# Extendible RDMA-based Remote Memory KV Store with Dynamic Perfect Hashing Index

Zirui Liu*, Xian Niu†, Wei Zhou‡, Yisen Hong*, Zhouran Shi§, Tong Yang*,
Yuchao Zhang†, Yuhan Wu*, Yikai Zhao*, Zhuochen Fan¶*, Bin Cui*
*Peking University, China †Beijing University of Posts and Telecommunications, China
‡University of Southern California, USA §HKUST, Hong Kong SAR, China ¶Pengcheng Laboratory, China

***Abstract*—Perfect hashing is a special hashing function that maps each item to a unique location without collision, which enables the creation of a KV store with small and constant lookup time. Recent dynamic perfect hashing attains high load factor by increasing associativity, which impacts bandwidth and throughput. This paper proposes a novel dynamic perfect hashing index without sacrificing associativity, and uses it to devise an RDMA-based remote memory KV store called CuckooDuo. CuckooDuo simultaneously achieves high load factor, fast speed, minimal bandwidth, and efficient expansion without item movement. We theoretically analyze the properties of CuckooDuo, and implement it in an RDMA-network based testbed. The results show CuckooDuo achieves 1.9∼17.6× smaller insertion latency and 9.0∼18.5× smaller insertion bandwidth than prior works.**

***Index Terms*—Perfect Hashing; KV Store; RDMA**

## I. Introduction

Perfect hashing is a special hashing index function that maps each item to a unique location without collision [1], [2], which enables the creation of a KV store with small and constant lookup time [3], [4]. Typical hashing based key-value (KV) stores consist of two parts: 1) A small hashing index storing the mapping relationship between keys and slots, which is typically kept in fast memory mediums like SRAM or caches in local devices. 2) A large KV table with many slots storing KV pairs, which is typically kept in slow memory mediums like DDR or remote memory server.

Standard Dynamic Perfect Hashing (DPH) [1] supports dynamic update at the cost of a low load factor (<15%). Recent DPH variants [3], [5], [6] improve the load factor to >90% by organizing the KV-slots into buckets. They map each item to one bucket (with $d$ slots) and ensure each bucket is mapped by no more than $d$ items. These solutions improve the associativity (defined in § II-A) of standard perfect hashing from 1 to $d$, and thus impact lookup bandwidth and throughput. We aim at devising a dynamic perfect hashing without sacrificing associativity, and using it to improve the performance of remote memory KV store.

This paper proposes CuckooDuo, an extensive RDMA-based remote memory KV store with a dynamic perfect hashing index. CuckooDuo has the following advantages: 1) High load factor (>97%); 2) Fast speed (1.9∼17.6× smaller insert latency than prior works); 3) Small bandwidth (9.0∼18.5× smaller insert bandwidth than prior works); 4) Efficient expansion without item movement (2× faster than prior solutions). The key design of CuckooDuo is the synergistic use of two interlinked cuckoo hash tables [7]. In slow memory (remote), we deploy a cuckoo hash table, called **CuckooVault**, to store KV pairs. In fast memory (local), we deploy another cuckoo hash table of identical size, called **CuckooIndex**, to store the fingerprints (hash values) of the inserted keys. Each fingerprint in CuckooIndex corresponds one-to-one with a KV pair in CuckooVault. To lookup a key, we check its candidate buckets in CuckooIndex. If its fingerprint is found, we further retrieve the KV pair from the corresponding slot in CuckooVault. CuckooDuo addresses the following key challenges.

• *Challenge 1: How to build an index with small and arbitrary sizes?* It is challenging to design an index that is small enough to fit into various fast memory. We propose a *Dual-Fingerprint* design to significantly reduce fingerprint collision, thereby reducing fingerprint length, improving load factor, and saving space. The key idea is to change another fingerprint hashing function for the items with fingerprint collisions, and thus grant the colliding item a second chance to be re-inserted. Additionally, the size of a cuckoo hash table storing fingerprints (CuckooFilter [8]) can only be a power of two, resulting in space inflation. We also propose techniques to break this size limitation.

• *Challenge 2: How to minimize insertion latency?* The kick-out process of traditional cuckoo hash requires multiple sequential reads/writes to remote memory. With our one-to-one design, we can directly find empty slot or kick-out path by only checking local index. In this way, each insertion can be completed in one round-trip-time (RTT) by directly writing the incoming item to an empty slot, or worst-case two RTTs by reading all items on the kick-out path and writing them along with the incoming item into their new locations.

• *Challenge 3: How to scale up table size?* Unlike existing works expanding tables by moving items [9], [6], [3], CuckooDuo expands its size by copying local index and remote KV table. Afterwards, with the one-to-one relationship between CuckooIndex and CuckooVault, we can delete redundant items by only modifying local CuckooIndex, and

Corresponding authors: Zhuochen Fan (fanzc@pku.edu.cn) and Tong Yang (yangtong@pku.edu.cn).

Zirui Liu, Yisen Hong, Tong Yang, Yuhan Wu, Yikai Zhao, and Bin Cui are with School of Computer Science, Peking University, Beijing, China

Xian Niu and Yuchao Zhang are with School of Computer Science (National Pilot Software Engineering School), BUPT, Beijing, China

thus reduce one RTT to write remote memory, achieving 2× smaller expansion time than RACE [9] and MapEmbed [3]. CuckooDuo also offers a 6.67× faster lazy-mode expansion, and is compatible with extendible expansion [10].

In summary, this paper makes the following contributions.

- We propose CuckooDuo, a dynamic perfect hashing based KV-store with high load factor, fast speed, minimal communication overhead, and efficient expansion.
- We theoretically analyze the properties of CuckooDuo, and validate the theoretical results with experiments.
- We conduct experiments showing CuckooDuo outperforms state-of-the-art (SOTA) remote memory KV-stores on both time- and space- efficiency. All codes are open-sourced [11].

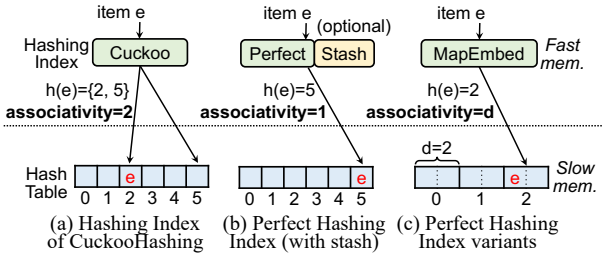



Fig. 1: Illustration of different hashing indexes.

## II. Background and Related Work

### A. Hashing Index and Perfect Hashing Index

Consider a set of items $S$. A hashing index $f$ is a data structure (or function) that maps items in $S$ to specific location(s) in a table (called *hash table*). For an item $e \in S$, $f(e)$ can be a set of $w$ candidate locations, meaning that $e$ can be stored in any one of these $w$ locations. We define $w$ as the **associativity** of a hashing index. For example, as shown in Figure 1(a), the hashing index of standard CuckooHashing [7] (described in § II-C) returns $w = 2$ locations for each item, and thus has an associativity of two. For a hashing index with an associativity of $w$, an item can be stored in any of the $w$ candidate locations, and thus lookup operation needs to access $w$ items in the hash table. The term "associativity" here has the same meaning as in set-associative cache in computer architecture. In a $w$-way set-associative cache, associativity $w$ refers to the number of cache lines within each set, meaning that an item can be stored in any of the $w$ locations (cache lines) within the cache.

A perfect hashing index is a specialized form of hashing index (or function) that maps each item in $S$ to a unique location without collision with other items, which is illustrated in Figure 1(b). A standard perfect hashing index has the associativity of exact one. With perfect hashing index, the lookup operation just need to access one item in hash table. This property ensures constant lookup time and minimal bandwidth overhead when the hash table is deployed on remote memory. Note that perfect hashing index cannot identify non-existent items. For $e' \notin S$, perfect hashing index returns an arbitrary location in the hash table. Therefore, negative lookups still need to access one item in the hash table.

There are two kinds of perfect hashing indexes. *1) Static Perfect Hashing* [12], [2], [13] that constructs perfect hashing index for static set. *2) Dynamic Perfect Hashing (DPH) [1]* that supports dynamic update. Standard DPH first hashes keys into multiple subtables and then finds a perfect hashing function for each subtable. When hash collision happens in a subtable, DPH reconstructs this subtable by finding another perfect hashing function. To ensure small reconstruction time, DPH requires the total load factor of the hash table to be low (<15%). **Load factor** of a hash table is defined as the number of inserted items divided by the total number of slots. Note that the load factor here is a property of the hash table storing items, rather than the property of the hashing index computing item locations. It has been proven that the minimal space usage of a perfect hashing index is $log_2(e) \times \alpha \approx 1.44\alpha$ bits per item [14], where $\alpha$ is the load factor of the hash table.

To improve load factor, recent DPH variants [3], [5], [15], [16] organize the slots in hash table into buckets, where each bucket has $d$ slots. As shown in Figure 1(c), given an item $e$, their indexes return the location of a bucket, and $e$ can be stored in any one of the $d$ slots in the bucket. Therefore, the associativity of these schemes is $d$, and their lookup operations need to access the $d$ items in a bucket. Fortunately, the $d$ items are located in a contiguous memory block, accessing them can be done with a single memory access, thus still achieving constant lookup time. However, their bandwidth overhead is $d$ times larger than that of standard perfect hashing. Strictly speaking, these approaches are not perfect hashing indexes. This also explains why MapEmbed [3] (described later in § II-B) can achieve a smaller index size (as small as 0.5 bits per item) than the theoretical minimum space (1.44 bits per item) of perfect hashing index. We aim at devising a standard perfect hashing index with the associativity of exact one. In addition, we hope the index can identify and filter non-existent items so as to accelerate insertion and negative lookup.

### B. KV Stores in Fast-Slow Memory Architecture

This paper focuses on the KV stores in tired memory. In such scenario, the large table storing KV pairs is kept in slow memory. In fast memory, KV stores can build cache [17], [18], index [19], [20], [21], [22], filter [23], or their combinations [24], [25], [26] to accelerate accessing the slow memory.

We highlight three types of fast-slow memory architectures. *1) Client-Server:* Besides popular server-based KV stores like Redis [27] and Memcached [18], recent KV stores in disaggregated memory [9], [28], [29] with a local compute pool and a remote memory pool (without computation power) also fall in this category. Designed for disaggregated memory, RACE [9] builds an RDMA-friendly hash table in remote memory achieving high load factor and fast lookups. *2) SRAM-DRAM:* This architecture is used by many works in networking community to achieve network function virtualization [30], [17], [31]. TEA [17] designs a variant of cuckoo hash table and uses it to store KVs (flow-level information) in external DRAM. It also uses a small cache in on-chip SRAM to store recent flows. *3) DRAM-NVMM (non-volatile main memory):* A line of recent works uses indexes (storing key fingerprints and location IDs) in DRAM to accelerate accessing the KVs stored in NVMM [19], [20], [21], [32], [33]. However, their

TABLE I: High-level comparison between CuckooDuo and prior works (under the default setting in § V, where the bucket size of all methods is $d = 8$). The best result(s) in each column is highlighted with yellow background. "BPI" refers to "bits per item", "Filter" refers to the functionality of identifying and filtering non-existent keys, "Expansion time" refers to the time of expanding a 30M KV table by 2× in active-mode (Figure 12).

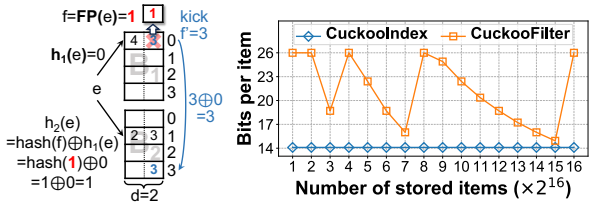
| Methods | Index Properties | | Table Properties | | Lookup Properties | | | Insert Properties | | | Update Properties | | | Delete Properties | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | BPI | Filter | Load factor | Expansion time ($\mu$s) | # accessed items (associativity) | #RTTs | Latency ($\mu$s) | # accessed items | #RTTs | Latency ($\mu$s) | # accessed items | #RTTs | Latency ($\mu$s) | # accessed items | #RTTs | Latency ($\mu$s) |
| CuckooDuo | 13∼16 | ✓ | 99% | 4.0 | 1 | 1 | 3.2 | 1∼6.7 | 1∼2.0 | 3.5∼6.9 | 2 | 2 | 6.2 | 1 | 1 | 3.2 |
| MapEmbed | 0.5∼4 | × | 91% | 8.1 | 8 | 1 | 3.3 | 9∼124 | 2∼3.4 | 6.7∼30.1 | 9 | 2 | 6.5 | 9 | 2 | 6.5 |
| RACE | 0 | × | 94% | 7.9 | 32 | 1 | 4.2 | 33 | 2 | 7.8 | 33 | 2 | 7.1 | 33 | 2 | 7.1 |
| TEA | 0 | × | 70% | N/A | 16 | 1 | 3.4 | 17 | 2∼12.7 | 6.7∼42.9 | 17 | 2 | 6.4 | 17 | 2 | 6.3 |

large indexes can cause high DRAM consumption, which is infeasible for production environment [34]. MapEmbed [3] proposes a small perfect hashing index theoretically fitting this scenario. EEPH [5] proposes another small index very similar to MapEmbed, and practically deploy it in a DRAM-NVMM architecture. The experiments in this paper are conducted in *client-server* architecture with RDMA networks. As a general KV store framework, our CuckooDuo can also be deployed in other fast-slow memory architectures. We assume remote server has no computation power as in disaggregated memory [29], [9]. Consider the scenario where remote server has computation power. The excellent work Catfish [35], [36] proposes to adaptively offload computation spent on accessing/modifying remote data structures to local server and RDMA-communication, thereby elegantly balancing the load between remote CPU and network communication. Inspired by Catfish, we could also batch some requests and send them to remote CPU for processing, so as to reduce network load and alleviate local CPU pressure. Recent RR-Compound [37] proposes an excellent RDMA-fused gRPC framework, which can also be applied in our setting for further speedup.

TABLE I provides a high-level comparison of CuckooDuo with the three typical prior works above. Although the index of CuckooDuo is larger than others, it provides the ability to identify non-existent keys, which helps improving the speed of insertion and negative lookup (§ III-B). Thanks to the perfect hashing index, CuckooDuo minimizes access bandwidth (# accessed items) and RTT to remote memory, and thus achieves the smallest latency. In addition, CuckooDuo also achieves the highest load factor and the smallest expansion time. We will present the detailed results and analyses in § V.

### C. Preliminary of CuckooHashing and CuckooFilter

**CuckooHashing [7]** is an efficient hashing table, which consists of 2 tables $\mathcal{B}_1$ and $\mathcal{B}_2$. Any item $e$ can be stored in one of its two candidate buckets $\mathcal{B}_1[h_1(e)]$ or $\mathcal{B}_2[h_2(e)]$ determined by two hash functions $h_1(\cdot)$ and $h_2(\cdot)$. To insert an item, we first check whether it can be directly inserted into one candidate bucket. If both candidate buckets are full, CuckooHashing randomly kicks away an item in one candidate bucket to make room, and the victim item is reinserted into its another candidate bucket. This procedure repeats until every item finds a bucket to settle down, or the maximum kick-out limit $L$ is reached. The load factor of CuckooHashing can be improved by increasing the number of candidate buckets and the number of slots in each bucket [38], [39]. Many subsequent works have emerged to optimize the speed of CuckooHashing [40], [41], [42], [43], [44], [45], [46]. There are many works using CuckooHashing to build KV tables [47], [48], [49], index [50], [40], or database systems [51], [52]. Mega-KV [50], [53] elegantly implements a CuckooHashing table (storing fingerprints and location IDs) on GPU, and uses it as an index to accelerate KV stores. This work is complementary to CuckooDuo. On the one hand, as CuckooIndex is essentially a CuckooHashing table, it can also be implemented on GPU following the design of Mega-KV to improve speed. On the other hand, some designs of CuckooDuo could help reducing the index size of Mega-KV. By using our one-to-one mapping design, Mega-KV could avoid explicitly storing location IDs. Mega-KV could also use our Dual-FP optimization to reduce the length of the stored fingerprints (from 32-bit to <16-bit).




(a) Example of CuckooFilter (m=$2^2$ and d=2) (b) Minimum space cost per item to achieve target false positive error rate of $\epsilon = 10^{-3}$ (d=8)

Fig. 2: Example of CuckooFilter and comparison of its space cost per item with our CuckooIndex.

**CuckooFilter (CF) [8]** is another excellent variant of CuckooHashing. It replaces the items in CuckooHashing with fingerprints (hash values) to perform membership query. As shown in Figure 2(a), a CF consists of 2 tables $\mathcal{B}_1$ and $\mathcal{B}_2$. Each table has $m$ buckets, each of which stores $d$ fingerprints. For an incoming item $e$, CF calculates hash functions to get its fingerprint $f = FP(e)$ and first candidate bucket index $h_1(e)$. Then it uses the exclusive-OR (XOR) operation to compute the second candidate bucket index $h_2(e) = hash(f) \oplus h_1(e)$. During the insertion process, if an existing item (with fingerprint $f'$ and current bucket index $i$) needs to be kicked away, CF computes its alternate bucket index $j$ by $j = hash(f') \oplus i$. Figure 2 shows an insertion example of CF. For incoming item $e$, CF calculates hash functions to acquire its fingerprint $f = FP(e) = 1$ and first candidate bucket index $h_1(e) = 0$. Then CF calculates its second candidate bucket index by $h_2(e) = hash(f) \oplus h_1(e) = 1 \oplus 0 = 1$ (we assume $hash(f) = f$ for simplicity). Since both candidate buckets are full, CF randomly kicks away a residing fingerprint $f' = 3$ in $\mathcal{B}_1[0]$ to make room for $f$. For fingerprint $f'$, CF calculates its alternate candidate bucket index by $j = hash(f') \oplus i = 3 \oplus 0 = 3$, and reinserts it into $\mathcal{B}_2[3]$.

Compared to standard BloomFilter [54], CF not only supports deletions but also has better time- and space- efficiency [8]. However, CF suffers a critical issue of space inflation, which varies with the input scale. This is because that CF uses XOR operation to compute alternate location, which requires

the number of buckets $m$ in each table to be a power of two. Otherwise, the XOR operation would result in an index landing on a memory address outside of the allocated boundaries. Consider a CF with $d = 1$. In the worst case, when the number of input items is $2^{16}$+1, CF still needs to set $m$=$2^{16}$, resulting in a total of $2{\times}m = 2^{17}$ buckets, with 50% buckets being wasted. Figure 2(b) further displays the minimum space cost per item of CF (and our CuckooIndex) to achieve $\epsilon = 10^{-3}$ false positive error rates under different input scales. We can see that CF has up to $2\times$ space inflation. By contrast, our CuckooIndex achieves constant and lower space cost because it has no size constraint. There are many variants of CuckooFilter. Adaptive CuckooFilter [55] optimizes the false positive error. VacuumFilter [56] breaks size limitation but cannot combine with the KV table. InfiniFilter [57] proposes a method to expand filter size by any power of two.

TABLE II: Symbols frequently used in this paper.

| Symbols | Meaning |
|---|---|
| $m$ | Number of buckets in each bucket array of CuckooDuo |
| $d$ | Number of slots in each bucket of CuckooDuo |
| $s$ | Size of the stash of CuckooDuo |
| $f$ | Length of the fingerprint (in bit) |
| $L$ | Predefined maximum length of kick-out path |
| $\mathcal{B}_i$ | The $i_{th}$ bucket array of CuckooVault ($i = 1$ or 2) |
| $\mathcal{I}_i$ | The $i_{th}$ bucket array of CuckooIndex ($i = 1$ or 2) |
| $h_i(\cdot)$ | Hash function mapping key into bucket |
| $FP(\cdot)$ | Hash function calculating the fingerprint of key |
| $FP_j(\cdot)$ | Fingerprint hashing function in *Dual-Fingerprint* optimization |
| $d_j$ | Number of slots per bucket in $\mathcal{I}_1$ using $FP_j(\cdot)$ |

## III. The CuckooDuo Algorithm

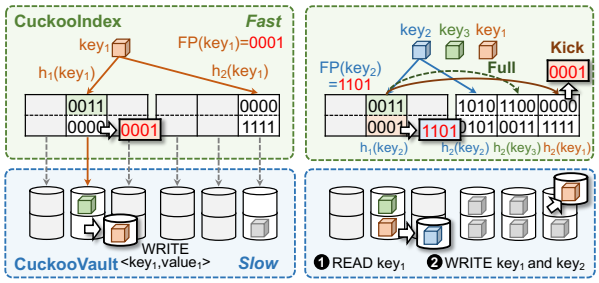
### A. Data Structure

CuckooDuo consists of two interlinked cuckoo hash tables (Figure 3): **1) A large KV table called CuckooVault** storing KV pairs (items). This table is deployed on slow remote memory. It has two bucket arrays ($\mathcal{B}_1$ and $\mathcal{B}_2$) with $m$ buckets. Each bucket has $d$ slots storing KV pairs. **2) A small perfect hashing index called CuckooIndex** storing fingerprints (calculated with a hash function $FP(\cdot)$) of inserted keys. This table is deployed on fast local memory. It has two bucket arrays $\mathcal{I}_1$ and $\mathcal{I}_2$ with the same size of $\mathcal{B}_1$ and $\mathcal{B}_2$. Each fingerprint in CuckooIndex has a one-to-one correspondence with a KV pair in CuckooVault. Each key is mapped into two candidate buckets in CuckooIndex/CuckooVault with two functions $h_1(\cdot)$ and $h_2(\cdot)$. To break the size constraint of CuckooFilter [8], we use modular addition/subtraction to replace XOR. For a $key$, the indexes of its two candidate buckets are $h_1(key)$=$\mathcal{H}(key)\%m$ and $h_2(key)$=$(h_1(key)$ $+hash(FP(key)))\%m$, where $\mathcal{H}(key)$ is a hash value. We have $h_1(key)$=$(h_2(key)-hash(FP(key)))\%m$. We deploy CuckooIndex/CuckooVault on local/remote server, where local server accesses remote memory with one-sided RDMA `READ`/`WRITE` requests.

### B. Basic Operations and Discussions

We temporarily assume that there are no fingerprint collisions. We will discuss how to handle fingerprint collisions and propose an optimization to reduce collisions in § III-C.

**Insertion (pseudocode in supplementary materials [58]):** Consider inserting item $e$=$\langle key, value\rangle$. We assume $key$ does not exist in CuckooVault. In practice, we should first judge whether $key$ has been inserted (if so, we update its value), which will be discussed later. The key idea is to use BFS to find the shortest kick-out path in local CuckooIndex, and then either directly `WRITE` $e$ into an empty slot (1 RTT), or simultaneously `READ` all items on the kick-out path and then `WRITE` them along with $e$ to their new locations (2 RTT).




(a) Example of directly inserting (b) Example of BFS

Fig. 3: Insertion examples ($m = 3$, $d = 2$, $L = 1$).

We first calculate $h_1(key) = \mathcal{H}(key)\%m$ and $h_2(key) = (h_1(key) + hash(FP(key)))\%m$ to locate candidate buckets $\mathcal{I}_1[h_1(key)]$ and $\mathcal{I}_2[h_2(key)]$ in CuckooIndex. We check whether there is an empty slot in $\mathcal{I}_1[h_1(key)]$ or $\mathcal{I}_2[h_2(key)]$. If so, we insert the item into this slot by writing $FP(key)$ into the slot in CuckooIndex, `WRITE` $e$ into the corresponding slot in CuckooVault, and return insertion success.

If both candidate buckets are full, we start the Breadth-First-Search (BFS) process to find the shortest kick-out path. We initialize an empty queue $Q$ and push all fingerprints in $\mathcal{I}_1[h_1(key)]$ and $\mathcal{I}_2[h_2(key)]$ into it. Then we repeatedly pop fingerprints from the front of $Q$ until the kick-out path length reaches predefined threshold $L$. For each popped fingerprint $f$, we check whether it can be inserted into its alternate candidate bucket. Specifically, we first calculate the index of its alternate candidate bucket. For example, if $f$ is stored in the first bucket array, we calculate $h_2(f) = (h_1(f) + hash(f))\%m$, where $h_1(f)$ is the index of the current bucket of $f$. Then we check the alternate candidate bucket $\mathcal{I}_2[h_2(f)]$. 1) If $\mathcal{I}_2[h_2(f)]$ has an empty slot, our BFS has found one of the shortest kick-out $path$. Then we perform the insertion operation on CuckooVault by first reading all items on the kick-out $path$ (with one batch of `READ` requests), and then writing these items (including $e$) into their destination slots (with another batch of `WRITE` requests). We finally update the fingerprints in CuckooIndex to ensure its one-to-one mapping relationship with CuckooVault, and return insertion success. 2) If $\mathcal{I}_2[h_2(f)]$ is also full, we continue BFS. Specifically, we push each fingerprint $f'$ in $\mathcal{I}_2[h_2(f)]$ and its corresponding kick-out path (if path length is smaller than the predefined maximum kick-out length $L$) into the tail of $Q$. This procedure repeats until $Q$ is empty, when we return insertion failure.

***Examples (Figure 3):*** We assume $hash(FP) = FP$. We provide example of insert failure in supplementary materials [58].

1) As shown in Figure 3(a), to insert $e_1 = \langle key_1, value_1\rangle$, we first locate its two candidate buckets. As there is an empty slot in $\mathcal{I}_1[h_1(key_1)]$, we directly insert the item into this slot by writing $FP(key_1)$ into one slot in CuckooIndex and writing

$e_1$ into the corresponding slot in CuckooVault.

2) As shown in Figure 3(b), both candidate buckets of $e_2$ are full, so we start BFS by enqueuing all fingerprints in candidate buckets. Then we dequeue the fingerprints one by one. For the first dequeued fingerprint $FP(key_3)$=0011, we calculate $h_2(key_3)$=$(h_1(key_3)+hash(FP(key_3)))\%m$=(1+3)%3=1. As the alternate bucket $\mathcal{I}_2[1]$ is also full, we continue to dequeue $FP(key_1)$=0001 and calculate $h_2(key_1)$=$(h_1(key_1)+hash(FP(key_1)))\%m$= (1+1)%3=2. As $\mathcal{I}_2[2]$ is not full, we have found the shortest kick-out path. We `READ` $e_1$= $\langle key_1,value_1\rangle$ from CuckooVault, `WRITE` $e_1$ and $e_2$ into their new locations, and finally update the fingerprints.

**Lookup:** To lookup a $key$, we check its two candidate buckets $\mathcal{I}_1[h_1(key)]$ and $\mathcal{I}_2[h_2(key)]$. If $FP(key)$ exists in $\mathcal{I}_1[h_1(key)]$ or $\mathcal{I}_2[h_2(key)]$, we send a RDMA `READ` request to read the corresponding KV pair $\langle key', value'\rangle$ from CuckooVault. If $key$ matches $key'$, we return $value'$ as the lookup result. Otherwise, if $FP(key)$ is not found in the candidate buckets or $key \neq key'$, we return a lookup failure.

**Deletion:** To delete a $key$, we first lookup it as described above. If the key exists in CuckooVault, we delete its fingerprint from CuckooIndex. Notice that we do not need to delete its KV pair from CuckooVault, because once the fingerprint of a key is removed from CuckooIndex, its KV pair in CuckooVault will eventually be overwritten by other item.

**Update:** To update the value of a $key$, we first lookup it as described above. If the key exists in CuckooVault, we send a RDMA `WRITE` request to update its $value$ in CuckooVault.

**CuckooIndex as a perfect hashing index:** Under current design, when looking up a $key$, CuckooIndex returns the location of a unique slot in CuckooVault without collision. Therefore, CuckooIndex satisfies the requirement of a perfect hashing index described in § II-A. Note that the internal structure of CuckooIndex (now using a double hashing approach) does not affect its overall status as a perfect hashing index. We can view the entire data structure of CuckooIndex as a function. This function qualifies as a perfect hashing index as long as it returns a unique location in the hash table (CuckooVault) for each inserted key. Actually, the simplest perfect hashing index could use a large hash table to store each key and its location ID, which suffers large space overhead.

**Working with a stash:** Following previous works [49], [59], [60], [61], [62], [63], we build a small stash in fast memory to store the items (KV pairs) with insertion failures as shown in Figure 1(b). This stash can be regarded a part of CuckooIndex. When lookup a $key$, we first lookup it in the stash. If $key$ is in stash, we directly return its value, and otherwise, we lookup its fingerprint in the candidate buckets. After using stash, for each $key$, CuckooIndex still returns the location of a unique slot in CuckooVault ($key$ will be filtered out if it is in stash), thus it still meets the requirement of perfect hashing index (§ II-A). Indeed, a large stash would impact both the space and speed of CuckooIndex. Fortunately, our results show that a small stash with the size of 8∼32 is sufficient to improve load factor (Figure 7(j)), and such a small stash does not affect lookup speed (Figure 7(k)).

**Multi-threading acceleration:** We can use multi-threading to accelerate CuckooDuo's operations, which requires maintaining consistency under concurrent access to remote memory. Many existing concurrency control approaches, such as lock-free or optimistic concurrency, incur retries upon access contention, leading to extra RTTs [9], [64], [16]. For example, RACE [9] ensures consistency by re-reading candidate buckets. With the one-to-one mapping design, CuckooDuo can perform concurrency control entirely in local CuckooIndex, ensuring RDMA requests do not conflict with each other. The key idea is to add Read/Write locks to the slots being read or modified in CuckooIndex. We describe the detailed design in supplementary materials [58]. With our lightweight concurrency control approach, CuckooDuo achieves higher throughput than prior works (Figure 10(g)-10(j)).

**Latency analysis:** *1) Lookup:* As CuckooIndex is a perfect hashing index, each lookup only reads one slot in remote memory, thus achieving a latency of one RTT. Note that the operations in local CuckooIndex (including double hashing calculation, checking candidate buckets, checking stash, *etc.*) can perform very fast, which accounts for only 3.8% total lookup latency (Figure 7(f)). *2) Insertion:* Under current one-to-one mapping design, local CuckooIndex is aware of the status of each slot in remote CuckooVault (including empty slots). By checking the local CuckooIndex, we can directly locate an empty slot in the candidate buckets, or find the kick-out path using BFS. Therefore, under low load factor (<70%), each insertion can be accomplished by directly `WRITE` the incoming item into an empty slot, taking only one RTT. In the worst case, each insertion first reads all items in kick-out path (with one batch of RDMA `READ` requests), and then writes these items and the incoming item into their new locations (with one batch of RDMA `WRITE` requests), taking two RTTs.

We neglect the time taken by local BFS. Indeed, when using large maximum kick-out path length $L$, BFS can be time-consuming and become latency bottleneck. However, our results show that a small kick-out path length $L$ is sufficient for CuckooDuo to achieve high load factor (Figure 7(d)), and under such small $L$, the time spent on local device accounts for only 5%∼10% total latency (Figure 7(e)). On the other hand, BFS ensures to find the shortest kick-out path, minimizing data movement and communication overhead. *3) Deletion:* Thanks to the one-to-one mapping design, deletion operation only deletes fingerprint in local CuckooIndex without deleting the actual item in remote memory, thus achieving one RTT latency. *4) Update:* Update operation takes two RTTs. However, when the fingerprint is sufficiently long such that the false positive error can be ignored, upon finding a matched fingerprint in CuckooIndex, we can directly update its $value$ in CuckooVault, thus reducing update latency to one RTT.

**Identify non-existent keys:** As discussed in § II-A, existing perfect hashing indexes [3], [1] cannot identify non-existent keys, and they return a random location for each non-existent key. By contrast, CuckooIndex offers the functionality of identifying and filtering out non-existent keys. For each non-existent $key$, if its fingerprint $FP(key)$ does not collide with

the fingerprints in its two candidate buckets, CuckooIndex can report it as non-existent. Similar as CuckooFilter [8], the upper bound of the probability of a false fingerprint hit is $\epsilon = 1 - \left(1 - 1/2^f\right)^{2d} \approx 2d/2^f$, where $1 - 1/2^f$ is the probability that $FP(key)$ does not collide with one stored $f$-bit fingerprint, and $\left(1 - 1/2^f\right)^{2d}$ is the probability lower bound that $FP(key)$ does not collide with all stored fingerprint in the two candidate buckets (there are at most $2d$ stored fingerprints in the two candidate buckets). We can see that when using $\geqslant 14$-bit fingerprints, the false positive error of CuckooIndex is $< 10^{-3}$ (Figure 8(b)).

We discuss the importance of the non-existent key identification functionality. *1) Accelerating illegal lookup (update / delete):* Consider looking up a non-existent $key$. Existing hashing indexes return a random location for $key$, and thus still require reading one item $key'$ from slow memory and comparing $key$ with $key'$. By contrast, CuckooIndex can directly identify and filter out $key$ without accessing slow memory. As discussed above, lookup time is dominated by the communication RTT of accessing slow memory. Therefore, CuckooDuo can significantly reduce the latency of negative lookups (>20× as in Figure 10(f)). *2) Accelerating insertion:* Consider inserting an item $e = \langle key, value \rangle$. In practice, we first need to check whether $key$ has already been inserted. If yes, we update its value; if not, we proceed with the standard insertion process. Similarly, existing indexes require accessing slow memory to determine whether $key$ has been inserted. By contrast, CuckooIndex can identify non-existent keys, allowing the insertion process to be directly executed. This significantly reduces the insertion latency of CuckooDuo by saving one RTT to `READ` slow memory (∼2× as in Figure 10(a)).

**Space overhead of CuckooIndex:** We discuss the space overhead of CuckooIndex and demonstrate that it only exceeds the theoretical lower bound by about 2.5 bits per item (BPI). As discussed above, CuckooIndex has $\epsilon \approx 2d/2^f$ false positive error. Therefore, BPI of CuckooIndex is $BPI = f/\alpha = \left(log_2(1/\epsilon) + log_2(2d)\right)/\alpha$, where $\alpha$ is the load factor of CuckooDuo. Recall that the theoretical space lower bound of perfect hashing index is $\underline{BPI_P} = 1.44\alpha$ [14]. On the other hand, the theoretical space lower bound of a filter is $\underline{BPI_F} = log_2(1/\epsilon)$ [65]. The space lower bound for a prefect hashing index that also provides the functionality of filters is $\underline{BPI} = \underline{BPI_P} + \underline{BPI_F}$[1]$= 1.44\alpha + log_2(1/\epsilon)$. We will see that CuckooIndex with $d = 8$ can achieve nearly 100% load factor (98.0% as in Figure 7(a)). When $\alpha \approx 100\%$, CuckooIndex with $d = 8$ has $BPI \approx log_2(1/\epsilon) + log_2(2 \times 8) = log_2(1/\epsilon) + 4$, which only exceeds the theoretical lower bound $\underline{BPI} = 1.44 + log_2(1/\epsilon)$ by about 2.5 BPI. Although exceeding the theoretical optimum, CuckooIndex supports dynamic update and deletion with $O(1)$ time-complexity. By contrast, most filters [65] and perfect hashing indexes [2], [66], [67] that approach the theoretical optimum can only be built on static set, which limits their practicality. Actually, CuckooIndex has the same structure as CuckooFilter [8], and thus has the same space overhead ($BPI \approx log_2(1/\epsilon) + log_2(2d)$). However, the original paper of CuckooFilter did not mention its potential usage as an index. We believe that the additional $log_2(2d)$ BPI space of CuckooFilter is underutilized. By exploring the indexing functionality, CuckooIndex maximizes the value of each bit in CuckooFilter and fully unlocks its potential.

We further discuss the performance of CuckooIndex under different space overhead in Figure 8, showing that when $f \geqslant 14$, the fingerprint collision rate is enough small for CuckooIndex to achieve high load factor and small insertion latency and negative lookup latency. In our setting with 64-byte keys/values and $f = 16$ bit fingerprints, the ratio of fast memory usage to slow memory usage is $\frac{M_{fast}}{M_{slow}} \approx \frac{16}{64 \times 2 \times 8} \approx 0.016$. A recent RocksDB study from Meta reports that the average key and value sizes are 27 bytes and 126 bytes [68], where $\frac{M_{fast}}{M_{slow}} \approx \frac{16}{(27+126) \times 8} \approx 0.013$. Such a memory consumption ratio can fit recent fast-slow tired memory systems [69].

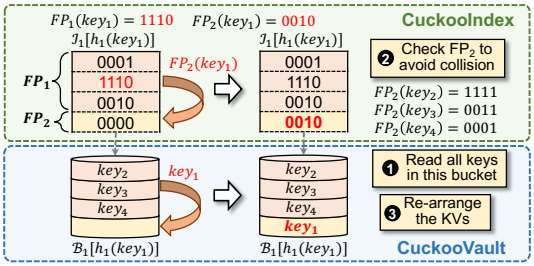



Fig. 4: Example of dual-fingerprint optimization.

### C. Handling Fingerprint Collision

In previous subsection, we assumed the fingerprints of different keys would not collide, which is unrealistic. This subsection describes how to handle fingerprint collision. We will theoretically prove that our solution can effectively reduce the number of collided items by $d/2^f$ in § IV.

**Naive solution:** A naive solution is to insert the items with fingerprint collision into the stash. When inserting a new item with $key$, we first check whether $FP(key)$ exists in its two candidate buckets. If so, we retrieve the corresponding $key'$ with the matched fingerprint from CuckooVault. If $key' \neq key$, we confirm that the new item is undergoing a fingerprint collision, and insert it into the stash. This solution ensures each inserted item has only one matched fingerprint in its two candidate buckets. To lookup a $key$, we first lookup it in the stash, and then lookup its fingerprint in candidate buckets. As discussed in § III-B, the stash should be small enough so as not to impact the space and speed of CuckooIndex. Unfortunately, our results show that, to achieve 90% load factor, there are $>$ 3000 collided items to be stored in the stash (Figure 6(a)), which is unacceptable. We further propose the *Dual-Fingerprint* optimization to reduce the probability of fingerprint collision and thus reduce the stash size.

**Dual-Fingerprint optimization:** The key idea is to change another fingerprint hashing function for the items with fingerprint collisions, and thus grant colliding item a second chance to be

[1] $\underline{BPI} = \underline{BPI_P} + \underline{BPI_F}$ rather than $\underline{BPI} = \max\left\{\underline{BPI_P}, \underline{BPI_F}\right\}$ because filter and perfect hashing index are independent with each other. Prefect hashing index returns a random location for a non-existent key, which provides no useful information for filters. Conversely, filter also provides no information for perfect hashing.

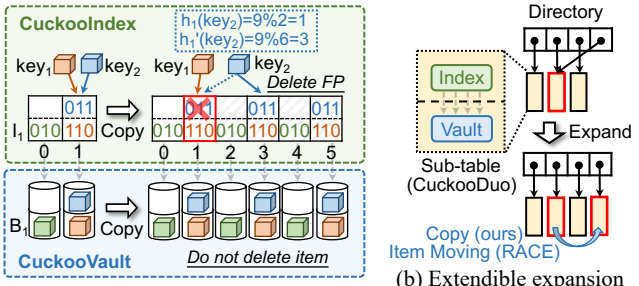
inserted. Only when collision cannot be resolved by changing fingerprint do we insert the colliding item into stash.

As shown in Figure 4, we use two hash functions $FP_1(\cdot)$ and $FP_2(\cdot)$ to calculate fingerprints. For each bucket in the first array of CuckooIndex ($\mathcal{I}_1$), we divide its $d$ slots into two parts ($d = d_1 + d_2$). For the first $d_1$ slots (called *primary slots*), we use the first fingerprint hash function $FP_1(\cdot)$. For the remaining $d_2$ slots (called *backup slots*), we use the second fingerprint hash function $FP_2(\cdot)$. The fingerprints in the backup slots will not be enqueued during BFS. In Figure 4, we have $d_1 = 3$ and $d_2 = 1$. We stipulate that when conducting fingerprint comparisons in lookup operation, fingerprints in the backup slots have higher priority. That is, to lookup a $key$, if its fingerprints $FP_1(key)$ and $FP_2(key)$ are both in $\mathcal{I}_1[h_1(key)]$, we consider fingerprint $FP_2(key)$ in the backup slot to be valid, and proceed to retrieve the item from its corresponding slot in CuckooVault. In insertion operation, we prioritize placing incoming keys into primary slots. Only when all primary slots in both candidate buckets are full do we insert the key into a backup slot. If, during insertion, we discover that a fingerprint of the incoming key already exists in one candidate bucket, we `READ` all keys in this bucket. If we ascertain that a fingerprint collision has occurred between the incoming key and an existing key, we make adjustments between the keys in primary slots and backup slots in an attempt to resolve the conflict. For example, if the incoming $key$ has a $FP_1$ fingerprint collision in one primary slot, we try to move one of colliding keys to a backup slot. As backup slot has higher priority, we must ensure that the keys in primary slots do not have $FP_2$ fingerprint collisions with the moved key. The results show that fingerprint adjustments happen very infrequently (Figure 7(i)), but can effectively reduce stash size from >3000 to <32 (Figure 7(g)) and improve load factor from 40% to 99% (Figure 7(h)).

**Example (Figure 4):** When inserting $key_1$, we find its fingerprint already exists in a primary slot in $\mathcal{I}_1[h_1(key_1)]$. We `READ` all keys in $\mathcal{I}_1[h_1(key_1)]$, and ascertain $key_1$ has $FP_1$ collision with $key_3$. Then we attempt to resolve collision by making fingerprint adjustment in $\mathcal{I}_1[h_1(key_1)]$. We try to insert $FP_2(key_1)$ into backup slot under the premise of the keys in primary slots do not have $FP_2$ collision with $key_1$. We calculate $FP_2(key_2)$, $FP_2(key_3)$, and $FP_2(key_4)$ to make sure they do not collide with $FP_2(key_1)$. Finally, we `WRITE` the moved items to their destinations.

### D. Dynamic Expansion

**Background:** When the table is full, KV store needs to expand its size to accommodate larger dataset. Most existing solutions expand table by moving items [9], [6], [5]. When the table is full, they create a new table of the same size, and move nearly half items to the new table. In the implementation of RACE [9], this requires the client to sequentially read each bucket in remote old table, judge which table each item belongs to (by computing a hash value), and move corresponding items by deleting them from the old table and writing them into the new table. Recent MapEmbed [3] proposes a lazy-mode expansion method based on memory copy. When the table is full, MapEmbed registers a large table with twice size and copies the old table into it twice following some rules. Afterwards, MapEmbed deletes redundant items when their buckets are first accessed. Additionally, MapEmbed also offers an active-mode expansion, which also requires moving items and is nearly identical to RACE. Next, we first introduce the basic method to expand one CuckooDuo, and then combine our method with extendible expansion [10] to amortize full-table expansion time into sub-table expansions and smooth space utilization during insertion process.




(a) Example of expanding a table by 3x

(b) Extendible expansion with a directory

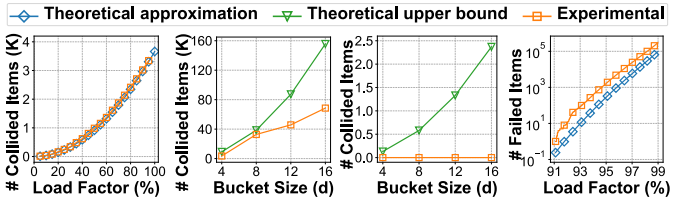
Fig. 5: Illustration of dynamic expansion.

**Basic expansion:** The expansion of CuckooDuo is based on memory copy, which supports any integer expansion ratio $r$. To expand a CuckooDuo by $r$ times, we first perform a copy operation to copy each bucket array of CuckooIndex and CuckooVault by $r-1$ times, and append the copied buckets to the original array. For example, consider the first array in CuckooIndex $\mathcal{I}_1[0], \cdots, \mathcal{I}_1[m-1]$. After memory copy, it becomes $\mathcal{I}_1[0], \cdots, \mathcal{I}_1[r \cdot m - 1]$ where $\mathcal{I}_1[i], \mathcal{I}_1[m+i], \cdots, \mathcal{I}_1[(r-1) \cdot m + i]$ ($i \in \{0, 1, \cdots, m-1\}$)) are identical. We modify the two hash functions used for selecting candidate buckets from $h_1(key) = \mathcal{H}(key)\%m$ and $h_2(key) = (h_1(key) + hash(FP(key)))\%m$ to $h_1'(key) = \mathcal{H}(key)\%(rm)$ and $h_2'(key) = (h_1(key) + hash(FP(key)))\%(rm)$. In this way, after memory copy, each $key$ still exists in its candidate buckets because $h_1'(key) = \mathcal{H}(key)\%(rm) \in \{\mathcal{H}(key)\%m, \mathcal{H}(key)\%m + m, \cdots, \mathcal{H}(key)\%m + (r-1)m\}$.

After memory copy, there exist redundant items in buckets. CuckooDuo also offers two expansion modes: active-mode and lazy-mode. 1) In active-mode, CuckooDuo immediately checks all buckets to remove the redundant items after memory copy. For each bucket $\mathcal{I}_1[j]$, we `READ` all keys stored in $\mathcal{B}_1[j]$, calculate hash function to judge whether each $key$ should be stored in $\mathcal{I}_1[j]$ by checking whether $h_1'(key) = j$. If not, we remove its fingerprint $FP(key)$ from $\mathcal{I}_1[j]$ in CuckooIndex. Thanks to the one-to-one mapping relationship between CuckooIndex and CuckooVault, CuckooDuo does not need to delete the actual item in remote CuckooVault like MapEmbed [3], thus achieving nearly 2× smaller expansion time (Figure 12(a)-12(c)). 2) In lazy-mode, CuckooDuo marks each bucket with a 1-bit indicator (stored in local memory) after expansion. Afterwards, every time a marked bucket $\mathcal{I}_1[j]$ is first accessed, we incidentally `READ` and check all items in it and clean the redundant items as described above. Lazy-mode further reduces the expansion time by 6.67× (Figure 12(d)).

**Example (Figure 5(a)):** We use an example to illustrate the process of expanding a CuckooDuo by $r = 3$ times. For clarity, we consider only the first bucket array $\mathcal{I}_1/\mathcal{B}_1$ with $m = 2$. The expansion operation works by copying $\mathcal{I}_1/\mathcal{B}_1$ by 2 times, resulting in $r \cdot m = 6$ buckets in each array. Afterwards, to clean the redundant items in $\mathcal{I}_1[1]$ in active-mode (or when it is first accessed in lazy-mode), we READ all keys in it and check whether each key should be stored here. For $key_2$, since $h_1'(key_2) = 9\%6 = 3$, its candidate bucket becomes $\mathcal{I}_1[3]$, so we delete its fingerprint from $\mathcal{I}_1[1]$. As described above, we do not delete the actual item in remote CuckooVault, which will eventually be overwritten by subsequent inserted items.

**Extendible expansion:** Prior work [10] developed extendible expansion to amortize the time of full-table expansion to multiple sub-table expansions, and also smooth space utilization during the insertion process (Figure 12(f)). Recently, extendible expansion is widely used by numerous works on KV store [9], [5], [6]. As shown in Figure 5(b), extendible expansion works by maintaining a directory, which indexes each item into a sub-table. When a sub-table is full, it is split into two by creating a new sub-table and moving nearly half items in it to the new sub-table [9]. The expansion procedure of CuckooDuo can also work in this extendible manner. As shown in Figure 5(b), we use CuckooDuo to replace the sub-table in extendible expansion, which consists of a CuckooIndex and a CuckooVault. The directory along with all CuckooIndexes are stored in fast memory, and all CuckooVaults are stored in slow memory. Every time a CuckooDuo (sub-table) is full, we split it into two CuckooDuos by performing the memory copy operation as described in basic expansion. Afterwards, we can either immediately read all buckets to clean redundant items (active-mode), or clean redundant items incidentally when their buckets are accessed (lazy-mode). Due to page limitation, we describe the specific details in our supplementary materials [58]. Our results show that CuckooDuo works well under extendible expansion, achieving higher space utilization than RACE [9] (Figure 12(f)-12(g)).




(a) FP (f = 16) (Thm IV.1) (b) Dual-FP (f = 8) (Thm IV.2) (c) Dual-FP (f = 16) (Thm IV.2) (d) BFS (L = 1) (Thm IV.3)

Fig. 6: Comparison of experimental and theoretical results.

## IV. Mathematical Analysis

We first analyze the probability of fingerprint collision and theoretically demonstrate the effectiveness of our *Dual-Fingerprint* optimization. Then we analyze the probability of insertion failure. The detailed proofs are provided in our supplementary materials [58]. We also validate our theoretical results with experiments under the default setup in § V.

**Analyses for fingerprint collision:** We first derive the number of fingerprint collisions in basic CuckooDuo in Theorem IV.1. Then we derive the the number of unresolvable fingerprint collision in CuckooDuo with *Dual-Fingerprint* in Theorem IV.2. Our theoretical results show that *Dual-Fingerprint* optimization effectively reduces the number of collided items by $d/2^f$.

**Theorem IV.1.** *Consider a basic CuckooDuo under the load factor of $\alpha$. Let $X_\alpha$ be the number of items failed to be inserted into CuckooDuo due to fingerprint collisions. We have* $\mathbb{E}(X_\alpha) \approx 2md^2\alpha^2/2^f \leqslant 4md^2\alpha^2/2^f = O\left(\frac{md^2\alpha^2}{2^f}\right)$.

**Theorem IV.2.** *Consider a CuckooDuo with Dual-Fingerprint optimization. Let $X$ be the number of items failed to be inserted into CuckooDuo due to fingerprint collisions. We have* $\mathbb{E}(X) \leqslant \frac{4md(d+1)(d-1)}{3\cdot 2^{2f}} = O\left(\frac{md^3}{2^{2f}}\right)$.

***Experimental analysis (Figure 6(a)-6(c)):*** Figure 6(a) shows the number of items with fingerprint collision in basic CuckooDuo (Theorem IV.1). The theoretical results are highly consistent with our experimental results, and there are thousands of collided items in basic CuckooDuo, meaning that basic CuckooDuo should use a large stash to hold these items. Figure 6(b) and Figure 6(c) show the number of items with unresolvable fingerprint collision in *Dual-Fingerprint* CuckooDuo with $f = 8$ and $f = 16$ at 99% load factor. The experimental results are always smaller than our theoretical upper bounds. When $f = 16$, both the theoretical and experimental number of collided items become smaller than 1, showing that the fingerprint collision rate is negligible.

**Analyses for insertion failure:** We derive the number of BFS failures in Theorem IV.3 based on basic CuckooDuo. Our conclusion can easily extend to *Dual-Fingerprint* CuckooDuo.

**Theorem IV.3.** *Consider a basic CuckooDuo under the load factor of $\alpha$. Let $Y_\alpha$ be the number of items failed to be inserted into CuckooDuo due to BFS failure (i.e., the length of kick-out path exceeds the predefined threshold $L$). We have* $\mathbb{E}(Y_\alpha) \approx 2md\int_0^\alpha \frac{\beta(r)^{2\Sigma_{i=0}^{L} d^i}}{1-\beta(r)^{2\Sigma_{i=0}^{L} d^i}} dr$, *where $\beta(r)$ is the ratio of full buckets under the load factor of $r$.*

***Experimental analysis (Figure 6(d)):*** Although we cannot directly obtain the analytical solution of $\mathbb{E}(Y_\alpha)$ from Theorem IV.3, we can give its numerical solution under a specific setting by numerical simulation. Figure 6(d) shows the number of items with BFS failure, where we set $L = 1$ to enhance the failure chance to obtain clearer results. The theoretical results are consistent with our experimental results, and the theoretical and experimental results both remain 0 at <91% load factor.

## V. Experimental Results

**Testbed, workloads, and setup:** We run all experiments in a testbed with two servers and one switch interconnected via RDMA networks. More details can be found in supplementary materials [58]. We use YCSB [70] to create request workloads with 64-byte keys and 64-byte values. The KV-table has 30M slots. We also conduct large-scale experiments with 1G table size, and the results are in our supplementary materials [58]. We use YCSB to create a loading workload with 30M distinct

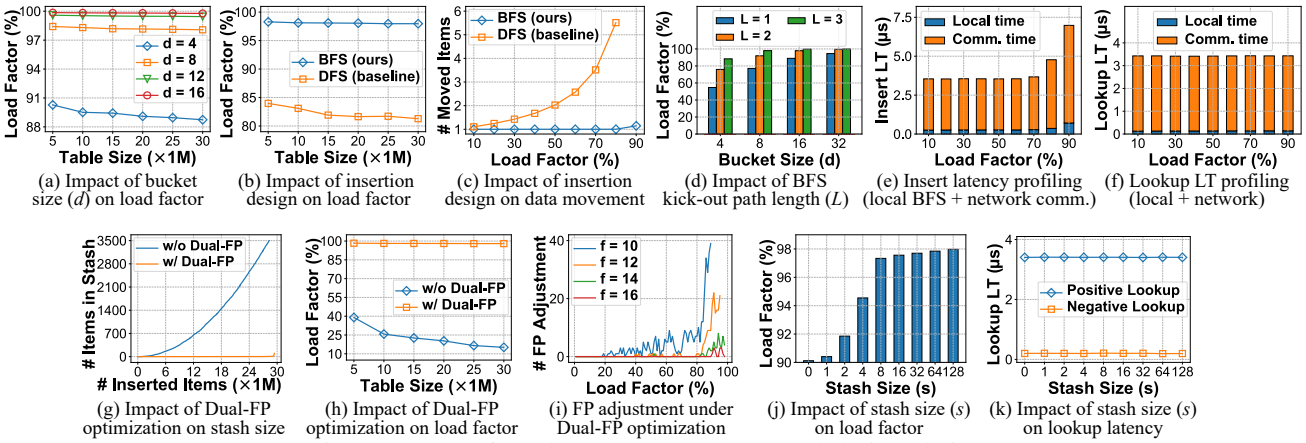


Fig. 7: Impact of CuckooDuo's parameters and design choices.

insert requests, which is used to load the KV-table to different load factors. We create multiple 300K running workloads with insert/lookup/update/delete (or hybrid) requests, which are used for performance evaluation at different load factors. By default, the lookup/update requests in the running workloads follow default Zipfian distribution of $\theta = 0.99$. The running workloads contain only legal requests, meaning they will not lookup/update/delete non-existent keys, nor insert existent keys. Some experiments also use running workloads with different ratios of illegal requests (Figure 11), or various request modes and distributions (Table III). By default, we set $d = 8$, $f = 16$, $s = 32$, $L = 3$, and enable Dual-Fingerprint optimization ($d_1/d_2 = 3/1$).

### A. Effect of Parameters and Ablation Studies

**Impact of bucket size ($d$) (Figure 7(a)):** We find larger bucket goes with higher load factor. The load factor of CuckooDuo is $90.0\%/98.0\%/99.7\%/99.9\%$ when $d = 4/8/12/16$. The results validate that CuckooDuo should store multiple items in one bucket to improve space utilization. We recommend setting $d = 8$, as larger bucket offers negligible performance gains, and could reduce speed and increase false positive error.

**Impact of BFS-based insertion (Figure 7(b)-7(c)):** We compare the load factor and data movement of CuckooDuo with our BFS-based insertion and traditional DFS-based insertion (with maximum kick-out path length of 50). BFS-based/DFS-based CuckooDuo has 98.0%/81.3% load factor, and 1.14/5.52 average data movement at the highest load factor. The results validate that our BFS-based insertion design described in § III-B can significantly improve load factor and reduce data movement compared to DFS-based insertion in standard CuckooHashing [7]. This is because BFS-based insertion can efficiently search for more feasible insertion solutions and ensure to find the shortest kick-out path, and thus minimize the chance of insertion failure and data movement.

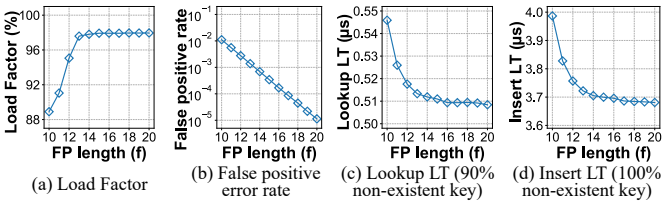
**Impact of BFS kick-out path length ($L$) (Figure 7(d)):** We evaluate the impact of BFS kick-out path length ($L$) on load factor. When $d = 8$, CuckooDuo with $L = 3$ already achieves $98.0\%$ load factor. The results validate that a small $L$ is sufficient for our BFS to find feasible insertion solutions and achieve high load factor. We recommend setting $L = 3$ to avoid excessively long BFS search time.

**Insert latency profiling (Figure 7(e)):** We conduct a profiling analysis on the insert latency of CuckooDuo, breaking it down into the time spent on local device (including BFS search, fingerprint collision adjustment, hash computation, *etc.*) and network communication. At 90% load factor, the time spent on local device and network communication are $0.71\mu s$ and $6.98\mu s$. Overall, local time accounts for only 5%∼10% total latency. This is because CuckooDuo uses small $L$ to ensure BFS search time remains short, and as a result, the insert latency is mainly dominated by the number of RTTs ($\sim 3.2\mu s$ per RTT) in network communication. The results validate that our BFS-based insertion in § III-B can perform very fast and will not become the latency bottleneck.

**Lookup latency profiling (Figure 7(f)):** We conduct a profiling analysis on the lookup latency, breaking it down into the time spent on local device (including hash computation, finding fingerprint in candidate buckets, checking stash, *etc.*) and network communication, which are about $0.13\mu s$ and $3.29\mu s$ respectively. Overall, local time accounts for only 3.8% total latency. Similar as insert latency, lookup latency is also dominated by network communication ($\sim 3.2\mu s$ per RTT). The results validate that the computation and lookup process in our double hashing based local CuckooIndex can perform very fast, and will not become the latency bottleneck.

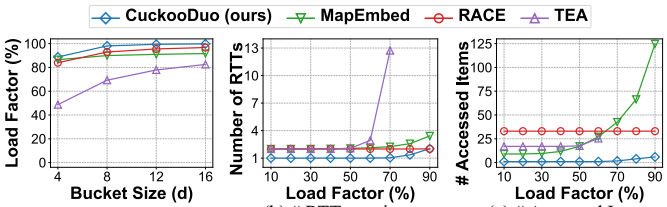
**Impact of Dual-Fingerprint optimization (Figure 7(g)-7(i)):** We compare the number of items in stash and the load factor of CuckooDuo before and after applying the Dual-FP optimization. As shown in Figure 7(g), without Dual-FP optimization, the number of items in stash grows quadratically with the input scale, which is consistent with Theorem IV.1. After using Dual-FP optimization, the number of items in stash drastically decreases and always remains below 32. As shown in Figure 7(h), when fixing $s = 32$, the Dual-FP optimization improves load factor from $40\%$ to $98\%$. We also evaluate the number of FP adjustment under Dual-FP optimization in Figure 7(i), indicating that fingerprint adjustments do not occur frequently, and thus will not lead to excessive data movement. These results validate that the Dual-FP optimization described in § III-C can effectively reduce fingerprint collisions, and thus significantly improve load factor with minimal data movement.

**Impact of stash size ($s$) (Figure 7(j)-7(k)):** We evaluate the




(a) Load Factor (b) False positive error rate (c) Lookup LT (90% non-existent key) (d) Insert LT (100% non-existent key)

Fig. 8: Impact of fingerprint length ($f$).




(a) Load factor (b) # RTTs per insert (latency) (c) # Accessed Items per insert (bandwidth)

Fig. 9: Comparison of insertion properties.

impact of stash size on load factor and lookup latency. As shown in Figure 7(j), a small stash of $s = 32$ can already improve the load factor from 90.1% to 97.1%. As shown in Figure 7(k), lookup latency is not affected by stash size. This is because our stash is small enough to fit in CPU cache, allowing for very fast query speed. We recommend to set $s = 32$ because this small stash is sufficient for high load factor and does not impact lookup speed. These results validate that the stash mechanism described in § III-B-III-C can effectively improve load factor without affecting speed.

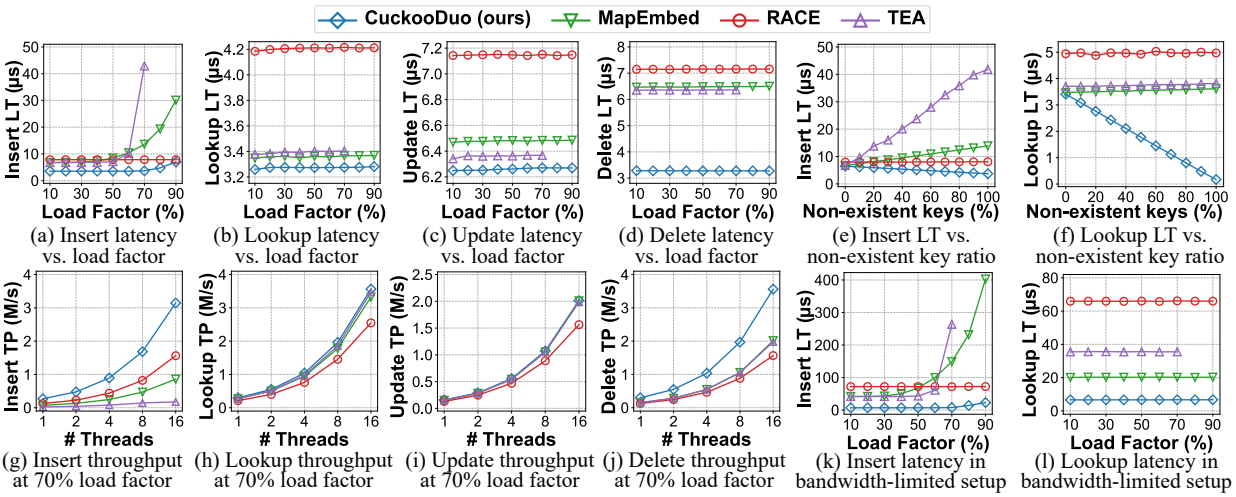
**Impact of fingerprint length ($f$) (Figure 8):** We evaluate the performance of CuckooDuo under different fingerprint length ($f$). As shown in Figure 8(a), when $f < 14$, larger $f$ goes with higher load factor because of fewer fingerprint collisions. When $f \geqslant 14$, load factor no longer increases with $f$ because of small fingerprint collision rate, and at this point, load factor is limited by bucket size $d$. Figure 8(b) shows the relationship between CuckooIndex's false positive error and $f$, which is consistent with the theoretical analysis in § III-B. Figure 8(c) shows the lookup latency, where lookup set has 90% non-existent keys. The lookup latency decreases with the increase of $f$ because of smaller error rate in filtering negative keys. Figure 8(d) shows the insert latency, where all keys in the insert set are non-existent keys. Similarly, the insert latency also decreases with the increase of $f$. However, when $f \geqslant 14$, the lookup/insert speed no longer significantly improves with increasing $f$, as the error rate is already very low. These results validate that our CuckooIndex can effectively identify non-existent keys, thereby enhancing lookup/insert speed by reducing unnecessary accesses to remote memory as described in § III-B. We recommend setting $f = 16$ to simultaneously attain high load factor and small filter error. Additionally, the 16-bit aligned memory addresses at this setting also allow for hardware optimizations like SIMD [71].

### *B. Comparison with Prior Art*

We compare CuckooDuo with three KV-stores: MapEmbed [3], RACE [9], and TEA [17]. To ensure a fair comparison, we implement these KV-stores in our testbed and use RDMA doorbell batching [72] to maximize their performance at our best. We set bucket size $d = 8$ for all solutions. We use a large stash ($s = 8192$) for TEA to improve its load factor, and the other solutions use a small stash $s = 32$. The other parameters of the baseline solutions are set according to their papers.

**Insert properties (Figure 9):** As shown in Figure 9(a), CuckooDuo has the highest load factor thanks to its BFS-based insertion and Dual-FP optimization. When $d = 8$, the load factor of CuckooDuo, MapEmbed, RACE, and TEA are 98.1%, 90.0%, 92.9%, and 69.2%. As shown in Figure 9(b), CuckooDuo has the fewest number of RTTs, which is 1 under <70% load factor and at most 2. As described in § III-B, this is because the one-to-one design ensures CuckooDuo can find the location of empty slot or the slots on kick-out path without accessing remote memory. Therefore, the insertion can be accomplished with 1 RTT of directly `WRITE` the empty slot (at low load factor), or worst-case 2 RTTs of `READ` and `WRITE` the slots on the kick-out path. By contrast, other solutions require at least 2 RTTs (one to `READ` bucket status and another to `WRITE`). At high load factor, the number of RTTs of MapEmbed and TEA increases due to more item movement. As shown in Figure 9(c), CuckooDuo achieves minimal item accessing, thus minimizing bandwidth overhead. As describe above, CuckooDuo nearly does not need to read items in remote memory under <70% load factor. By contrast, other solutions need to read remote memory at the granularity of entire buckets, and thus have large item accessing volume determined by bucket size. Additionally, the item movement procedure of MapEmbed and RACE at high load factor also increases their item accessing. We also evaluate the number of memory access and item movement during insertion, and the results are shown in our supplementary materials [58].

**Latency (Figure 10(a)-10(f)):** *1) Insert latency (Figure 10(a)):* The latency results are consistent with the RTT results in Figure 9(b) (∼3.2$\mu$s per RTT), which again validates insert latency is dominated by the number of RTTs in network communication rather than local computation time. The insert latency of CuckooDuo remains about 3.55$\mu$s under <70% load factor, at least half of the others. At 90% load factor, the insert latency of CuckooDuo grows to worst-case 6.98$\mu$s (2 RTTs), while that of MapEmbed and RACE are 30.1$\mu$s and 7.85$\mu$s. The latency improvement (1.9∼6.2×) is not proportional to bandwidth improvement (9.0∼18.5× in Figure 9(c)) because many accesses to remote memory can be executed concurrently. For example, prior works can access the $d$ items in a bucket with one memory access, thus requiring only one RTT. Additionally, the RDMA batch mechanism used in our implementation also further optimizes the speed of prior work. *2) Lookup latency (Figure 10(b)):* Although all solutions take 1 RTT for lookup, CuckooDuo has smaller lookup latency than others (0.10∼0.92$\mu$s) because of less bandwidth usage (Table I). *3) Update latency (Figure 10(c)):* Similarly, all solutions take 2 RTTs for update. CuckooDuo has smaller update latency than others (0.30∼0.91$\mu$s) because of less bandwidth usage (Table I). *4) Delete latency (Figure 10(d)):* The delete latency of CuckooDuo is ∼3.2$\mu$s (1 RTT), at least 2× smaller than others. This is because under the one-to-




(a) Insert latency vs. load factor (b) Lookup latency vs. load factor (c) Update latency vs. load factor (d) Delete latency vs. load factor (e) Insert LT vs. non-existent key ratio (f) Lookup LT vs. non-existent key ratio

(g) Insert throughput at 70% load factor (h) Lookup throughput at 70% load factor (i) Update throughput at 70% load factor (j) Delete throughput at 70% load factor (k) Insert latency in bandwidth-limited setup (l) Lookup latency in bandwidth-limited setup

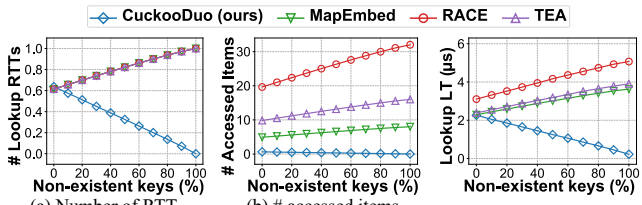
Fig. 10: Comparison of latency (LT) and throughput (TP) with prior solutions.

one mapping design, CuckooDuo can only delete fingerprint in local CuckooIndex without deleting the actual item in remote memory, thus reducing one `WRITE` RTT. *5) Latency with illegal requests (Figure 10(e)-10(f)):* We evaluate the insert/lookup latency on running workloads containing illegal requests at 70% load factor. As the ratio of non-existent keys increases from 0% to 100%, the insert latency of all solutions varies from their update latency to insert latency at 70% load factor (Figure 10(e)). CuckooDuo always has the smallest insert latency because both its update latency and insert latency at 70% load factor are smaller than others (Figure 10(a)-10(c)). As the ratio of non-existent keys increases from 0% to 100%, the lookup latency of CuckooDuo drops from 3.41$\mu$s to 0.17$\mu$s (Figure 10(f)), whereas that of the others remains unchanged. This is because CuckooIndex can identify non-existent keys, thus avoiding many unnecessary accesses to remote memory.

**Throughput (Figure 10(g)-10(j)):** We implement the multi-threading version of CuckooDuo and the other works using the RW-lock mechanism described in § III-B. The throughput results are consistent with the latency results in Figure 10(a)-10(d). As the number of threads increases by 2×, the speed improves by 1.6∼1.8×. The reason for this disparity is the concurrency control mechanism incurs extra overhead.

**Latency in bandwidth-limited setting (Figure 10(k)-10(l)):** To better highlight the advantages of CuckooDuo, we conduct experiments in a bandwidth-limited setting with 10× smaller NIC port bandwidth and 16× larger key/value length compared to the default setting. Under this bandwidth-limited setting, CuckooDuo demonstrates a more significant speed improvement with 6.1∼17.6× smaller insert latency and 3.0∼9.8× smaller lookup latency than the other works. We provide more details about the bandwidth-limited setting and more experimental results in supplementary materials [58].

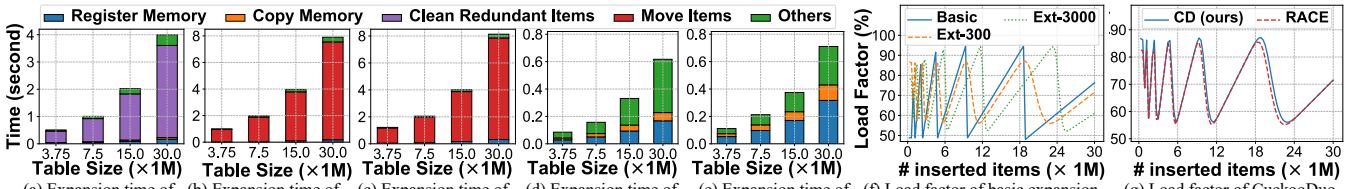
**Performance under the same fast memory size (Figure 11):** As shown in Table I, the index structure of CuckooDuo requires 13∼16 bits per item (BPI) of local memory. By contrast, MapEmbed uses smaller index (0.5∼4 BPI). RACE and TEA do not use index. To ensure a fair comparison, we compare the performance of all works under the same fast memory size. Following existing KV stores that prioritize fast memory as a cache [18], [27], we add an 8-way set-associative LRU cache to each candidate work. We ensure that the total size of fast memory is 65MB for all works by varying their cache sizes ($\frac{M_{fast}}{M_{slow}} \approx 0.017$). For example, CuckooDuo uses a 60MB CuckooIndex and 5MB cache. This setting of simultaneously using a cache and an index in fast memory has been widely used by recent works [24], [25]. As shown in Figure 11(a), as the ratio of illegal lookups (non-existent keys) increases, CuckooDuo's lookup RTT drops from 0.63 to nearly 0, while that of the other works increases from 0.61 to 1. This is because CuckooIndex can identify and filter out non-existent keys, whereas non-existent keys only incur more cache misses for the other works. Figure 11(b) shows CuckooDuo always has the smallest bandwidth thanks to the perfect hashing property of CuckooIndex. Figure 11(c) shows the lookup latency, which follows the same trend as the RTT results in Figure 11(a). Even when there are no illegal lookups, CuckooDuo still has slightly smaller latency because of smaller bandwidth. In summary, with the same fast memory size, CuckooDuo still achieves better lookup speed, especially when there are many illegal lookup requests. We will present more results on various hybrid workloads later. Although the cache cannot improve insert/delete speed, we can see that it indeed effectively reduces the lookup latency of CuckooDuo (from 3.3$\mu$s in Figure 10(b) to 2.2$\mu$s). Thus, it is a promising design for CuckooDuo to use both CuckooIndex and a small cache in fast memory like prior KV stores [24], [25].




(a) Number of RTT per lookup (b) # accessed items per lookup (bandwidth) (c) Lookup latency

Fig. 11: Performance under the same fast memory size.

**Performance under more workloads (Table III):** We conduct experiments on more workloads with various request modes and distributions. We use YCSB to create running




(a) Expansion time of CuckooDuo (active) (b) Expansion time of RACE (c) Expansion time of MapEmbed (active) (d) Expansion time of CuckooDuo (lazy) (e) Expansion time of MapEmbed (lazy) (f) Load factor of basic expansion vs. extendible expansion (g) Load factor of CuckooDuo vs. RACE (extendible-300)

Fig. 12: Dynamic expansion performance of CuckooDuo and comparison with existing expansion methods.

workloads with two request modes: YCSB-A (with 50% lookups and 50% updates), and YCSB-D (with 95% lookups and 5% inserts); and four request distributions: zipfian with $\theta = 0.99$ and $\theta = 0.90$, uniform, and latest. We provide the CDFs of these workloads in our supplementary materials [58]. The results show CuckooDuo achieves consistently smaller latency than others under various workloads. We also evaluate the latency under the same fast memory size (65MB) setting described above ($\frac{M_{fast}}{M_{slow}} \approx 0.017$), where CuckooDuo also achieves the smallest latency. The cache in fast memory reduces the latency of all works under skewed request distributions (zipfian, latest), with greater improvement as the skewness increases. We also evaluate the performance under hybrid workloads with different lookup/insert ratios, and the results are presented in our supplementary materials [58].

TABLE III: Latency ($\mu s$) under different request modes and distributions at 50% load factor ("CD" refers to "CuckooDuo", "ME" refers to "MapEmbed").

| Workload | | w/o cache | | | | w/ cache (fast/slow=0.017) | | | |
|---|---|---|---|---|---|---|---|---|---|
| Mode | Dist. | CD | ME | RACE | TEA | CD | ME | RACE | TEA |
| YCSB-A L50% U50% | zipf-0.99 | 4.91 | 5.04 | 6.39 | 5.27 | 4.38 | 4.52 | 5.61 | 4.66 |
| | zipf-0.90 | 4.95 | 5.09 | 6.41 | 5.27 | 4.78 | 4.93 | 6.12 | 5.06 |
| | uniform | 4.97 | 5.12 | 6.42 | 5.27 | 5.00 | 5.19 | 6.47 | 5.33 |
| | latest | 4.85 | 4.98 | 6.38 | 5.23 | 4.06 | 4.18 | 5.15 | 4.33 |
| YCSB-D L95% I5% | zipf-0.99 | 3.42 | 3.74 | 5.07 | 3.89 | 2.35 | 2.62 | 3.39 | 2.66 |
| | zipf-0.90 | 3.41 | 3.76 | 5.08 | 3.89 | 3.06 | 3.36 | 4.40 | 3.43 |
| | uniform | 3.46 | 3.77 | 5.09 | 3.92 | 3.50 | 3.89 | 5.14 | 3.98 |
| | latest | 3.35 | 3.66 | 5.03 | 3.88 | 1.84 | 2.09 | 2.68 | 2.11 |

### C. Performance of Dynamic Expansion

We evaluate the performance of CuckooDuo during expansion and compare it with MapEmbed [3] and RACE [9]. We organize the memory in remote server into many small chunks. Rather than pre-allocating enough memory for future insertion requests like RACE [9], we only allocate one small memory chunk in the initialization stage to build a small table. This is because in practice, we cannot predict the size of future workloads, and pre-allocating excessively large memory can lead to space waste. Every time an expansion is triggered, local server establishes a TCP connection to notify remote server of the expansion ratio. In response, remote server allocates an appropriate number of memory chunks based on the expansion ratio and performs corresponding expansion operations.

**Expansion time (Figure 12(a)-12(e)):** We break down the expansion time (with 2× expansion ratio) into the following components: the time spent on memory registration, memory copying (CuckooDuo and MapEmbed-lazy), cleaning redundant items (CuckooDuo), moving items (RACE and MapEmbed-active), and other overhead (including local computation time and communication time *etc.*). The active-mode expansion of CuckooDuo takes 4s to expand a 30M table (Figure 12(a)), which is about 2× faster than RACE (Figure 12(b)) and active-mode MapEmbed (Figure 12(c)). This is because active-mode expansion time is dominated by the time spent on accessing remote memory. CuckooDuo removes redundant items by modifying only local CuckooIndex, and thus only needs to READ remote memory. By contrast, RACE and MapEmbed need to modify remote memory to move items, and thus require sequential READ and WRITE to remote memory. The lazy-mode expansion of CuckooDuo takes only 0.6s to expand a 30M table (Figure 12(d)), which is 6.67× smaller than the active-mode. The expansion time of lazy-mode CuckooDuo is 1.18× smaller than lazy-mode MapEmbed (Figure 12(e)), because CuckooDuo only copies the table once, whereas MapEmbed requires two copies, thus needs 2× memory registration and copy time. Finally, we notice that the memory registration time is short (0.16s to register a 30M KV table), accounting for only 1/4 of CuckooDuo's lazy-mode expansion time and 4% of its active-mode expansion time.

**Space utilization (Figure 12(f)-12(g)):** We evaluate the space utilization of CuckooDuo (w/ and w/o extendible expansion) and RACE (w/ extendible expansion) during insertion process. In Figure 12(f), we compare CuckooDuo's basic expansion (with 2× expansion ratio) and extendible expansion (with 300 and 3000 sub-table sizes). The load factor fluctuates between 50% and 96%. For basic expansion, the load factor is halved after each expansion. By contrast, using extendible expansion with small sub-tables can smooth the load factor curve. In Figure 12(g), we compare the extendible expansion of CuckooDuo and RACE (with 300 sub-table size). During the insertion process, CuckooDuo has higher space utilization than RACE due to its higher single-table load factor (Figure 9(a)).

## VI. Conclusion

This paper presents an extendible RDMA-based remote memory KV store called CuckooDuo. The key design is to build a dynamic perfect hashing index called CuckooIndex in fast memory, and store the KVs in a cuckoo hash table called CuckooVault in slow memory, which has one-to-one mapping with CuckooIndex. We theoretically and empirically validate our solution achieves high load factor, fast speed, and small bandwidth. We hope our one-to-one mapping design and our dual-fingerprint adjustment idea could inspire future KV stores and and future fingerprint-based hashing indexes.

**Acknowledgment.** This work was supported by National Key R&D Program of China (No. 2024YFB2906602), National Natural Science Foundation of China (NSFC) (No. U20A20179, 623B2005, 624B2005, 62402012, 62372009), China Postdoctoral Science Foundation (No. 2023TQ0010, GZC20230055, 2024M750102), research grant No. SH-2024JK29, and High Performance Computing Platform of Peking University.

## REFERENCES


[1] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer Auf Der Heide, Hans Rohnert, and Robert E Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[2] Giulio Ermanno Pibiri and Roberto Trani. Pthash: Revisiting fch minimal perfect hashing. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 21)*, pages 1339–1348, 2021.

[3] Yuhan Wu, Zirui Liu, Xiang Yu, Jie Gui, Haochen Gan, Yuhao Han, Tao Li, Ori Rottenstreich, and Tong Yang. Mapembed: Perfect hashing with high load factor and fast update. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (SIGKDD 21)*, pages 1863–1872, 2021.

[4] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP 11)*, pages 1–13, 2011.

[5] Qi Chen, Hao Hu, Cai Deng, Dingbang Liu, Shiyi Li, Bo Tang, Ting Yao, and Wen Xia. Eeph: An efficient extendible perfect hashing for hybrid pmem-dram. In *2023 IEEE 39th International Conference on Data Engineering (ICDE 23)*, pages 1366–1378. IEEE, 2023.

[6] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, 2019.

[7] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[8] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies (CoNEXT 14)*, pages 75–88, 2014.

[9] Pengfei Zuo, Qihui Zhou, Jiazhao Sun, Liu Yang, Shuangwu Zhang, Yu Hua, James Cheng, Rongfeng He, and Huabing Yan. Race: one-sided rdma-conscious extendible hashing. *ACM Transactions on Storage (TOS)*, 18(2):1–29, 2022.

[10] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H Raymond Strong. Extendible hashing—a fast access method for dynamic files. *ACM Transactions on Database Systems (TODS)*, 4(3):315–344, 1979.

[11] Source codes of cuckooduo. https://github.com/CuckooDuo/CuckooDuo.

[12] Michael L Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0 (1) worst case access time. *Journal of the ACM (JACM)*, 31(3):538–544, 1984.

[13] Cmph - c minimal perfect hashing library. http://cmph.sourceforge.net/.

[14] Michael L Fredman and János Komlós. On the size of separating systems and families of perfect hash functions. *SIAM Journal on Algebraic Discrete Methods*, 5(1):61–68, 1984.

[15] Pengfei Zuo, Yu Hua, and Jie Wu. Level hashing: A high-performance and flexible-resizing persistent hashing index structure. *ACM Transactions on Storage (TOS)*, 15(2):1–30, 2019.

[16] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. Lock-free concurrent level hashing for persistent memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 799–812, 2020.

[17] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *Proceedings of the 2020 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 20)*, pages 90–106, 2020.

[18] Memcached. https://memcached.org/.

[19] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. Viper: An efficient hybrid pmem-dram key-value store. *Proceedings of the VLDB Endowment*, 14(9):1544–1556, 2021.

[20] Jihang Liu, Shimin Chen, and Lujun Wang. Lb+ trees: Optimizing persistent index performance on 3dxpoint memory. *Proceedings of the VLDB Endowment*, 13(7):1078–1090, 2020.

[21] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. Hikv: a hybrid index key-value store for dram-nvm memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, 2017.

[22] An Qin, Mengbai Xiao, Jin Ma, Dai Tan, Rubao Lee, and Xiaodong Zhang. Directload: A fast web-scale index system across large regional centers. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1790–1801. IEEE, 2019.

[23] Ruiyuan Li, Xiang He, Yingying Sun, Jun Jiang, You Shang, Guanyao Li, and Chao Chen. Spatio-temporal keyword query processing based on key-value stores. *Data Science and Engineering*, 2024.

[24] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed b+ tree index on disaggregated memory. In *Proceedings of the 2022 international conference on management of data*, pages 1033–1048, 2022.

[25] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[26] Jian-Zhong Li. opengauss: An open-source database for the era of artificial intelligence. *Journal of Computer Science and Technology*, 39(5):1005–1006, 2024.

[27] Redis. https://redis.io.

[28] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 249–264, 2016.

[29] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. Rolex: A scalable rdma-oriented learned key-value store for disaggregated memory systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 99–114, 2023.

[30] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostić, and Marco Chiesa. A high-speed stateful packet processing approach for tbps programmable switches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1237–1255, 2023.

[31] Danielle E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, 2016.

[32] Zi-Wei Xiong, De-Jun Jiang, Jin Xiong, and Ren Ren. Dalea: A persistent multi-level extendible hashing with improved tail performance. *Journal of Computer Science and Technology*, 38(5):1051–1073, 2023.

[33] Ying Wang, Wen-Qing Jia, De-Jun Jiang, and Jin Xiong. A survey of non-volatile main memory file systems. *Journal of Computer Science and Technology*, 38(2):348–372, 2023.

[34] Miao Cai, Junru Shen, Yifan Yuan, Zhihao Qu, and Baoliu Ye. Bonsaikv: Towards fast, scalable, and persistent key-value stores with tiered, heterogeneous memory system. *Proceedings of the VLDB Endowment*, 17(4):726–739, 2023.

[35] Mengbai Xiao, Hao Wang, Liang Geng, Rubao Lee, and Xiaodong Zhang. An rdma-enabled in-memory computing platform for r-tree on clusters. *ACM Transactions on Spatial Algorithms and Systems (TSAS)*, 8(2):1–26, 2022.

[36] Mengbai Xiao, Hao Wang, Liang Geng, Rubao Lee, and Xiaodong Zhang. Catfish: Adaptive rdma-enabled r-tree for low latency and high throughput. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 164–175. IEEE, 2019.

[37] Liang Geng, Hao Wang, Jingsong Meng, Dayi Fan, Sami Ben-Romdhane, Hari Kadayam Pichumani, Vinay Phegade, and Xiaodong Zhang. Rr-compound: Rdma-fused grpc for low latency and high throughput with an easy interface. *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[38] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems*, 38(2):229–248, 2005.

[39] Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science*, 380(1-2):47–68, 2007.

[40] Yuanyuan Sun, Yu Hua, Song Jiang, Qiuyu Li, Shunde Cao, and Pengfei Zuo. Smartcuckoo: A fast and cost-efficient hashing index scheme for cloud storage systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 553–565, 2017.

[41] Qiuyu Li, Yu Hua, Wenbo He, Dan Feng, Zhenhua Nie, and Yuanyuan Sun. Necklace: An efficient cuckoo hashing scheme for cloud storage services. In *2014 IEEE 22nd International Symposium of Quality of Service (IWQoS 14)*, pages 153–158. IEEE, 2014.

[42] Yuanyuan Sun, Yu Hua, Dan Feng, Ling Yang, Pengfei Zuo, and Shunde Cao. Mincounter: An efficient cuckoo hashing scheme for cloud storage systems. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST 15)*, pages 1–7. IEEE, 2015.

[43] Alan Frieze, Páll Melsted, and Michael Mitzenmacher. An analysis of random-walk cuckoo hashing. *SIAM Journal on Computing*, 40(2):291–308, 2011.

[44] Nikolaos Fountoulakis, Konstantinos Panagiotou, and Angelika Steger. On the insertion time of cuckoo hashing. *SIAM Journal on Computing*, 42(6):2156–2181, 2013.

[45] Alan Frieze and Tony Johansson. On the insertion time of random walk cuckoo hashing. *Random Structures & Algorithms*, 54(4):721–729, 2019.

[46] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys 14)*, pages 1–14, 2014.

[47] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, 2013.

[48] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proceedings of the 9th ACM conference on Emerging networking experiments and technologies*, pages 97–108, 2013.

[49] Salvatore Pontarelli, Pedro Reviriego, and Michael Mitzenmacher. Emoma: Exact match in one memory access. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 30(11):2120–2133, 2018.

[50] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment*, 8(11):1226–1237, 2015.

[51] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. The yin and yang of processing data warehousing queries on gpu devices. *Proceedings of the VLDB Endowment*, 6(10):817–828, 2013.

[52] Kai Zhang, Feng Chen, Xiaoning Ding, Yin Huai, Rubao Lee, Tian Luo, Kaibo Wang, Yuan Yuan, and Xiaodong Zhang. Hetero-db: next generation high-performance database systems by best utilizing heterogeneous computing and storage resources. *Journal of Computer Science and Technology*, 30(4):657–678, 2015.

[53] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Li, Xiaodong Zhang, Bingsheng He, Jiayu Hu, and Bei Hua. A distributed in-memory key-value store system on heterogeneous cpu–gpu cluster. *The VLDB Journal*, 26:729–750, 2017.

[54] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[55] Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. Adaptive cuckoo filters. *ACM Journal of Experimental Algorithmics*, 2020.

[56] Minmei Wang and Mingxun Zhou. Vacuum filters: more space-efficient and faster replacement for bloom and cuckoo filters. *Proceedings of the VLDB Endowment (VLDB 19)*, 2019.

[57] Niv Dayan, Ioana Bercea, Pedro Reviriego, and Rasmus Pagh. Infinifilter: Expanding filters to infinity and beyond. *Proceedings of the ACM on Management of Data (SIGMOD 23)*, 1(2):1–27, 2023.

[58] Supplementary materials of cuckooduo. https://github.com/CuckooDuo/CuckooDuo/blob/main/CuckooDuo_Supplementary.pdf.

[59] Dagang Li, Rong Du, Ziheng Liu, Tong Yang, and Bin Cui. Multi-copy cuckoo hashing. In *2019 IEEE 35th International Conference on Data Engineering (ICDE 19)*, pages 1226–1237. IEEE, 2019.

[60] Martin Aumüller, Martin Dietzfelbinger, and Philipp Woelfel. Explicit and efficient hash families suffice for cuckoo hashing with a stash. *Algorithmica*, 70(3):428–456, 2014.

[61] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *24th USENIX Security Symposium*, pages 515–530, 2015.

[62] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms (SODA 12)*, pages 157–167. SIAM, 2012.

[63] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, 39(4):1543–1561, 2010.

[64] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable hashing on persistent memory. *arXiv preprint arXiv:2003.07302*, 2020.

[65] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. Exact and approximate membership testers. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 59–65, 1978.

[66] Fabiano C Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and space-efficient minimal perfect hash functions. In *Algorithms and Data Structures: 10th International Workshop, WADS 2007, Halifax, Canada, August 15-17, 2007. Proceedings 10*, pages 139–150. Springer, 2007.

[67] Edward A Fox, Lenwood S Heath, Qi Fan Chen, and Amjad M Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35(1):105–121, 1992.

[68] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[69] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. Nomad: Non-exclusive memory tiering via transactional page migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 19–35, 2024.

[70] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[71] Intel instructions. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[72] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance rdma systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, 2016.