding Bloom filters to longest prefix matching algorithms," *IEEE Trans. Comput.*, vol. 63, no. 2, pp. 411–423, Feb. 2014.

[44] S. Ashok, A. Partap, and A. Tahir, "Fast and efficient lookups via data-driven FIB designs," in *Proc. ACM SIGCOMM Workshop Future Internet Routing Addressing*, Aug. 2022, pp. 66–71.

[45] R. Gandhi et al., "Duet: Cloud scale load balancing with hardware and software," in *Proc. ACM SIGCOMM Conf.*, 2014.

[46] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proc. 5th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2009, pp. 1–12.

[47] L. Zhang et al., "Named data networking (NDN) project," Tech. Rep. UCLA, NDN-0001, 2010.

[48] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly, "Symbiotic routing in future data centers," in *Proc. ACM SIGCOMM Conf.*, Aug. 2010, pp. 51–62.

[49] D. Raychaudhuri, K. Nagaraja, and A. Venkataramani, "MobilityFirst: A robust and trustworthy mobility-centric architecture for the future internet," *ACM SIGMOBILE Mobile Comput. Commun. Rev.*, vol. 16, no. 3, pp. 2–13, Dec. 2012.

[50] X. Jin et al., "NetCache: Balancing key-value stores with fast in-network caching," in *Proc. 26th Symp. Operating Syst. Princ.*, Oct. 2017, pp. 121–136.

[51] B. Lu, J. Ding, E. Lo, U. F. Minhas, and T. Wang, "APEX: A high-performance learned index on persistent memory," 2021, *arXiv:2105.00683*.

[52] J. Wu, Y. Zhang, S. Chen, J. Wang, Y. Chen, and C. Xing, "Updatable learned index with precise positions," 2021, *arXiv:2104.05520*.

[53] P. Li, Y. Hua, J. Jia, and P. Zuo, "FINEdex: A fine-grained learned index scheme for scalable and concurrent memory systems," *Proc. VLDB Endowment*, vol. 15, no. 2, pp. 321–334, Oct. 2021.

[54] S. Wu, Y. Cui, J. Yu, X. Sun, T.-W. Kuo, and C. Jason Xue, "NFL: Robust learned index via distribution transformation," 2022, *arXiv:2205.11807*.

[55] Y. Dai et al., "From WiscKey to bourbon: A learned index for log-structured merge trees," in *Proc. 14th USENIX Conf. Operating Syst. Design Implement.*, Jan. 2020, pp. 155–171.

[56] C. Tang et al., "XIndex: A scalable learned index for multicore data storage," in *Proc. 25th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, Feb. 2020, pp. 308–320.

[57] C. Ma, X. Yu, Y. Li, X. Meng, and A. Maoliniyazi, "FILM: A fully learned index for larger-than-memory databases," *Proc. VLDB Endowment*, vol. 16, no. 3, pp. 561–573, Nov. 2022.

[58] Z. Zhang et al., "PLIN: A persistent learned index for non-volatile memory with high performance and instant recovery," *Proc. VLDB Endowment*, vol. 16, no. 2, pp. 243–255, Oct. 2022.

[59] P. Li, Y. Hua, P. Zuo, Z. Chen, and J. Sheng, "Rolex: A scalable rdma-oriented learned key-value store for disaggregated memory systems," in *Proc. 21st USENIX Conf. File Storage Technol.*, 2023, pp. 99–114.

[60] X. Wei, R. Chen, H. Chen, and B. Zang, "XStore: Fast RDMA-based ordered key-value store using remote learned cache," *ACM Trans. Storage*, vol. 17, no. 3, pp. 1–32, Aug. 2021.

Yi Liu (Graduate Student Member, IEEE) received the B.E. degree in electronic information engineering from the University of Science and Technology of China in 2020. He is currently pursuing the Ph.D. degree with the University of California at Santa Cruz. His research work mainly focuses on computer networking and systems.

Shouqian Shi received the B.Sc. degree in physics and the B.E. degree in computer science and engineering from the University of Science and Technology of China in 2014 and the Ph.D. degree in computer engineering from the University of California at Santa Cruz, Santa Cruz, in 2021. He is currently with Google LLC. His research interests include computer networks, distributed systems, and other emerging computer systems, including quantum networks, cloud and edge computing, software-defined networks, network security and privacy, and network verification.

Ruilin Zhou (Graduate Student Member, IEEE) received the B.Sc. degree in applied physics from Beijing University of Posts and Telecommunications in 2019 and the M.Sc. degree in electrical engineering from Northwestern University in 2021. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, University of California at Santa Cruz, Santa Cruz. His research focuses mainly on quantum networks and quantum computing. He is currently focused on protocol design and simulation of distributed quantum computing.

Yuhang Gan (Graduate Student Member, IEEE) received the B.S. degree in computer science and technology from Nanjing University in 2021. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, University of California at Santa Cruz, Santa Cruz. His research interests include computer networks, quantum networks, and distributed quantum computing.

Chen Qian (Senior Member, IEEE) received the B.Sc. degree in computer science from Nanjing University in 2006, the M.Phil. degree in computer science from The Hong Kong University of Science and Technology in 2008, and the Ph.D. degree in computer science from The University of Texas at Austin in 2013. He is a Professor with the Department of Computer Science and Engineering, University of California at Santa Cruz, Santa Cruz. He has published more than 80 research articles in a number of top conferences and journals, including ACM SIGMETRICS, IEEE ICNP, IEEE ICDCS, IEEE INFOCOM, IEEE PerCom, ACM UBICOMP, ACM CCS, *IEEE/ACM Transactions on Networking*, and IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS. His research interests include computer networking, quantum networks, data-center networks and cloud computing, the Internet of Things, and software defined networks. He is a Senior Member of ACM. He received the NSF CAREER Award in 2018.