Ludo Hashing: Compact, Fast, and Dynamic Key-value Lookups for Practical Network Systems

SHOUQIAN SHI and CHEN QIAN, University of California, Santa Cruz, USA

Key-value lookup engines running in fast memory are crucial components of many networked and distributed systems such as packet forwarding, virtual network functions, content distribution networks, distributed storage, and cloud/edge computing. These lookup engines must be memory-efficient because fast memory is small and expensive. This work presents a new key-value lookup design, called Ludo Hashing, which costs the least space ($3.76 + 1.05l$ bits per key-value item for l-bit values) among known compact lookup solutions including the recently proposed partial-key Cuckoo and Bloomier perfect hashing. In addition to its space efficiency, Ludo Hashing works well with most practical systems by supporting fast lookups, fast updates, and concurrent writing/reading. We implement Ludo Hashing and evaluate it with both micro-benchmark and two network systems deployed in CloudLab. The results show that in practice Ludo Hashing saves 40% to 80%+ memory cost compared to existing dynamic solutions. It costs only a few GB memory for 1 billion key-value items and achieves high lookup throughput: over 65 million queries per second on a single node with multiple threads.

CCS Concepts: • **Theory of computation → Bloom filters and hashing**; • **Networks → Data path algorithms**.

Additional Key Words and Phrases: Compact data structures; Key-value lookups; Network algorithms; Forwarding algorithms

ACM Reference Format:
Shouqian Shi and Chen Qian. 2020. Ludo Hashing: Compact, Fast, and Dynamic Key-value Lookups for Practical Network Systems. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 2, Article 22 (June 2020), 32 pages. https://doi.org/10.1145/3392140

1 INTRODUCTION

Fast lookups of large-scale key-value items are fundamental functions and design blocks of numerous networked and distributed systems. These *in-memory key-value lookup engines* serve as the indices to store and find the locations, addresses, or directions of the destination devices or queried data. The representative applications of these lookup engines include:

(1) The forwarding information bases (FIBs) on network routers and switches run in SRAM. Many FIBs uses key-value lookup engines to forward packets by searching flat network addresses, as such MAC, in data center networks [27–29, 58], metropolitan networks [45], LTE [61], software defined networks (SDNs) [59, 62], and future Internet designs [47]. The values of the lookups are packet outgoing ports.

Authors' address: Shouqian Shi, sshi27@ucsc.edu; Chen Qian, cqian12@ucsc.edu, University of California, Santa Cruz, 1156 High Street, Santa Cruz, California, 95064, USA.

2476-1249/2020/6-ART22 $15.00
https://doi.org/10.1145/3392140

(2) In a content distribution network (CDN) or edge network, a number of proxy servers cache popular Internet contents [36, 48, 57]. A lookup table can be used to find the server that stores a particular content [22].
(3) In a distributed file system, an index is required to maintain metadata and the location of file storage [42, 54]. The lookup keys are usually file names or IDs, and the values are locations where the files are stored.
(4) Cloud load balancers are important components of a data center, which distribute packets to replicated backend servers [16, 38, 43]. Here the lookup engine stores the flow states and each key is a 5-tuple and each value is a server index. Network address translation (NAT) also stores flow states and performs lookups based on 4-tuple for every packet.
(5) In embedded IoT devices, lookup tables are required for sharing sensing data and public keys [32, 51].

The important requirement of these in-memory lookup engines is **space efficiency**. It is because they are hosted in high levels of the memory hierarchy or special network devices, where the memory is fast, small, expensive, and power-hungry. Another requirement is to support **dynamic updates** that allow the tables to work in practice, including key-value insertions, deletions, and changes.

Hash tables are the conventional solutions of fast in-memory key-value lookups. To resolve hash collisions, the item keys should be stored to tell which value belongs to which key. For example, the widely used version of Cuckoo Hashing [41] allows up to 8 key collisions [17, 19, 34, 61, 62]. Hence Cuckoo Hashing must store the keys or at least the digests of keys [35]. *Storing keys may cost more space than storing the values* in the above applications. For example, a typical file ID in a storage system has hundreds of bits and each value (disk address) is only tens of bits. For FIBs the network addresses (48 to >100 bits) are longer than the port values ($\leqslant 8$ bits). In a CDN the keys (URLs) could be thousands of bits.

Hence, recent efforts have been made to use *minimal perfect hash functions (MPHFs)* [10, 18, 26] for in-memory key-value lookups, which significantly reduce the space cost by avoiding storing keys. For a set of n key-value items where each item is a tuple (k_i, v_i) of key k_i and value v_i, a minimal perfect hash function H' maps the n keys to integers 0 to $n-1$ *without collision*. The lookup table can simply use the MPHF and an array of n values, where the i-th value corresponds to the key that is mapped to i by H'. The lookup table does not need to store keys. Unfortunately, none of the existing MPHFs support fast dynamic updates. When there is a single item insertion/deletion, the MPHF and whole array need reconstruction. Bloomier filters [13, 15] and SetSep [21, 61] are two alternative perfect hash tables that have been used for network applications [21, 49, 51, 56, 59–61]. However, Bloomier filters spend $>$ 2x space to store the values, and SetSep is also difficult to update.

This paper presents Ludo Hashing, a space-efficient lookup engine based perfect hashing, which supports $O(1)$ lookups and dynamic updates. To our knowledge, Ludo Hashing costs **the least space** compared with existing solutions of dynamic key-value lookups. We show the numerical comparison of these solutions in Table 1 and Fig. 1. Unless explicitly sourced, empirical values in Table 1 are based on experiments of $n = 64M$, and $l = 20$, as explained in § 6.

Ludo Hashing gains the space saving by removing the key storage while maintaining a low amplification factor (AF) on values. AF is the number of more bits taken per item when the length of values are incremented by 1 bit. The *core idea* of Ludo Hashing can be presented in two steps. **Step i)**: We first use a properly designed method to divide all key-value items to a number of small groups. Each group only contains at most four items. **Step ii)**: For each group we find a hash function H such that H maps the four keys to integers 0 to 3 *without collision*. For most modern random hash function algorithms, we may generate an independent hash function H_s by using a

Solution	Space cost (bits per item)	Lookup time per query	Update time per operation
MPHF +Array	$> 1.44 + l$ (∗)	$O(1)$, >67ns [18]	Not allowed
SetSep [21, 61]	$0.5 + 1.5l$	$O(1)$, 212ns	>120**ms**
Partial key Cuckoo [35]	$1.05(L' + l)$	$O(1)$, 163ns	>46ns [49]
Bloomier/Othello [13, 15, 59]	$2.33l$	$O(1)$, 187ns	173ns
Ludo Hashing (this work)	$3.76 + 1.05l$	$O(1)$, 303ns	163ns

Table 1. l: bit length of each value. L: bit length of each key. L': bit length of each key digest. (∗)The most compact version of MPHF [18] costs $1.56 + l$ bits per item, already at a prohibitively high construction time cost: 2ms per item. The SetSep papers [21, 61] include neither clear update function nor experimental results of updating. We designed an update function in our best effort.

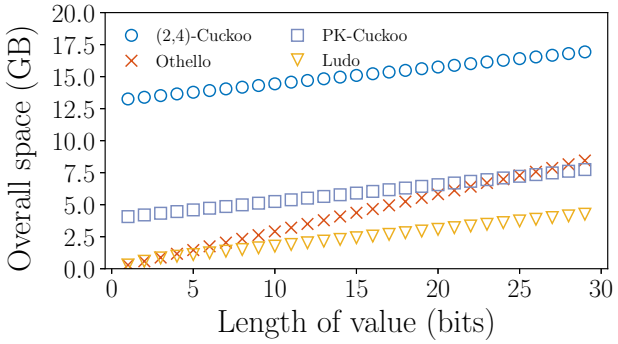


Fig. 1. Numerical space comparison of dynamic key-value lookups. $n = 1$ billion, $L = 100$ bits, $L' = 30$ bits. Ludo uses Bloomier for $l < 4$. n: number of items in table.

different seed s. Hence we find the right hash function for each group by trying different seeds with *brute-force*. Since each group contains only 4 keys, the seed can be found within a limited number of attempts — and costs only a few bits. Within each group, it is only necessary to store one seed s and four values that are in the order of the result of $H_s(k)$ for each key k. Both steps cost $O(1)$ time during lookups and each insertion/deletion/change can be updated in $O(1)$ amortized time. Eventually, we save the space of storing four keys — hundreds of bits or more — by using a seed that costs only 5 bits!

The main contribution of this work is a dynamic key-value lookup engine that works well in practical systems and achieves the least memory cost among existing methods to our knowledge. It is based on our discovery of a minimal perfect hashing method with $O(1)$ update cost. The compactness under dynamics is achieved via a novel combination of Bloomier filters, Cuckoo hashing, and brute-force based slot arrangement. We have implemented the complete software of Ludo Hashing with dynamic updates and single writer/multiple readers concurrency. We implement and evaluate Ludo Hashing in two working systems deployed in a real cloud environment. Experimental and analytical results are available for each design choice to inspire future methods and tools. The **source code** of Ludo Hashing is available for results reproducing [3].

The rest of this paper is organized as follows. Section 2 presents the related work. Section 3 defines the problem and system model. We present the detailed design of Ludo Hashing in Section 4 and the analysis results in Section 5. The system implementation and evaluation results are shown in Section 6. We have discussions in Section 7 and conclude this work in Section 8.

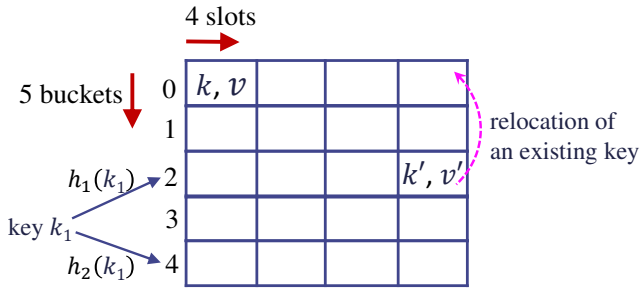


Fig. 2. (2,4)-Cuckoo Hash Table

2 RELATED WORK

In-memory key-value lookup engines with small memory footprint support vital functions of many networked and distributed systems, including network forwarding [58, 59, 61, 62], distributed storage [42, 54], cloud load balancers [38], and content distributions [22, 36]. Space efficiency is the most significant requirement of these applications because they are all running in fast and small memory, such as cache, DRAM, or ASICs, in order to serve frequent lookups.

Hash Tables are conventional tools for in-memory key-value lookups. Most existing hash table implementations require storing the complete keys. In particular, Cuckoo Hashing [41] is a key-value mapping data structure that could achieve $O(1)$ lookup time in the worst case and amortized $O(1)$ update time. As shown in Fig. 2: a (2,4)-Cuckoo has a number of buckets, each bucket has 4 slots, and every key-value pair is stored in one slot of the two *alternate buckets* based on the two hash values $h_0(k)$ and $h_1(k)$. The lookup of the value for a key k is to fetch the two buckets and match the keys in all 8 slots until a key matches k correctly. For an item insertion with key k_1, a single empty slot should be found in bucket $h_0(k_1)$ or $h_1(k_1)$. If both the buckets are full, one existing item (e.g., the one with key k' in Fig. 2) will be relocated to the other alternate bucket of k', and k_1 takes the slot of k'. If the alternate bucket of k' is full as well, an item in that bucket will be relocated recursively. This process stops when every item is placed in a slot. Many recent system designs choose the (2,4)-Cuckoo to achieve high memory utilization and fast lookups, such as the memory cache system MemC3 [19], the software switch CuckooSwitch [62], the LTE FIB ScaleBricks [61], and the cloud load balancer Silkroad [38]. The *amortized* insertion time of (2,4)-Cuckoo is proved to be constant [39, 55] and empirically shown [19, 41]. The insertions are proved to be successful asymptotically almost surely (a.a.s.) for load factor $< 98.03\%$ and $n \to \infty$ [12, 23].

Partial key Cuckoo hashing (PK Cuckoo) costs less space by storing the key digests instead of full keys. A basic version of PK Cuckoo is proposed in [20], and a more compact version, Vacuum filter, is proposed in [52]. SILT [35], an index for flash storage, proposes to use 15-bit key digests instead of the full keys. Using key digests is not a trivial solution. Short key digests incur hash collisions and false mappings, and a nontrivial two-level design is proposed in [49] to address the collisions.

EMOMA [44] is a lookup data structure with a full version of (2,4)-Cuckoo holding the key-value mappings. A counting block bloom filter (CBBF) is placed in cache to maintain the bucket choice of each key, such that each lookup costs exactly one off-chip memory load. There are three major differences between Ludo and EMOMA: 1) Ludo aims to reduce the memory cost while EMOMA requires significantly more memory cost – even higher than a full (2,4)-Cuckoo. The key reason is that Ludo resolves collisions within a bucket via a very short seed, instead of storing full keys in EMOMA. 2) EMOMA optimizes the lookup throughput while Ludo does not. 3) Ludo records the bucket choice of all keys without any error, while EMOMA uses a CBBF, which exhibits false

positives and counter overflows. 4) On a single insertion, keys in EMOMA may be inserted into and deleted from the CBBF multiple times, which hurt the update speed.

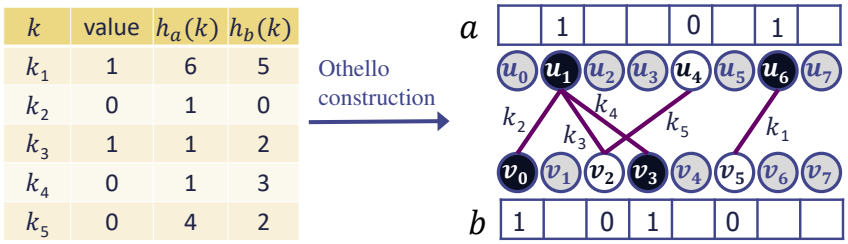


Fig. 3. Othello maintenance structure to build and update a and b

Bloomier filters [14, 15] are instances of minimal perfect hashing (MWHC) [9, 14, 15, 37, 59], originally proposed for static lookup tables. **Othello Hashing** is a data structure and a series of algorithms based on Bloomier filters designed for dynamic forwarding information bases [59]. Othello Hashing extends the Bloomier filter-based data plane by supporting runtime updates in programmable networks. Coloring Embedder [56] is a recent work with a similar design of Bloomier. Its space cost is also close to that of the Bloomier and Othello. Othello hashing includes both the lookup structure running in fast memory such as switch ASICs and a maintenance structure running in resource-rich platforms such as servers. For n key-value items and values with l bits, the *Othello lookup structure* only includes the two arrays a and b, each including m elements $m > n$. Each element in a and b is l bits. The lookup result of a key k is $\tau(k) = a[h_a(k)] \oplus b[h_b(k)]$, where h_a and h_b are two uniform hash functions. The Othello maintenance structure helps to compute the a and b to provide correct values as shown in Fig. 34. A good setup in practice is to allow a having $1.33n$ elements and b having n elements. The expected time cost of a construction of n keys is $O(n)$, and the expected time to add, delete, or change a key is O(1) [59]. More detailed explanation and examples can be found in Appendix A.

SetSep [21, 61] is a lookup table that uses brute force to resolve collisions. Suppose the key set has cardinality n, and all values are of the same length l. During SetSep construction, a global hash function distributes the keys across $\lceil n/4 \rceil$ buckets, each of which contains 4 keys *on expectation*, with high variations. 256 consecutive buckets form a block, and blocks are built independently. To build a block, a greedy algorithm is used to map its buckets to 64 groups, each holding 16 keys *on expectation*. For the i-th value bit in each group, a 16-bit array m and a 8-bit hash seed s are found by brute-force, such that for every key value pair (k, v) in the group, $m[h_s(k)] = v_i$, where $v_0, v_1, \cdots, v_{l-1}$ are bits of v. All key-value items of the failed groups are put into a small plain hash table. Ludo Hashing provides two major advantages over SetSep. First, Ludo Hashing can be updated in $O(1)$ complexity, while a single insertion into SetSep may cause reconstructions of the involved group, block, or even the whole data structure. The main challenge of its updates is that SetSep has no theoretical or empirical bound on the average number of group/block/global level reconstructions per update. Experimental results show that SetSep takes 10x construction time and >1000x update time compared to Ludo. Second, Ludo Hashing has small space cost when the value length is > 7, which is the case for most applications.

3 PROBLEM DEFINITION AND MODELS

We formally define the problem in this work. Given a set of key-value items S and $|S| = n$. Each item in S is a tuple $\langle k_i, v_i \rangle$ of key k_i and value v_i. Every key is unique in S. All values have the same size (i.e., number of binary digits), denoted by l. The goal of this work is to find a key-value lookup engine that provides the following functions, with minimized time and space costs.

(1) The lookup function query(k) returns the corresponding value v for the query key k, where $\langle k, v\rangle \in S$.
(2) The construction function construct(S) constructs a table for the set S.
(3) The insertion function insert(k, v) inserts the item $\langle k, v\rangle$ to the current table.
(4) The deletion function delete(k) deletes the item with key k from the current table.
(5) The value change function remap(k, v') changes the value of the item with key k to v', in the current table.

System model. The proposed Ludo Hashing includes the *lookup structure* and *maintenance structure*.

- The lookup structure in fast memory focuses on the lookup function. Its space cost and lookup time are minimized.
- The maintenance structure maintains the full key-value state and performs construction and update functions. It can run in a different thread or even on a different machine from where the lookup structure runs.
- Necessary update information will be constructed by the maintenance structure and sent to the lookup structure. The time complexity of each update is an important metric.

For space-efficient lookup engines that do not store full keys, a separate maintenance structure is *necessary* to support updates. Otherwise, update correctness cannot be guaranteed. In practice, the lookup structure is hosted in fast and small memory, while the maintenance structure can be hosted in slower but larger memory. This model has been extensively used in system designs [21, 22, 32, 35, 36, 38, 59–61].

4 DESIGN OF LUDO HASHING

4.1 Challenges and the main idea

A typical MPHF consists two-level hashing [10]. The first level hashing $g : U \to [0, r-1]$ divides the entire set K of n keys randomly into r buckets. The numbers of keys in all buckets vary significantly and the maximum number of keys in a bucket is much bigger than n/m based on the 'balls into bins' results [46]. The buckets are sorted in descending order of their size. In this order, the second level finds a hash function $f_i : U \to [0, m-1]$ for each bucket B_i such that the hash result of every key in B_i does not collide with any other key in all previous buckets. Let $\epsilon = m/n - 1$ and $\lambda = n/r$, the time complexity of the above construction is $O(n(2^\lambda + (1/\epsilon)^\lambda))$ [10]. In most cases, an insertion will cause the reconstructions of $O(r)$ second level hashes f_i.

Our main contribution is to allow each update to finish in $O(1)$ time by a novel utilization of (2,4)-Cuckoo and Othello, which has not been discovered before. Ludo first uses (2,4)-Cuckoo and Othello together to build a function F that divides the keys into r buckets, each of which has up to 4 keys, and then find a seed to resolve the collisions among each bucket. This design provides two unique benefits: 1) each insertion only affect $O(1)$ buckets (proved by [39, 55] and empirically < 6 among all our experiments), while in other MPHFs this number is unbounded; 2) within each bucket Ludo only needs to find a hash that maps four keys to $[0, 3]$ without collision, which is significantly easier than other MPHFs that need the results to be collision-free across all buckets.

Step 1: Uniform-sized grouping. By observing the (2,4)-Cuckoo Hash Table as shown in Fig. 2, we find that it includes a number of buckets, each containing up to 4 keys. This organization is close to our requirement of uniform-sized grouping. However, each key could be placed to any of its two alternate buckets based on the insertion process of (2,4)-Cuckoo. If we use a simple hash function to map keys to buckets, then the sizes of buckets suffer from a high variation. Resolving collisions of different numbers of keys will cause a significant waste of space, as shown in § 4.3.3. Hence, our idea is to combine a Bloomier filter [13, 59] and the bucket information of keys in the (2,4)-Cuckoo

as the uniform-sized grouping function F. In a (2,4)-Cuckoo, each key k can only stay in one of the two alternate buckets, indexed $h_0(k)$ and $h_1(k)$. Given a already constructed (2,4)-Cuckoo, we only need a Bloomier filter to maintain only 1 bit information per key: whether the key stays in the bucket $h_0(k)$ or $h_1(k)$. Recall that Othello Hashing is a dynamic extension to the original Bloomier filter, and Othello supports key-value lookups *with 100% correctness* using 2.33 bits per key for 1-bit values [59]. Hence, we need 2.33 bits per key to locate each key to the bucket holding it, in a constructed (2,4)-Cuckoo.

Step 2: Collision resolution. Given a bucket B of four keys, we want to find a function F'_B that maps the four keys to four different slots without collision. In this way, we can match all keys to their corresponding values without storing keys. Note that we may sample sufficiently many independent random hash functions from a universal hash function family $\mathcal{H}$. For example, Google's Farm Hash [2] accepts a 'seed' as input, and different seeds will result in independent hash functions. The probability that a randomly seeded hash function maps 4 keys to 4 slots without collision is $4!/4^4 = 3/32$. Therefore, by trying different hash functions with *brute force*, we can find a hash function that maps the 4 keys without collision in a limited number of attempts. Once a function is found for a bucket, the seed value is stored along the bucket. In our implementation, the seed costs 5 bits, i.e., 1.25 bits per key — a significant space saving comparing with storing the keys.

4.2 System overview

The complete Ludo Hashing includes two components, the Ludo lookup structure and Ludo maintenance structure. The Ludo lookup structure, considered as the data plane, runs in fast memory and supports lookup queries. The Ludo maintenance structure, considered as the control plane, can run in a slower memory, possibly on a separate machine. The lookup structure receives update information from the maintenance structure and updates accordingly.

Ludo lookup structure. As shown in Fig. 4, a Ludo lookup structure is a tuple $\langle O, B, h_0, h_1, \mathcal{H}\rangle$ where B is an array of buckets, each bucket $B[i]$ includes a hash seed s and 4 slots storing up to 4 values; h_0 and h_1 are two uniform hash functions; O is an Othello lookup structure that returns 1-bit value to indicate whether a key k is mapped to bucket $h_0(k)$ or $h_1(k)$; and $\mathcal{H}$ is a universal hash function family. The query of a key k will output the value v_k. Ludo lookup structure will query two locators in turn: the **bucket locator** to indicate the bucket that stores the value, and the **slot locator** to determine the slot that stores the value. The bucket locator will lookup k in Othello and get a result $b \in \{0, 1\}$. Then v is in bucket $h_b(k)$. The slot locator computes $t = \mathcal{H}_s(k)$ where s is the seed stored in this bucket and $t \in \{0, 1, 2, 3\}$. Finally, the value in slot t of bucket $h_b(k)$ is returned as v_k.

Ludo maintenance structure. As shown in Fig. 5, a Ludo maintenance structure is composed of two main parts: 1) a complete (2,4)-Cuckoo holding all inserted key-value items, and each bucket stores a seed for the slot locator; 2) an Othello maintenance structure that stores whether each key is in bucket $h_0(k)$ or $h_1(k)$. It can produce an Othello lookup structure used in the bucket locator. The seed s is found by brute force such that $\mathcal{H}_s$ maps the keys in the bucket to different slots without collision. We name the full (2,4)-Cuckoo as the *'source Cuckoo table'* of the lookup structure. To generate the Ludo lookup structure, the maintenance program first generates an Othello lookup structure and sets it as the bucket locator. Then it builds a table where each bucket includes the seed and only the four values in the order of the $\mathcal{H}_s(k)$. The Ludo maintenance structure supports updates including item insertions, deletions, and value changes (Sec. 4.5) and will reflect them in the lookup structure. Multiple Ludo lookup structures can be produced from and associated with the maintenance structure to receive update messages and update locally.

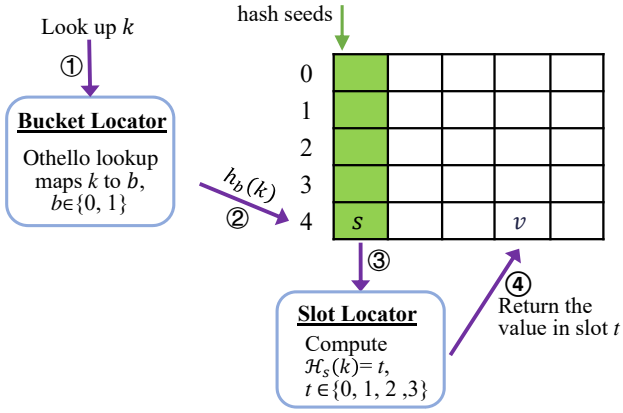


Fig. 4. Lookup workflow of Ludo lookup structure

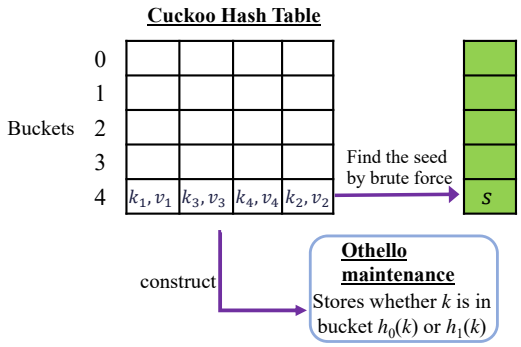


Fig. 5. Ludo maintenance structure

We define the *load factor* of a Ludo Hashing as the number of slots storing values to the number of total slots. We use load factor 95% as the target load factor of Ludo. The total space cost of Ludo Hashing is $3.76 + 1.05l$ for l-bit values.

4.3 Ludo lookup structure

We show the pseudocode of the Ludo Hashing lookup algorithm in Algorithm 1. This algorithm is simple and fast. It contains two steps: querying the bucket locator and the slot locator respectively. Each step takes $O(1)$ time.

4.3.1 The bucket locator. The bucket locator, implemented with an Othello lookup structure, maintains the bucket location of all inserted key-value items and serves in the *uniform-sized grouping* step. Given a query key k, the Ludo lookup structure locates k to a bucket by querying Othello. The return value b is 0 or 1, denoting the value of k is stored in the first alternate bucket $h_0(k)$ or the second one $h_1(k)$. The proposed bucket locator has the following properties.

1) It locates every inserted key-value item to the bucket holding it without error.

2) It costs amortized $O(1)$ time for dynamic updates, at a high throughput in practice (over 10 million operations per second [49]). During updates, it still supports fast lookup [59].

3) The current design is a good tradeoff among solutions that are fast in lookup and updates, and compact in mapping keys to $\{0, 1\}$, such as SetSep [21], Bloom filter cascades [32].

Input: The Ludo lookup structure and the key k
Output: The lookup result v of k
begin
    `// Step I: compute bucket location`
1   $b \leftarrow$ Othello lookup result of k
2   $B \leftarrow h_b(k)$-th bucket of the table
    `// Step II: compute slot location`
3   $s \leftarrow$ seed stored in B
    // $\mathcal{H}_s(k) \in \{0, 1, 2, 3\}$
4   $v \leftarrow B.slot[\mathcal{H}_s(k)]$
end

Algorithm 1: Ludo Hashing lookup algorithm

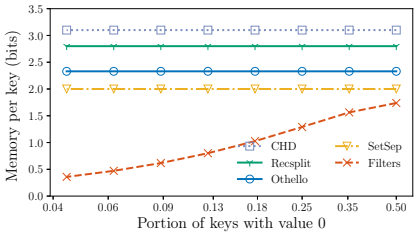


Fig. 6. Space cost of different algorithms as the bucket locator

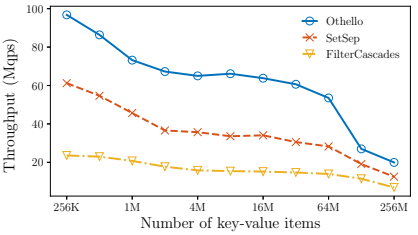


Fig. 7. Lookup throughput of different algorithms as the bucket locator

We compare their space cost in Fig. 6. Note that the filter cascades [32] cost different space when the distribution of keys to 0 and 1 changes. We collect the statistics of a (2,4)-Cuckoo with 100 million keys from 10 independent runs. The distribution of items stored in the bucket $(h_0(k), h_1(k))$ is $(0.7175, 0.2825)$ with the standard deviation 0.0008. By looking at 0.28 in Fig. 6, SetSep and filter cascades cost less space than Othello by about 0.3 bits per key. The reason for choosing Othello is that SetSep is difficult to update, as shown in § 6, and filter cascades are slow in lookup because each lookup costs higher number of memory loads on average. Fig. 7 shows the lookup throughput for different number of key-value items, where each key is a 32-bit integer and each value is 0 or 1 at the probability 0.7175 or 0.2825, respectively. Perfect hashing algorithms like CHD [10] and RecSplit [18] are also compared here, but they are not compact enough because an additional bit array is required to store the values, which costs 1 bit per item.

4.3.2 The slot locator. After locating the bucket, Ludo Hashing retrieves the bucket content that includes a seed s and 4 value slots. Ludo Hashing then calculates $\mathcal{H}_s(k)$ and gets a result in range $\{0, 1, 2, 3\}$. $\mathcal{H}$ is a universal hash family and each seed produces an independent random hash function. Finally, it returns the value that stored in the $\mathcal{H}_s(k)$-th slot.

It should be noted that the order of the values in each bucket of a Ludo lookup structure **does not necessarily follow the order in the source Cuckoo table** of the Ludo maintenance program. The order of the key-value items in a bucket of the source Cuckoo table is determined by the insertion and relocation processes. In Ludo lookup structure, however, we only need a **collision-free key-to-slot mapping** and the order of keys makes no difference.

The brute-force seed searching starts from $s = 0$. It increases s by 1 at each time until $\mathcal{H}_s(\cdot)$ maps the 4 keys of the bucket to $\{0, 1, 2, 3\}$ without collision (called a *valid seed*). This design is much less complex than finding the seed that produces the same order of the items in the source Cuckoo table. Our experimental studies show that it saves around 4.6 bits per key and use 4.2% time.

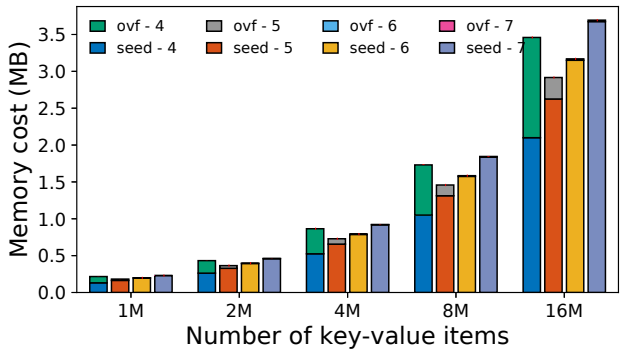
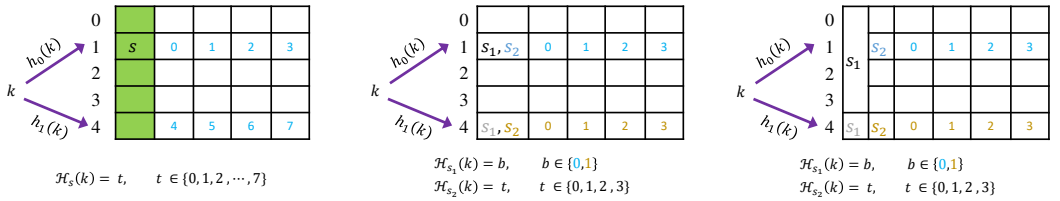


Fig. 8. Memory cost for different seed lengths (95% load)



(a) Look up a key in *Single*

(b) Look up a key in *Separate*

(c) Look up a key in *Grouped*

Fig. 9. Possible Ludo variants: *Single*, *Separate*, and *Grouped*

For $O(1)$ time lookups, each bucket should have the same size. Hence, the space to store the seed in every bucket should also be the same. For e-bit seed space, if the brute-force searching cannot find a valid seed by up to value $2^e - 2$, the seed space will store $2^e - 1$ (i.e., all 1 bits) to indicate that it is an *overflow seed*. Overflow seeds will be stored in a separate but much smaller table. We show the memory cost breakdown of seeds in buckets and the overflow table for different seed lengths in Fig. 8. Our implementation uses 5-bit seeds for minimal space cost.

The bucket and slot locators in total use 3.76 bits per key, including 2.33 bits for the bucket locator, 1.31 bits for the slot locator (assuming 95% load factor), and 0.12 bits for the overflow table. Each lookup takes 4 hash function calls and 3 memory loads — small constant time.

4.3.3 Design optimizations. The current design of Ludo lookup structure is chosen from a number of variants that achieves similar tasks, as shown in Fig. 9. We show the current design is more optimized than the others in the following.

Recall that each key can be mapped to two alternate buckets $h_0(k)$ and $h_1(k)$. For each bucket B, we define the '*T0 keys*' of B as the keys whose $h_0(k)$ buckets are B and the '*T1 keys*' as the keys whose $h_1(k)$ buckets are B.

Design option 1: Single locator ('*Single*'). We do not use the bucket locator. At each bucket, a hash seed is stored. For each key k, we always retrieve the seed s stored in the bucket $h_0(k)$. If the value of k is stored in the bucket $h_0(k)$, $\mathcal{H}_s(k)$ should be the correct slot position from 0 to 3. If the value is in the bucket $h_1(k)$, $\mathcal{H}_s(k)$ should be from 4 to 7 indicating one of the 4 slots in bucket $h_1(k)$. Hence, the seed s of bucket B is used for all T0 keys of B.

This method is simple to implement and requires fewer memory loads for each lookup: only one memory load with 71.75% possibility versus 3 for Ludo. However, the numbers of T0 keys of all buckets are not uniformly distributed and could possibly have high variation. In our experiments of 100 million items, some buckets may have > 20 T0 keys and thus the brute force process could

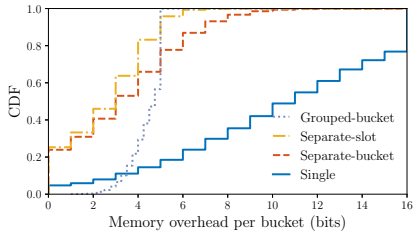


Fig. 10. Storage overhead per bucket for different design choices (1M keys, 95% load)

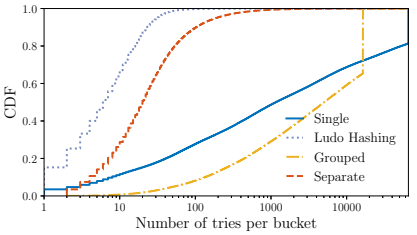


Fig. 11. Seed attempts per bucket for different design choices (1M keys, 95% load)

be very time-consuming and result in very long seeds. This introduces a **dilemma**: setting a short seed length leads to a large portion of seed overflow while setting a long seed length incurs big memory waste.

Design option 2: Separate seeds ('*Separate*'). This method stores two hash seeds s_1 and s_2 in each bucket: $\mathcal{H}_{s_1}(\cdot)$ computes an 1-bit value for all tier-1 keys, indicating whether the key is in bucket $h_0(k)$ or $h_1(k)$; and $\mathcal{H}_{s_2}(\cdot)$ maps all keys in this bucket to 4 slots without collision. Hence, s_1 works as the bucket locator and s_2 works as the slot locator. Compared to the Ludo Hashing design, it moves the time and space costs of Othello to the calculation of $\mathcal{H}_{s_1}(\cdot)$ and the storage of s_1. However, s_1 still needs to handle T0 keys with large variations.

Design option 3: Grouped buckets ('*Grouped*'). This method applies an additional optimization to save space for the seed s_1 in *Separate*. We combine the space of 4 consecutive buckets as a group and use a shared space for their s_1 seeds (4 is a number chosen for good cache locality and space saving). The shared space is used to store a long seed to filter all T1 keys in all 4 buckets. This method is designed to amortize the large variation of T0 keys in every bucket.

Design comparisons. We conduct the experiments of the 4 design choices *Single*, *Separate*, *Grouped*, and Ludo Hashing, and compare their results. We generate 1 million uniformly distributed 32-bit integers as keys and set the load factor of Ludo Hashing to 95%. To make the evaluations finish in a reasonable time, we set an upper bound 2^{16} for the number of seed attempts per bucket. We denote the seed length for the bucket locator of *Separate* as '*Separate-bucket*', the seed length for the slot locator of *Separate* as '*Separate-slot*', and the seed length per bucket for the bucket locator of *Grouped* as '*Grouped-bucket*'. Note that the seed lengths of the slot locators of Ludo Hashing and *Grouped* are both equal to *Separate-slot*.

Fig. 10 shows the cumulative distribution of the memory overhead (seed size) of each bucket and Fig. 11 shows the number of attempts to find the right seeds for each bucket. *Single* requires much longer seed sizes and higher computation overhead than other solutions. Note that *Grouped* fails to construct more than 30% groups in 2^{16} attempts as shown in Fig. 11. The sudden increase of the *Grouped* curve indicates the bound of this design. For *Separate* to work, the seed of the bucket locator requires 8 bits, allowing a small portion of overflow. This cost is thus about 2.11 bits per key, slightly less than using Othello. However, as shown in Fig. 11, *Separate* takes 3x time to compute the seeds compared to Ludo. Hence we believe the current design selects a good tradeoff.

Overflow seeds. As shown in Fig. 10, more than 98% slot locator seeds can be stored in 5 bits. Hence we set the seed length in each bucket to 5 bits. If a seed is larger than 30, it is marked overflow by storing the seed as 31. The map from the bucket index to the overflow seed is inserted into a small (2,4)-Cuckoo, called the *overflow table*, both in the maintenance structure and the lookup structure. According to the experiments in § 6, we show two facts: 1) We have never observed any seed that needs more than 8 bits. Hence, the value length is just 1 byte in the overflow table; 2) The

overflow rate is always around 1.2% and independent from the number of items in the table. The amortized cost of overflow seeds is around 0.12 bit per key.

Insertion fallback table. Recall we set the target load factor of the source Cuckoo table to 95%, which is a load factor in our experiments that never introduce a single insertion failure in breadth-first search (BFS) within 5 steps. For the strong robustness as a system, we set aside another small hash table to store the full key-value mapping for all items failed to be inserted, although in practice we have never seen failed insertions during the experiments for load factor $< 95\%$. The fallback table is similar to the stash approach used in Cuckoo [30, 31]. We store a fallback bit along with h_a and h_b. At the beginning of each lookup, if the fallback bit is 1, it means the fallback table stores some items. Hence the fallback table is first queried, and the corresponding value is returned if there is a match. If the fallback bit is 0, the query goes through the normal lookup procedure as shown in 4. In theory as long as the load factor $< 98.03\%$, the insertions are successful asymptotically almost surely (a.a.s.) assuming $n \to \infty$ [12, 23] as explain in Section 5 and Appendix C. This is the main reason why we never encounter a single insertion failure during our experiments. When the load factor reaches an application-dependent threshold (such as 94%) during system execution, the Ludo maintenance program will start to build a new Cuckoo table with higher capacity, which will be used to replace the original lookup table as soon as its load factor exceeds 95%. The implementation of this fallback table can be standard hash tables such as C++ unordered_map. The rebuild happens in the maintenance server, not on the query devices.

Why (2,4)-hash table? We conclude (2,4) is the best configuration for Ludo, based on the following reasons. 1) (2,4)-Cuckoo is almost optimal in load factor (maximum load $\approx$ 98% in theory [12, 23] and $\geqslant$ 96% in practice). 2) (2,4)-Cuckoo minimize the space costs of the bucket and slot locators. Recall the bucket locator costs $2.33\lceil \log_2 d \rceil$ bits per key, where d is the number of alternative buckets. Any increment in d will cost at least 2.33 bits per key, over 60% of the current overall overhead 3.72 bits per key. Besides, 5 or more slots in one bucket contributes little to the load factor [12, 23] but the expected number of slot locator tries grows from $\sim 4^4/4! \approx 10.7$ to $\sim 5^5/5! \approx 130$ or even higher.

4.4 Ludo Hashing construction algorithm

We design the Ludo maintenance structure to support fast construction and updates to the Ludo lookup structure. The construction takes $O(n)$ for n key-value items and each update takes amortized $O(1)$ time.

As shown in Fig. 5, the Ludo maintenance structure includes 1) a (2,4)-Cuckoo, which maintains all the inserted key-value items and decides their key-to-bucket mapping; 2) a seed in each bucket to determine the slot positions of the values; 3) an Othello maintenance structure to keep track of the current Othello lookup structure. As shown in Fig. 12, constructing a Ludo maintenance structure and Ludo lookup structure from scratch consists of the following steps.

Step 1. We start a standard (2,4)-Cuckoo construction. All key-value items are serially inserted into the Cuckoo table whose size is estimated by a load factor 0.95.

Step 2. For every bucket, a valid seed s is one that hashes keys to slots without collision. Numbers $0, 1, \cdots, 30$ are tested in order, to see if any is a valid seed. If all s from 0 to 30 are invalid, the algorithm stores 31 to indicate an overflow.

Step 3. For every key, get the 1-bit bucket placement information: 0 indicates the item is stored in bucket $h_0(k)$ and 1 indicates it is stored in bucket $h_1(k)$. The algorithm then constructs the Othello maintenance structure O to track this information for all keys.

Step 4. Construct the Othello lookup structure by simply copying the two data arrays from the Othello maintenance structure. Hence, the Othello lookup and maintenance structures give the same lookup result for every input key.

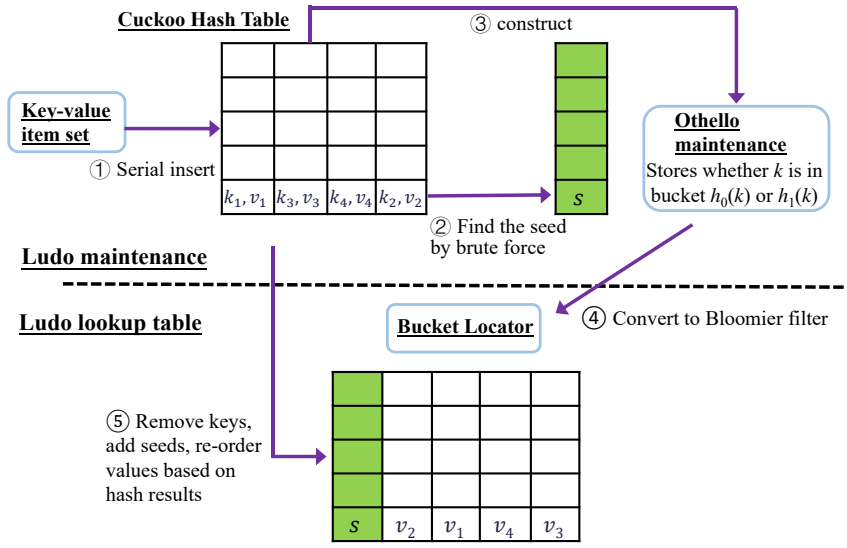


Fig. 12. Ludo construction algorithm

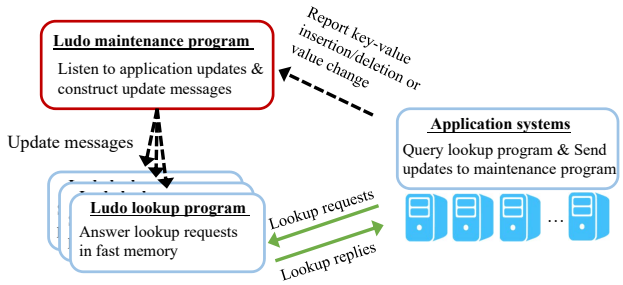


Fig. 13. Ludo Hashing system at runtime

Step 5. Construct a table with the same number of buckets as the source Cuckoo table. For each bucket in the source Cuckoo table (called the source bucket), copy the seed s to the bucket in the same position of the target table (called the target bucket). For each key-value item $\langle k, v \rangle$ in the source bucket, copy v into the $\mathcal{H}_s(k)$-th slot of the target bucket.

4.5 Ludo Hashing update algorithm

As a part of a practical system, Ludo Hashing at runtime consists two kinds of processes: the Ludo maintenance program holding a Ludo maintenance structure to maintain the full system state, possibly duplicated for robustness, and the Ludo maintenance program running as multiple instances (e.g., multiple lookup servers or routers), as shown in Fig. 13. The Ludo maintenance program and multiple Ludo lookup programs, receives update reports from applications, constructs update messages according to its current state, and sends them to all Ludo lookup programs. Each Ludo lookup program answers the lookup queries from applications and updates its memory according to the messages from the Ludo maintenance program. Similar to other key-value lookup tables, Ludo Hashing has three kinds of updates: key-value item insertions, item deletions, and value changes. We discuss the three update algorithms separately.

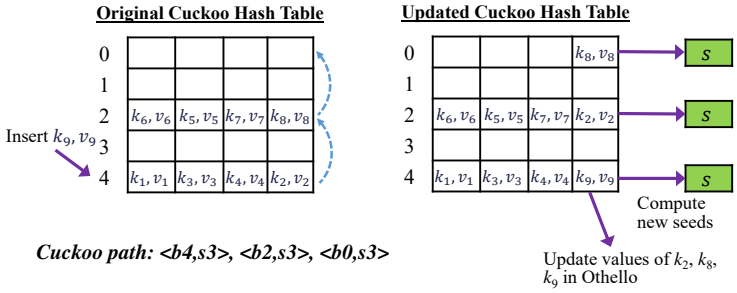


Fig. 14. Ludo maintenance program insertion process

Item insertion. The Ludo maintenance program takes three steps to construct the update message for an item insertion. 1) It first inserts this key-value item $\langle k, v\rangle$ into the source Cuckoo table, and records the *cuckoo path*. The cuckoo path of an item insertion is defined as the sequence of the positions of key relocations, where each position is determined by $(bucket\_index, slot\_index)$ [19]. In the example of Fig. 14, $\langle k_9, v_9\rangle$ is inserted to the table. $\langle k_2, v_2\rangle$ is relocated to from position $(b4, s3)$ to $(b2, s3)$ and $\langle k_8, v_8\rangle$ is relocated from $(b2, s3)$ to $(b0, s3)$. Hence the cuckoo path is $(b4, s3), (b2, s3), (b0, s3)$. 2) For each relocated key-value item, its position is switched between its alternate buckets $h_0(k)$ and $h_1(k)$. In Fig. 14, both k_2 and k_8 have switched between their alternate buckets. Ludo maintenance program updates the corresponding value in the Othello maintenance structure, and makes the changes in the Othello lookup structure. 3) For each modified bucket, Ludo maintenance program finds a new slot locator seed by brute force. The pseudocode is shown in Appendix B.

When the Ludo maintenance program finishes updating by the above steps. It creates an update message including three fields: *type* tells the update message type (insertion, deletion, or change), *val* is the value of the new item for insertion, and *update_sequence* is a sequence of nodes, representing the updates applied to the Ludo lookup structure. Each node in *update_sequence* corresponds to a position in the cuckoo path and includes the following: the bucket index *bIdx*, slot index *sIdx*, the new seed of this bucket *s*, the new order of values in the slots of this bucket *vodr*, and the changes made to the Othello lookup structure *Ochg*. The pseudocode of the update steps is shown in Appendix.B.

All associated Ludo lookup programs receive the same update message and follow the update sequence in that message to perform the insertion. Each Ludo lookup program traverses the nodes of the update sequence *reversely*, and takes three steps at each node: 1) Copy the bucket indicated in the node to a temporary memory. 2) Write the new seed into the bucket, reorder values according to *vodr*. 3) Atomically write the bucket back to the table and apply the change to the Othello lookup structure. The pseudocode of the update steps is shown in Appendix.B. The compiler barriers and version array are necessary for concurrent reads during updates.

Item deletion. In the Ludo maintenance program, deletions serve for space reclaim for future new items, and a deletion is achieved by deleting the item in the source Cuckoo table and the associated bucket location information in the Othello maintenance structure. There is no change to the Ludo lookup structure. If the number of items are lower than a threshold, e.g., the load factor $< 80\%$, a reconstruction can be triggered on the maintenance program to reduce the size of the lookup structure. During that process, the lookups are still on the existing lookup structure.

Value change. A value change only involves an update to a single slot and does not require any change in the bucket/slot locators. The Ludo maintenance program will perform a lookup in the source Cuckoo table to locate the bucket/slot position of the item, change the corresponding value, and send out a value change message to the lookup structure, specifying the new value and its

location in the target table. The Ludo lookup program will perform the value change according to the message.

Consistency under concurrent read/write. We design Ludo Hashing as a dynamic key-value lookup table under the single writer multiple reader model. To make the Othello lookup structure work well under concurrency, all modifications to the nodes belonging to the same key should appear atomic to the lookup threads. To allow concurrency in the lookup table, the value re-ordering should use the reverse order in the update sequence, and sequential writes of a single bucket should be atomic to the lookup threads. We extend the version-based optimistic locking scheme proposed in [19] and [59] for the target Cuckoo table and Othello lookup structure, respectively. Besides, we use the lock striping method proposed in [19] to reduce the size of the version array from the number of buckets to a constant 8192 at a 0.01% false retry rate. The pseudocode is shown in Appendix.B.

Ludo reconstruction. In very rare cases, such as table resizing, the Ludo maintenance program needs to reconstruct the Ludo lookup structure. During the reconstruction time, the data plane still queries the old lookup structure and use the fallback table to guarantee the correctness. When reconstruction finishes, the new lookup structure is sent from the maintenance program to the data plane. The update operations on the lookup structures are atomic. The new lookup structure is loaded from the update message and the old lookup structure is immediately discarded. Since then the queries will be based on the new lookup structure.

Parallel updates. The update algorithm on the maintenance program can be in some level of parallelism. If two updates do not touch the same bucket, then they can be computed in two threads without violating the correctness. The requirement is to have a shared array to store the locks of the buckets. If a bucket is currently in writing, the lock is set to 1 and other threads must wait to visit this bucket. We do not implement the parallel version of updates because the current update speed (>1M operations per second) is sufficiently high.

5 ANALYSIS

We summarize the performance analysis of Ludo Hashing: 1) The space cost of the Ludo Hashing is $3.76 + 1.05l$ bits per item; 2) Each lookup costs 3.02 memory loads on average; 3) Each insertion, deletion, or value change costs $O(1)$ time on average; 4) the communication cost for each update is $O(1)$ on average. The following presents the details.

5.1 Space cost of Ludo lookup structure

A Ludo lookup structure consists of three parts: the Bloomier filter for the bucket locator, the lookup table storing values and seeds, and a small table for the overflow seeds. The Bloomier filter costs 2.33 bits per key. The seeds cost 5 bits per bucket, i.e., 1.25 bits per key. The overflow table contains 1.2% of the seeds statistically, and each entry in the overflow table costs $29 + 8 = 37$ bits. Since the load factor of the Ludo lookup structure is 95%, it costs $1.05l$ bits per item, where l is the length of each value in bits. In total, the average memory cost per key-value item is: $2.33 + 5 \times 1.05/4 + 37 \times 1.05 \times 0.012/4 + 1.05l = 3.76 + 1.05l$ bits. The space cost of the fallback table is $O(n_f)$, where n_f is the number of fallback keys and $n_f \to 0$ based on the insertion correctness analysis below. Also our experiments never find a single fallback key. When the lookup structured is updated, the load factor may be set to an application-specific threshold (such as 94%) hence the space cost may increase to $3.78 + 1.06l$.

5.2 Lookup overhead

A key-value lookup in Ludo lookup structure always requires 3 memory loads: two for the Bloomier filter, and one to fetch the bucket including the value. If the seed overflows (with probability 1.2%),

another 1 or 2 random loads are required in the overflow table to get the seed. Hence we get the average number of memory loads $3 + 0.012 \times (1 \times 0.71 + 2 \times 0.29) = 3.016$.

5.3 Insertion correctness

From existing theoretical results of random graphs presented by Cain et al. [12] and independently Fernholz and Ramachandran [23], it has been proved that all n keys can be inserted to a (2,4)-Cuckoo table asymptotically almost surely (a.a.s.) such that each bucket has at most 4 keys if the load factor $< 98.03\%$, assuming uniform hashing and $n \to \infty$. This result has been confirmed by later studies [24, 25, 33, 55]. Detailed explanation can be found in the Appendix. In practice, our design sets the load factor threshold to 95% to avoid hitting the tight threshold. In fact we have not observed a single failure case among over 20 billions of insertions during our tests.

When the load factor $< 95\%$, the insertions are unlikely to fail from the above results. When the Ludo maintenance program detects the current load factor reaches 94%, it will start to build a new Cuckoo table with a higher capacity. The insertion failures (if any) will be stored in the fallback table. This design guarantees the correctness via these properties: 1) the runtime load factor will not be higher than 95% in most time; 2) even if the load factor temporarily exceeds 95% while the rebuild of Ludo with higher capacity has not finished, most insertions are still successful as the theoretical threshold is 98%; 3) even if there is an insertion failure, the fallback table is able to store it and guarantees the correctness of lookups.

5.4 Update overhead

Item insertion. The time complexity of each insertion to Ludo includes three parts: 1) the time to addition the item to Othello; 2) the number of nodes in the update sequence of each Cuckoo insertion; and 3) the time of updating the bucket of each node. We show the time of each insertion to Ludo is amortized $O(1)$ and independent of n based on the facts that all these three parts are either $O(1)$ or amortized $O(1)$. Inserting an item to Othello is proved to be amortized $O(1)$ [59]. From the theoretical results in [39], for a $(2,k)$-Cuckoo with load factor $1/(1+\epsilon)$ and $k \geq 16(\ln(1/\epsilon))$, each insertion costs amortized constant time $((1/\epsilon)^{O(\log\log(1/\epsilon))})$ by breadth-first search [39, 55]. Our design uses $k = 4$, which is less than $16(\ln(1/\epsilon))$. There is no proof of constant-time insertion for this setting. In our experiments, all insertions finish within 5 levels of breadth-first search. For each node in the update sequence, the update includes re-compute a seed (up to 31 attempts) and re-ordering the values (up to 4). It costs constant time for each node. We list the lengths of the update message fields. *type*: 1 bit; For each node in the update sequence, *bIdx*: 30 bits; *sIdx*: 2 bits; *seed*: 8 bits; *vorder*: 2 bits for each slot and 8 bits in total; *Bchg* contains the indices of the influenced nodes in the Bloomier filter, 32 bits for each index.

Each item deletion or value change costs $O(1)$ time and communication cost.

We discussed the average case above. In the worst case (very rare), an update may cause re-construction of Othello, but it only happens with probability $O(1/n)$ as proved in [59]. The Cuckoo table will not experience re-construction when the load factor is no more than 95% as shown above.

6 IMPLEMENTATION AND EVALUATIONS

6.1 Evaluation methodology

In this section we conduct two types of performance evaluation of Ludo Hashing: 1) Evaluation of the in-memory lookup tables on a commodity workstation with two Intel E5-2660 v3 10-core CPUs at 2.60GHz, with 160GB 2133MHz DDR4 memory and 25MB LLC; 2) Case study of Ludo Hashing on two real network systems, namely distributed content storage and packet forwarding.

We implement the Ludo maintenance structure and Ludo lookup structure prototypes in 3272 lines of C++ code. We also make use the open source implementation of Cuckoo Hashing (*pre-sized_cuckoo_map* in the Tensorflow repository [7]) and Othello Hashing (its authors' implementation [4]), with several major modifications to implement bucket/slot locator, update, and concurrent reading/writing. The buckets of Ludo lookup structure are stored as an array of 64-bit integers by carefully applying a series of bit-wise operations, such that there is no single bit waste on storing the buckets. The source code of Ludo Hashing is available for results reproducing [3].

We identify the following metrics to be evaluated:

(1) **Memory cost**, the most important metric to characterize the space efficiency.
(2) **Speed of update** to characterize the update time.
(3) **Lookup throughput** for single thread, multiple threads, and with concurrent reading/writing.
(4) **Construction time** of the lookup engine.

Each data point shown in the figures is the average of 10 independent experimental runs. We also use the error bars to show the standard deviation among the 10 results. For lookup throughput evaluations, the request workloads are in two types: in the uniform distribution and Zipfian distribution. For the uniform distribution, all items are requested with an equal probability. For the Zipfian distribution, items are requested with biased probabilities, which better simulates the workload in most practical systems. We set the Zipfian parameter to be 1.

We compare Ludo Hashing with the following dynamic lookup solutions: (2,4)-Cuckoo [19, 41], partial key Cuckoo [35, 49], Othello Hashing [59], and SetSep [21, 61]. We implement partial key Cuckoo based on the Tensorflow repository [7], with several major modifications to support fingerprint collision resolution. We implement SetSep and made several extensions to allow some level of updates of SetSep after construction – but still reconstructions are frequently needed. We use the Google FarmHash [2] as the hash function for all experiments.

6.2 Evaluation of in-memory lookup engines

We denote the number of key-value items as n, the sizes of each value, key, and digest as l, L, and L' respectively, all in bits.

Memory cost. Fig. 15 shows the memory cost breakdown of Ludo lookup structure, SetSep, Othello Hashing lookup structure, Cuckoo hashing, and partial key Cuckoo hashing, where n = 1B, $L = 100$, and $L' = 30$. We set l as 10 and 20. Clearly, Ludo Hashing needs the least memory cost among all designs for both $l = 10$ and 20. By comparing the breakdown parts of each design, we find that Ludo uses similar space to store the values, which seems unavoidable for every key-value lookup table. Note that Othello embeds the values in the two arrays A and B. Ludo saves much space cost by reducing the key storage while maintaining a low amplification on value storage. Despite being difficult to update, SetSep costs more space than Ludo Hashing, especially for large l.

From the analytical comparison in Fig. 1, Ludo always costs the least memory when $l > 3$. We then compare the actual memory cost of the in-memory lookup tables in three practical setups. 1) For the application of indexing distributed contents, we set $l = 20, L = 500, L' = 60$, assuming there are 1M content storage nodes. We set n to be 512M and 1B and show the results in Fig. 16. Ludo only requires 3.3GB for 1B items, while other designs need at least 6.3GB. Here Ludo saves almost 50% memory. 2) For the application of network FIBs, we set $l = 8, L = 48, L' = 30$, assuming a switch has 256 ports and MAC addresses are used. The results are shown in Fig. 17. It is known that a commodity switch has < 100MB SRAM [38], and Ludo only needs 50.5MB for 32M addresses. 3) For the application of indexing key-value storage, we set $l = 40, L = 200, L' = 60$ and show the results in Fig. 18. Ludo only uses 6.1GB memory to support 1B items, while other designs need > 12GB.

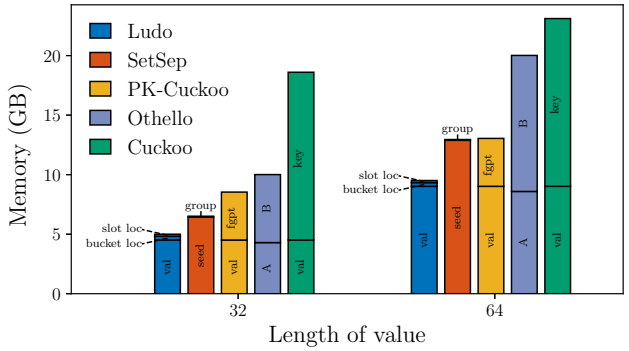


Fig. 15. Memory cost for different value lengths (1B keys)

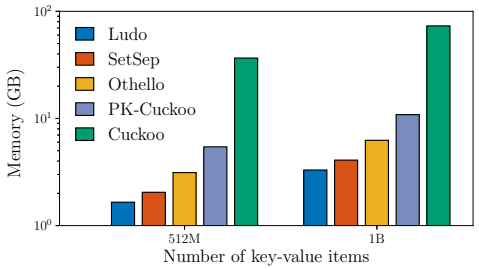


Fig. 16. Memory cost for indexing distributed contents

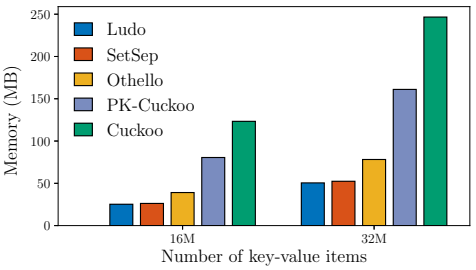


Fig. 17. Memory cost for FIBs

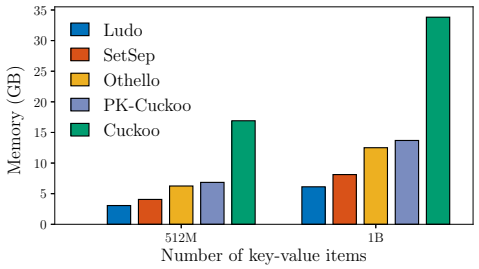


Fig. 18. Memory cost for indexing key-value storage

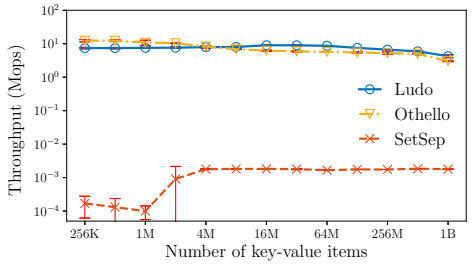


Fig. 19. Throughput (speed) of updates

Dynamic update. We evaluate the update throughput of Ludo Hashing, Othello Hashing, and SetSep, which characterizes the maximum number of updates a table can support, in the unit of millions of operations per second (Mops). All experiments are performed in a single thread, with equal numbers of insertions, deletions, and changes. We set $L = 32$ and $l = 20$, change the table size, and show the results in Fig. 19 where SetSep only performs the updates that do not cause reconstruction. Each update event may be an insertion, deletion, or value change, with the equal probability. The results show that Ludo allows > 5Mops updates, which is sufficient for most applications. Othello shows comparable performance with Ludo, while SetSep performs > 1000x worse than the other two even if we only consider the updates that do not cause reconstruction. As shown in the results below, each reconstruction of SetSep may take hundreds of seconds to >5 hours.

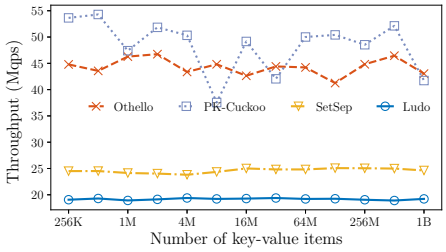


Fig. 20. Single-thread throughput for Zipfian

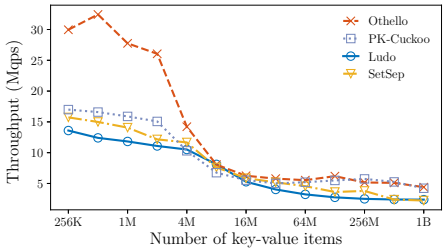


Fig. 21. Single-thread throughput for uniform

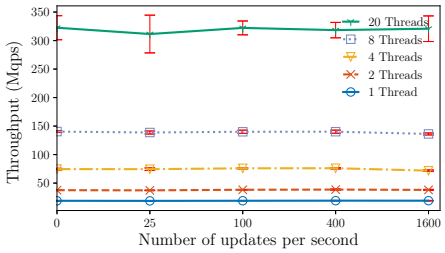


Fig. 22. Lookup throughput under updates

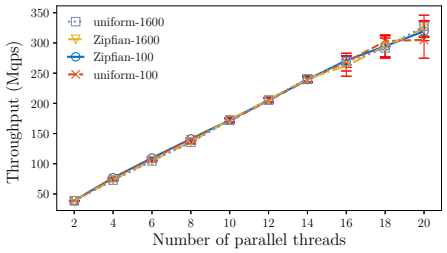


Fig. 23. Lookup throughput scales with # of threads

Single-thread lookup throughput. We compare the single-thread lookup throughput of Ludo, Othello, partial key Cuckoo, and SetSep, in Zipfian (Fig. 20) and uniform (Fig. 21) workload respectively. We set $l = 20$ and vary n, and the throughputs are in the unit of million queries per second (Mqps).

The throughput under uniform queries decreases with the growth of table size because the memory is randomly accessed and larger table incurs higher cache miss rate. The throughput under Zipfian distribution is less degraded by the table size because the L3 cache satisfies most queries. Othello/Bloomier shows the highest lookup throughput. Ludo Hashing is slower because for a single lookup, it requires 1 more hash function calculation and 1 more memory load. However, it still satisfies $>$ 5M queries per second when $n \leqslant$ 16M and $>$ 3M when $n =$ 1B. The throughput satisfies most applications and unlikely to become the system bottleneck.

Throughput under updates. We wonder whether concurrent writing/reading would affect the performance. Fig. 22 shows the lookup throughput of Ludo under concurrent updates (writing) by varying the update frequency, where $L = 64$, $l = 20$, and $n =$ 16M. Our observation is that there is no noticeable throughput degradation when the update frequency grows to up to 1.6K updates per second. Since 1.6K updates per second are sufficient for most dynamic applications, we may conclude that the lookup throughput is stable under concurrent writing.

Multi-thread throughput. We also show the results of multi-thread lookup throughput in Fig. 23, with up to 20 threads on a single machine and concurrent updates (100 and 1600 times per second). We find that the throughput scales linearly with the multi-thread. It achieves $>$ 300Mqps with 20 threads for $n =$ 1B.

Construction time. We also examine the construction time of the lookup engines. Fig. 24 shows the construction time of different designs by varying n, for $L = 64$, and $l = 20$. SetSep is >10x slower than other tables and takes >5.5 hours to construct for 1B keys. All other tables have similar construction time. For 1B items, Ludo Hashing can be constructed in 30 minutes.

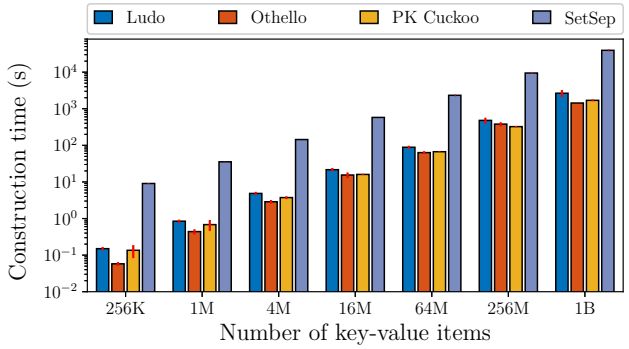


Fig. 24. Construction time

6.3 Case studies of real systems

We study the practical system performance with Ludo Hashing for two applications. **All experiments in this subsection perform real query packet receiving and forwarding.**

6.3.1 ***Case 1: indexing distributed contents.*** In this system, a large number of data contents are stored among the distributed storage nodes. There is an index node that accepts the queries of contents and forwards them to the correct storage nodes. The index node can be easily replicated to avoid the single point of failure. This model may be applied to many practical systems such as distributed data storage in a data center [8], CDNs [36], or edge computing [50]. In our experiments, the requested keys are uniformly sampled from the *std::string* representation of the content IDs (45 bytes).

Implementation details. We run the experiments in CloudLab [1], a research infrastructure to host experiments for real networks and systems. We implement Ludo Hashing, Bloom filter based lookup table (Summary Cache [22]), partial key Cuckoo hashing, and Othello Hashing to serve as the content lookup engine. We use two nodes in CloudLab to construct the evaluation platform of the forwarder prototypes. Each of the two nodes is equipped with one Dual-port Intel X520 10Gbps NIC, with 8 lanes of PCIe V3.0 connections between the CPU and the NIC. They are denoted by Node 1 and Node 2 in the following presentation. Each node has two Intel E5-2660 v3 10-core CPUs at 2.60GHz. The Ethernet connection between the two nodes is 2x10Gbps. The network between the two nodes provides full bandwidth. Logically, Node 1 works as the index node, and Node 2 works as all storage nodes in the system. The clients generate queries from the content IDs with Zipfian and uniform distributions.

Throughput of query processing and forwarding. We evaluate the query processing and forwarding throughput of Ludo Hashing, Bloom filters, partial key Cuckoo, and Othello in the distributed content storage system, in million queries per second (Mqps). We vary the number of contents from 16K to 16M. Figures 25 to 28 show the throughput versus number of items, in single and two threads, with Zipfian and uniform workload, respectively. Ludo Hashing provides the highest throughput as the index among the four methods. The reason is that the bucket locator of Ludo Hashing is compact enough to fit into the L3 cache so that it is likely to have only one load from the main memory for the table bucket access. Other solutions may have two main memory loads. Another interesting observation is that the capacity of querying processing and forwarding is bounded by 7 Mqps, which is smaller than the network bandwidth. The throughput does not grow significantly when we add more threads, which infers computation is not the bottleneck. Hence we consider the throughput is bounded by the bus bandwidth between CPU and memory.

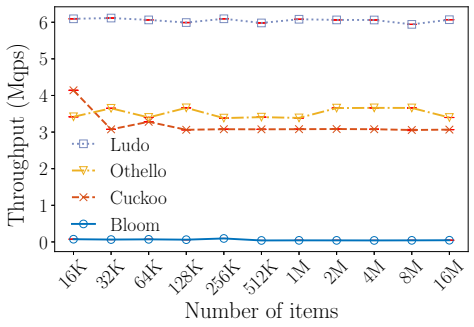


Fig. 25. Throughput of querying contents with Zipfian workload (single thread)

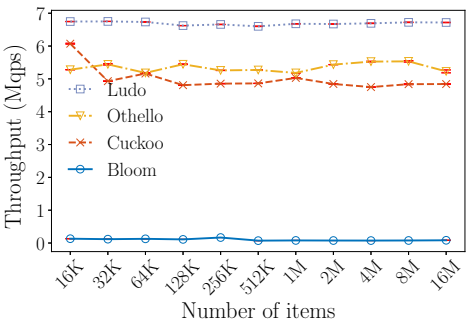


Fig. 26. Throughput of querying contents with Zipfian workload (two threads)

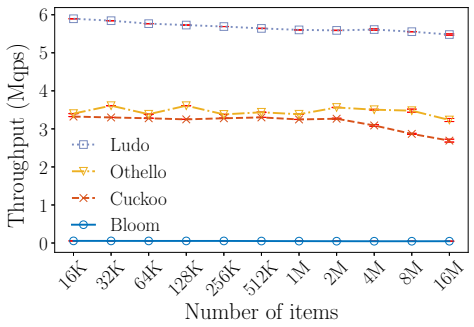


Fig. 27. Throughput of querying contents with uniform workload (single thread)

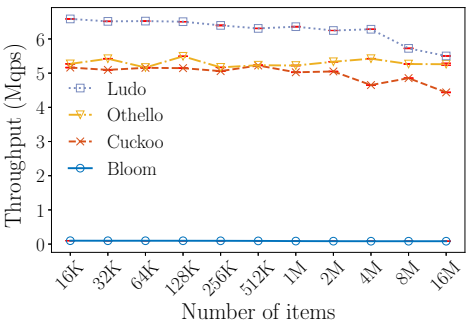


Fig. 28. Throughput of querying contents with uniform workload (two threads)

6.3.2 ***Case 2: forwarding information bases (FIBs).*** A modern data center network includes a large amount of physical servers [27, 29, 45]. Each server is identified by its network address (e.g., its MAC address). An interconnection of switches connects the servers. Each switch has multiple ports connecting neighboring switches and servers. A switch forwards the packet to a neighbor based on FIB lookups using the packet address. Many modern networks are variants of this model [27, 29, 45]. For software defined networks [40], the flow ID may be a combination of source/destination IPs, MACs, and other header fields. The forwarding may be per flow basis, rather than per destination basis. LTE backhaul networks and core networks can also be regarded as an instance of this network model, especially for the down streams from the Internet to mobile phones, where the destination addresses are Tunnel End Point Identifiers (TEIDs) of mobiles [61].

Implementation details. In the CloudLab prototype, we implement the FIBs as software switches [59, 62] that are running on the end hosts. We implement the FIBs using Ludo Hashing, Bloom filter based method (Buffalo [58]), partial key Cuckoo hashing [62], and Othello Hashing [59]. For each FIB implementation, we make several major modifications to support Dijkstra routing. The prototypes work with Intel Data Plane Development Kit (DPDK) [5] to support packet forwarding using end hosts. DPDK is a series of libraries for fast user-space packet processing [5] and is useful for bypassing the complex networking stack in Linux kernel, and it has utility functions for huge-page memory allocation and lockless FIFO, etc. We modify the code of the key-value lookup tables and link them with DPDK libraries. The query keys are in four types: 32-bit IPv4 addresses,

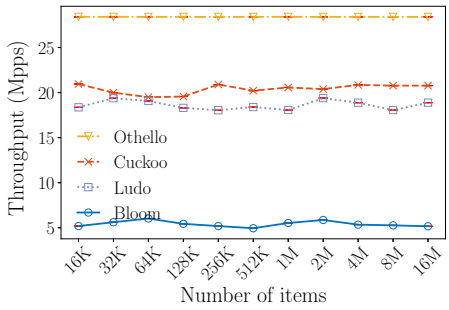


Fig. 29. FIB throughput with Zipfian workload (single thread)

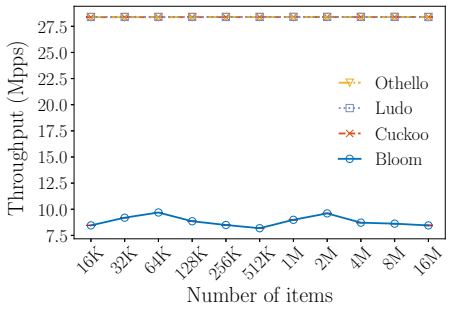


Fig. 30. FIB throughput with Zipfian workload (two threads)

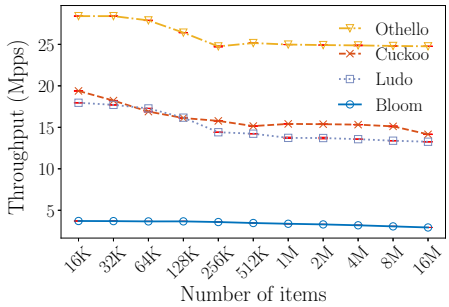


Fig. 31. FIB throughput with uniform workload (single thread)

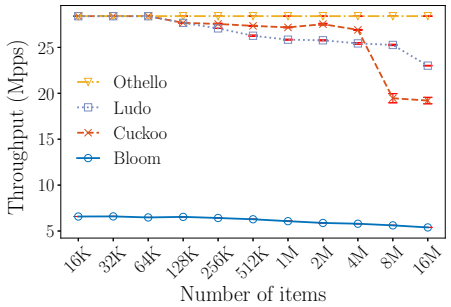


Fig. 32. FIB throughput with uniform workload (two threads)

48-bit MAC addresses, 128-bit IPv6 addresses, and 104-bit 5-tuples. We still use the two nodes in CloudLab (denoted by Nodes 1 and 2) for this prototype. The Ethernet connection between the two nodes is 2x10Gbps. The switches between the two nodes support OpenFlow [40] and provide full bandwidth. Logically, Node 1 works as a switch in the network, and Node 2 works as the neighboring switches and end hosts in the network.

Node 2 uses the DPDK official packet generator Pktgen-DPDK [6] to generate random packets and sends them to Node 1. The packets sent from Node 2 carry the destination addresses with Zipfian or uniform distributions. Each FIB prototype is deployed on Node 1 and forwards each packet back to Node 2 after determining the outbound link of the packet. By specifying a virtual link between the two servers, CloudLab configures the OpenFlow switches such that all packets from Node 1, with different destination addresses, will be received by Node 2. Node 2 then records the receiving bandwidth as the throughput of the whole system. The maximum network bandwidth is 28.40 million packets per second (Mpps).

Packet forwarding throughput. Figures 29 to 32 show the packet forwarding throughput of the four solutions, by vary the number of addresses stored in the FIB, with Zipfian and uniform distributions, for single thread and two threads, respectively. While Othello Hashing performs the best on a single thread, two threads of Ludo Hashing, partial key Cuckoo hashing, and Othello Hashing are sufficient to fill the full network bandwidth (called line rate) for a 16M FIB. For all cases, FIBs with Ludo Hashing, Othello Hashing, and partial key Cuckoo hashing performs >2x higher throughput than Bloom filters.

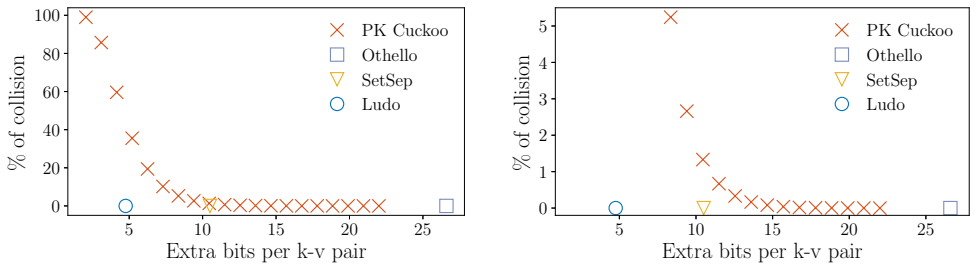


Fig. 33. Memory cost and collision rate

6.4 Summary of evaluation

Memory footprint. Ludo Hashing is the most compact among all dynamic in-memory lookup tables, under all configurations.

Lookup throughput. Ludo lookup structure achieves 5 to 20 Mqps single-thread throughput for up to 1B items. The throughput scales linearly with the number of threads and can achieves 65Mqps on one node.

Runtime update. Ludo lookup structure performs > 6M updates per second. The throughput of Ludo lookup structure is stable with concurrent updates.

Construction time. Ludo Hashing can be constructed for 1B items in 10 minutes.

Performance in real systems. Ludo Hashing provides higher throughput than other methods in the content lookup system. In the packet forwarding system, Ludo Hashing can easily achieve maximum network bandwidth with two threads.

7 DISCUSSION

Partial-key Cuckoo. One may consider setting short digests in partial-key Cuckoo [35] is a straightforward solution. However, short digests cannot be used because the key collision rate grows. Assuming values are l-bit long and $l = 20$, we change the key digest bit length L' from 1 to 20 for partial key Cuckoo, and observe the relation between the extra memory cost and key collision rate. The extra memory cost is defined as the overall memory cost of the lookup data structure minus nl, where n is the number of keys. We insert 1M random MAC addresses into different partial key Cuckoos, and the results are shown in Fig. 33.The right figure zooms in and shows the results near 1% of key collision. If we configure the PK Cuckoo to take no more than the memory of Ludo, > 40% keys will be mapped to more than one values. If we control the collision rate under 0.1%, the PK Cuckoo takes > 3x extra memory than Ludo.

Alien keys. Let K be the set of the keys of all items. An alien key (k_α) is defined as a key that was never inserted to the item set, i.e., $k_\alpha \notin K$. The lookup of an alien key may result in an arbitrary value by a perfect hash table, and we denote this as the 'alien key problem'. The alien key problem is not unique for Ludo. It exists for all perfect hashing based designs that do not store keys, including SetSep [21], Bloomier filters [11], and Othello [59]. This is a simple trade-off: either store the keys with several times higher memory cost, or accept the alien key problem and try to limit its impact. However, for any key $k \in K$, the lookup by Ludo Hashing will always be correct. Hence there is no false lookup result.

Most applications in the context of this work are not sensitive to alien keys, namely the distributed content index, network forwarding, and storage index. For a distributed content index, querying an alien key will make the index forward the request to an arbitrary storage node. The storage node will then find that no data in the node match this key. Hence it simply notifies the client a 'not exist' message. For a network forwarding device, a packet with an alien address will be

forwarded to an arbitrary port. Note that every packet will carry the time-to-live (TTL) field that will decrease by 1 after each forwarding action. Hence a packet will either be dropped when the TTL becomes 0 or dropped at a destination that does not match the address. Also, most networks will have firewalls that can filter all packets with alien addresses. In the above situations, an alien key has limited negative impact.

Alien keys will become a problem for applications that need to filter keys such as firewalls. Hence none of the perfect hashing method can be used for firewalls. For applications that really need to filter alien keys, a filter function can be added to the lookup table. Ludo Hashing can be perfectly combined with a Cuckoo filter [20, 53] that have a better trade-off between false positives and memory, compared to Bloom filters. Other methods such as Othello and SetSep will need either extra memory or lookup time to work with a filter. This topic is beyond the scope of this work, and we skip the details due to page limit.

8 CONCLUSION

Ludo Hashing is a practical solution for space-efficient, fast, and dynamic key-value lookup engines that can fit into fast memory. Its core idea is to use perfect hashing and resolve the hash collisions by finding the seeds of collision-free hash functions, instead of storing the keys. We present the detailed design of Ludo Hashing, including the lookup, construction, and update algorithms under concurrent reading and writing. The analytical and experimental results show that Ludo Hashing costs the least memory among known solutions that can be used for in-memory key-value lookups, while satisfying > 65 million queries per second for 1 billion key-value items on a single node. Ludo allows fast updates. We further demonstrate that Ludo Hashing achieves high performance in practice by implementing it in two working systems deployed in CloudLab.

9 ACKNOWLEDGEMENT

This work was partially supported by National Science Foundation Grants 1750704 and 1932447. We thank Bin Fan, Phokion Kolaitis, Heiner Litz, Ying Zhang, our shepherd Stefan Schmid, and the anonymous reviewers for their suggestions and comments.

REFERENCES

[1] CloudLab. https://www.cloudlab.us/.
[2] Implementation of farmhash. https://github.com/google/farmhash.
[3] Implementation of Ludo Hashing in C++. https://github.com/QianLabUCSC/Ludo.
[4] Implementation of Othello: a concise and fast data structure for classification. https://github.com/sdyy1990/Othello.
[5] Intel DPDK: Data Plane Development Kit. https://www.dpdk.org.
[6] Pktgen-DPDK. https://github.com/pktgen/Pktgen-DPDK.
[7] Implementation of presized cuckoo map. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/util/presized$_{c}$uckoo$_{m}$ap.h, 2016.
[8] Abu-Libdeh, H., Costa, P., Rowstron, A., O'Shea, G., and Donnelly, A. Symbiotic routing in future data centers. In *Proc. of ACM SIGCOMM* (2010).
[9] Belazzougui, D., Boldi, P., Pagh, R., and Vigna, S. Monotone minimal perfect hashing: searching a sorted table with $O(1)$ accesses. In *Proc. of ACM SODA* (2009).
[10] Belazzougui, D., and Botelho, F. C. Hash, displace, and compress. In *Proc. of Algorithms-ESA* (2009).
[11] Bloom, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM 13*, 7 (1970), 422–426.
[12] Cain, J. A., Sanders, P., and Wormald, N. The Random Graph Threshold for k-orientiability and a Fast Algorithm for Optimal Multiple-Choice Allocation. In *Proc. of ACM-SIAM SODA* (2007).
[13] Charles, D., and Chellapilla, K. Bloomier Filters: A Second Look. In *Proc. of European Symposium on Algorithms* (2008).
[14] Charles, D., and Chellapilla, K. Bloomier Filters: A Second Look. In *Proc. of ESA* (2008).

[15] Chazelle, B., Kilian, J., Rubinfeld, R., and Tal, A. The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables. In *Proc. of ACM SODA* (2004), pp. 30–39.

[16] Eisenbud, D. E., Yi, C., Contavalli, C., Smith, C., Kononov, R., Mann-Hielscher, E., Cilingiroglu, A., Cheyney, B., Shang, W., and Hosein, J. D. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proc. of USENIX NSDI* (2016).

[17] Erlingsson, U., Manasse, M., and McSherry, F. A cool and practical alternative to traditional hash tables. In *Proc. 7th Workshop on Distributed Data and Structures (WDAS'06)* (2006).

[18] Esposito, E., Graf, T. M., and Vigna, S. Recsplit: Minimal perfect hashing via recursive splitting. Tech. rep., 2019.

[19] Fan, B., Andersen, D., and Kaminsky, M. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proc. of USENIX NSDI* (2013).

[20] Fan, B., Andersen, D. G., Kaminsky, M., and Mitzenmacher, M. D. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies* (2014), ACM.

[21] Fan, B., Zhou, D., Lim, H., Kaminsky, M., and Andersen, D. G. When cycles are cheap, some tables can be huge. In *Proc. of USENIX HotOS* (2013).

[22] Fan, L., Cao, P., Almeida, J., and Broder, A. Z. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking* (2000).

[23] Fernholz, D., and Ramachandran, V. The k-orientability Thresholds for $G_{n,p}$. In *Proc. of ACM/SIAM SODA* (2007).

[24] Fountoulakis, N., Khosla, M., and Panagiotou, K. The multiple-orientability thresholds for random hypergraphs. In *Proc. of ACM/SIAM SODA* (2011).

[25] Gao, P., and Wormald, N. C. Load balancing and orientability thresholds for random hypergraphs. In *Proc. of ACM STOC* (2010).

[26] Genuzio, M., Ottaviano, G., and Vigna, S. Fast Scalable Construction of (Minimal Perfect Hash) Functions. In *Proceedings of the International Symposium on Experimental Algorithms* (2016).

[27] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. VL2: a scalable and flexible data center network. In *Proceedings of ACM SIGCOMM* (2009).

[28] Jain, S., Chen, Y., Jain, S., and Zhang, Z.-L. VIRO: A Scalable, Robust and Name-space Independent Virtual Id ROuting for Future Networks. In *Proc. of IEEE INFOCOM* (2011).

[29] Kim, C., Caesar, M., and Rexford, J. Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises. In *Proc. of Sigcomm* (2008).

[30] Kirsch, A., and Mitzenmacher, M. Using a queue to de-amortize cuckoo hashing in hardware. In *Proceedings of the Forty-Fifth Annual Allerton Conference on Communication, Control, and Computing* (2007), vol. 75.

[31] Kirsch, A., Mitzenmacher, M., and Wieder, U. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing* (2009).

[32] Larisch, J., Choffnes, D., Levin, D., Maggs, B. M., Mislove, A., and Wilson, C. CRLite: A Scalable System for Pushing All TLS Revocations to All Browsers. In *Proc. of IEEE S&P* (2017).

[33] Lelarge, M. A new approach to the orientation of random hypergraphs. . In *Proc. of ACM-SIAM SODA* (2012).

[34] Li, X., Andersen, D., Kaminsky, M., and Freedman, M. J. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proc. of ACM EuroSys* (2014).

[35] Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proc. of ACM SOSP* (2011).

[36] Maggs, B. M., and Sitaraman, R. K. Algorithmic Nuggets in Content Delivery. *ACM SIGCOMM Computer Communication Review* (2015).

[37] Majewski, B. S., Wormald, N. C., Havas, G., and Czech, Z. J. A Family of Perfect Hashing Methods. *The Computer Journal* (1996).

[38] Mao, R., Zeng, H., Kim, C., Lee, J., and Yu, M. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proc. of ACM SIGCOMM* (2017).

[39] Martn Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science* (2007).

[40] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.* (2008).

[41] Pagh, R., and Rodler, F. F. Cuckoo hashing. *Journal of Algorithms* (2004).

[42] Parekh, A. K., and Gallager, R. G. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking 1*, 3 (1993), 344–357.

[43] Patel, P., Bansal, D., Yuan, L., Murthy, A., Greenberg, A., Maltz, D. A., Kern, R., Kumar, H., Zikos, M., Wu, H., Kim, C., and Karri, N. Ananta: Cloud scale load balancing.

[44] Pontarelli, S., Reviriego, P., and Mitzenmacher, M. Emoma: Exact match in one memory access. *IEEE Transactions*

on Knowledge and Data Engineering (2017).
[45] Qian, C., and Lam, S. ROME: Routing On Metropolitan-scale Ethernet . In *Proceedings of IEEE ICNP* (2012).
[46] Raab, M., and Steger, A. Balls into Bins – A Simple and Tight Analysis. In *Lecture Notes in Computer Science* (1998).
[47] Raychaudhuri, D., Nagaraja, K., and Venkataramani, A. MobilityFirst: A Robust and Trustworthy MobilityCentric Architecture for the Future Internet. *Mobile Computer Communication Review* (2012).
[48] Schlinker, B., et al. Engineering Egress with Edge Fabric: Steering Oceans of Content to the World. In *Proc. of ACM SIGCOMM* (2017).
[49] Shi, S., Qian, C., and Wang, M. Re-designing Compact-structure based Forwarding for Programmable Networks. In *Proc. of IEEE ICNP* (2019).
[50] Shi, W., Cao, J., Zhang, Q., Li, Y., and Xu, L. Edge computing: Vision and challenges. *IEEE Internet of Things Journal 3*, 5 (2016).
[51] Wang, M., et al. Collaborative Validation of Public-Key Certificates for IoT by Distributed Caching. In *Proc. of IEEE INFOCOM* (2019).
[52] Wang, M., Zhou, M., Shi, S., and Qian, C. Vacuum filters: more space-efficient and faster replacement for bloom and cuckoo filters. *Proceedings of the VLDB Endowment* (2019).
[53] Wang, M., Zhou, M., Shi, S., and Qian, C. Vacuum Filters: More Space-Efficient and Faster Replacement for Bloom and Cuckoo Filters. In *Proceedings of VLDB* (2020).
[54] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. Ceph: A scalable, high-performance distributed file system. In *Proc. of USENIX OSDI* (2006).
[55] Wieder, U. *Hashing, Load Balancing and Multiple Choice.* Now Publishers, 2017.
[56] Yang, T., Yang, D., Jiang, J., Gao, S., Cui, B., Shi, L., and Li, X. Coloring Embedder: a Memory Efficient Data Structure for Answering Multi-set Query. In *Proc. of IEEE ICDE* (2019).
[57] Yap, K.-K., et al. Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering. In *Proc. of ACM SIGCOMM* (2017).
[58] Yu, M., Fabrikant, A., and Rexford, J. BUFFALO: Bloom filter forwarding architecture for large organizations. In *Proc. of ACM CoNEXT* (2009).
[59] Yu, Y., Belazzougui, D., Qian, C., and Zhang, Q. Memory-efficient and Ultra-fast Network Lookup and Forwarding using Othello Hashing. *IEEE/ACM Transactions on Networking* (2018).
[60] Yu, Y., Li, X., and Qian, C. SDLB: A Scalable and Dynamic Software Load Balancer for Fog and Mobile Edge Computing. In *Proc. of ACM SIGCOMM Workshop on Mobile Edge Computing (MECCOM)* (2017).
[61] Zhou, D., Fan, B., Lim, H., Andersen, D. G., Kaminsky, M., Mitzenmacher, M., Wang, R., and Singh, A. Scaling up clustered network appliances with scalebricks. In *SIGCOMM* (2015).
[62] Zhou, D., Fan, B., Lim, H., Kaminsky, M., and Andersen, D. G. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proc. of ACM CoNEXT* (2013).

APPENDIX

A. Bloomier filters and Othello hashing

We propose to use Othello Hashing [59] for the bucket locator of Ludo Hashing. Othello Hashing is a data structure and a series of algorithms based on Bloomier filters [14, 15]. Bloomier filters are instances of minimal perfect hashing (MWHC) [9, 14, 15, 37, 59], originally proposed for static lookup tables.[1] The recently proposed Othello Hashing [59] is an application of Bloomier filters for dynamic forwarding information bases. Othello Hashing includes the construction, update, and consistency maintenance of the Bloomier filter based data plane in programmable networks. Othello finds a setting of Bloomier filters to achieve good time/space trade-off for dynamic network environments.

An Othello Hashing is used as a mapping for a set of key-value pairs. Let S be the set of keys and $n = |S|$. A basic version of Othello Hashing supports the key-value pairs with 1-bit value. The lookup of each key returns an 1-bit value corresponding to the key. An advanced version of Othello supports l-bit values.

Othello maintenance structure construction. We use an example in Fig. 34 to show the construction process, which results in an Othello maintenance structure of a set of five key-value pairs. Each of the keys k_1 to k_5 has a corresponding value 0 or 1. We build two bitmaps a and b,

[1] Bloomier filters are completely different from the well-known Bloom filters.

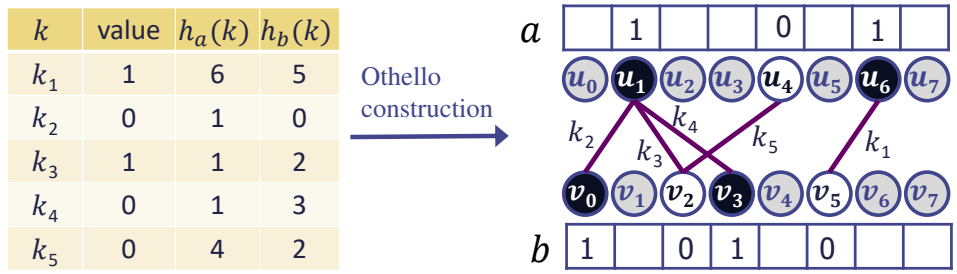
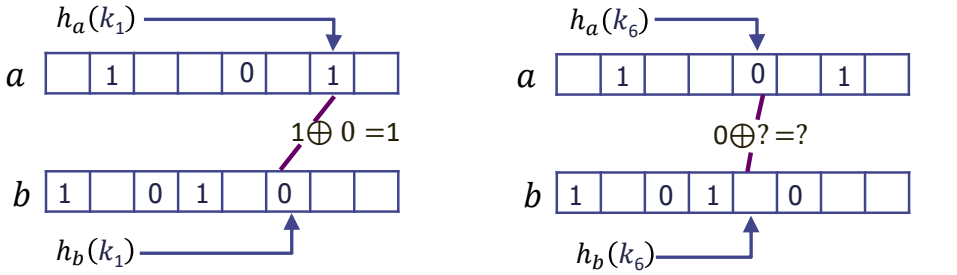


Fig. 34. Othello maintenance structure construction. An Othello only includes arrays A and B for lookups.



(a) Look up a key known during construction: Specified Result

(b) Look up a key unknown during construction: Deterministic Random

Fig. 35. Othello lookup structure

each with m bits and $m > n$. In this example $m = 8$. For every bit i in a we place a vertex u_i and for every bit j in b we place a vertex w_j. In this example $m = m_a = m_b = 8$. Two hash functions h_a and h_b are used to compute the integer hash values in $[0, m-1]$ for all keys. Then, for each key, an edge is placed between the two vertices that correspond to its hash values. For example, $h_a(k_1) = 6$ and $h_b(k_1) = 5$, so an edge is placed to connect u_6 and w_5.Each vertex is colored by black or white to represent the corresponding bit to be 1 or 0 respectively. For a key with value 0, the two vertices of the edge should have the same color. For a key with value 1, the two vertices of the edge should have different colors, so that the two bits have different values. k_1 is with value 1, hence u_6 and w_5 are with different colors. Gray color vertices represent "not care" bits. Note that after placing the edges for all keys, the bipartite graph G needs to be *acyclic*. If G is acyclic, a valid coloring plan is easily built by traversing each connected component of G, and setting bits based on corresponding values [59]. If a cycle is found, Othello needs to find another pair of hash functions to re-build G. It is proved that during the construction of n keys, the expected total number of re-hashing is < 1.51 when $n \leq 0.75m$ [59]. The expected time cost to construct G of n keys is $O(n)$, and the expected time to add, delete, or change a key is O(1).The design can be trivially extended to $l > 2$.

Key-value lookups in the Othello lookup structure. As shown in Fig. 35 (a), the *Othello lookup structure* only includes the two arrays A and B, and does not store the key-value array and the bipartite graph. To look up the value of k_1, we only need to compute h_a and h_b, which are mapped to position 6 of A and position 5 of B (starting from 0). Then we compute the bit-wise XOR of the two bits and get the value 01. Hence, the lookup result is $\tau(k) = a[h_a(k)] \oplus b[h_b(k)]$.

Lookups of Othello lookup structure are memory-efficient and fast. 1) The lookup structure only needs to maintain the two arrays. The keys themselves are not stored in the arrays. Hence, the space cost is small ($2m/n$ per key). 2) Each lookup costs just two memory access operations to read one element from each of A and B. It fits the programmable network architecture: the data plane only needs to store the lookup structure, two arrays; the control plane stores the key-value pairs

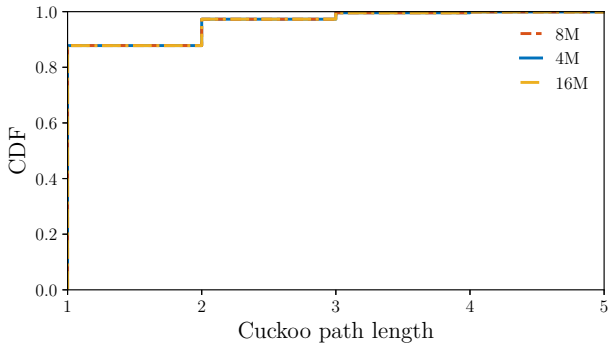


Fig. 36. CDF of Cuckoo path length

and the acyclic bipartite graph G. When there is any change, the control plane updates the two arrays and let the data plane to accept the new ones. When an Othello performs a lookup of a key that does not exist during construction, it returns an arbitrary value. For example in Fig. 35(b), $k_6 \notin S$ and its result may be an arbitrary value.

A lookup structure does not maintain the full states and must be updated by its associated maintenance structure. During an insertion or a value change, the maintenance structure keeps a list L of the influenced bits in arrays a and b. Then the list L is input to the update function in the lookup structure. The lookup structure simply flips the influenced bits to perform the updates. More details (e.g., read/write concurrency, l-bit updates) are found in [49, 59].

B. Pseudocode

We also show the pseudocode of the `insertion` algorithm on Ludo maintenance program algorithm in Algorithm 2, the `insertion` on the Ludo lookup program in Algorithm 3, the concurrent `lookup` algorithm of Ludo in Algorithm 4, and the `construct` algorithm for Ludo lookup structure from the Ludo maintenance structure in Algorithm 5. Algorithm 6 shows the subroutine in Ludo control plane to find a seed for a bucket.

C. Load factor for successful insertions.

From existing theoretical results of random graphs, it has been proved by both [23] and [12] that, if the average degree d of a random directed graph G of n vertices is no higher than a threshold d_k, then

$$\lim_{n\to\infty} \Pr(G \text{ is } k\text{-orientable}) = 1 \text{ if } d < d_k$$

We say G is asymptotically almost surely (a.a.s.) k-orientable. A graph is k-orientable if every vertex has in-degree at most k. Consider that each bucket of (2,4)-Cuckoo corresponds to a vertex of a random graph and each key corresponds to an edge. A key stored in a bucket can be considered an edge contributing to an in-degree to the vertex. Hence, a 4-orientable graph is equivalent to a (2,4)-Cuckoo where each bucket stores at most 4 keys. The above proved result [12, 23] is equivalent to the following statement. If the load factor is no higher than $d_4/8$ and the table is sufficiently large, all inserted keys can be stored in a (2,4)-Cuckoo such that every bucket has at most 4 keys. The numerical value of d_4 is 7.843, meaning the threshold of the load factor can be as much as 0.9803, provided by both [23] and [12]. Many later studies confirm this result [24, 25, 33, 55]. Note the extreme cases in practice that cause failed insertions do not conflict with this theoretical result. In practice, the length of a cuckoo path is within a small constant. We show the experimental results

of the lengths of cuckoo paths in Fig. 36, for Ludo Hashing with load factor < 95%, 4, 8, and 16 million items, and 10 runs for each setup. We find that all lengths of the Cuckoo paths are $\leqslant 5$ and more than 95% are smaller than 3. In our design, we set the load factor threshold to be 95% due to practical issues such as the maximum number of steps of evictions in implementation. We have not observed a single failure among over 20 billions of insertions during our tests.

Input: The Ludo maintenance structure $\langle O_M, C\rangle$ and the item to insert $\langle k, v\rangle$
Result: The insertion message $\langle val, update\_seq, failed\_key\rangle$ for Ludo lookup program
begin
1 $val \leftarrow v$
2 $update\_seq \leftarrow$ new empty list
`// I: Insert item and record cuckoo path`
3 $cuckoo\_path \leftarrow C.Insert(k, v)$
4 **if** $cuckoo\_path$ **is** empty **then**
5 Insert to fallback table
6 $failed\_key \leftarrow k$
7 **return**
8 **for** $position$ **in** $cuckoo\_path$ **do**
9 $bIdx, sIdx \leftarrow position$
10 $b \leftarrow C.buckets[bIdx]$
11 $k \leftarrow b.keys[sIdx]$
12 `// II: Reverse the bucket locator record, and record the influenced bits in Othello`
13 $Ochg \leftarrow O.Insert(k, 1 - O.LookUp(k))$
`// III: Find a new seed`
14 $b.s \leftarrow$ FindSeed (b)
15 $vorder \leftarrow$ Order of the values based on $b.s$
16 $update\_seq.add(\langle bIdx, sIdx, b.s, vorder, Ochg\rangle)$;
end
end

Algorithm 2: `insertion` algorithm on Ludo maintenance program

Received January 2020; revised February 2020; accepted March 2020

Input: Ludo lookup structure $\langle O_L, T\rangle$ and the insertion message $\langle val, update\_seq, failed\_key\rangle$, the version array V, a global lock L for fallback
Result: Ludo lookup structure is updated
begin
1 **if** $failed\_key$ **is set then**
2 $L.lock()$
3 Insert to fallback table
4 $L.unlock()$
5 **return**
6 **for** $i = update\_seq\_size - 1, \cdots, 3, 2, 1, 0$ **do**
7 $bIdx, sIdx, s, vorder, Ochg \leftarrow update\_seq[i]$
`// I: Copy current bucket`
8 $b \leftarrow$ copy of $T.buckets[bIdx]$
`// II: Update the temporary bucket`
9 $b.s = s$
10 Order values in b according to $vorder$
`// III: Consistency under concurrent R/W`
11 $V[bIdx \mod 8192] \leftarrow V[bIdx \mod 8192] + 1$
12 `compiler barrier`
13 Othello `atomic` update ($Ochg$)
14 $C.buckets[bIdx] \leftarrow b$
15 `compiler barrier`
16 $V[bIdx \mod 8192] \leftarrow V[bIdx \mod 8192] + 1$
end
end

Algorithm 3: `insertion` on the Ludo lookup program

Input: Ludo lookup structure, the version array V, and the key k to look up
Output: The query result v
begin
    `// Never entered in practice, under 95% load.`
1   **if** *Fallback table has entries* **then**
2       $L.lock()$
3       $v \leftarrow$ read from fallback table
4       $L.unlock()$
5   **return**
6   **while** `true` **do**
        `// Enssure bucket versions are even`
7       $v_0, v_1 \leftarrow V[h_0(k) \mod 8192], V[h_1(k) \mod 8192]$
8   `compiler barrier`
9   **if** v_0 or v_1 *is* odd **then continue**
        `// Atomically query bucket locator`
10      $l \leftarrow$ Othello `atomic` lookup (k)
        `// Fetch the bucket holding` k
11      $b \leftarrow h_l(k)$-th bucket of the table
        `// Enssure versions have not changed`
12   `compiler barrier`
13      $v_0', v_1' \leftarrow V[h_0(k) \mod 8192], V[h_1(k) \mod 8192]$
14   **if** $v_0 \neq v_0'$ or $v_1 \neq v_1'$ **then continue**
        `// Fetch the value of` k
15      $s \leftarrow$ *slot locator seed stored in* b
16      $v \leftarrow b.slots[\mathcal{H}_s(k) \mod 4]$
17   **break**
    end
end

Algorithm 4: Concurrent `lookup` algorithm on the Ludo lookup structure

Input: The Ludo maintenance structure $\langle O_M, C \rangle$
Output: The Ludo lookup structure $\langle O_L, T \rangle$
begin
    `// I: Othello maintenance to lookup`
1   $O_L \leftarrow O_M$ converts to a lookup structure
    `// II: New empty (2,4)-Cuckoo Hash Table`
2   $T \leftarrow$ empty table of size $C.size$
3   **for** $i = 1, 2, 3, \cdots, C.bucket\_size$ **do**
4       $b \leftarrow C.buckets[i]$
5       $b' \leftarrow T.buckets[i]$
        `// III: Copy locator seeds`
6       $s \leftarrow b.seed$
7       $b'.seed \leftarrow s$
        `// IV: Copy values to target buckets`
8   **for** $\langle k, v \rangle$ **in** valid items of b **do**
9           $sidx \leftarrow \mathcal{H}_s(k) \mod 4$
10          $b'.values[sidx] \leftarrow v$
        end
    end
end

Algorithm 5: `construct` algorithm for Ludo lookup structure from the Ludo maintenance structure

Input: The Ludo maintenance structure bucket b
Input: The new seed s
begin
1   **for** $s = 0, 1, 2, \cdots$ **do**
2       $taken \leftarrow$ 4-element boolean array
3       $success \leftarrow$ `true`
4   **for** k **in** valid keys of b **do**
5           $sid \leftarrow \mathcal{H}_s(k)$
6   **if** $taken[sid]$ **then**
7               $success \leftarrow$ `false`
8   **break**
9           $taken[sid] \leftarrow$ `true`
        end
10   **if** $success$ **then**
11   **return** s
        end
    end
end

Algorithm 6: Subroutine `FindSeed`